Automata over Infinite Data Domains: Learnability and Applications in Program Verification and Repair

Hadar Frenkel

Automata over Infinite Data Domains: Learnability and Applications in Program Verification and Repair

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Hadar Frenkel

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tammuz 5781 Haifa June 2021

This research was carried out under the supervision of Prof. Orna Grumberg and Dr. Sarai Sheinvald, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Hadar Frenkel, Orna Grumberg, Corina S. Pasareanu, and Sarai Sheinvald. Assume, guarantee or repair. In *TACAS Part I*, pages 211–227. Springer, 2020.

Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to modeling systems and specifications over infinite data. In *NFM*, pages 1–18. Springer, 2017.

Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to model-checking systems and specifications over infinite data domains. In *J. Automated Reasoning 63*, pages 1077–1101. Springer, 2019.

ACKNOWLEDGEMENTS

I couldn't have written this thesis without the help and guidance of my two advisors, Orna Grumberg and Sarai Sheinvald. You were the best advisors a person could hope for. You were much more to me than academic advisors. You were my home and family for the past seven years. I've learned so much from you. I can only thank you for all you are to me.

I am grateful to Dana Fisman - you considered me a partner while teaching me so much. I enjoyed every minute of our work together. I am also grateful to my co-authors, Sandra Zilles and Corina Pasareanu - thank you for giving me the opportunity to work together, for your faith in me and for our collaboration.

My parents - Michal and adin. My mother, who was my role model, always believed in me and taught me to work hard for what I want. My father, who seeded in me the love of mathematics, the love of knowledge, and always helped me aim as far as I can.

My parents in law - Nechama and Yonatan. Our studies were your goal. Without your help with the kids and your never-ending support I could have never found the time to complete this thesis.

My friends that I've met along the way. I am blessed to have you with me.

And mostly, the one person that was always there for me. You knew when to let go, and when to push me further. You truly believed in me even when I did not. You are my best friend. Thank you for being.

This thesis is dedicated to my sons, Roee-Aaron, Uriah-Samuel and Sinai-Eliya. You are my entire world.

The generous financial help of the Technion, the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau is gratefully acknowledged.

Contents

List of Figures

Abstract

This thesis focuses on different aspects of formal verification of systems over infinite data domains. Applications for such systems can be found in communication systems, e-commerce systems, large data bases and more. A widely used approach in program verification is model-checking. The problem of model-checking is, given a system and a specification, to determine whether the system satisfies the specification (and thus the verification succeeds); or that the system violates the specification (thus verification fails) and some error is found, witnessing the violation. We focus on the automata-theoretic approach to model-checking. In this approach, both the system and the specification are modeled as finite automata, and model-checking is then reduced to reasoning about these automata. However, real-life systems often contain infinitely many different configurations, as they refer to the data in the system, which is unbounded. In this case, the model-checking of such systems does not scale well, and may even become undecidable.

In this thesis we use finite-state automata in order to model different types of such systems. First, we consider ongoing systems over infinite data domains, with respect to temporal specifications. An example for such a system is a server that communicates with an unknown number of clients. We propose a new automaton model that is able to capture the fragment of $\exists^*$-Variable LTL, which is an extension of LTL that allows reasoning about infinite data domains. We use this model to suggest a bounded model-checking algorithm for such systems, and we characterize decidable fragments of the logic, for which we suggest a complete model-checking process.

Next, we consider the model of *communicating systems*, which is most suitable to model communication and security protocols. We exploit the partition of the system into smaller components (e.g. server and clients). We also use the finite automata representation, to suggest a modular verification and repair algorithm that is based on automata learning using the $\mathbf{L}^*$ algorithm.

Finally, we consider *symbolic automata*, whose alphabet is the set of predicates over some Boolean algebra. We study the $\mathbf{L}^*$ algorithm in the context of symbolic automata, since, as we demonstrate, it is widely used in program verification. Next, we study a different learning paradigm, namely *identification is the limit*. To the best of our knowledge, this is the first time that this paradigm is considered in the context of infinite data domains. We suggest an automata learning algorithm for systems with data over the natural or real numbers, and present some complexity results regarding the learnability of symbolic automata under this paradigm.

Chapter 1

Introduction

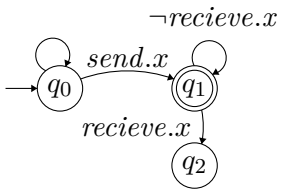
Program verification aims to formally prove that a system is correct with respect to a given specification. It allows proving that the system is correct for all inputs and for all possible behaviors – with respect to the specification – rather than looking for errors using testing. One of the most useful techniques for automated program verification is model checking. The problem of model-checking is defined as follows. Given a system and a specification, automatically determine whether the system satisfies the specification and thus the verification succeeds; or the system violates the specification and the verification fails. In case the verification fails, an error is found, a witness to the violation.

A widely used methodology for model checking is the *automata-theoretic approach* [Var95, VW86]. In the automata-theoretic approach to model checking, the system is modeled by a finite-state automaton over a finite alphabet, whose language matches the set of computations of the system. The specification is modeled as a finite-state automaton over a finite alphabet, whose language is exactly the set of all computations satisfying the specification. Model-checking is then reduced to reasoning about these automata. However, real-life systems are often infinite-state, containing unbounded or infinite amount of data, and modeling them using finite state automata is not straight-forward. One of the main challenges of this thesis is to finitely model systems over infinite data domains.

We briefly describe some examples for automata over large or infinite alphabets. In model-checking of finite systems, the state of the system is represented by a set of properties that hold at this state. These properties are called *atomic propositions* and are denoted by AP. Then, the different configurations of the system are over the alphabet 2^{AP}, and automata over the alphabet 2^{AP} are used in model checking [CGP01] in order to verify temporal properties. Another example, used in string sanitizer algorithms [HLM$^+$11], are automata over predicates on the Unicode alphabet which consists of over a million symbols. An infinite alphabet is used in *event recording automata*, a determinizable class of timed automata [AFH99], in which an alphabet letter consists of both a symbol from a finite alphabet, and a non-negative real number. We refer to such systems as *systems over infinite data domains*.

