SYMBOLIC TECHNIQUES FOR PARAMETERISED VERIFICATION

Chih-Duo Hong

St Hugh's College, Oxford

Thesis submitted for the degree of Doctor of Philosophy

University of Oxford

Trinity 2022

SYMBOLIC TECHNIQUES FOR PARAMETERISED VERIFICATION

Chih-Duo Hong
St Hugh's College, Oxford

Abstract

Parameterised systems are infinite-state systems comprising a parameterised number of components. The problem of verifying parameterised systems is both important practically and challenging theoretically: it is important because a wide range of real-world computer systems are parameterised in nature, and it is challenging because most relevant properties of parameterised systems are undecidable in general.

In this thesis, we study automated verification approaches using automata-based symbolic model checking (a.k.a. regular model checking) as a generic formalism for parameterised systems. Although this formalism is Turing powerful (i.e. it can simulate the computation of Turing machines), the nice closure and algorithmic properties of finite automata still allow us to design effective heuristics for practical applications. Our main proposal for regular model checking is to use first-order theories over regular relations to reason about correctness properties, and use regular language inference as a means of proof generation. We further develop powerful reduction and abstraction techniques to reduce difficult verification problems to simpler tasks for several classes of parameterised systems. These techniques include reductions from liveness properties to safety properties for regular systems, reductions from infinite data domains to finite alphabets for array systems, and reductions from quantitative reasoning to qualitative reasoning for probabilistic systems. All these reductions are performed in a uniform logical way based on decidable first-order theories.

We have implemented and evaluated our automated approaches against a nontrivial collection of examples covering concurrent systems, distributed algorithms, and cryptographic protocols. The promising experimental results confirm our hypothesis that regular languages and relations provide effective heuristics for parameterised systems verification.

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my supervisors, Luke Ong and Anthony Lin, for all of their guidance, assistance, and encouragement. I would like to thank Anthony in particular for offering me the opportunity to study in Oxford. As my main supervisor, Anthony is full of sound advice and enlightening insights, and is always eager to help when I encounter academic or personal problems. I cannot hope for a more supportive supervisor than him.

I would like to thank my coauthors and colleagues, with whom I have had many inspiring discussions and fruitful collaborations before and during my DPhil studies: Yu-Fang Chen, Bow-Yaw Wang, Parosh Aziz Abdulla, Lorenzo Clemente, Lukáš Holík, Ondřej Lengál, Richard Mayr, Tomáš Vojnar, Rupak Majumdar, Philipp Rümmer, Lijun Zhang, Frédéric Haziza, Ahmed Rezine, Matthew Hague, Shin-Cheng Mu, Nishant Sinha, Xuan-Bach Le, Reino Niskanen, Pablo Barceló, Daniel Stan, Oliver Markgraf, Muhammad Najib, and Daniel Neider. I have learnt a lot from them regarding many aspects of research, careers, and life. My special thanks go to Daniel Stan and Wei-Lin Wu, who read an early version of my thesis and gave me valuable comments.

I would like to thank Yu-Fang Chen for introducing me to the amazing area of model checking. My research journey in this area started from his decision to hire and train me at the time when I knew virtually nothing about model checking. I am also grateful to have Wei-Lin Wu as my best friend in both academic and personal life. His enthusiasm about mathematical logic and his courage to chase the dream have been a great source of inspiration for me to proceed in academia.

I would like to thank the Department for their studentship, and thank Google for funding my DPhil studies.

Last but not least, I am deeply indebted to my parents for their incredible love, patience, and understanding. My DPhil studies would not be possible without their support. This thesis is dedicated to them.

Declaration

I declare that the work presented in this thesis is my own except where explicitly stated otherwise in the text, and that no portion of this work has been submitted for any degree or professional qualification except as specified.

Contents

1 Introduction **1**

1.1 Infinite-state model checking 4

1.2 Parameterised model checking 7

1.3 Contributions 10

2 Preliminaries **14**

2.1 Automata and languages 14

2.1.1 Sets, relations, and functions 14

2.1.2 Finite automata and regular languages 16

2.2 Logic and structures 19

2.2.1 Structures 19

2.2.2 FO : first-order logic 22

2.2.3 $\mathrm{FO}_{\mathrm{REG}}$: a first-order framework for regular relations 30

2.2.4 MSO : monadic second-order logic 35

2.3 Systems and properties 38

2.3.1 Kripke structures 38

2.3.2 Safety and liveness 39

2.3.3 Probabilistic systems 43

2.3.4 Bisimulation equivalence 44

3 Regular Model Checking **46**

3.1 Techniques for safety verification 48

3.2 Techniques for liveness verification 54
3.3 Extensions and generalisations 56

4 Safety and Liveness Verification of Regular Transition Systems 58
4.1 Related work 61
4.2 Safety verification 62
4.3 Liveness verification 65
4.3.1 Regular encoding of fairness requirements 68
4.3.2 Verification under fairness requirements 71
4.4 L-star automata learning 73
4.5 Learning inductive invariant 76
4.6 Experiments and evaluation 79

5 Safety and Liveness Verification of Array Systems 89
5.1 Related work 91
5.2 Specification for array systems 95
5.3 Regular languages as abstractions 99
5.3.1 Predicate abstraction with indexed predicates 99
5.3.2 Approximations with regular languages 103
5.4 Computation of regular abstractions 106
5.4.1 Abstractions for state formulae 107
5.4.2 Abstractions for transition formulae 113
5.5 Verification of temporal array properties 116
5.5.1 Specific method for safety and liveness properties 117
5.5.2 Liveness verification under fairness requirements 119
5.5.3 Generic method for monodic temporal properties 121
5.6 Case studies 125
5.6.1 Selection Sort 125
5.6.2 Dijkstra's self-stabilising algorithm 128
5.6.3 Chang-Robert's algorithm 133

5.7 Experiments and evaluation 138

6 Equivalence Verification of Regular Probabilistic Systems 141

6.1 Related work 143

6.2 Probabilistic bisimulation in regular relations 144

6.2.1 The non-probabilistic case 145

6.2.2 Specifying a probabilistic transition system 145

6.2.3 Proof rules for probabilistic bisimulation 149

6.3 Application to anonymity verification 153

6.3.1 Preliminaries 153

6.3.2 Case studies 156

6.4 Learning probabilistic bisimulation 159

6.5 Experiments and evaluation 162

7 Conclusion and Future Work 164

Bibliography 167

Appendices 202

A.1 More liveness-to-safety reduction examples 203

A.2 Word representation of regular abstraction 204

A.3 A modelling language for array systems 205

A.4 Modelling the generalised dining cryptographers 209

A.5 Modelling the grades protocol 211

Chapter 1

Introduction

Computer systems have become ubiquitous in our daily lives, with many systems being safety critical in the sense that they are responsible for complex and sensitive tasks in crucial areas, such as health care, traffic control, stock market, and military defence. Even a small fault in such systems could jeopardise human lives or have catastrophic financial consequences. The most well-known disaster caused by software failures is probably the Ariane-5 rocket's first launch, which ended in an explosion due to a floating-point conversion error in the pilot system. This single bug caused approximately $370 million loss for the European Space Agency and delayed related scientific research plans for almost 4 years. The Knight Capital Group, once the largest trader in U.S. equities, lost $460 million because a flag intended for debug purpose was mistakenly enabled by the trading software in production. Car companies including Toyota, General Motors, and Mercedes-Benz have spent billions of dollars recalling cars to address various software glitches. Even in areas where failures of computer systems are less crucial, accumulative impacts caused by these failures can also be tremendous. Indeed, a US Government report estimated that malfunctioned software cost the US economy approximately $60 billion each year [Pla02]. Therefore, it is critical to ensure, ideally in the design phase, that a computer system will behave as expected under all circumstances. This problem of *design verification* — ensuring design correctness at as early stage as possible — is a major challenge for modern system

development, and has attracted considerable industrial and academic research efforts.

The prevailing approaches towards design verification are testing, simulation, and static analysis. *Testing* comprises running a system under specific conditions, called test cases, and checking whether the outcomes of a given input matches expected outputs. Test cases are carefully designed to cover a maximal number of system executions in presumed production scenarios. *Simulation* is similar to testing, but performed on an abstract model of the system by removing parts irrelevant to the testing purpose. The use of models allows a system to be analysed and tested prior to the build phase of the development process. While testing and simulation are convenient to detect errors, practically they can only explore subsets of all possible execution traces, leaving open the question of whether unexplored traces may contain a fatal bug. *Static analysis* is an automatic technique that examines system correctness by directly inspecting the design. In principle, static analysis can achieve full coverage of system behaviour without executing or simulating the system. However, practical analysis methods often focus on precision and efficiency rather than expressiveness, aiming to locate obvious bugs precisely and quickly while leaving more subtle errors undetected. As a consequence, the correctness assurance one can obtain from static analysis is often quite limited.

In the early 1980s, Clarke and Emerson [CE81] and Queille and Sifakis [QS82] separately introduced the *model checking* approach to address design verification. Model checking is a (often fully automated) process to determine whether or not the proposed model of a computer system satisfies a property, where the model is a mathematically precise and unambiguous characterisation of possible system behaviour, and the property is a formal description of the expected behaviour, commonly expressed as formulae in some temporal logic. Given a model and a property as inputs, a model checker either confirms that the property holds for the model, or reports that the property is violated and provides a *counterexample* demonstrating behaviour that leads to the violation. This framework is quite general and has been applied to various kinds of models and logics with effective model checking algorithms.

Compared with the traditional methods, model checking operates on abstract

models, similar to simulation, but provides correctness assurance with a superior degree of confidence. Model checking is often fully automatic, similar to static analysis, but can verify complex behavioural properties that are outside the capacities of static analysis methods. However, a verdict returned by a model checker may arise from the simplification applied to the model rather than from real system behaviour since the approach analyses a model rather than the actual system. Complementary techniques such as code review and testing might still be essential to establish or refute properties of interest when applying model checking.

Many model checking techniques have been proposed to reason about properties of systems including digital circuits, hardware designs, and communication protocols. While most model checking techniques can be performed in a push-button manner, these techniques often resort to exhaustive searches of state space, whose size can grow exponentially as model complexity increases. This rapid growth of state space, known as the *state explosion* phenomenon, has become a limiting factor for the applicability of model checking. Indeed, industrial systems often yield models of scale several orders of magnitude larger than those investigated in academia, and a large body of research regarding model checking has been devoted to address scalability issues [CGJ+01; GV08; CKNZ11].

A remarkable milestone in the evolution of model checking was the introduction of symbolic techniques. Most approaches before 1990 used explicit representations, where states were stored individually in memory. Burch et al. [BCM+92; BCL+94] and McMillan [McM93] showed that it was possible to encode the state space with a concise symbolic representation, where states are encoded as vectors of Boolean variables, and sets and relations are expressed with propositional formulae. These formulae were further represented by a canonical data structure called *reduced ordered binary decision diagram* (BDD) [Bry86]. Burch et al. and McMillan provided several efficient algorithms to verify properties expressed in mu-calculus based on manipulating BDDs. The space complexity of model checking has been greatly mitigated thanks to the development of symbolic techniques, allowing many huge industrial systems to be formally verified for

the first time.

The major drawback of symbolic techniques using BDDs is that the amount of memory required to store these structures can be exponential in the number of variables. Biere et al. [BCCZ99; BCC+03] proposed an alternative approach, *bounded model checking* (BMC), which sacrificed completeness of verification for efficiency of bug detection. The concept of BMC was to look for counterexamples in program executions up to a certain number of steps, which increased incrementally until either a bug was found or the problem became intractable. The subsequent *k-induction* technique [SSS00; DRS03] extended BMC with the power to prove invariance properties. The technique was based on an induction scheme: an inductive hypothesis holds universally if (i) it holds in the first k steps of any execution run, and (ii) its truth in k consecutive steps implies its truth in the $(k+1)$-th step. BMC and k-induction can both be encoded as propositional formulae whose satisfiability leads directly to a counterexample. The strength of this approach comes mainly from the progress of *Boolean satisfiability* (SAT) solving technology. Modern SAT solvers can solve propositional formulae with hundreds, thousands, or even millions of variables in reasonable time. In practice, SAT-based methods can often handle problem instances orders of magnitude larger than those handled by BDD-based counterparts. Since a circuit can be concisely expressed as propositional constraints, SAT-based model checking has been especially successful in verifying the implementation of hardware circuits [MS00; Mar08]. Nowadays, SAT-based verification has become a standard part of industrial circuit design process, where highly optimised tools make it possible to automatically discover subtle errors, e.g., due to the presence of overflows.

1.1 Infinite-state model checking

Model checking was originally developed to verify finite-state systems. Indeed, most real-world computer systems can be considered as finite systems since a physical computer has a finite amount of memory and disk space. Nevertheless, it is often

more intuitive to model a system as infinite-state, e.g., by permitting the use of infinite data types, such as integers and strings, and unbounded data structures, such as arrays and queues. This is perhaps also the reason why infinite-state computation models like Turing machines are used as formal definitions for algorithms. Most interesting properties of infinite-state systems cannot be determined algorithmically due to limitations from computability theory (cf. [HMU07; Koz12]). Thus, research efforts in infinite-state verification have roughly clustered along two directions: one is to identify decidable classes and design efficient decision procedures for them, and the other is to develop effective semi-algorithms for general undecidable problems. These two directions are not orthogonal, however, since a decision procedure easily becomes a semi-algorithm by removing some assumptions on the input, and a popular semi-algorithm often attracts research interest to characterise sufficient conditions for its completeness.

The diversity of problem domains for infinite-state verification has given birth to many interesting model checking techniques, which can be broadly summarised into two approaches: representing the system symbolically, and constructing an abstraction for the system. In the first approach, logical formulae are employed as symbolic representations, and techniques previously using SAT solvers, such as bounded model checking and k-induction, are migrated to *satisfiability modulo theories* (SMT) solvers to deal with infinite domains. An SMT solver extends an SAT solver by replacing Boolean variables with atomic propositions interpreted in some background theory. The power of SMT solvers comes from the large number of theories they support, including the theory of equality and uninterpreted functions, linear integer arithmetic, real arithmetic, array theory, bit-vector theory, etc. A representative tool for this approach is Kind2 [CMST16].

The second approach, constructing an abstraction for the system, simplifies the verification problem by sacrificing the precision of analysis. Most models considered in this approach arise from the framework of *abstract interpretation* [CC77; CC92], where the target system is interpreted on an abstract domain to obtain a model amenable to

finite-state model checking. For abstract interpretation to work in practice, it is crucial to select a suitable interpretation for the problem at hand. Thus, abstraction techniques are mostly heuristics whose usefulness has to be demonstrated by practical problem instances. A popular technique is *predicate abstraction* [GS97], which typically uses a finite number of predicates to describe plausible properties for the given system. The abstraction function then associates each configuration of the system with a subset of the predicates. By identifying configurations that satisfy the same predicates, finite abstract models emerge as an over-approximation for the original system. Classical finitary methods can then be applied to analyse the model and infer the strongest possible invariant expressible using the predicates, which hopefully is sufficiently strong to prove the desired property. Well-known verification tools exploiting predicate abstractions include SLAM [BMMR01; BR02], SatAbs [CKSY05], BLAST [BHJM07], and CPAchecker [BK11].

Generally, coarser abstractions tend to produce smaller and simpler abstract models that are easier to analyse. When the abstraction is too coarse, however, the abstract model may contain counterexamples that do not exist in the original system. Thus, it is desirable to have an automated procedure to refine the abstract domain and eliminate false counterexamples. This concept has developed into an iterative strategy called *counterexample guided abstraction refinement* (CEGAR) [CGJ+00; Sai00; BMMR01]. This strategy has proved to be quite successful, and various improvements have been subsequently proposed. For example, Henzinger et al. [HJMS02; HJMS03] computed necessary predicates on demand for *lazy abstraction*, allowing different parts of the abstract model to have different degrees of precision. McMillan [McM06] furthermore combined lazy abstraction with refinement by *Craig interpolation* [Cra57] to avoid deducing irrelevant information that might complicate the analysis or even lead to divergence.

A system can be infinite not because it has variables over infinite data domains, but because it is composed by an unbounded number of components. For example, communication protocols are often intended to work regardless of the number of

agents participating in the protocol. Therefore, the system of a protocol in its entirety comprises finite systems resulting from every possible number of agents. This kind of systems are known as parameterised systems, and are more relevant to the verification problems we shall be studying in this thesis.

1.2 Parameterised model checking

Many infinite-state systems can be naturally represented as an infinite family of finite structures indexed by parameters. These parameters often quantify certain unbounded resources, such as number of processes in a distributed system, size of message buffers in a communication network, or number of caches in a cache coherence protocol. Properties of parameterised systems have commonly been studied in settings where parameter values can be very large or unknown in advance. Particularly, a parameterised system is considered correct only if it works properly with all possible parameter values.

Take the well-known dining philosophers protocol [LR81] as an example. The correctness of this protocol consists in two properties: (i) no neighbouring philosophers can eat at the same time, and (ii) no philosopher will be prevented from eating forever. The former is often called a safety property, while the latter is called a liveness property. For any given number of philosophers, checking these properties is decidable using finite model checking techniques. To verify the protocol for all possible numbers of philosophers, however, the number of configurations a verification algorithm has to explore is potentially infinite. Indeed, safety verification is already undecidable for parameterised systems as simple as token-passing rings [Suz88; EN95]. Despite these difficulties, parameterised systems have continued to attract interest from the formal verification community. Unfortunately, no decidable formalism has been shown sufficiently expressive to cover a reasonably large class of interesting problems, despite considerable research efforts to identify decidable classes for parameterised verification (see [BJK+15] for a survey),

Although parameterised systems can be handled by generic infinite-state verification methods, it is often favourable to develop specialised techniques that take advantage of their parameterised structure (e.g. as an infinite family of finite systems). *Cutoff* is a representative technique of this type, which seeks to discover an upper bound on the parameter based on both the system and property. This bound, i.e., the value of cutoff, is a finite quantity such that a property holding for parameters up to this bound is guaranteed to hold after the bound is removed. If a cutoff exists for a parameterised verification problem, solving the problem amounts to checking a finite number of system instances, which are then amenable to finite-state verification methods. The cutoff approach has been integrated with several formalisms, leading to decidability results for various classes of parameterised systems [GS92; EN95; EK00; EK03a; EK03b]. Sometimes, a cutoff can be statically calculated from the specification, as is the approach of *invisible invariants* [APR+01; PRZ01]. However, even when a cutoff is computable, the value can often be prohibitive for full-sized parameterised systems. Several techniques have been developed to discover cutoff values dynamically and hence mitigate this issue [KKW10; AHH13].

Similar to the generic infinite-state cases, the major challenge of parameterised model checking amounts to finding an appropriate symbolic representation for infinite sets and relations. *Regular model checking* [KMM+97; WB98; BJNT00; JN00] emerged around 2000 as a symbolic framework for parameterised verification, and has subsequently been developed with the aim to extend the success of symbolic model checking to parameterised systems. The main concepts underlying regular model checking are relatively simple: (i) strings are an adequate data type to encode configurations of parameterised systems, and (ii) regular languages are adequate symbolic representations for infinite sets of strings. In fact, regular languages can be employed to represent not only systems and specifications, but also correctness proofs for desired properties. Examples of such proofs include inductive invariants for safety, ranking functions for liveness, and bisimulation relations for observational equivalence. Highly optimised semi-algorithms have been developed for regular model checking to verify various

correctness properties in diverse problem domains. This thesis aims to extend this line of research for parameterised verification.

A proof-based verification scheme

Since regular languages are enumerable through their automata presentations, we can exhaustively search for a regular proof of a correctness property provided that the property has decidable *verification conditions*. If counterexamples violating the property are also enumerable, it is possible to solve the verification problem by designing two semi-procedures, one searching for a (regular) proof and the other searching for a counterexample. This technique is quite general and has been applied to various classes of systems [Sif82]. For example, to prove that a transition system $\langle S, T, I \rangle$ is safe with respect to a set $B \subseteq S$ of bad states, it suffices to provide a proof $P \subseteq S$ satisfying the following verification conditions: (i) $I \subseteq P$, (ii) $T(P) \subseteq P$, and (iii) $P \cap B = \emptyset$. These conditions are decidable in any symbolic computation framework that is "regular" in the sense that the encoding of sets is effectively closed under Boolean operations, and the post-image $T(X)$ or pre-image $T^{-1}(X)$ of a set X is computable from the encoding of X. Many symbolic representations satisfy this regularity requirement aside from word-regular languages, including tree-regular languages [AJNd02; AHD+08], omega-regular languages [BLW05; LW10], symbolic regular languages [VHL+12; DV17], and visibly pushdown languages [AM04; AM09; DA14]. These representations are more expressive than word-regular languages and can be incorporated with regular model checking techniques to reason about more complex systems and properties.

In the word-regular setting, many automated techniques have been leveraged to search for safety proofs, including automata learning [VSVA04b; HV05; Nei14; CHLR17], SAT-based enumeration [NJ13; LNRS15; LR16], and abstraction refinement [BHRV06; Voj07]. The main advantage of automata learning techniques is their elegance, as well as the complexity and completeness results from algorithmic learning theory. However, since learning-based methods use membership queries for sampling, they often have difficulties handling systems without decidable point-to-point reachability. SAT-based

enumeration encodes proofs and verification conditions as Boolean constraints, and exploits an SAT solver to explore the proof space. Although SAT-based methods generally work for a broader class of systems, they are also more prone to scalability issues, and hence require tailored optimisations to deal with larger problem instances. Abstraction refinement methods tend to scale better in practice comparing with other techniques, but it can be difficult to establish complexity results and termination guarantees when these methods are applied with aggressive heuristics. While all of these techniques have found successful use cases in the literature, none of them offers dominantly better performance for all applications. One must carefully consider each method's advantages and disadvantages when attempting to apply them to a specific problem domain.

1.3 Contributions

This thesis develops automatic techniques to model check parameterised systems based on regular model checking. Correctness properties considered in this thesis include safety, liveness, and anonymity. Safety and liveness are fundamental correctness properties of computer systems, whereas anonymity plays an important role in privacy-preserving computation. We successfully apply our techniques to reason about these properties for various concurrent systems, distributed algorithms, and cryptographic protocols, some of which have not been verified by fully automated techniques previously.

Most verification problems we are dealing with have mature model checkers in the finite setting, i.e., when the parameters are fixed. For example, anonymity of cryptographic protocols can be checked by the finite-state model checker PRISM [HKNP06; PRI] and APEX [KMO+11; KMO+12]. However, the required runtime often grows rapidly with the number of agents, and can easily become unacceptable when the number of agents reaches a few hundreds [LMOW08]. By developing dedicated symbolic techniques, we show that it is possible to prove an anonymity property for *all* possible

numbers of agents with a far shorter runtime than required by finite model checkers for a fixed number of agents.

The main contributions of this thesis are summarised as follows.

– In Chapter 2, we introduce some background knowledge about automata, logic, transition systems, correctness properties, etc. Particularly, we show how finite-state automata and first-order theories can be elegantly connected through the notion of automatic structure and first-order interpretation. In Chapter 3, we briefly review the development of safety and liveness verification techniques in regular model checking. The material in these two chapters provides a foundation for topics discussed in subsequent chapters.

– In Chapter 4, we study the proof generation problem for safety and liveness verification of regular transition systems. We begin with a formal reduction that translates a liveness property to a safety property, which can be proved by pinpointing an appropriate inductive invariant. We then elaborate on a semi-algorithm to synthesise regular inductive invariants based on Angluin's L^* automata learning algorithm and its variants. We evaluate our learning-based proof generation method by verifying safety and liveness properties of representative benchmarks in regular model checking. Experimental results show that our method offers competitive, if not better, performance as compared with existing sophisticated verification tools.

– In Chapter 5, we turn our attention to *array systems*, a broader class of parameterised systems than regular transition systems. Array systems can model sequential array programs as well as concurrent systems with infinite-state processes, which are not directly amenable to regular model checking techniques. Nonetheless, array systems are similar to regular transition systems in concept: they both model a system configuration as a finite word, or equivalently an array of values from some background theory, and they both represent sets of states and transition relations as constraints in this theory. In this chapter, we extend the expressive power of regular model checking to array systems by "finitising" the infinite data

domains. We propose a symbolic framework to reason about quantified temporal properties of array systems based on a logic combining LTL and array theory. We subsequently provide systematic abstraction methods using indexed predicates, which yield abstract systems that can be analysed in the classic setting of regular model checking. We demonstrate that our numerical abstraction approach is sufficiently powerful to verify safety and liveness for distributed algorithms such as Dijkstra's self-stabilizing protocol and Chang-Robert's leader election protocol, whose correctness proofs are difficult to formalise using decidable formalisms.

– In Chapter 6, we study the anonymity verification problem for regular probabilistic systems. Anonymity is a property of communication protocols asserting that an observer cannot obtain information about the identities of other participating agents by observing actions these agents execute. Anonymity is essential for cryptographic and multi-party computation protocols, and can be established by providing an appropriate bisimulation relation. Our approach is in the spirit of deductive verification, where a decidable first-order theory is employed to formulate verification conditions and proofs. A candidate proof of anonymity is specified as a formula in this theory, and formally checked against verification conditions with a model checker. By restricting to regular systems and proofs, we can leverage classic automata learning algorithms to automate the search of candidate proofs thanks to the correspondence between finite automata and first-order logic. We show that our regular framework is sufficiently expressive to prove anonymity for the parameterised version of several cryptographic protocols, including the dining cryptographers protocol and the grades protocol, which could not be previously verified by automatic methods. We also show that candidate bisimulation relations can be synthesised with standard automata learning algorithms, making the verification process fully automated.

This thesis does not focus on theoretical results, such as decidability of a certain problem or complexity of a specific algorithm. Rather, this thesis concentrates on finding effective semi-procedures and heuristics for undecidable problems on interesting

benchmarks and case studies. The quality of our techniques is mostly evaluated via experiments and empirical comparisons with other tools. In subsequent chapters, we shall present various techniques to enable practical design verification for parameterised systems, which is the main challenge we aim to tackle in this thesis.

Previous publications

The invariant generation method in Chapter 4 was published in the paper *Learning to Prove Safety over Parameterised Concurrent Systems* [CHLR17], where I wrote the manuscript jointly, implemented the prototype, and conducted the experiments. Chapter 6 is based on the paper *Probabilistic Bisimulation for Parameterised Systems* [HLMR19], where I proved the theorems, wrote most of the manuscript, developed the tool jointly, and conducted the experiments. I am the sole author of the rest chapters in this thesis.

Chapter 2

Preliminaries

This chapter introduces definitions and notions we shall use throughout this thesis. In Section 2.1, we fix some general mathematical notations and recap fundamentals of formal language theory including words, languages, finite automata and transducers, and the concept of representing relations as languages. In Section 2.2, we discuss structures, logics, theories, and the important notions of regular presentations and logical interpretations. We also sketch several important results that will play a role in subsequent development. Most logical structures we will encounter in this thesis can be formalised as transition systems, and most correctness properties we shall be dealing with can be described in some modal logic. We briefly review these transition systems and modal logics in Section 2.3.

2.1 Automata and languages

2.1.1 Sets, relations, and functions

Given two sets A and B, we use $A \ominus B$ to denote their symmetric difference $(A \setminus B) \cup (B \setminus A)$, and use $A \times B$ to denote their *Cartesian product* $\{(a,b) : a \in A,\ b \in B\}$. We may write $A \uplus B$ instead of $A \cup B$ when A and B are disjoint. An *n-ary relation* is a set of n-tuples; a unary relation is also called a *predicate*. The Cartesian product

of an n-ary relation R and an m-ary relation T is identified with an $(n+m)$-ary relation $R \times T := \{(a_1, \ldots, a_{n+m}) : (a_1, \ldots, a_n) \in R,\ (a_{n+1}, \ldots, a_{n+m}) \in T\}$. The *composition* of two *binary* relations $R \subseteq A \times B$ and $T \subseteq B \times C$ is a binary relation $R \circ T := \{(a,c) : \text{there exist } b \text{ s.t. } (a,b) \in R \text{ and } (b,c) \in T\}$.

Fix a binary relation $R \subseteq A \times B$. We define the *domain* and the *range* of R as $Dom(R) := \{a \in A : \text{there is } b \in B \text{ such that } (a,b) \in R\}$ and $Img(R) := \{b \in B : \text{there is } a \in A \text{ such that } (a,b) \in R\}$, respectively. Given $S \subseteq A$, we define $R(S) := \{b \in B : \text{there is } a \in A \text{ such that } (a,b) \in R\}$ as the *image* of S under R. When $R \subseteq B \times B$, we define relations $R^n \subseteq B \times B$ for $n \in \mathbb{N}$ by induction on n: $R^0 := \{(b,b) : b \in B\}$, and $R^k := R \circ R^{k-1}$ for $n \geq 1$. The relation $R^* := \bigcup_{i=0}^{\infty} R^i$ is called the *reflexive and transitive closure* of R. Depending on the context, we may use $a\,R\,b$ or $R(a,b)$ as an alternative notation for $(a,b) \in R$.

A binary relation $R \subseteq S \times S$ is an *equivalence relation* if R satisfies (i) reflexivity: $(a,a) \in R$ for all $a \in Dom(R)$; (ii) symmetry: $(a,b) \in R$ if $(b,a) \in R$; (iii) transitivity: $(a,c) \in R$ whenever $(a,b) \in R$ and $(b,c) \in R$. Given an equivalence relation R over set S, the *equivalence classes induced by* R are the collection of subsets of S such that two elements $a, b \in S$ are contained in the same subset if and only if $(a,b) \in R$. These equivalence classes form a partitioning of S: the equivalence classes are disjoint, and the union of the classes coincides with S. We use S/R to denote the set of equivalence classes induced by R.

For $n \geq 1$, an n-ary function f is a mapping $X \to Y$ from a set of n-tuples X, called the *domain* of f and written $Dom(f)$, to a set Y, called the *range* of f and written $Img(f)$. Given an n-ary function f, the *graph* of f is the $(n+1)$-ary relation $\{(x_1, ..., x_n, f(x_1, ..., x_n)) : (x_1, ..., x_n) \in Dom(f)\}$. We shall often use $\bar{e}$ as an abbreviation for a list of elements $e_1, \ldots, e_n$ when the number n of the elements is inessential or clear from the context. Using this notation, for example, the graph of a function f may be written as $\{(\bar{x}, f(\bar{x})) : (\bar{x}) \in Dom(f)\}$.

2.1.2 Finite automata and regular languages

An *alphabet* Σ is a finite set of symbols. A *word* over Σ is a finite sequence of symbols in Σ. An empty word is denoted by ε, and we always assume that $\varepsilon \notin \Sigma$. Given a word $w := a_1 \cdots a_n$ over Σ, we define $|w| := n$ as the *size* of w, and let $w[i] := a_i$ for $i \in \{1, \ldots, n\}$. A set of words is also called a *language*.

A *finite automaton* is a tuple $\mathcal{A} := (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A *run* of $\mathcal{A}$ on a word $a_1 \cdots a_n$ is a sequence of states $q_0, \ldots, q_n$ such that $(q_i, a_{i+1}, q_{i+1}) \in \delta$. A run is *accepting* if the last state $q_n \in F$. A word is *accepted* by $\mathcal{A}$ if it has an accepting run. The *language* of $\mathcal{A}$, written A, is the set of words accepted by $\mathcal{A}$. A language is *regular* if it can be recognised by a finite automaton. A finite automaton is a *deterministic finite automaton (DFA)* if $|\{q' : (q, a, q') \in \delta\}| \leq 1$ for each $q \in Q$ and $a \in \Sigma$. Otherwise, it is a *non-deterministic finite automaton (NFA)*. It is well-known that DFAs suffice to recognise regular languages. A finite automaton over alphabet $\Sigma \times \Sigma$ is also called a *transducer*.

As a convention, we denote automata in calligraphic fonts $\mathcal{A}, \mathcal{B}, \ldots$, and the corresponding regular languages in italic roman fonts $A, B, \ldots$ whenever the correspondence is clear from the context.

Operations on regular languages

The *concatenation* of two words u and v, denoted as $u \cdot v$, is a word w of size $|u| + |v|$ such that $w[i] = u[i]$ for $i \in \{1, \ldots, |u|\}$ and $w[|u| + i] = v[i]$ for $i \in \{1, \ldots, |v|\}$. The concatenation of two languages is defined as $L \cdot M := \{w \cdot u : w \in L,\ u \in M\}$. We use $L^n := L \cdots L$ to denote the result of concatenating L with itself for n times. The *Kleene star* of a language L is defined as $L^* := \bigcup_{n \geq 0} L^n$, with $L^0 := \{\varepsilon\}$ by convention. We use L^+ as a shorthand of $L \cdot L^*$. We stipulate that exponentiations (e.g. L^n, L^*, L^+, etc.) have a higher precedence than concatenations, which in turn have a higher precedence than set operations. To simplify notation, we often omit the concatenation operator, and

write a singleton set $\{a\}$ as a whenever there is no danger of confusions. For example, we may write a^+b instead of $\{a\} \cdot \{a\}^* \cdot \{b\}$.

Given a language $L \subseteq \Sigma^*$, we call $\Sigma^* \setminus L$ the *complement* of L. A mapping h between two languages is a *word homomorphism* if $h(w) = h(w_1) \cdots h(w_n)$ holds whenever $w = w_1 \cdots w_n$. Notice that this implies $h(\varepsilon) = \varepsilon$. The following closure properties of regular languages are standard.

Theorem 2.1.1 ([Kle51], cf. [HMU07; Koz12])**.** *Regular languages are closed under the operations of union, intersection, concatenation, complementation, Kleene star, and word homomorphism. Furthermore, one can effectively compute a finite automaton recognising the outcome of any of these operations given the finite automata recognising the input languages.*

Regular expressions

A *regular expression* over Σ is a finite string w representing some language $L_w \subseteq \Sigma^*$. Let RE denote the regular expressions over Σ. Then the strings in RE and the languages they represent are recursively defined as follows.

- (Basic sets) $\perp \in \mathsf{RE}$ with $L_\perp := \emptyset$, and for all $a \in \Sigma \uplus \{\varepsilon\}$, $a \in \mathsf{RE}$ with $L_a := \{a\}$.
- (Union) If $w_1, w_2 \in \mathsf{RE}$, then $(w_1 + w_2) \in \mathsf{RE}$ with $L_{(w_1+w_2)} := L_{w_1} \cup L_{w_2}$.
- (Concatenation) If $w_1, w_2 \in \mathsf{RE}$, then $(w_1 \cdot w_2) \in \mathsf{RE}$ with $L_{(w_1 \cdot w_2)} := \{u_1 \cdot u_2 : u_1 \in L_{w_1},\ u_2 \in L_{w_2}\}$.
- (Kleene closure) If $w \in \mathsf{RE}$, then $(w^*) \in \mathsf{RE}$ with $L_{(w^*)} := \{u_1 \cdots u_n : u_1, \ldots, u_n \in L_w,\ n \geq 0\}$.

It is well-known that languages representable by regular expressions coincide with regular languages. Note that, whereas complementation and intersection are among the closure operations over regular languages, they do not have counterpart syntactic operators in the language of regular expressions.

Theorem 2.1.2. *Given a regular expression w over a finite alphabet Σ, one can compute a finite automaton $\mathscr{A}$ recognising the language L_w. Conversely, given a finite automaton $\mathscr{A}$ over Σ, one can compute a regular expression w such that $\mathscr{A}$ recognises L_w.*

For convenience, we often identify a regular expression with the regular language it represents by writing w in place of L_w when there is no danger of confusions.

Regular relations over words

Let $\Sigma_\#$ denote $\Sigma \uplus \{\#\}$ where $\# \notin \Sigma$ is regarded as a *blank symbol*. The *convolution* $w_1 \otimes \cdots \otimes w_n$ of n words $w_1, \ldots, w_n \in \Sigma^*$ is the word $w \in (\Sigma_\#^n)^*$ satisfying (i) $|w| = \max\{|w_1|, \ldots, |w_n|\}$, and (ii) for $i \in \{1, \ldots, |w|\}$, the i-th symbol of w is $w[i] := (a_1, \ldots, a_n)$, where

$$a_k := \begin{cases} w_k[i] & \text{if } |w_k| \geq i, \\ \# & \text{otherwise.} \end{cases}$$

In other words, w is the shortest word obtained by juxtaposing $w_1, \ldots, w_n$ and padding the shorter words with the blank symbol #. For example, $abb \otimes aa = (a,a)(b,a)(b,\#) \in (\Sigma_\#^2)^*$ when $\{a,b\} \subseteq \Sigma$. The convolution of two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ is the set of the convolutions of their words: $L_1 \otimes L_2 := \{w_1 \otimes w_2 : w_1 \in L_1,\ w_2 \in L_2\}$. It is easy to see that if L_1 and L_2 are regular languages, then $L_1 \otimes L_2$ is also a regular language.

Given an n-ary relation $R \subseteq (\Sigma^*)^n$ over words, we define the *language representation* of R as $\langle R \rangle := \{w_1 \otimes \cdots \otimes w_n : (w_1, \ldots, w_n) \in R\} \subseteq (\Sigma_\#^n)^*$. A relation $R \subseteq (\Sigma^*)^n$ is *regular* if and only if $\langle R \rangle \subseteq (\Sigma_\#^n)^*$ is a regular language. A unary regular relation is also called a *regular set*. The following closure properties of regular relations are standard, see, e.g., [HMU07].

- Regular relations are closed under projection and Cartesian product.
- Regular relations of the same arity are closed under intersection and union.
- Let $E \subseteq A$ be a regular set, and $R \subseteq A \times B$, $T \subseteq B \times C$ be regular binary relations. Then $R(E)$ is a regular set, and $R \circ T$ is a regular relation.

An n-ary relation $R \subseteq (\Sigma^*)^n$ is *length-preserving* if $(w_1, \ldots, w_n) \in R$ implies $|w_i| = |w_j|$ for $i, j \in \{1, \ldots, n\}$. Given two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, the *length-preserving convolution* of L_1 and L_2, denoted as $L_1 \odot L_2$, is the set of the convolution of

their words without padding: $L_1 \odot L_2 := (L_1 \otimes L_2) \cap (\Sigma_1 \times \Sigma_2)^*$. It is easy to see that if L_1 and L_2 are regular languages, then $L_1 \odot L_2$ is also a regular language.

2.2 Logic and structures

2.2.1 Structures

A *vocabulary* is a countable set comprising two types of objects: function symbols and relation symbols. Each symbol in a vocabulary is associated with an *arity*, which is a natural number specifying the number of arguments the symbol can take. A symbol of arity n is said to be *n-ary*. A relation symbol always has a nonzero arity. A unary relation symbol is also called a *predicate symbol*, and a nullary function symbol is also called a *constant symbol*.

A *structure* over a vocabulary σ, or a *σ-structure* for short, is a tuple $\mathfrak{S} := \langle S, \{s^{\mathfrak{S}}\}_{s\in\sigma}\rangle$ where S is a nonempty set called *universe* (a.k.a *domain of discourse*), and $\cdot^{\mathfrak{S}}$ is a function, called *interpretation*, assigning each constant symbol $c \in \sigma$ an element $c^{\mathfrak{S}} \in S$, each n-ary function symbol $f \in \sigma$ a function $f^{\mathfrak{S}} : S^n \to S$, and each n-ary relation symbol $r \in \sigma$ a relation $r^{\mathfrak{S}} \subseteq S^n$. When the vocabulary σ is finite, say $\sigma := \{s_1, \ldots, s_n\}$, we often simply write $\mathfrak{S}$ as $\langle S, s_1^{\mathfrak{S}}, \ldots, s_n^{\mathfrak{S}}\rangle$. In this case, we may further omit the superscript $\mathfrak{S}$ and write $\mathfrak{S}$ as $\langle S, s_1, \ldots, s_n\rangle$ when there is no danger of ambiguity in the interpretations of the symbols $s_1, \ldots, s_n$.

A σ-structure is *relational* if σ has only relation symbols. We say that a relational structure is the *relational variant* of another structure if the former can be obtained from the latter by replacing functions with their graphs. More precisely, suppose that structures $\mathfrak{S}$ and $\mathfrak{T}$ have vocabularies σ and τ, respectively. Then $\mathfrak{S}$ is the relational variant of $\mathfrak{T}$ if (i) $\mathfrak{S}$ and $\mathfrak{T}$ have the same universe, (ii) σ can be obtained from replacing each n-ary function symbol $f \in \tau$ with an $(n+1)$-ary relation symbol r_f, and (iii) $\mathfrak{S}$ interprets r_f as the graph of the function $f^{\mathfrak{T}}$ for each relation symbol $r_f \in \sigma$.

We say that two relational structures $\mathfrak{S}$ and $\mathfrak{T}$ are *isomorphic* if (i) $\mathfrak{S}$ and $\mathfrak{T}$ have the same vocabulary, say σ, and (ii) there is a computable bijective mapping $\eta : S \to T$

between the universes of $\mathfrak{S}$ and $\mathfrak{T}$ such that for every k-ary relation symbol $r \in \sigma$ and elements $a_1, \ldots, a_k \in S$, it holds that $(a_1, \ldots, a_k) \in r^{\mathfrak{S}} \iff (\eta(a_1), \ldots, \eta(a_k)) \in r^{\mathfrak{T}}$. We regard two general structures as isomorphic if their relational variants are isomorphic.

Example 2.2.1 (Word structure)**.** A word over a finite alphabet can be represented as a finite structure. Formally, given an alphabet Σ, we define a vocabulary $\sigma_\Sigma := \{<\} \cup \{P_a : a \in \Sigma\}$ where $<$ is a binary relation symbol and the P_a's are predicate symbols. Given a word $w \in \Sigma^*$ of length n, the *word structure* of w is a σ_Σ-structure $\mathfrak{S}_w := \langle \{1, \ldots, n\}, <, \{P_a\}_{a \in \Sigma} \rangle$, where $<$ is the usual order on natural numbers, and each $P_a \subseteq \{1, \ldots, n\}$ consists of the positions where the letter a occurs in w. For example, if $w = ababa$, then $P_a = \{1, 3, 5\}$ and $P_b = \{2, 4\}$. #

Automatic structures

A relational structure $\mathfrak{A} := \langle A, \{r^{\mathfrak{A}}\}_{r \in \sigma} \rangle$ is *automata presentable* if there is a collection of finite automata $\{\mathscr{A}\} \uplus \{\mathscr{A}_r\}_{r \in \sigma}$ such that $\mathscr{A}$ recognises $\langle A \rangle$, and $\mathscr{A}_r$ recognises $\langle r^{\mathfrak{A}} \rangle$ for each $r \in \sigma$. A relational structure $\mathfrak{S} := \langle S, \{r^{\mathfrak{S}}\}_{r \in \sigma} \rangle$ is an *automatic structure* if it is isomorphic to an automata presentable structure $\mathfrak{A}$. If $\eta : S \to A$ yields such an isomorphism, we say that $\mathfrak{A}$ is the *regular presentation of* $\mathfrak{S}$ *under the encoding* η. As a result, a relational structure is automatic if and only if it has a regular presentation. We regard a non-relational structure as automatic if its relational variant is automatic.

As a first observation, note that all finite structures are automatic. In fact, we can arbitrarily specify the encoding of a finite structure since finite relations on words are trivially regular. Our purpose of using automatic presentations is hence to represent infinite structures. Below we introduce some well-known examples of infinite automatic structures. We begin with structures that are presentable with regular relations over a unary alphabet.

Example 2.2.2. The linear order on natural numbers $\langle \mathbb{N}, \leq \rangle$ is automatic, since it has a regular presentation $\langle \mathtt{1}^*, \{(x, xy) : x, y \in \mathtt{1}^*\} \rangle$ under the unary encoding $\eta : n \mapsto \mathtt{1}^n$. Similarly, the structure $\langle \mathbb{N}, \{\equiv_k\}_{k \geq 1} \rangle$ with $\equiv_k$ standing for the congruence relation

modulo k (i.e. $x \equiv_k y$ iff k divides $x - y$) is automatic, since it has a regular presentation $\langle 1^*, \{R_k\}_{k\geq 1}\rangle$ where R_k is the symmetric closure of $\{(xy^n, x) : x, y \in 1^*,\ k \text{ divides } n\}$. #

We proceed to consider automatic structures presentable with a binary alphabet.

Example 2.2.3. The addition relation over natural numbers $\langle \mathbb{N}, +\rangle$ is automatic. The encoding is given by the binary expansion of the naturals with the least significant bit first and without leading zeros. This encoding yields a bijection between natural numbers and the words in $\{0\} \cup \{0,1\}^*1$. It is easy to see that the addition relation $\{(x, y, z) : x + y = z\}$ is regular under this encoding: the finite automaton just needs to keep track of the carry bit in its states. #

Example 2.2.4. The (strict) linear order on rational numbers $\langle \mathbb{Q}, <\rangle$ is automatic. Consider a structure $\mathfrak{S} := \langle \{0,1\}^*1, \sqsubset\rangle$, where $\sqsubset$ is a lexicographical order defined such that $x \sqsubset y$ iff either (i) x is a proper prefix of y, or (ii) $x = z0x'$ and $y = z1y'$ for some $z, x', y' \in \{0,1\}^*$. It is easy to check that $\mathfrak{S}$ is an automata presentable linear order. Now we show that $\mathfrak{S}$ is dense without endpoints.[1] To see that $\mathfrak{S}$ is dense, consider two words $x, y \in \{0,1\}^*1$ with $x \sqsubset y$. If x is a prefix of y, say $y = x0^n1z$, then we have $x \sqsubset x0^{n+1}1 \sqsubset y$. Otherwise, x is not a prefix of y and we have $x \sqsubset x1 \sqsubset y$. To see that $\mathfrak{S}$ has no endpoints, just note that $0x \sqsubset x \sqsubset 1x$ for any $x \in \{0,1\}^*1$. Since countable dense linear orders without endpoints are unique up to isomorphism (cf. [Koz12]), we conclude that $\langle \mathbb{Q}, <\rangle$ is isomorphic to $\mathfrak{S}$ and hence is automatic. #

Finally, let us introduce some automata presentations with general alphabets. Before we proceed, we define the *configuration graph* of a system as a structure $\langle V, E\rangle$ where V consists of elements called *configurations*, and $E \subseteq V \times V$ consists of pairs called *transitions*. Intuitively, $(u, v) \in E$ means that system can move from configuration u to v in one step.

Example 2.2.5. The configuration graph $\langle V, E\rangle$ of a pushdown automaton is automatic. A pushdown automaton is a 6-tuple $(\Sigma, \Gamma, Q, q_0, F, \{\delta_a\}_{a\in\Sigma})$ with a finite input alphabet

[1] A strict linear order $\langle S, \prec\rangle$ is *dense* if for any $x, y \in S$ satisfying $x \prec y$, there exists $z \in S$ such that $x \prec z \prec y$. Moreover, $\langle S, \prec\rangle$ *has no endpoints* if for any $z \in S$, there exist $x, y \in S$ such that $x \prec z \prec y$.

Σ, a finite stack alphabet Γ, a finite set of control states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, and a transition relation $\delta_a \subseteq Q \times \Gamma \times Q \times (\Gamma \uplus \{\varepsilon\})$ for each $a \in \Sigma$. A configuration is a word $qw \in Q\Gamma^*$ consisting of a control state $q \in Q$ and a "stack" $w \in \Gamma^*$. Given an input symbol $a \in \Sigma$, the pushdown automaton can move from configuration qxz to $q'yz$ if $(q, x, q', y) \in \delta_a$. The configuration graph $\langle V, E\rangle$ is therefore defined with $V := Q\Gamma^*$ and $E := \bigcup_{a\in\Sigma}\{(qxz, q'yz) : z \in \Gamma^* \text{ and } (q, x, q', y) \in \delta_a\}$, which is a regular relation since both Σ and δ_a are finite. Consequently, $\langle V, E\rangle$ is automata presentable and hence automatic. #

Example 2.2.6. The configuration graph $\langle V, E\rangle$ of a Turing machine is automatic. A Turing machine is a 5-tuple $(\Gamma, Q, q_0, F, \delta)$ comprised of a finite tape alphabet Γ, a finite set of control states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, and a transition relation $\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$. A configuration is a word $uqv \in \Gamma^* Q \Gamma^*$ with $u, v \in \Gamma^*$ representing the tape content to the left and right of the head, and $q \in Q$ indicating the current state. A Turing machine can transit from configuration $uxqyv$ to $ux'q'y'v$ for $x, x', y, y' \in \Gamma \cup \{\varepsilon\}$ and $q, q' \in Q$ in accordance with the transition relation δ. Since the number of possible transitions is finite, and each transition involves only a bounded number of changes, the edge relation E is regular. Therefore, the configuration graph $\langle V, E\rangle$ is automata presentable and hence automatic. #

2.2.2 FO : first-order logic

First-order logic is based on a vocabulary, a set of variables, and a syntax that defines how formulae have to be constructed from variables, symbols of the vocabulary, and logical symbols such as $\neg$, $\wedge$, $\forall$, etc. A formula is interpreted in a structure. By restricting the interpretation or syntax of a formula, we can obtain different instances of first-order logic. Let us first introduce the syntax and the semantics of first-order logic.

In first-order logic, we build formulae from a vocabulary as well as a countable number of first-order variables. A *term* is either a variable or an object in form of $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol. The syntax of first-order formulae is defined inductively as follows.

- If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula.
- If $r \in \sigma$ is an n-ary relation symbol and $t_1, \ldots, t_n$ are terms, then $r(t_1, \ldots, t_n)$ is a formula.
- If ϕ and ψ are formulae, then $\phi \wedge \psi$, $\phi \vee \psi$, and $\neg\phi$ are formulae.
- If ϕ is a formula and x is a variable, then $\forall x.\, \phi$ and $\exists x.\, \phi$ are formulae.

We use the standard shorthand for logical operators such as $t_1 \neq t_2 := \neg(t_1 = t_2)$, $\phi \Rightarrow \psi := \neg\phi \vee \psi$, and $\phi \Leftrightarrow \psi := (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$. The symbols $\forall$ and $\exists$ are called *universal* and *existential* quantifiers, respectively. A formula is a *subformula* of a formula ϕ if it has been used in the inductive construction of ϕ. A formula is *atomic* if it does not have a subformula. A *literal* refers to an atomic formula or its negation. A variable x is *quantified* in a formula ϕ if ϕ has a subformula in form of $\delta x.\, (\ldots x \ldots)$ for $\delta \in \{\forall, \exists\}$; otherwise x is a *free* variable in ϕ. A formula is a *sentence* if it does not have any free variable. A formula is *quantifier-free* if it does not have any quantified variable. We may write a formula ϕ as $\phi(\overline{x})$ to emphasise that the free variables in ϕ are among $\overline{x}$. Given a formula ϕ, a term t, and a variable x, we use $\phi[t/x]$ to denote the formula obtained by substituting t for all free occurrences of x in ϕ. Note that $\phi[t/x] = \phi$ if x is not free in ϕ. Given a formula $\phi(x_1, \ldots, x_n)$ and terms $t_1, \ldots, t_n$, we write $\phi(t_1, \ldots, t_n)$ for the formula $\phi[t_1/x_1] \cdots [t_n/x_n]$.

The semantics of first-order formulae is determined based on structures of an appropriate vocabulary. Fix a σ-structure $\mathfrak{S}$. We extend the interpretation $\cdot^{\mathfrak{S}}$ to σ-terms by mapping each variable x to itself, namely, $x^{\mathfrak{S}} := x$, and recursively mapping each term $t := f(t_1, \ldots, t_n)$ to $t^{\mathfrak{S}} := f^{\mathfrak{S}}(t_1^{\mathfrak{S}}, \ldots, t_n^{\mathfrak{S}})$. The notion of truth with respect to the

structure $\mathfrak{S}$ is defined as below.

$$
\begin{array}{rcl}
\mathfrak{S} \models \mathsf{true} & \text{and} & \mathfrak{S} \not\models \mathsf{false} \\
\mathfrak{S} \models \neg\phi & \text{iff} & \mathfrak{S} \not\models \phi \\
\mathfrak{S} \models t_1 = t_2 & \text{iff} & t_1^{\mathfrak{S}} = t_2^{\mathfrak{S}} \\
\mathfrak{S} \models r(t_1, \ldots, t_n) & \text{iff} & (t_1^{\mathfrak{S}}, \ldots, t_n^{\mathfrak{S}}) \in r^{\mathfrak{S}} \\
\mathfrak{S} \models \phi \vee \psi & \text{iff} & \mathfrak{S} \models \phi \text{ or } \mathfrak{S} \models \psi \\
\mathfrak{S} \models \phi \wedge \psi & \text{iff} & \mathfrak{S} \models \phi \text{ and } \mathfrak{S} \models \psi \\
\mathfrak{S} \models \exists x.\, \phi & \text{iff} & \text{there exists } a \in S \text{ such that } \mathfrak{S} \models \phi[a/x] \\
\mathfrak{S} \models \forall x.\, \phi & \text{iff} & \text{for all } a \in S \text{ it holds that } \mathfrak{S} \models \phi[a/x]
\end{array}
$$

A σ-sentence ϕ is *true* in $\mathfrak{S}$ iff $\mathfrak{S} \models \phi$. Two σ-formulae ϕ and ψ are *equivalent* in $\mathfrak{S}$ if the sentence $\forall \overline{x}.\, (\phi(\overline{x}) \Leftrightarrow \psi(\overline{x}))$ is true in $\mathfrak{S}$. We say that ϕ and ψ are *logically equivalent*, written as $\phi \equiv \psi$, if they are equivalent in every σ-structure. Let $\phi(\overline{x})$ be a formula with precisely n free variables $x_1, \ldots, x_n$. We say that $\phi(\overline{x})$ is *satisfiable* in $\mathfrak{S}$ (or $\mathfrak{S}$ *satisfies* $\phi(\overline{x})$) if there exists a mapping $\rho : \{\overline{x}\} \to S$ such that $\phi(\rho(x_1), \ldots, \rho(x_n))$ is true in $\mathfrak{S}$. In this case, we say $\mathfrak{S}$ is a *model* of $\phi(\overline{x})$ and write $\mathfrak{S}, \rho \models \phi(\overline{x})$. We say $\phi(\overline{x})$ is *valid* in $\mathfrak{S}$ if $\neg\phi(\overline{x})$ is not satisfiable in $\mathfrak{S}$. We use $\phi^{\mathfrak{S}}$ to denote the relation $\{(\overline{a}) \in S^n : \mathfrak{S} \models \phi(\overline{a})\}$. We say that an n-ary relation R is *first-order definable* in $\mathfrak{S}$ if there is a first-order formula ϕ with precisely n free variables such that $R = \phi^{\mathfrak{S}}$. A function f is first-order definable in $\mathfrak{S}$ if the graph of f is first-order definable in $\mathfrak{S}$.

A *logic* is a set of formulae, possibly constrained under some syntactic or semantic specification. We say that a relation is *expressible* in a logic if it is definable by some formula in that logic. The class of sets and relations expressible in a logic determines the expressive power of the logic. We use $\mathrm{FO}(\sigma)$ to denote the set of first-order formulae defined in the vocabulary σ, and use $\mathrm{FO}(\mathfrak{S})$ to denote the set of first-order formulae defined over the vocabulary of $\mathfrak{S}$. It is easy to see that $\mathrm{FO}(\mathfrak{S})$ and $\mathrm{FO}(\mathfrak{T})$ have the same expressive power if $\mathfrak{S}$ is the relational variant of $\mathfrak{T}$.

A *theory* refers to a specific set of sentences. The theory of a structure $\mathfrak{S}$ is the set of

sentences that are true in $\mathfrak{S}$. A structure $\mathfrak{S}$ is a *model* of a theory if all sentences in that theory are true in $\mathfrak{S}$. Now fix a theory T. A formula ϕ is *T-satisfiable* if its existential closure $\exists \overline{x}.\, \phi(\overline{x})$ is satisfiable in some model of T. A formula ϕ is *T-valid* if $\neg\phi$ is not T-satisfiable. We write $T \models \phi$ if every model of T is also a model of ϕ. Two formulae ϕ and ψ are *T-equivalent* if $T \models \forall \overline{x}.\, \phi(\overline{x}) \Leftrightarrow \psi(\overline{x})$.

Example 2.2.7 (Propositional logic)**.** *Propositional logic* refers to the set of quantifier-free first-order formulae over the structure $\langle\{\mathsf{true}, \mathsf{false}\}\rangle$. Since the logic has an empty vocabulary, the truth of a propositional formula is solely determined by the evaluation of the variables to Boolean values in accordance with *Boolean algebra* [Boo54]. #

Example 2.2.8 (Difference arithmetic)**.** The first-order theory of the linear order on natural numbers, $\mathrm{FO}(\mathbb{N}, \leq)$, will play an important role in our subsequent development. This theory is not as trivial as it appears to be at first glance: it is in fact equally expressive to *difference arithmetic* [Dil89], a first-order arithmetic over atomic difference bound constraints in form of $x \sim y$, $x \sim z$, and $x \sim y + z$, where x and y are variables over the naturals, z is an integer, and $\sim\, \in \{\leq, <, \geq, >, =, \neq\}$. See Example 2.2.11 for a further discussion. #

Example 2.2.9 (Logical characterisation of languages)**.** Recall from Example 2.2.1 that a word $w \in \Sigma^n$ can be represented as a σ_Σ-structure $\mathfrak{S}_w := \langle\{1, \ldots, n\}, <, \{P_a\}_{a\in\Sigma}\rangle$. A formula $\phi \in \mathrm{FO}(\sigma_\Sigma)$ therefore defines a language $\{w \in \Sigma^* : \mathfrak{S}_w \models \phi\}$. For example,

$$P_a(1) \wedge \forall i.\, P_a(i) \Rightarrow \exists j.\, \exists k.\, P_b(j) \wedge P_b(k) \wedge succ(i,j) \wedge succ(j,k) \\ \wedge\, (max(k) \vee (\exists t.\, succ(k,t) \wedge P_a(t)))$$

defines the regular language $(abb)^+$, where $succ(x,y) := \forall z.\, x < z \Rightarrow (y < z \vee y = z)$ is the successor relation over positions, and $max(x) := \forall z.\, z < x \vee z = x$ captures the rightmost position of a word. We hence say that $(abb)^+$ is *definable* in $\mathrm{FO}(\sigma_\Sigma)$. In fact, it is well known that the languages definable in $\mathrm{FO}(\sigma_\Sigma)$ coincide with the *star-free languages* [MP71], which is a strict subclass of regular languages that can be generated from finite languages using Boolean operations and concatenation. See [DG08] for a survey of first-order definable languages. #

Many-sorted logic

An interpretation of a vocabulary is often expected to assign different kinds of objects to different kinds of symbols. For example, it should assign a relation to a relation symbol and a function to a function symbol, but not the other way around. We can refine this notion and annotate *every* symbol in a vocabulary to specify the kinds of objects a symbol is allowed to be assigned. This leads us to the definition of *many-sorted* logic.

A many-sorted vocabulary σ is associated with a set $\{\mathbb{S}_1, \ldots, \mathbb{S}_m\}$ of *sort symbols*. The symbols in the vocabulary are arranged as follows.

- Each variable and constant symbol is associated with a sort symbol in *Sort*.
- Each n-ary relation symbol is associated with sort $\mathbb{S}_{i_1} \times \cdots \times \mathbb{S}_{i_n}$, where $i_1, \ldots, i_n \in \{1, \ldots, m\}$.
- Each n-ary function symbol is associated with sort $\mathbb{S}_{i_1} \times \cdots \times \mathbb{S}_{i_n} \to \mathbb{S}_{i_{n+1}}$, where $i_1, \ldots, i_{n+1} \in \{1, \ldots, m\}$.
- For each $\mathbb{S}_i \in Sort$, there are quantifier symbols $\forall_i$ and $\exists_i$ associated with sort $\mathbb{S}_i$.
- For each $\mathbb{S}_i \in Sort$, there is an equality symbol $=_i$ associated with sort $\mathbb{S}_i$.

We say that a symbol is *of sort* $\mathbb{S}_i$ if and only if it is associated with the sort symbol $\mathbb{S}_i$. Each term is also associated with a unique sort specified as follows: (i) a variable or constant symbol of sort $\mathbb{S}_i$ is a term of sort $\mathbb{S}_i$, and (ii) if f is a function symbol of sort $\mathbb{S}_{i_1} \times \cdots \times \mathbb{S}_{i_n} \to \mathbb{S}_{i_{n+1}}$ and $t_1, \ldots, t_n$ are terms of sorts $\mathbb{S}_{i_1}, \ldots, \mathbb{S}_{i_n}$, then $f(t_1, \ldots, t_n)$ is a term of sort $\mathbb{S}_{i_{n+1}}$. An atomic formula is either in form of $t_1 =_i t_2$, where t_1 and t_2 are terms of sort $\mathbb{S}_i$, or in form of $r(t_1, \ldots, t_n)$, where r is a relation symbol of sort $\mathbb{S}_{i_1} \times \cdots \times \mathbb{S}_{i_n}$ and $t_1, \ldots, t_n$ are terms of sorts $\mathbb{S}_{i_1}, \ldots, \mathbb{S}_{i_n}$, respectively.

A many-sorted σ-structure $\mathfrak{S}$ with sorts $\{\mathbb{S}_1, \ldots, \mathbb{S}_m\}$ is a tuple $(\{S_i\}_{i=1}^m, \{s^{\mathfrak{S}}\}_{s \in \sigma})$, where each S_i is a nonempty set called the *universe of sort* $\mathbb{S}_i$, and the interpretation $\cdot^{\mathfrak{S}}$ assigns each symbol $s \in \sigma$ an object of correct sort:

- Each equality symbol $=_i$ is assigned the identity relation over S_i.

– Each constant symbol c of sort $\mathbb{S}_i$ is assigned an element $c^{\mathfrak{S}} \in S_i$.

– Each relation symbol r of sort $\mathbb{S}_{i_1} \times \cdots \times \mathbb{S}_{i_n}$ is assigned a relation $r^{\mathfrak{S}} \subseteq S_{i_1} \times \cdots \times S_{i_n}$.

– Each function symbol f of sort $\mathbb{S}_{i_1} \times \cdots \times \mathbb{S}_{i_n} \to \mathbb{S}_{i_{n+1}}$ is assigned a function $f^{\mathfrak{S}} : S_{i_1} \times \cdots \times S_{i_n} \to S_{i_{n+1}}$.

The definitions concerning truth and satisfaction can be naturally derived in a similar manner to one-sorted logic, except that we shall use $\forall_i$ is to mean "for all elements in S_i" and use $\exists_i$ is to mean "for some element in S_i." To simplify notation, we may omit the subscript i of the symbols $\forall_i$, $\exists_i$, and $=_i$ in a formula when there is no danger of confusion.

Normal forms

A formula is in *negation normal form* (NNF) if the negation symbol only appears inside the literals. A formula can be converted to a logically equivalent formula in NNF by repeatedly applying the following three rewrite rules until a fixed point is reached: (i) replace syntactic sugar such as $\Rightarrow$ and $\Leftrightarrow$ with their definitions, (ii) use De Morgan's laws to push negation inwards, and (iii) eliminate double negations. It is easy to see that the above rewriting process does produce a logically equivalent formula in negation normal form.

A formula is in *prenex normal form* (PNF) if it is written in form of a string of quantifiers and quantified variables, called the *quantifier prefix*, followed by a quantifier-free formula, called the *matrix*. A first-order formula can be transformed into a logically equivalent formula in PNF as follows. We first convert the given formula to NNF and rename the variables so that each variable name is used at most once by the formula. We then move quantifiers outwards by repeatedly applying the following rewrite rules

in turn, where $\delta \in \{\forall, \exists\}$, $\circ \in \{\wedge, \vee\}$, and x is a fresh variable:

$$\begin{aligned}(\forall y.\, \phi) \circ (\forall z.\, \psi) &\rightarrow \forall x.\, (\phi[x/y] \circ \psi[x/z]) \\ \phi \circ (\delta y.\, \psi) &\rightarrow \delta x.\, (\phi \circ \psi[x/y]) \\ (\delta y.\, \phi) \circ \psi &\rightarrow \delta x.\, (\phi[x/y] \circ \psi)\end{aligned}$$

It is easy to see that these rules preserve logical equivalence and, after a fixed point is reached, the resulting formula is in prenex normal form.

We introduce two other useful normal forms of quantifier-free formulae: *conjunctive normal form* (CNF) and *disjunctive normal forms* (DNF). A formula is in CNF if it is in form of $\bigwedge_i \psi_i$, where each ψ_i, called a *conjunct*, is a disjunction of literals. A formula is in DNF if it is in form of $\bigvee_i \psi_i$, where each ψ_i, called a *disjunct*, is a conjunction of literals. It is well-known that every propositional formula can be converted into a logically equivalent formula in CNF or DNF based on De Morgan's laws, double negation elimination, and the distributive law, see e.g., [HMU07]. Now, consider a quantifier-free formula ϕ. We define the *propositional skeleton* of ϕ as the propositional formula produced by (i) associating each atomic formula f in ϕ with a fresh auxiliary variable x_f, and then (ii) replacing the occurrences of each atomic formula f in ϕ with the variable x_f. For example, if $\phi := (a < b) \wedge (a = c) \wedge \neg(b < c)$, then the propositional skeleton of ϕ is $x_{a<b} \wedge x_{a=c} \wedge \neg x_{b<c}$. Finally, we define the CNF (resp. DNF) of a quantifier-free formula ϕ as the formula obtained by first computing the CNF (resp. DNF) of the propositional skeleton of the NNF of ϕ, and then restoring the atomic formulae from the auxiliary variables they are associated with. For example, given $a = 2 \Rightarrow a > 1 \wedge a < 3$, we can compute an logically equivalent formula in CNF as $(\neg(a = 2) \vee a > 1) \wedge (\neg(a = 2) \vee a < 3)$.

First-order interpretations

Fix two finite vocabularies σ and τ such that τ is relational. We define a *first-order interpretation* $\mathscr{I}$ as a list of first-order σ-formulae

$$\mathscr{I} := (\Delta(x), \{\Phi_r(\overline{x})\}_{r \in \tau}).$$

Given a σ-structure $\mathfrak{S}$, called the *host* structure, the interpretation $\mathscr{I}$ determines a τ-structure $\mathscr{I}^{\mathfrak{S}} := \langle \Delta^{\mathfrak{S}}, \{\Phi_r^{\mathfrak{S}}\}_{r\in\tau}\rangle$ where the universe $\Delta^{\mathfrak{S}}$ is $\{a \in S : \mathfrak{S} \models \Delta(a)\}$, and each relation symbol $r \in \tau$ is interpreted as

$$\Phi_r^{\mathfrak{S}} = \{(a_1, \ldots, a_n) \in S^n : \mathfrak{S} \models \Phi_r(a_1, \ldots, a_n),\ n \text{ is the arity of } r\}.$$

Furthermore, a first-order interpretation $\mathscr{I}$ naturally induces a syntactical transformation $\cdot^{\mathscr{I}}$ from τ-formulae to σ-formulae, which are defined recursively as follows:

$$\begin{aligned}
(r(\overline{x}))^{\mathscr{I}} &:= \Phi_r(\overline{x}) \\
(x = y)^{\mathscr{I}} &:= x = y \\
(\exists x.\, \phi)^{\mathscr{I}} &:= \exists x.\, \Delta(x) \wedge \phi^{\mathscr{I}} \\
(\forall x.\, \phi)^{\mathscr{I}} &:= \forall x.\, \Delta(x) \Rightarrow \phi^{\mathscr{I}}
\end{aligned}
\qquad
\begin{aligned}
(\neg\phi)^{\mathscr{I}} &:= \neg\phi^{\mathscr{I}} \\
(\phi \wedge \psi)^{\mathscr{I}} &:= \phi^{\mathscr{I}} \wedge \psi^{\mathscr{I}} \\
(\phi \vee \psi)^{\mathscr{I}} &:= \phi^{\mathscr{I}} \vee \psi^{\mathscr{I}}
\end{aligned}$$

That is, $\phi^{\mathscr{I}}$ is obtained from ϕ by replacing each atomic relation with its definition in $\mathscr{I}$, as well as restricting quantification to the elements satisfying $\Delta(x)$. A structure $\mathfrak{T}$ is *first-order interpretable* in a structure $\mathfrak{S}$ if there is a first-order interpretation $\mathscr{I}$ defined in the vocabulary of $\mathfrak{S}$ such that $\mathscr{I}^{\mathfrak{S}}$ is isomorphic to $\mathfrak{T}$ or its relational variant.

When $\mathfrak{T}$ is first-order interpretable in $\mathfrak{S}$, it is not difficult to show (e.g. by structural induction) that for every first-order sentence ϕ defined over $\mathfrak{T}$, we have $\mathfrak{T} \models \phi$ if and only if $\mathfrak{S} \models \phi^{\mathscr{I}}$. Particularly, any set and relation that is first-order definable in the interpreted structure $\mathfrak{T}$ is also definable in the host structure $\mathfrak{S}$. A first-order interpretation therefore provides a convenient apparatus to reduce the first-order reasoning about the interpreted structure to that about the host structure.

Example 2.2.10. To illustrate the use of first-order interpretations, we show that the two structures $\langle \mathbb{N}, +, \times\rangle$ and $\langle \mathbb{N}, +, \cdot^2\rangle$ are first-order interpretable in each other, where $+$ and $\times$ are usual addition and multiplication over the naturals, and x^2 computes the square of x. To interpret $\langle \mathbb{N}, +, \times\rangle$ in $\langle \mathbb{N}, +, \cdot^2\rangle$, we define a first-order interpretation $\mathscr{I} := (\Delta(x), \Phi_+(x,y,z), \Phi_\times(x,y,z))$ of the relational variant of $\langle \mathbb{N}, +, \times\rangle$, such that $\Delta(x) := \mathsf{true}$, $\Phi_+(x,y,z) := x + y = z$, and $\Phi_\times(x,y,z) := z + z + x^2 + y^2 = (x+y)^2$. Similarly, $\langle \mathbb{N}, +, \cdot^2\rangle$ can be interpreted in $\langle \mathbb{N}, +, \times\rangle$ by defining $\Phi_{\cdot^2}(x,y) := y = x \times x$.

This result in fact indicates that the two theories $\mathrm{FO}(\mathbb{N}, +, \times)$ and $\mathrm{FO}(\mathbb{N}, +, \cdot^2)$ are equally expressive. #

Example 2.2.11. As another example, we show that difference arithmetic (see Example 2.2.8) is equally expressive to $\mathrm{FO}(\mathbb{N}, \leq)$. First, note that $x \sim y$ can be expressed as a Boolean combination of $x \leq y$ and $y \leq x$ for $\sim \in \{\leq, <, \geq, >\}$. Furthermore, note that the successor relation over $\mathbb{N}$ can be captured by

$$s(x,y) := x \neq y \wedge x \leq y \wedge \forall z.\ x \neq z \wedge x \leq z \Rightarrow y \leq z.$$

Thus, we can define a first-order interpretation that expresses $y \sim x + n$ as $\exists x_1 \ldots \exists x_n.\ y \sim x_n \wedge s(x, x_1) \wedge \cdots \wedge s(x_{n-1}, x_n)$, $y \sim x - n$ as $\exists x_1 \ldots \exists x_n.\ y \sim x_n \wedge s(x_n, x_{n-1}) \wedge \cdots \wedge s(x_1, x)$, and $y \sim n$ as $\exists x.\ y \sim x + n \wedge \forall z.\ x \leq z$. Since $\mathrm{FO}(\mathbb{N}, \leq)$ is trivially reducible to difference arithmetic, we conclude that $\mathrm{FO}(\mathbb{N}, \leq)$ and difference arithmetic are equally expressive over natural numbers. #

2.2.3 $\mathrm{FO}_{\mathrm{REG}}$: a first-order framework for regular relations

In this subsection, we introduce a theory that will play an important role throughout this thesis. Given a non-unary alphabet Σ, we define a structure

$$\mathfrak{U}_\Sigma := \langle \Sigma^*,\ \preceq,\ eqL,\ \{\prec_a\}_{a \in \Sigma} \rangle \tag{2.1}$$

where $\preceq$ is the prefix-of relation, eqL is the equal-length relation, and $\prec_a$ is the a-successor relation. More precisely, $w \preceq w'$ holds iff there exists $w'' \in \Sigma^*$ such that $w \cdot w'' = w'$, $eqL(w, w')$ holds iff $|w| = |w'|$, and $w \prec_a w'$ holds iff $w \cdot a = w'$. We use $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ to denote the first-order logic $\mathrm{FO}(\mathfrak{U}_\Sigma)$, and write $\mathrm{FO}_{\mathrm{REG}}$ when the alphabet Σ is inessential or clear from the context.

Note that $\mathfrak{U}_\Sigma$ can first-order interpret $\langle \mathbb{N}, \leq \rangle$ under unary encoding. More precisely, we can identify each $n \in \mathbb{N}$ with 1^n for a designated letter $1 \in \Sigma$, and define a first-order interpretation $\langle \Delta(x), \Phi_{\leq}(x,y) \rangle$ in $\mathfrak{U}_\Sigma$ with $\Delta(x) := \forall u.\, u \preceq x \wedge u \neq \varepsilon \Rightarrow \exists v.\, v \prec_1 u$ and $\Phi_{\leq}(x,y) := x \preceq y$. As discussed in Example 2.2.11, this encoding is capable of expressing difference bound constraints. The structure $\mathfrak{U}_\Sigma$ furthermore allows us to

reason about formulae involving word positions and lengths. For example, we can use $|w| = t$ as a shorthand of $eqL(w,t)$, and $|w| \leq t$ as a shorthand of $\exists u.\, eqL(w,u) \wedge u \preceq t$. Moreover, we can express $w[i] = a$ with the formula $\exists u.\, \exists v.\, u \preceq w \wedge v \preceq w \wedge |v| = i \wedge u \prec_a v$, meaning that a is the i-th letter of the word w, and express $w[i] = u[j]$ with $\bigvee_{a\in\Sigma}(w[i] = a \wedge u[j] = a)$. In the sequel, we shall allow formulae in $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ to contain constants $n \in \mathbb{N}$, relation such as $<$ and $\leq$, and functions $\cdot[\cdot]$, $|\cdot|$, $\cdot + n$, and $\cdot - n$, where $n \in \mathbb{N}$, with their expected sorts and interpretations.

Example 2.2.12. Fix a nonzero $m \in \mathbb{N}$ and a non-unary alphabet Σ. Suppose without loss of generality that $\top$ and $\bot$ are two distinct letters in Σ. Then the following formula in $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ defines the set of words $w \in \Sigma^*$ with length divisible by m :

$$\begin{aligned}\phi(x) := \exists u.\, |u| = |x| \wedge (u = \varepsilon \vee u[1] = \top) \wedge (\forall i.\, 1 \leq i \wedge i \leq |x| \wedge u[i] = \top \\ \Rightarrow ((\textstyle\bigwedge_{j=1}^{m-1} u[i+j] = \bot) \wedge (i + m - 1 = |u| \vee u[i+m] = \top))).\end{aligned} \tag{2.2}$$

Intuitively, the formula asserts that there is a word $u \in (\top\bot\cdots\bot)^*$ such that $|u| = |x|$ and u consists of repetitive segments of a $\top$ followed by $m-1$ occurrences of $\bot$'s. It is clear that such a word u exists if and only if the length of word x is divisible by m. #

Recall that the logic $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ assumes a non-unary alphabet Σ. In fact, the logic is equally expressive to difference arithmetic when Σ is unary. Interestingly, restricting the alphabet to binary does not weaken the expressive power of $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$, as is shown by the following lemma.

Proposition 2.2.1. *$\mathfrak{U}_\Sigma$ is first-order interpretable in $\mathfrak{U}_{\{0,1\}}$.*

Proof. The idea of the interpretation is to encode Σ in binary on blocks of uniform length $\lceil \lg|\Sigma| \rceil$. More precisely, let $m := \lceil \lg|\Sigma| \rceil$, $f : \Sigma \to \{0,1\}^m$ be an arbitrary injective function, and $\eta : \Sigma^* \to (\{0,1\}^m)^*$ be the word homomorphism $a_1 \cdots a_n \mapsto f(a_1)\cdots f(a_n)$. The first-order interpretation $\mathscr{I} := (\Delta, \Phi_\preceq, \Phi_{eqL}, \{\Phi_{\prec_a}\}_{a\in\Sigma})$ can be given such that $\Delta(x)$ is defined as in (2.2), $\Phi_\preceq(x,y) := x \preceq y$, $\Phi_{eqL}(x,y) := eqL(x,y)$, and $\Phi_{\prec_a}(x,y) := \Phi_\preceq(x,y) \wedge \exists n \in \mathbb{N}.\, n = |x| \wedge n + m = |y| \wedge \bigwedge_{i=1}^m y[n+i] = f(a)[i]$, that is, the first $|y| - m$ bits of y constitute the word x, and the last m bits of y constitute

the bit-vector $f(a)$. It is easy to see that $\mathscr{I}$ does provide a first-order interpretation of $\mathfrak{U}_\Sigma$ in $\mathfrak{U}_{\{0,1\}}$ under the encoding η. #

The following result establishes the cornerstone of the development in this thesis.

Theorem 2.2.2. *Fix a non-unary alphabet Σ. Given a first-order formula $\phi(x_1, \ldots, x_n)$ in $FO_{REG}(\Sigma)$, $\phi^{\mathfrak{U}_\Sigma}$ is a regular relation and one can effectively compute a finite automaton recognising $\langle \phi^{\mathfrak{U}_\Sigma} \rangle$. Conversely, given a regular relation $R \subseteq (\Sigma^*)^n$ and a finite automaton recognising $\langle R \rangle$, one can effectively compute a first-order formula $\phi(x_1, \ldots, x_n)$ in $FO_{REG}(\Sigma)$ such that $\phi^{\mathfrak{U}_\Sigma} = R$.*

An analogous version of this theorem was first proved in [EES69] and later became a consequence or special case of several results, e.g., [BLSS03; BG04; CL07]. We shall sketch a constructive proof of this theorem in some details, as a similar construction will be employed in Chapter 5 to compute finite automata for regular abstractions.

Proof Sketch. (Formula to automaton) For each first-order formula ϕ defined in $\mathfrak{U}_\Sigma$, we give a regular language $\lambda(\phi)$ satisfying $\lambda(\phi) = \langle \phi^{\mathfrak{U}_\Sigma} \rangle$. That is, if ϕ has n free variables, then $\lambda(\phi)$ is a regular language over alphabet $\Sigma_\#^n$ such that $w_1 \otimes \cdots \otimes w_n \in \lambda(\phi)$ if and only if $\mathfrak{U}_\Sigma \models \phi(w_1, \ldots, w_n)$. We define $\lambda(\phi)$ by induction on the structure of ϕ, which allows an NFA recognising $\lambda(\phi)$ to be built in a bottom-up manner.

When $\phi := \phi(x_1, x_2)$ is an atomic relation, we may specify $\lambda(\phi)$ as

- $\lambda(x_1 \preceq x_2) := L_1 \cdot L_2 \cdot L_3$, and $\lambda(x_2 \preceq x_1) := L_1 \cdot L_2' \cdot L_3$.
- $\lambda(x_1 = x_2) := \lambda(x_2 = x_1) := L_1 \cdot L_3$.
- $\lambda(eqL(x_1, x_2)) := \lambda(eqL(x_2, x_1)) := (\Sigma \times \Sigma)^* \cdot L_3$.
- $\lambda(x_1 \prec_a x_2) := L_1 \cdot (\#, a)$, and $\lambda(x_2 \prec_a x_1) := L_1 \cdot (a, \#)$.

Here we have $L_1 := \{(a,a) : a \in \Sigma\}^*$, $L_2 := \{(\#,a) : a \in \Sigma\}^*$, $L_2' := \{(a,\#) : a \in \Sigma\}^*$, and $L_3 := (\#,\#)^*$. Therefore, when ϕ is atomic, an NFA over alphabet $\Sigma_\#^2$ can be computed for $\lambda(\phi)$.

We encode logical connectives over formulae via automata operations over the NFAs that recognise the encodings of these formulae. Note that, when two formulae have different sets of free variables, we need to lift their encodings to a common alphabet before we encode logical connectives between the formulae. For example, suppose we have formulae $\phi(x_{j_1}, \ldots, x_{j_m})$ and $\psi(x_{r_1}, \ldots, x_{k_r})$, and assume without loss of generality that $\{j_1, \ldots, j_m\} \cup \{k_1, \ldots, k_r\} = \{1, \ldots, n\}$. Let $\mathscr{A}_\phi := (Q, \Sigma_\#^m, \delta, q_0, F)$ be an NFA recognising $\lambda(\phi(x_{j_1}, \ldots, x_{j_m}))$. Then we extend the alphabet of $\mathscr{A}_\phi$ to $\Sigma_\#^n$ by computing an NFA $\mathscr{A}'_\phi := (Q, \Sigma_\#^n, \delta', q_0, F)$ with

$$\delta' := \{(q, (a_1, \ldots, a_n), q') : (q, (a_{j_1}, \ldots, a_{j_m}), q') \in \delta, \text{ and } a_i \in \Sigma_\# \text{ for } i \in \{1, \ldots, n\}\}.$$

By the same token, we extend the alphabet of an NFA $\mathscr{A}_\psi$ recognising $\lambda(\phi(x_{r_1}, \ldots, x_{k_r}))$ to obtain an NFA $\mathscr{A}'_\psi$ over alphabet $\Sigma_\#^n$. The encoding $\lambda(\phi(x_{j_1}, \ldots, x_{j_m}) \vee \psi(x_{r_1}, \ldots, x_{k_r}))$ of the conjunction of ϕ and ψ is then defined as $(A'_\phi \cup A'_\psi) \cap A_n$, where $A_n := \Sigma^* \otimes \cdots \otimes \Sigma^*$ denotes the n-convolution of Σ^*. Similarly, the encoding $\lambda(\phi(x_{j_1}, \ldots, x_{j_m}) \wedge \psi(x_{r_1}, \ldots, x_{k_r}))$ of the disjunction is defined as $(A'_\phi \cap A'_\psi) \cap A_n$.

To encode quantification, it is sufficient to consider the existential quantifier since $\forall x.\, \phi \equiv \neg\exists x.\, \neg\phi$. The encoding $\lambda(\exists x_i.\, \phi(x_1, \ldots, x_n))$ can be obtained by erasing the i-th track of the words in $\lambda(\phi(x_1, \ldots, x_n))$. More precisely, suppose that $\lambda(\phi(x_1, \ldots, x_n))$ is recognised by an NFA $\mathscr{A} := (Q, \Sigma_\#^n, \delta, q_0, F)$. Then we can compute an NFA $\mathscr{A}' := (Q, \Sigma_\#^{n-1}, \delta', q_0, F')$ for $\lambda(\exists x_1.\, \phi(x_1, \ldots, x_n))$ by modifying the definition of $\mathscr{A}$ as follows. First, we compute the transition relation δ' by iteratively replacing each transition $(q, (a_1, \ldots, a_n), q')$ in δ with $(q, (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n), q')$ if the latter transition is not already present. The resulting automaton, however, might accept words with redundant tailing blanks. For example, when $\Sigma = \{a, b\}$, $n = 2$, and $i = 1$, it might accept $a\#\#$ instead of a when $bbb \otimes a \in A$. To remove these redundant symbols, we saturate the final states F' by including all states that are backward reachable from F through $\#^*$. The resulting automaton accepts $w_1 \otimes \cdots \otimes w_{i-1} \otimes w_{i+1} \otimes \cdots \otimes w_n$ if and only if there exists $w \in \Sigma^*$ such that $\mathscr{A}$ accepts $w_1 \otimes \cdots \otimes w_{i-1} \otimes w \otimes w_{i+1} \otimes \cdots \otimes w_n$.

With the above definitions, it is now straightforward to show that $\lambda(\phi)$ is effectively

regular by structural induction on ϕ.

(Automaton to formula) Let $\mathscr{A} := (Q, \Sigma_{\#}^{n}, \delta, q_0, F)$ be a finite automaton recognising $\langle R \rangle$, and suppose $Q = \{q_0, \ldots, q_m\}$. Let $\perp$ and $\top$ be two distinct letters in Σ. We shall give a first-order formula $\phi(x_1, \ldots, x_n)$ in $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ such that $\mathfrak{U}_\Sigma \models \phi(w_1, \ldots, w_n)$ iff $\mathscr{A}$ accepts $w_1 \otimes \cdots \otimes w_n$. Intuitively, we use word variables $u_0, \ldots, u_m$ over $\{\perp, \top\}^*$ to characterise the accepting runs of $\mathscr{A}$, such that a run $\rho_1, \ldots, \rho_r$ is accepting iff $u_j[i] = \top \Leftrightarrow \rho_i = q_j$ for $i \in \{1, \ldots, r\}$. To this end, we define $\phi(x_1, \ldots, x_n) := \exists u_0 \ldots \exists u_m.\, \psi(u_0, \ldots, u_m, x_1, \ldots, x_n)$ where ψ is the conjunction of the formulae

$$
\begin{aligned}
\psi_1 &:= \forall k.\, f(k, \bar{x}) \Rightarrow \bigwedge\nolimits_{i \neq j} (u_i[k] = \perp \vee u_j[k] = \perp), \\
\psi_2 &:= \forall k.\, f(k, \bar{x}) \Rightarrow \bigvee\nolimits_{(q_i, \bar{a}, q_j) \in \delta} (u_i[k] = \top \wedge u_j[k+1] = \top \wedge g(k, \bar{a}, \bar{x})), \text{ and} \\
\psi_3 &:= u_0[1] = \top \wedge \exists t.\, h(t, \bar{x}) \wedge \bigvee\nolimits_{q_i \in F} u_i[t+1] = \top,
\end{aligned}
$$

with macros $f(k, \bar{x}) := 1 \leq k \wedge \bigvee_i (k \leq |x_i|)$, $g(k, \bar{a}, \bar{x}) := \bigwedge_{a_i \neq \#} (x_i[k] = a_i) \wedge \bigwedge_{a_i = \#} (|x_i| < k)$, and $h(t, \bar{x}) := \bigvee_i (t = |x_i|) \wedge \bigwedge_i (t \leq |x_i|)$. Intuitively, ψ_1 asserts that $\phi(w_1, \ldots, w_n)$ captures at most one run for an input word $w_1 \otimes \cdots \otimes w_n$ of the automaton $\mathscr{A}$; ψ_2 asserts that the run proceeds correctly as the automaton reads the input word; ψ_3 asserts that the run starts from the initial state and ends at some final state. The three formulae ψ_1, ψ_2, and ψ_1 together assure that a word $w_1 \otimes \cdots \otimes w_n$ is accepted by $\mathscr{A}$ if and only if $\mathfrak{U}_\Sigma \models \phi(w_1, \ldots, w_n)$. #

Theorem 2.2.2 has several important implications. One of them is that relations first-order definable by regular relations are also regular. We shall frequently exploit this fact in subsequent chapters without explicit reference. Theorem 2.2.2 also implies that the first-order theory of $\mathfrak{U}_\Sigma$ is decidable. Indeed, one can observe from the proof of the theorem that a first-order sentence ϕ is true in $\mathfrak{U}_\Sigma$ if and only if $\lambda(\phi) \neq \emptyset$. Therefore, deciding the theory amounts to checking language emptiness of finite automata, which is a decidable problem. Also note that $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ remains decidable (and in fact has the same expressive power) even if we extend the structure $\mathfrak{U}_\Sigma$ with arbitrary regular relations over Σ^*. As a result, we may freely use regular relations as syntactic sugar when we are defining an $\mathrm{FO}_{\mathrm{REG}}(\Sigma)$ formula.

By the transitivity of first-order interpretations, we can subsequently deduce from Theorem 2.2.2 the following fundamental results about automatic structures. These results, first pioneered by Büchi, Elgot, and Trahtenbrot in the early 1960's, demonstrate the deep correspondence between first-order logic and automata.

Theorem 2.2.3 (Fundamental Theorem of Automatic Structures [Hod83; KN94; BG04]).

- *A structure is automatic iff it is first-order interpretable in the structure* $\mathfrak{U}_{\{0,1\}}$.
- *The class of automatic structures is closed under first-order interpretations.*
- *The first-order theory of an automatic structure is decidable.*

Due to the first item of this theorem, $\mathfrak{U}_{\{0,1\}}$ is also called a *universal* [BLSS03] or a *complete* [BG00] automatic structure with respect to first-order interpretation.

As an immediate consequence of Theorem 2.2.3, the first-order theories introduced in Examples 2.2.2-2.2.6 are all decidable. These theories remain decidable after the underlying structures are extended with relations and functions that are automata presentable under the same encodings of the structures. For example, difference arithmetic is still decidable even if the logic is enriched with the congruence relations $\{\equiv_n\}_{n\geq 1}$ over the naturals. Similarly, Presburger arithmetic $\mathrm{FO}(\mathbb{N},+)$ remains decidable with the extension of bitwise shift operators $\{\cdot \gg n\}_{n\geq 1}$ and $\{\cdot \ll n\}_{n\geq 1}$, since the extended structure is still automata presentable under binary encoding of natural numbers.

2.2.4 MSO : monadic second-order logic

Monadic second-order logic (MSO) is an extension of first-order logic by allowing quantification over sets. More precisely, this logic allows building a formula by universally or existentially quantifying over *set variables*, which are variables interpreted over subsets of the universe of discourse. As a convention, we shall always write set variables in uppercase and first-order variables in lowercase. *Weak monadic second-order logic* (WMSO) further restricts MSO by only allowing quantification over *finite* sets. Hence, for example, $\langle \mathbb{N} \rangle \models \exists X.\, \forall y.\, X(y)$ holds in MSO but not in WMSO.

Example 2.2.13. Let L_m be the regular language comprised of the words $w \in \Sigma^*$ with length divisible by m. Then L_m is WMSO-definable with the following formula ϕ_m over the vocabulary σ_Σ (see Example 2.2.1), in the sense that $w \in L_m$ if and only if $\mathfrak{S}_w \models \phi_m$ holds in WMSO :

$$\begin{aligned}\exists X_0 \cdots \exists X_{m-1}. \forall x. \quad & \Big(\bigvee\nolimits_{i=0}^{m-1} X_i(x)\Big) \wedge \Big(\bigwedge\nolimits_{i=1}^{m-1} (X_i(x) \Rightarrow \exists y.\, succ(y,x) \wedge X_{i-1}(y))\Big) \\ & \wedge \Big(\bigwedge\nolimits_{i=0}^{m-2} (X_i(x) \Rightarrow \exists y.\, succ(x,y) \wedge X_{i+1}(y))\Big)\,,\end{aligned}$$

where $succ(x,y) := \forall z.\, x < z \Rightarrow (y < z \vee y = z)$ defines the successor relation over positions. When $\mathfrak{S}_w \models \phi_m$, the universe $\{1, \ldots, |w|\}$ of $\mathfrak{S}_w$ can be partitioned into m sets $X_0, \ldots, X_{m-1}$ of the same cardinality such that $t \in X_k$ if and only if $t \equiv_m k$. This condition is sufficient and necessary for m to divide the length of w.

In fact, it is a celebrated result of Büchi [Buc63] and Elgot [Elg61] that a language $L \subseteq \Sigma^*$ is regular if and only if it is WMSO-definable over the vocabulary σ_Σ, see also [Tho97]. #

Subset interpretations

A subset interpretation reduces the first-order theory of one structure to the MSO theory of another. Fix two finite vocabularies σ and τ such that τ is relational. A *subset interpretation* is given as a list of monadic second-order σ-formulae

$$\mathscr{I} := (\Delta(X), \{\Phi_r(\overline{X})\}_{r\in\tau})$$

such that the only free variables in each formula are set variables. Given a σ-structure $\mathfrak{S}$, we use $\mathscr{I}^{\mathfrak{S}}$ to denote the τ-structure $\langle \Delta^{\mathfrak{S}}, \{\Phi_r^{\mathfrak{S}}\}_{r\in\tau}\rangle$. That is, the universe of $\mathscr{I}^{\mathfrak{S}}$ consists of sets defined by $\Delta^{\mathfrak{S}}$, and each relation symbol $r \in \tau$ is interpreted as $\Phi_r^{\mathfrak{S}}$ in $\mathscr{I}^{\mathfrak{S}}$. A subset interpretation is furthermore called a *finite-set interpretation* if the set variables in the interpretation are only interpreted over finite sets.

As is the case with first-order interpretations, we can associate each subset (resp. finite-set) interpretation $\mathscr{I}$ with a syntactical transformation $\cdot^{\mathscr{I}}$ that maps a first-order formula $\phi(\overline{x})$ to a (resp. monadic) second-order formula $\phi^{\mathscr{I}}(\overline{X})$, which is recursively defined as

follows:

$$(r(\overline{x}))^{\mathscr{I}} := \Phi_r(\overline{X}) \qquad (\neg\phi)^{\mathscr{I}} := \neg\phi^{\mathscr{I}}$$
$$(x = y)^{\mathscr{I}} := X = Y \qquad (\phi \wedge \psi)^{\mathscr{I}} := \phi^{\mathscr{I}} \wedge \psi^{\mathscr{I}}$$
$$(\exists x.\, \phi)^{\mathscr{I}} := \exists x.\, \Delta(X) \wedge \phi^{\mathscr{I}} \qquad (\phi \vee \psi)^{\mathscr{I}} := \phi^{\mathscr{I}} \vee \psi^{\mathscr{I}}$$
$$(\forall x.\, \phi)^{\mathscr{I}} := \forall x.\, \Delta(X) \Rightarrow \phi^{\mathscr{I}}$$

A structure $\mathfrak{T}$ is *MSO-interpretable* (resp. *WMSO-interpretable*) in a structure $\mathfrak{S}$ if there is a subset (resp. finite-set) interpretation $\mathscr{I}$ defined in the vocabulary of $\mathfrak{S}$ such that $\mathscr{I}^{\mathfrak{S}}$ is isomorphic to $\mathfrak{T}$ or its relational variant. It is not difficult to see that, if $\mathscr{I}$ is subset or finite-set interpretation of $\mathfrak{T}$ in $\mathfrak{S}$, then for any first-order sentence ϕ defined over $\mathfrak{T}$, we have $\mathfrak{T} \models \phi$ if and only if $\mathfrak{S} \models \phi^{\mathscr{I}}$. Consequently, the interpretation $\mathscr{I}$ reduces the first-order theory of the interpreted structure $\mathfrak{T}$ to the MSO (resp. WMSO) theory of the host structure $\mathfrak{S}$.

Example 2.2.14. As an example, let us give a finite-set interpretation of $\langle \mathbb{N}, + \rangle$ in $\langle \mathbb{N}, s \rangle$, where $s : n \mapsto n + 1$ is the successor function over $\mathbb{N}$. The idea is to represent each natural number $n \in \mathbb{N}$ as a finite set $S_n \subset \mathbb{N}$ such that S_n contains k if and only if the k-th bit in the least-significant-bit-first binary encoding of n is 1. Formally, we define $\mathscr{I} := (\Delta(X), \Phi_+(X, Y, Z))$ with $\Delta(X) := \mathsf{true}$, and

$$\Phi_+(X, Y, Z) := \exists C.\, \forall n.\, (Z(n) \Leftrightarrow xor(n, X, Y, C)) \wedge (C(s(n)) \Leftrightarrow maj(n, X, Y, C)) \wedge \neg C(0).$$

Here, C represents the carry bits (i.e. $k \in C$ iff the addition leads to a carry at the k-th position), $xor(n, X, Y, C)$ expresses the truth value of $X(n) \oplus Y(n) \oplus C(n)$, and $maj(n, X, Y, C)$ expresses that at least two of $X(n)$, $Y(n)$, and $C(n)$ are true.

It is easy to see that the mapping $n \mapsto S_n$ forms an isomorphism between $\mathscr{I}^{\langle \mathbb{N}, s \rangle}$ and the relational variant of $\langle \mathbb{N}, + \rangle$ in the finite-set semantics. Therefore, the first-order theory of $\langle \mathbb{N}, + \rangle$ can be reduced to the WMSO theory of $\langle \mathbb{N}, s \rangle$, also known as the *weak monadic second-order theory with one successor* (WS1S). #

Example 2.2.15. The universal automatic structure $\mathfrak{U}_{\{0,1\}}$ is WMSO-interpretable in

$\langle \mathbb{N}, s \rangle$. A finite-set interpretation can be given as $\mathscr{I} := (\mathsf{true}, \Phi_{\preceq}, \Phi_{eqL}, \Phi_{\prec_0}, \Phi_{\prec_1})$ with

$$\begin{aligned}
\Phi_{\preceq}(X,Y) &:= \exists x. \exists y. m(x,X) \wedge m(y,Y) \wedge x \leq y \wedge \forall z. (z < x \wedge X(z)) \Leftrightarrow (z < x \wedge Y(z)) \\
\Phi_{eqL}(X,Y) &:= \exists x. \exists y. m(x,X) \wedge m(y,Y) \wedge x = y \\
\Phi_{\prec_0}(X,Y) &:= \exists x. m(x,X) \wedge m(s(x),Y) \wedge \neg Y(x) \wedge \Phi_{\preceq}(X,Y) \\
\Phi_{\prec_1}(X,Y) &:= \exists x. m(x,X) \wedge m(s(x),Y) \wedge Y(x) \wedge \Phi_{\preceq}(X,Y).
\end{aligned}$$

Here, $m(x,X) := x \in X \wedge \forall y. X(y) \Rightarrow y \leq x$ specifies that x is the maximal element in X, $x \leq y$ is an abbreviation of $\neg(y < x)$, and $x < y := \exists Z. Z(x) \wedge \neg Z(y) \wedge \forall z. Z(z) \Rightarrow z = 0 \vee (\exists i. Z(i) \wedge z = s(i))$ with Z meant to be interpreted as $\{z \in \mathbb{N} : z < y\}$. Note that $\mathfrak{U}_{\{0,1\}}$ and $\mathscr{I}^{\langle \mathbb{N}, s\rangle}$ are isomorphic in the finite-set semantics via the mapping

$$w \in \{0,1\}^* \mapsto \{n \in \mathbb{N} : w[n+1] = 1\} \cup \{|w|\}.$$

Therefore $\mathfrak{U}_{\{0,1\}}$ is WMSO-interpretable in $\langle \mathbb{N}, s \rangle$, and hence the first-order theory of $\mathfrak{U}_{\{0,1\}}$ is syntactically reducible to WS1S. #

The WS1S logic is decidable, see e.g., [Tho97], with highly optimised model checkers such as Mona [KM01; KMS02] and Gaston [FHJ+17]. These model checkers can be leveraged to reason about formulae in $\mathrm{FO}_{\mathrm{REG}}$ by transforming these formulae to WS1S.

2.3 Systems and properties

2.3.1 Kripke structures

A *Kripke structure* is a structure $\mathfrak{S} := \langle S, I, T \rangle$ where S is a countable set of configurations, $I \subseteq S$ defines the set of *initial configuration*, and $T \subseteq S \times S$ defines the *transition relation*. We call $T^*(I)$ the *reachability set* of $\mathfrak{S}$. A configuration $s \in S$ *reachable* if $s \in T^*(I)$. A configuration $s \in S$ is *terminal* if $\forall s' \in S. (s,s') \notin T$. We say that $\mathfrak{S}$ is *non-blocking* if no terminal configuration is reachable from I. We say that $\mathfrak{S}$ is *weakly finite* if $|T^*(\{s\})| < \infty$ for all $s \in S$. A finite or infinite sequence of configurations $\pi := s_0\, s_1\, s_2 \ldots$ is called a *path* if $(s_i, s_{i+1}) \in T$ for $0 \leq i < |\pi|$, where $|\pi|$ denotes length

of π. A path is *maximal* if either the path is infinite, or the last configuration of the path is terminal. A path is called an *execution run* if it starts from an initial configuration.

2.3.2 Safety and liveness

Safety and liveness are probably the most relevant properties in model checking. We shall recall their informal definitions in this section, and postpone more concrete expositions to subsequent sections. We refer the interested reader to the classic texts [MP12; BBF+13] for a thorough treatment of these properties.

Safety properties

A safety property is of the form that "bad things never happen". In practice, engineering methods such as testing, simulation, and monitoring are often exploited to detect violations of safety properties. There methods, nevertheless, are generally not capable of concluding the absence of violations. In model checking, the system of interest is abstracted to a mathematical model such as a Kripke structure, and the desired safety property is described with a formal specification. The specification is then verified against the model to establish safety of the real system. In this thesis, we are especially interested in a class of safety properties called *invariance*. Verifying an invariance property for a Kripke structure often amounts to checking that a designated set of *bad configurations* is not reachable through any execution run of the system. Since a configuration is reachable if and only if it can be reached within a finite number of steps, model checking safety properties requires reasoning about only finite execution runs.

As an illustrating example, consider a simple token passing protocol for n concurrent processes $1, \ldots, n$ on a ring. The protocol begins with some process holding a token. At each step, the system arbitrarily schedules a process for execution. If the scheduled process holds a token, it will pass the token to the right; otherwise the process will simply stay idle. For this protocol, "there is exactly one token in the system" is an invariance property. This property is violated precisely when there is a finite execution

run taking the system to a bad configuration, i.e., a configuration where the system has zero or multiple tokens. Therefore, we may prove the property by showing that all finite runs of the system does not visit a bad configuration.

To assure that the behaviour of a system is feasible, it is often not enough to consider safety. Safety properties are passive, in the sense that a system that does nothing satisfies virtually every non-trivial safety property. For example, a token ring system that never schedules a token-holding process is trivially safe. This characteristic of safety is analogous to the concept of *partial correctness* in program verification. A program is partially correct as long as it never produces a wrong answer. However, a partially correct program may well produce nothing at all. A certain kind of eventuality assertions in addition to partial correctness are necessary to specify that a program does compute a correct answer, in which case the *total correctness* of the program can be established. These eventuality assertions correspond to the so-called liveness properties, as we shall introduce below.

Liveness properties

A liveness property is of the form "good things eventually happen". The most well-known liveness property is arguably the termination property. Nonetheless, the notion of liveness is also important for systems that are not supposed to terminate, but to run indefinitely and interact with their environment. For a mail server, "every email sent is eventually delivered" is a liveness property. For an operating system, "every job launched is eventually executed" is a liveness property. Such systems are known as *reactive systems* [MP12], and can typically be modelled as a Kripke structure or its variants. In this thesis, we are especially interested in a class of liveness properties call *eventuality*. Verifying an eventuality property for a Kripke structure often amounts to checking that every maximal execution run of the system eventually visits a designated set of *accepting configurations*. Continuing with the token passing example, "process n will eventually hold the token" is an eventuality property, where the accepting configurations are precisely those with process n holding the token. This property can

therefore be proved by showing the absence of a maximal execution run that never visits an accepting configuration.

For reactive systems and concurrent process systems, a liveness property is often only meaningful under some *fairness requirement*. A fairness requirement specifies the infinite behaviour of feasible computations. In the token ring example, the liveness property "process n will eventually hold the token" can be violated in an "unfair" system where the process holding the token is never scheduled. It is therefore typical to assume *process fairness*, i.e., that each process in the system should be scheduled infinitely often, when we are considering liveness of a concurrent process system. Indeed, process fairness is sufficient to guarantee the aforementioned liveness property for the token passing protocol. Other typical fairness requirements for concurrent systems include *weak fairness* (e.g. if a process is holding a token, then it must eventually be scheduled) and *strong fairness* (e.g. if a process holds a token infinitely often, then it must be scheduled infinitely often). It is the system designer's responsibility to ensure that adequate fairness requirements are fulfilled by the underlying architecture.

LTL : Linear Temporal Logic

Safety, liveness, and fairness are special cases of linear-time properties. A linear-time property specifies the behaviour of a system in terms of the order in which a sequence of events are supposed to occur. This order can be precisely described with a logic called *Linear Temporal Logic* (LTL), which was first advocated by Pnueli [Pnu77] (see also [MP12]) as a formalism for verifying concurrent and reactive systems. Essentially, LTL extends propositional logic with a couple of temporal operators including $\Box$ ("always"), $\Diamond$ ("eventually"), $\bigcirc$ ("next"), and U ("until"). The syntax of LTL formulae is given by the grammar

$$\phi ::= \mathsf{true} \mid \mathsf{false} \mid P \mid \phi \vee \phi \mid \neg\phi \mid \Box\phi \mid \Diamond\phi \mid \bigcirc\phi \mid \phi\, U\, \phi,$$

where P is a set of configurations. Logical operators like $\wedge$ and $\Rightarrow$ can be defined with the operators $\neg$ and $\vee$ as expected. The semantics of an LTL formula is specified with

the paths of a Kripke structure. More precisely, let $\mathfrak{S} := \langle S, I, T \rangle$ be a Kripke structure and $\pi := s_0\, s_1\, s_2 \ldots$ be a path of $\mathfrak{S}$. Let π^i denote the suffix $s_i\, s_{i+1}\, s_{i+2} \ldots$ of π starting from the i-th configuration. Then the semantics of LTL over paths is given as

$$
\begin{aligned}
\mathfrak{S}, \pi \models P \quad & \text{iff} \quad s_0 \in P \text{ and } P \subseteq S \\
\mathfrak{S}, \pi \models \neg\phi \quad & \text{iff} \quad \mathfrak{S}, \pi \not\models \phi \\
\mathfrak{S}, \pi \models \phi \vee \psi \quad & \text{iff} \quad \mathfrak{S}, \pi \models \phi \text{ or } \mathfrak{S}, \pi \models \psi \\
\mathfrak{S}, \pi \models \bigcirc\phi \quad & \text{iff} \quad \mathfrak{S}, \pi^1 \models \phi \\
\mathfrak{S}, \pi \models \phi\, U\, \psi \quad & \text{iff} \quad \text{there is } j \geq 0 \text{ such that } \mathfrak{S}, \pi^j \models \psi \text{ and } \mathfrak{S}, \pi^i \models \phi \text{ for } 0 \leq i < j
\end{aligned}
$$

We shall use true and false as abbreviations of S and $\emptyset$, respectively. Moreover, we define the semantics of the temporal operators $\Diamond$ and $\Box$ with the logical equivalences

$$\Diamond\phi \equiv \mathsf{true}\, U\, \phi \quad \text{and} \quad \Box\phi \equiv \neg\Diamond\neg\phi.$$

Given a Kripke structure $\mathfrak{S}$ and an LTL formula ϕ, we call the set $\{\pi \mid \mathfrak{S}, \pi \models \phi\}$ of paths in $\mathfrak{S}$ the (linear-time) property defined by ϕ. We write $\mathfrak{S} \models \phi$ if every maximal execution run of $\mathfrak{S}$ is contained in the property defined by ϕ.

Most correctness properties we encounter in this thesis can be expressed in LTL.

- *Invariance.* An invariance property is an LTL property in form of $\Box P$. For example, the invariance property "the system never crashes" can be expressed as $\Box\neg crash$ such that *crash* corresponds to the set of configurations where the system crashes.
- *Eventuality.* A eventuality property is an LTL property in form of $\Diamond P$. For example, the program termination property, expressed by $\Diamond\, termination$ with *termination* standing for the set of terminal configurations, is an eventuality property. In contrast, the non-termination property $\neg\Diamond\, termination$, which is logically equivalent to $\Box\neg termination$, is an invariance property.
- *Response.* A response property says that one thing must eventually happen after the other. For example, the LTL formula $\Box(request \Rightarrow \Diamond\, delivered)$ expresses a response property that a request made at any time will lead to a request being

delivered thereafter. Here, *request* and *delivered* denote the sets of configurations where a request is made and delivered, respectively.

– *Fairness.* A fairness property is often used to specify that a service should be provided sufficiently often. For example, the *process fairness* of a concurrent system asserts that each process in the system should be scheduled infinitely often. If the system has n processes, then this assertion can be expressed in LTL as $\Box\Diamond\, scheduled_1 \wedge \cdots \wedge \Box\Diamond\, scheduled_n$. Here, $schedule_i$ is the set of configurations where the ith process is scheduled.

2.3.3 Probabilistic systems

Given a countable set S, a function f from S to non-negative rational numbers is a *distribution over* S if $f(s) \geq 0$ for all $s \in S$ and $\sum_{s\in S} f(s) \leq 1$. We use $\mathcal{D}_S$ to denote the set of distributions over S. A *probabilistic transition system* (*PTS*) is a two-sorted structure $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a\in\mathsf{ACT}}\rangle$ with sort symbols $\mathbb{S}$ and $\mathbb{P}$, where $S = \mathbb{S}^{\mathfrak{S}}$ is a countable set of configurations, $P = \mathbb{P}^{\mathfrak{S}}$ is the set of non-negative rational numbers, $+ : P \times P \to P$ is the usual addition operator over the rationals, ACT is a finite set of action symbols, and each $\delta_a : S \times S \to P$ defines a mapping $p_a : S \to \mathcal{D}_S$ from configurations to distributions such that $\delta_a(s, s') = p_a(s)(s')$. We call $\delta_a(s, s')$ the *transition probability* from s to s' through action a, and write $\delta_a(s, S')$ for $\sum_{s'\in S'} \delta_a(s, s')$. We say that $\mathfrak{S}$ is *bounded branching* if the number of configurations that are reachable from a configuration in one step is bounded by a universal constant.

A finite or infinite sequence $\pi := s_0 \to_{a_1} s_1 \to_{a_2} \cdots$ with $s_0, s_1, \ldots \in S$ and $a_1, a_2, \ldots \in \mathsf{ACT}$ is called a *path* of $\mathfrak{S}$ if $\delta_{a_{i+1}}(s_i, s_{i+1}) > 0$ for $0 \leq i < |\pi|$. A *trace* of an PTS is the sequence of actions triggered by a path. For example, the path π above induces a trace $\tau(\pi) := a_1\, a_2\, a_3 \ldots$. Notions of Kripke structures such as reachability sets, terminal configurations, etc., can be defined analogously for probabilistic transition systems.

2.3.4 Bisimulation equivalence

Let $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a \in \mathsf{ACT}} \rangle$ be a PTS. A *probabilistic bisimulation* (or simply a *bisimulation*) over $\mathfrak{S}$ is an equivalence relation $R \subseteq S \times S$ such that $(s,t) \in R$ implies

$$\forall a \in \mathsf{ACT}.\ \forall E \in S/R.\ \delta_a(s,E) = \delta_a(t,E). \tag{2.3}$$

Two configurations s, t of $\mathfrak{S}$ are *bisimilar* (written as $s \simeq t$) if there is a probabilistic bisimulation R over $\mathfrak{S}$ such that $(s,t) \in R$. Intuitively, bisimilar configurations emit the same amount of probability mass to the same equivalence class for any action. It is easy to see that the union of all bisimulation relations over $\mathfrak{S}$ is itself a bisimulation relation over $\mathfrak{S}$. This maximal bisimulation relation is called *bisimulation equivalence* or *bisimilarity* [SL95].

The notion of bisimilarity can be lifted to PTSs. Given PTSs $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a \in \mathsf{ACT}} \rangle$ and $\mathfrak{S}' := \langle S', P, +, \{\delta'_a\}_{a \in \mathsf{ACT}} \rangle$ with $S \cap S' = \emptyset$, we define the *disjoint union* of $\mathfrak{S}$ and $\mathfrak{S}'$ as the PTS $\langle S \uplus S', P, +, \{\delta''_a\}_{a \in \mathsf{ACT}} \rangle$ where

$$\delta''_a(s,t) := \begin{cases} \delta_a(s,t), & s \in S\,; \\ \delta'_a(s,t), & s \in S'. \end{cases}$$

A binary relation R over $S \uplus S'$ is called a probabilistic bisimulation between $\mathfrak{S}$ and $\mathfrak{S}'$ if R is a probabilistic bisimulation over the disjoint union of $\mathfrak{S}$ and $\mathfrak{S}'$.

PML : Probabilistic Modal Logic

Probabilistic Modal Logic (PML) is a logic proposed by Larsen and Skou [LS91] to characterise bisimulation equivalence for PTSs. The logic is specified by the syntax

$$\phi ::= \mathsf{true} \mid \mathsf{false} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \Delta_a \mid \langle a \rangle_r\, \phi. \tag{2.4}$$

Given a PTS $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a \in \mathsf{ACT}} \rangle$ and a configuration $s \in S$, the satisfaction relation $\mathfrak{S}, s \models \phi$ is defined as usual for true, false, $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$. $\mathfrak{S}, s \models \Delta_a$ means that $\delta_a(s,s') = 0$ for all $s' \in S$, i.e., $\mathfrak{S}$ cannot perform action a in configuration s. $\mathfrak{S}, s \models \langle a \rangle_r\, \phi$, where $0 \leq r \leq 1$ is a rational number, means that $\left(\sum_{s' :\, \mathfrak{S}, s' \models \phi} \delta_a(s,s')\right) \geq r$, i.e.,

$\mathfrak{S}$ can transit from s to some s' satisfying ϕ with probability at least r by performing the action a. The following result is well known.

Theorem 2.3.1 ([LS91; DEP98; DEP02])**.** *Two configurations of a PTS are bisimilar if and only if they satisfy the same PML formulae.*

We note that PML is subsumed by *Probabilistic Computation Tree Logic* (PCTL) [HJ94]. In fact, a PML formula can be translated into a PCTL formula of roughly the same size. It is well-known that PCTL can be checked against a finite *Markov Decision Processes* (MDP) in time polynomial in the sizes of the formula and the system, see e.g., [BK08]. Moreover, MDPs and PTSs have only superficial differences and can be formally converted to each other in a way that preserves bisimulation between configurations. For finite-state systems, this conversion can be done with a polynomial-time overhead. As a consequence, PML has a polynomial-time model checking algorithm for finite PTSs.

Chapter 3

Regular Model Checking

Regular model checking [KMM+97; WB98; BJNT00; JN00] is a generic and usually fully automated framework for analysing and verifying a broad spectrum of infinite-state and parameterised systems. In regular model checking, configurations are represented as words over a finite alphabet, sets of configurations are represented as finite automata, and transition relation is represented as finite transducers. Finite automata and transducers provides an effective symbolic representation of a class of structures called regular transition systems.

Definition 3.0.1 (Regular transition system)**.** A *regular transition system* is a Kripke structure $\langle \Sigma^*, I, T \rangle$ such that Σ is a finite set of alphabet, $I \subseteq \Sigma^*$ is a regular set, and $T \subseteq \Sigma^* \times \Sigma^*$ is a regular length-preserving relation.

The main motivation behind the notion of regular transition systems is to model parameterised systems. Particularly, the length-preservation assumption corresponds to the limitation on the amount of resources available to an instance of the parameterised system. For example, consider a concurrent system comprised of a parameterised number of finite-state processes, and let Σ denote the set of local states. Suppose that a configuration is encoded as a word $w \in \Sigma^*$ such that $w[i]$ corresponds to the local state of the i-th process. Then the configuration graph of the system can be modelled as a regular transition system when the transition relation is regular over $\Sigma^* \times \Sigma^*$.

This encoding, however, does not allow the number of processes to change at runtime. To circumvent this limitation, we may introduce a padding symbol # to serve as a placeholder. Thus, a configuration is a word over $\Sigma_{\#}^{*}$, and a transition can create a new process by overwriting the position occupied by the padding symbol. Even so, the maximal number of processes allowed to exist at runtime is still bounded *a priori* in a regular transition system, since the size of configurations on each execution run is fixed at the initial configuration.

Owing to the length-preservation assumption, regular transition systems are strictly less expressive than automatic Kripke structures. Particularly, regular transition systems cannot capture the (automatic) configuration graphs of pushdown automata and Turing machines, since these graphs permit the size of configurations to grow unboundedly along an execution run. Albeit the length-preservation limitation, most interesting properties, including safety and liveness properties, are still undecidable for regular transition systems. In fact, the halting problem of a Turing machine can be solved by checking the safety of a regular transition system: given a Turing machine M, one can design a length-preserving finite transducer to examine all possible computations of M, regarding a halting computation as a bad configuration. Thus, the resulting regular system is unsafe if and only if M halts.