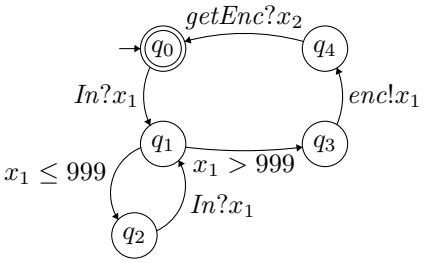
Automata over infinite data domains. We study three different models of finite-state automata, that are used to model systems over infinite data domains (see Figure 1.1). In Chapter 3 we use

Specification: *"there exists a message that is sent but is not eventually received"*

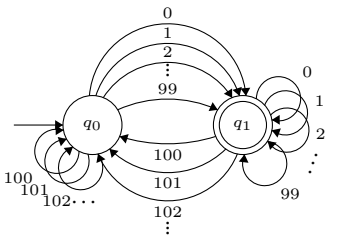


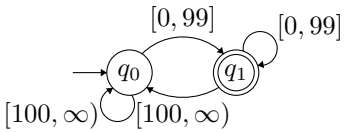
A Variable automaton used to model ongoing communication between server and clients

```
1:  while(true)
2:    x1 = readInput;
3:    while(x1 ≤ 999)
4:      x1 = readInput;
5:    x2 = encrypt(x1);
```



A communicating systems that models a C-like program





A symbolic automaton models infinite alphabet using predicates

Figure 1.1: Different systems studied in this thesis

variable automata over infinite words, in order to model ongoing computations and temporal specifications. The left-most automaton in Figure 1.1 accepts a word if at some position in the word a data value that is assigned to the variable x is sent, but this value is never received later in the computation.

In Chapter 4 we study *communicating systems*, which are a composition of C-like components, with the ability to communicate data between one another. We model such systems using finite automata, each representing the control-flow graph of one of the components of the system. The middle automaton of Figure 1.1 is a component representing the short code given above the automaton. The models of Chapters 3 and 4 use variables in order to keep track of the data values throughout the computation. These models are most suitable to model communication systems such as a network of server and clients, and security protocols.

In Chapter 5 we study symbolic finite-state automata (SFAs) over finite words. There, we use predicates in order to succinctly describe the transitions between states. The model of SFAs has no notion of "memory" and data values are not kept along the computation. At the right-most part of Figure 1.1 we present an SFA (bottom) with predicates labeling transitions. For example, the predicate $[0, 99]$ is used to describe the 100 transitions of type $q_0 \xrightarrow{\mathrm{d}} q_1$ for $0 \leq d \leq 99$, presented in the upper automaton given in the figure.

To summarize the different automata types discussed in this thesis, we first observe that we only consider finite-state automata. We consider finite-state automata over infinite alphabets, as opposed to standard finite-state automata that are defined over a finite alphabet. In addition, in Chapter 3 we consider automata over infinite alphabets and over infinite words, while in

Chapters 4 and 5 we consider automata over infinite alphabets and finite words.

Model-checking and repair. We use the representation of systems over infinite data domains as finite-state automata in order to apply automata-theoretic methods for the verification of such systems. In particular, in Chapter 3 we present a new automata model, that is used to model ongoing systems and specifications over infinite data. Using this new model, we suggest a bounded model-checking algorithm for systems over infinite data domains. In Chapter 4 we use the automata-like representation of communicating systems in order to apply $\mathbf{L}^*$, an automata learning algorithm, to modularly prove the correctness of the system. In case an error is found, we repair the system by eliminating the error, and try to prove correctness of the repaired system.

Learnability. As we mentioned above, in Chapter 4 we apply automata learning in order to verify the correctness of the system. In Chapter 5 we study more fundamental aspects of automata learning. We consider symbolic automata, and study the complexity of $\mathbf{L}^*$ algorithm for these automata. In addition, we study the learning paradigm of *identification in the limit using polynomial time and data*, which has not yet been studied in the context of symbolic automata.

We now elaborate more on each of the approaches.

Model-Checking Systems over Infinite Data

Temporal logic, particularly linear temporal logic (LTL) [Pnu79], is widely used for specifying properties of ongoing systems. However, LTL is unable to specify computations that handle infinite data. Consider, for example, a system containing several processes and a scheduler. If the set of processes is finite and known in advance, we can express and verify properties such as "every process is eventually active". However, if the system is dynamic, in which new processes can log in and out, and the total number of processes is unbounded, LTL is unable to express such a property. This is because, in order to make sure that all processes are active, we need a different ID for each process, resulting in an unbounded amount of data, while LTL is defined over a finite set of propositions.

VLTL (LTL with variables) [GKS12] extends LTL with variables that range over an infinite domain, making it a natural logic for specifying ongoing systems over infinite data domains. In the example above, a VLTL formula can be expressed as $\varphi_1 = \forall x \, \mathsf{G}\,(loggedIn(x) \rightarrow \mathsf{F}\,(active(x)))$, where G and F are temporal operators meaning *Globally* and *eventually*, respectively, and x ranges over an unbounded domain of process IDs.[1] Thus, φ_1 specifies that for every process ID, once it is logged in, it will eventually be active. Notice that φ_1 now specifies this property for an unbounded number of processes. As another example, the formula $\varphi_2 = \mathsf{G}\,\exists x(send(x) \wedge \mathsf{F}\,receive(x))$, where x ranges over the message contents (or message IDs), specifies that in every step of the computation, some message is sent, and this particular message is eventually received. Using variables enables handling infinitely many messages along a single computation.

[1] A formal definition for the semantics of temporal operators appears in Chapter 3.2.1.

For ongoing systems, automata over infinite words and finite alphabets, particularly *nondeterministic* and *alternating Büchi automata* (NBWs and ABWs, respectively) are used [Var95] in order to model-check the system. Thus, for ongoing systems over infinite data and VLTL, a similar model is needed, capable of handling infinite alphabets. In [GKS10, GKS12], the authors suggested *non-deterministic variable Büchi word automata* (NVBWs), a model that augments NBWs with variables. NVBWs were used to construct a model-checking algorithm for a fragment of VLTL, limited to $\exists$-quantifiers that may appear only at the head of the formula.