While safety and liveness verification of regular transition systems is undecidable, the nice closure property of finite automata often allows for effective automata-theoretic verification techniques that are useful in practice. In the rest of this chapter, we shall briefly review some of these techniques that are relevant to the expositions in this thesis. We refer to the excellent work by Abdulla [Abd12], Nilsson [Nil05], Vojnar [Voj07], and Lin and Rümmer [LR20] for more general surveys of the development of regular model checking.

3.1 Techniques for safety verification

A safety property in regular model checking is defined as a quadruple (Σ, I, T, B) where Σ is a finite alphabet, $I, B \subseteq \Sigma^*$ are regular sets, and $T \subseteq \Sigma^* \times \Sigma^*$ is a regular length-preserving relation. The safety property holds if and only if the set B of the "unsafe" configurations are not reachable in the regular transition system $\mathfrak{S} := \langle \Sigma^*, I, T \rangle$, or equivalently, $\Sigma^* \setminus B$ is an invariance of $\mathfrak{S}$. The length-preserving assumption in the problem definition is not a critical restriction for safety analysis. Indeed, a system violating a safety property always has a finite counterexample execution run, and this run can be simulated by the length-preserving restriction of the original system with arbitrarily large (but finite) paddings. Experience shows that safety of many practical examples in parameterised systems, including most of the distributed algorithms studied in the literature (cf. [Lyn96; Fok18]), can be naturally captured and analysed in the length-preserving setting [Nil05; Abd12; LR16; LLMR17].

Since safety verification of regular model checking is undecidable, researchers have been focusing on studying efficient decision procedures for decidable subclasses, or developing effective semi-algorithms for more general undecidable systems. These two directions are not orthogonal: a decision procedure easily becomes a semi-algorithm after some assumptions are relaxed, and a semi-algorithm can become a decision procedure after certain sufficient conditions are satisfied. For decidable subclasses, specialised verification methods have been geared towards, among others, lossy channel systems [Fin94; AJ96; CFI96; ABB01], linear arithmetic systems [BW94; WB98; FL02; BFL04b; BB04; BFLP08], and pushdown systems [BEM97; FWW97; EHRS00]. In more general settings, safety analysis often involves computing or approximating reachability sets or transitive closures. Representative automated techniques for this purpose include acceleration, widening/extrapolation, abstraction refinement, and language inference.

Acceleration-based techniques

Checking a safety property (Σ, I, T, B) amounts to showing that $T^*(I) \cap B = \emptyset$, i.e., the set B of the bad configurations is not reachable from the set I of the initial configurations. Techniques for checking safety in regular model checking hence often involves computing T^* or $T^*(I)$, or an over-approximation thereof. To this end, Pnueli and Shahar [PS00] first proposed the *acceleration* method for exact fixed-point computation. An acceleration in this context essentially means to compute the effect of arbitrarily long sequences of transitions in one single step. Bouajjani et al. [BJNT00] developed semi-algorithms to construct a finite-state quotient transducer that is language equivalent to an infinite-state transducer recognising the reachability relation T^*. In [JN00; Nil05], a systematic framework was established to compute the quotient transducer based on forward and backward simulations. The authors also identified several syntactic constraints on the input system to guarantee the termination of their acceleration algorithms. The extrapolation-based acceleration was first considered by Bouajjani et al. [BJNT00], and later improved and elaborated by Legay and Wolper in T(O)RMC [Leg08; LW10]. The concept of their method is to sample DFAs recognising the intermediate reachability sets $T^{n_1}(I), T^{n_2}(I), \ldots$ in accordance with a user-provided increasing sequence $\{n_i\}_{i\geq 1}$. The method then attempts to identify repetitive increment between the sampled DFAs and approximate (i.e. extrapolate) the effect of the repetition by adding loops to a limiting automaton. If the extrapolation process ever converges, the obtained automaton will encode an over-approximation for $T^*(I)$. T(O)RMC also provides heuristics to check whether this approximation is exact. When the approximation is too rough for verification purpose, i.e., it results in a spurious counterexample, the sampling sequence has to be adjusted by the user to extrapolate the reachability set along a difference trajectory.

Other more specialised acceleration-based frameworks include FAST [BFLP03; BFLP08] and FASTER [BFL04b], which are geared towards safety verification of linear systems. A linear system is a transition system where configurations are integer vectors of fixed dimensions, and transitions are defined with linear functions in form of $v \mapsto M \cdot v + u$.

FAST and FASTER convert the input linear system into a counter automaton where transitions are labelled with Presburger formulae. The counter automaton can be expressed as a regular transition system due to the connections between Presburger formulae and finite automata. FAST and FASTER then compute the exact reachability set using acceleration. These tools inspect the stepwise reachable sets of configurations to identify repetitive structures in the system. Once such a structure, called a *cycle* in [BFL04a], is found, the tools compute an acceleration by deriving a DFA that captures the limiting effect of iterating the cycle. Leroux and Sutre [LS05] showed that acceleration is complete if the reachability set of the examined linear system is Presburger-definable. This holds precisely when the system can be translated into a *flat counter machine* [CJ98], where nested loops are forbidden in the control-flow structure, and transitions occurring inside loops are all labelled with difference bound constraints. Safety verification of such machines was shown to be NP-complete using acceleration techniques [BIK14].

Abstraction-based techniques

The acceleration-based methods typically compute the reachability set or transitive closure independently of the property to be verified. Alternatively, one can attempt to approximate the reachability set or transitive closure in a way such that the resulting approximation is just precisely enough to prove the target property. For example, *abstract regular model checking* (ARMC) [BHV04; BHRV06] is an automated approximation framework based on the *abstract-check-refine* paradigm [CGJ+00; Sai00; BMMR01]. The technique performs reachability analysis over the finite quotienting obtained by merging automaton states based on forward or backward language equivalence. The analysis always terminates and provides an over-approximation for the safety of the original system. Whenever a spurious counterexample is found, the quotienting can be automatically refined by splitting some of the merged states. In many circumstances, counting the number of components in a certain state is sufficient to verify a property. This is the concept underlying *counter abstraction*, a light-weight abstraction technique

over-approximating a configuration with a set of counters [GS92; EN98; Del00a; Del00b; PXZ02]. The technique is often devised such that reachability analysis of the resulting abstract models is decidable. For example, the method by German and Sistla [GS92] essentially generates a Petri net, while the method by Esparza et al. [EFM99] produces well-quasi-ordered systems. Pnueli et al. [PXZ02] furthermore showed that, for many practical distributed algorithms, the counter values larger than 1 can be abstracted away and the obtained *finite* abstract models are still precise enough to verify the desired property. Counter abstraction is typically designed for parameterised systems with symmetric finite-state processes, where processes are indistinguishable and the topology (e.g. the organisation of the processes) is irrelevant to the correctness of the system. *Environmental abstraction* [CTV06; CTV08] is a novel technique combining counter abstraction with predicate abstraction [GS97]. Instead of counting the processes in a given states, a counter stores the number of processes satisfying a given predicate. A predicate essentially expresses a relation between the local variables in a reference process and in the other processes. Using predicates, environmental abstraction is capable of verifying parameterised systems with numerical local variables and non-symmetric process identifiers. Like classic counter abstraction, however, the technique abstracts away the system topology and therefore is more suitable for dealing with unstructured systems.

Monotonic abstraction [ADHR07; ADR08; ADHR09] over-approximates transition relations for regular model checking to produce a well-structured transition system [ACJT96; FS01]. The abstract transitions are monotonic with respect to the subword relation over configurations in the sense that the subword relation forms a simulation relation in the abstract system. As a result, backward state exploration can operate on upwards-closed sets with a guarantee to converge in a finite number of steps. Monotonic abstraction is complete for a large class of well quasi-ordered systems including lossy channel systems [AJ96] and Petri nets [GRV06]. The abstraction scheme has been extended in [ACD+10] to support CEGAR via Craig interpolation, and in [ADR09] to handle concurrent processes with numerical local variables. In the latter extension, relations over

the numerical values are required to be expressible in the *gap-order constraints* [Rev93] to ensure termination of the reachability analysis. *View abstraction* [ADHR07; AHH13; AHH16] further parametrises monotonic abstraction on the size of basis allowed for computing a upwards-closed set. Particularly, the technique degenerates to monotonic abstraction when the view size is set to infinity. The strength of view abstraction arises from an empirical evidence that safety verification of parameterised systems often has a *small model property* [EN95], that is, it suffices to inspect small instances of the system to prove or disprove the (un)reachability of bad configurations. Indeed, the minimal view sizes sufficient to prove safety are surprisingly small for a large number of benchmarks studied in the literature of regular model checking [AHH13; AHH16].

Inference-based techniques

Unlike the acceleration- and the abstraction-based methods, which often employ fixed-point computation to verify a safety property, inference-based methods find a safety proof via some sort of intelligent search. An proof for a safety property (Σ, I, T, B) is a set $P \subseteq \Sigma^*$ satisfying the following *proof rules*: (i) $I \subseteq P$ (i.e. all initial configurations are contained in P), (ii) $P \cap B = \emptyset$ (i.e. P does not contain bad configurations), and (iii) $T(P) \subseteq P$ (i.e. P is *inductive*: applying T to any configuration in P does not take it outside P). These proof rules provide a convenient device to algorithmically guess and refine candidate proofs. For regular transition systems, this guess-and-refine procedure can be effectively carried out via language inference. For example, Habermehl and Vojnar [HV05] used a passive language inference algorithm by Trakhtenbrot [Tra77] to search for safety proofs. The learning process keeps sampling words of a fixed length $n \in \mathbb{N}$ until a finite automaton compatible with all seen samples can be concluded. If the words up to length n fails to produce an appropriate proof, the parameter n is increased and the learning procedure is repeated. Other work [Var06; VV07; CHLR17] adopted active inference algorithms such as L^* *learning* [Ang87] to infer proofs in a teacher-learner framework. Theses techniques do not guarantee to pinpoint a regular proof even when one exists; however, they are complete for systems with a

regular reachability set. Albeit simple in concept, the inference-based techniques were observed to have remarkable performance comparing to the acceleration-based and the abstraction-based methods [CHLR17].

Inference-based techniques have also been applied to verify more general regular systems such as automatic Kripke structures. Among these techniques, SAT-based learning [NJ13; Nei14; LNRS15; LR16] is a CEGAR-style technique to enumerate and refine regular proofs. The technique uses Boolean constraints to encode finite-state automata, and exploits an SAT solver to search for candidate proofs according to the proof rules. Each time a candidate proof is found, the technique either concludes that the desired property is proved, or obtains a spurious counterexample that can be further added to the Boolean constraints. By imposing an increasing bound on the size of candidate automata, the technique is able to pinpoint a minimal regular proof whenever one exists. Neider and Jansen [NJ13] crafted a similar learning framework that allows incomplete answers to membership queries [GLP06]. While the framework uses the same data structure as the L^* algorithm, it still relies on SAT-based learning to synthesise a regular proof. Particularly, termination of the learning process is not guaranteed by the Myhill-Nerode theorem. The Lever tool [VV06] also exploits the L^* algorithm to infer regular proofs for automatic systems. To make membership queries decidable, Lever annotates each configuration with distance information, i.e., a natural number d indicating that the configuration is reachable from some initial configuration by making at most d transitions. Lever then attempts to learn the set of annotated configurations, from which the reachability set can be obtained by projection. Beside learning configurations endowed with distances, Vardhan et al. [VSVA04a; VSVA04b] also proposed to learn paths annotated with transition names, with which concrete execution paths can be recovered and reachability of a specific configuration can be checked. The techniques of [VSVA04a; VSVA04b; VV06] will terminate only if there exists a regular proof with respect to the specific annotation. However, while a regular proof of the annotated system always induces a regular proof of the original system, the annotations often make a regular proof of the original system non-regular [NJ13].

3.2 Techniques for liveness verification

A liveness property in regular model checking can be specified as a quadruple (Σ, I, T, F) where I and F are regular sets and T is a length-preserving regular relation. The property holds if and only if every maximal execution run of the regular transition system $\langle \Sigma^*, I, T \rangle$ visits the set F of accepting configurations. For finite-state systems, liveness verification can be reduced to a repeated reachability problem for the product of the input system and the negated property [VW84; Var91; Wol00]. As pioneered by Pnueli [PS00], a similar approach is possible for weakly finite transition systems: in such systems, every path can reach only finitely many configurations, and hence an infinite path must repeatedly go through a certain configuration. Schuppan and Biere [BAS02; SB06] formalised this observation and established a syntactic reduction framework for several families of infinite-state systems, including regular transition systems. Abdulla et al. [AJN+12] furthermore elaborated a logical formalism called LTL(MSO), which is expressive enough to capture a broad class of temporal properties including liveness. This formalism is equipped with an automatic procedure to translate an LTL(MSO) specification to a regular transition system $\langle \Sigma^*, I, T \rangle$ with Büchi accepting conditions. Verifying a liveness property then amounts to checking that the induced regular system has no accepting lasso runs. In [AJN+12], a lasso run was detected by first computing the transitive closure T^* and the reachability set $Inv := T^*(I)$ using acceleration [BJNT00; Nil05], and then checking that $((Inv \cap F) \times (Inv \cap F)) \cap T^+$ contains no pairs of identical configurations, where F is the set of configurations satisfying the Büchi accepting conditions. Bouajjani et al. [BHV04; BHRV06] employed a similar reduction from liveness verification to lasso detection, but additionally integrated an abstraction refinement scheme to confine the fixed-point analysis in finite abstract domains. Overall, liveness checking methods via lasso detection often resort to calculations of the reachability set or transitive closure, which are computationally expensive and difficult to scale in practice [JN00; DLS01; BLW03; JS07].

Automated techniques not based on lasso detection have also been developed for

liveness verification. For example, Abdulla et al. [AJRS06] proved liveness using a backward reachability analysis from the "terminated" states. The analysis is able to produce a refinable under-approximation for the set of terminating states without calculating transitive closures. Other popular approaches take advantage of *ranking functions* [MP84; MP91], which have long been an effective device for analysing termination and other liveness properties. The analysis essentially involves equipping the transition relation with a well-founded ordering, thereby assuring that the evolution of system is proceeding in a desired direction. Fang et al. [FPPZ04a; FPPZ04b] described a deductive framework to infer ranking functions based on the proof rules of Floyd [Flo93]. The framework can be automated for several classes of parameterised systems by inspecting small instances of the system thanks to a small model property. Lin and Rümmer [LR16] checked liveness for concurrent systems by solving two-player reachability games. For reachability games expressible in regular model checking, they proposed a refinement-based approach to synthesise regular winning strategies. A winning strategy is essentially a ranking function guiding a player towards their goal under adversarial interference from the other player. This strategy can be naturally transformed to a proof for liveness under arbitrary schedulers. Lengál et al. [LLMR17] further extended the approach to handle reachability games under the *finitary fairness* [AH98] assumption, allowing them to verify liveness properties for concurrent systems under finitary-fair schedulers.

Liveness verification for more general regular systems, e.g., automatic Kripke structures, is significantly more difficult. In such systems, a liveness property can be violated without the presence of a lasso-shaped counterexample, making most of the aforementioned approaches no longer applicable. Bouajjani et al. [BLW05] proposed to detect a stronger evidence of violation than a bad lasso: finding a reachable accepting state s from which an accepting state simulating s is reachable.[1] To check this condition, the authors developed a semi-algorithm to compute the greatest simulation relation over the accepting states. Bouajjani et al. subsequently outlined a generic

[1] A state s *simulates* another state s' if s allows at least the same trace behaviour of s', see [Mil89].

framework to reason about linear-time properties for automatic systems based on their simulation technique. Their framework however depended on a critical assumption that a transducer encoding the reachability relation is available. Under a similar assumption, To and Libkin [TL08] proved that recurrent reachability is decidable for an automatic Kripke structure when the reachability relation is regular and computable. Their decidability results give rise to optimal polynomial-time algorithms for several subclasses of automatic structures such as the prefix-rewriting systems [Cau92].

For automatic systems where a regular presentation of the transitive closure is not available or computable, Vardhan et al. [VSVA05; VV06] and Neider [Nei10] presented liveness verification methods based on language inference. Vardhan et al. proposed to learn reachability set for configurations enriched with information (i, j), meaning that some path starting from the configuration can visit the accepting states for i times in j steps. They proved that the enriched reachability set has a unique fixed-point characterisation, and that the liveness property can be determined with this fixed point. Vardhan et al. further described an automata learning algorithm to infer an over-approximation for the fixed point. The learning process is guaranteed to terminate when the fixed point is regular. Neider [Nei10] studied two-player reachability games on automatic game graphs, and posited an L^*-based learning algorithm to infer winning regions, i.e., the set of configurations on which a player has a winning strategy. Inspired by the work of Vardhan et al. [Var06; VV06], Neider enriched the winning region with distance information to make the membership queries decidable. However, like Vardhan's approach, such enrichments can make a regular winning region non-regular and not amenable to automata learning [VSVA05; NJ13].

3.3 Extensions and generalisations

The research of regular model checking has largely focused on safety and liveness verification of systems encoded in finite automata and transducers. Nevertheless, variations and extensions of the classic setting have also been investigated in the

literature. These results can be roughly aligned in two directions. The first direction aims to extend the formalism for specifying the systems. For example, the symbolic representation has been generalised from word regular languages to, among others, ω-regular languages [BLW05; VSVA05; Leg08; LW10; Leg12], tree-regular languages [AJNd02; ALdR06; BHRV06; BHRV12], and subclasses of context-free languages [BH99; FP01]. The classes of regular transduction for expressing transition relations have also been generalised from regular and automatic relations to rational relations (cf. [Sak09; Ber13]) as well as the more theoretical setting of MSO-definable transformations [Cou94; EH01] over finite words [AČ10; AČ11], infinite words [AFT12; ADT13], and trees [AD17].

The second direction attempts to enrich the type of properties that can be verified. For example, tree-regular model checking has been applied to shape analysis of programs manipulating dynamic linked data structures such as lists and heaps [HHR+12; HLR+13; AHJ+16]. Omega-regular model checking has been applied to reason about properties of hybrid and dynamic systems with unbounded mixed variables taking integer or real values [BW02; BJW05; LW10]. More theoretical explorations in this aspect include the so-called regular real analysis, which studies analytic properties such as continuity and differentiability of real functions representable by ω-automata [CSV13; CLR15; BHK+20]. Regular model checking also plays a role in algorithmic game theory: the related techniques have been leveraged to synthesise winning strategies for reachability games [Nei10; LR16], concurrent safety games [NT16; MHL+20], and incomplete information games [BMP15]. Recently, regular model checking has found interesting applications in the verification of probabilistic systems. Particularly, it is exploited in several fully automatic methods for checking almost-sure termination [LLMR17] and probabilistic bisimulation [HLMR19] of Markov chains and Markov decision processes.

Chapter 4

Safety and Liveness Verification of Regular Transition Systems

In this chapter, we study the proof generation problem for safety and liveness verification in regular model checking. Given a Kripke structure $\mathfrak{S} := \langle S, I, T \rangle$, a safety property asserts that a set $B \subseteq S$ of bad configurations is not reachable from I, or equivalently, $T^*(I) \cap B = \emptyset$. This assertion can be established with an *inductive invariant*, which is a set of configurations $Inv \subseteq S$ satisfying the following *proof rules*:

- $I \subseteq Inv$, i.e., Inv subsumes the set of initial configurations;
- $Inv \subseteq T(Inv)$, i.e., Inv is closed under an application of the transition relation;
- $B \cap Inv = \emptyset$, i.e., Inv does not intersect a designated set B of unsafe configurations.

These rules are decidable when $\mathfrak{S}$ is a regular transition system and B and Inv are both regular. A natural tactic to prove safety in regular model checking is hence to exhibit a *regular* inductive invariant. In addition to safety properties, it is also possible to verify liveness properties using inductive invariants. Given a Kripke structure $\mathfrak{S} := \langle S, I, T \rangle$, a liveness property asserts that a set $F \subseteq S$ of accepting configurations is reached by every possible path from I. When $\mathfrak{S}$ is weakly finite (e.g. when $\mathfrak{S}$ is regular), any infinite path will eventually enter a cycle. A liveness property can therefore be formulated as a

safety property stating that no "bad" cycles are reachable, which can in turn be proved with a suitable inductive invariant. In the literature, a few semi-algorithms inspired by automata learning — some based on the passive learning algorithms [HV05; NJ13; AAC+14] and some others based on active learning algorithms [VSVA04b; NJ13] — have been proposed to synthesise a regular inductive invariant. Despite these semi-algorithms, not much attention has been paid to the applications of automata learning in regular model checking.

Our goal in this chapter is to investigate a basic question about regular model checking: can we effectively apply Angluin's L^* automata learning [Ang87] (or its variants such as [RS93; KV94]) to learn a regular inductive invariant for safety and liveness verification? Hitherto this question, perhaps surprisingly, has no satisfactory answer in the literature. A more careful consideration reveals one problem: a regular inductive invariant satisfying the proof rules might not be unique, and so strictly speaking we are not dealing with a well-defined learning problem. More precisely, we need to determine what the teacher should answer when the learner asks whether v is in the desired invariant, but v is unreachable from *Init*. Discarding v might not be a good idea, since this could force the learning algorithm to look for a *minimal* (in the sense of set inclusion) inductive invariant, which might not be regular. Similarly, we need to determine what the teacher should answer in the case when we found a pair (v, w) of configurations such that (i) v is in the candidate proof *Inv*, (ii) $w \notin Inv$, and (iii) there is a transition from v to w. In the ICE-learning framework [GLMN13; GLMN14; Nei14], such a pair (v, w) is called an *implication counterexample*. To satisfy the inductive invariant constraint, the teacher may respond that w should be added to *Inv*, or that v should be removed from *Inv*. Some work in the literature have proposed using a three-valued logic/automaton (with "don't know" as an answer) because of the teacher's incomplete information [GLP06; CFC+09].

Based on the above ideas, we propose a simple and practical solution to synthesise regular inductive invariants in regular model checking. Our solution exploits the classic L^* automata learning algorithm and its variants. For the problem mentioned in the

previous paragraph, we propose that a *strict teacher* be employed in L^* learning for regular inductive invariants in regular model checking. A strict teacher attempts to teach the learner the minimal inductive invariant (be it regular or not), but is satisfied when the candidate answer posed by the learner is an inductive invariant satisfying the proof rules without being minimal. (In this sense, perhaps a more appropriate term is a *strict but generous teacher*, who tries to let a student pass a final exam whenever possible.) For this reason, when the learner asks whether w is in the desired inductive invariant, the teacher will reply "no" if w is not reachable from *Init*. The same goes with an implication counterexample (v, w) such that the teacher will say that an unreachable v is not in the desired inductive invariant.

To exploit our invariant learning method in liveness analysis, we elaborate a syntactic liveness-to-safety transformation that is capable of reducing several linear-time properties to safety verification problems in regular model checking. Similar to [SB06], the transformation constructs a regular transition system that detects a reachable cycle by non-deterministically guessing the entry point of the cycle on an execution run. Our transformation furthermore supports reductions under *fairness conditions*, which are a class of LTL formulae to express scheduling assumptions for asynchronous concurrent and distributed systems. When provided with fairness conditions, the transformation will guarantee that only fair execution runs are taken into account in the induced safety verification problem. By applying the liveness-to-safety transformation, the target temporal proof goal is converted into a safety proof goal. It is then possible to leverage for liveness proofs the learning algorithms we have developed for safety verification.

We have implemented our learning-based approach in a prototype tool with an interface to the LibAlf library [BKK+10], which supports the L^* algorithm and its variants. We have taken numerous standard examples from regular model checking for safety and liveness verification. As the experimental results suggest, our method, albeit simple in concept, works extremely well in practice. For safety verification, we have compared the performance of our algorithm with well-known and established techniques such as SAT-based learning [NJ13; Nei14; LNRS15; LR16], which enu-

merates and refines candidate invariants using a SAT solver, abstract regular model checking (ARMC) [BHRV12], which performs abstraction-refinement using predicate and finite-length abstractions, and T(O)RMC [Leg08], which is based on extrapolation for transducer iteration. For liveness verification, we have compared with the tool SLRP [LR16]. The results show that our solution is comparable with (and, in fact, often outperform) the existing sophisticated semi-algorithms.

4.1 Related work

In the literature of regular model checking, many sophisticated semi-algorithms have been developed to analyse safety for parameterised systems. Among them, Vardhan et al. [VSVA04b; VV06] apply the L^* algorithm to learn regular inductive invariants for systems representable by automatic Kripke structures. Notably, membership query (i.e. checking point-to-point reachability) for L^* is undecidable in their setting. To resolve this issue, Vardhan et al. propose to learn an inductive invariant enriched with "distance" information. This enrichment, however, can make the resulting set not regular even if the set of reachable configurations is regular, in which case our method is guaranteed to terminate. We are unable to make a direct comparison with their approach, since neither their tool Lever [VV06] nor the models used in their experiments are available. A learning algorithm allowing incomplete information [GLP06] has been applied in [NJ13] for inferring inductive invariants of regular model checking. While the learning algorithm in [GLP06] uses the same data structure as the standard L^* algorithm, it is essentially a SAT-based learning algorithm. In particular, its termination is not guaranteed by the Myhill-Nerode theorem. Despite our results that SAT-based learning seems to be less efficient than L^* learning for synthesising regular inductive invariants, SAT-based learning is more general and applicable when verifying other properties, e.g., liveness [LR16], fair termination [LLMR17], and safety games [NT16].

Liveness-to-safety reduction has been demonstrated to be a powerful technique for

verifying liveness of finite-state systems [BAS02; SB04]. Generalisation of the reduction to parameterised systems was first pioneered by Pnueli [PS00] and later formalised by Schuppan and Biere [SB06]. Our formulation of the safety reduction is conceptually close to that of Schuppan and Biere's. However, Schuppan and Biere only described the reduction of the eventuality property $\Diamond P$ without fairness conditions. We consider more complex linear-time properties such as recurrence and response, and integrate the reduction method with an expressive family of fairness conditions via regular encodings. This integration with fairness has largely enhanced the applicability of our reduction framework to concurrent and distributed systems verification.

We remark that a similar algorithm for learning safety proofs has appeared in Vardhan's thesis [Var06, Section 6] from 2006. In his thesis, Vardhan proposes to make a membership query decidable by bounding the space of the transducers. Vardhan then applies the trick to verify safety of counter systems (i.e. parameterised concurrent systems without topologies). The research presented here is conducted independently. We investigate several aspects that are not considered in [Var06], including the incorporation of the learning method with liveness-to-safety transformation and fairness encoding, engineering heuristics such as using shortest counterexamples and caching, as well as experimental results on a variety of parameterised systems other than counter systems. Unfortunately, we fail to compare our implementation in details with Vardhan's tool, since the latter is not publicly available.

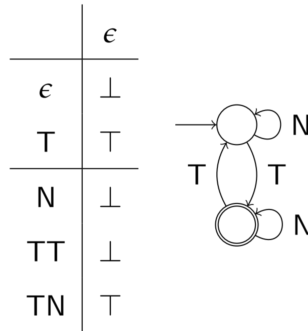
4.2 Safety verification

We first formulate the safety verification problem in the framework of regular model checking. For this purpose, fix a finite alphabet Σ throughout this section. Formally, we use a triple $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ to denote a safety property, where $\mathcal{I}$ is an automaton recognising the set of initial configurations, $\mathcal{T}$ is a transducer representing the transition relation, and $\mathcal{B}$ is an automaton recognising the set of bad configurations. Then verifying the property $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ amounts to showing that $T^*(I) \cap B = \emptyset$ for the regular transition

system $\langle \Sigma^*, I, T \rangle$. In regular model checking, a popular method for proving safety is to find a *regular proof*, i.e., a regular inductive invariant Inv satisfying the following three conditions: (i) $I \subseteq Inv$, i.e., all initial configurations are contained in Inv), (ii) $Inv \cap B = \emptyset$, i.e., Inv does not contain bad configurations), and (iii) $T(Inv) \subseteq Inv$, i.e., Inv is inductive). Because regular languages are effectively closed under Boolean operations and taking pre-/post-images with respect to finite transducers, an algorithm for verifying whether a given regular language is an inductive invariant can be obtained by using language inclusion algorithms for finite automata [ACH+10; BP13].

Example 4.2.1 (Herman's Protocol)**.** Herman's Protocol is a *self-stabilising* protocol for n processes arranged on a directed ring. The protocol ensures that if the processes collectively hold an *odd* number of tokens, then the system will almost surely converge to a configuration where precisely one process holds a token. The protocol works iteratively, and a random process is scheduled in each iteration. If the scheduled process holds a token, it will toss a coin to decide whether to keep the token, or to pass the token to the next process on the ring. If a process holds two tokens in the same iteration, it will discard both tokens.

Consider a safety property "the system always has at least one token". We devise a a regular model $(\mathscr{I}, \mathscr{T}, \mathscr{B})$ for this property as follows. Each process has two states: the symbol T denotes the state that the process has a token and N denotes the state that the process does not have a token. A system configuration is a finite word over alphabet $\{\mathsf{T}, \mathsf{N}\}$. For example, the word NTTNN denotes a system configuration with five processes, where only the second and the third processes are in the state with tokens. The set of initial configurations is $I = \mathsf{N}^*\mathsf{T}(\mathsf{N}^*\mathsf{T}\mathsf{N}^*\mathsf{T}\mathsf{N}^*)^*$, i.e., an odd number of processes has tokens. The set of bad configuration is $B = \mathsf{N}^*$, i.e., all tokens have disappeared. Let $E = ((\mathsf{T}, \mathsf{T}) + (\mathsf{N}, \mathsf{N}))$ denote the regular relation that a process is idle. Then the transition relation T can be specified as the union of the following regular languages: i) E^* (*Idle*), ii) $E^*(\mathsf{T}, \mathsf{N})(\mathsf{T}, \mathsf{N})E^* + (\mathsf{T}, \mathsf{N})E^*(\mathsf{T}, \mathsf{N})$ (*Discard both tokens*), and iii) $E^*(\mathsf{T}, \mathsf{N})(\mathsf{N}, \mathsf{T})E^*+ (\mathsf{N}, \mathsf{T})E^*(\mathsf{T}, \mathsf{N})$ (*Pass the token*). #



	ϵ
ϵ	$\bot$
T	$\top$
N	$\bot$
TT	$\bot$
TN	$\top$

Figure 4.1 Using learning to verify the safety property $(\mathscr{I}, \mathscr{T}, \mathscr{B})$ of Herman's Protocol. The table on the left is the content of the *observation table* used by the automata learning algorithm of Rivest and Shaphire [RS93] and the automaton on the right is the inferred candidate DFA.

Example 4.2.2 (RMC of the Herman's Protocol)**.** Consider the safety property of Herman's Protocol in Example 4.2.1. Initially, several membership queries will be posed to the teacher to produce the closed observation table on the left of Fig. 4.1. In this example, the teacher returns $\top$ only for words containing an odd number of the symbol T. The learner will then construct the candidate automaton $\mathscr{A}_h$ on the right of Fig. 4.1 and pose an equivalence query on $\mathscr{A}_h$. Observe that $\mathscr{A}_h$ can be used as an inductive invariant in a regular proof. It is easy to verify that $I = \mathsf{N}^*\mathsf{T}(\mathsf{N}^*\mathsf{T}\mathsf{N}^*\mathsf{T}\mathsf{N}^*)^* \subseteq A_h$ and $A_h \cap B = A_h \cap \mathsf{N}^* = \emptyset$. The condition $T(A_h) \subseteq A_h$ of a regular proof can be proved to be correct based on the following observation: the automaton $\mathscr{A}_h$ recognises exactly the set of all configurations with an odd number of tokens. When tokens are discarded in a transition, the total number of discarded tokens in all processes is always 2. The other two types of transitions will not change the total number of tokens in the system. It follows that taking a transition from any configurations in A_h will arrive a configuration with an odd number of tokens, which is still in A_h. #

The verification of Herman's Protocol finishes after the first iteration of learning and hence we cannot see how the learning algorithm uses a counterexample for refinement. Below we introduce a slightly more difficult problem.

Example 4.2.3 (RMC of the Israeli-Jalfon's Protocol)**.** Israeli-Jalfon's Protocol is a routing

protocol of n processes, numbered by $0, \ldots, n-1$. The processes are organised on an undirected ring and each process may hold a token at the beginning. In each iteration, if the scheduled process holds a token, it will toss a coin to decide to pass the token to the left or right, i.e., process i can pass its token to process $i-1 \bmod n$ or $i+1 \bmod n$. When two tokens are held by the same process, they will be merged. The safety property of interest is that every system configuration has at least one token.

The protocol and the corresponding safety property can be modelled as $(\mathscr{I}, \mathscr{T}, \mathscr{B})$ with the set of initial configurations $I = (\mathsf{T}+\mathsf{N})^*\mathsf{T}(\mathsf{T}+\mathsf{N})^*\mathsf{T}(\mathsf{T}+\mathsf{N})^*$, i.e., at least two processes have tokens, and the set of bad configurations $B = \mathsf{N}^*$, i.e., all tokens have disappeared. Again we use the regular language $E = ((\mathsf{T},\mathsf{T}) + (\mathsf{N},\mathsf{N}))$ to denote the relation that a process is idle, i.e, the process does not change its state. The transition relation T then can be specified as the union of the following regular languages:

$$E^*(\mathsf{T},\mathsf{N})((\mathsf{T},\mathsf{T})+(\mathsf{N},\mathsf{T}))E^*,\ ((\mathsf{T},\mathsf{T})+(\mathsf{N},\mathsf{T}))E^*(\mathsf{T},\mathsf{N}) \qquad (\textit{Pass the token right})$$
$$E^*((\mathsf{T},\mathsf{T})+(\mathsf{N},\mathsf{T}))(\mathsf{T},\mathsf{N})E^*,\ (\mathsf{T},\mathsf{N})E^*((\mathsf{T},\mathsf{T})+(\mathsf{N},\mathsf{T})) \qquad (\textit{Pass the token left})$$

When the automata learning algorithm of Rivest and Shaphire is applied to safety verification, we can obtain an inductive invariant with 4 states in 3 iterations. The first candidate automaton in Fig. 4.2(a) is incorrect because it does not include the initial configuration TT. By analysing the counterexample TT, the learning algorithm adds the suffix T to the set E in the observation table (see Section 4.4 for definition). The second candidate automaton in Fig. 4.2(b) is still incorrect because it contains an unreachable bad configuration NNN. The learning algorithm analyses the counterexample NNN and adds the suffix N to the set E. This time it obtains the candidate automaton in Fig. 4.2(c), which is a valid regular inductive invariant. #

4.3 Liveness verification

We proceed to formulate the liveness verification problem. Fix an alphabet Σ as before. In regular model checking, a liveness property can be specified as a triple $(\mathscr{I}, \mathscr{T}, \mathscr{F})$ where $\mathscr{I}$ is an automaton recognising the set of initial configurations, $\mathscr{T}$ is a transducer

recognising the transition relation, and $\mathcal{F}$ is an automaton recognising the set of accepting configurations. Define $\mathfrak{S} := \langle \Sigma^*, I, T \rangle$. Then the liveness property holds if and only if $\mathfrak{S} \models \Diamond F$, namely, every maximal execution run of $\mathfrak{S}$ eventually visits F. Note that we can modify the transducer $\mathcal{T}$ (see below) to attach a self-loop to each terminal configuration of $\mathfrak{S}$. Clearly, the modified system satisfies $\Diamond F$ if and only if the original system does. We may therefore assume without loss of generality that $\mathfrak{S}$ has no terminal configurations. As a consequence, a liveness property can be phrased as a safety property stating that the system $\mathfrak{S}$ does not have an infinite execution run comprised of configurations in $\overline{F} := \Sigma^* \setminus F$. Since $\mathfrak{S}$ is weakly finite, an infinite path is ultimately periodic. Hence, proving the safety property amounts to checking that any execution run proceeding inside $\overline{F}^*$ should never visit the same configuration twice. This fact provides a sound and complete reduction from liveness verification to safety verification.

More precisely, given a liveness property $(\mathcal{I}, \mathcal{T}, \mathcal{F})$, we define a safety property $(\mathcal{I}', \mathcal{T}', \mathcal{B}')$ over alphabet $\Sigma \times \Sigma_\perp$ such that

$$
\begin{aligned}
I' &= \{(u, \perp^{|u|}) : u \in I\} \\
T' &= \{((u,v),(u',v)) : (u,u') \in T,\ u \notin F,\ v \in \Sigma^* \uplus \perp^*\} \\
&\quad \cup \{((u,\perp^{|u|}),(u,u)) : (u,u') \notin T \text{ for all } u' \in \Sigma^*\} \qquad (4.1) \\
&\quad \cup \{((u,\perp^{|u|}),(u',u)) : (u,u') \in T,\ u \notin F\} \\
B' &= \{(u,u) : u \notin F\}.
\end{aligned}
$$

Here, we have treated I', T', B', etc., as length-preserving relations instead of languages to simplify the notations. Informally speaking, the new transition relation T' simulates the execution runs of $\mathfrak{S}$ inside the set of configurations $\overline{F}$, and loops whenever a terminal configuration of $\mathfrak{S}$ is visited. During the simulation, T' non-deterministically stores a copy of a configuration visited by $\mathfrak{S}$. From that point, it will compare the current configuration and the copy at each step, detecting whether the stored configuration is visited twice. It is clear that the regular transition system $\langle \Sigma^* \times \Sigma_\perp^*, I', T' \rangle$ has a path

$$(s_0, \perp^{|s_0|}) \ldots (s_n, \perp^{|s_n|})\, (\cdot, s_n)\, (\cdot, s_n) \ldots (s_n, s_n)$$

violating the safety property $(\mathscr{I}', \mathscr{T}', \mathscr{B}')$ if and only if $\langle \Sigma^*, I, T \rangle$ has either a maximal finite path $s_0 \dots s_n$ or a lasso-shaped infinite path $s_0 \dots s_n \dots s_n \dots$ violating the liveness property $(\mathscr{I}, \mathscr{T}, \mathscr{F})$. Therefore, the induced safety property $(\mathscr{I}', \mathscr{T}', \mathscr{B}')$ holds if and only if the original liveness property does. The fact that the safety property is effectively regular follows from Theorem 2.2.2 and the observation that I', T', and B' are first-order definable.

Termination Termination is a special case of liveness. In regular model checking, a termination property can be given as a tuple $(\mathscr{I}, \mathscr{T})$, which holds if and only if the regular transition system $\langle \Sigma^*, I, T \rangle$ has no infinite execution runs. Since the system is weakly finite, any infinite run is ultimately periodic. Therefore, a termination property can be phrased as a safety property stating that there exists no execution run visiting the same configuration twice. This fact leads us to the following definition of the safety property $(\mathscr{I}', \mathscr{T}', \mathscr{B}')$.

$$
\begin{aligned}
I' &= \{(u, \bot^{|u|}) : u \in I\} \\
T' &= \{((u, \bot^{|u|}), (u', u)) : (u, u') \in T\} \\
&\quad \cup \{((u, v), (u', v)) : (u, u') \in T,\ v \in \Sigma^{|u|} \cup \{\bot^{|u|}\}\} \\
B' &= \{(u, u) : u \in \Sigma^*\}.
\end{aligned}
$$

Again, here we treat I', T', B', etc., as length-preserving relations instead of languages. Note that, unlike the transformation for a liveness property, the new transition relation T' here does not need to loop at a terminal configuration. The correctness of the transformation follows similarly from the argument in the liveness case.

The reduction method we have presented in this section can be easily modified to verify a couple of closely related linear-time properties such as *recurrence* (good things should happen infinitely often), *persistence* (eventually good things will always be happening), and *response* (if a bad thing happen, then eventually a good thing will happen). These properties have similar characteristics to liveness, and are reducible to safety using similar augmentation techniques. We sketch some of these reductions in

Appendix A.1.

4.3.1 Regular encoding of fairness requirements

When we specify a liveness property for a concurrent process system, it is often necessary to add a fairness requirement for each process, e.g., that the process should be scheduled infinitely often. Since the system admits a finite but arbitrary number of processes, we generally need to express an infinite number of fairness requirements. In this section, we shall describe how to express such requirements in the framework of regular model checking. Our presentation strictly generalises the regular specification of fairness in [Nil05; AJN+12; LLMR17].

Definition 4.3.1 (Fairness requirements)**.** A countable set $\Phi := \{\phi_i : i \geq 1\}$ is a set of *(regular) fairness requirements* if there exist regular languages $A_{i,j} \subseteq \Sigma^*$ for $i \geq 1$ and $j \in \{1, \ldots, n\}$ such that each $\phi_i \in \Phi$ is a propositional formula over variables $\{\Box\Diamond\, A_{i,j} : j \in \{1, \ldots, n\}\}$. #

We remark that, by expressing each formula ϕ_i in negation normal form and noting that $\neg\Box\Diamond\, A \equiv \Diamond\Box\, \overline{A}$, the above definition of fairness requirements can be alternatively stated as follows: a countable set $\Phi := \{\phi_i : i \geq 1\}$ of propositional formulae is a set of fairness requirements if there exist a series of regular languages $A_{i,j} \subseteq \Sigma^*$ such that for $i \geq 1$ and $j \in \{1, \ldots, n\}$, each $\phi_i \in \Phi$ can be generated from variables $\Box\Diamond\, A_{i,j}$'s and $\Diamond\Box\, A_{i,j}$'s through conjunctions and disjunctions.

Definition 4.3.2 (Regular encoding of fairness requirements)**.** Let $\Phi := \{\phi_i : i \geq 1\}$ be a set of fairness requirements over variables $\{\Box\Diamond\, A_{i,j} : i \geq 1, j \in \{1, \ldots, n\}\}$. A tuple of automata $(\mathscr{A}_\Phi, \mathscr{B}_\Phi)$ is a *regular encoding* of Φ if

- $A_\Phi \subseteq (\{\mathtt{0}, \mathtt{1}\}^n)^*$ such that $a_1 \odot \cdots \odot a_n \in A_\Phi$ iff $\phi_i\, [a_1[i]/\Box\Diamond\, A_{i,1}] \cdots [a_n[i]/\Box\Diamond\, A_{i,n}]$ is true (by regarding $\mathtt{1}$ as true and $\mathtt{0}$ as false) in propositional logic for $i \in \{1, \ldots, |a_1|\}$.

– $B_\Phi \subseteq (\Sigma \times \{0,1\}^n)^*$ such that for each $u \in \Sigma^*$, there exist n words $a_1, \ldots, a_n \in \{0,1\}^{|u|}$ satisfying (i) $u \odot a_1 \odot \cdots \odot a_n \in B_\Phi$, and (ii) for $i \in \{1, \ldots, |u|\}$ and $j \in \{1, \ldots, n\}$, $a_j[i] = 1$ iff $u \in A_{i,j}$. #

Observe that A_Φ is solely determined by the propositional structures of the formulae in Φ, while B_Φ is solely determined by the definitions of the atomic languages $A_{i,j}$'s. Therefore, the information encoded in the regular languages A_Φ and B_Φ are in some sense orthogonal.

Definition 4.3.3 (Correctness under fairness requirements)**.** Let $\Phi := \{\phi_i : i \geq 1\}$ be a set of fairness requirements. Given a regular transition system $\mathfrak{S} := \langle \Sigma^*, I, T \rangle$, a path $\pi := s_0 \to s_1 \to \cdots$ of $\mathfrak{S}$ is *fair* if $\mathfrak{S}, \pi \models \phi_i$ for $i \in \{1, \ldots, |s_0|\}$. A temporal property P holds for the transition system $\mathfrak{S}$ under the fairness requirements Φ if every fair execution run of $\mathfrak{S}$ satisfies P. #

Example 4.3.1. Consider a toy concurrent process system where each process has three local states: waiting (W), requesting (R), and being served (S). Suppose that we have modelled the configurations of this system as words over alphabet $\Sigma := \{\mathsf{W}, \mathsf{R}, \mathsf{S}\}$, such that the ith letter of a word indicates the local state of process i. Consider the fairness requirement "if a process makes a request infinitely often, then it will be served infinitely often." For process i, this requirement can be expressed as $\phi_i := \neg\Box\Diamond A_{i,1} \vee \Box\Diamond A_{i,2}$, where $A_{i,1} := \Sigma^{i-1}\,\mathsf{R}\,\Sigma^*$ and $A_{i,2} := \Sigma^{i-1}\,\mathsf{S}\,\Sigma^*$. The regular encoding $(\mathscr{A}_\Phi, \mathscr{B}_\Phi)$ of the fairness requirements $\Phi := \{\phi_i : i \geq 1\}$ can be computed from the following regular expressions: $A_\Phi := ((0,0) + (0,1) + (1,1))^*$; $B_\Phi := ((\mathsf{R},1,0) + (\mathsf{S},0,1) + (\mathsf{W},0,0))^*$. #

The following result shows that a regular encoding of complex fairness requirements can be obtained by composing regular encodings of simple requirements.

Proposition 4.3.1. *Let $\Phi := \{\phi_i : i \geq 1\}$ and $\Psi := \{\psi_i : i \geq 1\}$ be two sets of fairness requirements. Given regular encodings of Φ and Ψ, we can effectively compute regular encodings of the fairness requirements $\neg\Phi := \{\neg\phi_i : i \geq 1\}$, $\Phi \vee \Psi := \{\phi_i \vee \psi_i : i \geq 1\}$, and $\Phi \wedge \Psi := \{\phi_i \wedge \psi_i : i \geq 1\}$.*

Proof Sketch. Let $(\mathscr{A}_\Phi, \mathscr{B}_\Phi)$ and $(\mathscr{A}_\Psi, \mathscr{B}_\Psi)$ be regular encodings of Φ and Ψ, respectively. Suppose that Φ is defined over variables $\{\Box\Diamond A_{i,j} : i \geq 1, j \in \{1, \ldots, p\}\}$. For $\neg\Phi$, we can compute a regular encoding by letting $\mathscr{B}_{\neg\Phi} := \mathscr{B}_\Phi$ and $\mathscr{A}_{\neg\Phi}$ be a finite automaton recognising the language $\{w \in (\{\mathtt{0}, \mathtt{1}\}^p)^* : \text{there exists } w' \in A_\Phi \text{ with } |w| = |w'|, \text{ such that } w[i] \neq w'[i] \text{ for all } i \in \{1, \ldots, |w|\}\}$. For $\Phi \vee \Psi$ and $\Phi \wedge \Psi$, we can compute a regular encoding by composing the encodings of Φ and Ψ. To this end, we need to lift the definitions of Φ and Ψ to the same set of variables. Without loss of generality, assume that Φ is defined over variables $\{\Box\Diamond A_{i,j} : i \geq 1, j \in \{j_1, \ldots, j_m\}\}$, Ψ is defined over variables $\{\Box\Diamond A_{i,k} : i \geq 1, k \in \{k_1, \ldots, k_r\}\}$, and $\{j_1, \ldots, j_m\} \cup \{k_1, \ldots, k_r\} = \{1, \ldots, n\}$. Now we compute regular encodings of $\Phi \wedge \Psi$ and $\Phi \vee \Psi$ as follows.

- We let $\mathscr{A}_{\Phi\wedge\Psi}$ be a finite automaton recognising $\{a_1 \odot \cdots \odot a_n : a_{j_1} \odot \cdots \odot a_{j_m} \in A_\Phi \text{ and } a_{k_1} \odot \cdots \odot a_{k_r} \in A_\Psi\}$.
- We let $\mathscr{A}_{\Phi\vee\Psi}$ be a finite automaton recognising $\{a_1 \odot \cdots \odot a_n : \text{there exist } \alpha \in A_\Phi \text{ and } \alpha' \in A_\Psi \text{ with } |\alpha| = |\alpha'|, \text{ such that for all } 1 \leq i \leq |a_1|, \text{ it holds that } \alpha[i] = (a_{j_1} \odot \cdots \odot a_{j_m})[i] \text{ or } \alpha'[i] = (a_{k_1} \odot \cdots \odot a_{k_r})[i]\}$.
- Both $\mathscr{B}_{\Phi\wedge\Psi}$ and $\mathscr{B}_{\Phi\vee\Psi}$ are encoded as a finite automaton recognising $\{u \odot a_1 \odot \cdots \odot a_n : u \odot a_{j_1} \odot \cdots \odot a_{j_m} \in B_\Phi \text{ and } u \odot a_{k_1} \odot \cdots \odot a_{k_r} \in B_\Psi\}$.

All finite automata mentioned above are computable by Theorem 2.2.2. #

Example 4.3.2. We illustrate the use of Proposition 4.3.1 by computing the regular encoding in Example 4.3.1 via composition. Consider first the atomic fairness requirements $\Phi_1 := \{\Box\Diamond A_{i,1} : i \geq 1\}$ and $\Phi_2 := \{\Box\Diamond A_{i,2} : i \geq 1\}$. The regular encodings of Φ_1 and Φ_2 satisfy $A_{\Phi_1} = A_{\Phi_2} = \mathtt{1}^*$, $B_{\Phi_1} = ((\mathsf{R}, \mathtt{1}) + (\mathsf{S}, \mathtt{0}) + (\mathsf{W}, \mathtt{0}))^*$, and $B_{\Phi_2} = ((\mathsf{R}, \mathtt{0}) + (\mathsf{S}, \mathtt{1}) + (\mathsf{W}, \mathtt{0}))^*$. Note that $A_{\neg\Phi_1} = \mathtt{0}^*$ and $B_{\neg\Phi_1} = B_{\Phi_1}$. By writing Φ as $\neg\Phi_1 \vee \Phi_2$, we can define $(\mathscr{A}_\Phi, \mathscr{B}_\Phi)$ such that

$$
\begin{aligned}
A_\Phi &= \{a_1 \odot a_2 : \exists \alpha_1 \in A_{\neg\Phi_1}, \alpha_2 \in A_{\Phi_2}. \forall i \in \{1, \ldots, |a_1|\}.\ a_1[i] = \alpha_1[i] \vee a_2[i] = \alpha_2[i]\} \\
&= ([\mathtt{0}, \cdot] + [\cdot, \mathtt{1}])^* = ((\mathtt{0}, \mathtt{0}) + (\mathtt{0}, \mathtt{1}) + (\mathtt{1}, \mathtt{1}))^*,
\end{aligned}
$$

and

$$\begin{aligned}
B_\Phi &= \{u \odot a_1 \odot a_2 : u \odot a_1 \in B_{\neg\Phi_1},\ u \odot a_2 \in B_{\Phi_2}\} \\
&= ([\mathsf{R},1,\cdot] + [\mathsf{S},0,\cdot] + [\mathsf{W},0,\cdot])^* \cap ([\mathsf{R},\cdot,0] + [\mathsf{S},\cdot,1] + [\mathsf{W},\cdot,0])^* \\
&= ((\mathsf{R},1,0) + (\mathsf{S},0,1) + (\mathsf{W},0,0))^*,
\end{aligned}$$

where $[x,\cdot] := (x,0) + (x,1)$, $[x,y,\cdot] := (x,y,0) + (x,y,1)$, etc., with $x, y \in \{0,1\}$, are abbreviations of the regular expressions of the lifted regular languages. #

4.3.2 Verification under fairness requirements

In this subsection, we introduce a regularity-preserving safety reduction method under fairness constraints. Fix a regular linear-time property P that supports safety reduction. Let Φ be a set of fairness requirements, and $(\mathscr{A}_\Phi, \mathscr{B}_\Phi)$ be a regular encoding of Φ. We shall provide a procedure to compute a safety property P'', such that P'' holds if and only if the original property P holds under the fairness requirements Φ.

Recall that an infinite counterexample path to the property P must be lasso-shaped. Also, the truth of a fairness condition is irrelevant to the finite prefixes of a path. Consequently, a lasso-shaped path is fair if and only if the cycle part of the path satisfies the fairness requirements, namely, the path contains a *fair cycle*. Therefore, assuming that all counterexamples to P are infinite, we can verify the property under fairness requirements by checking that P has no counterexample path containing a fair cycle. The reduction from the target property P to a safety property P'' hence consists of two steps. The first step reduces P to a safety property $P' := (\mathscr{I}', \mathscr{T}', \mathscr{B}')$ as discussed in Section 4.3. This reduction establishes a one-to-one correspondence between the counterexample paths to P and the counterexample paths to P'. The second step aims to remove the non-fair counterexamples from P': we compute another safety property $P'' := (\mathscr{I}'', \mathscr{T}'', \mathscr{B}'')$ based on P', making sure that bad configurations in B'' correspond precisely to the bad configurations in B' representing the fair cycles. In this way, a counterexample path to the safety property P'' will correspond to a fair counterexample path violating the original property P, and vice versa. The presence of the regular

encoding $(\mathscr{A}_\Phi, \mathscr{B}_\Phi)$ allows us to specify such a property P'' as a regular property.

Let us describe the second step of the reduction more formally. Suppose that after step 1, we obtain a safety property $(\mathscr{I}', \mathscr{T}', \mathscr{B}')$ over alphabet $\Sigma \times \Sigma_\perp$. Now we augment each configuration w in $(\Sigma \times \Sigma_\perp)^*$ with flags $a_1, \ldots, a_n \in \{0,1\}^{|w|}$ to detect the existence of fair cycles starting from and ending at w. More precisely, we define a new safety property $(\mathscr{I}'', \mathscr{T}'', \mathscr{B}'')$ over alphabet $(\Sigma \times \Sigma_\perp) \times \{0,1\}^n$ such that

$$
\begin{aligned}
I'' &= \{(w, (a_1, \ldots, a_n)) : w \in I', \, a_1, \ldots, a_n \in \{0,1\}^{|w|}\} \\
T'' &= T_1'' \cup T_2'' \\
B'' &= \{(w, (a_1, \ldots, a_n)) : w \in B', \, (a_1, \ldots, a_n) \in A_\Phi\}
\end{aligned}
$$

Here, T_1'' and T_2'' are defined as

$$
\begin{aligned}
T_1'' &= \{(((u, \perp^{|u|}), a), ((u', v), a')) : ((u, \perp^{|u|}), (u', v)) \in T', \, (u, a') \in B_\Phi\} \\
T_2'' &= \{(((u, v), a), ((u', v), a')) : ((u, v), (u', v)) \in T, \text{ and there exist } \alpha \text{ s.t. } (u, \alpha) \in B_\Phi, \\
&\qquad \text{and for all } i \in \{1, \ldots, |u|\} \text{ and } j \in \{1, \ldots, n\}, (a_j[i] = 1 \vee \alpha_j[i] = 1) \Leftrightarrow a_j'[i] = 1\}
\end{aligned}
$$

Again, here we have identified regular languages with length-preserving relations to simplify the notation. Intuitively, the transition relation T'' constantly updates the flags $a_1, \ldots, a_n$ to indicate whether the path so far has visited configurations from the corresponding sets. For example, T'' updates the flag $a_j[i]$ to 1 when the path encounters a configuration in $A_{i,j}$. Note that these flags are relevant only after the path has entered a cycle. Inside a cycle, $a_j[i] = 1$ means that the current cycle satisfies the atomic proposition $\Box\Diamond A_{i,j}$. By properly maintaining the flags $a_1, \ldots, a_n$, the transducer $\mathscr{T}$ enables the invariant that a cycle is fair if and only if $(a_1, \ldots, a_n) \in A_\Phi$ holds after a path have finished traversing that cycle. The fact that the safety property $(\mathscr{I}'', \mathscr{T}'', \mathscr{B}'')$ is effectively regular can be seen by noting that I'', T'', and B'' are definable in $\mathrm{FO}_{\mathrm{REG}}$ given the automata $\mathscr{I}'$, $\mathscr{T}'$, $\mathscr{B}'$, $\mathscr{A}_\Phi$, and $\mathscr{B}_\Phi$.

4.4 L-star automata learning

Suppose R is a regular *target* language whose definition is not directly accessible. *Automata learning* algorithms [Ang87; RS93; KV94; BHKL] automatically infer an automaton $\mathscr{A}$ recognising R. The setting of an online learning algorithm assumes a *teacher* who has access to R and can answer the following two queries: i) Membership query $Mem(w)$: is the word w a member of R, i.e., $w \in R$? ii) Equivalence query $Equ(\mathscr{A})$: is the language of $\mathscr{A}$ equal to R, i.e., $A = R$? If not, it returns a counterexample $w \in A \ominus R$. The learning algorithm will then construct an automaton $\mathscr{A}$ such that $A = R$ by interacting with the teacher. Such an algorithm works iteratively: In each iteration, it performs membership queries to get from the teacher information about R. Using the results of the queries, it proceeds by constructing a candidate automaton $\mathscr{A}_h$ and makes an equivalence query $Equ(\mathscr{A}_h)$. If $A_h = R$, the algorithm terminates with $\mathscr{A}_h$ as the resulting automaton. Otherwise, the teacher returns a word w distinguishing A_h from R. The learning algorithm uses w to refine the candidate automaton of the next iteration. In the last decade, automata learning algorithms have been frequently applied to solve formal verification and synthesis problems, see e.g. [HV05; GLP06; FCC+08; CFC+09; CCK+15; CHL+16].

Below we describe the details of the automata learning algorithm proposed by Rivest and Schapire [RS93] (RS), which is an improved version of the classic L^* learning algorithm by Angluin [Ang87]. The foundation of the learning algorithm is the Myhill-Nerode theorem, from which one can infer that the states of the minimal DFA recognizing R are isomorphic to the set of equivalence classes defined by the following relations: $x \equiv_R y$ iff $\forall z \in \Sigma^* : xz \in R \leftrightarrow yz \in R$. Informally, two strings x and y belong to the same state of the minimal DFA recognising R iff they cannot be distinguished by any suffix z. In other words, if one can find a suffix z' such that $xz' \in R$ and $yz' \notin R$ or vice versa, then x and y belong to different states of the minimal DFA.

The algorithm uses a data structure called *observation table* (S, E, T) to find the equivalence classes correspond to $\equiv_R$, where S is a set of strings denoting the set of

identified states, E is the set of suffixes to distinguish if two strings belong to the same state of the minimal DFA, and T is a mapping from $(S \cup (S \cdot \Sigma)) \cdot E$ to $\{\top, \bot\}$ such that $T(w) = \top \iff w \in R$. We use $row_E(x) = row_E(y)$ as a shorthand for $\forall z \in E : T(xz) = T(yz)$. That is, the strings x and y cannot be identified as two different states using only strings in the set E as the suffixes. Observe that $x \equiv_R y$ implies $row_E(x) = row_E(y)$ for all $E \subseteq \Sigma^*$. We say that an observation table is *closed* iff $\forall x \in S, a \in \Sigma : \exists y \in S : row_E(xa) = row_E(y)$. Informally, with a closed table, every state can find its successors wrt. all symbols in Σ. Initially, $S = E = \{\epsilon\}$, and $T(w) = Mem(w)$ for all $w \in \{\epsilon\} \cup \Sigma$.

Algorithm 1: The improved L^* algorithm by Rivest and Schapire

Input: A teacher answers $Mem(w)$ and $Equ(\mathcal{A})$ about a target regular language R and the initial observation table (S, E, T).

1 **repeat**
2   **while** (S, E, T) *is not closed* **do**
3   Find a pair $(x, a) \in S \times \Sigma$ such that $\forall y \in S : row_E(xa) \neq row_E(y)$. Extend S to $S \cup \{xa\}$ and update T using membership queries accordingly;
4   Build a candidate DFA $\mathcal{A}_h = (S, \Sigma, \delta, \epsilon, F)$, where $\delta = \{(s, a, s') \mid s, s' \in S \wedge row_E(sa) = row_E(s)\}$, the empty string ϵ is the initial state, and $F = \{s \mid T(s) = \top \wedge s \in S\}$;
5   **if** $Equ(\mathcal{A}_h) = (\mathsf{false}, w)$, *where* $w \in A_h \ominus R$ **then** Analyse w and add a suffix of w to E;
6 **until** $Equ(\mathcal{A}_h) = \mathsf{true}$;
7 **return** $\mathcal{A}_h$ is the minimal DFA for R;

The details of of the improved L^* algorithm by Rivest and Schapire can be found in Algorithm 1. Observe that, in the algorithm, two strings x, y with $x \equiv_R y$ will never be simultaneously contained in the set S. When the equivalence query $Equ(\mathcal{A})$ returns false together with a counterexample $w \in A \ominus R$, the algorithm will perform a binary search over w using membership queries to find a suffix e of w and extend E to $E \cup \{e\}$. The

suffix e has the property that $\exists x, y \in S, a \in \Sigma : row_E(xa) = row_E(y) \wedge row_{E\cup\{e\}}(xa) \neq row_{E\cup\{e\}}(y)$, that is, add e to E will identify at least one more state. The existence of such a suffix is guaranteed. We refer the readers to [RS93] for the proof.

Proposition 4.4.1 ([RS93])**.** *Algorithm 1 will find the minimal DFA $\mathcal{R}$ for R using at most n equivalence queries and $n(n + n|\Sigma|) + n \log m$ membership queries, where n is the number of state of $\mathcal{R}$ and m is the length of the longest counterexample returned from the teacher.*

Because each equivalence query with a false answer will increase the size (number of states) of the candidate DFA by at least one and the size of the candidate DFA is bounded by n according to the Myhill-Nerode theorem, the learning algorithm makes at most n equivalence queries. The number of membership queries required to fill in the entire observation table is bounded by $n(n + n|\Sigma|)$. Since a binary search is used to analyse the counterexample, and the number of counterexample from the teacher is bounded by n, the number of membership queries required is $O(n \log m)$.

For comparison, we introduce two other important variants of the L^* learning algorithm. The algorithm proposed by Kearns and Vazirani [KV94] (KV) uses a *classification tree* data structure to replace the observation table data structure of the classic L^* algorithm. The algorithm of Kearns and Vazirani has a similar query complexity to the one of Rivest and Schapire [RS93]; it uses at most n equivalence queries and $n^2(n|\Sigma| + m)$ membership queries. However, the worst case bound of the number of membership queries is very loose. It assumes the structure of the classification tree is linear, i.e., each node has at most one child, which happens very rarely in practice. In our experience, the algorithm of Kearns and Vazirani usually requires a few more equivalence queries, with a significant lower number of membership queries comparing to Rivest and Schapire when applied to verification problems.

The NL^* algorithm [BHKL] learns a non-deterministic finite automaton instead of a deterministic one. More concretely, it makes use of a canonical form of non-deterministic finite automaton, named *residual finite-state automaton (RFSA)* to express the target regular language. In some examples, RFSA can be exponentially more

succinct than DFA recognising the same languages. In the worst case, the NL^* algorithm uses $O(n^2)$ equivalence queries and $O(|\Sigma| \cdot mn^3)$ membership queries to infer a canonical RFSA of the target language.

4.5 Learning inductive invariant

We apply automata learning algorithms, including Angluin's L^* and its variants, to solve the safety verification problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. These learning algorithms require a teacher answering both equivalence and membership queries. Our strategy is to design a "strict teacher" targeting the minimal inductive invariant $T^*(I)$. For a membership query on a word w, the teacher checks if $w \in T^*(I)$, which is decidable under the assumption that $\mathcal{T}$ is length-preserving. For an equivalence query on a candidate automaton $\mathcal{A}_h$, the teacher analyses if $\mathcal{A}_h$ can be used as an inductive invariant in a proof of the problem $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. It performs one of the following actions depending on the result of the analysis (Fig. 4.3):

– Determine that $\mathcal{A}_h$ does not represent an inductive invariant, and return $false$ together with an explanation $w \in \Sigma^*$ to the learner.

– Conclude that $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = true$, and terminate the learning process with an inductive invariant $\mathcal{A}_h$ as the proof.

– Conclude that $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = false$, and terminate the learning with a word $w \in T^*(I) \cap B$ as an evidence.

Similar to the typical regular model checking approach, our learning-based technique tries to find a "regular proof", which amounts to finding an inductive invariant in the form of a regular language. Our approach is incomplete in general since it could happen that there only non-regular inductive invariants exist. Pathological cases where only non-regular inductive invariant exist do not, however, seem to occur frequently in practice, c.f., [BJNT00; HV05; Nil05; TL10; BHRV12; Lin12].

Answering membership queries. Answering a membership query on a word w, i.e.,

checking whether $w \in T^*(I)$, is straightforward: since $\mathcal{T}$ is length-preserving, we can construct an automaton recognising $Post^{|w|} := \{w' \mid w' \in T^*(I) \land |w'| = |w|\}$ and then check if $w \in Post^{|w|}$. In practice, $Post^{|w|}$ can be efficiently computed and represented using BDDs and symbolic model checking.

Answering equivalence queries. For an equivalence query on a candidate automaton $\mathcal{A}_h$, we need to check whether or not $\mathcal{A}_h$ is a proof of $(\mathcal{I}, \mathcal{T}, \mathcal{B})$. This is done by checking the three conditions in turn: (1) $I \subseteq A_h$, (2) $A_h \cap B = \emptyset$, and (3) $T(A_h) \subseteq A_h$. The check can be divided into four cases.

Case 1: The condition (1) is violated, i.e., there exists a word $w \in I \setminus A_h$. Since $I \subseteq T^*(I)$, the teacher can infer that $w \in T^*(I) \setminus A_h$ and return w as a *positive* counterexample to the learner. A counterexample is positive if it represents a word in the target language that was missing in the candidate language. The definition negative counterexamples is symmetric.

Case 2: The condition (2) is violated, i.e., there exists a word $w \in A_h \cap B$. The teacher checks if $w \in T^*(I)$ by constructing $Post^{|w|}$ and checking if $w \in Post^{|w|}$. If $w \notin T^*(I)$, the teacher obtains that $w \in A_h \setminus T^*(I)$ and returns $false$ together with w as a negative counterexample to the learner. Otherwise, the teacher infers that $w \in T^*(I) \cap B$ and outputs $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = false$ with the word w as an evidence.

Case 3: The condition (3) is violated, i.e., there exists a pair of words $(w, w') \in T$ such that $w \in A_h \land w' \notin A_h$. The teacher will check if $w \in T^*(I)$. If it is, then the teacher knows that $w' \in T^*(I) \land w' \notin A_h$ and hence returns $false$ together with w' as a positive counterexample to the learner. If $w \notin T^*(I)$, then the teacher knows that $w \notin T^*(I) \land w \in A_h$ and hence returns $false$ together with w as a negative counterexample to the learner.

Case 4: All of the three conditions are satisfied. The teacher concludes that $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = true$ and provides an inductive invariant $\mathcal{A}_h$ as a proof.

Our learning-based algorithm is correct by construction: when the algorithm concludes $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = true$, it provides a proof of $T^*(I) \cap B = \emptyset$; when the algorithm concludes $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = false$, it provides an evidence of $T^*(I) \cap B \neq \emptyset$. We summarise

this observation in the following theorem.

Theorem 4.5.1 (Correctness)**.** *If the algorithm from Fig. 4.3 terminates, it gives correct answer to the RMC problem* $(\mathscr{I}, \mathscr{T}, \mathscr{B})$.

Furthermore, if one of the L^* learning algorithms[1] from Section 4.4 is used, we can obtain an additional result about termination:

Theorem 4.5.2 (Termination)**.** *When* $T^*(I)$ *is regular, the algorithm from Fig. 4.3 is guaranteed to terminate in at most* n *iterations, where* n *is the size of the minimal DFA of* $T^*(I)$.

Proof. Observe that in the algorithm, the counterexample obtained by the learner in each iteration locates in the symmetric difference of the candidate language and $T^*(I)$. Hence, when $T^*(I)$ can be recognised by a DFA of n states, the algorithm will not execute more than n iterations by Proposition 4.4.1. #

Two remarks are in order. First, it is known that $T^*(I)$ is regular for many subclasses of infinite-state systems that can be modelled in regular model checking including pushdown systems, reversal-bounded counter systems, two-dimensional VASS (Vector Addition Systems with States), and other subclasses of counter systems [Iba78; BFLS05; LS05; TL10; Lin12]. Second, even in the case when $T^*(I)$ is not regular, termination may still happen thanks to the "generosity" of the teacher, which will accept any inductive invariant as an answer.

Forward exploration v.s backward exploration Our approach employs forward exploration to search for a proof. Namely, the target language of the teacher is $T^*(I)$ for an RMC problem $(\mathscr{I}, \mathscr{T}, \mathscr{B})$. One can also explore proofs in a backward manner, such that the target language of the teacher is the complement of $(T^{-1})^*(B)$. In our experiments, nevertheless, the performance of a backward search is often worse than its forward counterpart. This result may reflect the fact that the set $T^*(I)$ is "more regular" (i.e., can be expressed by a DFA with fewer states) than the set $(T^{-1})^*(B)$ in practical cases.

[1] If NL^* is used, the bound in Theorem 4.5.2 will increase to $O(k^2)$.

Considerations on implementation The implementation of the learning-based algorithm is very simple. Since it is based on standard automata learning algorithms and uses only basic automata/transducer operations, one can find existing libraries for them. The implementation only need to take care of how to answer queries. The core of our implementation has only around 150 lines of code (excluding the parser of the input models). We provide a few suggestions to make the implementation more efficient. First, each time when an automaton recognising $Post^k$ is produced, we store the pair $(k, Post^k)$ in a cache. It can be reused when a query on any word of length k is posed. We can also check if $Post^k \cap B = \emptyset$. The algorithm can immediately terminate and return $(\mathcal{I}, \mathcal{T}, \mathcal{B}) = false$ if $Post^k \cap B \neq \emptyset$. Second, for each language inclusion test, if the inclusion does not hold, we suggest to return the shortest counterexample. This heuristic helped to shorten the average length of strings sent for membership queries and hence reduced the cost of answering them. Recall that the algorithm needs to build the automaton of $Post^k$ to answer membership queries. The shorter the average length of query strings is, the fewer instances of $Post^k$ have to be built.

4.6 Experiments and evaluation

Experiments for safety verification

To evaluate our techniques, we have developed a prototype tool in Java and used the LibAlf library [BKK+10] as the default inference engine. For safety properties, we use our prototype to check a range of parameterised systems including cache coherence protocols (German's protocol [GS92]), self-stabilising protocols (Israeli-Jalfon's protocol [IJ90] and Herman's protocol [Her90]), synchronisation protocols (Lehmann-Rabin's dining philosophers protocol [LR81]), secure multi-party computation protocol (David Chaum's dining cryptographers protocol [Cha88]), mutual exclusion protocols (mutex array [FO97], Szymanski's protocol [Szy88], Burn's protocol [ADHR07], Dijkstra's protocol [Dij84; FO97; ADHR07], Lamport's bakery protocol [Fok18], and resource-allocator protocol [Don07]), and counter systems (Coffee Can [LR16], Water Jugs [Wikb], and

Kanban [KKW10]). Most of the examples we consider are standard benchmarks in the literature of regular model checking (cf. [BJNT00; Nil05; ADHR07; BHRV12; AHH13]). Among them, German's protocol and Kanban are considered more difficult than the other examples for fully automatic verification (cf. [ADHR07; KKW10; AHH13]).

We have compared our learning method with several automated safety verification techniques for regular model checking, including SAT-based refinement [NJ13; Nei14; LNRS15; LR16], extrapolating/widening [Leg08; LW10], and abstraction refinement [BHRV12]. The SAT-based approach encodes automata as propositional formulae and exploits a SAT-solver to search for candidate regular inductive invariants. The approach uses automata-based algorithms to either verify the correctness of a candidate or obtain a counterexample that can be further encoded as a Boolean constraint. T(O)RMC [Leg08; LW10] searches for an inductive invariant by extrapolating automata representing the stepwise reachable sets. The extrapolation is computed by first identifying the increment between successive automata, and then over-approximating the repetition of the increment by adding loops to the limiting automaton. ARMC [BHRV12] is an automated technique that integrates abstraction refinement into the fixed-point computation of reachable configurations. It begins with an existential abstraction obtained by merging states in the automata/transducers. Each time a spurious counterexample is found, the abstraction can be refined by splitting some of the merged states.

The results of our comparisons are reported in Table 4.1. The experiments were conducted on a desktop computer with 3.6GHz Intel i7 processor, 4GB memory limit, and 10-minute timeout. It is tricky to compare performance with T(O)RMC and ARMC, since one needs to set up a bundle of options before using these tools. In our evaluation, we selected the settings that we found to have the best overall performance for the benchmarks.[2] Overall, our prototype managed to verify all instances but Kanban, of which the minimal inductive invariant, if it is regular, has at least ten thousand states.

[2] For T(O)RMC, we set the sampling sequence to be $n_i := 1 + 12i$. For ARMC, we enabled the heuristics strpres bwcomp bwcoll allstpred sigma_str init_str tau_str bad_str [sigstar_str].

The safety property								Learning			SAT			T(O)RMC			ARMC
Name	#L	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{bad}	T_{bad}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time
Bakery	3	3	3	5	19	3	9	0.0s	6	18	0.5s	2	5	0.0s	6	11	0.0s
Burns	12	3	3	10	125	3	36	0.2s	8	96	1.1s	2	10	0.1s	7	38	0.6s
Szymanski	11	9	9	123	469	13	40	0.6s	48	528	t.o.	–	–	0.9s	51	102	t.o.
German	581	3	3	17	9.5k	4	2112	4.8s	14	8134	t.o.	–	–	t.o.	–	–	2.3s
Dijkstra, linear	42	1	1	13	827	3	126	0.1s	9	378	1.7s	2	24	6.1s	8	83	t.o.
Dijkstra, ring	12	3	3	13	199	3	36	1.4s	22	264	0.9s	2	14	t.o.	–	–	t.o.
Dining Crypt.	14	10	30	17	70	12	70	0.1s	32	448	t.o.	–	–	0.2s	37	164	1.3s
Coffee Can	6	8	18	13	34	5	8	0.0s	3	18	0.2s	2	7	0.1s	6	13	0.0s
Herman, linear	2	2	4	4	10	1	1	0.0s	2	4	0.2s	2	4	0.0s	2	4	0.0s
Herman, ring	2	2	4	9	22	1	1	0.0s	2	4	0.4s	2	4	0.0s	2	4	0.0s
Israeli-Jalfon	2	3	6	24	62	1	1	0.0s	4	8	0.1s	2	4	0.0s	4	8	0.0s
Lehmann-Rabin	6	4	4	14	96	3	13	0.1s	8	48	0.5s	2	11	0.8s	19	105	0.0s
LR Dining Phil.	4	4	4	3	10	3	4	0.0s	4	16	0.2s	2	6	0.1s	7	18	0.0s
Mux Array	6	3	3	4	31	3	18	0.0s	5	30	0.4s	2	7	0.2s	4	14	0.1s
Res. Allocator	3	3	3	7	25	4	11	0.0s	5	15	0.0s	3	7	0.0s	4	9	0.0s
Kanban	3	25	48	98	250	37	68	t.o.	–	–	t.o.	–	–	t.o.	–	–	t.o.
Water Jugs	11	5	6	23	132	5	12	0.1s	24	264	t.o.	–	–	t.o.	–	–	t.o.

Table 4.1 Comparing the performance of different RMC techniques. #L stands for the size of alphabet; S_x and T_x stand for the numbers of states and transitions, respectively, in the automata/transducers x. The **Learning** column lists the results of our prototype using Rivest and Schapire's version of L^*, while **SAT**, **T(O)RMC**, and **ARMC** present the results of the other three techniques.

The performance of SAT-based refinement was comparable to the other approaches when a small regular inductive invariant exists. On the other hand, the method timed out for instances that require larger proofs, which might be expectable since its runtime grows exponentially with the sizes of candidate automata. T(O)RMC and ARMC solved most of the instances amenable to the SAT-based method, plus several instances that require relatively larger proofs. Interestingly, while ARMC successfully verified several large instances (e.g. German) in a short time, it timed out for some instances (e.g. Dijkstra linear/ring) that have relatively small proofs.

Table 4.2 reports the results of the learning-based algorithm geared with different automata learning algorithms implemented in LibAlf. As the table shows, these algorithms had similar performance on small examples; however, the algorithm of

	RS			L^*			L^*c			KV			NL^*		
	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}
Bakery	0.0s	6	18	0.0s	6	18	0.1s	6	18	0.0s	6	18	0.1s	6	18
Burns	0.2s	8	96	0.5s	8	96	0.2s	8	96	0.2s	8	96	0.4s	6	72
Szymanski	0.3s	43	473	2.4s	51	561	1.2s	41	451	0.3s	41	451	1.4s	59	649
German	4.8s	14	8134	13s	15	8715	26s	15	8715	4.2s	14	8134	40s	15	8715
Dijkstra	0.1s	9	378	0.4s	9	378	0.1s	9	378	0.2s	9	378	0.2s	10	420
Dijkstra, ring	1.4s	22	264	2.7s	20	240	8.9s	22	264	1.5s	14	168	1.8s	20	240
Dining Crypt.	0.1s	32	448	0.2s	34	476	0.2s	38	532	0.1s	19	266	0.3s	36	504
Coffee Can	0.0s	3	18	0.0s	3	18	0.0s	4	24	0.0s	3	18	0.0s	4	24
Herman, linear	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4
Herman, ring	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4	0.0s	2	4
Israeli-Jalfon	0.0s	4	8	0.0s	4	8	0.0s	4	8	0.0s	4	8	0.0s	4	8
Lehmann-Rabin	0.1s	8	48	0.2s	8	48	0.1s	8	48	0.1s	8	48	0.2s	8	48
LR Dining Phil.	0.0s	4	16	0.2s	4	16	0.0s	5	20	0.0s	4	16	0.0s	8	32
Mux Array	0.0s	5	30	0.0s	5	30	0.0s	5	30	0.0s	5	30	0.0s	5	30
Res. Allocator	0.0s	5	15	0.0s	4	12	0.0s	5	15	0.0s	5	15	0.0s	5	15
Kanban	t.o.	–	–	t.o.	–	–	t.o.	–	–	t.o.	–	–	t.o.	–	–
Water Jugs	0.1s	24	264	0.5s	25	275	0.5s	25	275	0.1s	24	264	0.5s	25	275

Table 4.2 The performance of our learning method based on different automata learning algorithms. The columns $\mathbf{L}^*$, $\mathbf{L}^*\mathbf{c}$, **RS**, **KV**, and $\mathbf{NL}^*$ are the results of the original L^* algorithm by Angluin [Ang87], a variant of L^* that adds all suffixes of the counterexample to columns, the version by Rivest and Shapire [RS93], the version by Kearns and Vazirani [KV94], and the NL^* algorithm [BHKL], respectively.

Rivest and Schapire [RS93] and the algorithm of Kearns and Varzirani [KV94] were significantly more efficient than the other algorithms on some large examples such as Szymanski and German. Table 4.2 shows that Kearns and Varzirani's algorithm could often find inductive invariants with fewer states than those found by the other L^* variants, which explains the performance difference. For NL^*, our implementation pays an additional cost to determinise the learned FA in order to answer the equivalence queries; this cost is significant when a large invariant is needed.

The liveness property								The induced safety property						
Name	#L	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{fin}	T_{fin}	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{bad}	T_{bad}	TimeR
Bakery	6	3	3	8	25	2	12	6	6	17	43	7	12	0.0s
Res. Allocator	6	3	3	18	62	3	16	6	6	36	91	8	12	0.0s
Dining Crypt.	12	7	22	13	62	2	9	14	29	41	150	13	24	0.2s
LR Din. Phil.	7	4	4	18	47	2	13	8	8	38	116	8	14	0.1s
Selection Sort	2	7	8	25	34	5	10	8	8	70	102	3	4	0.0s
Szymanski	11	9	9	79	233	6	22	14	14	532	2021	12	22	1.1s
The liveness property under fairness requirements								**The induced safety property**						
Name	#L	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{fin}	T_{fin}	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{bad}	T_{bad}	TimeR
Ticket	5	6	9	32	73	4	6	11	13	249	506	9	13	0.2s
Din. Phil. Encl.	4	75	100	158	340	40	101	66	80	1117	2270	12	19	2.8s
Self-stabilising	2	245	424	839	1461	18	35	185	236	4683	7515	18	23	4.1s
Leader Election	2	74	95	1393	2452	12	24	57	66	2333	3754	15	29	4.3s

The liveness property		Learning			SAT			T(O)RMC			ARMC	SLRP
Name	#L	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	Time	Time
Bakery	6	16	96	0.4s	4	12	1.5s	15	30	0.0s	0.1s	1.1s
Res. Allocator	6	28	168	0.5s	4	20	17s	32	65	0.1s	0.3s	1.0s
Dining Crypt.	12	12	60	1.2s	3	35	16s	111	246	0.3s	0.2s	2.2s
LR Din. Phil.	7	43	301	1.0s	4	24	17s	78	164	0.2s	2m09s	1.3s
Selection Sort	2	52	104	0.7s	5	10	10s	–	–	t.o.	t.o.	1.5s
Szymanski	11	780	8580	8m35s	–	–	t.o.	1062	1610	1m45s	t.o.	3.0s
Liveness under fairness		**Learning**			**SAT**			**T(O)RMC**			**ARMC**	**SLRP**
Name	#L	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	S_{inv}	T_{inv}	Time	Time	Time
Ticket	5	75	375	1.3s	–	–	t.o.	82	115	0.7s	0.56s	n/a
Din. Phil. Encl.	4	175	700	4.3s	–	–	t.o.	–	–	t.o.	t.o.	n/a
Self-stabilising	2	1041	2082	7m34s	–	–	t.o.	–	–	t.o.	t.o.	n/a
Leader Election	2	770	1540	2m56s	–	–	t.o.	–	–	t.o.	t.o.	n/a

Table 4.3 Comparing the performance of different RMC techniques for liveness verification based on the liveness-to-safety procedure described in Section 4.3. The first table lists the liveness problems and their induced safety properties, with TimeR standing for the runtime of the reduction procedures. The second table presents the results of applying various tools to these induced safety properties. These results include the sizes of the inductive invariants, whenever available, and the runtime of the tools for producing these invariants. As in Table 4.1, our tool uses Rivest and Schapire's version of L^* as the default automata learning algorithm.

Experiments for liveness verification

To evaluate the performance of our learning-based method for liveness verification, we considered unconditional liveness properties of Lamport's bakery protocol [Fok18], the dining cryptographers protocol [Cha88], the left-right dining philosophers protocol [LSS94], the resource-allocator protocol [Don07], Szymanski's mutual exclusion protocol [Szy88], and a selection sort algorithm (see Section 5.6.1) in our experiments. For liveness properties under fairness requirements, we considered the ticket protocol [Wika], an asynchronised version of Kurshan's dining philosophers with encyclopaedia [KM95], and two case studies Dijkstra's self-stabilising protocol and Chang-Robert's leader election protocol from Chapter 5.

For these examples, our prototype tool first performed the liveness-to-safety reduction described in Section 4.3 to obtain their induced safety properties. We then applied our learning-based methods and the three RMC techniques SAT-based refinement, T(O)RMC, and ARMC to check these safety properties as we have done in the safety verification experiments. Additionally, we included the SLRP tool from [LR16] in the experiments involving unconditional liveness properties. SLRP is a sophisticated model checker capable of verifying, among others, the liveness property of a regular transition system under arbitrary schedulers. Simply speaking, the SLRP tool checks liveness by synthesising a regular progress relation that guides the system toward the set of accepting configurations. The synthesis algorithm of SLRP is incremental and exploits both L^* learning and SAT-based refinement.

The experiments were conducted on a desktop computer with 3.6GHz Intel i7 processor, 16GB memory limit, and 10-minute timeout. The results are presented in Table 4.3. For liveness without fairness requirements, the performance of our tool was comparable to that of SLRP on all examples except Szymanski's protocol, which took about 8.5 minutes for our tool to verify but merely 3 seconds for SLRP. This gap may possibly be ascribed to SLRP's advantage of working directly on the original liveness property rather than on the much larger induced safety property. In particular, a liveness property that is provable with a small regular progression relation may require

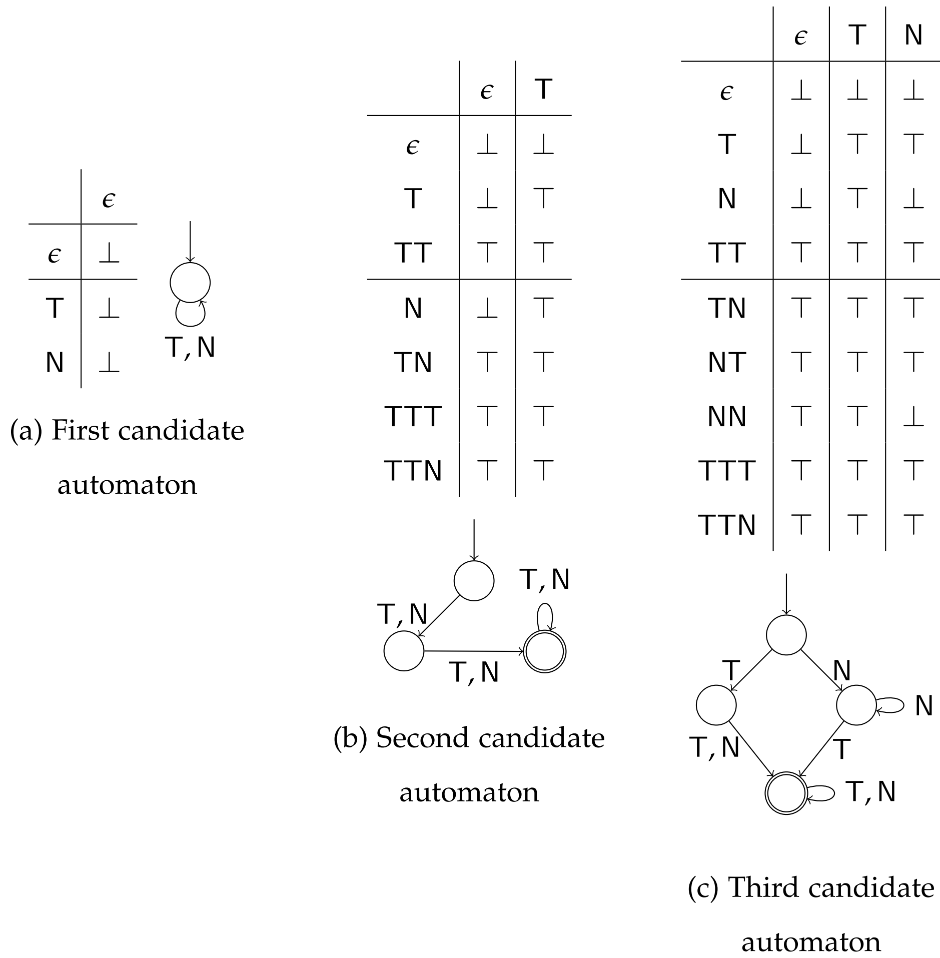
a large safety proof after it is reduced to a safety property. When fairness is involved, the SLRP tool is no longer directly applicable, whereas our learning method can still work by encoding the fairness requirements in the induced safety property. This encoding however incurs a significant overhead in all aspects: the reduction time, the size of the resulting property, and the runtime of the learning algorithm. Fortunately, our tool managed to solve all the challenging examples in our evaluation. The SAT-based method spotted minimal proofs for the smaller problems but timed out on the larger ones. T(O)RMC and ARMC appeared to suffer from a similar scalability issue, but they successfully verified some instances with proof sizes beyond the capacities of SAT-based enumeration.

Concluding remarks

In this chapter, we study a verification framework for safety and liveness properties of regular transition systems. We describe a framework that exploits automata learning to infer regular proofs for these properties. The encouraging experimental results suggest that, albeit the simplicity in concept, our learning-based framework has comparable performance to the sophisticated techniques in the literature. From a theoretical viewpoint, learning-based approaches (including ours and [HV05; Var06; NJ13]) have a termination guarantee when the set of reachable configurations is regular, which is the case with, e.g., lossy channel systems and pushdown systems. Approaches based on fixed-point computation (e.g. [BLW03; Nil05; LW10; BHRV12]) are in lack of such guarantees.

As future work, we plan to improve our framework in two aspects. The first is to automate the liveness-to-safety reduction for regular model checking. We expect to formally perform the reduction from LTL-like specifications to regular safety properties in a way similar to the translation from LTL(MSO) formulae to Büchi transition systems [AJN+12]. The second direction is to increase the chance of convergence by introducing some form of over-approximation in the learning process. For example, when the learning algorithm diverges and keeps guessing larger and larger candidate automata,

it is possible to apply the extrapolation techniques from T(O)RMC [Leg08] on these automata to obtain a proof. Another possibility is to combine regular language inference with abstraction refinement. For example, we may quotient the transition system with the equivalence relations considered in ARMC [BHV04; BHRV06]. A regular proof inferred for the quotient system suffices to verify the original system. Whenever a spurious counterexample is found, the algorithm analyses the counterexample to determine whether to refine the quotient system and restart the learning process for the new system, or to refine the candidate proof and continue the learning process in the current system. We have conducted preliminary experiments with these extensions and the results look promising.



	ϵ
ϵ	$\bot$
T	$\bot$
N	$\bot$

(a) First candidate automaton

	ϵ	T
ϵ	$\bot$	$\bot$
T	$\bot$	$\top$
TT	$\top$	$\top$
N	$\bot$	$\top$
TN	$\top$	$\top$
TTT	$\top$	$\top$
TTN	$\top$	$\top$

(b) Second candidate automaton

	ϵ	T	N
ϵ	$\bot$	$\bot$	$\bot$
T	$\bot$	$\top$	$\top$
N	$\bot$	$\top$	$\bot$
TT	$\top$	$\top$	$\top$
TN	$\top$	$\top$	$\top$
NT	$\top$	$\top$	$\top$
NN	$\top$	$\top$	$\bot$
TTT	$\top$	$\top$	$\top$
TTN	$\top$	$\top$	$\top$

(c) Third candidate automaton

Figure 4.2 Using learning to verify the safety property $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ of Israeli-Jalfon's Protocol. The table on the left of each sub-figure is the content of the *observation table* used by the automata learning algorithm of Rivest and Shaphire [RS93].

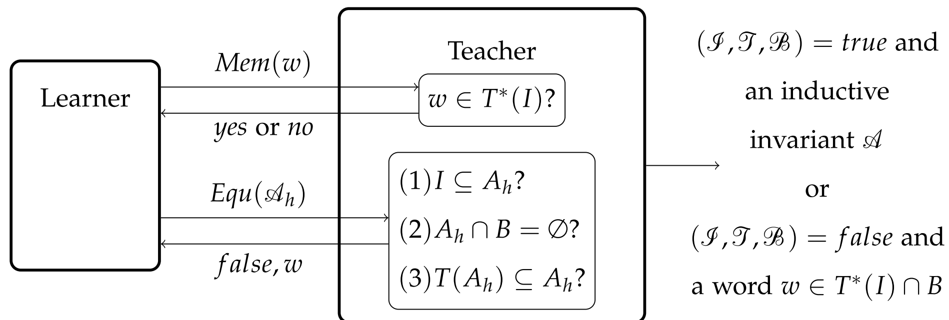


Figure 4.3 Overview: using automata learning to model check the safety property $(\mathcal{I},\mathcal{T},\mathcal{B})$. Recall that we use calligraphy font for automata/transducers and roman font for the corresponding languages/relations.

Chapter 5

Safety and Liveness Verification of Array Systems

Model checking of infinite-state systems manipulating arrays, which we shall refer to as *array systems* hereafter, is a popular topic in formal verification. Typical examples of array systems range from sequential array programs to concurrent process systems with numerical local and shared variables. Even though such variables might not directly appear in the specification, they can be convenient in the model for verification purpose. For example, it is often helpful to introduce auxiliary integer variables to count the number of processes in certain states in the verification of distributed protocols. When speaking of an array system, one naturally expresses the properties by quantifying over the elements on an array. For instance, the property "array a is sorted in an ascending order" is expressed by a universally quantified formula stating that each element on array a is no smaller than its preceding elements. To verify the correctness of an array system, it is often necessary to reason about quantified formulae, which is known to be a challenging task for decidable formalisms such as SMT over array theories.

In this chapter, we introduce a unified abstraction framework to verify linear-time properties of array systems. We propose a formalism that combines Linear Temporal Logic and a parametric quantified array theory. This formalism may be regarded as a

variant of *indexed LTL* [CGB86; GS92] with atomic propositions replaced by expressions from the array theory. We furthermore employ a generalised version of predicate abstraction to compute over-approximations of the specification in the abstract domain. While most abstraction techniques for array systems in the literature compute a finite abstract system, our approach produces an infinite-state abstract system. Particularly, we have chosen to abstract away the array values but keep the concrete values of array sizes and index variables in the abstract domain. This strategy often allows us to compute a precise enough abstraction based on a small set of predicates. Since array sizes are often unbounded in an array system, the cardinality of the abstract domain is inevitably infinite. We therefore exploit regular model checking as a symbolic framework to represent and analyse the abstract system. For this purpose, we over-approximate sets and relations in the abstract domain with regular languages called *regular abstractions*. Thanks to this regular representation, we can leverage off-the-shelf regular model checker to perform reachability and termination analysis for the abstract system. To evaluate our approach, we consider non-trivial safety and liveness properties for several case studies, including Selection Sort, Dijkstra's self stabilising algorithm [Dij82], and Chang-Robert's leader election algorithm [CR79]. We report promising experimental results for these case studies using existing regular model checkers as the backend inference engines.

Our work in this chapter initiates the study of numerical abstractions for array systems, with the goal towards "finitising" the infinite data domain. Particularly, we provide a mechanic approach to compute models of array systems that are suitable for analysis in regular model checking. The main contribution of this work is hence to lift the powerful verification methods of regular model checking from finite alphabets to infinite data domains. While our approach is based on predicate abstraction, we have not considered the orthogonal problem of computing the set of adequate predicates. We leave for future work the interesting direction of combining our work with automatic predicate discovery and refinement.

5.1 Related work

In this section, we provide some context for our work and compare with the most relevant work in the literature. There is a substantial body of work regarding automated verification of array-like systems. We shall concentrate on the development concerning the verification techniques of quantified temporal properties over unbounded arrays.

For array systems over finite data domains, automated approaches are often studied in the context of parameterised verification [CES86; CGB86; GS92]. The most generic approach along this line is arguably LTL(MSO) [AJN+12], which is a symbolic framework for specifying and verifying array systems with bounded array values. An LTL(MSO) formula provides a space-time specification where space (i.e. array contents) is constrained by the logic MSO, and time (i.e. evolution of array values) is constrained by the logic LTL. We note that LTL(MSO) is essentially as expressive as the symbolic framework our abstraction method is based on. In fact, under certain technical conditions, it is possible to rewrite the specification of our abstract system in LTL(MSO) and use an automata-theoretic formalism similar to [Nil05; AJN+12] to reason about the abstract system.

The work [APR+01; PRZ01] proposed a method called *hidden invariants* to reason about universally quantified safety properties for *bounded-data* array systems, where specifications are given in difference bound constraints on variables and array values ranging over finite but unbounded data domains. The method first computes reachability sets for finite and small instances of the system, and then generalises them to construct an inductive invariant. Abdulla et al. [ADR09] extended the parameterised verification framework of [ADHR07] to handle a class of numerical array systems. Their approach employs monotonic abstraction for transition relation to approximate an array system with a well-structured transition system. To ensure decidability of reachability analysis, valuations of numerical variables in this approach are expressed via *gap-order constraints* [Rev93], which are strictly weaker than difference bound constraints.

For array systems over infinite data domains, numerous automated techniques have

been developed to infer universally quantified invariance properties. Many of these techniques exploit predicate abstraction and interpolation theorems for array theories [FQ02; JM07; LB07; McM08; SPW09; ABG+12]. Notably, Flanagan and Qadeer [FQ02] introduced *indexed predicates*, which are essentially predicates containing free index variables. This apparatus allowed them to compute complex invariants by composing simple predicates under universal quantification. The same notion was later formalised by Lahiri and Bryant in the framework of *indexed predicate abstraction* [LB04a; LB04b; LB07]. While our method also utilises indexed predicates to build abstractions, we allow indexed predicates to be composed under more general quantification, and we only need to resolve decidable theories in the construction of abstract systems.

For multi-threaded programs, specialised predicate abstraction techniques were developed to reason about universally quantified inter-thread properties [Mal10; DKK+12; KKW14]. These techniques however assume symmetric systems, i.e., the system is supposed to behave correctly regardless of the arrangements of the threads. Particularly, expressions like $a[i] \leq a[j]$ are not permitted in the correctness specification since they would introduce asymmetry by making the thread order significant. Environment abstraction [CTV06; CTV08] is a technique capable of verifying inter-thread properties for protocols that break symmetry with thread identifiers. The technique enhances counter abstraction with predicate abstraction by counting processes satisfying an *environment predicate*, which essentially expresses a relationship between numerical variables in reference process and non-reference processes. Like conventional counter abstraction, environment abstraction abstracts away system topology and is thus more suitable to deal with unstructured systems.

MCMT [GR10; RG10] and its derivative *Cubicle* [CGK+12; CGK+13] are SMT-based frameworks targeting safety verification of *array-based systems*, a syntactically restricted functional subclass of array systems [GNRZ08]. These frameworks represent state sets and transition relations as first-order formulae, and perform a symbolic backward reachability analysis [ACJT96; ADHR07; ADR09] to infer universally quantified inductive invariants. The reachability analysis requires effective instantiation heuristics for

universal quantifiers to reduce fixed-point checking to SMT problems. The work of [GSM16] and [MIG+21] provide two alternative SMT-based approaches for inferring universal invariance properties of arrays. The former iteratively fixes the expected number of universal quantifiers in the candidate formulae, and uses eager quantifier instantiation [BMR13; McM18] to transform the verification condition to non-linear constraint Horn clauses. The latter integrates automated auxiliary variables and axioms discovery with the standard array abstraction to reduce the search of quantified invariants to quantifier-free and array-free reasoning.

To handle invariance properties with quantifier alternation, several work have combined predicate abstraction with templates-based reasoning [GMT08; GSV09; SG09; JKD+15]. These approaches tend to impose restrictive syntactic formats on feasible formulae [GMT08; GSV09; SG09], and/or need to resolve undecidable logics as part of the reasoning [GSV09; JKD+15]. In comparison, our method generates complex invariants without templates, and expresses verification conditions in decidable theories via regular abstractions. Other approaches proposed to perform exact quantified reasoning based on quantifier elimination. For example, Garg et al. [GLMN13] represented quantified invariants expressed in the decidable *array property fragment* [BMS06] with a data structure called *quantified data automata*, and leveraged classic automata learning algorithms to synthesise invariants based on sample executions of the input system. Ivy [PMP+16; MP18; MP20] is a semi-automatic verification framework geared towards a decidable fragment of first-order logic called *Effectively Propositional Logic* (EPR). To keep the reasoning decidable, modelling tricks such as partial axiomatisation, module isolation, and theory hiding are often necessary to express non-trivial systems [PLSS17; TLM+18]. Mypyvy [KBI+17; FWSS19] is a model checker built on top of Ivy, and enables a fully automatic inference of universal invariants through reduction to EPR. Recently, mypyvy has gained the power of learning general quantified formulae by combining the IC3/PDR algorithm [Bra11] and a procedure for inferring first-order separators [KPIA20].

Abstraction techniques for model checking liveness properties are also available.

Among them, *transition predicate abstraction* [PR07; GPR11] extends the classical predicate abstraction by annotating abstract states with abstract transitions. The technique reduces a liveness property to a fair termination problem via abstraction, and determines termination by proving that the abstract transitions are well-founded. Daniel et al. [DCG+16] integrated predicate abstraction refinement with classic symbolic liveness model checking techniques such as liveness-to-safety and k-liveness. Notably, their method used implicit abstraction techniques [CGMT14] to avoid quantifier elimination when computing the abstract system, and this allowed them to work sorely on quantifier-free formulas throughout the reasoning.

In the context of multi-threaded programs, *thread-modular analysis* is a popular liveness and termination verification technique [CPR07; PR12; FKP16; KD17]. The technique operates essentially by considering each thread in isolation, assuming that the effects of all the other threads are either passive or at least do not disrupt its progress. Thread-modular analysis is hence not suitable for verifying liveness that requires coordination among all threads, which is the case with Dijkstra's self-stabilising algorithm in Section 5.6.2: proving that the first thread eventually updates its register requires showing that all the other threads collectively make progress on updating their registers.

The Ivy deductive verification system [PMP+16; MP18; MP20] offers a liveness-to-safety proof tactic to prove temporal properties in the first-order linear temporal logic FO-LTL [PHL+17; PHM+18]. This tactic allows the user to verify a liveness property using a safety proof, i.e., an inductive invariant. For infinite-state systems, Ivy supports *dynamic abstraction*, which is an over-approximation for cycle detection by a finite projection dynamically chosen based on the system's executions. Using dynamic abstraction, Ivy can automatically discharge temporal proof goals for infinite-state systems that are more general than parameterised systems. Notably, the concrete specification, the abstract system, and the candidate safety proofs for the abstract system are all expressed in EPR under this abstraction scheme. While our approach only targets parameterised systems, our formalism is not limited to EPR but can in

principle handle array logics over a wide range of background theories.

5.2 Specification for array systems

Notation

Throughout this chapter, we shall consistently use V to denote a set of variables. We say that ϕ is a $\sigma(V)$-formula if ϕ is a formula defined over vocabulary σ and the free variables in ϕ are contained in V. Given a structure $\mathfrak{S}$, we use $V^{\mathfrak{S}}$ to denote the set of sort-consistent assignments from the variables V to the universes of $\mathfrak{S}$. Given $\rho_x \in V_x^{\mathfrak{S}}$ and $\rho_y \in V_y^{\mathfrak{S}}$ such that $V_x \cap V_y = \emptyset$, we use $\rho_x \cdot \rho_y$ to denote the assignment $\rho \in (V_x \uplus V_y)^{\mathfrak{S}}$ such that $\rho(v) = \rho_x(v)$ for $v \in V_x$ and $\rho(u) = \rho_y(u)$ for $u \in V_y$. Conversely, given $\rho \in (V_x \uplus V_y)^{\mathfrak{S}}$, we use ρ_x and ρ_y to denote the assignments obtained by restricting the domain of ρ to V_x and V_y, respectively. We write v' for the primed version of variable v, and define $V' := \{v' : v \in V\}$. Given $\rho \in V^{\mathfrak{S}}$, we use ρ' to denote the assignment $\{v' \mapsto \rho(v) : v \in V\}$ for the variables in V'.

Array logic

An array logic permits reasoning about arrays, which are formalised as mappings from index positions to data elements. Our definition of the array logic combines two one-sorted theories: an *index theory* T_I, which is used to express relations between indices, and an *element theory* T_E, which is used to express properties of the entries in an array. We assume that the index theory T_I has vocabulary σ_I and sort symbol $\mathbb{I}$, the element theory T_E has vocabulary σ_E and sort symbol $\mathbb{E}$, and $\sigma_I \cap \sigma_E = \emptyset$.

The array logic induced by T_I and T_E is a three-sorted first-order theory where array symbols are of sort $\mathbb{A}$, array indices are of sort $\mathbb{I}$, and array entries are of sort $\mathbb{E}$. The vocabulary of the logic is $\sigma := \sigma_I \uplus \sigma_E \uplus \{\cdot[\cdot]\}$ where $\cdot[\cdot]$ is a function symbol of sort $\mathbb{A} \times \mathbb{I} \to \mathbb{E}$. The semantics of arrays is formalised by exclusively allowing models $\mathfrak{S} := \langle \mathbb{I}^{\mathfrak{S}}, \mathbb{E}^{\mathfrak{S}}, \mathbb{A}^{\mathfrak{S}}, \{s^{\mathfrak{S}}\}_{s \in \sigma} \rangle$ that interpret $\mathbb{A}$ as the set of functions $f : \mathbb{I}^{\mathfrak{S}} \to \mathbb{E}^{\mathfrak{S}}$ and interpret $\cdot[\cdot]$ as function application. Namely, it should hold that $(a[t])^{\mathfrak{S}} = a^{\mathfrak{S}}(t^{\mathfrak{S}})$ for

any array symbol a and σ_I-term t. Furthermore, it is required that $\mathfrak{S}_I := \langle \mathbb{I}^{\mathfrak{S}}, \{s^{\mathfrak{S}}\}_{s \in \sigma_I} \rangle$ and $\mathfrak{S}_E := \langle \mathbb{E}^{\mathfrak{S}}, \{s^{\mathfrak{S}}\}_{s \in \sigma_E} \rangle$ are models of T_I and T_E, respectively.

We shall fix an array logic T_A with vocabulary $\sigma_A := \sigma_I \uplus \sigma_E \uplus \{\cdot[\cdot]\}$ throughout this chapter. Furthermore, we shall fix the index theory of T_A to be difference arithmetic, and assume that the quantifier-free fragment of T_A is decidable. This assumption is satisfied when the combined theory of difference arithmetic, the element theory T_E, and the logic of equality and uninterpreted functions (EUF) are compatible with the Nelson-Oppen decision procedure [NO79; Opp80]. Logics that are qualified as a compatible element theory T_E include EUF, difference arithmetic, linear integer arithmetic, rational arithmetic, real arithmetic, and lots more. We refer to the textbooks [BM98; KS16] for more details about the Nelson-Oppen procedure and the combination of logics.

First-order array systems

A *first-order array system* $\mathfrak{T}$ is a triple $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ such that V is a finite set of first-order variables, ϕ_{init} is a $\sigma_A(V)$-formula, and ϕ_{tran} is a $\sigma_A(V \uplus V')$-formula. The variables in V are grouped into two sets V_{st} and V_{par}, called the *state variables* and the *system parameters*, respectively. A state variable can be of index, element, or array sort, whereas a system parameter is of index sort. A state variable of index sort is also called an *index variable*. For convenience, we shall use V_{int} to denote the set of parameters and index variables in V, and use V_{arr} to denote the set of array variables in V. Furthermore, we assume that each array variable $a \in V_{st}$ is associated with a parameter $|a| \in V_{par}$ representing the size of the array.

The semantics of an array system is determined by models of the array logic T_A as well as assignments for V. Formally, given a model $\mathfrak{A}$ of T_A, an $\mathfrak{A}$-*state* (or simply a *state*) of an array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ is a sort-consistent assignment from the variables V to the universes of $\mathfrak{A}$. The formula ϕ_{init} specifies the set of initial states. The formula ϕ_{tran} specifies the evolution of the system by relating the current variables V to their updated counterparts V'. A (finite or infinite) sequence of $\mathfrak{A}$-states $s_0 \cdots s_n$ with $n \in \mathbb{N} \uplus \{\infty\}$ is an *execution run* if $\mathfrak{A}, s_0 \models \phi_{init}$ and $\mathfrak{A}, s_i \cdot s'_{i+1} \models \phi_{tran}$ for

$i \in \{0, \ldots, n-1\}$. We stipulate that system parameters are *immutable*, i.e., their values are unchanged throughout the system execution. Therefore, an execution run $s_0 \cdots s_n$ is legitimate only if $s_0(v) = s_i(v)$ for all $v \in V_{par}$ and $i \in \{1, \ldots, n\}$.

In this chapter, we shall be mostly concerned with index-bounded array systems. An array system over variables V is *index-bounded* if for any index variable $x \in V_{st}$, there is a system parameter $p_x \in V_{par}$ such that $s(x) \leq s(p_x)$ holds in any reachable $\mathfrak{A}$-state s and model $\mathfrak{A}$ of T_A. Since system parameters are immutable, the range of an index variable is finite during the computation of an index-bounded system. This assumption is reasonable considering that the main purpose of using index variables is to access elements on a finite array. Examples of such variables include the pivot variable in a sorting algorithm, whose value is bounded by the size of the working array, and the process pointer in a parameterised concurrent system, whose value is bounded by the maximal possible number of processes. Note that index boundedness does not impose an restriction on array elements: a numerical array value in an index-bounded system can still become indefinitely large at runtime.

Indexed array logic

Given a set of variables $V := V_{st} \uplus V_{par}$, we define the *indexed array logic* over the variables V as the set of $\sigma_A(V)$-formulae in which only index variables are allowed to be quantified. A literal is a *value expression* if it contains an array variable; otherwise it is an *index expression*. An indexed array formula is said to be *singly indexed* if every value expression in ϕ contains *at most* one quantified index variable. We shall refer to such formulae as *singly indexed array* formulae, or *SIA* formulae for short. It turns out that the satisfiability problem of singly indexed array formulae is already undecidable when the element theory is instantiated to difference arithmetic, see [HIV08, Lemma 5] for details.

Observe that SIA formulae are closed under Boolean operations. Moreover, an SIA formula can be syntactically rewritten to a logically equivalent SIA formula in which every value expression has *exactly* one quantified index variable — hence the name

"singly indexed". Indeed, a value expression without a quantified index variable is equivalent to an SIA formula with exactly one quantified index variable, e.g., $a[1] \neq b[n]$ is equivalent to $\exists i.\, i = n \wedge a[1] \neq b[i]$. The rewriting can therefore be performed by iteratively replacing a value expression with a suitable SIA formula.

Specification language

We provide a specification language combining the indexed array logic with LTL to express the temporal properties of an array system. For ease of exposition, we only consider the temporal operators "globally" and "eventually". Our approach can be extended to handle other standard operators such as "next" and "until" in a straightforward manner.

Definition 5.2.1. A *temporal array property* ϕ is a formula constrained by the grammar

$$\phi ::= \psi \mid \neg\phi \mid \phi \vee \phi \mid \exists x.\, \phi \mid \Box\, \phi \mid \Diamond\, \phi,$$

where ψ is an indexed array formula, x is an index variable, and $\Box$ and $\Diamond$ are the standard "globally" and the "eventually" temporal operators, respectively, in LTL. As usual, we use the standard shorthand for logical operators such as $\phi \Rightarrow \psi := \neg\phi \vee \psi$, and $\forall x.\, \phi := \neg\exists x.\, \neg\phi$. #

A temporal array property ϕ is interpreted over paths. Let $\pi := s_0\, s_1 \ldots$ be a path, and $\pi^t := s_t\, s_{t+1} \ldots$ denote the suffix of π starting at position t. The semantics of ϕ with respect to a path π and a structure $\mathfrak{A}$ is listed as below.

$$\begin{aligned}
\mathfrak{A}, \pi \models \phi \quad &\text{iff} \quad \mathfrak{A}, s_0 \models \phi, \text{ given that } \phi \text{ is an indexed array formula} \\
\mathfrak{A}, \pi \models \neg\phi \quad &\text{iff} \quad \mathfrak{A}, \pi \not\models \phi \\
\mathfrak{A}, \pi \models \phi_1 \vee \phi_2 \quad &\text{iff} \quad \mathfrak{A}, \pi \models \phi_1 \text{ or } \mathfrak{A}, \pi \models \phi_2 \\
\mathfrak{A}, \pi \models \exists x.\, \phi \quad &\text{iff} \quad \text{there exists } a \in \mathbb{N} \text{ such that } \mathfrak{A}, \pi \models \phi[a/x] \\
\mathfrak{A}, \pi \models \Box\, \phi \quad &\text{iff} \quad \text{for all } t \geq 0, \text{ it holds that } \mathfrak{A}, \pi^t \models \phi \\
\mathfrak{A}, \pi \models \Diamond\, \phi \quad &\text{iff} \quad \text{for some } t \geq 0, \text{ it holds that } \mathfrak{A}, \pi^t \models \phi
\end{aligned}$$

A first-order array system $\mathfrak{T}$ *satisfies* a temporal array property ϕ if there exists a model $\mathfrak{A}$ of T_A such that $\mathfrak{A}, \pi \models \phi$ for some execution run π of $\mathfrak{T}$. We say that a temporal array property ϕ *holds* for an array system $\mathfrak{T}$ when $\mathfrak{T}$ does not satisfy $\neg\phi$. In other words, a property holds for $\mathfrak{T}$ if *every* execution run of $\mathfrak{T}$ satisfies the property in *every* model of T_A.

5.3 Regular languages as abstractions

5.3.1 Predicate abstraction with indexed predicates

We use indexed predicates to express constraints on the system states.

Definition 5.3.1 (Indexed predicate)**.** An *indexed predicate* is an indexed array literal containing a designated index variable i. A set of indexed predicates $\mathbb{P} := \langle P_1, \ldots, P_m \rangle$ is said to be *defined over variables* V if the variables contained in the predicates is a subset of $V \uplus \{i\}$. #

For convenience, we shall fix a set of variables V and a set of indexed predicates $\mathbb{P} := \langle P_1, \ldots, P_m \rangle$ over variables V throughout this section. For each bit-vector $v := b_1 \cdots b_m \in \Sigma_m$, we define an indexed array formula $\phi_v^{\mathbb{P}}$ as follows:

$$\phi_v^{\mathbb{P}} := \bigwedge\nolimits_{1 \le k \le m} \left(\Gamma(P_k) \Rightarrow \tilde{P}_k \right), \text{ where } \tilde{P}_k = \begin{cases} P_k, & b_k = 1; \\ \neg P_k, & b_k = 0. \end{cases} \tag{5.1}$$

Intuitively, $\phi_v^{\mathbb{P}}$ asserts that v is the "valuation" of the predicates in $\mathbb{P}$ at position i, and $\Gamma(e)$ denotes the sanity condition of the array access in expression e. For example, when the element theory is the equality logic with equality relation $\simeq$, the sanity condition is specified as

- $\Gamma(I \bowtie_1 J) := \mathsf{true}$
- $\Gamma(a[I] \bowtie_2 b[J]) := (1 \le I \wedge I \le |a|) \wedge (1 \le J \wedge J \le |b|)$

where a and b are array symbols, I and J are index terms, $\bowtie_2 \in \{\simeq, \not\simeq\}$, and $\bowtie_1 \in \{=, \neq, \leq, <, \geq, >\}$. The definition of $\phi_v^{\mathbb{P}}$ suggests that the k-th bit of a valuation v is meaningful only when the array access in the predicate P_k occurs within proper bounds.

Having identified the bit-vectors $v \in \Sigma_m$ with the valuations of the predicates in $\mathbb{P}$, we further use words $w \in \Sigma_m^*$ over bit-vectors to encode the collective valuations of $\mathbb{P}$ at positions $1, \ldots, |w|$. Formally, for each word $w \in \Sigma_m^*$, we define an indexed array formula

$$\Phi_w^{\mathbb{P}} := \bigwedge\nolimits_{1 \leq t \leq |w|} \phi_{w[t]}^{\mathbb{P}}[t/i]. \tag{5.2}$$

By construction, $\Phi_w^{\mathbb{P}}$ is true if and only if for $t \in \{1, \ldots, |w|\}$ and $k \in \{1, \ldots, m\}$, if the array access in $P_k[t/i]$ is legitimate, then the truth value of $P_k[t/i]$ will coincide with the k-th bit of vector $w[t]$.

Fix an array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ and a model $\mathfrak{A}$ of T_A. We define the *concrete domain* of the system as $V^{\mathfrak{A}}$, the set of the sort-consistent assignments for V. Let $r := |V_{int}|$ and $m := |\mathbb{P}|$. Since index sort is interpreted as natural numbers by $\mathfrak{A}$, we shall identify an assignment in $V_{int}^{\mathfrak{A}}$ with an r-tuple in $\mathbb{N}^r$, namely, it assigns the i-th component of the tuple to the i-th variable in V_{int}. We define the *abstract domain* induced by $\mathbb{P}$ and V as $\mathbb{N}^r \times \Sigma_m^*$, where each abstract state $(\overline{p}, w)$ consists of an assignment $\overline{p}$ for the variables in V_{int} and a word $w \in \Sigma_m^*$ encoding the valuations of $\mathbb{P}$ at positions $1, \ldots, |w|$.