The emptiness problem for NVBWs is NLOGSPACE-complete. Since the emptiness problem is crucial for model checking, NVBWs are an attractive model. However, they are quite weak. For example, NVBWs are unable to model the formula φ_2 above.

In Chapter 3, we present a new model for VLTL specifications, namely *alternating variable Büchi word automata* (AVBWs). These are an extension of NVBWs, which we prove to be stronger and able to express a much richer fragment of VLTL. Specifically, we show that AVBWs are able to express the entire fragment of $\exists^*$-VLTL, which is a fragment of VLTL in negation normal form (NNF) with only $\exists$-quantifiers, whose positions in the formula is unrestricted.

There is a well-known translation from LTL to ABW [Var95]. Thus, AVBWs are a natural candidate for modeling VLTL. Indeed, as we show, AVBWs are able to express all of $\exists^*$-VLTL, following a translation that is just as natural as the LTL to ABW translation. We further show that, unlike the finite alphabet case, in which NBWs and ABWs are equally expressive, in the infinite alphabet case, alternation proves to be not only syntactically stronger but also semantically stronger, and AVBWs are more expressive than NVBWs.

As we have noted, our goal is to provide a suitable model for a model-checking algorithm for VLTL, and as such, this model should be easily checked for emptiness. However, we show that the strength of AVBWs comes with a price, and their emptiness problem is unfortunately undecidable. To keep the advantage of ease of translation of VLTL to AVBWs, as well as the ease of using NVBWs for model-checking purposes, we would then like to translate AVBWs to NVBWs, in cases where such a translation is possible. This allows us to enjoy the benefit of both models, and gives rise to a model-checking algorithm that is able to handle a richer fragment of VLTL than the one previously studied.

We present such a translation procedure, inspired by the construction of [MH84]. As noted, such a translation is not always possible, and our procedure is then sound but incomplete. However, we give a characterization for AVBWs for which our procedure does halt, relying on the graphical structure of the underlying automaton.

The importance of our procedure and structural characterization is twofold: (1) given an AVBW $\mathcal{A}$, one does not need to know the semantics of $\mathcal{A}$ in order to know if it is translatable, and to automatically translate $\mathcal{A}$ to an equivalent NVBW when possible; and (2) Given a general $\exists^*$-VLTL formula, one can easily construct an equivalent AVBW $\mathcal{A}$, use our characterization to check whether it is translatable, and continue with the NVBW that our translation outputs.

We use our translation from AVBW to NVBW as a basis for a *Bounded Model Checking (BMC)* procedure, even in cases where the translation does not halt. Our BMC procedure exploits the natural iterative behavior of our translation procedure, and the fact that in every

iteration it produces an NVBW whose language is contained in that of the given AVBW. This partial NVBW can then be used for finding an erroneous computation of the system.

As an additional contribution, we characterize fragments of $\exists^*$-VLTL that have a direct translation to NVBWs, making them an "easy" case for modeling and model checking.

The work presented in this chapter was published in [FGS, FGS19].

Summary of Contribution of Chapter 3

1. We present AVBWs, a new model that can capture the whole fragment of $\exists^*$-VLTL and is strictly more expressive than NVBWs.

2. We suggest a partial translation algorithm from AVBWs to NVBWs, that yields a bounded model-checking procedure for VLTL specifications. Moreover, we present a characterization for AVBWs on which our algorithm terminates, allowing us to choose between the bounded verification or the full verification for these cases.

3. We present a characterization of easy fragments of $\exists^*$-VLTL, for which the model-checking problem is decidable. We also suggest a sound and complete model-checking algorithm for these fragments.

Compositional Verification and Repair

In Chapter 4 we turn to investigate more complicated systems, namely *communicating systems*. These are infinite-state C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as finite-state automata over an *action alphabet*, which reflects the program statements. The accepting states in these automata model points of interest in the program that the specification can relate to. The automata representation is similar in nature to that of control-flow graphs. The composition of the two program components, M_1 and M_2, denoted $M_1||M_2$, synchronizes on read-write actions on the same channel. Between two synchronized actions, the individual actions of both systems interleave.

The composition of two components can result in a large system, for which the verification process does not scale well. Verification of large-scale systems is indeed a main challenge in the field of formal verification. *Compositional verification* aims to verify small components of a system separately, and from the correctness of the individual components, to conclude the correctness of the entire system. This, however, is not always possible, since the correctness of a component often depends on the behavior of its environment.

The Assume-Guarantee (AG) style compositional verification [MC81, Pnu85] suggests a solution to this problem. The simplest AG rule checks if a system composed of components M_1 and M_2 satisfies a property P by checking that M_1 under *assumption* A satisfies P and that any system containing M_2 as a component satisfies A. The assumption A is a component that is used to model the environment of M_1. Several frameworks have been proposed to support this style of reasoning. Finding a suitable assumption A is then a common challenge in such frameworks.

In Chapter 4, we present *Assume-Guarantee-Repair* (AGR) – a fully automated framework which applies the Assume-Guarantee rule, and while seeking a suitable assumption A, incrementally repairs the given program in case the verification fails. Our framework is inspired by [PGB+08], which presented a learning-based method to finding an assumption A, using the $\mathbf{L}^*$ [Ang87b] algorithm for learning regular languages. Here, we exploit the representation of the program as a finite automaton in order to apply $\mathbf{L}^*$.

The specifications we consider are also modeled as finite automata, that does not contain assignment actions. It may contain communication actions in order to specify behavioral requirements, as well as constraints over the variables of both system components, that express requirements on their values in various points in the run.

Consider, for example, the programs M_1 and M_2, and the specification P seen in Figure 1.2. M_1 reads a bound b on the number of times an action must be performed in M_2 (this action can be, say, a $push$ action on a stack). The variable act in M_2 counts the number of times the action has been performed. M_2 performs a sequence of actions, and then reads a value – b – from M_1 through the channel C. If the number of actions M_2 has performed matches b, then M_2 finishes the current iteration successfully. The property P makes sure that in the parallel run of the programs, the number of actions never exceeds b, and that this number eventually reaches b in every iteration. The $sync$ actions here denote communication actions on which the components synchronize, and are used for the clarity of the description. Notice that P expresses temporal requirements that contain unquantified first order constraints.