Given a set of concrete states $S_C \subseteq V^{\mathfrak{A}}$, we define $\alpha(S_C)$ as the set of abstract states

$$\alpha(S_C) := \{(s_{int}, w) \in \mathbb{N}^r \times \Sigma_m^* : \text{there is } s \in S_C \text{ such that } |w| = \max_{x \in V_{int}} s(x) \text{ and } \mathfrak{A}, s \models \Phi_w^{\mathbb{P}}\}.$$

Given a set of abstract states $S_A \subseteq \mathbb{N}^r \times \Sigma_m^*$, we define $\gamma(S_A)$ as the set of concrete states

$$\gamma(S_A) := \{s \in V^{\mathfrak{A}} : \text{for all } w \in \Sigma_m^* \text{ with } |w| = \max_{x \in V_{int}} s(x),\ \mathfrak{A}, s \models \Phi_w^{\mathbb{P}} \text{ implies } (s_{int}, w) \in S_A\}.$$

To simplify the notation, we shall write $\alpha(s)$ instead of $\alpha(\{s\})$, and write $\gamma(s)$ instead of $\gamma(\{s\})$ whenever there is no danger of confusions.

Lemma 5.3.1. *The set $\gamma(S_A)$ consists of precisely the concrete states s satisfying $\alpha(s) \subseteq S_A$.*

Proof. It suffices to fix a concrete state $s \in V^{\mathfrak{A}}$ and show that $s \in \gamma(S_A) \iff \alpha(s) \subseteq S_A$. For this purpose, define $W_s := \{w \in \Sigma_m^* : |w| = \max_{x \in V_{int}} s(x)\}$.

($\Rightarrow$) Assume first that $s \in \gamma(S_A)$. Let $(\overline{p}, w)$ be an abstract state in $\alpha(s) \subseteq \{s_{int}\} \times W_s$. Then we have $\mathfrak{A}, s \models \Phi_w^{\mathbb{P}}$ by the definition of α, which implies that $(\overline{p}, w) \in S_A$ by the definition of γ. Since $(\overline{p}, w)$ is arbitrary in $\alpha(s)$, we conclude that $\alpha(s) \subseteq S_A$.

($\Leftarrow$) Now assume that $\alpha(s) \subseteq S_A$. Then S_A contains all abstract states (s_{int}, w) with $w \in W_s$ and $\mathfrak{A}, s \models \Phi_w^{\mathbb{P}}$ by the definition of α. However, this implies that for any $w \in W_s$, if $\mathfrak{A}, s \models \Phi_w^{\mathbb{P}}$ holds, then $(s_{int}, w) \in S_A$ must also hold. It follows that $s \in \gamma(S_A)$ by the definition of γ. #

The definitions of α and γ allow them to form a *Galois connection* [CC77; CC92], as stated in the following proposition.

Proposition 5.3.2. *For any set S_C of concrete states and set S_A of abstract states, it holds that*

$$\alpha(S_C) \subseteq S_A \iff S_C \subseteq \gamma(S_A). \tag{5.3}$$

Proof. ($\Rightarrow$) Assume first that $\alpha(S_C) \subseteq S_A$. Since α is monotonic, i.e., $E \subseteq F$ implies $\alpha(E) \subseteq \alpha(F)$, we have $\alpha(s) \subseteq S_A$ for every $s \in S_C$. By Lemma 5.3.1, $\gamma(S_A)$ consists of the concrete states s satisfying $\alpha(s) \subseteq S_A$. It then follows that $S_C \subseteq \gamma(S_A)$.

($\Leftarrow$) Now assume that $S_C \subseteq \gamma(S_A)$. By Lemma 5.3.1, $\gamma(S_A)$ consists of the concrete states s satisfying $\alpha(s) \subseteq S_A$. The assumption $S_C \subseteq \gamma(S_A)$ thus implies that $\alpha(s) \subseteq S_A$ for every $s \in S_C$, which in turn implies that $\alpha(S_C) \subseteq S_A$ by the fact that α distributes over set union. #

Predicate abstraction for safety and liveness verification of an array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ involves performing reachability analysis in the abstract domain of $\mathfrak{T}$. In each step of the analysis, we concretise the current set of reachable abstract states via the concretisation operation $\gamma(\cdot)$, apply the concrete next-step function

$$N_C(S_C) := \{t \in V^{\mathfrak{A}} : \text{there is } s \in S_C \text{ such that } \mathfrak{A}, s \cdot t' \models \phi_{tran}\} \tag{5.4}$$

in the concrete state space, and map the result back to the abstract state space via the abstraction operation $\alpha(\cdot)$. We can view this process as performing state exploration in an abstract transition system with the set of initial states $\alpha(\phi^{\mathfrak{A}}_{init})$ and an abstract next-step function

$$N_A(S_A) := \alpha(N_C(\gamma(S_A))). \tag{5.5}$$

By Proposition 5.3.2 and the monotonicity of N_C, if $\alpha(S_C) \subseteq S_A$, then it holds that $N_C(S_C) \subseteq \gamma(N_A(S_A))$. Therefore, the abstract next-step function N_A induces an over-approximation $S_C \mapsto \gamma(N_A(\alpha(S_C))) \supseteq N_C(S_C)$ for the concrete next-step function N_C. This fact, together with Proposition 5.3.2, establishes the soundness of the abstract reachability analysis. For example, one can easily show (e.g. by induction on the length of a concrete counterexample run) that the safety property $N_C^*(Init) \cap Bad = \emptyset$ holds in the concrete domain if $N_A^*(\alpha(Init)) \cap \alpha(Bad) = \emptyset$ holds in the abstract domain.

Multi-valued predicates

We have represented abstract states as words over alphabets of bit-vectors, i.e., each predicate is evaluated to either true or false. However, such representations are not optimal in certain situations. For example, it may be useful to have a single predicate indicating which of the three cases $a[i] < a[j]$, $a[i] = a[j]$, and $a[i] > a[j]$ holds in the concrete domain. Also, one may want to record the exact value of a program counter or control state in the abstract domain. In these use cases, our abstraction framework can be extended by allowing different predicates to have valuations over different ranges. More precisely, suppose that we have a set of indexed predicates $\mathbb{P} := \langle P_1, \ldots P_m \rangle$ such that the valuation of each P_k ranges over $\{1, \ldots, n_k\}$. Then the abstract domain induced by $\mathbb{P}$ will consist of words in $V^{\mathfrak{A}}_{int} \times (\{1, \ldots, n_1\} \times \cdots \times \{1, \ldots, n_m\})^*$ instead of $V^{\mathfrak{A}}_{int} \times \Sigma_m^*$. Definition 5.1 then will become

$$\phi^{\mathbb{P}}_{b_1 \cdots b_m} := \bigwedge\nolimits_{1 \le k \le m} \left(\Gamma(P_k) \Rightarrow \tilde{P}_k\right), \text{ where } \tilde{P}_k = \begin{cases} P_k^1, & b_k = 1; \\ \quad \ldots & \\ P_k^{n_k}, & b_k = n_k, \end{cases} \tag{5.6}$$

and $P_k^1, \ldots, P_k^{n_k}$ are indexed array literals that are disjoint and exhaustive. That is, for any $k \in \{1, \ldots, m\}$ and $s \in V^{\mathfrak{A}}$, it holds that (i) $\mathfrak{A}, s \models \bigvee_{i=1}^{n_k} P_k^i$, and (ii) $\mathfrak{A}, s \not\models P_k^i \wedge P_k^j$ for $i \neq j$. Following the previous example, we can use predicate P_k to encode the ordering between the array values $a[i]$ and $a[j]$ by setting $n_k := 3$, $P_k^1 := (a[i] < a[j])$, $P_k^2 := (a[i] = a[j])$, and $P_k^3 := (a[i] > a[j])$. To simplify the exposition, we shall mainly consider two-valued predicates in the sequel. All results presented in this chapter, however, can be applied to general multi-valued predicates with small modifications.

5.3.2 Approximations with regular languages

Since indexed array logic is undecidable in general, the abstraction operation $\alpha(\cdot)$ on indexed array formulae is generally not computable. Therefore, we propose to over-approximate these formulae in the abstract domain using regular languages.

As before, fix a model $\mathfrak{A}$ of T_A, a set of variables V, and a set of indexed predicates $\mathbb{P}$ over V. Let $r := |V_{int}|$ and $m := |\mathbb{P}|$. Recall that $\mathbb{P}$ and V together induce an abstract domain $\mathbb{N}^r \times \Sigma_m^*$. We say that a relation $R \subseteq \mathbb{N}^r \times \Sigma_m^*$ is *appropriate* if (i) for each tuple $(p_1, \ldots, p_r, w) \in R$, it holds that $|w| = \max\{p_1, \ldots, p_r\}$, and (ii) R has a regular presentation under unary encoding of natural numbers, i.e., each tuple $(p_1, \ldots, p_r, w) \in R$ is encoded as $(1^{p_1}, \ldots, 1^{p_r}, w)$ in the presentation.

Definition 5.3.2 (Regular abstraction for state formula). Let ϕ be an indexed array formula over variables V. A tuple $(R, \mathbb{P})$ is a *regular abstraction* for ϕ if

- $R \subseteq \mathbb{N}^r \times \Sigma_m^*$ is an appropriate relation.
- For any model $\mathfrak{A}$ of T_A and $s \in V^{\mathfrak{A}}$, if $\mathfrak{A}, s \models \phi$, then $(s_{int}, w) \in R$ for all $w \in \Sigma_m^*$ with $\mathfrak{A}, s \models \Phi_w^{\mathbb{P}}$.

#

From the above definition, it is easy to see that an appropriate regular relation R induces a regular abstraction for a state formula ϕ if and only if for any model $\mathfrak{A}$ of T_A, $\alpha(\phi^{\mathfrak{A}}) \subseteq R$ (or equivalently $\phi^{\mathfrak{A}} \subseteq \gamma(R)$, since (α, γ) is a Galois connection) holds under unary encoding of natural numbers.

Given $\mathbb{P} := \langle P_1, \ldots, P_m \rangle$ over V, we use $\mathbb{P} \cdot \mathbb{P}' := \langle P_1, \ldots, P_m, P_{m+1}, \ldots, P_{2m} \rangle$ to denote the set of predicates over $V \uplus V'$ such that for $k \in \{1, \ldots, m\}$, P_{m+k} is obtained from P_k by replacing v with v' for each variable $v \in V$. Given two words $u := u_1 \cdots u_n$ and $v := v_1 \cdots v_n$ of the same length from Σ_m^*, we write $u \star v$ for the word $(u_1 \cdot v_1) \cdots (u_n \cdot v_n) \in \Sigma_{2m}^*$, where $(u_i \cdot v_i)$ denotes the concatenation of the bit-vectors u_i and v_i. We say that a relation $R \subseteq (\mathbb{N}^r \times \Sigma_m^*) \times (\mathbb{N}^r \times \Sigma_m^*)$ is *appropriate* if (i) R has a regular presentation under unary encoding of natural numbers, and (ii) $Dom(R)$ and $Img(R)$ are appropriate over $\mathbb{N}^r \times \Sigma_m^*$.

Definition 5.3.3 (Regular abstraction for transition formula)**.** Let ϕ be an indexed array formula over variables $V \uplus V'$. A tuple $(R, \mathbb{P})$ is a *regular abstraction* for ϕ if

- $R \subseteq (\mathbb{N}^r \times \Sigma_m^*) \times (\mathbb{N}^r \times \Sigma_m^*)$ is an appropriate relation.
- For any model $\mathfrak{A}$ of T_A and $s, t \in V^{\mathfrak{A}}$, if $\mathfrak{A}, s \cdot t' \models \phi$, then $((s_{int}, u), (t_{int}, v)) \in R$ for all $u, v \in \Sigma_m^*$ satisfying $|u| = |v|$ and $\mathfrak{A}, s \cdot t' \models \Phi_{u \star v}^{\mathbb{P} \cdot \mathbb{P}'}$.

#

Similar to the case of state formulae, it follows from the above definition that an appropriate regular relation R induces a regular abstraction for a transition formula ϕ if and only if for any model $\mathfrak{A}$ of T_A, $\alpha(\phi^{\mathfrak{A}}) \subseteq R$ holds under unary encoding of natural numbers *plus* the bijective mapping $(s_{int} \cdot t_{int},\ u \cdot v) \mapsto ((s_{int}, u), (t_{int}, v))$.

For an index-bounded system, a regular abstraction for the transition formula serves as an over-approximation for the abstract next-step function N_A defined at (5.5):

Proposition 5.3.3. *Let $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ be an indexed-bounded array system, $\mathbb{P}$ be a set of indexed predicates over V, and N_A be the abstract next-step function of $\mathfrak{T}$ induced by $\mathbb{P}$. If $(R, \mathbb{P})$ is a regular abstraction for ϕ_{tran}, then $N_A(S_A) \subseteq R(S_A)$ holds for any set of abstract states S_A.*

Proof. If $\gamma(S_A) = \emptyset$, then $N_A(S_A) = \alpha(N_C(\gamma(S_A))) = \alpha(N_C(\emptyset)) = \emptyset \subseteq R(S_A)$. Otherwise, suppose that $\gamma(S_A) \neq \emptyset$ and consider an arbitrary state $s \in \gamma(S_A) \subseteq V^{\mathfrak{A}}$.

Note that

$$\begin{aligned}
\alpha(N_C(s)) &= \alpha(\{t \in V^{\mathfrak{A}} : \mathfrak{A}, s\cdot t' \models \phi_{tran}\}) \\
&= \{(t_{int}, v) : \mathfrak{A}, s\cdot t' \models \phi_{tran},\ |v| = \max_{x\in V_{int}} t(x),\ \mathfrak{A}, t \models \Phi_v^{\mathbb{P}}\ \} \\
&\subseteq R(\{(s_{int}, u) : |u| = \max_{x\in V_{int}} s(x),\ \mathfrak{A}, s \models \Phi_u^{\mathbb{P}}\ \}) \\
&= R(\alpha(s)),
\end{aligned}$$

where the set inclusion in the third line follows from the assumption that $\mathfrak{T}$ is index-bounded, and the fact that $\mathfrak{A}, s \models \Phi_u^{\mathbb{P}}$ and $\mathfrak{A}, t \models \Phi_v^{\mathbb{P}}$ hold at the same time if and only if $\mathfrak{A}, s\cdot t' \models \Phi_{u\star v}^{\mathbb{P}\cdot\mathbb{P}'}$. Moreover, we have $\alpha(s) \subseteq S_A$, and therefore $\alpha(N_C(s)) \subseteq R(\alpha(s)) \subseteq R(S_A)$, by Lemma 5.3.1 and the monotonicity of regular relations. Since both α and N_C are distributive over set union, we conclude that $N_A(S_A) = \alpha(N_C(\gamma(S_A))) = \bigcup_{s\in\gamma(S_A)} \alpha(N_C(s)) \subseteq R(S_A)$. #

The following handy facts allow us to compute a regular abstraction via composition.

Proposition 5.3.4. *Let $\phi_1, \ldots, \phi_n$ be indexed array formulae over a common set of variables V, and $(R_i, \mathbb{P})$ be a regular abstraction for ϕ_i for $i \in \{1, \ldots, n\}$. Then it holds that*

– $(\bigcup_{i=1}^n R_i, \mathbb{P})$ *is a regular abstraction for* $\bigvee_{i=1}^n \phi_i$.

– $(\bigcap_{i=1}^n R_i, \mathbb{P})$ *is a regular abstraction for* $\bigwedge_{i=1}^n \phi_i$.

Proof. Recall that $(R, \mathbb{P})$ is a regular abstraction for a formula ϕ if and only if $\alpha(\phi^{\mathfrak{A}}) \subseteq R$ for any model $\mathfrak{A}$ of T_A under a suitable encoding. Suppose that $\alpha(\phi_i^{\mathfrak{A}}) \subseteq R_i$ for $i \in \{1, \ldots, n\}$. Note that $\alpha((\bigvee_{i=1}^n \phi_i)^{\mathfrak{A}}) = \alpha(\bigcup_{i=1}^n \phi_i^{\mathfrak{A}}) = \bigcup_{i=1}^n \alpha(\phi_i^{\mathfrak{A}}) \subseteq \bigcup_{i=1}^n R_i$. Similarly, we have $\alpha((\bigwedge_{i=1}^n \phi_i)^{\mathfrak{A}}) = \alpha(\bigcap_{i=1}^n \phi_i^{\mathfrak{A}}) = \bigcap_{i=1}^n \alpha(\phi_i^{\mathfrak{A}}) \subseteq \bigcap_{i=1}^n R_i$. This concludes the proof. #

Symbolic representation with $\mathrm{FO}_{\mathrm{REG}}$

We shall use the logic $\mathrm{FO}_{\mathrm{REG}}$ as a symbolic framework to manipulate and analyse the abstract domain, see Section 2.2.3 for an exposition of this logic. Since we will mainly be concerned with the alphabets comprised of bit-vectors, we shall write $\{\mathtt{0}, \mathtt{1}\}^m$ as Σ_m,

and $\mathfrak{U}_{\Sigma_m}$ as $\mathfrak{S}_m$ for convenience. Also, we extend each structure $\mathfrak{S}_m$ with m binary relations $L_1, \ldots, L_m \subseteq \mathbb{N} \times \Sigma_m^*$ such that $\mathfrak{S}_m \models L_k(i, w)$ iff (i) $1 \leq i \leq |w|$, and (ii) the k-th bit of the bit-vector $w[i]$ is $\texttt{1}$. Note that these relations do not increase the expressive power of $\mathfrak{S}_m$. Indeed, one can easily check that they are regular relations under unary encoding of natural numbers.

In our prototype implementation, we interpret $\mathfrak{S}_m$ in $\mathfrak{S}_1$ to reduce the alphabet size, since the regular model checkers we use in the evaluation tend to perform better on smaller alphabets. Our interpretation of $\mathfrak{S}_m$ in $\mathfrak{S}_1$ is almost the same as the one we have demonstrated in the proof of Proposition 2.2.1. The only difference is that here we specifically define the encoding $\eta : \Sigma_m^* \to (\Sigma_1^m)^*$ as $a_1 \cdots a_n \mapsto a_1[1] \cdots a_1[m] \cdots a_n[1] \cdots a_n[m]$. Conceptually, if we regard a word in Σ_m^* as a list of bit-vectors, then the encoding "flattens" the bit-vectors to obtain a list of bits in Σ_1^*. For the atomic relations $L_1, \ldots, L_m$, note that $\mathfrak{S}_m \models L_k(i, w)$ holds if and only if $\mathfrak{S}_1 \models \eta(w)[\eta(i) - m + k] = \texttt{1}$. As a result, we can replace each occurrence of $L_k(i, w)$ with the expression $w[m \cdot (i-1) + k] = \texttt{1}$ in the transformed formula. In Section A.2, we give more details about the word-level encoding we have adopted in our implementation for regular abstractions.

5.4 Computation of regular abstractions

In this section, we provide two methods to mechanically compute regular abstractions for singly indexed array formulae. As before, we shall fix a set of variables V with $r := |V_{int}|$, and a set of indexed predicates $\mathbb{P} := \langle P_1, \ldots, P_m \rangle$ over V throughout this section.

Normalisation We say that an indexed array formula ϕ over variables V is *normalised* if (i) ϕ is in prenex normal form, (ii) each value expression in ϕ has exactly one quantified index variable, and (iii) ϕ is in form of $\ldots \forall i \ldots a[i] \ldots$ for some array variable a. An SIA formula can be syntactically normalised to a logically equivalent SIA formula. For

instance, if a formula ϕ does not meet the third condition, we can simply rewrite it to an equivalent formula in form of $\forall i.\, \psi \wedge a[i] = a[i]$ with a fresh index variable i and an arbitrary array variable $a \in V$. In the sequel, we shall always assume that the given formula is normalised when we are computing a regular abstraction for an SIA formula.

5.4.1 Abstractions for state formulae

A relation-based abstraction procedure Suppose that ϕ is a singly indexed formula over variables V. The idea of the procedure is to compute a regular abstraction for ϕ by replacing the matrix ψ of ϕ with a suitable quantifier-free formula ψ^* defined over structure $\mathfrak{S}_m$. To this end, fix a fresh word variable w over domain Σ_m^*. We first convert the matrix of ϕ to DNF in form of $\bigvee_j (g_j \wedge h_j)$, such that each g_j is a conjunction of index expressions, and each h_j is a conjunction of value expressions. We then replace each disjunct $g \wedge h$ in the DNF with the formula

$$\tau := g \wedge R(w[t_1], \ldots, w[t_d]), \tag{5.7}$$

where $t_1, \ldots, t_d$ are the index terms appearing in g, h, and $\mathbb{P}$, and R is a finite relation

$$R := \{ (v_1, \ldots, v_d) \in \Sigma_m^d : g \wedge h \wedge \phi_{v_1}^{\mathbb{P}}[t_1/i] \wedge \cdots \wedge \phi_{v_d}^{\mathbb{P}}[t_d/i] \text{ is } T_A\text{-satisfiable} \} \tag{5.8}$$

with the $\phi_v^{\mathbb{P}}$'s defined at (5.1). Note that R can be effectively computed as the quantifier-free fragment of T_A is decidable by assumption. After all disjuncts in the DNF are replaced in this way, the resulting formula, denoted by ϕ^*, is a formula in $\mathrm{FO}(\mathfrak{S}_m)$ over variables $V_{int} \uplus \{w\}$. Now we set $\tilde{\phi}(\overline{p}, w) := \phi^*(\overline{p}, w) \wedge \lambda(\overline{p}, w)$, where $\{\overline{p}\} = V_{int}$ and

$$\lambda(\overline{p}, w) := \Big(\bigwedge\nolimits_{p \in \{\overline{p}\}} p \le |w| \Big) \wedge \Big(\bigvee\nolimits_{p \in \{\overline{p}\}} p = |w| \Big) \tag{5.9}$$

is a length constraint specifying that $|w|$ should equal to the largest value of the index variables in V. We then use $(\tilde{\phi}^{\mathfrak{S}_m}, \mathbb{P})$ as a regular abstraction for the state formula ϕ.

It is possible to simplify the computation of abstraction for formulae in special forms. For example, suppose that the matrix of ϕ is in form of $\bigwedge_j (g_j \Rightarrow h_j)$ rather

than $\bigvee_j (g_j \wedge h_j)$. In this case, we can directly compute a regular abstraction for ϕ by replacing each conjunct $g \Rightarrow h$ with

$$g \Rightarrow R(w[t_1], \ldots, w[t_d]). \tag{5.10}$$

As before, $t_1, \ldots t_d$ are index terms contained in g, h, and $\mathbb{P}$, while R is a finite relation as defined at (5.8). Note that the term $R(w[t_1], \ldots, w[t_d])$ can be encoded over structure $\mathfrak{S}_m$ (see Example 5.4.1). After all conjuncts $g \Rightarrow h$ in ϕ are replaced, we will obtain a formula defined over structure $\mathfrak{S}_m$ and variables $V_{int} \uplus \{w\}$. Denote this formula by $\phi^*(\overline{p}, w)$, and let $\tilde{\phi}(\overline{p}, w) := \phi^*(\overline{p}, w) \wedge \lambda(\overline{p}, w)$ such that $\lambda(\overline{p}, w)$ is the length constraint defined at (5.9). We then use $(\tilde{\phi}^{\mathfrak{S}_m}, \mathbb{P})$ as a regular abstraction for the formula ϕ.

Example 5.4.1. As an example, let us instantiate the element theory of T_A to equality theory[1] and compute a regular abstraction for the SIA formula

$$\phi := |a| \geq 2 \wedge \forall i. \, (i = 1 \Rightarrow a[i] \neq a[|a|]) \wedge (i > 1 \wedge i \leq |a| \Rightarrow a[i] = a[i-1])$$

over variables $V := \{a, |a|\}$ with respect to predicates $\mathbb{P} := \langle i = |a|, \, a[i] = a[i-1], \, a[i] = a[|a|] \rangle$. Let $m := |\mathbb{P}| = 3$. The abstraction formula $\tilde{\phi} = \tilde{\phi}(|a|, w)$ computed by the relation-based method can be written as $\phi^*(|a|, w) \wedge |a| \leq |w| \wedge |a| = |w|$, where

$$\begin{aligned} \phi^*(|a|, w) \quad := \quad & \forall i. \, (i = 1 \wedge |a| \geq 2 \Rightarrow R_1(w[i], w[i-1], w[|a|])) \\ & \wedge \, (i > 1 \wedge i \leq |a| \wedge |a| \geq 2 \Rightarrow R_2(w[i], w[i-1], w[|a|])), \end{aligned}$$

and the two finite relations R_1 and R_2 are defined by

$$\begin{aligned} R_1 \quad := \quad & \{(x, y, z) \in \Sigma_m^3 : \neg x[1] \wedge \neg x[3] \wedge \neg y[1] \wedge z[1] \wedge z[3] \wedge \neg(x[2] \wedge y[3])\} \\ R_2 \quad := \quad & \{(x, y, z) \in \Sigma_m^3 : \neg y[1] \wedge z[1] \wedge z[3] \wedge x[2] \wedge (\neg x[1] \vee z[2]) \\ & \wedge \, (\neg x[1] \vee x[3]) \wedge (\neg x[3] \vee y[3]) \wedge (\neg y[3] \vee x[3])\}. \end{aligned}$$

The occurrences of R_1 and R_2 in ϕ^* can be formally encoded as propositional formulae over the index terms $i, i-1, |a|$ and the relation symbols $L_k(t, w)$'s. For example, the

[1]Technically, the element and the index theory should use different symbols to denote the equality relation. To avoid notational clutter, we shall just use the same symbol "=" for equality in both theories.

literal $R_1(w[i], w[i-1], w[|a|])$ in ϕ^* can be encoded as

$$\neg L_1(i,w) \wedge \neg L_3(i,w) \wedge \neg L_1(i-1,w) \wedge L_1(|a|,w) \wedge L_3(|a|,w) \wedge \neg(L_2(i,w) \wedge L_3(i-1,w)).$$

The abstraction formula $\tilde{\phi}$ will be expressible over structure $\mathfrak{S}_m$ after the occurrences of R_1 and R_2 in ϕ^* are replaced with their respective propositional encodings.

Interestingly, one can check that the language $\langle \tilde{\phi}^{\mathfrak{S}_m} \rangle$ is empty, e.g., by computing a finite automaton recognising $\langle \tilde{\phi}^{\mathfrak{S}_m} \rangle$. Since regular abstractions are over-approximations, we can conclude that the formula ϕ is T_A-unsatisfiable. #

A rule-based abstraction procedure In the relation-based method, computing a regular abstraction for ϕ requires computing several finite relations as defined at (5.8). These relations effectively contain all the feasible valuations of the predicates $\mathbb{P}$ at the positions represented by the relevant index terms in ϕ and $\mathbb{P}$. In the worst case, the sizes of the relations (and the regular abstraction so obtained) can be exponential in $|\mathbb{P}| \cdot |\phi|$. Practically, however, it is often possible to achieve a sufficiently precise abstraction by specifying a set of constraints (i.e. necessary conditions) for the feasible valuations without explicitly computing these relations. We discuss how to specify such constraints in this subsection.

Fix a state formula ϕ as before, and suppose that the matrix of ϕ is expressed in form of $\bigwedge_j (g_j \Rightarrow h_j)$ such that each g_j is a conjunction of index expressions, and each h_j is a conjunction of value expressions. (We omit the case where ϕ is in form of $\bigvee_j (g_j \wedge h_j)$ as it can be handled in a very similar manner.) Consider a conjunct $g \Rightarrow h$ of ϕ. Let Π denote the set of index terms contained in the formula $g \Rightarrow h$ and the predicates $\mathbb{P}$. Define $\tilde{\mathbb{P}} := \mathbb{P} \uplus \{\neg P_k : P_k \in \mathbb{P}\}$.

First, we compute a formula τ^1 by conjuncting the literals in the set

$$\{\tilde{P}_k[t/i] : \tilde{P}_k \in \tilde{\mathbb{P}},\ t \in \Pi,\ \text{and } g \wedge h \Rightarrow \tilde{P}_k[t/i] \text{ is } T_A\text{-valid}\}. \tag{5.11}$$

Note that τ^1 is computable, since $\tilde{\mathbb{P}}$ and Π are finite sets, and $g \wedge h \Rightarrow \tilde{P}_k$ is T_A-valid if and only if $g \wedge h \wedge \neg\tilde{P}_k$ is not T_A-satisfiable. This satisfiability checking is algorithmic as T_A has a decidable quantifier-free theory by assumption.

$$\frac{t_1 = t_2}{a[t_1] = a[t_2]} \qquad \frac{t_1 = t_2 \qquad a_1[t_1] \sim a_3[t_3]}{a_1[t_2] \sim a_3[t_3]} \qquad \frac{a_1[t_1] = a_2[t_2] \qquad a_1[t_1] \sim a_3[t_3]}{a_2[t_2] \sim a_3[t_3]}$$

Figure 5.1 A selection of inference rule templates for state formulae, where $\sim \in \{=, \leq, \geq, \neq\}$.

Second, we compute a formula τ^2 by conjuncting adequate inference rules. An *inference rule* is a formula in form of $(\bigwedge_j \phi_j) \Rightarrow \psi$, where ϕ_j's are literals called the *premises*, and ψ is a literal called the *consequence*. We say that a literal is *expressible in* $\mathbb{P}$ if either it is an index expression where each variable is contained in V, or it is a value expression in form of $\tilde{P}_k[t/i]$ for some $\tilde{P}_k \in \tilde{\mathbb{P}}$ and $t \in \Pi$. An inference rule is *adequate* if (i) the rule is T_A-valid; (ii) each literal occurring in the premises is either contained in $g \wedge h$ or expressible in $\mathbb{P}$; (iii) the consequence is expressible in $\mathbb{P}$. The formula τ^2 is then defined as the conjunction of a selective set of adequate rules.

Finally, we remove from $\tau^1 \wedge \tau^2$ the literals not expressible in $\mathbb{P}$, and substitute $L_k(t, w)$ for $P_k[t/i]$ for each $k \in \{1, \ldots, m\}$ and $t \in \Pi$. The resulting formula, denoted by f, is defined over structure $\mathfrak{S}_m$ and variables $\{\overline{p}, w\}$ with $\{\overline{p}\} := V_{int}$. We then replace the conjunct $g \Rightarrow h$ in ϕ with the formula $g \Rightarrow f$. The rest of the construction is the same as the relation-based method: we let $\phi^*(\overline{p}, w)$ denote the resulting formula after all such conjuncts in ϕ are replaced, and define $\tilde{\phi}(\overline{p}, w) := \phi^*(\overline{p}, w) \wedge \lambda(\overline{p}, w)$ with $\lambda(\overline{p}, w)$ being the length constraint at (5.9). We then use $(\tilde{\phi}^{\mathfrak{S}_m}, \mathbb{P})$ as a regular abstraction for ϕ.

Example 5.4.2. Recall from Example 5.4.1 the formula

$$\phi := |a| \geq 2 \wedge \forall i.\, (i = 1 \Rightarrow a[i] \neq a[|a|]) \wedge (i > 1 \wedge i \leq |a| \Rightarrow a[i] = a[i-1]).$$

and the predicates $\mathbb{P} := \langle i = |a|,\, a[i] = a[i-1],\, a[i] = a[|a|] \rangle$. Using the rule-based method, the abstraction formula $\tilde{\phi} = \tilde{\phi}(|a|, w)$ can be written as $\phi^*(|a|, w) \wedge |a| \leq |w| \wedge |a| = |w|$, where

$$\phi^*(|a|, w) := \forall i.\, (i = 1 \wedge |a| \geq 2 \Rightarrow f_1(i, |a|, w)) \wedge (i > 1 \wedge i \leq |a| \wedge |a| \geq 2 \Rightarrow f_2(i, |a|, w)).$$

We can compute the two constraints f_1 and f_2 in ϕ^* by instantiating the templates in Figure 5.1. To compute constraint f_1, we instantiate the first templates to obtain the adequate rule

$$\frac{|a| = |a|}{a[|a|] = a[|a|]}$$

Hence we have $\tau^1 \wedge \tau^2 = (a[i] \neq a[|a|]) \wedge (|a| = |a| \Rightarrow a[|a|] = a[|a|])$ for f_1, leading to

$$f_1(i, |a|, w) := \neg L_3(i, w) \wedge (|a| = |a| \Rightarrow L_3(|a|, w)).$$

Similarly, we can derive f_2 by instantiating the first and the last templates in Figure 5.1 to obtain the adequate rules

$$\frac{|a| = |a|}{a[|a|] = a[|a|]} \qquad \frac{a[i] = a[i-1] \qquad a[i] = a[|a|]}{a[i-1] = a[|a|]}$$

Constraint f_2 hence can be computed as

$$f_2(i, |a|, w) := L_2(i, w) \wedge (|a| = |a| \Rightarrow L_3(|a|, w)) \wedge (L_2(i, w) \wedge L_3(i, w) \Rightarrow L_3(i-1, w)).$$

Now, by plugging the constraints f_1 and f_2 into the definition of ϕ^*, we obtain

$$\begin{aligned} &\forall i.\, (i = 1 \wedge |a| \geq 2 \Rightarrow (\neg L_3(i, w) \wedge L_3(|a|, w))) \\ &\quad \wedge (i > 1 \wedge i \leq |a| \wedge |a| \geq 2 \Rightarrow (L_2(i, w) \wedge L_3(|a|, w) \wedge (L_2(i, w) \wedge L_3(i, w) \Rightarrow L_3(i-1, w)))) \end{aligned}$$

for ϕ^* after simplification. Although the regular abstraction obtained in this way is much coarser than the one computed by the relation-based method in Example 5.4.1, it still yields an empty regular language and hence is sufficient to prove that the formula ϕ is T_A-unsatisfiable. #

We summarise two important facts about our abstraction methods below.

Theorem 5.4.1. *Let ϕ be a normalised SIA state formula. Then the following statements hold:*

1. *Both the relation-based and the rule-based method compute a regular abstraction for ϕ.*

2. *A regular abstraction computable by the relation-based method is also computable by the rule-based method given a suitable set of inference rules.*

Proof. The first fact follows by construction since both the relation-based method and the rule-based method aim to compute a necessary condition of ϕ in the predicate language (i.e. the SIA formulas whose value expressions are expressible in the given predicates). In other words, both methods compute the abstraction formula based on a formula ψ in the predicate language such that $\phi \Rightarrow \psi$ is T_A-valid. Therefore, the regular relation induced by the abstraction formula over-approximates $\alpha(\phi^{\mathfrak{A}})$ for any model $\mathfrak{A}$ of T_A. Finally, the length constraint introduced at (5.9) assures that the induced regular relation is appropriate, thereby yielding a regular abstraction for ϕ.

To prove the second fact, suppose without loss of generality that the matrix of ϕ is in form of $\bigwedge_j (g_j \Rightarrow h_j)$. Then it suffices to show that for any conjunct $g \Rightarrow h$, the formula $\tau^1 \wedge \tau^2$ in the rule-based method can be computed in a way to capture the semantics of $R(w[t_1], \ldots, w[t_d])$ at (5.10). Observe that $R(w[t_1], \ldots, w[t_d])$ can be encoded as a propositional formula ϕ_R such that each value expression in ϕ_R is in form of either $P[t_k/i]$ or $\neg P[t_k/i]$ for some $P \in \mathbb{P}$ and $k \in \{1, \ldots, d\}$. Convert ϕ_R into CNF and let C be any of its clauses. There are two cases:

Case 1: $C = \{l\}$, i.e., C is a singleton. Then $g \wedge h \Rightarrow l$ is T_A-valid, meaning that l will be contained in the set defined at (5.11), and hence be included in the conjunctive formula τ^1 as a conjunct.

Case 2: $C = \{l_1, \ldots, l_k\}$ for some $k \geq 2$. Then we can include an adequate inference rule $\neg l_1 \wedge \cdots \wedge \neg l_{k-1} \Rightarrow l_k$ in the conjunction τ^2 of inference rules.

Consequently, for each finite relation R computed by the relation-based method, we can choose a finite set of adequate rules such that the resulting $\tau^1 \wedge \tau^2$ is T_A-equivalent to ϕ_R. With these inference rules, the rule-based method will compute precisely the same regular abstraction as the outcome of the relation-based method. #

$$\frac{t_1 = t_2 \qquad a'[t_1] \sim a[t_1]}{a'[t_2] \sim a'[t_2]} \qquad \frac{a_1[t_1] \sim a_3[t_3] \qquad a'_2[t_2] = a_1[t_1] \qquad a'_3[t_3] = a_3[t_3]}{a'_2[t_2] \sim a'_3[t_3]}$$

$$\frac{a'_1[t_1] \sim a_3[t_3] \qquad a'_2[t_2] = a_3[t_3]}{a'_1[t_1] \sim a'_2[t_2]} \qquad \frac{t_1 = t_2 \qquad a'_2[t_2] \sim a_3[t_3] \qquad a'_3[t_3] = a_3[t_3]}{a'_1[t_1] \sim a'_3[t_3]}$$

Figure 5.2 A selection of inference rule templates for transition formulae, where $\sim \in \{=, \leq, \geq, \neq\}$.

5.4.2 Abstractions for transition formulae

Now suppose that ϕ is a transition formula with free variables $V \uplus V'$. Let $\mathbb{Q} := \langle Q_1, \ldots, Q_n \rangle$ be the set of the indexed predicates $\{a'[i] = a[i] : a \in V_{arr}\}$. As before, we assume that the matrix of ϕ is in form of $\bigvee_j (g_j \wedge h_j)$ such that each g_j is a conjunction of index expressions and each h_j is a conjunction of value expressions. The case where the matrix is in form of $\bigvee_j (g_j \Rightarrow h_j)$ can be handled in a similar manner. We first introduce a relation-based method. Fix fresh word variables w, w' and u such that w, w' are over domain Σ_m^* and u is over domain Σ_n^*. (Recall that $m = |\mathbb{P}|$ and $n = |\mathbb{Q}|$.) We replace each disjunct $g \wedge h$ in the DNF of ϕ with

$$g \wedge R(w[t_1], \ldots, w[t_d];\, w'[t_1], \ldots, w'[t_d];\, u[t_1], \ldots, u[t_d]),$$

where $t_1, \ldots, t_d$ are the index terms contained in g, h, and $\mathbb{P}$, and R is a finite relation

$$\begin{aligned} R \;\; := \;\; & \{(v_1, \ldots, v_d;\, u_1, \ldots, u_d;\, w_1, \ldots, w_d) \in (\Sigma_m^d) \times (\Sigma_m^d) \times (\Sigma_n^d) : \\ & \quad g \wedge h \wedge \bigwedge\nolimits_{k=1}^{d} \left(\phi_{v_k}^{\mathbb{P}}[t_k/i] \wedge \phi_{u_k}^{\mathbb{P}'}[t_k/i] \wedge \phi_{w_k}^{\mathbb{Q}}[t_k/i] \right) \text{ is } T_A\text{-satisfiable} \} \end{aligned}$$

with the $\phi_v^{\mathbb{P}}$'s being defined at (5.1). After all disjuncts in the matrix are replaced in this way, the resulting formula will be defined over $\mathfrak{S}_m$ and variables $V_{int} \uplus V'_{int} \uplus \{w, w', u\}$. Write this formula as $\phi^\star(\overline{p}, \overline{p}', w, w', u)$ and define

$$\tilde{\phi}((\overline{p}, w), (\overline{p}', w')) := \left(\exists u \in \Sigma_n^*.\ \phi^\star(\overline{p}, \overline{p}', w, w', u)\right) \wedge \lambda(\overline{p}, w, w'), \tag{5.12}$$

where $\lambda(\overline{p}, w, w')$ is the usual length constraint

$$\lambda(\overline{p}, w, w') := \left(\bigwedge_{p \in \{\overline{p}\}} p \leq |w|\right) \wedge \left(\bigvee_{p \in \{\overline{p}\}} p = |w|\right) \wedge \left(|w| = |w'|\right). \tag{5.13}$$

We then use $(\tilde{\phi}^{\mathfrak{S}_m}, \mathbb{P})$ as a regular abstraction for the transition formula ϕ.

The rule-based abstraction method for a transition formula is analogous to the counterpart rule-based method for a state formula. The only difference is that, when we are instantiating a literal of the form $a'[t] = a[t]$ in a template, we need to replace it with the literal $Q_k[t/i]$ such that $Q_k = (a'[i] = a[i])$ is an indexed predicate in $\mathbb{Q}$. Similarly, to instantiate a literal of the form $a'[t] \neq a[t]$, we need replace it with the literal $\neg Q_k[t/i]$. The rest of the abstraction procedure is the same as the rule-based method for a state formula.

Example 5.4.3. Consider the following transition formula, which is generated from the guarded command Update_3 in the specification of Dijkstra's algorithm in Section 5.6.2:

$$\begin{aligned} &\forall i.\,(i = pid \wedge x[i] \neq x[i-1] \wedge x'[i] = x[i-1] \wedge pid > 1 \wedge 1 \leq pid' \wedge pid' \leq |x|) \\ &\quad \vee (i \neq pid \wedge 1 \leq i \wedge i \leq |x| \wedge x'[i] = x[i] \wedge pid > 1 \wedge 1 \leq pid' \wedge pid' \leq |x|) \end{aligned}$$

Let $\mathbb{P} := \langle i = |x|,\ x[i] = x[i-1],\ x[i] = x[|x|]\rangle$ and $\mathbb{Q} := \langle x'[i] = x[i]\rangle$. The rule-based regular abstraction for the transition formula is $\tilde{\phi}((pid, |x|, w), (pid', |x'|, w'))$, which is defined as

$$\begin{aligned} &(\exists u \in \Sigma_1^*.\,\forall i.\,(i = pid \wedge pid > 1 \wedge 1 \leq pid' \wedge pid' \leq |x| \wedge f_1(i, |x|, w, w', u)) \\ &\quad \vee (i \neq pid \wedge 1 \leq i \wedge i \leq |x| \wedge pid > 1 \wedge 1 \leq pid' \wedge pid' \leq |x| \wedge f_2(i, |x|, w, w', u)) \\ &\hspace{30em} (5.14) \\ &\wedge (|x| \leq |w| \wedge pid \leq |w|) \wedge (|x| = |w| \vee pid = |w|) \wedge (|w| = |w'|). \end{aligned}$$

To derive the constraint f_1, we instantiate the templates in Figure 5.2 to obtain the rules

$$\frac{x[i] \neq x[i-1] \qquad x'[i] = x[i-1]}{x'[i] \neq x[i]}$$

$$\frac{x[i-1] = x[|x|] \qquad x'[i] = x[i-1] \qquad x'[|x|] = x[|x|]}{x'[i] = x'[|x|]}$$

$$\frac{i = |x| \qquad x'[i] = x[i-1] \qquad x'[i-1] = x[i-1]}{x'[i-1] = x'[|x|]}$$

We then conjunct these rules to obtain a constraint

$$\begin{aligned}(\neg L_2(i,w) \Rightarrow \neg L_1(i,u)) \wedge (L_3(i-1,w) \wedge L_1(|x|,u) \Rightarrow L_3(i,w'))\\ \wedge\ (i = |x| \wedge L_1(i-1,u) \Rightarrow L_3(i-1,w'))\end{aligned}$$

for $f_1(i,|x|,w,w',u)$ after simplification. In a similar manner, we can use the rules

$$\frac{x[i] = x[i-1] \qquad x'[i] = x[i] \qquad x'[i-1] = x[i-1]}{x'[i] = x'[i-1]}$$

$$\frac{x[i] = x[|x|] \qquad x'[i] = x[i] \qquad x'[|x|] = x[|x|]}{x'[i] = x'[|x|]}$$

to compute a constraint

$$L_1(i,u) \wedge (L_2(i,w) \wedge L_1(i-1,u) \Rightarrow L_2(i,w')) \wedge (L_3(i,w) \wedge L_1(|x|,u) \Rightarrow L_3(i,w'))$$

for the formula $f_2(i,|x|,w,w',u)$ after simplification. The final abstraction formula $\tilde{\phi}$ is obtained by substituting the constraints derived as above for the occurrences of f_1 and f_2 at (5.14). #

The following result is the counterpart of Theorem 5.4.1 for transition formulae and can be proved analogously.

Theorem 5.4.2. *Let $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ be an index-bounded array system, and suppose that ϕ_{tran} is normalised. Then the following statements hold:*

1. *Both the relation-based and the rule-based method compute a regular abstraction for ϕ_{tran}.*
2. *A regular abstraction computable by the relation-based method is also computable by the rule-based method given a suitable set of inference rules.*

5.5 Verification of temporal array properties

In this section, we discuss how to verify linear-time properties of index-bounded array systems in the framework of regular abstractions. For this purpose, we first introduce the notion of abstractable specifications. We then present two verification methods for dealing with standard safety and liveness properties. Finally, we sketch a general proof technique that reduces a monodic temporal formula [HWZ00] to the fair termination property.

Definition 5.5.1 (Abstractable specification)**.** An indexed array formula is *abstractable* if it is in form of $\exists \overline{x}.\, \phi$ such that ϕ is an singly indexed formula. A first-order array system is *abstractable* if it is specified with abstractable formulae. A temporal array property ϕ is *abstractable* if it is constrained by the grammar

$$\phi ::= \psi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists x.\, \phi \mid \Box \phi \mid \Diamond \phi$$

such that ψ is a singly indexed formula. Note that ψ itself is allowed to be quantified. #

The specification of an abstractable array system $\mathfrak{T}$ can be syntactically transformed into singly index formulae using *Skolemisation*. More precisely, suppose that $\phi := \exists \overline{x}.\, \psi$ is an array formula over variables V, and ψ is an SIA formula over variables $V \uplus \{\overline{x}\}$. Let $\psi^\star := \psi[\overline{c}/\overline{x}]$ be an SIA formula over variables $V \uplus \{\overline{c}\}$, where $\overline{c}$ are fresh system parameters. We can then add the new parameters $\overline{c}$ to the vocabulary of $\mathfrak{T}$ and substitute $\psi^\star$ for ϕ in the specification. It is easy to see that a temporal array property holds for the array system $\mathfrak{T}$ if and only if the property holds for the Skolemised version of $\mathfrak{T}$.

An abstractable temporal array property can furthermore be converted to an equi-satisfiable temporal array property wherein all maximal non-temporal subformulae are singly indexed. The conversion is essentially the same as Skolemisation, except that here we replace an indexed variable quantified inside a globally operator with a fresh index variable, and replace an indexed variable that is *only* quantified inside an eventually operator with a fresh system parameter. The maximal non-temporal subformulae of the transformed temporal array property are all singly indexed, meaning that their semantics can be over-approximated by regular abstractions. In other words, an abstractable temporal array property can be abstracted to a propositional temporal formula with atoms being regular languages. This fact makes it possible to leverage existing verification techniques in regular model checking to perform abstract analysis for abstractable specifications and properties.

5.5.1 Specific method for safety and liveness properties

A *safety array property* is a temporal array property in form of $\forall \bar{x}. (\psi_1 \Rightarrow \Box\, \psi_2)$. Fix a safety array property ϕ and an index-bounded array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$. Suppose that both $\mathfrak{T}$ and $\neg\phi \equiv \exists \bar{x}. (\psi_1 \wedge \Diamond \neg\psi_2)$ are abstractable. Define $\psi_1^\star := \psi_1[\bar{c}/\bar{x}]$ and $\psi_2^\star := \psi_2[\bar{c}/\bar{x}]$ with fresh system parameters $\bar{c}$. Moreover, define a safe transition system $\mathfrak{T}^\star := (V^\star, \phi_{init}^\star, \phi_{tran}, \phi_{bad})$ with $V_{par}^\star := V_{par} \uplus \{\bar{c}\}$, $V_{st}^\star := V_{st}$, $\phi_{init}^\star := \phi_{init} \wedge \psi_1^\star$, and $\phi_{bad} := \neg\psi_2^\star$. It is clear that $\mathfrak{T}^\star$ is index-bounded and abstractable. Given a set of indexed predicates $\mathbb{P}$ over $V^\star$, we can compute regular abstractions $(Init, \mathbb{P})$, $(Tran, \mathbb{P})$, and $(Bad, \mathbb{P})$ of $\phi_{init}^\star$, ϕ_{tran}, and ϕ_{bad}, respectively, as described in Section 5.4. By Theorem 2.2.2, we can furthermore compute finite automata $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{B}$ recognising the languages of $Init$, $Tran$, and Bad, respectively. Particularly, since $\mathfrak{T}^\star$ is index-bounded, $\mathcal{T}$ is a length-preserving finite transducer. It is easy to see that the regular safety property $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ holds only if $\mathfrak{T}$ satisfies the array property ϕ. We summarise this result in the following theorem.

Theorem 5.5.1. *Let $\mathfrak{T}$ be an index-bounded array system, and ϕ be a safety array property. Suppose that $\mathfrak{T}$ and $\neg\phi$ are abstractable. Given a set of indexed predicates, we can effectively*

compute finite automata $\mathscr{I}$, $\mathscr{T}$, and $\mathscr{B}$ such that ϕ holds for $\mathfrak{T}$ if the safety property $(\mathscr{I},\mathscr{T},\mathscr{B})$ holds.

A *liveness array property* is a temporal array property in form of $\forall\bar{x}.\,(\psi_1 \Rightarrow \Diamond\,\psi_2)$. Fix a liveness array property ϕ and an index-bounded array transition system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$. We say that $\mathfrak{T}$ is *non-blocking with respect to* $\neg\phi$ if every maximal execution run of $\mathfrak{T}$ satisfying $\neg\phi$ is infinite. Suppose that both $\mathfrak{S}$ and $\neg\phi \equiv \exists\bar{x}.\,(\psi_1 \wedge \Box\,\neg\psi_2)$ are abstractable. Define $\psi_1^\star := \psi_1[\bar{c}/\bar{x}]$ and $\psi_2^\star := \psi_2[\bar{c}/\bar{x}]$ with fresh system parameters $\bar{c}$, and specify an induced transition system $\mathfrak{T}^\star := (V^\star, \phi_{init}^\star, \phi_{tran}, \phi_{fin})$, where $V^\star := V \uplus \{\bar{c}\}$, $\phi_{init}^\star := \phi_{init} \wedge \psi_1^\star$, and $\phi_{fin} := \psi_2^\star$. It is clear that $\mathfrak{T}^\star$ is index-bounded and abstractable. Given a set of indexed predicates $\mathbb{P}$ over $V^\star$, we can compute regular abstractions for $\phi_{init}^\star$, ϕ_{tran}, and $\neg\phi_{fin}$ as described in Section 5.4. By Theorem 2.2.2, we can furthermore compute finite automata $\mathscr{I}$ and $\mathscr{F}$, as well as a length-preserving finite transducer $\mathscr{T}$, such that $\mathscr{I}$ and $\mathscr{T}$ recognise the regular abstractions of $\phi_{init}^\star$ and ϕ_{tran}, respectively, and $\mathscr{F}$ recognises the complement of the regular abstraction for $\neg\phi_{fin}$. Now we define a regular liveness property $(\mathscr{I},\mathscr{T},\mathscr{F})$. Observe that if $\mathfrak{T}$ is non-blocking with respect to $\neg\phi$, then $(\mathscr{I},\mathscr{T},\mathscr{F})$ holds *only* when the liveness array property ϕ holds for $\mathfrak{T}$. Particularly, any maximal execution run of $\mathscr{T}$ violating ϕ will lead to a counterexample run violating $(\mathscr{I},\mathscr{T},\mathscr{F})$. We summarise this result in the following theorem.

Theorem 5.5.2. *Let $\mathfrak{T}$ be an index-bounded array system, and ϕ be a liveness array property. Suppose that $\mathfrak{T}$ and $\neg\phi$ are abstractable, and that $\mathfrak{T}$ is non-blocking with respect to $\neg\phi$. Then given a set of indexed predicates, we can effectively compute finite automata $\mathscr{I}$, $\mathscr{T}$, and $\mathscr{F}$ such that ϕ holds for $\mathfrak{T}$ if the liveness property $(\mathscr{I},\mathscr{T},\mathscr{F})$ holds.*

Theorems 5.5.1 and 5.5.2 in fact demonstrate the same abstraction technique that over-approximates the desired temporal property for an array system by a counterpart temporal property for a regular transition system. Generally, this technique works for any temporal array properties with abstractable negated formulae as long as a false property remains false after new transitions are introduced to the array system. For

temporal properties involving eventuality, this requires that the array system has no finite maximal counterexample runs, namely, the system is non-blocking with respect to the negation of the property. As might be expected, checking the fulfilment of a non-blocking condition is undecidable for general array systems and properties. It nevertheless is possible to formulate and verify the non-blocking condition as a safety array property as follows. Consider a liveness array property $\phi := \forall \overline{x}. (\psi_1 \Rightarrow \Diamond \psi_2)$ and an array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$. Suppose that we would like to check whether or not $\mathfrak{T}$ is non-blocking with respect to $\neg\phi$. As before, define $\psi_1^\star := \psi_1[\overline{c}/\overline{x}]$ and $\psi_2^\star := \psi_2[\overline{c}/\overline{x}]$ with fresh system parameters $\overline{c}$. We then specify a transition system $\mathfrak{T}^\dagger := (V^\star, \phi_{init}^\star, \phi_{tran}^\star, \phi_{bad})$ where

$$
\begin{aligned}
V^\star &:= V \uplus \{\overline{c}\} \\
\phi_{init}^\star &:= \phi_{init} \wedge \psi_1^\star \\
\phi_{tran}^\star &:= \phi_{tran} \wedge \neg\psi_2^\star \\
\phi_{bad} &:= (\forall \overline{x}'. \neg\phi_{tran}) \wedge \neg\psi_2^\star
\end{aligned}
$$

Here, $\forall \overline{x}'. \neg\phi_{tran}$ is the formula obtained by universally quantifying $\neg\phi_{tran}$ over all variables in V'. (Recall that $\neg\phi_{tran}$ is a formula over variables $V \uplus V'$.) It is easy to see that the system $\mathfrak{T}^\dagger$ is safe if and only if $\mathfrak{T}$ is non-blocking with respect to $\neg\phi$. Therefore, when $\mathfrak{T}^\dagger$ is abstractable, the non-blocking condition can be verified by regular model checking using our abstraction method. In Section A.3, we give a modelling language for array systems that always induces an abstractable system $\mathfrak{T}^\dagger$ for non-blocking checking.

5.5.2 Liveness verification under fairness requirements

We have introduced how to verify the typical safety and liveness properties for array systems by combining our abstraction methods with regular model checking. We now proceed to discuss how to deal with liveness verification in the presence of fairness requirements for an index-bounded array system.

Definition 5.5.2 (Fairness specification)**.** A *fairness specification* is a temporal array property ϕ containing a designated index variable i, and constrained by the grammar

$$\phi ::= \Box\Diamond\,\psi \mid \Diamond\Box\,\psi \mid \phi \vee \phi \mid \phi \wedge \phi,$$

where ψ is an abstractable indexed array formula. Let $\mathfrak{T}$ be an index-bounded array system defined over V, ϕ be a fairness specification, and $p \in V_{par}$ be a designated parameter. Given a model $\mathfrak{A}$ of T_A, an execution run $\pi := s_0\, s_1 \cdots$ is *fair* if $\mathfrak{A}, \pi \models \phi[t/i]$ for $1 \le t \le s_0(p)$ (recall that p is immutable alone an execution run). A temporal array property λ holds for an array system $\mathfrak{T}$ under fairness specification ϕ if $\mathfrak{T}$ does not have an fair execution run satisfying $\neg\lambda$. #

Fix a fairness specification ϕ and a set of indexed predicates $\mathbb{P} := \langle P_1, \ldots, P_m\rangle$. We shall compute a regular abstraction for ϕ such that the abstraction can serve as a regular encoding of fairness in the sense of Definition 4.3.2. This regular encoding will over-approximate ϕ in the abstract domain using a fairness requirement $\Phi := \{\phi_i : i \ge 1\}$, where each ϕ_t essentially represents a regular abstraction for the formula $\phi[t/i]$.

More precisely, we compute a regular abstraction for ϕ by structural induction as follows. Consider the base cases $\phi = \Box\Diamond\,\psi$ and $\phi = \Diamond\Box\,\psi$. Since ψ is abstractable, we can obtain a regular abstraction $(R, \mathbb{P})$ of ψ using Skolemisation and the abstraction methods described in Section 5.4. Without loss of generality, we set the valuation of the index variable i to be the first component of the tuples in R. Then the language representation of R can be written as $A := \bigcup_{i\ge 1} A_i \subseteq \Sigma_n^*$ for some $n > m$, with each $A_i := \mathtt{1}^i\mathtt{0}^* \odot B_i$ for a regular language B_i.

Case 1: $\phi = \Box\Diamond\,\psi$. We use $\Phi := \{\Box\Diamond\, A_i : i \ge 1\}$ as a regular abstraction for ϕ, and $(\mathtt{1}^*, \{w \odot a \in (\Sigma_n \times \Sigma_1)^* : w \in A_i \Leftrightarrow a[i] = \mathtt{1} \text{ for } i \in \{1, \ldots, |w|\}\})$ as the regular encoding of Φ.

Case 2: $\phi = \Diamond\Box\,\psi$. We use $\Phi := \{\neg\Box\Diamond\, A_i^c : i \ge 1\}$ as a regular abstraction for ϕ, and $(\mathtt{0}^*, \{w \odot a \in (\Sigma_n \times \Sigma_1)^* : w \notin A_i \Leftrightarrow a[i] = \mathtt{1} \text{ for } i \in \{1, \ldots, |w|\}\})$ as the regular encoding of Φ.

In both cases, the regular encoding is definable in $\mathrm{FO}_{\mathrm{REG}}$, and thus can be effectively

computed from A. By Proposition 4.3.1, regular encoding of fairness requirements is effectively closed under disjunctions and conjunctions. We can therefore compute a regular encoding of ϕ from the base cases as in the proof of Proposition 4.3.1. The resulting fairness requirement Φ, by Proposition 5.3.4, is a regular abstraction that over-approximates ϕ in the abstract domain. That is, if $s_0\, s_1\, s_2 \cdots$ is a concrete path that is fair with respect to ϕ, then there is an abstract path $t_0\, t_1\, t_2 \cdots$ that is fair with respect to Φ and $t_i \in \alpha(s_i)$ for each $i \geq 0$. This fact, together with Theorem 5.5.2, allows us to state the following result.