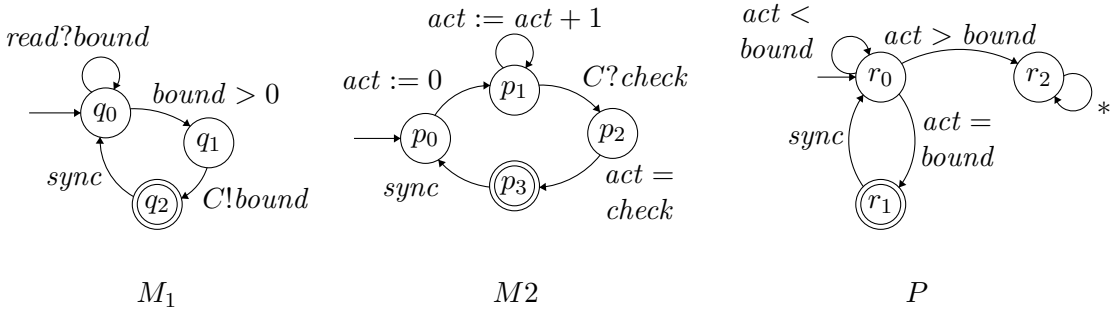


Figure 1.2: The programs M_1 and M_2, and the specification P

The $\mathbf{L}^*$ algorithm aims at learning a regular language U. Its entities consist of a *teacher* – an oracle who answers *membership queries* ("is the word w in U?") and *equivalence queries* ("is $\mathcal{A}$ an automaton whose language is U?"); and a *learner*, who iteratively constructs a finite deterministic automaton $\mathcal{A}$ for U by submitting a sequence of membership and equivalence queries to the teacher.

In using the $\mathbf{L}^*$ algorithm for learning an assumption A for the AG-rule, membership queries are answered according to the satisfaction of the specification P: If M_1 in parallel with a trace t satisfies P, then the trace t in hand should be in A. Otherwise, t should not be in A. Once the learner constructs a stable system A, it submits an equivalence query. The teacher then checks whether A is a suitable assumption, that is, whether $M_1||A$ satisfies P, and whether the language of M_2 is contained in the language of A. According to the results, the process

either continues or halts with an answer to the verification problem. The learning procedure aims at learning the weakest assumption A_w, which contains all of the traces that in parallel with M_1 satisfy P . The key observation that guarantees termination is that the components in this procedure – M_1,M_2, P and A_w – are all regular.

Our setting is more complicated, since the traces in the components – both the programs and the specification – contain constraints, which are to be checked semantically and not syntactically. These constraints may cause some traces to become infeasible. For example, if a trace contains an assignment $x := 3$ followed by a constraint $x \geq 4$ (modeling an *if* statement), then this trace does not contribute any concrete runs, and therefore does not affect the behavior of the system. Thus, we must add feasibility checks to the process.

Constraints in the specification also pose a difficulty, as satisfiability of a specification is determined by the semantics of the constraints and not just by the syntax of the language, and hence there is more here to check than standard language containment. Moreover, A_w above may no longer be regular, as we prove in Chapter 4.4. However, our method manages to overcome this problem in a way that still guarantees termination in case the verification succeeds, and progress, otherwise.

As we have described above, not only do we construct a learning-based method for the AG-rule for communicating programs, but we also repair the programs in case the verification fails. An AG-rule can either conclude that $M_1||M_2 \vDash P$ (i.e. $M_1||M_2$ satisfies P), or return a real, non-spurious counterexample of a computation of $M_1||M_2$ that violates P. In our case, instead of returning the counterexample, we repair M_2 in a way that eliminates this counterexample. We do so by using *abduction* [PH32] to infer a new constraint which makes the counterexample infeasible.

Following this step we now have an updated component M_2, and we apply the AG-rule again, using information we have gathered in the previous steps. In addition to removing the error trace, we update the alphabet of M_2 with the new constraint.

Thus, AGR operates in a verify-repair loop, where each iteration runs a learning-based process to determine whether the (current) system satisfies P, and if not, eliminates bad behaviors from M_2 while enriching the set of constraints derived from these bad behaviors, which often leads to quicker convergence. In case the current system satisfies P, we return the repaired M_2 together with an assumption A that abstracts M_2 and acts as a smaller proof for the correctness of the system.

We have implemented a tool for AGR and evaluated it on examples of various sizes and of various types of errors. Our experiments show that for most examples, AGR converges and finds a repair after 2-5 iterations of verify-repair. Moreover, our tool generates assumptions that are significantly smaller then the (possibly repaired) M_2, thus constructing a compact and efficient proof of correctness.

The work presented in this chapter was published in [FGPS20].

Summary of Contribution of Chapter 4

1. We present a learning-based Assume-Guarantee-Repair algorithm with the following

properties.

- The AGR algorithm for infinite-state communicating programs, manages to overcome the difficulties such programs present. In particular, our algorithm overcomes the inherent irregularity of the first-order constraints in these programs, and offers syntactic solutions to the semantic problems they impose.
- An algorithm in which the Assume-Guarantee and the Repair procedures intertwine to produce a repaired program which, due to our construction, maintains many of the "good" behaviors of the original program. Moreover, in case the original program satisfies the property, our algorithm is guaranteed to terminate and return this conclusion.
- An incremental learning algorithm that uses query results from previous iterations in learning a new language with a richer alphabet.

2. We apply a novel use of abduction to repair communicating programs over first-order constraints.

3. We have implemented our algorithm, demonstrating the effectiveness of our framework.

Learning Symbolic Automata

Symbolic finite state automata, SFAs for short, are an automata model in which transitions between states correspond to predicates over a domain of concrete alphabet letters. Their purpose is to cope with situations where the domain of concrete alphabet letters is large or infinite. The transitions in an SFA are then predicates over the infinite data domain, allowing to succinctly describe the transition relation. Formally, the transition predicates are defined with respect to an effective Boolean algebra as defined in Chapter 5.1.

SFAs have proven useful in many applications [DVLM14, PGLM15, ASJ$^+$16, HD17, SV17, MRA$^+$17], and consequently have been studied as a theoretical model of automata.

Recently, the subject of learning automata in verification has also attracted attention, as it has been shown useful in many applications, see Vaandrager's survey [Vaa17]. Most works consider the query learning paradigm, in which a *learner* tries to learn an automaton by issuing queries to a *teacher*. These works provide extensions to Angluin's $\mathbf{L}^*$ algorithm for learning DFAs using membership and equivalence queries [Ang87a]. In particular, in Chapter 4 we apply $\mathbf{L}^*$ algorithm in order to find assumptions for compositional verification.

In [AD18], the authors study the learnability of SFAs taking as a parameter the learnability of the underlying algebras, providing positive results regarding specific Boolean algebras. One of our contributions is to demonstrate that these positive learnability results are far from trivial. In particular, we show that there are limitations to the power of membership and equivalence queries when it comes to learning SFAs. To do so, we provide a necessary condition for efficient learnability of SFAs in the query learning paradigm, from which we obtain a negative result regarding query learning of SFAs over the propositional algebra. This is, to the best of our

knowledge, the first negative result on learning SFAs with membership and equivalence queries and thus gives useful insights into the limitations of the $\mathbf{L}^*$ framework in this context.

The main focus of our work lies on the learning paradigm of *identification in the limit using polynomial time and data*.[2] We are interested in providing sufficient or necessary conditions for a class of SFAs to be learnable under this paradigm. To this aim, we show that the type of the algebra, in particular whether it is monotonic or not, largely influences the learnability of the class.

Learnability of a class of languages in a certain paradigm greatly depends on the representation chosen for the language. For instance, regular languages are efficiently learnable (in both paradigms) when represented as DFAs but not when represented as NFAs. While we are interested in SFAs as the representations, there are various types of SFAs (with the same expressive power), and the learnabilty results for them may vary.

The literature on SFAs has mainly focused on a special type of SFA, termed *normalized*, in which there is at most one transition between every pair of states. This minimization of the number of transitions comes at the cost of obtaining more complex predicates. We also consider another special type of SFA, which we term a *neat SFA*, which by contrast, allows several transitions between the same pair of states, but restricts the predicates to be *basic*, as formally defined in Section 5.1.1.

To get on the right track, we first take a global look at the complexity of the standard operations on SFAs, and how they vary according to the special form. We revisit the results in the literature and analyze them along the measures we find adequate for a size of an SFA: the number of states, the number of transitions and the size of the most complex predicate.[3] The results show that most procedures are more efficient on neat SFAs. We note that in many applications of learning in verification, the challenging part is implementing the teacher, as we do in Chapter 4. In such cases the complexity of membership and equivalence queries as well as standard automata operations plays a major role.

We then turn to study identification of SFAs in the limit using polynomial time and data. We provide a necessary condition and a sufficient condition a class of SFAs $\mathbb{M}$ should meet in order to be efficiently identifiable in the limit. These conditions are expressed in terms of the existence of certain efficiently computable functions, which we call $\mathsf{Generalize}_{\mathbb{M}}$, $\mathsf{Concretize}_{\mathbb{M}}$, and $\mathsf{Decontaminate}_{\mathbb{M}}$. We then provide positive and negative results regarding learnability of specific classes of SFAs in this paradigm. In particular, we show that general SFAs over the propositional algebra cannot be learned in the limit using polynomial time and data, whereas SFAs over monotonic algebras, such as the interval algebra, can be learned in the limit using polynomial time and data.

Summary of Contribution of Chapter 5

1. We suggest neat SFAs and show that they are more efficient for Boolean operations and membership queries, thus making them good candidates for automata learning algorithms.

[2] This paradigm relates to conformance testing. The relation between conformance testing for Mealy machines and automata learning of DFAs has been explored in [BGJ+05].

[3] Previous results have concentrated mainly on the number of states.

2. We discuss the trade-off between the different types of SFAs and study the complexity of the different automata algorithms for each type.

3. We study the learning paradigm of identification in the limit using polynomial time and data, in the context of symbolic automata. We present a necessary condition and a sufficient condition for a class of SFAs to be efficiently learnable in this paradigm.

4. We present an efficient learning algorithm in the paradigm of identification in the limit, for certain classes of SFAs, for example for SFAs over the interval algebra, while presenting negative results for the learnability of SFAs over the propositional algebra in this paradigm.

5. We present a necessary condition for a class of SFAs to be efficiently learnable using the query learning paradigm, and present the first negative result in that context, for the learnability of SFAs over the propositional algebra.

1.1 Related Work

Translation of standard LTL formulas to automata over infinite words can be found in [BCM$^+$92, RV11, Var95, VW86].

Several other models of automata over infinite alphabets have been defined and studied. In [KF94] the authors define *register automata over infinite alphabets*, and study their decidability properties. [NSV01] use register automata as well as *pebble automata* to reason about first order logic and monadic second order logic, and to describe XML documents. [BMS$^+$06] limits the number of variables and uses extended first order logic to reason about both XML and some verification properties. In [BHJS07] the authors model infinite state systems as well as infinite data domains, in order to express some extension of monadic first order logic. The definition of our model of AVBWs is closer to finite automata over infinite words than the models above, making it easier to understand. Moreover, due to their similarity to ABWs, we were able to construct a natural translation of $\exists^*$-VLTL to AVBWs, inspired by [Var95]. We then translate AVBWs to NVBWs. Our construction is consistent with [MH84] which provides an algorithm for translating ABWs to NBWs. However, in our case additional manipulations are needed in order to handle the variables and track their possible assignments.

The notion of LTL over infinite data domains was studied also in the field of runtime verification (RV) [CM04, BFH$^+$12, BLS11]. Specifically, in [BFH$^+$12], the authors suggest a model of quantified automata with variables, in order to capture traces of computations with different data values. The purpose in runtime verification is to check whether a single given trace satisfies the given specification. Moreover, the traces under inspection are finite traces. This comes into play in [BFH$^+$12] where the authors use the specific data values that appear on such a trace in order to evaluate satisfiability. In [BLS11] the authors suggest a 3-valued semantics in order to capture the uncertainty derived from the fact that traces are finite. LTL with existential and universal quantifiers was also discussed in the context of RV in the following. In both [BKV13] and [DLT16] the authors suggest LTL with first order formulas, and

present monitor construction for this logic. In [BKMZ15] the authors allow unrestricted use of quantifiers and negations, thus using metric first-order temporal logic as a specification language for monitoring system properties. In [HPU17] the authors use Binary Decision Diagrams (BDDs) as an implementation for quantified temporal logic with past operators. The authors of [MJG+12] use different logics, including LTL, in order to efficiently generate monitors for runtime verification.