Theorem 5.5.3. *Let $\mathfrak{T}$ be an index-bounded array system, ϕ be a liveness array property, and ϕ be a fairness specification. Suppose that $\mathfrak{T}$ and $\neg\phi$ are abstractable. Then we can compute finite automata $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{F}$, as well as regular encoding $(\mathcal{A}_\Phi, \mathcal{B}_\Phi)$ of a fairness requirement Φ, such that ϕ holds for $\mathfrak{T}$ under the fairness specification ϕ if the liveness property $(\mathcal{I}, \mathcal{T}, \mathcal{F})$ holds under the fairness requirement Φ.*

5.5.3 Generic method for monodic temporal properties

In this section, we briefly discuss a method to verify (essentially) the monodic fragment [HWZ00] of temporal array properties by reducing the properties to fair termination. This method straightforwardly extends the classic construction of Büchi automata in automata-based LTL model checking [VW84; Wol00] to infinite structures. Similar reductions are employed in [AJN+12] for verifying LTL(MSO) properties of parameterised systems, and in [PHM+18] for verifying FO-LTL properties of first-order transition systems. The presentation here is essentially adopted from [PHM+18]. To simplify our exposition, we shall only consider the "globally" operator $\Box$ and refer the interested reader to [AJN+12] for treatments of other temporal operators.

Consider a temporal array property ϕ over variables V. Suppose that $\neg\phi$ is given in negation normal form, and that $\Box$ is the only temporal operator contained in $\neg\phi$. Let $sub(\neg\phi)$ denote the set of subformulae of $\neg\phi$. We define

$$V_{\neg\phi} := V \uplus \{G_{\Box\,\psi} \mid \Box\,\psi \in sub(\neg\phi)\}.$$

That is, $V_{\neg\phi}$ is obtained by introducing a fresh array variable $G_{\Box\psi}$ to V for each subformula $\Box\psi$ of $\neg\phi$. In a nutshell, to check that an array system satisfies ϕ, we take the product of the system and a "monitor" of $\neg\phi$ over the extended set of variables $V_{\neg\phi}$. The product system will be associated with a fairness condition such that the projection of the *fair* execution runs from $V_{\neg\phi}$ to V coincides with the set of runs violating ϕ in the original array system. This reduction is sound and complete, meaning that the original system satisfies ϕ if and only if the product system has no fair execution runs. When the input array system has no terminal states, checking that the system satisfies ϕ amounts to showing that the induced fair transition system always terminates.

To apply the aforementioned reduction in our regular framework, the formula ϕ needs to fulfil some technical requirements. Define a *core subformula* of a temporal formula as a subformula that has a temporal operator as its main connective. A temporal array formula is *monodic* [HWZ00; AJN+12] if every core subformula of it contains at most one free index variable. For example, $\exists i.\,\Box\,\forall j.\,a[i] > a[j]$ is monodic, but $\exists i.\,\forall j.\,\Box\,a[i] > a[j]$ is not. A formula is *index-bounded* if each quantified variable in the formula is explicitly bounded above by an integer or a system parameter that is also occurring in the same formula. For example, $\exists i.\,\Box\,\forall j.\,a[i] > a[j]$ is monodic but not index-bounded, while $\exists i.\,i \le |a| \wedge \Box\,\forall j.\,j \le |a| \Rightarrow a[i] > a[j]$ is both monodic and index-bounded. Note that we have not rigorous defined what "explicitly bounded above" means: for our purpose, it suffices to assume that a system parameter or integer serving as an upper bound can be determined from the formula for each quantified variable. If a formal treatment is desirable, one can adopt the formalism in [AJN+12; AHH16], wherein a semantical restriction equivalent to index-boundedness is made on the interpretation of index quantifiers, or adopt the formalism in [EN95; EN03], wherein a range expression is syntactically imposed for each quantified index variable in a well-formed formula.

The reduction method we shall be discussing only applies to index-bounded array systems and index-bounded monodic array properties. In the rest of this section, we shall write a formula ϕ as $\phi(i)$ to indicate that either ϕ does not have a free index

variable, or i is the unique free index variable in ϕ. Furthermore, we shall use FO[ϕ] to denote a representation of ϕ as an indexed formula over V_ϕ. This representation is defined inductively as follows, with $\circ \in \{\wedge, \vee\}$ and $\delta \in \{\exists, \forall\}$:

$$
\begin{aligned}
\mathrm{FO}[\phi] &:= \phi, \quad \phi \text{ contains no temporal operator} \\
\mathrm{FO}[\Box\, \phi(i)] &:= G_{\Box\,\phi}[i+1] = 1 \\
\mathrm{FO}[\neg\phi] &:= \neg\, \mathrm{FO}[\phi] \\
\mathrm{FO}[\psi_1 \circ \psi_2] &:= \mathrm{FO}[\psi_1] \circ \mathrm{FO}[\psi_2] \\
\mathrm{FO}[\delta i.\, \phi] &:= \delta i.\ \mathrm{FO}[\phi]
\end{aligned}
$$

Notice that FO[ϕ] and ϕ have the same set of free variables. Now, let $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ be a *non-blocking* array transition system. Define $\mathfrak{T}_{fair} := (V^\star_{\neg\phi}, \phi^\star_{init}, \phi^\star_{tran}, \phi_{fair})$, called a *fair transition system*, such that

$$
\begin{aligned}
V^\star_{\neg\phi} &:= V_{\neg\phi} \uplus \{p_i : \Box\, \psi(i) \in sub(\neg\phi)\} \\
\phi^\star_{init} &:= \phi_{init} \wedge \mathrm{FO}[\neg\phi] \wedge \textstyle\bigwedge_{\Box\,\psi(i) \in sub(\neg\phi)} |G_{\Box\,\psi}| = p_i + 1 \\
\phi^\star_{tran} &:= \phi_{tran} \wedge \textstyle\bigwedge_{\Box\,\psi(i) \in sub(\neg\phi)} \forall i.\, i \le p_i \Rightarrow (G_{\Box\,\psi}[i+1] = 1 \Leftrightarrow (\mathrm{FO}[\psi(i)] \wedge G'_{\Box\,\psi}[i+1] = 1)) \\
\phi_{fair} &:= \textstyle\bigwedge_{\Box\,\psi(i) \in sub(\neg\phi)} \forall i.\, \Box\Diamond\, (i \le p_i \Rightarrow G_{\Box\,\psi}[i] = 1 \vee \mathrm{FO}[\neg\psi(i)])
\end{aligned}
$$

At above, for each $\Box\, \psi(i) \in sub(\neg\phi)$, we have used p_i to denote the upper bound on i when i is the unique free index variable in $\Box\, \psi$. If $\Box\, \psi$ has no free index variable, we simply set $p_i := 0$. Intuitively, the fair transition system $\mathfrak{T}_{fair}$ uses an array variable $G_{\Box\,\phi}$ to annotate each state s of $\mathfrak{T}$ such that

- when $\Box\, \phi$ has a unique free index variable i, $G_{\Box\,\phi}[i+1] = 1$ if and only if all outgoing fair runs from state s satisfy the core subformula $\Box\, \phi(i)$;
- when $\Box\, \phi$ has no free index variable, we have $p_i = 0$, and $G_{\Box\,\phi}[1] = 1$ if and only if all outgoing fair runs from state s satisfy the core subformula $\Box\, \phi$;

The reason that this annotation only works for index-bounded monodic formulae is because our logical formalism only allows one-dimensional finite array variables.

Therefore, it is not possible to use an array variable to encode the truth values of a subformula when the subformula has more than one free index variable, or has a free index variable ranging over an infinite domain.

As mentioned earlier, the fair transition system $\mathfrak{T}_{fair}$ is constructed as the product of the original system $\mathfrak{T}$ and a monitor of $\neg\phi$ over the extended variable set $V_{\neg\phi}$. Given a subformula $\Box\,\psi$ of $\neg\phi$, the monitor updates the array $G_{\Box\,\psi}$ along a path of $\mathfrak{T}$ in accordance with whether or not $\Box\,\psi$ is satisfied by the current path. More concretely, consider a fair path π of $\mathfrak{T}_{fair}$, and suppose that $\Box\,\psi$ has a unique free index variable i. Given an integer $t \in \{0, \ldots, p_i\}$, if $\mathfrak{A}, \pi \models \Box\,\psi[t/i]$, then the monitor maintains the array value $G_{\Box\,\psi}[t+1]$ along the path π to make sure $\psi[t/i]$ is satisfied at every state of the path. If $\mathfrak{A}, \pi \not\models \Box\,\psi[t/i]$, then the monitor maintains the array value $G_{\Box\,\psi}[t+1]$ along the path π to make sure $\psi[t/i]$ is falsified at some state of the path. In the second case, the fairness condition ϕ_{fair} guarantees that the event of $\psi[t/i]$ being falsified will not be postponed forever.

Recall that a path of the transition system $\mathfrak{T}_{fair}$ consists of states over the extended variables $V^{\star}_{\neg\phi}$. Given an infinite fair path of $\mathfrak{T}_{fair}$, the projection of the path to the original variables V yields an infinite path of the original system $\mathfrak{T}$ that satisfies $\neg\phi$. Conversely, an infinite path satisfying $\neg\phi$ in the original system $\mathfrak{T}$ can be lifted to $V^{\star}_{\neg\phi}$ to obtain an infinite fair path of the transition system $\mathfrak{T}_{fair}$. As a consequence, one can verify the temporal array property ϕ against $\mathfrak{T}$ by checking the termination property for the fair transition system $\mathfrak{T}_{fair}$. We summarise this fact in the following result.

Proposition 5.5.4 ([AJN+12; PHM+18]). *Let $\mathfrak{T}$ be a non-blocking array system, and ϕ be an index-bounded monodic array property. Let $\mathfrak{T}_{fair}$ be the fair transition system defined as above from $\mathfrak{T}$ and ϕ. Then ϕ holds for $\mathfrak{T}$ if and only if $\mathfrak{T}_{fair}$ has no infinite fair execution run.*

When the fair transition system $\mathfrak{T}_{fair}$ is abstractable, the abstract termination analysis of $\mathfrak{T}_{fair}$ is amenable to regular model checking using the fairness encoding and the termination-to-safety reduction techniques we have described in Section 4.3. We summarise this section with the following theorem.

Theorem 5.5.5. *Let $\mathfrak{T}$ be a non-blocking index-bounded array system, and ϕ be an index-bounded monodic array property in negation normal form. If the induced fair transition system $\mathfrak{T}_{fair}$ is abstractable, then we can compute finite automata $\mathcal{I}$ and $\mathcal{T}$, as well as the regular encoding of a set of fairness requirements Φ, such that ϕ holds for $\mathfrak{T}$ if the termination property $(\mathcal{I}, \mathcal{T})$ holds under Φ.*

5.6 Case studies

We elaborate on three case studies to demonstrate the application of our regular framework. In these case studies, we shall use a simple guarded command modelling language (defined in Section A.3) to specify array systems. While being less expressive, this language is more compact and comprehensible than the full-fledged array logic for specification purpose. To make the presentation succinct, we shall often omit the range constraints for index variables and system parameters in a formula whenever these constraints are clear from the context. For example, we shall simply write formula $\forall i.\,(1 \leq i \wedge 1 \leq |a| \Rightarrow a[i] = 0)$ as $\forall i.\, a[i] = 0$.

5.6.1 Selection Sort

Consider the Selection Sort algorithm in Figure 5.3a. We specify the algorithm in our modelling language as follows, with the element theory instantiated to difference arithmetic:

DecHigh : $low \geq high, 1 < high \rhd high := high - 1, low := 1$

IncLow : $low < high, a[low] \leq a[high] \rhd low := low + 1$

Swap : $low < high, a[low] > a[high] \rhd a[high] := a[low], a[low] := a[high], low := low + 1$

The array transition system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ of Selection Sort is defined as

$$\begin{aligned} V &:= \{low, high, a, |a|\} \\ \phi_{init} &:= low = 1 \wedge high = |a| \\ \phi_{tran} &:= DecHigh \vee IncLow \vee Swap \end{aligned}$$

```
// indices of a start from 1 and end at a.length
let low = 1, high = a.length;
while (high > 1) {
   if (low == high)
      high = high - 1, low = 1;
   else if (a[low] > a[high])
      swap(a,low,high), low = low + 1;
   else
      low = low + 1;
}
```

(a) Pseudocode of Selection Sort, where `swap(a,i,j)` is a subroutine swapping the array values `a[i]` and `a[j]`.

Property	Specification	Explanation
P_1	$\Box\, \forall i. \forall j.\, (i < j \land high < j \Rightarrow a[i] \leq a[j])$	Sortedness
P_2	$(\forall i.\, a[i] = a_0[i]) \Rightarrow \Box\, (\forall i. \exists j.\, a[i] = a_0[j]) \land (\forall i. \exists j.\, a_0[i] = a[j])$	Permutation
P_3	$\Diamond\, high \leq 1$	Termination

(b) Correctness specification of Selection Sort, assuming that the input array a_0 has distinct array values.

Figure 5.3 Selection Sort: pseudocode and correctness specification.

where *DecHigh*, *IncLow*, and *Swap* are SIA formulae derived from their corresponding commands, see Example A.3.1.

We verify the usual loop invariants and termination property for Selection Sort; see Figure 5.3b. Property P_1 asserts that at any time point, the array segment above the pivot position *high* is sorted; property P_2 asserts that the algorithm produces a permutation of the input array a_0 given that the array values are distinct; property P_3 asserts that the algorithm eventually terminates. These three properties together establish the total correctness of the algorithm for arrays with distinct values.

Safety verification To verify the safety property P_1, we check that $\mathfrak{T} \not\models \neg P_1$. Note that

$$\begin{aligned} \neg P_1 &\equiv \Diamond\, \exists i.\, \exists j.\ i < j \wedge high < j \wedge a[i] > a[j] \\ &\equiv \exists p.\, \Diamond\, (\exists i.\ i < p \wedge high < p \wedge a[i] > a[p]), \end{aligned}$$

which is an abstractable temporal array property. We then construct a transition system $\mathfrak{T}_1 := (V^\star, \phi_{init}, \phi_{tran}, \phi_{bad})$ where

$$\begin{aligned} V^\star &:= V \uplus \{p\} \\ \phi_{bad} &:= \exists i.\, i < p \wedge high < p \wedge a[i] > a[p] \end{aligned}$$

As discussed in Section 5.5.1, we can reduce the task of verifying P_1 for $\mathfrak{T}$ to checking the safety of $\mathfrak{T}_1$. In our experiments, we use predicates $\mathbb{P} := \langle a[i] \le a[high],\, a[i] \le a[p] \rangle$ to construct an abstract system for $\mathfrak{T}_1$ and verify that it is safe.

To verify P_2, we define another system $\mathfrak{T}_2 := (V^\star, \phi^\star_{init}, \phi^\star_{tran}, \phi_{bad})$ based on

$$\begin{aligned} \neg P_2 &\equiv (\forall i.\, a[i] = a_0[i]) \wedge \Diamond\, ((\exists i. \forall j.\, a[j] \ne a_0[i]) \vee (\exists i. \forall j.\, a_0[j] \ne a[i])) \\ &\equiv (\forall i.\, a[i] = a_0[i]) \wedge \exists p.\, \Diamond\, ((\forall i.\, a[i] \ne a_0[p]) \vee (\forall i.\, a_0[i] \ne a[p])), \end{aligned}$$

which is an abstractable temporal array property. This time, we define $V^\star$ such that $V^\star_{arr} = V_{arr} \uplus \{a_0\}$ and $V^\star_{par} := V_{par} \uplus \{p, |a_0|\}$, and let

$$\begin{aligned} \phi^\star_{init} &:= \phi_{init} \wedge |a| = |a_0| \wedge (\forall i.\, a[i] = a_0[i]) \\ \phi^\star_{tran} &:= \phi_{tran} \wedge (\forall i.\, a'_0[i] = a_0[i]) \\ \phi_{bad} &:= (\forall i.\, a[i] \ne a_0[p]) \vee (\forall i.\, a_0[i] \ne a[p]) \end{aligned}$$

We then use predicates $\mathbb{P} := \langle a[i] = a_0[p],\, a_0[i] = a[p] \rangle$ to construct an abstract system for $\mathfrak{T}_2$ and verify that it is safe.

Liveness verification The property P_3 asserts that the sorting algorithm eventually terminates. To verify P_3 using our abstraction method, we first need to make sure that $\mathfrak{T}$ is non-blocking with respect to $\neg P_3$. As discussed in Section A.3, this amounts to proving the safety of the induced system $\mathfrak{T}^\dagger := (V, \phi_{init}, \phi^\star_{tran}, \phi_{bad})$, where

$$\begin{aligned}
\phi^\star_{tran} &:= \phi_{tran} \wedge high > 1 \\
\phi_{bad} &:= (high > low \vee high \leq 1) \wedge (high \leq low \vee a[low] > a[high]) \\
&\quad\ \wedge (high \leq low \vee a[low] \leq a[high]) \wedge high > 1
\end{aligned}$$

We can then use the predicate $\langle a[i] \leq a[high] \rangle$ to construct an abstract system for $\mathfrak{T}^\dagger$ and verify its safety, thereby establishing the required non-blocking condition.

For the verification of P_3 itself, we define $\mathfrak{T}_3 := (V, \phi_{init}, \phi_{tran}, \phi_{fin})$ with $\phi_{fin} := high \leq 1$ and check that every maximal execution run of $\mathfrak{T}_3$ eventually visits ϕ_{fin}. This can be proved using our abstraction method based on an empty set of predicates, which effectively produces an abstract system that only keeps track of the values of $|a|$, low, and $high$. Alternatively, one may add ϕ_{fin} to the specifications of $\mathfrak{T}_1$ or $\mathfrak{T}_2$ and check that P_3 holds for the modified systems. In fact, our tool can prove P_3 using the same predicates and abstract models we have computed for verifying the properties P_1 and P_2. This result reflects that our abstraction method is robust with respect to the choice of specifications and predicates.

5.6.2 Dijkstra's self-stabilising algorithm

Dijkstra's self-stabilising algorithm [Dij82] for mutual exclusion assumes a directed ring of N processes $p_1, \ldots, p_N$. Each process p_i holds a register x_i with values in $\{0, \ldots, K-1\}$. Each process can also read the value of the register at its predecessor in the ring. The concept of a *privileged* process is given as follows.

– Process p_1 is privileged if $x_1 = x_N$.

- For $i \in \{2, \ldots, N\}$, process p_i is privileged if $x_i \neq x_{i-1}$.

A privileged process will change the value of its register, causing the loss of its privilege:

- $x_1 := (x_1 + 1) \bmod K$ when $x_1 = x_N$.
- $x_i := x_{i-1}$ when $x_i \neq x_{i-1}$, where $i \in \{2, \ldots, N\}$.

Dijkstra's algorithm can be initialised with arbitrarily many privileged processes. When $K \geq N$, however, the system will converge to a stabilised state with exactly one privileged process, hence the term "self-stabilising". We specify Dijkstra's algorithm in the guarded command language where the element theory is instantiated to difference arithmetic.

$$\begin{aligned}
&\text{Update}_1 : pid = 1, x[pid] = x[\mathsf{n}], x[pid] < \mathsf{k} - 1 \rhd x[pid] := x[pid] + 1, pid := \mathsf{ndet}(1, \mathsf{n});\\
&\text{Update}_2 : pid = 1, x[pid] = x[\mathsf{n}], x[pid] = \mathsf{k} - 1 \rhd x[pid] := 0, pid := \mathsf{ndet}(1, \mathsf{n});\\
&\text{Update}_3 : pid > 1, x[pid] \neq x[pid - 1] \rhd x[pid] := x[pid - 1], pid := \mathsf{ndet}(1, \mathsf{n});
\end{aligned}$$

In the specification, n and k represent the parameters N and K, respectively, $x[i]$ represents the register x_i, and pid indicates which process is going to be selected by the scheduler. For simplicity, we have identified the symbol n with $|x|$ in the specification, and assumed that only privileged processes can be scheduled. Interestingly, Dijkstra's algorithm is non-blocking under this scheduling assumption, since it always maintains at least one privileged process in the system. To see this, notice that the SIA formula

$$\forall i. \, (i = 1 \Rightarrow x[i] \neq x[\mathsf{n}]) \wedge (i > 1 \wedge i \leq \mathsf{n} \Rightarrow x[i] = x[i - 1]),$$

meaning that the system has no privileged process, is unsatisfiable in the array logic T_A. This unsatisfiability can be proved using our abstraction method as demonstrated in Examples 5.4.1 and 5.4.2.

Now, let ψ_1, ψ_2, and ψ_3 be the three SIA formulae derived from the above specification (see Example A.3.2). The first-order array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ of Dijkstra's

algorithm is then specified as

$$\begin{aligned} V &:= \{pid, \mathsf{n}, \mathsf{k}, x\} \\ \phi_{init} &:= \mathsf{n} \le \mathsf{k} \wedge \wedge(\forall i.\, x[i] < \mathsf{k}) \\ \phi_{tran} &:= \psi_1 \vee \psi_2 \vee \psi_3 \end{aligned} \tag{5.15}$$

For convenience, let

$$\mathrm{Priv}(i) := (i = 1 \wedge x[i] = x[\mathsf{n}]) \vee (i \neq 1 \wedge x[i] \neq x[i-1]),$$

meaning that process i is privileged. The self-stabilising property of Dijkstra's algorithm then can be expressed as

$$P := \Diamond\Box\,(\exists i.\, \mathrm{Priv}(i) \wedge \forall i.\forall j.\, i \neq j \Rightarrow \neg\mathrm{Priv}(i) \vee \neg\mathrm{Priv}(j)),$$

which states that the system will eventually "stabilise", i.e., it will eventually contain exactly one privileged process and remain so forever. To prove the self-stabilising property P, we manually create subgoals $P_1, \ldots, P_4$ for the property as listed in the following table.

Property	Specification	Explanation
Q_1	$x[1] = x[\mathsf{n}]$	p_1 is privileged
Q_2	$\forall i.\, i \neq 1 \Rightarrow x[i] \neq x[1]$	$x_i \neq x_1$ for all $i \neq 1$
Q_3	$\forall i.\, i \neq 1 \Rightarrow x[i] = x[i-1]$	No p_i is privileged for $i \neq 1$
Q_4	$\exists i.\, \mathrm{Priv}(i) \wedge \forall i.\forall j.\, i \neq j \Rightarrow \neg\mathrm{Priv}(i) \vee \neg\mathrm{Priv}(j)$	Exactly one p_i is privileged
P_1	$\Box\Diamond\,(Q_1 \wedge pid = 1)$	Recurrence property
P_2	$\Box\Diamond\,(Q_1 \wedge pid = 1) \Rightarrow \Diamond\, Q_2$	Liveness under fairness
P_3	$Q_2 \Rightarrow \Diamond\,(Q_1 \wedge Q_3)$	Liveness property
P_4	$(Q_1 \wedge Q_3) \Rightarrow \Box\, Q_4$	Safety property
P	$\Diamond\Box\, Q_4$	Self-stabilising property

Intuitively, P_1 states that process p_1 is privileged and scheduled infinitely often; P_2 states that if process p_1 is privileged and scheduled infinitely often, then eventually

register x_1 differs from all the other registers; P_3 states that if x_1 differs from all the other registers at some point, then eventually p_1 is the only privileged process; P_4 states that if p_1 is the only privileged process, then the system has stabilised. To verify that Dijkstra's algorithm is self stabilising, it suffices to check the properties $P_1, \ldots, P_4$ separately, since the self-stabilising property P is subsumed by $P_1 \wedge P_2 \wedge P_3 \wedge P_4$.

Safety verification Let us start from the safety property P_4. Negating P_4 leads to

$$x[1] = x[\mathsf{n}] \wedge (\forall i.\, i \neq 1 \Rightarrow x[i] = x[i-1]) \wedge \Diamond\, (\forall i.\, \neg \mathrm{Priv}(i) \vee \exists i. \exists j.\, i \neq j \wedge \mathrm{Priv}(i) \wedge \mathrm{Priv}(j)).$$

Note that

- $\forall i.\, \neg \mathrm{Priv}(i)$ is T_A-equivalent to $x[1] \neq x[\mathsf{n}] \wedge (\forall i.\, i \neq 1 \Rightarrow x[i] = x[i-1])$
- $\exists i. \exists j.\, i \neq j \wedge \mathrm{Priv}(i) \wedge \mathrm{Priv}(j)$ is T_A-equivalent to $(x[1] = x[\mathsf{n}] \wedge \exists i.\, i > 1 \wedge x[i] \neq x[i-1]) \vee (\exists i. \exists j.\, i > 1 \wedge j > i \wedge x[i] \neq x[i-1] \wedge x[j] \neq x[j-1])$

Given the first-order specification $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ defined at (5.15), we construct a safe transition system $\mathfrak{T}_4 := (V, \phi^\star_{init}, \phi_{tran}, \phi_{bad})$ with

$$\begin{aligned}
\phi^\star_{init} &:= \phi_{init} \wedge x[1] = x[\mathsf{n}] \wedge (\forall i.\, i \neq 1 \Rightarrow x[i] = x[i-1]) \\
\phi_{bad} &:= (x[1] \neq x[\mathsf{n}] \wedge (\forall i.\, i \neq 1 \Rightarrow x[i] = x[i-1])) \\
&\quad\ \vee (x[1] = x[\mathsf{n}] \wedge \exists i.\, i > 1 \wedge x[i] \neq x[i-1]) \\
&\quad\ \vee (\exists i. \exists j.\, i > 1 \wedge j > i \wedge x[i] \neq x[i-1] \wedge x[j] \neq x[j-1])
\end{aligned}$$

Finally, we use the predicates $\mathbb{P} := \langle x[i] = x[i-1],\, x[i] = x[\mathsf{n}],\, x[i] = x[1] \rangle$ to compute an abstract system for $\mathfrak{T}_4$ and verify that it is safe, meaning that P_4 holds for $\mathfrak{T}$.

Liveness verification We proceed to prove the properties P_1 and P_3. To prove the recurrence property $P_1 := \Box\Diamond\, Q_1$, we check a stronger property that every maximal path of $\mathfrak{T}$ satisfies Q_1. If this property holds, then every infinite execution run of $\mathfrak{T}$ will visit Q_1 infinitely often, i.e., P_1 will hold. For this purpose, it suffices to specify $\mathfrak{T}_1 := (V, \phi^\star_{init}, \phi_{tran}, \phi_{fin})$ with $\phi^\star_{init} := \mathsf{true}$ and $\phi_{fin} := Q_1$. Similarly, to prove the

liveness property $P_3 := Q_2 \Rightarrow \Diamond (Q_1 \wedge Q_3)$, we specify $\mathfrak{T}_3 := (V, \phi^\star_{init}, \phi_{tran}, \phi_{fin})$ with $\phi^\star_{init} := \phi_{init} \wedge Q_2$ and $\phi_{fin} := Q_1 \wedge Q_3$. Both systems $\mathfrak{T}_1$ and $\mathfrak{T}_3$ are abstractable. We then use the set of indexed predicates $\mathbb{P} := \langle x[i] = x[i-1],\, x[i] = x[\mathsf{n}],\, x[i] = x[1] \rangle$ to construct abstract systems of $\mathfrak{T}_1$ and $\mathfrak{T}_3$ and check that the induced regular properties $(\mathscr{I}, \mathscr{T}, \mathscr{F})$ holds, thereby proving the properties P_1 and P_3 for $\mathfrak{T}$.

Property P_2, however, cannot be handled by our abstraction method given the current specification. To verify this property in the parameterised setting, we need to record the precise value of $x[1]$ in the abstract domain to avoid spurious counterexamples. Since the range of $x[1]$ is unbounded, tracking its value is impossible using a fixed number of predicates. (This is actually a common problem faced by the classic predicate abstraction when it is applied to check liveness, see e.g., Section 4.2 of [DCG+16] and the example therein.) To circumvent this issue, we augment the specification to represent $x[1]$ with an auxiliary index variable x_1. Technically, the augmentation is carried out by including a fresh index variable x_1 in V_{st} and making $x[1] = x_1$ an axiom of the specification. Since our regular encoding stores the concrete value of an index variable, binding the array value $x[1]$ with x_1 allows us to infer the precise value of $x[1]$ in the abstract analysis.

Furthermore, recall that the correctness of Dijkstra's algorithm depends on the premise that $\mathsf{k} \geq \mathsf{n}$. Since Dijkstra's algorithm maintains the invariant $\forall i.\, 0 \leq x[i] \wedge x[i] \leq \mathsf{k} - 1$, the premise implies, by the Pigeonhole Principle, that there exists an integer $z \in \{0, \ldots, \mathsf{k} - 1\}$ such that $x[i] \neq z$ for $i \in \{2, \ldots, \mathsf{n}\}$. This fact is crucial for verifying the property P_2. Unfortunately, the Pigeonhole Principle is not expressible by an abstractable formula. The existence of such an integer z hence cannot be inferred from the premise $\mathsf{k} \geq \mathsf{n}$ in our abstraction framework. To remedy, we shall directly specify the existence of z as a system parameter in the initial condition of our specification. More precisely, given the specification $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ of Dijkstra's algorithm, we

define a fair array system $\mathfrak{T}_2 := (V^\star, \phi^\star_{init}, \phi^\star_{tran}, \phi_{fin}, \phi_{fair})$ where

$$\begin{aligned}
V^\star &:= V \uplus \{z, x_1\} \\
\phi^\star_{init} &:= \phi_{init} \wedge x[1] = x_1 \wedge 0 \leq z \wedge z < \mathsf{k} \wedge (\forall i.\, i \neq 1 \Rightarrow x[i] \neq z) \\
\phi^\star_{tran} &:= \phi_{tran} \wedge x[1] = x_1 \wedge x'[1] = x'_1 \\
\phi_{fin} &:= \forall i.\, i \neq 1 \Rightarrow x[i] \neq x[1] \\
\phi_{fair} &:= \Box\Diamond\,(x[1] = x[\mathsf{n}] \wedge pid = 1)
\end{aligned}$$

Note that comparing an an array value with an index variable is generally not allowed by the syntax of the parametric array theory T_A. Nevertheless, using atoms like $x[1] = x_1$ in the specification will not pose a problem for our abstraction method, since we can instantiate both the index theory and the element theory of T_A to difference arithmetic.

Finally, we use predicates $\mathbb{P} := \langle x[i] = z,\ x[i] = x[1] \rangle$ to construct an abstract system for $\mathfrak{T}_2$ and check that every fair execution run of $\mathfrak{T}_2$ reaches ϕ_{fin}. This implies that $\Diamond\, Q_2$ holds for $\mathfrak{T}$ under the fairness condition $\Box\Diamond\,(Q_1 \wedge pid = 1)$, thereby proving the property P_2 for $\mathfrak{T}$.

5.6.3 Chang-Robert's algorithm

Chang-Robert's algorithm [CR79] is a ring-based leader election protocol. The algorithm assumes that each process has a unique ID, and that messages can be passed on the ring in the clockwise direction. The algorithm starts when an active process turns into an initiator and sends a message tagged with its ID to the next process. Let id_i denote the ID of process i. When process i receives a message tagged with id, it reacts in three cases:

- $id < id_i$: then process i purges the message.
- $id > id_i$: then process i forwards the message and becomes passive.
- $id = id_i$: then process i is elected as a leader.

The rationale behind Chang-Robert's algorithm is that only the message tagged with the highest ID will complete the round trip and make its sender the leader. The original algorithm requires that each process in the system has a message buffer at least as large as the the number of processes in the system. In this case study, we simplify the protocol by assuming that each process keeps at most one message — when a process receives more than one message, the message tagged with the highest ID will overtake the other messages.

Below we specify the transitions of Chang-Robert's algorithm in our modelling language, where the element theory is instantiated to difference arithmetic, and the symbols active and passive are aliases of 0 and 1, respectively.

$\text{Snd}_1 : pid < \mathsf{n}, id[pid] > msg[pid+1], status[pid] = \mathsf{active}$ /* send message */

$\quad \triangleright\ status[pid] := \mathsf{passive}, msg[pid+1] := id[pid], pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Snd}_2 : pid = \mathsf{n}, id[pid] > msg[1], status[pid] = \mathsf{active}$ /* send message */

$\quad \triangleright\ status[pid] := \mathsf{passive}, msg[1] := id[pid], pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Snd}_3 : pid < \mathsf{n}, id[pid] \leq msg[pid+1], status[pid] = \mathsf{active}$ /* drop message */

$\quad \triangleright\ status[pid] := \mathsf{passive}, pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Snd}_4 : pid = \mathsf{n}, id[pid] \leq msg[1], status[pid] = \mathsf{active}$ /* drop message */

$\quad \triangleright\ status[pid] := \mathsf{passive}, pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Fwd}_1 : pid < \mathsf{n}, msg[pid] \geq id[pid], msg[pid] \geq msg[pid+1]$ /* forward message */

$\quad \triangleright\ msg[pid] := 0, status[pid] := \mathsf{passive}, msg[pid+1] := msg[pid], pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Fwd}_2 : pid = \mathsf{n}, msg[pid] \geq id[pid], msg[pid] \geq msg[1]$ /* forward message */

$\quad \triangleright\ msg[pid] := 0, status[pid] := \mathsf{passive}, msg[1] := msg[pid], pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Fwd}_3 : pid < \mathsf{n}, msg[pid] \geq id[pid], msg[pid] < msg[pid+1]$ /* drop message */

$\quad \triangleright\ msg[pid] := 0, status[pid] := \mathsf{passive}, pid := \mathsf{ndet}(1, \mathsf{n});$

$\text{Fwd}_4 : pid = \mathsf{n}, msg[pid] \geq id[pid], msg[pid] < msg[1]$ /* drop message */

$\quad \triangleright\ msg[pid] := 0, status[pid] := \mathsf{passive}, pid := \mathsf{ndet}(1, \mathsf{n});$

The initial and the fairness conditions of the algorithm are given by

$$\phi_{init} := 2 \leq \mathsf{n} \wedge (\forall i.\, id[i] \neq 0 \wedge msg[i] = 0)$$
$$\phi_{fair} := \forall i.\, \Box\Diamond\, (i = pid)$$

where the index variable i ranges over $\{1, \ldots, \mathsf{n}\}$.

In the specification, we use $status[i]$ to denote the status of process i. The status of a process turns from active to passive when the process has either emitted or forwarded a message. A process can have an arbitrary status at the beginning of the protocol. We use $id[i]$ and $msg[i]$ to represent the ID and the message buffer, respectively, of process i. We stipulate that all process IDs are positive, and that $msg[i] = 0$ if and only if the message buffer of process i is empty. A process is called *enabled* if its status is active, or its message buffer is nonempty. We shall make a simplifying assumption that only an enabled process will be scheduled by the system.

We consider the following correctness specification:

Property	Specification
P_1	$\forall i.\, ((\exists j.\, id[i] < id[j]) \Rightarrow \Box\, (msg[i] \neq id[i]))$
P_2	$\forall i.\, ((\forall j.\, j \neq i \Rightarrow id[j] < id[i]) \wedge status[i] = \mathsf{active} \Rightarrow \Diamond\, (msg[i] = id[i]))$

Note that we have identified the event "process i receives a message tagged with its own ID" with "process i is elected as a leader." Hence, P_1 is a safety property stating that a process not holding the highest ID is never elected as a leader, while P_2 is a liveness property stating that a process holding the highest ID will eventually be elected as a leader after it has been initiated. The validity of specifying the correctness properties as such is based on the facts that process IDs are immutable, and that there is a unique process holding the highest ID. These facts may be verified separately or simply taken as assumptions.

Safety verification Let $\mathfrak{T} := (V, \phi_{init}, \phi_{tran}, \phi_{fair})$ be the first-order specification of Chang-Robert's algorithm. Proving the property P_1 for $\mathfrak{T}$ amounts to showing that $\mathfrak{T}$

does not satisfy

$$\neg P_1 \equiv \exists p. ((\exists q. id[p] < id[q]) \land \Diamond (msg[p] = id[p])).$$

To this aim, we construct a safety transition system $\mathfrak{T}_1 := (V^\star, \phi^\star_{init}, \phi_{tran}, \phi_{bad})$ based on the Skolemisation $id[p] < id[q] \land \Diamond (msg[p] = id[p])$ of $\neg P_1$, such that

$$\begin{aligned} V^\star &:= V \uplus \{p, q\} \\ \phi^\star_{init} &:= \phi_{init} \land id[p] < id[q] \\ \phi_{bad} &:= msg[p] = id[p] \end{aligned}$$

We thus reduce the verification problem to checking the safety of $\mathfrak{T}_1$. We use the predicates

$$\mathbb{P} := \langle id[i] < id[q], msg[i] = id[p], msg[i] < id[q] \rangle$$

to construct an abstract system for $\mathfrak{T}_1$ and verify that it is safe.

Optimisations. At the above, we have specified $id[p] < id[q]$ in the induced initial formula $\phi^\star_{init}$. To propagate this information to all reachable abstract states, we further include a predicate $id[i] < id[q]$ in $\mathbb{P}$. Note that this predicate $id[i] < id[q]$ is always evaluated to true at $i = p$ along an execution run, since p, q, and $id[\cdot]$ are all immutable in the specification. As an optimisation, we can specify $id[p] < id[q]$ as an axiom for $\mathfrak{T}_1$ to avoid introducing the predicate $id[i] < id[q]$. Technically, this can be achieved by annotating the system $\mathfrak{T}_1$ with the formula $id[p] < id[q]$, which leads to the system

$$(V^\star, \phi^\star_{init}, \phi_{tran} \land id[p] < id[q] \land id'[p] < id'[q], \phi_{bad} \land id[p] < id[q]).$$

We can then remove the predicate $id[i] < id[q]$ from $\mathbb{P}$ and perform the usual abstract safety analysis on the annotated system using a smaller predicate set $\langle msg[i] = id[p], msg[i] < id[q] \rangle$.

Liveness verification To verify the liveness property P_2, we first need to show that $\mathfrak{T}$ is non-blocking with respect to the negated property

$$\neg P_2 \equiv \exists p. (\forall i. i \neq p \Rightarrow id[i] < id[p]) \land status[p] = \mathsf{active} \land \Box (msg[p] \neq id[p]).$$

Under our scheduling assumption, a state is non-blocking if and only if the ring has at least one enabled process, i.e., a process i such that $msg[i] \neq 0$ or $status[i] = \mathsf{active}$. Therefore, to check the non-blocking condition, we define a transition system $\mathfrak{T}^\dagger := (V^\star, \phi^\star_{init}, \phi^\star_{tran}, \phi_{bad})$, where

$$
\begin{aligned}
V^\star &:= V \uplus \{p\} \\
\phi^\star_{init} &:= \phi_{init} \wedge (\forall i.\, i \neq p \Rightarrow id[i] < id[p]) \wedge status[p] = \mathsf{active} \\
\phi^\star_{tran} &:= \phi_{tran} \wedge msg[p] \neq id[p] \\
\phi_{bad} &:= (\forall i.\, msg[i] = 0 \wedge status[i] \neq \mathsf{active}) \wedge msg[p] \neq id[p]
\end{aligned}
$$

Intuitively, $\mathfrak{T}^\dagger$ is safe if and only if whenever the status of the process holding the highest ID is active, there will always be at least one enabled process in the system before that process is elected as a leader. Note that whether or not a leader is eventually elected is irrelevant to the non-blocking condition. To verify the safety of $\mathfrak{T}^\dagger$, we use the set of predicates

$$\mathbb{P} := \langle id[i] \neq 0,\ msg[i] \neq 0,\ status[i] = \mathsf{active},\ id[i] < id[p],\ msg[i] < id[p],\ msg[i] = id[p] \rangle$$

to construct a regular abstract system of $\mathfrak{T}^\dagger$ and verify that it is indeed safe. As discussed earlier, since $id[\cdot]$ and p are both immutable, we can reduce the size of the abstract system by making the constraints $\forall i.\, id[i] \neq 0$ and $\forall i.\, i \neq p \Rightarrow id[i] < id[p]$ axioms of $\mathfrak{T}^\dagger$, and then removing the predicates $id[i] \neq 0$ and $id[i] < id[p]$ from $\mathbb{P}$.

To verify the property P_2, define a fair transition system $\mathfrak{T}_2 := (V^\star, \phi^\star_{init}, \phi_{tran}, \phi_{fin}, \phi_{fair})$ where

$$
\begin{aligned}
V^\star &:= V \uplus \{p\} \\
\phi^\star_{init} &:= \phi_{init} \wedge (\forall i.\, i \neq p \Rightarrow id[i] < id[p]) \wedge status[p] = \mathsf{active} \\
\phi_{fin} &:= msg[p] = id[p]
\end{aligned}
$$

This time, we use the set of predicates

$$\mathbb{P} := \langle status[i] = \mathsf{active},\ id[i] < id[p],\ msg[i] < id[p],\ msg[i] = id[p] \rangle$$

The safety property								Learning		
Name	#L	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{bad}	T_{bad}	S_{inv}	T_{inv}	Time
Selection Sort P_1	2	28	39	337	597	22	38	110	220	2.4s
Selection Sort P_2	2	22	35	143	240	16	28	233	466	7.3s
Selection Sort NB	2	5	4	18	24	1	0	9	18	0.4s
Dijkstra P_4	2	15	18	55	133	112	165	35	70	0.7s
Chang-Robert P_1	2	18	21	181	379	10	20	34	68	3.3s
Chang-Robert NB	2	34	43	360	686	7	8	280	560	8.4s

Table 5.4 Results of applying our learning-based model checker (see Chapter 4) to the abstract safety analysis of the case studies. In this table, #L shows the size of the alphabet; S_x and T_x are the numbers of states and transitions of automaton/transducer x, respectively; **Learning** lists the results of our tool using Rivest and Schapire's version of L^* as the learning algorithm, and NB means non-blocking condition.

to construct an abstract system for $\mathfrak{T}_2$. Again, since $id[\cdot]$ and p are both immutable, we can make the constraint $\forall i.\, i \neq p \Rightarrow id[i] < id[p]$ in $\phi^{\star}_{init}$ an axiom to avoid the need of introducing the predicate $id[i] < id[p]$ to $\mathbb{P}$. Finally, we prove that the liveness property P_2 holds for $\mathfrak{T}$ by verifying that every fair execution run of $\mathfrak{T}_2$ eventually visits ϕ_{fin}.

5.7 Experiments and evaluation

We have implemented a prototype in Java to evaluate our approach over the case studies. We manually selected index predicates for each case study as described in the previous section. The verification of an array system consists of two steps: computing regular abstractions from the logical specification of the system, and performing abstract analysis on the obtained regular transition system. For the first step, recall that regular abstractions can be computed by relation-based and rule-based methods. While the former method is fully automatic, the computation runtime and the abstraction sizes were infeasible for most of the formulae we encountered in the case studies. Therefore, in our experiments, we computed regular abstractions using the rule-based methods with manually chosen inference rules. The resulting abstractions (as FO_{REG} formulae) were mechanically converted into finite automata and transducers to comprise abstract

The liveness property								The induced safety property							Learning			SLRP
Name	#L	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{fin}	T_{fin}	S_{init}	T_{init}	S_{tran}	T_{tran}	S_{bad}	T_{bad}	TimeR	S_{inv}	T_{inv}	Time	Time
Selection Sort P_3	2	7	8	25	34	5	10	9	8	71	102	4	4	0.0s	52	104	0.7s	1.5s
Dijkstra P_1	2	132	205	818	1391	12	22	91	117	1026	1679	4	4	1.0s	–	–	t.o.	3m00s
Dijkstra P_3	2	39	56	888	1637	4	7	41	47	1331	2215	4	4	1.5s	–	–	t.o.	3m48s
Dijkstra P_2	2	245	424	839	1461	18	35	185	236	4683	7515	18	23	4.1s	1041	2082	7m34s	n/a
Chang-Robert P_2	2	74	95	1393	2452	12	24	57	66	2333	3754	15	29	4.3s	770	1540	2m56s	n/a

Table 5.5 Results of applying our learning-based model checker and the SLRP tool (see Chapter 4) to the abstract liveness analysis of the case studies. In this table, #L stands for the size of the alphabet; S_x and T_x stand for the numbers of states and transitions of x, respectively; **Learning** lists the sizes of the invariants learnt by our tool using Rivest and Schapire's version of L^*; TimeR is the runtime of the liveness-to-safety reduction procedure.

regular transition systems. These abstract systems were then analysed by suitable regular model checkers. For safety properties, the abstract analysis was done by the learning-based safety verifier we have developed in Chapter 4. For liveness properties without fairness requirements, we applied two independent automated techniques for comparison: one was the liveness-to-safety reduction method described in Section 4.3, and the other was the SLRP tool developed by Lin and Rümmer [LR16]. For liveness properties under fairness requirements, we integrated regular encodings of these requirements in the reduction method as outlined in Section 4.3.2. We also tested T(O)RMC, ARMC, and the SAT-based verifier (see Section 4.6 for a description of these tools) over the problem instances obtained from the abstract safety properties, and from the abstract liveness properties after liveness-to-safety reductions.

We have run the experiments on a desktop computer with 3.6GHz Intel i7 processor, 16GB memory limit, and 10-minute timeout. Part of the results are listed in Table 5.4 and Table 5.5. The safety analysis of the case studies turned out to be relatively easy for our learning-based method: all of them could be completed in seconds by our prototype tool. As for the liveness analysis, SLRP proved all of the three liveness properties without fairness requirements, whereas our tool proved only one of them and timed out for the rest. On the other hand, our learning-based method successfully proved two liveness properties under fairness requirements, which were beyond the

capacities of SLRP. As for the other tools, T(O)RMC, ARMC, and the SAT-based verifier finished zero, one (Selection Sort NB in 0.1 second), and two (Selection Sort NB in 0.1 second, and Selection Sort P_3 in 10 seconds) examples, respectively, within the 10-minute timeout. We have omitted these results from the tables to simplify the presentation.

Concluding remarks

By combining predicate abstraction, decision procedures, and regular model checking, we present a framework to verify linear-time properties for index-bounded array systems. Given a temporal correctness specification, our framework provides a systematic method to compute regular over-approximations of the specification that are suitable for off-the-shelf model checkers to analyse. The experimental results show that this approach is able to verify non-trivial properties of array systems in several intriguing case studies.

In the future, we would like to harness the full power of our techniques by fully automating the computation of regular abstractions from a logical specification, as well as the conversion from regular abstractions to finite automata and transducers. While naive implementations are straightforward (for example, it is not difficult to use the Mona tool [KM01; KMS02] to compute a DFA from an $\mathrm{FO}_{\mathrm{REG}}$ formula), it takes careful optimisations and fine-tuning to make the computational costs feasible even for input instances of moderate size. Furthermore, we would like to study automatic generation and refinement for predicates in singly indexed array logic. The integration of various refinement techniques with our regular framework will be an important future work.

Chapter 6

Equivalence Verification of Regular Probabilistic Systems

Equivalence checking using bisimulation relations plays a fundamental role in formal verification. Bisimulation is the basis for substitutability of systems: if two systems are bisimilar, then their behaviours are the same and they satisfy the same formulae in expressive temporal logics. The notion of bisimulation is defined both for deterministic [Mil89] and for probabilistic transition systems [LS91]. In both contexts, checking bisimulation has many applications, such as proving correctness of anonymous communication protocols [CNP09], reasoning about knowledge [FHMV03], program optimization [KTL09], and optimizations for computational problems (e.g. language equivalence and minimization) of finite automata [BP13].

The problem of checking bisimulation between two given systems has been widely studied. It is decidable in polynomial-time for both probabilistic and non-probabilistic *finite-state* systems [Bai96; DHS03; VF10; CvBW12]. For infinite-state systems, such as parameterised versions of communication protocols (i.e. infinite families of finite-state systems with an arbitrary number n of processes), the problem is undecidable in general. Most research hitherto has focused on identifying and studying decidable subcases (e.g. strong bisimulations for pushdown systems in probabilistic and non-probabilistic

contexts [Srb04; Sén05; FJKW18]), rather than on providing tool support for practical problems.

In this chapter, we describe a first-order verification approach — inspired by software verification techniques — for reasoning about bisimulation over infinite-state probabilistic transition systems. In our approach, we provide first-order *proof rules* to determine if a given binary relation is a bisimulation. We aim to find an adequate symbolic representation of systems and relations, as well as a decidable first-order theory that can formalise the system, the property, and the proof rules. To this end, we propose to use the decidable first-order theory of the universal automatic structure as our logical framework to reasoning about bisimulations over infinite-state systems. We demonstrate the effectiveness of this approach by encoding and automatically verifying some challenging examples from the literature of parameterised systems in our framework: the parameterised versions of the dining cryptographers protocol [Cha88] and the grades protocol [KMO+12]. Prior to our work, anonymity of these protocols was only automatically verified for some fixed parameters using finite-state model checkers or equivalence checkers (cf. [HKNP06; KMO+12]). Just as invariant verification for software separates out the proof rules (verification conditions in a decidable logic) from the synthesis of invariants, we separate out proof rules for bisimulation from the synthesis of bisimulation relations. We demonstrate how recent development in generating and refining candidate proofs as automata (e.g. [NJ13; LR16; CHLR17]) can be used to automate the search of proofs, making our verification fully "push button."

We list our contributions in this chapter are as follows. First, we demonstrate how probabilistic infinite-state systems can be faithfully encoded in the first-order theory of the universal automatic structure. An analogous logic has been used to reason about qualitative liveness of weakly finite MDPs (e.g. see [LR16; LLMR17]). However, to the best of our knowledge, no encoding of probabilistic transition systems in the logic was available before our work.

Second, we demonstrate how to encode verification conditions of probabilistic bisimulation in the aforementioned first-order theory. The decidability of the theory

gives us an effective means of checking bisimulation over regular probabilistic transition systems. Since the theory can be syntactically reduced to WS1S, the encoding is amenable to highly optimised model checkers such as Mona [KM01; KMS02] and Gaston [FHJ+17].

Thus far, our framework requires a candidate proof to be supplied by the user. Our final contribution is to demonstrate how standard techniques from the synthesis literature can be used to fully automate the proof search. Using language inference, we successfully pinpointed regular proofs for the anonymity property of three examples: the dining cryptographers protocol and its generalised variant were verified in 6 and 28 seconds, respectively, and the grades protocol in 35 seconds.

6.1 Related work

The verification framework in this chapter can be construed as a probabilistic model checking framework using regular relations. Our framework exploits first-order logic as a specification language, making it convenient to express verification conditions (as is well-known from first-order theorem proving [BM98]). The use of the universal automatic structure allows us to express two different sorts (configurations and probability values) in one sort (i.e. strings).

Some automated techniques can prove the anonymity property of the dining cryptographers protocol and the grades protocol in the finite setting, e.g., the PRISM model checker [HKNP06; PRI], where anonymity is expressed in the PCTL logic, and the APEX tool [KMO+12; KMO+13], where anonymity can be formulated as a language equivalence problem. To the best of our knowledge, nevertheless, our method is the first automated technique proving the anonymity of these protocols in the parameterised setting.

Our work is in spirit of deductive software verification (e.g. [BM98; FLL+02; Lei10; ABB+16; PMP+16]), where one provides inductive invariants manually, and a tool automatically checks correctness of the candidate invariants. In theory, our result yields

a fully automatic procedure by enumerating all candidate proofs, and at the same time enumerating all candidate counterexamples (note that we avoid undecidability by restricting attention to proofs encodable as regular relations). In our implementation, we use recent advances in automata-learning based synthesis to explore the proof space effectively [LR16; CHLR17].

Our method exploits probabilistic bisimulation as a means of anonymity proof. The notion of bisimulation we consider is sometimes called strong bisimulation, and the behavioural equivalence we use to establish anonymity is called equivalence modulo strong bisimilarity [Par81; Mil90; SL95]. In the literature, other types of behavioural equivalences have also been studied for probabilistic systems, including weak bisimilarity [BH97; PLS00] and branching bisimilarity [AW06]. See [ZYS+18] for a survey of these variants.

Recently, it was shown that strong bisimilarity over probabilistic systems can be reduced to strong bisimilarity over non-probabilistic labelled transition systems [FJKW18]. This result indicates that specialised proof rules are not essential for verifying probabilistic bisimulation. However, the reduction presented in [FJKW18] does not generally preserve regularity of system encodings, and hence is not compatible with our regular model checking framework. While it might be possible to modify the reduction method to preserve regularity or to avoid explicit construction of the induced system, details of such modification and its effectiveness in automation would require further investigation.

6.2 Probabilistic bisimulation in regular relations

In this section, we show how to express a probabilistic system and the corresponding proof rules for probabilistic bisimulation in the framework of regular relations. We shall first discuss how to specify a PTS in the first-order theory FO_{REG}, and then show that it is possible to specify verification conditions for probabilistic bisimulation in the same theory, which allows us to express systems, properties, and proofs, all in a

uniform symbolic way.

6.2.1 The non-probabilistic case

As a warm-up, let us first consider bisimulation relations over non-probabilistic systems. A *labelled transition system (LTS)* is a structure $\mathfrak{S} := \langle S, \{\to_a\}_{a\in\mathsf{ACT}}\rangle$ such that $\to_a \subseteq S \times S$ for each $a \in \mathsf{ACT}$. A binary relation $R \subseteq S \times S$ is a bisimulation on the LTS $\mathfrak{S}$ if for all $a \in \mathsf{ACT}$ and $(s,t) \in R$, it holds that

- $s \to_a s'$ implies that there is some $t' \in S$ such that $t \to_a t'$ and $(s',t') \in R$;
- $t \to_a t'$ implies that there is some $s' \in S$ such that $s \to_a s'$ and $(s',t') \in R$.

This condition can be expressed in first-order logic. For each $a \in \mathsf{ACT}$, define

$$\begin{aligned}\phi_a(s,t) \quad := \quad & (\forall s' \in S.\,(s \to_a s' \Rightarrow \exists t' \in S.\,(t \to_a t' \wedge R(s',t')))) \\ & \wedge\, (\forall t' \in S.\,(t \to_a t' \Rightarrow \exists s' \in S.\,(s \to_a s' \wedge R(s',t')))).\end{aligned}$$

Then a binary relation $R \subseteq S \times S$ is a bisimulation for $\mathfrak{S}$ if and only if

$$\forall s \in S.\,\forall t \in S.\,R(s,t) \Rightarrow \left(\bigwedge\nolimits_{a\in\mathsf{ACT}} \phi_a(s,t)\right) \tag{6.1}$$

is true in the structure $\mathfrak{S}_R$ that extends $\mathfrak{S}$ with R. Note that this formula only asserts that R is a bisimulation but not necessarily the maximal one. Also note that (6.1) is expressible in $\mathrm{FO}_{\mathrm{REG}}$ when both R and $\mathfrak{S}$ are automata presentable. Since bisimulation relations are preserved under isomorphism, it is decidable to check whether or not a regular relation is a bisimulation for an automatic LTS.

6.2.2 Specifying a probabilistic transition system

Things become more complicated when we take transition probabilities into consideration. To analyse a PTS in the logic $\mathrm{FO}_{\mathrm{REG}}$, it would be convenient if the given PTS has a regular presentation. This motivates the following definition.

Definition 6.2.1. A relational automatic structure $\langle S, P, r_+, \{r_a\}_{a \in \mathsf{ACT}} \rangle$ is a *regular probabilistic transition system* (or a *regular PTS* for short) if it is a regular presentation of a PTS. #

A subclass of PTSs is of practical interest and worth mentioning. A PTS is said to satisfy the *minimal probability assumption* [LS91] if the transition probabilities of the PTS are multiples of some $\epsilon > 0$. This assumption is practically sensible since it is satisfied by most PTSs we encounter, e.g., finite PTSs, probabilistic pushdown automata [EE04], and most examples from probabilistic parameterised systems [LR16; LLMR17] including our case studies in Section 6.3. When a PTS satisfies the minimal probability assumption, we can convert its transition probabilities to natural numbers, called *weights*, by multiplying the probability values by $1/\epsilon$. This trick is known in the literature of probabilistic verification (see e.g. [AHM07]), and is sound for the purpose of model checking probabilistic bisimulations.

The definition of a regular PTS permits any legitimate encoding of the transition probabilities. However, when a regular PTS satisfies the minimal probability assumption, we can assume without loss of generality that the substructure $\langle P, r_+ \rangle$ of the regular PTS coincides with the regular presentation of Presburger arithmetic described in Example 2.2.3.

Example 6.2.1. We demonstrate a regular presentation of a toy PTS: a random walk on the natural numbers. At each position $x \in \mathbb{N}$, the system non-deterministically decides to loop or move. If the system decides to loop, then it stays at the current position with probability 1. If the system decides to move, then there are two cases: when $x \neq 0$, it moves to $x + 1$ with probability $3/4$ and moves to $x - 1$ with probability $1/4$; when $x = 0$, it moves to 1 with probability 1. This PTS satisfies the minimal probability assumption — multiplying the probabilities by 4, we obtain four weights 0, 1, 3, and 4 for the transitions.