Our work regarding $\exists^*$-VLTL approaches infinite data domains in a different manner. Since we want to capture both infinite data domains and infinite traces, we need a much more expressive model, and this is where AVBWs come into play.

In the context of compositional verification, assume-guarantee style compositional verification [MC81, Pnu85] has been extensively studied. The assumptions necessary for compositional verification were first produced manually, limiting the practicality of the method.

More recent works [CGP03a, GPB05, GGP07, CS] proposed techniques for automatic assumption generation using learning and abstraction refinement techniques, making assume-guarantee verification more appealing. In [PGB+08, CS] alphabet refinement has been suggested as an optimization, to reduce the alphabet of the generated assumptions, and consequently their sizes. This optimization can easily be incorporated in our AGR framework as well.

Other learning-based approaches for automating assumption generation have been described in [CCF+10, GMF08, CFC+09]. All these works address non-circular rules and are limited to finite state systems. Automatic assumption generation for circular rules is presented in [EGPS15, EGPS16], using compositional rules similar to the ones studied in [McM99, NT00].

Our approach is based on a non-circular rule but it targets complex, infinite-state concurrent systems, and addresses not only verification but also repair. The compositional framework presented in [LH14] addresses $\mathbf{L}^*$-based compositional verification and synthesis but it only targets finite state systems.

Also related is the work in [LDD+13], which addresses automatic synthesis of circular compositional proofs based on logical abduction; however the focus of that work is sequential programs, while here we target concurrent programs. A sequential setting is also considered in [ADG16], where abduction is used for automatically generating a program environment. Our computation of abduction is similar to that of [ADG16]. However, we require our constraints to be over a predefined set of variables, while they look for a minimal set.

The approach presented in [SGP10] aims to compute the *interface* of an infinite-state component. Similar to our work, the approach works with both over- and under- approximations but it only analyzes one component at a time. Furthermore, the component is restricted to be deterministic (necessary for the permissiveness check). In contrast, we use both components of a system to compute the necessary assumptions, and as a result they can be much smaller than in [SGP10]. Furthermore, we do not restrict the components to be deterministic and, mainly, we also address the system repair (in case of dissatisfaction).

A substantial body of literature covers learning restricted forms of SFAs [GJL10, MM14, ASKK16, MM17, CDYS17], as well as general SFAs [DD17, AD18], and even non-deterministic residual SFAs [CHYS19]. For other types of automata over infinite alphabets, [HSM11] sug-

gests learning abstractions, and [She19] presents a learning algorithm for deterministic variable automata. All these works consider the query learning paradigm, and provide extensions to Angluin's $\mathbf{L}^*$ algorithm for learning DFAs using membership and equivalence queries [Ang87a]. Unique to these works is the work [AD18] which studies the learnability of SFAs taking as a parameter the learnability of the underlying algebras, providing positive results regarding specific Boolean algebras. In Chapter 5 we provide the first negative result on learning SFAs from membership and equivalence queries. In addition, we study the learnability of SFAs in the paradigm of identification in the limit, which was not studied before in this context.

Algorithms for other natural questions over SFAs already exist in the literature, in particular, Boolean operations, determinization, and emptiness [VdHT10]; minimization [DV16]; and language inclusion [KT14].

1.2 Thesis Structure

This thesis is constructed as follows. In Chapter 2 we give some basic notions of finite automata and review the paradigm of learning from membership and equivalence queries. In Chapter 3 we discuss model-checking of ongoing systems with respect to Variable LTL specifications. In Chapter 4 we turn to study communicating programs, and present a compositional verification and repair algorithm for this setting. This algorithm is based on learning assumptions using the $\mathbf{L}^*$ algorithm, which we address again in Chapter 5. There, we study the learnability of symbolic automata both in the query learning paradigm and in the paradigm of identification in the limit using polynomial time and data. We conclude in Chapter 6 and discuss some interesting ideas for future work.

Chapter 2

Preliminaries

First, we introduce finite automata and related definitions and notations. Then, we briefly describe the $\mathbf{L}^*$ algorithm for learning regular languages, used in Chapters 4 and 5.

2.1 Finite Automata

In Chapter 4 we introduce communicating systems, and in Chapter 5 we discuss symbolic automata. Both are extensions of finite automata to different types of automata over infinite data domains and finite words.

In Chapter 3 we use automata over infinite words, that extend finite automata with both infinite computations and infinite data domains.

A *finite automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ where

- Σ is a non-empty finite set of *alphabet letters*.
- Q is a non-empty and finite set of *states*.
- $q_0 \in Q$ is the initial state.
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
- $F \subseteq Q$ is the set of final states.

A *deterministic finite automaton* (DFA) is a finite automaton where $\delta : Q \times \Sigma \to Q$ is a function. That is, for every $p \in Q$ and $a \in \Sigma$, if $\langle p, a, q \rangle \in \delta$ and $\langle p, a, q' \rangle \in \delta$ then $q = q'$.

A *word* $w \in \Sigma^*$ is a sequence of alphabet letters. Let $w = \sigma_1 \sigma_2 \cdots \sigma_n$ be a word where $\sigma_i \in \Sigma$ for $1 \leq i \leq n$. We denote by $w[i]$ the letter in position i in w (that is, σ_i), and by w^i the suffix $\sigma_i \sigma_{i+1} \cdots \sigma_n$ of w starting from position i.

A *run* of a finite automaton on a word $w = \sigma_1 \cdots \sigma_n$ is a finite sequence of transitions $\langle q_0, \sigma_1, q_1 \rangle \langle q_1, \sigma_2, q_2 \rangle \ldots \langle q_{n-1}, \sigma_n, q_n \rangle$ where q_0 is the initial state and for each $1 \leq i \leq n-1$ we have $\langle q_{i-1}, \sigma_i, q_i \rangle \in \delta$.