We specify a regular presentation of the PTS as follows. The presentation has two actions: a for looping and b for moving. The set S of configurations are encoded in unary as words in 1^*. The set P of weights and the addition relation r_+ over weights

are encoded as in Example 2.2.3. Finally, the transition relations r_a and r_b are captured by the following formulae in $\mathrm{FO}_{\mathrm{REG}}(\{0,1\})$:

$$\begin{aligned} r_a(x,y,z) &:= (x = y \land z = 001) \lor (x \neq y \land z = 0); \\ r_b(x,y,z) &:= (x = \varepsilon \land y = 1 \land z = 001) \lor (x = \varepsilon \land y \neq 1 \land z = 0) \lor \\ &\quad (x \neq \varepsilon \land y = x1 \land z = 11) \lor (x \neq \varepsilon \land x = y1 \land z = 1) \lor \\ &\quad (x \neq \varepsilon \land y \neq x1 \land x \neq y1 \land z = 0). \end{aligned}$$

#

Example 6.2.2. As a second example, consider a regular PTS (from [FJKW18], Example 1) induced by a probabilistic pushdown automaton with states $Q = \{p,q,r\}$ and stack symbols $\Gamma = \{X, X', Y, Z\}$. The pushdown automaton has a single action a, and the transition rules δ_a are as follows:

$$\begin{array}{llll} pX \xrightarrow{0.5} qXX & pX \xrightarrow{0.5} p & qX \xrightarrow{1} pXX & rY \xrightarrow{1} rXX \\ rX \xrightarrow{0.3} rYX & rX \xrightarrow{0.2} rYX' & rX \xrightarrow{0.5} r & \\ rX' \xrightarrow{0.4} rYX & rX' \xrightarrow{0.1} rYX' & rX' \xrightarrow{0.5} r & \end{array}$$

A configuration of the PTS is a word in $Q\Gamma^*$, consisting of a state in Q and a word over the stack symbols. A transition can be applied if the prefix of the configuration matches the left hand side of the transition rules above. We encode the PTS as follows: the set of configurations is $Q\Gamma^*$, the weights are represented in binary as usual after normalization, and the transition relation $r_a(x,y,z)$ encodes the transition rules in disjunction. For example, the disjunct corresponding to the rule $pX \xrightarrow{0.5} qXX$ is $\exists u.\, x = pXu \land y = qXXu \land z = 101$. The branching of the PTS so obtained is bounded by 3. #

Two remarks are in order. First, bisimilarity over a regular PTS is undecidable even when the PTS is bounded branching. Indeed, the halting problem can be solved by checking bisimilarity for a bounded-branching regular PTS. To see this, notice that the configuration graph of a Turing machine is automatic and bounded branching (see Example 2.2.6). Therefore, given a Turing machine M, we can associate its transitions

with arbitrary nonzero transition probabilities and a single action a to obtain a bounded-branching regular PTS M'. This PTS M' is bisimilar to a PTS with a single configuration s, action a, and transition $\delta_a(s,s) = 1$ if and only if M does not halt.

Second, it is decidable to check whether or not a regular presentation of a bounded-branching PTS is well-defined, provided that the branching bound of the PTS is known. To see this, suppose we are given a set of $\mathrm{FO}_{\mathrm{REG}}$ formulae $\{\phi_a\}_{a\in\mathsf{ACT}}$ that are claimed to define the transitions of a regular PTS with branching bound n. To check that the PTS is well-defined, we need to verify the following three conditions for each ϕ_a:

- First, we need to check that the branching is bounded by n. That is, for all $x \in S$, there are at most n distinct y's in S such that $\phi_a(x,y,z) \wedge z \neq 0$ is satisfiable.
- Second, we need to check that ϕ_a encodes a function. This amounts to checking that $\forall x.\, \forall y.\, \exists! z.\, \phi_a(x,y,z)$ is true in $\mathrm{FO}_{\mathrm{REG}}$, where $\exists! z.\, \phi_a(x,y,z)$ is a shorthand for the assertion that given x and y, there is precisely one z satisfying $\phi_a(x,y,z)$.
- Third, we need to check that each ϕ_a encodes a mapping $S \to \mathscr{D}_S$.

The first two conditions are easily seen to be expressible in $\mathrm{FO}_{\mathrm{REG}}$ and hence are algorithmic. To verify the third condition, we check that there exists a number $p_a \in P$ satisfying the formula

$$\forall x.\, (\forall y.\, \forall z.\, \phi_a(x,y,z) \Leftrightarrow z = 0) \vee (\exists \overline{y}.\, \exists \overline{z}.\, \psi_a(x,\overline{y}) \wedge \textstyle\bigwedge_i \phi_a(x,y_i,z_i) \wedge \sum_i z_i = p_a),$$

where $\psi_a(x,\overline{y})$ asserts that the successors of x are among the configurations $\overline{y}$:

$$\psi_a(x,\overline{y}) := \forall y.\, \left(\phi_a(x,y,z) \wedge z \neq 0 \Rightarrow \textstyle\bigvee_i (y = y_i)\right).$$

Again, this satisfiability problem is decidable in $\mathrm{FO}_{\mathrm{REG}}$ and, when the formula is satisfiable, the exact value of p_a can be computed (e.g. by enumeration). This value, if exists, is unique for each $a \in \mathsf{ACT}$. Verifying the third condition then amounts to checking that $p_a = p_b$ for all $a, b \in \mathsf{ACT}$. We therefore conclude that it is decidable to check whether a regular presentation of a PTS is well-defined.

6.2.3 Proof rules for probabilistic bisimulation

Let $\sigma := \langle \{\delta_a\}_{a\in\mathsf{ACT}}, +, R\rangle$ be a two-sorted vocabulary with sort symbols $\mathbb{S}$ and $\mathbb{P}$, such that δ_a is a function of sort $\mathbb{S}\times\mathbb{S}\to\mathbb{P}$, $+$ is a function of sort $\mathbb{P}\times\mathbb{P}\to\mathbb{P}$, and R is a binary relation of sort $\mathbb{S}\times\mathbb{S}$. Given a structure $\mathfrak{S}$ and a relation R, let $\mathfrak{S}_R$ denote the structure extending $\mathfrak{S}$ with R. The main result we shall show next is summarised in the following theorem:

Theorem 6.2.1. *There is a fixed first-order sentence Φ over σ such that a given binary relation R is a probabilistic bisimulation for a bounded branching PTS $\mathfrak{S}$ if and only if $\mathfrak{S}_R \models \Phi$. Furthermore, checking $\mathfrak{S}_R \models \Phi$ is decidable when both R and $\mathfrak{S}$ are regular.*

We shall establish a proof of Theorem 6.2.1 in the rest of this section. Fix a PTS $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a\in\mathsf{ACT}}\rangle$ with a branching bound n. The fact that a binary relation R is an equivalence relation over S can be expressed as a first-order formula

$$\Psi := \forall s.\,\forall s'.\,\forall s''.\, R(s,s) \wedge (R(s,s') \Rightarrow R(s',s)) \wedge ((R(s,s') \wedge R(s',s'') \Rightarrow R(s,s'')).$$

The idea of the proof is to define formulae ψ_a and ϕ_a for each $a \in \mathsf{ACT}$, as well as a formula

$$\Phi := \Psi \wedge \forall s.\,\forall t.\, R(s,t) \Rightarrow \Big(\bigwedge\nolimits_{a\in\mathsf{ACT}} (\psi_a(s) \wedge \psi_a(t)) \vee \phi_a(s,t)\Big) \tag{6.2}$$

such that R is a probabilistic bisimulation for $\mathfrak{S}$ if and only if $\mathfrak{S}_R \models \Phi$. The formula $\psi_a(s)$ asserts that configuration s cannot move to any configuration through action a:

$$\psi_a(s) := \forall t.\ \delta_a(s,t) = 0. \tag{6.3}$$

Before we describe the definition of ϕ_a, we provide some intuition and auxiliary macros. Given configurations s and t, the formula $\phi_a(s,t)$ will first guess a set of n configurations $\overline{u}$ containing the successors of s through action a, and a set of n configurations $\overline{v}$ containing the successors of t through action a. The formula will also guess a labelling $\overline{\alpha}$ and $\overline{\beta}$ that corresponds to the partitioning of the configurations $\overline{u}$ and $\overline{v}$, respectively. The intuition here is that the labelling "names" the partitions:

$\alpha_i = \alpha_j$ (resp. $\beta_i = \beta_j$) means that u_i and u_j (resp. v_i and v_j) are guessed to be in the same partition. The formula then checks that the guessed partitioning is compatible with the equivalence relation R (i.e. u_i and u_j have the same label iff $R(u_i, u_j)$ holds), and that the probability masses of the partitions assigned by configurations s and t satisfy the constraint given in (2.3).

For the first part, we define a formula

$$\mathsf{succ}_a(w, \overline{u}) := \left(\bigwedge\nolimits_{i<j} u_i \neq u_j\right) \wedge \left(\forall v.\, \delta_a(w, v) \neq 0 \Rightarrow \bigvee\nolimits_i v = u_i\right), \qquad (6.4)$$

stating that the successors of configuration w on action a are among the n distinct configurations $\overline{u}$. Note that a configuration may have fewer than n successors. In this case, we can set the rest of the variables to arbitrary distinct configurations.

For the second part, we check that R is compatible with the guessed partitions, and that configurations s and t assign the same probability mass to the same partition. Let $\overline{k}$ be a labelling for configurations $\overline{s}$. To check that the partitioning induced by the labelling is compatible with R, we need to express the condition that $k_i = k_j$ if and only if $R(s_i, s_j)$ holds. This condition can be expressed as a formula

$$\mathsf{compat}(\overline{s}, \overline{k}) := \bigwedge\nolimits_{i<j} \left(R(s_i, s_j) \Leftrightarrow k_i = k_j\right).$$

Now, we are ready to define the formula ϕ_a.

$$\begin{aligned} \phi_a(s,t) \quad := \quad & \exists \overline{u}.\, \exists \overline{v}.\, \exists \overline{\alpha}.\, \exists \overline{\beta}.\, \mathsf{succ}_a(s, \overline{u}) \wedge \mathsf{succ}_a(t, \overline{v}) \wedge \mathsf{compat}(\overline{u}, \overline{v}, \overline{\alpha}, \overline{\beta}) \\ & \wedge \forall k.\, \left(\sum\nolimits_{i:\, \alpha_i = k} \delta_a(s, u_i) = \sum\nolimits_{i:\, \beta_i = k} \delta_a(t, v_i)\right). \end{aligned} \qquad (6.5)$$

With this definition, $\mathfrak{S}_R \models \phi_a(s,t)$ iff $\delta_a(s, E) = \delta_a(t, E)$ for any equivalence class $E \in S/R$.

Example 6.2.3. Consider the regular PTS from Example 6.2.2. The configurations pXZ and rX are bisimilar. This can be seen using a probabilistic bisimulation relation with equivalence classes $\{pX^kZ\} \cup \{rw : w \in \{X, X'\}^k\}$ for all $k \geq 0$ and $\{qX^{k+1}Z\} \cup \{rYw : w \in \{X, X'\}^k\}$ for all $k \geq 1$. The probabilistic bisimulation relation is definable as the

symmetric closure of a regular relation R, where $(w_1, w_2) \in R$ iff

$$\begin{aligned}
&(w_1 = w_2) \\
&\vee (w_1 \in PX^*Z \wedge w_2 \in r(X+X')^*\bot \wedge |w_1| = |w_2|) \\
&\vee (w_1 \in r(X+X')^* \wedge w_2 \in r(X+X')^* \wedge |w_1| = |w_2|) \\
&\vee (w_1 \in qX^*Z \wedge w_2 \in rY(X+X')^*\bot \wedge |w_1| = |w_2|) \\
&\vee (w_1 \in rY(X+X')^* \wedge w_2 \in rY(X+X')^* \wedge |w_1| = |w_2|).
\end{aligned}$$

For this example, the formula (6.2) simplifies to $\Psi \wedge \forall s, t.\, \phi_a(s,t)$ for the unique action a. This formula defines a condition that checks the bisimulation relation for all states symbolically. To see the formula in action, fix configurations pXZ and rX which are bisimilar. In the PTS, pXZ has two successors, $qXXZ$ and pZ, each with probability 0.5, and rX has three successors, rYX with probability 0.3, rYX' with probability 0.2, and r with probability 0.5. In the formula for ϕ_a, we can set the successors u_i of pXZ and the successors v_j of rX as above (the third "successor" u_3 is set to an arbitrary configuration not reachable from pXZ), and set $\alpha_1 = 1$, $\alpha_2 = 2$, $\beta_1 = \beta_2 = 1$, and $\beta_3 = 2$, corresponding to the equivalence classes of the bisimulation relation. One can check that the probability masses to these classes are the same.

We remark that the logic $\mathrm{FO}_{\mathrm{REG}}$ is sufficient to encode any probabilistic pushdown automaton, not just this example. The generic encoding will look like the one we have given for pushdown automata in Example 2.2.5, but with actions replaced by natural weights. #

We proceed to prove the second half of Theorem 6.2.1. Let $\mathfrak{S} := \langle S, P, r_+, \{r_a\}_{a \in \mathsf{ACT}} \rangle$ be a regular PTS. By Theorem 2.2.3, it suffices to show that the formula ψ_a given in (6.3) and the formula ϕ_a given in (6.5) are both expressible in $\mathrm{FO}(\mathfrak{S})$. The translation of ψ_a to $\mathrm{FO}(\mathfrak{S})$ is straightforward: just define

$$\psi_a(s) := \forall t.\, \exists z.\, r_a(s,t,z) \wedge r_+(z,z,z).$$

To translate the formula ϕ_a, the key point is to express the conditional summation of transition probabilities. For this purpose, we write a formula that performs iterated

additions. Formally, for each $a \in \mathsf{ACT}$ we define a formula χ_a such that

$$\chi_a(u, \overline{u}, \overline{\alpha}, k, z) := \exists \overline{z}.\ r_+(z_1, z_1, z_1) \wedge z_{n+1} = z \wedge \bigwedge_{1 \le i \le n} \chi'_a(u, u_i, \alpha_i, k, z_i, z_{i+1}),$$

where $u \in S$, $(\overline{u}) \in S^n$, $k \in P$, $(\overline{\alpha}) \in P^n$, $(\overline{z}) \in P^{n+1}$, and

$$\chi'_a(u, u', \kappa, k, x, y) := (\kappa = k \wedge \exists z.\ r_a(u, u', z) \wedge r_+(x, z, y)) \vee (\kappa \neq k \wedge x = y)$$

effectively computes summation over the transition probabilities $\{\delta_a(u, u_i) : \alpha_i = k\}$ and stores the result in y. Therefore, given $k \in P$, $(\overline{u}), (\overline{v}) \in S^n$ and $(\overline{\alpha}), (\overline{\beta}) \in P^n$, we have

$$\sum_{i:\, \alpha_i = k} \delta_a(s, u_i) = \sum_{i:\, \beta_i = k} \delta_a(t, v_i)$$

if and only if it holds that

$$\mathfrak{S} \models \exists z.\ \chi_a(s, \overline{u}, \overline{\alpha}, k, z) \wedge \chi_a(t, \overline{v}, \overline{\beta}, k, z).$$

Consequently, the definition of the formula ϕ_a in (6.5) can be equivalently written as

$$\begin{aligned} \phi_a(s, t) \quad := \quad & \exists \overline{u}.\ \exists \overline{v}.\ \exists \overline{\alpha}.\ \exists \overline{\beta}.\ \mathsf{succ}_a(s, \overline{u}) \wedge \mathsf{succ}_a(t, \overline{v}) \wedge \mathsf{compat}(\overline{u}, \overline{v}, \overline{\alpha}, \overline{\beta}) \\ & \wedge \forall k.\ \exists z.\ \chi_a(s, \overline{u}, \overline{\alpha}, k, z) \wedge \chi_a(t, \overline{v}, \overline{\beta}, k, z). \end{aligned}$$

It follows that the formulae ψ_a and ϕ_a, and therefore the sentence Φ given in (6.2), can be defined in $\mathrm{FO}(\mathfrak{S}_R)$. Consequently, by Theorem 2.2.3, checking $\mathfrak{S}_R \models \Phi$ is decidable when both $\mathfrak{S}$ and R are regular. This concludes our proof for Theorem 6.2.1.

Theorem 6.2.1 leads to the following decidability result.

Theorem 6.2.2. *Given a bounded branching regular PTS $\mathfrak{S}$ and a regular relation $E \subseteq S \times S$, there exists a procedure that finds either a non-bisimilar pair $(u, v) \in E$, or a regular probabilistic bisimulation relation R over $\mathfrak{S}$ such that $E \subseteq R$. Furthermore, the procedure terminates if and only if E contains a non-bisimilar pair, or E is a subset of a regular probabilistic bisimulation relation.*

Note that when verifying parameterised systems we are typically interested in checking bisimulation over *a set of pairs* (instead of just one pair) of configurations, and hence E in the above statement.

Proof of Theorem 6.2.2. To prove the theorem, it suffices to provide two semi-procedures, one for checking the existence of R and the other for showing that a pair $(v, w) \in E$ is a witness for non-bisimulation. By Theorem 6.2.1, we can enumerate all possible candidate regular relation R and effectively check that R is a probabilistic bisimulation over $\mathfrak{S}$. The condition that $E \subseteq R$ is a first-order property and hence can be checked effectively as well.

To see that non-bisimulation is recursively enumerable, let $\mathfrak{S}_v^d$ denote the tree-structured finite PTS induced by unfolding the PTS $\mathfrak{S}$ up to d steps starting from v. By Theorem 2.3.1, two configurations v, w of $\mathfrak{S}$ are not bisimilar if and only if there is some PML formula ϕ and distance d such that $\mathfrak{S}_v^d, v \models \phi$ and $\mathfrak{S}_w^d, w \not\models \phi$. Therefore, we can locate a non-bisimilar pair, if exists, by enumerating and checking all pairs $(v, w) \in E$, distances $d \geq 0$, and PML formulae ϕ over actions ACT. The checks are effective since model checking a PML formula against a finite PTS is decidable, see Section 2.3.4. #

6.3 Application to anonymity verification

In this section, we demonstrate the power of our regular framework by showing that it is expressive enough to capture and verify the anonymity property of interesting cryptographic protocols. We shall first formalise the connection between the concept of anonymity and probabilistic bisimulation. We then exploit this connection to verify the anonymity of two classic cryptographic protocols: the dining cryptographers protocol [Cha88] and the grades protocol [KMO+12].

6.3.1 Preliminaries

Fix a PTS $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a \in \mathsf{ACT}} \rangle$. Let $\pi(\mathfrak{S})$ denote the set of all finite paths of $\mathfrak{S}$. An *adversary* $f : \pi(\mathfrak{S}) \to \mathcal{D}_{\mathsf{ACT}}$ resolves the non-deterministic choices of $\mathfrak{S}$ and induces a

PTS $\mathfrak{S}_f := \langle S', \{\delta'_a\}_{a\in\mathsf{ACT}'}\rangle$ defined as follows: $S' := \pi(\mathfrak{S}) \uplus \{\bot\}$, $\mathsf{ACT}' := \mathsf{ACT} \uplus \{\#\}$, and $\delta'_a : S' \to \mathcal{D}_{\mathsf{ACT}'}$ is defined such that for any paths $\pi := s_0 \to_{a_1} \cdots \to_{a_n} s_n$ and $\pi' := \pi \to_a s_{n+1}$ in $\pi(\mathfrak{S})$, we have

$$\delta'_a(\pi, \pi') := f(\pi)(a) \cdot \delta_a(s_n, s_{n+1}).$$

Furthermore, it holds that $\delta'_\#(\bot, \bot) := 1$ and $\delta'_\#(\pi, \bot) := 1 - \sum_{a\in\mathsf{ACT}} \delta'_a(\pi, \pi(\mathfrak{S}))$ for all $\pi \in \pi(\mathfrak{S})$. Informally speaking, $\mathfrak{S}_f$ describes the behaviour of $\mathfrak{S}$ under the adversary f by successively extending the paths of $\mathfrak{S}$ according to the distributions selected by f. The special "sink state" $\bot$ indicates that f has stuck at some terminal configuration, from which the path can only be extended with the dummy action #. Observe that $\mathfrak{S}_f$ is a Markov chain: precisely one distribution is enabled at each configuration of $\mathfrak{S}_f$. The paths of $\mathfrak{S}_f$ therefore induce a probability measure that can be formalised using the standard cylinder construction, see e.g., [Var85; Kwi03]. More precisely, a finite path $\pi := \pi_0 \to_{a_1} \cdots \to_{a_n} \pi_n$ of $\mathfrak{S}_f$ defines a *basic cylinder* Run_π, which is the set of all finite/infinite paths with π as a prefix. We associate this cylinder with probability $\Pr^{\pi_0}(Run_\pi \mid \mathfrak{S}_f) := \prod_{i=1}^n \delta'_{a_i}(\pi_{i-1}, \pi_i)$. This definition gives rise to a unique probability measure for the σ-algebra over the set of paths from π_0. We define $\tau(\pi) := a_1 \cdots a_n$ to be the *trace* of π, and call a set $\mathcal{T} \subseteq \mathsf{ACT}^+$ a *trace event*. The probability of a trace event $\mathcal{T}$ with respect to a configuration $s \in S$ is given by

$$\Pr^s(\mathcal{T} \mid \mathfrak{S}_f) := \Pr^s(\bigcup \{Run_\pi : \tau(\pi) \in \mathcal{T}, \pi \text{ starts from } s\} \mid \mathfrak{S}_f).$$

We shall write $\Pr(t \mid \mathfrak{S}_f)$ instead of $\Pr(\{t\} \mid \mathfrak{S}_f)$ when there is no danger of confusion.

Now we are ready to define the concept of anonymity. A PTS $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a\in\mathsf{ACT}}\rangle$ is *anonymous to an adversary* f if for all initial configuration $s \in S$ and trace event $\mathcal{T}$, the value of $\Pr^s(\mathcal{T} \mid \mathfrak{S}_f)$ is solely determined by $\mathcal{T}$, and thus independent of s. Intuitively, this means that the adversary cannot obtain any information about a specific initial configuration by experimenting on the system and observing the traces.

We shall only consider external adversaries in our case study. An adversary $f : \pi(\mathfrak{S}) \to \mathcal{D}_{\mathsf{ACT}}$ is *external* if $f(s) = f(s')$ for all $s, s' \in S$, and $f(s_0 \to_{a_1} \cdots \to_{a_n} s_n) =$

$f(s'_0 \to_{a'_1} \cdots \to_{a'_n} s'_n)$ when $a_i = a'_i$ for $i \in \{1, \ldots, n\}$. That is, an external adversary resolves a non-deterministic choice solely based on the trace she has observed so far. We call a PTS *anonymous* if it is anonymous to any external adversary. The following result establishes a connection between the anonymity property and probabilistic bisimulations.

Theorem 6.3.1. *Let* $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a \in \mathsf{ACT}} \rangle$ *be a PTS and* f *be an external adversary for* $\mathfrak{S}$*. Then for all configurations* $u, v \in S$ *such that* $u \simeq v$*,* $\Pr^u(\mathcal{T} \mid \mathfrak{S}_f) = \Pr^v(\mathcal{T} \mid \mathfrak{S}_f)$ *holds for any trace event* $\mathcal{T}$ *in* $\mathfrak{S}_f$*. That is,* u *and* v *induce the same trace distribution in* $\mathfrak{S}_f$*.*

Proof. Assume without loss of generality that $\mathcal{T}$ is prefix-free, that is, any $t \in \mathcal{T}$ is not a prefix of another $t' \in \mathcal{T}$. Then $\Pr^s(\mathcal{T} \mid \mathfrak{S}_f) = \sum_{t \in \mathcal{T}} \Pr^s(t \mid \mathfrak{S}_f)$. It is hence sufficient to prove the following statement: given any external adversary f for $\mathfrak{S}$, $u, v \in S$, and $t \in \mathsf{ACT}^+$, if $u \simeq v$ then it holds that $\Pr^u(t \mid \mathfrak{S}_f) = \Pr^v(t \mid \mathfrak{S}_f)$. We prove this by induction on the length of t. For the base case, suppose that $t = a \in \mathsf{ACT}$. Since $u \simeq v$, we have $\delta_a(u, S) = \delta_a(v, S)$. It follows that $\Pr^u(t \mid \mathfrak{S}_f) = f(u)(a) \cdot \delta_a(u, S) = f(v)(a) \cdot \delta_a(v, S) = \Pr^v(t \mid \mathfrak{S}_f)$ by the definition of external adversary. Now suppose that the hypothesis holds for $|t| = n$. Consider $t := a \cdot t'$ with $a \in \mathsf{ACT}$ and $t' \in \mathsf{ACT}^n$. Then $u \simeq v$ implies that

$$
\begin{aligned}
\Pr^u(t \mid \mathfrak{S}_f) &= f(u)(a) \textstyle\sum_{s \in S} \delta_a(u, s) \cdot \Pr^s(t' \mid \mathfrak{S}_{f_{u,a}}) \\
&= f(u)(a) \textstyle\sum_{[s] \in S/\simeq} \delta_a(u, [s]) \cdot \Pr^s(t' \mid \mathfrak{S}_{f_{u,a}}) \text{ by induction hypothesis} \\
&= f(v)(a) \textstyle\sum_{[s] \in S/\simeq} \delta_a(v, [s]) \cdot \Pr^s(t' \mid \mathfrak{S}_{f_{v,a}}) \text{ by the assumption } u \simeq v \\
&= f(v)(a) \textstyle\sum_{s \in S} \delta_a(v, s) \cdot \Pr^s(t' \mid \mathfrak{S}_{f_{v,a}}) \text{ by induction hypothesis} \\
&= \Pr^v(t \mid \mathfrak{S}_f),
\end{aligned}
$$

where $[s] := \{s' \in S : s' \simeq s\}$, and $f_{s,a}$ is an external adversary defined by

$$f_{s,a}(s_0 \to_{a_1} \cdots \to_{a_n} s_n) := f(s \to_a s_0 \to_{a_1} \cdots \to_{a_n} s_n).$$

Therefore the hypothesis also holds for $|t| = n + 1$, which concludes our proof. #

Based on Proposition 6.3.1, we propose a framework to verify the anonymity property of a PTS $\mathfrak{S} := \langle S, P, +, \{\delta_a\}_{a \in \mathsf{ACT}} \rangle$ as follows. Given a set of initial configurations $I \subseteq S$, we specify a "reference system" $\mathfrak{S}' := \langle S, P, +, \{\delta'_a\}_{a \in \mathsf{ACT}} \rangle$ that has the same configurations and actions as those of $\mathfrak{S}$, except that the trace distribution of $\mathfrak{S}'_f$ is independent of specific initial configurations for any external adversary f. We then try to find a probabilistic bisimulation relation R between $\mathfrak{S}$ and $\mathfrak{S}'$ such that $R \cap (I \times I)$ coincides with the identity relation over I. When such a relation R is found, we can conclude that the trace distribution of $\mathfrak{S}_f$ is also independent of the initial configurations for any external adversary f, and hence prove the anonymity property of $\mathfrak{S}$.

6.3.2 Case studies

The Dining Cryptographers Protocol

Dining cryptographers protocol [Cha88] is a multi-party computation algorithm aiming to securely compute the XOR of the secret bits held by the participants. More precisely, consider a ring of $n \geq 3$ participants $p_0, \ldots, p_{n-1}$ such that each participant p_i holds a secret bit x_i. To compute $x_0 \oplus \cdots \oplus x_{n-1}$ without revealing information about the values of $x_0, \ldots, x_{n-1}$, the participants carry out a two-stage computation as follows: i) Each two adjacent participants p_i, p_{i+1} compute a random bit b_i that is accessible only to them; ii) Each participant p_i announces the value $a_i := x_i \oplus b_i \oplus b_{i-1}$[1] to the other participants. Hence, every participant p_i can observe the values of x_i, b_i, b_{i-1} and $a_0, \ldots, a_{n-1}$. It turns out that $a_0 \oplus \cdots \oplus a_{n-1} = x_0 \oplus \cdots \oplus x_{n-1}$, so all participants are able to compute the XOR of the secret bits after executing the protocol. Furthermore, the anonymity property of the protocol assures that any individual participant p_i cannot infer the values of the other secret bits from the information she has observed during the execution of the protocol.

We model the protocol as a regular PTS. The configurations of a ring of n participants are encoded as words of size n. The initial configurations are words $w \in \{\mathtt{0}, \mathtt{1}\}^*$ such

[1] All arithmetical operations on the subscripts are performed modulo n to take the ring structure into account.

that $w[i]$ represents x_i for $i \in \{0, \ldots, |w| - 1\}$. The transition relation consists of six transitions: observer non-deterministically tossing head (via action head), observer non-deterministically tossing tail (via action tail), non-observer tossing head with probability 0.5 (via action toss), non-observer tossing tail with probability 0.5 (via action toss), participant announcing zero (via action 0), and participant announcing one (via action 1). The outcomes of the tosses by the observer are visible (i.e. as actions head and tail), while the outcomes of the tosses by the other participants are hidden (i.e. as action toss). Each maximal trace from an initial configuration of size n consists of n successive tossing actions, followed by n successive announcing actions. Starting from an initial configuration w and for $i \in \{0, \ldots, n-1\}$, the i-th toss action updates the value of $w[j]$ to $w[j] \oplus b_i$ for $j \in \{i, i+1\}$, where $b_i = 1$ if a head is tossed and $b_i = 0$ otherwise. Any configuration v reached after n tosses would satisfy $v[i] = x_i \oplus b_i \oplus b_{i-1}$ for $i \in \{0, \ldots, n-1\}$. The PTS then "prints out" the configuration by going through n announcement transitions via actions $a_0, \ldots, a_{n-1}$, such that a_i is 1 if $v[i] = 1$ and a_i is 0 if $v[i] = 0$.

We consider the case where the first participant of the protocol is the observer. The maximal traces of the PTS in this case are in form of $t \cdot t'$, where $|t| = |t'|$, $t \in \{\mathsf{head}, \mathsf{tail}\}\,\mathsf{toss}^*\{\mathsf{head}, \mathsf{tail}\}$, and $t' \in \{0, 1\}^*$. For example, head toss tail 1 0 0 is a maximal trace starting from initial configuration 010. To prove anonymity, we define a reference system such that the initial configurations and the actions are the same as those of the original PTS, except that the announcements $a_0, \ldots, a_{n-1}$ encoded in the maximal traces from an initial configuration w are uniformly distributed over $\{(a_0, \ldots, a_{n-1}) : a_0 \oplus \cdots \oplus a_{n-1} = w[0] \oplus \cdots \oplus w[n-1],\ a_0 = w[0] \oplus b_0 \oplus b_{n-1}\}$.[2] In this way, the distribution of the announcements is independent of the initial configuration once the values of $x_0 \oplus \cdots \oplus x_{n-1}$, x_0, b_0, and b_{n-1} (i.e. the information revealed to the first participant) are fixed. We then compute a probabilistic bisimulation between the original system and the reference system, establishing the anonymity property

[2]Such a distribution can be obtained by i) choose $a_1, \ldots, a_{n-2} \in \{0, 1\}$ uniformly at random; ii) set $a_0 = w[0] \oplus b_0 \oplus b_{n-1}$; iii) set $a_{n-1} = a_0 \oplus \cdots \oplus a_{n-2} \oplus w[0] \oplus \cdots \oplus w[n-1]$.

that the first participant cannot infer the secret bits of the other participants from the information she observes.

A generalised dining cryptographers protocol We have also considered a slightly generalised version of the protocol where the secret messages $x_0, \ldots, x_{n-1}$ of the n participants are bit-vectors of the same size. Note that the set of the initial configurations is not regular when the size of the secret messages is parameterised. To construct a regular model, we allow a configuration to encode secret messages of different sizes, and devise the transition system such that an initial configuration w can finish the protocol (i.e. can have a trace containing all of the announcements $a_0, \ldots, a_{n-1}$) if and only if the messages encoded in w have same size. The resulting PTS is a regular system; it over-approximates the PTS of the generalised dining cryptographers protocol in the sense that the anonymity property of the former implies that of the latter. More details of the model can be found in Appendix A.4.

The Grades Protocol

The grades protocol [KMO+12] is a multi-party computation algorithm aiming to securely compute the sum of the secrets held by the participants. The setting of the protocol is pretty similar to that of the dining cryptographers: given $n \geq 3$ and $g \geq 2$, we have a ring of n participants $p_0, \ldots, p_{n-1}$ where each participant p_i holds a secret $x_i \in \{0, \ldots, g-1\}$. Note that both g and n are parameterised in this protocol. The goal of the participants is to compute the sum $x_0 + \cdots + x_{n-1}$ without revealing information about the individual secrets. Define $M := (g-1) \cdot n + 1$. The protocol consists of two steps: i) Each two adjacent participants p_i, p_{i+1} compute a random number $y_i \in \{0, \ldots, M-1\}$; ii) Each participant p_i announces $a_i := (x_i + y_i - y_{i-1}) \bmod M$ to the other participants. After executing the protocol, the participants compute $a := a_0 + \cdots + a_{n-1} \bmod M$. Because of the ring structure, the y_i's will be cancelled out in the sum. Thus the value of a will equal to the sum of all secrets. The anonymity property of the protocol asserts that no participant can infer the secrets held by the

other participants from the information she has observed.

We consider a variant of the grades protocol where M can be any power of two greater than $(g-1)\cdot n$. Observe that the same anonymity and correctness property of the original protocol also holds for this variant. To verify the anonymity property, we model an over-approximation for the protocol where the secrets are allowed to range over $\{0,\ldots,M-1\}$. This model is similar to the one we have constructed for the generalised dining cryptographers protocol except that, e.g., the XOR operations are now replaced with bitwise additions and negations. A reference system is specified such that the announcements $a_1,\ldots,a_{n-1}$ observed by the first participant p_0 are uniformly distributed over the values satisfying $a_0+\cdots+a_{n-1} \bmod M = x_0+\cdots+x_{n-1} \bmod M$. By computing a probabilistic bisimulation between the original system and the reference system, we establish the anonymity property that the grades protocol is anonymous whenever M is chosen as a power of two with $M \geq (g-1)\cdot n+1$. See Appendix A.5 for model details of the model.

6.4 Learning probabilistic bisimulation

We propose a synthesis method to automatically compute regular probabilistic bisimulations for length-preserving regular PTSs, which cover all examples given in the previous section. A *length-preserving* PTS regular is simply a PTS where the system can transit from one configuration to another only if the two configurations have the same length. Our synthesis method uses active automata learning, for instance Angluin's L^* algorithm [Ang87] or refinements of it, to compute regular probabilistic bisimulations. This approach is inspired by the success of applying language inference to inductive invariant generation, see Section 4.5. Our procedure assumes (i) as input a length-preserving regular PTS $\mathfrak{S}$, as well as a length-preserving regular relation $E \subseteq \Sigma^* \times \Sigma^*$ expected to consist of bisimilar pairs of words. (ii) an effective way to check whether a given regular relation R is a probabilistic bisimulation for $\mathfrak{S}$, i.e., a decision procedure in the sense of Theorem 6.2.1; (iii) a procedure to compute the greatest probabilistic

bisimulation $\tilde{R}_n \subseteq \Sigma^n \times \Sigma^n$ for $\mathfrak{S}$ restricted to configurations of length $n \in \mathbb{N}$. The last assumption can easily be satisfied by length-preserving PTSs. Indeed, such systems, restricted to configurations of length n, are finite-state, and therefore amenable to efficient existing methods [Bai96; DHS03; VF10; CvBW12]. A solution R is represented as a deterministic letter-to-letter transducer, i.e., as a deterministic finite-state automaton over the alphabet $\Sigma \times \Sigma$,

Since L^*-style learning requires the taught language to be uniquely defined, our approach attempts to learn a representation of the greatest length-preserving probabilistic bisimulation relation $\tilde{R} \subseteq \Sigma^* \times \Sigma^*$, which is the unique bisimulation relation formed by the union of all length-preserving probabilistic bisimulations of $\mathfrak{S}$, i.e., $\tilde{R} := \bigcup_{n \geq 1} \tilde{R}_n$. Because $\tilde{R}$ is not in general computable, the learning process might diverge and fail to produce any probabilistic bisimulation. It can also happen that learning terminates, but yields a probabilistic bisimulation relation strictly smaller than $\tilde{R}$.

The L^* method requires a teacher that is able to answer two kinds of queries:

- **membership queries**: given a pair of words $v, w \in \Sigma^*$, whether or not (v, w) is contained in the the greatest bisimulation $\tilde{R}$. This amounts to checking whether v and w are bisimilar. Since $\tilde{R}$ is length-preserving, the teacher can answer this query by computing the probabilistic bisimulation $\tilde{R}_{|v|}$ and checking $(v, w) \in \tilde{R}_{|v|}$.
- **equivalence queries**: whether or not a given finite automaton $\mathcal{A}_h$ represents a probabilistic bisimulation containing E. Let R_h denote the regular relation represented by $\mathcal{A}_h$. To answer this query, the teacher essentially checks that R_h satisfies the proof rule Φ at (6.2). The complete algorithm is given in Algorithm 2. The algorithm first attempts to find a shortest counterexample to the proof rule. If a counterexample of length n is found, it will yield a pair of words in the difference set of R_h and $\tilde{R}_n$, which is a valid counterexample for automata learning since the learner attempts to learn the greatest bisimulation. The teacher thus reports this pair to be a positive or negative counterexample according to its membership in $\tilde{R}_n$.

Algorithm 2: Equivalence check for L^*

Input: A regular PTS $\mathfrak{S}$ and two length-preserving regular relations $R, E \subseteq \Sigma^* \times \Sigma^*$

Result:

NoSolution(v, w)	$(v, w) \in E$ is a non-bisimilar pair;
PositiveCEX(v, w)	R should contain (v, w), but it does not;
NegativeCEX(v, w)	R contains (v, w), but it should not;
Correct	R is a bisimulation on $\mathfrak{S}$ and $E \subseteq R$;

```
1  Check whether E ⊆ R, and whether S_R ⊨ Φ holds in the sense of
   Theorem 6.2.1;
2  if there is a counterexample of minimal length n then
3      Compute R̃_n, the greatest bisimulation restricted to configurations of length
       n;
4      if there is (v, w) ∈ E \ R̃_n with |v| = |w| = n then
5          Output NoSolution(v, w) and abort;
6      else if there is (v, w) ∈ R \ R̃_n with |v| = |w| = n then
7          return NegativeCEX(v, w);
8      else if there is (v, w) ∈ R̃_n \ R then
9          return PositiveCEX(v, w);
10 else
11     return Correct;
```

The learning procedure terminates when the teacher outputs *NoSolution* or returns *Correct* for an equivalence query. In the former case, the teacher explicitly provides a pair of non-bisimilar configurations in E. In the latter case, the procedure computes an automaton $\mathscr{A}_h$ such that $\langle E \rangle \subseteq A_h$ and A_h encodes a probabilistic bisimulation (as it satisfies the proof rule based on Theorem 6.2.1), though not necessarily the greatest one. Since all counterexamples reported by the teacher are contained in the difference set of R_h and $\tilde{R}$, the learning procedure is guaranteed to terminate for a PTS $\mathfrak{S}$ whenever the greatest probabilistic bisimulation $\tilde{R}$ over $\mathfrak{S}$ is regular.

Optimisation with inductive invariants There is a natural way to optimise the learning procedure by considering a *regular* inductive invariant *Inv* such that *Inv* contains the set of reachable configurations and $E \subseteq Inv \times Inv$. The optimization is done by simply replacing the greatest finite-length bisimulations $\tilde{R}_n$ in Algorithm 2, and when answering membership queries, with the greatest bisimulation $\tilde{R}_n^I := \tilde{R}_n \cap Inv$ on the inductive invariant. Since $\tilde{R}_n^I$ can be a lot smaller than $\tilde{R}_n$, this can lead to significant speed-ups. Note that a bisimulation R' on *Inv* can be extended to a bisimulation R on all configurations by setting $R := R' \cup \{(v, v) : v \notin Inv\}$. The inductive invariant *Inv* may be manually specified, or automatically generated using techniques like [CHLR17].

6.5 Experiments and evaluation

We have implemented a prototype in Scala to test our learning method. Given a PTS specified over $\mathfrak{U}$, our tool first translates it to WS1S formulae and obtains finite automata for these formulae using the Mona tool [KM01]. Our prototype then applies the L^* learning procedure as described in this section, including the optimization to consider only the configurations of valid format. When answering an equivalence query, our tool invokes Mona to verify candidate automata and obtain counterexamples (line 1-2 of Algorithm 2). We used the prototype tool to prove anonymity for the three protocols described in Section 6.3. The proofs produced by the tool were probabilistic

bisimulation relations encoded in finite-state automata. The experimental results are summarised in the following table.

Case study	#states	#trans	mona	bisim	total
Dining Cryptographers, single-bit	13	832	2s	2s	6s
Dining Cryptographers, multi-bit	16	1024	3s	24s	28s
The Grades Protocol	25	1600	5s	28s	35s

For each case study, we list the size of the final proof produced by our tool, the time taken by Mona to verify the candidate automata, the time taken by our tool to compute the fixed-length bisimulations, and the total computation time of the learning procedure. Experiments were conducted on a Windows laptop with 2.4GHz Intel i5 processor and 2GB memory limit. The experimental results indicate that all of the three case studies have regular proofs for their anonymity property; furthermore, these proofs are of moderate sizes and can be pinpointed by our tool in reasonable time.

Chapter 7

Conclusion and Future Work

The main theme of the development in this thesis is to exploit decidable first-order theories and regular model checking for parameterised verification. By using appropriate encodings and abstractions, we have shown that the first-order theories over automatic structures can be made surprisingly powerful to reason about various classes of parameterised systems represented by regular transition systems, array transition systems, and probabilistic transition systems. For these systems, temporal properties like safety and liveness, as well as quantitative properties like probabilistic bisimulation, can be uniformly formulated in our first-order verification scheme of regular model checking. Thanks to the connection between logic and automata theory, our approach allows to combine constraint solvers and language inference algorithms to automate the exploration of regular correctness proofs for logical specifications. We have implemented and evaluated our approaches against a nontrivial collection of examples including concurrent systems, distributed algorithms, and cryptographic protocols. The promising experimental results justify our use of regular language inference as a proof generation method, and confirm our hypothesis that practical systems have simple proofs (here "simple" stands for "regular") and, consequently, that regular languages and relations provide effective heuristics for verifying parameterised systems.

Future work

An immediate future research direction of this thesis is to extend the proposed regular framework to handle more complex infinite structures with different sources of infinity. For example, rather than concurrent process systems, we may consider regular frameworks for sequential programs with unbounded recursion, or timed systems with real-valued clocks. In fact, the liveness-to-safety reduction we have employed in Chapter 4 for regular systems also applies to recursive programs and timed systems [SB06]. To reason about such systems in a unified regular framework, we will need to design new symbolic representations for their state space, as well as efficient algorithms to manipulate these representations. For example, to capture real numbers in the symbolic representation, we might need to migrate the current word-regular framework to the ω-regular setting, and use a first-order theory over ω-automatic structures as logical formulation. Another possible research direction is to combine the existing techniques for new application areas. For example, the regular abstraction framework in Chapter 5 may be integrated with the regular encoding of probabilistic transition systems in Chapter 6 to reason about quantitative properties of probabilistic array systems.

We believe that the regular verification framework presented in this thesis can help designing automatic or semi-automatic heuristics in deductive reasoning about software programs. To apply our approaches to program analysis, however, it would be important to further explore the modelling power of first-order theories over automatic structures. For software programs, correctness reasoning often requires logical formalisms supporting arithmetic, set cardinalities, graph connectivity, inductive data structures, etc. Is it possible to capture these notions in regular model checking? A hint to this question might have been provided by the Ivy theorem prover [PMP+16; MP18; MP20]. Ivy advocates the use of EPR, which is a decidable fragment of pure and uninterpreted first-order logic, for deductive reasoning. This is unusual, since pure first-order logic is commonly considered too weak for program verification, e.g., it cannot express notions as essential as linear ordering. Surprisingly, the developers

of Ivy showed that, through engineering efforts and modelling tricks such as partial axiomatisation, module isolation, and theory hiding, EPR is sufficiently powerful to reason about complex protocols that involve ordering, arithmetic, counting, and set cardinalities [PLSS17; TLM+18; MP20]. These results are quite inspiring, and we believe that similar techniques can be adopted in our logical formulation of regular model checking to enhance its applicability.

Another challenge to harnessing the power of regular model checking is how to automatically extract regular models suitable for analysis from a high-level system description such as program source code. Chapter 5 provides an initial exploration in this direction by generalising predicate abstraction to produce regular systems rather than finite-state models. Part of the difficulties in automated regular model extraction arise from the variety of regular presentations one can use to encode an automatic system. Ideally, users of a model checker should only need to work on the descriptions of the specification to be checked without dealing with the underlying symbolic representation. Unfortunately, existing semi-algorithms for regular model checking tend to be sensitive to encodings in respect of performance and convergence, rendering manual adjustments of representations essential in practice. It therefore remains for future research to study and strengthen the robustness of the existing techniques, our learning-based approaches included, with respect to semantically equivalent (e.g. isomorphic) regular presentations.

Bibliography

[Abd12] P. A. Abdulla, "Regular model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 14, no. 2, pp. 109–118, 2012.

[AAC+14] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "String constraints for verification," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2014, pp. 150–166.

[ABB01] P. A. Abdulla, L. Boasson, and A. Bouajjani, "Effective lossy queue languages," in *International Colloquium on Automata, Languages, and Programming (ICALP)*, Springer, 2001, pp. 639–651.

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 1996, pp. 313–321.

[ACD+10] P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine, "Constrained monotonic abstraction: A CEGAR for parameterized verification," in *International Conference on Concurrency Theory (CONCUR)*, Springer, 2010, pp. 86–101.

[ACH+10] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar, "When simulation meets antichains," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2010, pp. 158–174.

[ADHR07] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine, "Regular model checking without transducers (on efficient verification of parameterized systems)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2007, pp. 721–736.

[ADHR09] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine, "Monotonic abstraction: On efficient verification of parameterized systems," *International Journal of Foundations of Computer Science*, vol. 20, no. 05, pp. 779–801, 2009.

[ADR08] P. A. Abdulla, G. Delzanno, and A. Rezine, "Monotonic abstraction in parameterized verification," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 223, pp. 3–14, 2008.

[ADR09] P. A. Abdulla, G. Delzanno, and A. Rezine, "Approximated parameterized verification of infinite-state processes with global conditions," *Formal Methods in System Design (FMSD)*, vol. 34, no. 2, pp. 126–156, 2009.

[AHH16] P. A. Abdulla, F. Haziza, and L. Holík, "Parameterized verification through view abstraction," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 18, no. 5, pp. 495–516, 2016.

[AHH13] P. A. Abdulla, F. Haziza, and L. Holík, "All for the price of few," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2013, pp. 476–495.

[AHD+08] P. A. Abdulla, N. B. Henda, G. Delzanno, F. Haziza, and A. Rezine, "Parameterized tree systems," in *International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, Springer, 2008, pp. 69–83.

[AHM07] P. A. Abdulla, N. B. Henda, and R. Mayr, "Decisive markov chains," *Logical Methods in Computer Science (LMCS)*, vol. 3, no. 4, 2007.

[AHJ+16] P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar, "Verification of heap manipulating programs with ordered data by extended forest automata," *Acta Informatica*, vol. 53, no. 4, pp. 357–385, 2016.

[AJ96] P. A. Abdulla and B. Jonsson, “Verifying programs with unreliable channels,” *Information and Computation*, vol. 127, no. 2, pp. 91–101, 1996.

[AJNd02] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d’Orso, “Regular model checking made simple and efficient,” in *International Conference on Concurrency Theory (CONCUR)*, Springer, 2002, pp. 116–131.

[AJN+12] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d’Orso, and M. Saksena, “Regular model checking for LTL(MSO),” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 14, no. 2, pp. 223–241, 2012.

[AJRS06] P. A. Abdulla, B. Jonsson, A. Rezine, and M. Saksena, “Proving liveness by backwards reachability,” in *International Conference on Concurrency Theory (CONCUR)*, Springer, 2006, pp. 95–109.

[ALdR06] P. A. Abdulla, A. Legay, J. d’Orso, and A. Rezine, “Tree regular model checking: A simulation-based approach,” *The Journal of Logic and Algebraic Programming*, vol. 69, no. 1-2, pp. 93–121, 2006.

[ABB+16] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, *Deductive Software Verification - The KeY Book - From Theory to Practice*. Springer, 2016, vol. 10001.

[ABG+12] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, “Safari: SMT-based abstraction for arrays with interpolants,” in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2012, pp. 679–685.

[AČ10] R. Alur and P. Černỳ, “Expressiveness of streaming string transducers,” in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[AČ11] R. Alur and P. Černỳ, “Streaming transducers for algorithmic verification of single-pass list-processing programs,” *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 599–610, 2011.

[AD17] R. Alur and L. D'antoni, "Streaming tree transducers," *Journal of the ACM (JACM)*, vol. 64, no. 5, pp. 1–55, 2017.

[ADT13] R. Alur, A. Durand-Gasselin, and A. Trivedi, "From monadic second-order definable string transformations to transducers," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2013, pp. 458–467.

[AFT12] R. Alur, E. Filiot, and A. Trivedi, "Regular transformations of infinite strings," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2012, pp. 65–74.

[AH98] R. Alur and T. A. Henzinger, "Finitary fairness," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 6, pp. 1171–1194, 1998.

[AM04] R. Alur and P. Madhusudan, "Visibly pushdown languages," in *Symposium on Theory of Computing (STOC)*, ACM, 2004, pp. 202–211.

[AM09] R. Alur and P. Madhusudan, "Adding nesting structure to words," *Journal of the ACM (JACM)*, vol. 56, no. 3, p. 16, 2009.

[AW06] S. Andova and T. A. Willemse, "Branching bisimulation for probabilistic systems: Characteristics and decidability," *Theoretical Computer Science (TCS)*, vol. 356, no. 3, pp. 325–355, 2006.

[Ang87] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[APR+01] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2001, pp. 221–234.

[Bai96] C. Baier, "Polynomial time algorithms for testing probabilistic bisimulation and simulation," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 1996, pp. 50–61.

[BH97] C. Baier and H. Hermanns, "Weak bisimulation for fully probabilistic processes," in *International Conference on Computer Aided Verification (CAV)*, Springer, 1997, pp. 119–130.

[BK08] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Programming Language Design and Implementation (PLDI)*, 2001, pp. 203–213.

[BR02] T. Ball and S. K. Rajamani, "The slam project: Debugging system software via static analysis," in *Symposium on Principles of Programming Languages (POPL)*, ACM, 2002, pp. 1–3.

[BFLP08] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci, "Fast: Acceleration from theory to practice," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, no. 5, pp. 401–424, 2008.

[BFLS05] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen, "Flat acceleration in symbolic model checking," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Springer, 2005, pp. 474–488.

[BFL04a] S. Bardin, A. Finkel, and J. Leroux, "Faster acceleration of counter automata in practice," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2004, pp. 576–590.

[BFL04b] S. Bardin, A. Finkel, and J. Leroux, "Faster acceleration of counter automata in practice," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2004, pp. 576–590.

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci, "Fast: Fast acceleration of symbolic transition systems," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2003, pp. 118–121.

[BB04] C. Bartzis and T. Bultan, "Widening arithmetic automata," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2004, pp. 321–333.

[BLSS03] M. Benedikt, L. Libkin, T. Schwentick, and L. Segoufin, "Definable relations and first-order query languages over strings," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 694–751, 2003.

[BBF+13] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Science & Business Media, 2013.

[Ber13] J. Berstel, *Transductions and Context-Free Languages*. Springer-Verlag, 2013.

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5, pp. 505–525, 2007.

[BK11] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2011, pp. 184–190.

[BAS02] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 66, no. 2, pp. 160–177, 2002.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 1999, pp. 193–207.

[BCC+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, 2003.

[BMR13] N. Bjørner, K. McMillan, and A. Rybalchenko, "On solving universally quantified horn clauses," in *International Static Analysis Symposium (SAS)*, Springer, 2013, pp. 105–125.

[BHK+20] A. B. Block Gorman, P. Hieronymi, E. Kaplan, R. Meng, E. Walsberg, Z. Wang, Z. Xiong, and H. Yang, "Continuous regular functions," *Logical Methods in Computer Science (LMCS)*, vol. 16, no. 1, 2020.

[BJK+15] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder, *Decidability of Parameterized Verification* (Synthesis Lectures on Distributed Computing Theory). Morgan & Claypool Publishers, 2015.

[BG00] A. Blumensath and E. Gradel, "Automatic structures," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2000, pp. 51–62.

[BG04] A. Blumensath and E. Grädel, "Finite presentations of infinite structures: Automata and interpretations," *Theory of Computing Systems (TCS)*, vol. 37, no. 6, pp. 641–674, 2004.

[BLW03] B. Boigelot, A. Legay, and P. Wolper, "Iterating transducers in the large (extended abstract)," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2003, pp. 223–235.

[BJW05] B. Boigelot, S. Jodogne, and P. Wolper, "An effective decision procedure for linear arithmetic over the integers and reals," *ACM Transactions on Computational Logic (TOCL)*, vol. 6, no. 3, pp. 614–633, 2005.

[BW94] B. Boigelot and P. Wolper, "Symbolic verification with periodic sets," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 1994, pp. 55–67.

[BW02] B. Boigelot and P. Wolper, "Representing arithmetic constraints with finite automata: An overview," in *International Conference on Logic Programming (ICLP)*, Springer, 2002, pp. 1–20.

[BHKL] B. Bollig, P. Habermehl, C. Kern, and M. Leucker, "Angluin-style learning of NFA," in *International Joint Conference on Artificial Intelligence*, pp. 1004–1009.

[BKK+10] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, "Libalf: The automata learning framework," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2010, pp. 360–364.

[BP13] F. Bonchi and D. Pous, "Checking NFA equivalence with bisimulations up to congruence," *ACM SIGPLAN Notices*, vol. 48, no. 1, pp. 457–468, 2013.

[Boo54] G. Boole, *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities*. Walton and Maberly, 1854, vol. 2.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *International Conference on Concurrency Theory (CONCUR)*, Springer, 1997, pp. 135–150.

[BHRV12] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, "Abstract regular (tree)-model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 14, no. 2, pp. 167–191, 2012.

[BH99] A. Bouajjani and P. Habermehl, "Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations," *Theoretical Computer Science (TCS)*, vol. 221, no. 1-2, pp. 211–250, 1999.

[BHRV06] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, "Abstract regular tree model checking," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 149, no. 1, pp. 37–48, 2006.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar, "Abstract regular model checking," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2004, pp. 372–386.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, "Regular model checking," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2000, pp. 403–418.

[BLW05] A. Bouajjani, A. Legay, and P. Wolper, "Handling liveness properties in (ω-)regular model checking," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 138, no. 3, pp. 101–115, 2005.

[BIK14] M. Bozga, R. Iosif, and F. Konečný, "Safety problems are NP-complete for flat integer programs with octagonal loops," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2014, pp. 242–261.

[BMP15] L. Bozzelli, B. Maubert, and S. Pinchinat, "Uniform strategies, rational relations and jumping automata," *Information and Computation*, vol. 242, pp. 80–107, 2015.

[Bra11] A. R. Bradley, "SAT-based model checking without unrolling," in *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2011, pp. 70–87.

[BM98] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 1998.

[BMS06] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2006, pp. 427–442.

[Bry86] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[Buc63] J. R. Buchi, "Weak second-order arithmetic and finite automata," *Journal of Symbolic Logic*, vol. 28, no. 1, 1963.

[BGP97] T. Bultan, R. Gerber, and W. Pugh, "Symbolic model checking of infinite state systems using presburger arithmetic," in *International Conference on Computer Aided Verification (CAV)*, Springer, 1997, pp. 400–411.

[BGP99] T. Bultan, R. Gerber, and W. Pugh, "Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 4, pp. 747–789, 1999.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, 1994.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} States and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[Cau92] D. Caucal, "On the regular structure of prefix rewriting," *Theoretical Computer Science (TCS)*, vol. 106, no. 1, pp. 61–86, 1992.

[CFI96] G. Cécé, A. Finkel, and S. P. Iyer, "Unreliable channels are easier to verify than perfect channels," *Information and Computation*, vol. 124, no. 1, pp. 20–31, 1996.

[CMST16] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 model checker," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2016, pp. 510–517.

[CR79] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Communications of the ACM (CACM)*, vol. 22, no. 5, pp. 281–283, 1979.

[CCK+15] M. Chapman, H. Chockler, P. Kesseli, D. Kroening, O. Strichman, and M. Tautschnig, "Learning the language of error," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Springer, 2015, pp. 114–130.

[CLR15] É. Charlier, J. Leroy, and M. Rigo, "An analogue of Cobham's theorem for graph directed iterated function systems," *Advances in Mathematics*, vol. 280, pp. 86–120, 2015.

[CNP09] K. Chatzikokolakis, G. Norman, and D. Parker, "Bisimulation for demonic schedulers," in *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, 2009, pp. 318–332.

[CSV13] S. Chaudhuri, S. Sankaranarayanan, and M. Y. Vardi, "Regular real analysis," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2013, pp. 509–518.

[Cha88] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of Cryptology*, vol. 1, no. 1, pp. 65–75, 1988.

[CvBW12] D. Chen, F. van Breugel, and J. Worrell, "On the complexity of computing probabilistic bisimilarity," in *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, 2012, pp. 437–451.

[CFC+09] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, "Learning minimal separating DFA's for compositional verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2009, pp. 31–45.

[CHLR17] Y.-F. Chen, C.-D. Hong, A. W. Lin, and P. Rümmer, "Learning to prove safety over parameterised concurrent systems," in *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Springer, 2017, pp. 76–83.

[CHL+16] Y.-F. Chen, C. Hsieh, O. Lengál, T.-J. Li, M.-H. Tsai, B.-Y. Wang, and F. Wang, "PAC learning-based verification and model synthesis," in *International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 714–724.

[CGMT14] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Ic3 modulo theories via implicit predicate abstraction," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2014, pp. 46–61.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2000, pp. 154–169.

[CGJ+01] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics*, Springer, 2001, pp. 176–194.

[CKSY05] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2005, pp. 570–574.

[CTV08] E. Clarke, M. Talupur, and H. Veith, "Proving ptolemy right: The environment abstraction framework for model checking concurrent systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2008, pp. 33–47.

[CTV06] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2006, pp. 126–141.

[CE81] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, Springer, 1981, pp. 52–71.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[CGB86] E. M. Clarke, O. Grumberg, and M. C. Browne, "Reasoning about networks with many identical finite state processes," in *Symposium on Principles of Distributed Computing (PODC)*, ACM, 1986, pp. 240–248.

[CKNZ11] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*, Springer, 2011, pp. 1–30.

[CL07] T. Colcombet and C. Löding, "Transforming structures by set interpretations," *Logical Methods in Computer Science (LMCS)*, vol. 3, no. 2, paper–4, 2007.

[CJ98] H. Comon and Y. Jurski, "Multiple counters automata, safety analysis and presburger arithmetic," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 1998, pp. 268–279.

[CGK+12] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Cubicle: A parallel SMT-based model checker for parameterized systems," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2012, pp. 718–724.

[CGK+13] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Invariants for finite instances and beyond," in *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Springer, 2013, pp. 61–68.

[CPR07] B. Cook, A. Podelski, and A. Rybalchenko, "Proving thread termination," in *Programming Language Design and Implementation (PLDI)*, 2007, pp. 320–330.

[Cou94] B. Courcelle, "Monadic second-order definable graph transductions: A survey," *Theoretical Computer Science (TCS)*, vol. 126, no. 1, pp. 53–75, 1994.

[CC77] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Symposium on Principles of Programming Languages (POPL)*, ACM, 1977, pp. 238–252.

[CC92] P. Cousot and R. Cousot, "Abstract interpretation and application to logic programs," *The Journal of Logic Programming*, vol. 13, no. 2-3, pp. 103–179, 1992.

[Cra57] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 250–268, 1957.

[DA14] L. D'Antoni and R. Alur, "Symbolic visibly pushdown automata," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2014, pp. 209–225.

[DV17] L. D'Antoni and M. Veanes, "The power of symbolic automata and transducers," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2017, pp. 47–67.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen, "Iterating transducers," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2001, pp. 286–297.

[DCG+16] J. Daniel, A. Cimatti, A. Griggio, S. Tonetta, and S. Mover, "Infinite-state liveness-to-safety via implicit abstraction and well-founded relations," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2016, pp. 271–291.

[DRS03] L. De Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2003, pp. 14–26.

[Del00a] G. Delzanno, "Automatic verification of parameterised cache coherence protocols," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2000, pp. 53–68.

[Del00b] G. Delzanno, "Verification of consistency protocols via infinite-state symbolic model checking," in *Formal Methods for Distributed System Development*, Springer, 2000, pp. 171–186.

[DHS03] S. Derisavi, H. Hermanns, and W. H. Sanders, "Optimal state-space lumping in markov chains," *Information Processing Letters*, vol. 87, no. 6, pp. 309–315, 2003.

[DEP98] J. Desharnais, A. Edalat, and P. Panangaden, "A logical characterization of bisimulation for labeled markov processes," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 1998, pp. 478–487.

[DEP02] J. Desharnais, A. Edalat, and P. Panangaden, "Bisimulation for labelled markov processes," *Information and Computation*, vol. 179, no. 2, pp. 163–193, 2002.

[DG08] V. Diekert and P. Gastin, "First-order definable languages," *Logic and Automata*, vol. 2, pp. 261–306, 2008.

[Dij82] E. W. Dijkstra, "Self-stabilization in spite of distributed control," in *Selected Writings on Computing: A Personal Perspective*, Springer, 1982, pp. 41–46.

[Dij84] E. Dijkstra, "Invariance and non-determinacy," *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 312, no. 1522, pp. 491–499, 1984.

[Dil89] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 1989, pp. 197–212.

[Don07] A. F. Donaldson, "Automatic techniques for detecting and exploiting symmetry in model checking," Ph.D. dissertation, University of Glasgow, 2007.

[DKK+12] A. F. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl, "Counterexample-guided abstraction refinement for symmetric concurrent programs," *Formal Methods in System Design (FMSD)*, vol. 41, no. 1, pp. 25–44, 2012.

[EES69] S. Eilenberg, C. C. Elgot, and J. C. Shepherdson, "Sets recognized by n-tape automata," *Journal of Algebra*, vol. 13, no. 4, pp. 447–464, 1969.

[Elg61] C. C. Elgot, "Decision problems of finite automata design and related arithmetics," *Transactions of the American Mathematical Society*, vol. 98, no. 1, pp. 21–51, 1961.

[EK00] E. A. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *International Conference on Automated Deduction (CADE)*, Springer, 2000, pp. 236–254.

[EK03a] E. A. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Springer, 2003, pp. 247–262.

[EK03b] E. A. Emerson and V. Kahlon, "Model checking guarded protocols," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2003, pp. 361–370.

[EN95] E. A. Emerson and K. S. Namjoshi, "Reasoning about rings," in *Symposium on Principles of Programming Languages (POPL)*, ACM, 1995, pp. 85–94.

[EN98] E. A. Emerson and K. S. Namjoshi, "On model checking for non-deterministic infinite-state systems," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 1998, pp. 70–80.

[EN03] E. A. Emerson and K. S. Namjoshi, "On reasoning about rings," *International Journal of Foundations of Computer Science*, vol. 14, no. 04, pp. 527–549, 2003.

[EH01] J. Engelfriet and H. J. Hoogeboom, "Mso definable string transductions and two-way finite-state transducers," *ACM Transactions on Computational Logic (TOCL)*, vol. 2, no. 2, pp. 216–254, 2001.

[EE04] J. Esparza and K. Etessami, “Verifying probabilistic procedural programs,” in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2004, pp. 16–31.

[EFM99] J. Esparza, A. Finkel, and R. Mayr, “On the verification of broadcast protocols,” in *Symposium on Logic in Computer Science (LICS)*, IEEE, 1999, pp. 352–359.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, “Efficient algorithms for model checking pushdown systems,” in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2000, pp. 232–247.

[FHMV03] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning about Knowledge*. MIT Press, 2003.

[FPPZ04a] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, “Liveness with incomprehensible ranking,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2004, pp. 482–496.

[FPPZ04b] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, “Liveness with invisible ranking,” in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2004, pp. 223–238.

[FCC+08] A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, “Extending automated compositional verification to the full class of ω-regular languages,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2008, pp. 2–17.

[FKP16] A. Farzan, Z. Kincaid, and A. Podelski, “Proving liveness of parameterized programs,” in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2016, pp. 1–12.

[FWSS19] Y. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv, “Inferring inductive invariants from phase structures,” in *International Conference on Computer Aided Verification (CAV)*, Springer, 2019, pp. 405–425.

[FHJ+17] T. Fiedor, L. Holík, P. Janků, O. Lengál, and T. Vojnar, "Lazy automata techniques for WS1S," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2017, pp. 407–425.

[Fin94] A. Finkel, "Decidability of the termination problem for completely specified protocols," *Distributed Computing*, vol. 7, no. 3, pp. 129–135, 1994.

[FL02] A. Finkel and J. Leroux, "How to compose Presburger-accelerations: Applications to broadcast protocols," in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2002, pp. 145–156.

[FS01] A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere!" *Theoretical Computer Science (TCS)*, vol. 256, no. 1-2, pp. 63–92, 2001.

[FWW97] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 9, pp. 27–37, 1997.

[FP01] D. Fisman and A. Pnueli, "Beyond regular model checking," in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2001, pp. 156–170.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Programming Language Design and Implementation (PLDI)*, ACM, 2002, pp. 234–245.

[FQ02] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *Symposium on Principles of Programming Languages (POPL)*, ACM, 2002, pp. 191–202.

[Flo93] R. W. Floyd, "Assigning meanings to programs," in *Program Verification*, Springer, 1993, pp. 65–81.

[Fok18] W. Fokkink, *Distributed Algorithms: An Intuitive Approach*. MIT Press, 2018.

[FJKW18] V. Forejt, P. Jančar, S. Kiefer, and J. Worrell, "Game characterization of probabilistic bisimilarity, and applications to pushdown automata," *Logical Methods in Computer Science (LMCS)*, vol. 14, no. 4, 2018.

[FO97] L. Fribourg and H. Olsén, "Reachability sets of parameterized rings as regular languages," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 9, p. 40, 1997.

[GRV06] P. Ganty, J.-F. Raskin, and L. Van Begin, "A complete abstract interpretation framework for coverability properties of wsts," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2006, pp. 49–64.

[GLMN13] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Learning universally quantified invariants of linear data structures," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2013, pp. 813–829.

[GLMN14] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2014, pp. 69–87.

[GS92] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *Journal of the ACM (JACM)*, vol. 39, no. 3, pp. 675–735, 1992.

[GNRZ08] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli, "Towards SMT model checking of array-based systems," in *International Joint Conference on Automated Reasoning (IJCAR)*, Springer, 2008, pp. 67–82.

[GR10] S. Ghilardi and S. Ranise, "MCMT: A model checker modulo theories," in *International Joint Conference on Automated Reasoning (IJCAR)*, Springer, 2010, pp. 22–29.

[GS97] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 1997, pp. 72–83.

[GLP06] O. Grinchtein, M. Leucker, and N. Piterman, "Inferring network invariants automatically," in *International Joint Conference on Automated Reasoning (IJCAR)*, Springer, 2006, pp. 483–497.

[GV08] O. Grumberg and H. Veith, *25 Years of Model Checking: History, Achievements, Perspectives*. Springer, 2008, vol. 5000.

[GMT08] S. Gulwani, B. McCloskey, and A. Tiwari, "Lifting abstract interpreters to quantified logical domains," in *Symposium on Principles of Programming Languages (POPL)*, ACM, 2008, pp. 235–246.

[GSV09] S. Gulwani, S. Srivastava, and R. Venkatesan, "Constraint-based invariant inference over predicate abstraction," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2009, pp. 120–135.

[GPR11] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *Symposium on Principles of Programming Languages (POPL)*, 2011, pp. 331–344.

[GSM16] A. Gurfinkel, S. Shoham, and Y. Meshman, "Smt-based verification of parameterized systems," in *International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 338–348.

[HHR+12] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar, "Forest automata for verification of heap manipulation," *Formal Methods in System Design (FMSD)*, vol. 41, no. 1, pp. 83–106, 2012.

[HIV08] P. Habermehl, R. Iosif, and T. Vojnar, "A logic of singly indexed arrays," in *International Conference on Logic Programming and Automated Reasoning (LPAR)*, Springer, 2008, pp. 558–573.

[HV05] P. Habermehl and T. Vojnar, “Regular model checking using inference of regular languages,” *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 138, no. 3, pp. 21–36, 2005.

[HJ94] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability,” *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *Symposium on Principles of Programming Languages (POPL)*, ACM, 2002, pp. 58–70.

[HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Software verification with BLAST,” in *International SPIN Workshop on Model Checking of Software*, Springer, 2003, pp. 235–239.

[Her90] T. Herman, “Probabilistic self-stabilization,” *Information Processing Letters*, vol. 35, no. 2, pp. 63–67, 1990.

[HKNP06] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker, “PRISM: A tool for automatic verification of probabilistic systems,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2006, pp. 441–444.

[Hod83] B. R. Hodgson, “Décidabilité par automate fini,” 1983.

[HWZ00] I. Hodkinson, F. Wolter, and M. Zakharyaschev, “Decidable fragments of first-order temporal logics,” in *Annals of Pure and Applied Logic*, Elsevier, 2000.

[HLR+13] L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar, “Fully automated shape analysis based on forest automata,” in *International Conference on Computer Aided Verification (TACAS)*, Springer, 2013, pp. 740–755.

[HLMR19] C.-D. Hong, A. W. Lin, R. Majumdar, and P. Rümmer, "Probabilistic bisimulation for parameterised systems," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2019, pp. 455–474.

[HMU07] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," 2007.

[Iba78] O. H. Ibarra, "Reversal-bounded multi-counter machines and their decision problems," *Journal of the ACM (JACM)*, vol. 25, no. 1, pp. 116–133, 1978.

[IJ90] A. Israeli and M. Jalfon, "Token management schemes and random walks yield self-stabilizing mutual exclusion," in *Symposium on Principles of Distributed Computing (PODC)*, 1990, pp. 119–131.

[JM07] R. Jhala and K. L. McMillan, "Array abstractions from proofs," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2007, pp. 193–206.

[JN00] B. Jonsson and M. Nilsson, "Transitive closures of regular relations for verifying infinite-state systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2000, pp. 220–235.

[JS07] B. Jonsson and M. Saksena, "Systematic acceleration in regular model checking," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2007, pp. 131–144.

[JKD+15] Y. Jung, S. Kong, C. David, B.-Y. Wang, and K. Yi, "Automatically inferring loop invariants via algorithmic learning," *Mathematical Structures in Computer Science*, vol. 25, no. 4, p. 892, 2015.

[KKW10] A. Kaiser, D. Kroening, and T. Wahl, "Dynamic cutoff detection in parameterized concurrent programs," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2010, pp. 645–659.

[KKW14] A. Kaiser, D. Kroening, and T. Wahl, "Lost in abstraction: Monotonicity in multi-threaded programs," in *International Conference on Concurrency Theory (CONCUR)*, Springer, 2014, pp. 141–155.

[KBI+17] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, "Property-directed inference of universal invariants or proving their absence," *Journal of the ACM (JACM)*, vol. 64, no. 1, pp. 1–33, 2017.

[KV94] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory*. MIT press, 1994.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, "Symbolic model checking with rich assertional languages," in *International Conference on Computer Aided Verification (CAV)*, Springer, 1997, pp. 424–435.

[KD17] J. Ketema and A. F. Donaldson, "Termination analysis for GPU kernels," *Science of Computer Programming*, vol. 148, pp. 107–122, 2017.

[KN94] B. Khoussainov and A. Nerode, "Automatic presentations of structures," in *International Workshop on Logic and Computational Complexity (LCC)*, Springer, 1994, pp. 367–392.

[KMO+11] S. Kiefer, A. S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell, "Language equivalence for probabilistic automata," in *International Conference on Computer Aided Verification (CAV)*, Springer, 2011, pp. 526–540.

[KMO+12] S. Kiefer, A. S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell, "APEX: an analyzer for open probabilistic programs," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2012, pp. 693–698.

[KMO+13] S. Kiefer, A. S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell, "Algorithmic probabilistic game semantics," *Formal Methods in System Design (FMSD)*, vol. 43, no. 2, pp. 285–312, 2013.

[KM01] N. Klarlund and A. Møller, *Mona Version 1.4: User Manual*. BRICS, Department of Computer Science, University of Aarhus Denmark, 2001.

[KMS02] N. Klarlund, A. Møller, and M. I. Schwartzbach, "MONA implementation secrets," *International Journal of Foundations of Computer Science*, vol. 13, no. 04, pp. 571–586, 2002.

[Kle51] S. C. Kleene, "Representation of events in nerve nets and finite automata," Rand Project Air Force Santa Monica CA, Tech. Rep., 1951.

[KPIA20] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken, "First-order quantified separators," in *Conference on Programming Language Design and Implementation (PLDI)*, 2020, pp. 703–717.

[Koz12] D. C. Kozen, *Automata and Computability*. Springer Science & Business Media, 2012.

[KS16] D. Kroening and O. Strichman, *Decision Procedures*. Springer, 2016.

[Krs05] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," *Automated verification of infinite state systems*, 2005.

[KTL09] S. Kundu, Z. Tatlock, and S. Lerner, "Proving optimizations correct using parameterized program equivalence," in *Programming Language Design and Implementation (PLDI)*, ACM, 2009, pp. 327–337.

[KM95] R. P. Kurshan and K. L. McMillan, "A structural induction theorem for processes," *Information and Computation*, vol. 117, no. 1, pp. 1–11, 1995.

[Kwi03] M. Kwiatkowska, "Model checking for probability and time: From theory to practice," in *Symposium on Logic in Computer Science (LICS)*, IEEE, 2003, pp. 351–360.

[LB04a] S. K. Lahiri and R. E. Bryant, "Constructing quantified invariants via predicate abstraction," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2004, pp. 267–281.

[LB04b] S. K. Lahiri and R. E. Bryant, “Indexed predicate discovery for unbounded system verification,” in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2004, pp. 135–147.

[LB07] S. K. Lahiri and R. E. Bryant, “Predicate abstraction with indexed predicates,” *ACM Transactions on Computational Logic (TOCL)*, vol. 9, no. 1, 2007.

[LS91] K. G. Larsen and A. Skou, “Bisimulation through probabilistic testing,” *Information and Computation*, vol. 94, no. 1, pp. 1–28, 1991.

[Leg08] A. Legay, “T(O)RMC: A tool for (ω-)regular model checking,” in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2008, pp. 548–551.

[Leg12] A. Legay, “Extrapolating (ω-)regular model checking,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 14, no. 2, pp. 119–143, 2012.

[LMOW08] A. Legay, A. S. Murawski, J. Ouaknine, and J. Worrell, “On automated verification of probabilistic programs,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2008, pp. 173–187.

[LW10] A. Legay and P. Wolper, “On ω-regular model checking,” *ACM Transactions on Computational Logic (TOCL)*, vol. 12, no. 1, p. 2, 2010.

[LR81] D. Lehmann and M. O. Rabin, “On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem,” in *Symposium on Principles of Programming Languages (POPL)*, ACM, 1981, pp. 133–138.

[Lei10] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *International Conference on Logic Programming and Automated Reasoning (LPAR)*, Springer, 2010, pp. 348–370.

[LLMR17] O. Lengál, A. W. Lin, R. Majumdar, and P. Rümmer, "Fair termination for parameterized probabilistic concurrent systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2017, pp. 499–517.

[LS05] J. Leroux and G. Sutre, "Flat counter automata almost everywhere!" In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Springer, 2005, pp. 489–503.

[LR20] A. W. Lin and P. Rümmer, "Regular model checking revisited (technical report)," *arXiv:2005.00990*, 2020.

[Lin12] A. W. Lin, "Accelerating tree-automatic relations," in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[LNRS15] A. W. Lin, T. K. Nguyen, P. Rümmer, and J. Sun, "Regular symmetry patterns," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2015, pp. 455–475.

[LR16] A. W. Lin and P. Rümmer, "Liveness of randomised parameterised systems under arbitrary schedulers," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2016, pp. 112–133.

[Lyn96] N. A. Lynch, *Distributed Algorithms*. Elsevier, 1996.

[LSS94] N. A. Lynch, I. Saias, and R. Segala, "Proving time bounds for randomized distributed algorithms," in *Symposium on Principles of Distributed Computing (PODC)*, 1994, pp. 314–323.

[Mal10] A. Malkis, "Cartesian abstraction and verification of multi-threaded programs," Ph.D. dissertation, University of Freiburg, 2010.

[MIG+21] M. Mann, A. Irfan, A. Griggio, O. Padon, and C. Barrett, "Counterexample-guided prophecy for model checking modulo the theory of arrays," in

International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer, 2021, pp. 113–132.

[MP84] Z. Manna and A. Pnueli, "Adequate proof principles for invariance and liveness properties of concurrent programs," *Science of computer programming*, vol. 4, no. 3, pp. 257–289, 1984.

[MP91] Z. Manna and A. Pnueli, "Completing the temporal picture," *Theoretical Computer Science (TCS)*, vol. 83, no. 1, pp. 97–130, 1991.

[MP12] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Science & Business Media, 2012.

[MHL+20] O. Markgraf, C.-D. Hong, A. W. Lin, M. Najib, and D. Neider, "Parameterized synthesis with safety properties," in *Asian Symposium on Programming Languages and Systems*, Springer, 2020, pp. 273–292.

[Mar08] J. Marques-Silva, "Practical applications of boolean satisfiability," in *International Workshop on Discrete Event Systems*, IEEE, 2008, pp. 74–80.

[MS00] J. Marques-Silva and K. Sakallah, "Boolean satisfiability in electronic design automation," in *Design Automation Conference*, 2000, pp. 675–680.

[McM93] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*, Springer, 1993, pp. 25–60.

[McM06] K. L. McMillan, "Lazy abstraction with interpolants," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2006, pp. 123–136.

[McM08] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2008, pp. 413–427.

[McM18] K. L. McMillan, "Eager abstraction for symbolic model checking," in *International Conference on Computer Aided Verification (CAV)*, Springer, 2018, pp. 191–208.

[MP18] K. L. McMillan and O. Padon, "Deductive verification in decidable fragments with ivy," in *International Static Analysis Symposium (SAS)*, Springer, 2018, pp. 43–55.

[MP20] K. L. McMillan and O. Padon, "Ivy: A multi-modal verification tool for distributed algorithms," in *International Conference on Computer Aided Verification (CAV)*, Springer, 2020, pp. 190–202.

[MP71] R. McNaughton and S. A. Papert, *Counter-Free Automata*. The MIT Press, 1971.

[Mil89] R. Milner, *Communication and Concurrency*. Prentice hall Englewood Cliffs, 1989, vol. 84.

[Mil90] R. Milner, "Operational and algebraic semantics of concurrent processes," in *Formal Models and Semantics*, Elsevier, 1990, pp. 1201–1242.

[Nei10] D. Neider, "Reachability games on automatic graphs," in *International Conference on Implementation and Application of Automata (CIAA)*, Springer, 2010, pp. 222–230.

[Nei14] D. Neider, "Applications of automata learning in verification and synthesis," Ph.D. dissertation, RWTH Aachen, 2014.

[NJ13] D. Neider and N. Jansen, "Regular model checking using solver technologies and automata learning," in *NASa Formal Methods Symposium (NFM)*, 2013, pp. 16–31.

[NT16] D. Neider and U. Topcu, "An automaton learning approach to solving safety games over infinite graphs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2016, pp. 204–221.

[NO79] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 2, pp. 245–257, 1979.

[Nil05] M. Nilsson, “Regular model checking,” Ph.D. dissertation, Uppsala University, 2005.

[Opp80] D. C. Oppen, “Complexity, convexity and combinations of theories,” *Theoretical Computer Science (TCS)*, vol. 12, no. 3, pp. 291–302, 1980.

[PHL+17] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, “Reducing liveness to safety in first-order logic,” *Symposium on Principles of Programming Languages (POPL)*, vol. 1–33, 2017.

[PHM+18] O. Padon, J. Hoenicke, K. L. McMillan, A. Podelski, M. Sagiv, and S. Shoham, “Temporal prophecy for proving temporal properties of infinite-state systems,” in *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Springer, 2018, pp. 1–11.

[PLSS17] O. Padon, G. Losa, M. Sagiv, and S. Shoham, “Paxos made EPR: Decidable reasoning about distributed protocols,” *OOPSLA*, vol. 1, pp. 1–31, 2017.

[PMP+16] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: Safety verification by interactive generalization,” in *Programming Language Design and Implementation (PLDI)*, 2016, pp. 614–630.

[Par81] D. Park, “Concurrency and automata on infinite sequences,” in *Theoretical Computer Science (TCS)*, Springer, 1981, pp. 167–183.

[PLS00] A. Philippou, I. Lee, and O. Sokolsky, “Weak bisimulation for probabilistic systems,” in *International Conference on Concurrency Theory (CONCUR)*, Springer, 2000, pp. 334–349.

[Pla02] S. Planning, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology*, 2002.

[Pnu77] A. Pnueli, “The temporal logic of programs,” in *Symposium on Foundations of Computer Science (SFCS)*, IEEE, 1977, pp. 46–57.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2001, pp. 82–97.

[PS00] A. Pnueli and E. Shahar, "Liveness and acceleration in parameterized verification," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2000, pp. 328–343.

[PXZ02] A. Pnueli, J. Xu, and L. Zuck, "Liveness with $(0, 1, \infty)$-counter abstraction," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2002, pp. 107–122.

[PR07] A. Podelski and A. Rybalchenko, "Transition predicate abstraction and fair termination," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, 15–es, 2007.

[PR12] C. Popeea and A. Rybalchenko, "Compositional termination proofs for multi-threaded programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2012, pp. 237–251.

[PRI] PRISM, Dining Cryptographers. http://www.prismmodelchecker.org.

[QS82] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *International Symposium on Programming*, Springer, 1982, pp. 337–351.

[RG10] S. Ranise and S. Ghilardi, "Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis," *Logical Methods in Computer Science (LMCS)*, vol. 6, 2010.

[Ray13] M. Raynal, *Distributed Algorithms for Message-Passing Systems*. Springer, 2013, vol. 500.

[Rev93] P. Z. Revesz, "A closed-form evaluation for Datalog queries with integer (gap)-order constraints," *Theoretical Computer Science (TCS)*, vol. 116, no. 1, pp. 117–149, 1993.

[RS93] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," *Information and Computation*, vol. 103, no. 2, pp. 299–347, 1993.

[Sai00] H. Saidi, "Model checking guided abstraction and analysis," in *Static Analysis Symposium (SAS)*, Springer, 2000, pp. 377–396.

[Sak09] J. Sakarovitch, *Elements of Automata Theory*. Cambridge University Press, 2009.

[SB04] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 5, no. 2, pp. 185–204, 2004.

[SB06] V. Schuppan and A. Biere, "Liveness checking as safety checking for infinite state spaces," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 149, no. 1, pp. 79–96, 2006.

[SL95] R. Segala and N. Lynch, "Probabilistic simulations for probabilistic processes," *Nordic Journal of Computing*, vol. 2, no. 2, pp. 250–273, 1995.

[SPW09] M. N. Seghir, A. Podelski, and T. Wies, "Abstraction refinement for quantified array assertions," in *Static Analysis Symposium (SAS)*, Springer, 2009, pp. 3–18.

[Sén05] G. Sénizergues, "The bisimulation problem for equational graphs of finite out-degree," *SIAM Journal on Computing*, vol. 34, no. 5, pp. 1025–1106, 2005.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Springer, 2000, pp. 127–144.

[Sif82] J. Sifakis, "A unified approach for studying the properties of transition systems," *Theoretical Computer Science (TCS)*, vol. 18, no. 3, pp. 227–258, 1982.

[Srb04] J. Srba, "Roadmap of infinite results," in *Current Trends in Theoretical Computer Science: The Challenge of the New Century Vol 1: Algorithms and Complexity Vol 2: Formal Models and Semantics*, World Scientific, 2004, pp. 337–350.

[SG09] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *Programming Language Design and Implementation (PLDI)*, 2009, pp. 223–234.

[Suz88] I. Suzuki, "Proving properties of a ring of finite-state machines," *Information Processing Letters*, vol. 28, no. 4, pp. 213–214, 1988.

[Szy88] B. K. Szymanski, "A simple solution to lamport's concurrent programming problem with linear wait," in *International Computer Symposium*, 1988, pp. 621–626.

[TLM+18] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, "Modularity for decidability of deductive verification with applications to distributed systems," in *Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 662–677.

[Tho97] W. Thomas, "Languages, automata, and logic," in *Handbook of Formal Languages*, Springer, 1997, pp. 389–455.

[TL08] A. W. To and L. Libkin, "Recurrent reachability analysis in regular model checking," in *International Conference on Logic Programming and Automated Reasoning (LPAR)*, Springer, 2008, pp. 198–213.

[TL10] A. W. To and L. Libkin, "Algorithmic metatheorems for decidable LTL model checking over infinite systems," in *International Conference on Foun-*

dations of Software Science and Computation Structures (FoSSaCS), Springer, 2010, pp. 221–236.

[Tra77] B. A. Trakhtenbrot, "Finite automata. behavior and synthesis," *Journal of Symbolic Logic*, vol. 42, no. 1, 1977.

[VF10] A. Valmari and G. Franceschinis, "Simple $O(m \log N)$ time markov chain lumping," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2010, pp. 38–52.

[Var06] A. Vardhan, "Learning to verify systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2006.

[VSVA04a] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha, "Actively learning to verify safety for FIFO automata," in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2004, pp. 494–505.

[VSVA04b] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha, "Learning to verify safety properties," in *International Conference on Formal Engineering Methods (ICFEM)*, Springer, 2004, pp. 274–289.

[VSVA05] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha, "Using language inference to verify ω-regular properties," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, 2005, pp. 45–60.

[VV06] A. Vardhan and M. Viswanathan, "Lever: A tool for learning based verification," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 2006, pp. 471–474.

[VV07] A. Vardhan and M. Viswanathan, "Learning to verify branching time properties," *Formal Methods in System Design (FMSD)*, vol. 31, no. 1, pp. 35–61, 2007.

[Var85] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite state programs," in *Symposium on Foundations of Computer Science (SFCS)*, IEEE, 1985, pp. 327–338.

[Var91] M. Y. Vardi, "Verification of concurrent programs: The automata-theoretic framework," *Annals of Pure and Applied Logic*, vol. 51, no. 1-2, pp. 79–98, 1991.

[VW84] M. Y. Vardi and P. Wolper, "Automata theoretic techniques for modal logics of programs," in *Symposium on Theory of Computing (STOC)*, ACM, 1984, pp. 446–456.

[VHL+12] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," in *ACM SIGPLAN Notices*, ACM, vol. 47, 2012, pp. 137–150.

[Voj07] T. Vojnar, *Cut-offs and automata in formal verification of infinite-state systems*, Habilitation Thesis, Faculty of Information Technology, Brno University of Technology, 2007.

[Wika] Wikipedia, *Ticket lock*, https://en.wikipedia.org/wiki/Ticket_lock, (Accessed: 24-10-2020).

[Wikb] Wikipedia, *Water pouring puzzle*, https://en.wikipedia.org/wiki/Water_pouring_puzzle, (Accessed: 24-02-2017).

[Wol00] P. Wolper, "Constructing automata from temporal logic formulas: A tutorial," in *School Organized by the European Educational Forum*, Springer, 2000, pp. 261–277.

[WB98] P. Wolper and B. Boigelot, "Verifying systems with infinite but regular state spaces," in *International Conference on Computer-Aided Verification (CAV)*, Springer, 1998, pp. 88–97.

[YB09] T. Yavuz-Kahveci and T. Bultan, "Action language verifier: An infinite-state model checker for reactive software specifications," *Formal Methods in System Design*, vol. 35, no. 3, pp. 325–367, 2009.

[ZYS+18] L. Zhang, P. Yang, L. Song, H. Hermanns, C. Eisentraut, D. N. Jansen, and J. C. Godskesen, "Probabilistic bisimulation for realistic schedulers," *Acta Informatica*, vol. 55, no. 6, pp. 461–488, 2018.

Appendices

A.1 More liveness-to-safety reduction examples

In this section, we demonstrate the versatility of our safety reduction method in Section 4.3 by modifying the method for two slightly more complex linear-time properties: recurrence and response. Reduction for this two properties have been implemented in our prototype tool along with the reduction for liveness and termination.

Recurrence A recurrence property states that good things should happen infinitely often. In regular model checking, recurrence can be specified as a triple $(\mathscr{I}, \mathscr{T}, \mathscr{F})$ that holds if $\mathfrak{S} \models \Box\Diamond F$ for the regular transition system $\mathfrak{S} := \langle \Sigma^*, I, T \rangle$. Reducing recurrence to safety is essentially the same as reducing liveness, with the only difference lying in how counterexample runs are identified in the reduction. A counterexample to liveness is not supposed to visit any configuration in F, whereas a counterexample to recurrence is allowed to visit F finitely many times. We hence need to modify the definitions in (4.1) to take this difference into account, i.e., by allowing a counterexample path to visit F before entering a cycle. The resulting safety property $(\mathscr{I}', \mathscr{T}', \mathscr{B}')$ has the same I' and B' as in (4.1); only the transition relation T' is different.

$$
\begin{aligned}
I' &= \{(u, \bot^{|u|}) \mid u \in I\} \\
T' &= \{((u,v),(u',v)) \mid (u,u') \in T,\ v \notin \bot^*,\ u \notin F\} \\
&\quad \cup \{((u,\bot^{|u|}),(u',\bot^{|u'|})) \mid (u,u') \in T\} \\
&\quad \cup \{((u,\bot^{|u|}),(u,u)) \mid (u,u') \notin T \text{ for all } u' \in \Sigma^*\} \\
&\quad \cup \{((u,\bot^{|u|}),(u',u)) \mid (u,u') \in T,\ u \notin F\} \\
B' &= \{(u,u) \mid u \notin F\}.
\end{aligned}
$$

Response A response property states that it is always the case that if something bad happens, then something good will happen thereafter. We can specify this property as a quadruple $(\mathscr{I}, \mathscr{T}, \mathscr{E}, \mathscr{F})$ that holds if $\mathfrak{S} \models \Box(E \Rightarrow \Diamond F)$ for the regular system $\mathfrak{S} := \langle \Sigma^*, I, T \rangle$. Reducing a response property to safety is essentially the same as reducing a liveness property to safety; just this time we identify a counterexample

with a lasso-shaped run that stays in $\overline{F}$ after it visits E. The resulting safety property $(\mathscr{I}', \mathscr{T}', \mathscr{B}')$ can be defined over the alphabet $\Sigma_{\top,\perp}$ as follows. Again, only the definition of the transition relation T' is different from those in (4.1).

$$
\begin{aligned}
I' &= \{(u, \perp^{|u|}) \mid u \in I\} \\
T' &= \{((u,v),(u',v)) \mid (u,u') \in T,\ u \notin F,\ v \in \Sigma^* \uplus \top^*\} \\
&\quad \cup \{((u,\perp^{|u|}),(u',\perp^{|u|})) \mid (u,u') \in T\} \\
&\quad \cup \{((u,\perp^{|u|}),(u,u)) \mid (u,u') \notin T \text{ for all } u' \in \Sigma^*\} \\
&\quad \cup \{((u,\perp^{|u|}),(u',u)) \mid (u,u') \in T,\ u \in E,\ u \notin F\} \\
&\quad \cup \{((u,\perp^{|u|}),(u',\top^{|u|})) \mid (u,u') \in T,\ u \in E\} \\
&\quad \cup \{((u,\top^{|u|}),(u',u)) \mid (u,u') \in T,\ u \notin F\} \\
B' &= \{(u,u) \mid u \notin F\}.
\end{aligned}
$$

The correctness of the reduction can be argued similarly to the liveness case in Section 4.3.

A.2 Word representation of regular abstraction

Fix a array system $\mathfrak{T} := (V, \phi_{init}, \phi_{tran})$ and a set of indexed predicates $\mathbb{P}$ over V. Let $r := |V_{int}|$ and $m := |\mathbb{P}|$. Recall from Section 5.3.1 that $\mathbb{P}$ and V together induce an abstract domain $\mathbb{N}^r \times \Sigma_m^*$ for the concrete system $\mathfrak{T}$. A straightforward (but naive) language representation for the abstract domain $\mathbb{N}^r \times \Sigma_m^*$ uses the alphabet $(\Sigma_m \uplus \{\#\})^{r+1}$, where # denotes blank symbol, to encode an abstract state $(p_1, \ldots, p_r, w)$ as a word in the $(r+1)$-convolution $\Sigma_m^* \otimes \cdots \otimes \Sigma_m^*$. A more concise representation is to encode the domain over alphabet $\{1, \#\}^r \times (\Sigma_m \uplus \{\#\})$. An abstract state is then represented as a word in $1^* \otimes \cdots \otimes 1^* \otimes \Sigma_m^*$ such that the first r tracks are unary encodings of $p_1, \ldots, p_r$ and the last track stores the word w.

In our implementation, we adopt a further optimised representation that uses 0 instead of # for padding. In fact, we encode the abstract domain $\mathbb{N}^r \times \Sigma_m^*$ as words in Σ_{r+m}^*. The encoding is defined with a mapping $f : \mathbb{N}^r \times \Sigma_m^* \to \Sigma_{r+m}^*$ such that

$f((p_1,\ldots,p_r,w)) = \hat{p}_1 \odot \cdots \odot \hat{p}_r \odot \hat{w}$. Here, the word $\hat{p}_1 \odot \cdots \odot \hat{p}_r \odot \hat{w}$ is obtained from $p_1 \otimes \cdots \otimes p_r \otimes w$ by replacing # with `0` on the first r tracks, and replacing # with $\mathtt{0}^m$ on the last track. To see that $f(\cdot)$ does provide a legitimate encoding of the abstract domain, consider a regular abstraction $(R,\mathbb{P})$ of a state formula with $R \subseteq \mathbb{N}^r \times \Sigma_m^*$. Note that $f(R) = h(\langle R \rangle)$, where $f(R) := \{f(x) \mid x \in R\}$, $\langle R \rangle$ is the usual language representation of R, and $h : p_1 \otimes \cdots \otimes p_r \otimes w \mapsto \hat{p}_1 \odot \cdots \odot \hat{p}_r \odot \hat{w}$ is a word homomorphism. Moreover, $f(R)$ is a regular language since word homomorphisms preserve language regularity, and $f_{|R}$ is injective since R is an appropriate relation by definition. Therefore, $f(R)$ serves as a legitimate language representation of the regular abstraction $(R,\mathbb{P})$. For transition formulae, the representation of a regular abstraction can be defined in a similarly manner as a regular language over the alphabet $\Sigma_{r+m} \times \Sigma_{r+m}$.

A.3 A modelling language for array systems

In this section, we describe a guarded command language to specify transitions of an array system. We have used this language in our case studies in Section 5.6, where we found it more concise and comprehensible than full-fledged transition formulae for specification purpose. Similar formalisms are common in the literature of concurrent and parameterised systems verification [BGP97; BGP99; GNRZ08; Krs05; YB09].

Recall that an array system is defined with a triple $(V, \phi_{init}, \phi_{tran})$ over the vocabulary σ_A. In this formalism, a transition formula ϕ_{tran} is specified with a couple of guarded commands in form of *guard* $\triangleright$ *update*. A command is *enabled* if its guard is satisfied. When the system reaches a state where no command is enabled, it simply halts. Otherwise, the system will proceed to a new state produced by the update instruction of an enabled command. More precisely, a guarded command is given in form of $\alpha_1,\ldots,\alpha_n \triangleright \beta_1,\ldots,\beta_m$, where each α_i is a $\sigma_A(V)$-literal, and each β_i is a literal in either of the following two forms:

- $t_1 := t_2$, where t_1 is a $\sigma_A(V')$-term, t_2 is a $\sigma_A(V)$-term, and t_1, t_2 have the same sort.

– $x := \mathsf{ndet}(c, d)$, where x is an index variable, and c, d are $\sigma_A(V)$-terms of index sort.

Here, the literals $\alpha_1, \ldots, \alpha_n$ define a condition $\alpha_1 \wedge \cdots \wedge \alpha_n$ for the command to be enabled, and the literals $\beta_1, \ldots, \beta_m$ serve as parallel assignments for the variables in V, indicating how the variables are supposed to be updated by the command. After a command is executed at a state s, the valuations of the variables not assigned a new value will remain the same, while those that are assigned $\mathsf{ndet}(c, d)$ will be updated to a non-deterministic value x such that $s(c) \leq x \leq s(d)$.

A guarded command can be syntactically translated to an abstractable SIA formula capturing the aforementioned semantics. Formally, consider a guarded command $\alpha_1, \ldots, \alpha_n \rhd \beta_1, \ldots, \beta_m$, and let $i \notin V$ be a fresh index variable. Partition the set $\{1, \ldots, m\}$ into X, Y, and Z such that an assignment β_j is in form of $x_j := t_j$ for $j \in X$, in form of $x_j := \mathsf{ndet}(c_j, d_j)$ for $j \in Y$, and in form of $a_j[t_j] := h_j$ for $j \in Z$. Then the command is translated to

$$
\begin{aligned}
&(\textstyle\bigwedge_{j=1}^{n} \alpha_j) \wedge (\bigwedge_{j \in X} x'_j = t_j) \wedge (\bigwedge_{j \in Z} a'_j[t_j] = h_j) \\
&\quad \wedge (\textstyle\bigwedge_{j \in Y} c_j \leq x'_j \wedge x'_j \leq d_j) \wedge (\bigwedge_{x \in P} x' = x) \\
&\quad \wedge \forall i.\, (\textstyle\bigwedge_{a \in Q} 1 \leq i \wedge i \leq |a| \Rightarrow a'[i] = a[i]) \\
&\quad \wedge \forall i.\, (\textstyle\bigwedge_{j \in Z} 1 \leq i \wedge i \leq |a_j| \wedge (\bigwedge_{j \in Z} i \neq t_j) \Rightarrow a'_j[i] = a_j[i]),
\end{aligned}
$$

where $P := V_{int} \setminus \{x_k \mid j \in X \cup Y\}$, $Q := V_{arr} \setminus \{a_j \mid j \in Z\}$, V_{int} is the set of index variables in V, and V_{arr} is the set of array variables in V. A transition formula can be specified with multiple guarded commands: these commands are translated to universal SIA formulae individually, and the final transition formula are formed by disjuncting all the translated SIA formulae.

Example A.3.1. Consider the guarded commands for Selection Sort in Section 5.6.1:

DecHigh : $low \geq high, 1 < high \rhd high := high - 1, low := 1$

IncLow : $low < high, a[low] \leq a[high] \rhd low := low + 1$

Swap : $low < high, a[low] > a[high] \rhd a[high] := a[low], a[low] := a[high], low := low + 1$

These commands can be syntactically translated into the following SIA formulae.

$$\begin{aligned}
DecHigh &:= low \geq high \wedge 1 < high \wedge high' = high - 1 \wedge low' = 1 \\
&\quad \wedge (\forall i.\ 1 \leq i \wedge i \leq |a| \Rightarrow a'[i] = a[i]) \\
IncLow &:= \forall i.\ low < high \wedge a[low] \leq a[high] \wedge low' = low + 1 \wedge high' = high \\
&\quad \wedge (\forall i.\ 1 \leq i \wedge i \leq |a| \Rightarrow a'[i] = a[i]) \\
Swap &:= low < high \wedge a[low] > a[high] \wedge low' = low + 1 \wedge high' = high \wedge a'[high] = a[low] \\
&\quad \wedge a'[low] = a[high] \wedge (\forall i.\ 1 \leq i \wedge i \leq |a| \wedge i \neq low \wedge i \neq high \Rightarrow a'[i] = a[i])
\end{aligned}$$

#

Example A.3.2. In Section 5.6.2, we specified Dijkstra's self-stabilising algorithm as

$$\begin{aligned}
\text{Update}_1 &: pid = 1, x[pid] = x[\mathsf{n}], x[pid] < \mathsf{k} - 1 \triangleright x[pid] := x[pid] + 1, pid := \mathsf{ndet}(1, \mathsf{n}) \\
\text{Update}_2 &: pid = 1, x[pid] = x[\mathsf{n}], x[pid] = \mathsf{k} - 1 \triangleright x[pid] := 0, pid := \mathsf{ndet}(1, \mathsf{n}) \\
\text{Update}_3 &: pid > 1, x[pid] \neq x[pid - 1] \triangleright x[pid] := x[pid - 1], pid := \mathsf{ndet}(1, \mathsf{n})
\end{aligned}$$

These commands can be converted to the following *normalised* SIA formulae.

$$\begin{aligned}
\psi_1 &:= \forall i.\ (i = pid \wedge x[i] = x[\mathsf{n}] \wedge x[i] < \mathsf{k} - 1 \wedge x'[i] = x[i] + 1 \wedge pid = 1 \wedge 1 \leq pid' \wedge pid' \leq \mathsf{n}) \\
&\quad \vee (i \neq pid \wedge 1 \leq i \wedge i \leq |x| \wedge x'[i] = x[i] \wedge pid = 1 \wedge 1 \leq pid' \wedge pid' \leq \mathsf{n}) \\
\psi_2 &:= \forall i.\ (i = pid \wedge x[i] = x[\mathsf{n}] \wedge x[i] = \mathsf{k} - 1 \wedge x'[i] = 0 \wedge pid = 1 \wedge 1 \leq pid' \wedge pid' \leq \mathsf{n}) \\
&\quad \vee (i \neq pid \wedge 1 \leq i \wedge i \leq |x| \wedge x'[i] = x[i] \wedge pid = 1 \wedge 1 \leq pid' \wedge pid' \leq \mathsf{n}) \\
\psi_3 &:= \forall i.\ (i = pid \wedge x[i] \neq x[i-1] \wedge x'[i] = x[i-1] \wedge pid > 1 \wedge 1 \leq pid' \wedge pid' \leq \mathsf{n}) \\
&\quad \vee (i \neq pid \wedge 1 \leq i \wedge i \leq |x| \wedge x'[i] = x[i] \wedge pid > 1 \wedge 1 \leq pid' \wedge pid' \leq \mathsf{n})
\end{aligned}$$

Here, we have additionally normalised the translated formulae according to the requirements stated in Section 5.4. Namely, for each translated formula ψ_i,

- ψ_i is in PNF and the matrix of ψ_i is in DNF.
- ψ_i is in form of $\ldots \forall i \ldots a[i] \ldots$ for some array variable a.

– each value expression in ψ_i has exactly one quantified index variable.

The transition formula of Dijkstra's algorithm is then defined as $\phi_{tran} := \psi_1 \vee \psi_2 \vee \psi_3$. #

We remark that the guard-and-update formalism as presented above is strictly less expressive than a transition formula, which is powerful enough to perform an unbounded number of updates in one step. For example, one can fill an array a with the maximal array element in one step using the (abstractable) transition formula

$$\exists x.\, (\exists i.\, x = a[i]) \wedge (\forall i.\, 1 \leq i \wedge i \leq |a| \Rightarrow (a[i] \leq x \wedge a'[i] = x)).$$

In contrast, a guarded command can only access and update a bounded number of variables and array values. Hence, for example, to locate the maximal element in an array using guarded commands, one may need to introduce auxiliary variables and devise a loop-like control flow to carry out the task. We believe that the latter practice more faithfully reflects the design of realistic array programs. When modelling distributed systems, guarded commands can easily capture most communication primitives based on message passing and shared memory. They however cannot express atomic operations involving an unbounded number of processes, such as message broadcasting and universal global condition checking. This is not a big problem in practice as many of these operations can be simulated non-atomically on message passing systems [Ray13].

Checking non-blocking conditions Recall from Section 5.5.1 that, given an array system $\mathfrak{T}$ and a liveness array property ϕ, we need to check whether $\mathfrak{T}$ is non-blocking with respect to $\neg\phi$ before applying the liveness verification method based on Theorem 5.5.2. For this purpose, we have proposed to consider a safe transition system $\mathfrak{T}^\dagger := (V^\star, \phi^\star_{init}, \phi^\star_{tran}, \phi_{bad})$ and reduce the problem of checking the non-blocking condition to checking the safety of $\mathfrak{T}^\dagger$. When the transition formula of $\mathfrak{T}$ is specified with guarded commands, say $A_1 \rhd B_1, \ldots, A_m \rhd B_m$ with $A_i := \alpha^i_1, \ldots, \alpha^i_{n_i}$, the formula ϕ_{bad} can be defined as $A'_1 \wedge \cdots \wedge A'_m \wedge \neg\psi^\star_2$ where $A'_i := \neg\alpha^i_1 \vee \cdots \vee \neg\alpha^i_{n_i}$. Notably, given that both $\mathfrak{T}$ and the negated property $\neg\phi$ are abstractable (which is precisely the

assumption made in Theorem 5.5.2), the induced system $\mathfrak{T}^{\dagger}$ will also be abstractable. Consequently, the desired non-blocking condition can be checked formally using the safety verification method stated in Theorem 5.5.1.

A.4 Modelling the generalised dining cryptographers

Given parameters $n \geq 3$ and $m \geq 1$, consider a ring of n participants $p_0, \ldots, p_{n-1}$ where each participant p_i possesses a secret bit-vector x_i of size m. The protocol consists of two stages. At stage one, each pair of neighbouring participants p_i and p_{i+1}[1] computes a random vector b_i by tossing a coin for m times. At stage two, each participant p_i announces a vector $a_i := x_i \oplus b_i \oplus b_{i-1}$ to the other participants. After the protocol is carried out, the participants compute the XOR of the secrets as $a_0 \oplus \cdots \oplus a_{n-1} = x_0 \oplus \cdots \oplus x_{n-1}$. Note that both n and m are parameterised in this protocol.

We model the generalised dining cryptographers protocol as a regular PTS. The set of the initial configurations is $I := \{\mathtt{0},\mathtt{1}\}^*(\#\{\mathtt{0},\mathtt{1}\}^*)^*$. Given an initial configuration $w := w_0 \# \cdots \# w_{m-1} \in I$, define $N(w) := \min_{0 \leq i < m} |w_i|$. The configuration w encodes secret bit-vectors $x_0, \ldots, x_{N(w)-1}$ of size m for a ring of $N(w)$ participants. The encoding is specified such that $w_k[i] = x_i[k]$ for $k \in \{0, \ldots, m-1\}$ and $i \in \{0, \ldots, N(w)-1\}$. In other words, the first $N(w)$ bits of segment w_k are $x_0[k], \ldots, x_{N(w)-1}[k]$.

The transition relation consists of seven transitions: observer non-deterministically tossing head (via action head), observer non-deterministically tossing tail (via action tail), non-observer tossing head with probability 0.5 (via action toss), non-observer tossing tail with probability 0.5 (via action toss), participant finishing computing a random vector (via action $\perp$), participant announcing zero (via action 0), and participant announcing one (via action 1). The outcomes of the tosses by the observer are visible (i.e. as actions head and tail), while the outcomes of the tosses by the other participants

[1] All arithmetic operations on the subscripts are performed modulo the number of the participants in the protocol.

are hidden (i.e. as action toss). Recall that a participant needs m successive tosses to compute a random vector of size m. We would refer to such a succession of tosses as a *round* and stipulate that an $\perp$ action must be made after each round.

Fix an arbitrary initial configuration $w := w_0\#\cdots\#w_{m-1}$ and let $n := N(w)$. A maximal trace from configuration w begins with n rounds (each followed by a $\perp$ action) and ends with nm successive announcements. For each $i \in \{0,\ldots,n-1\}$ and $k \in \{0,\ldots,m-1\}$, the k-th toss made in the i-th round updates $w_k[j]$ to $w_k[j] \oplus b_i^k$ for $j \in \{i, i+1\}$, where $b_i^k := \mathtt{1}$ if a head is tossed and $b_i^k := \mathtt{0}$ otherwise. The configuration $u := u_0\#\cdots\#u_{m-1}$ reached from w after n rounds of tosses will satisfy $u_k[i] = x_i[k] \oplus b_i^k \oplus b_{i-1}^k$ for $i \in \{0,\ldots,n-1\}$ and $k \in \{0,\ldots,m-1\}$. Hence for each $i \in \{0,\ldots,n-1\}$,

$$\begin{aligned}(u_0[i],\ldots,u_{m-1}[i]) &= (x_i[0],\ldots,x_i[m-1]) \oplus (b_i^0,\ldots,b_i^{m-1}) \oplus (b_{i-1}^0,\ldots,b_{i-1}^{m-1}) \\ &= x_i \oplus b_i \oplus b_{i-1},\end{aligned}$$

where the b_i's are bit-vectors defined as $b_i := (b_i^0,\ldots,b_i^{n-1})$. The PTS then prints the first n bits of each of the segments $u_0,\ldots,u_{m-1}$ by going through a sequence of announcement transitions via actions $\{a_i^k : 0 \le i < n, 0 \le k < m\}$, such that a_i^k is $\mathtt{1}$ if $u_k[i] = 1$, and a_i^k is $\mathtt{0}$ if $u_k[i] = 0$. Hence, the announcement made by participant p_i is $a_i := (a_i^0,\ldots,a_i^{m-1}) = x_i \oplus b_i \oplus b_{i-1}$ for $i \in \{0,\ldots,n-1\}$.

As before, consider the case where the first participant is the observer. The maximal traces of the PTS in this case are in form of $t_0\cdots t_{n-1}\,t'$, where $t_0, t_{n-1} \in \{\mathsf{head},\mathsf{tail}\}^*\perp$, $t_i \in \mathsf{toss}^*\perp$ for $i \in \{1,\ldots,n-2\}$, and $t' \in \{\mathtt{0},\mathtt{1}\}^*$. To prove anonymity, we define a reference system such that the initial configurations and the actions are the same as those of the original PTS, except that for each $k \in \{0,\ldots,m-1\}$, the announcement bits $a_0^k,\ldots,a_{n-1}^k$ encoded in the maximal trace from an initial configuration w are uniformly distributed over $\{(a_0^k,\ldots,a_{n-1}^k) : a_0^k \oplus\cdots\oplus a_{n-1}^k = w_k[0] \oplus\cdots\oplus w_k[n-1] \text{ and } a_0^k = w_k[0] \oplus b_0^k \oplus b_{n-1}^k\}$. However, note that

$$a_0^k \oplus\cdots\oplus a_{n-1}^k = (a_0 \oplus\cdots\oplus a_{n-1})[k],$$

and

$$w_k[0] \oplus \cdots \oplus w_k[n-1] = (x_0 \oplus \cdots \oplus x_{n-1})[k].$$

It follows that the distribution of the announcements a_i's is independent of the initial configuration once the values of $x_0 \oplus \cdots \oplus x_{n-1}$, x_0, b_0, and b_{n-1} (i.e. the information observed by the first participant) are fixed. We then compute a probabilistic bisimulation over the disjoint union of the original system and the reference system, establishing the anonymity property that the first participant cannot infer the secret bits of the other participants from the information revealed to her.

Remark We have stipulated that an initial configuration $w := w_0\#\cdots\#w_{m-1} \in I$ should encode $N(w) = \min_{0\le i<m} |w_i|$ secrets. Due to the limited expressiveness of regular relations, however, it is impossible for a regular PTS to determine the value of $N(w)$ in a constant number of steps. To remedy, we have simulated the effect of $N(w)$ in our regular PTS as follows: we mark the bits $w_0[i], \ldots, w_{m-1}[i]$ in the i-th round of tosses for each $i \in \{0, \ldots, n-1\}$, and allow the system to enter the announcing stage only if all bits of the segments $w_0, \ldots, w_{m-1}$ are marked. As a consequence, an initial configuration $w := w_0\#\cdots\#w_{m-1}$ has a maximal trace containing the announcement actions only if $|w_k| = N(w)$ for $k \in \{0, \ldots, m-1\}$. The PTS designed in this way is an over-approximation for the original system in the sense that the anonymity property of the former would imply that of the latter.

A.5 Modelling the grades protocol

Given parameters $n \ge 3$ and $m \ge 1$, consider a ring of n participants $p_0, \ldots, p_{n-1}$ and a bound $M = 2^m$ such that each participant p_i possesses a secret $x_i \in \{0, \ldots, M-1\}$. The protocol consists of two stages. At stage one, each pair of neighbouring participants p_i and p_{i+1} computes a random integer $u_i \in \{0, \ldots, M-1\}$ by tossing a coin for m times. At stage two, each participant p_i announces $a_i := (x_i + u_i - u_{i-1}) \bmod M$ to the other participants. It is easy to see that $a_0 + \cdots + a_{n-1} \bmod M = x_0 + \cdots + x_{n-1} \bmod M$.

Particularly, when there exists an integer g such that $(g-1)\cdot n < M$ and $x_0,\ldots,x_{n-1} < g$, it holds that $a_0+\cdots+a_{n-1} \bmod M = x_0+\cdots+x_{n-1}$.

We model the grades protocol as follows. An initial configuration w consists of n bit-vectors $w_0,\ldots,w_{n-1}$ such that $w_i := ((b_i^0,c_i^0),\ldots,(b_i^{m-1},c_i^{m-1})) \in \{0,1\}^{2m}$. We use $(b_i^0,\ldots,b_i^{m-1})$ to encode x_i for $i\in\{0,\ldots,n-1\}$. We say that the configuration w is *compatible* with n integers $u_0,\ldots,u_{n-1}\in\{0,\ldots,M-1\}$ if for $k\in\{0,\ldots,m-1\}$ and $i\in\{0,\ldots,n-1\}$, c_i^k is the parity of the k-th carry in $(x_i+u_i-u_{i-1}) \bmod M$. If w is compatible with $u_0,\ldots,u_{n-1}$, then it holds that

$$(c_i^0,\ldots,c_i^{m-1})\oplus x_i\oplus u_i\oplus u_{i-1} = (x_i+u_i-u_{i-1}) \bmod M.$$

Conversely, given x_i and u_i for $i\in\{0,\ldots,n-1\}$, there exists a unique initial configuration w compatible with $u_0,\ldots,u_{n-1}$. We therefore employ a transition system similar to that of the generalised dining cryptographers protocol (cf. Appendix A.4) to produce announcements $a_0,\ldots,a_{n-1}$. We over-approximate the behaviours of the protocol by allowing an initial configuration to produce announcements based on both compatible and incompatible random integers.

To prove anonymity, we define a reference system of which the initial configurations and the actions are the same as those of the original system. The transitions are specified such that for each $k\in\{0,\ldots,m-1\}$, the announcement bits $a_0^k,\ldots,a_{n-1}^k$ encoded in the maximal trace from an initial configuration are uniformly distributed over $\{(a_0^k,\ldots,a_{n-1}^k) : a_0^k\oplus\cdots\oplus a_{n-1}^k = b_0^k\oplus\cdots\oplus b_{n-1}^k\oplus c_0^k\oplus\cdots\oplus c_{n-1}^k,\ a_0^k = b_0^k\oplus c_0^k\oplus u_0^k\oplus u_{n-1}^k\}$. Note that

$$a_0^k\oplus\cdots\oplus a_{n-1}^k = b_0^k\oplus\cdots\oplus b_{n-1}^k\oplus c_0^k\oplus\cdots\oplus c_{n-1}^k$$

implies that

$$(a_0+\cdots+a_{n-1} \bmod M)[k] = (x_0+\cdots+x_{n-1} \bmod M)[k].$$

The distribution of the announcements a_i's is thus independent of a specific initial configuration once the values of $x_0+\cdots+x_{n-1} \bmod M$, $c_0,\ldots,c_{n-1}$, x_0, u_0, and u_{n-1}

are fixed. However, if we compute a probabilistic bisimulation $R \supseteq \{(w, w) : w \in I\}$ as usual, we would only verify indistinguishability between initial configurations with the same parity $c_0, \ldots, c_{n-1}$ of carry bits. Instead, we strengthen our proof by computing a probabilistic bisimulation $R \supseteq \{(w, w') \in I \times I : \exists n.\ N(w) = N(w') = n \wedge \forall i \in \{0, \ldots, n-1\}.\ x_i = x'_i\}$, where $N(w)$ denotes the number of participants encoded in configuration w. This establishes the desired anonymity property that the first participant cannot infer the secrets of the other participants from the information revealed to her in the protocol.