We sometimes omit the letters, and denote a run as the sequence of states $q_0 \ldots q_n$.

We say that a run $r = q_0 \cdots q_n$ on a word w is *accepting* iff $q_n \in F$. If there exists an accepting run of a finite automaton $\mathcal{A}$ on a word w, we say that w is *accepted* by $\mathcal{A}$. Note that for a DFA, there exists a single run for each word w.

The *language* of an automaton $\mathcal{A}$ is the set of all words that are accepted by $\mathcal{A}$, that is, $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \ : \ w \text{ is accepted by } \mathcal{A}\}$.

A language $\mathcal{L}$ is said to be *regular* iff there exists a DFA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

2.2 Automata Learning

In this thesis we consider the exact model of automata learning.[1] In exact learning, given a regular language $\mathcal{L}$, we wish to construct a DFA $\mathcal{D}$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}$. In case the language $\mathcal{L}$ is known and we are given a finite representation of it, for example as a regular expression, then constructing a DFA for $\mathcal{L}$ is relatively an easy task. However, in most cases we do not have a full description of the language. Then, algorithms for learning a DFA for the language $\mathcal{L}$ from a limited amount of information, preferably polynomial, are put into use. In this thesis we consider the paradigm of *query learning*, and in particular the $\mathbf{L}^*$ algorithm [Ang87b], which we briefly describe in Section 2.2.1 below. In addition, we investigate the paradigm of *identification in the limit from polynomial time and data*, which we present in Chapter 5, as we only discuss it in the context of symbolic automata.

2.2.1 Query Learning and L* Algorithm

The $\mathbf{L}^*$ algorithm consists of two entities: a *learner* and a *teacher*, where the goal of the learner is to construct a DFA for an unknown language $\mathcal{L}$; and the role of the teacher is to answer queries issued by the learner, according to the language to be learned. Angluin [Ang87b] showed that the class of regular languages, when represented by DFAs, can be learned in polynomial time using membership and equivalence queries, where the complexity of learning is usually measured by the complexity of operations preformed by the learner, namely the number of membership and equivalence queries that the learner issues.

The language is known to the teacher, who can answer queries and is assumed to have unlimited resources. However, most applications implement both the learner and the teacher (as we demonstrate in Chapter 4), thus the complexity of answering queries needs to be taken into account as well. In Chapter 5 we discuss different procedures for automata that can affect the complexity of answering queries, as well as the number of queries issued by the learner, for the task of learning symbolic automata.

The learner in $\mathbf{L}^*$ can ask two types of queries: *membership queries* (MQs) and *equivalence queries* (EQs). In a membership query, the learner chooses a word $w \in \Sigma^*$, and asks the teacher whether $w \in \mathcal{L}$. The teacher answers *yes/ no* accordingly. After initiating some number of MQs, the learner is able to construct a candidate automaton $\mathcal{A}$, for which it initiates an EQ. In an EQ, given a candidate DFA $\mathcal{A}$, the learner asks asks whether $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. If the answer is *yes*,

[1] As opposed to PAC (Probably approximately correct) learning [CT04, GP16].

then the learning process terminates with the correct DFA for the language. In case the answer is *no*, the teacher provides the learner a counterexample $cex \in \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}$ or $cex \in \mathcal{L} \setminus \mathcal{L}(\mathcal{A})$. Then, the learner uses this counterexample to initiate a new round of membership queries and candidate construction.

We say that a word w is a *positive example* if the teacher answered that $w \in \mathcal{L}$ when a MQ was issued, or if the teacher provided $w \in \mathcal{L}$ when answering an EQ. If the teacher provided $w \notin \mathcal{L}$ either in the MQ or the EQ phase, we call w a *negative example*.

We say that a learning algorithm is *sound*, if, given that S_i^+ and S_i^- are the sets of positive and negative examples provided by the teacher up to stage i, then at stage $i+1$ the learner will not ask a MQ for a word in $S_i^+ \cup S_i^-$. Further, it will not ask an EQ for an automaton that rejects a word in S_i^+ or accepts a word in S_i^-.

A learning algorithm is *efficient* if, in case the teacher answers queries according to a regular language $\mathcal{L}$, then the follwoing are polynomial in the size of the minimal DFA for $\mathcal{L}$:

- The number of MQs in each iteration, until the learner generates a candidate automaton.
- The number of EQs until the learner constructs a DFA $\mathcal{A}$ such that $\mathcal{L}(A) = \mathcal{L}$.

The $\mathbf{L}^*$ algorithm is sound and efficient, and thus in case the unknown language $\mathcal{L}$ is regular, $\mathbf{L}^*$ is guaranteed to terminate, and the number of MQs and EQs is polynomial in the number of states of the minimal DFA for $\mathcal{L}$.

Chapter 3

Model-Checking Systems over Infinite Data

3.1 Systems and Specifications over Infinite Data Domains

In this chapter we discuss the verification of LTL specifications over infinite data domains. We use an extenstion of LTL, namely VLTL (LTL with variables) [GKS12] that extends LTL with variables that range over an infinite domain, making it a natural logic for specifying ongoing systems over infinite data domains. For ongoing systems, automata over infinite words, particularly *nondeterministic* and *alternating Büchi automata* (NBWs and ABWs, respectively) are used [Var95]. Thus, for ongoing systems with infinite data and VLTL, a similar model is needed, capable of handling infinite alphabets. In [GKS10, GKS12], the authors suggested *non-deterministic variable Büchi word automata* (NVBWs), a model that augments NBWs with variables.

In this chapter, we present a new model for VLTL specifications, namely *alternating variable Büchi word automata* (AVBWs). These are an extension of NVBWs, which we prove to be stronger and able to express a much richer fragment of VLTL. Specifically, we show that AVBWs are able to express the entire fragment of $\exists^*$-VLTL, which is a fragment of VLTL in negation normal form (NNF) with only $\exists$-quantifiers, whose positions in the formula are unrestricted.

We now elaborate more on NVBWs and AVBWs. As mentioned, an NVBW $\mathcal{A}$ uses variables that range over an infinite alphabet Γ. A *run* of $\mathcal{A}$ on a word w assigns values to the variables in a way that matches the letters in w. For example, if a letter $a.8$ occurs in w, then a run of $\mathcal{A}$ may read $a.x$, where x is assigned 8. In addition, the variables may be reset at designated states along the run, and so $a.x$ can be later used for reading another letter $a.5$, provided that x has been reset. Resetting then allows reading an unbounded number of letters along a single computation, using a fixed set of variables. Another component of NVBWs is an *inequality set* E, that allows restricting different variables from being assigned the same value at the same point in the computation. Our new model of AVBWs extends NVBWs by adding *alternation*. An alternating automaton may split its run and continue reading the input along several different paths simultaneously, all of which must accept.

There is a well-known translation from LTL to ABWs [Var95]. Thus, AVBWs are a natural candidate for modeling VLTL. Indeed, as we show in Section 3.3.3, AVBWs are able to express all of $\exists^*$-VLTL, following a translation that is just as natural as the LTL to ABW translation. Existential quantifiers (anywhere) in the formula are translated to corresponding resets in the automaton. However, we show that the strength of AVBWs comes with a price, and unlike the finite alphabet case, for infinite data domains AVBWs are not equivalent to NVBWs. However, we present a partial translation algorithm from AVBWs to NVBWs, inspired by the construction of [MH84]. We give a characterization for AVBWs for which our procedure does halt, relying on the graphical structure of the underlying automaton. The essence of the characterization is that translatable AVBWs do not have a cycle that contains a reset action that leads to an accepting state. Consider the specification *"there always exists a message that is currently sent, and each such message will be eventually recieved"*, which corresponds to the VLTL formula $\varphi_2 = \mathsf{G}\,\exists x(send.x \wedge \mathsf{F}\, receive.x)$. Here, we keep sending messages that must arrive eventually. However, there is no bound on when they will arrive. Since this is a global requirement, there must be some cycle that verifies it, and such cycles are exactly the ones that prevent the run of the translation procedure from halting.

We use our translation from AVBW to NVBW as a basis for a *Bounded Model Checking (BMC)* procedure, even in cases where the translation does not halt.

In addition, we characterize fragments of $\exists^*$-VLTL that have a direct translation to NVBWs, making them an "easy" case for modeling and model checking. One such fragment is $\exists^*_{PNF}$-VLTL, which is $\exists^*$-VLTL in prenex normal form. We present a reduction from $\exists^*$-VLTL satisfiability to $\exists^*_{PNF}$-VLTL satisfiability for the fragment of $\exists^*$-VLTL with no negations. This makes $\exists^*$-VLTL with no negations a decidable fragment in terms of the satisfiability problem. Moreover, model-checking for $\exists^*$-VLTL with no negations, as well as for the other fragments we discuss in Section 3.5, is decidable.

3.2 Preliminaries

3.2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [Pnu79] is a specification language that is used to reason about ongoing computations. Let AP be a finite set of atomic propositions. LTL is inductively defined as follows.

- a is an LTL formula for all $a \in AP$
- Let φ_1 and φ_2 be LTL formulas. Then the following are LTL formulas.
  - Boolean operations. $\neg\varphi$; $\varphi_1 \vee \varphi_2$; and $\varphi_1 \wedge \varphi_2$.
  - Temporal operators. $\mathsf{X}\,\varphi_1$; $\mathsf{F}\,\varphi_1$; $\mathsf{G}\,\varphi_1$; $\varphi_1\,\mathsf{U}\,\varphi_2$; $\varphi_1\,\mathsf{V}\,\varphi_2$.

LTL formulas are interpreted over infinite words over the alphabet 2^{AP}. Similar to the definition for finite words, we use $w[i]$ to denote the letter of w in position i, and w^i to denote

the suffix of w starting from position i. Then, each position $w[i]$ is a set of atomic propositions. When w is a computation of a system, then the set $w[i]$ corresponds to all atomic propositions that are true at step i of the computation. The semantics of LTL is defined as follows. Let $w \in (2^{AP})^\omega$.

- For $a \in AP$, we define $w \models a$ iff $a \in w[0]$.
- $w \models \neg\varphi_1$ iff $w \not\models \varphi_1$.
- $w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$; and $w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$.
- $w \models \mathsf{X}\,\varphi_1$ iff $w^1 \models \varphi$.
- $w \models \mathsf{F}\,\varphi_1$ iff there exists $0 \leq i$ such that $w^i \models \varphi$.
- $w \models \mathsf{G}\,\varphi_1$ iff for all $0 \leq i$ it holds that $w^i \models \varphi$.
- $w \models \varphi_1\,\mathsf{U}\,\varphi_2$ iff there exists $0 \leq j$ such that $w^j \models \varphi_2$, and for all $0 \leq i < j$ it holds that $w^i \models \varphi_1$.
- $w \models \varphi_1\,\mathsf{V}\,\varphi_2$ iff one of the following holds.
  - There exists $0 \leq j$ such that $w^j \models \varphi_1$, and for all $0 \leq i \leq j$ it holds that $w^i \models \varphi_2$.
  - For all $0 \leq i$ it holds that $w^i \models \varphi_2$.

Note that the temporal operators G and F can be expressed using U and V as follows.

- $\mathsf{G}\,\varphi \equiv \mathit{false}\,\mathsf{V}\,\varphi$, since this requires φ to hold all along the computation.
- $\mathsf{F}\,\varphi \equiv \mathit{true}\,\mathsf{U}\,\varphi$.

Variable LTL

Variable LTL, or *VLTL*, defined in [GKS12], extends LTL by augmenting atomic propositions with variables that range over a possibly infinite domain. In this context, the set AP is a set of *parameterized* atomic propositions. Let X be a finite set of variables, and let $\bar{x}$ be a vector of variables of X. The formulas in VLTL are over $AP \times X$.

We inductively define the syntax of VLTL.

- For every $a \in AP$ and $x \in X$ the formulas $a.x$ and $\neg a.x$ are VLTL formulas.
- For a VLTL formula $\varphi(\bar{x})$ and $x \in X$, the formulas $\exists x\varphi(\bar{x})$ and $\forall x\varphi(\bar{x})$ are VLTL formulas.
- If $\varphi_1(\bar{x})$ and $\varphi_2(\bar{x})$ are VLTL formulas, then so are $\varphi_1(\bar{x}) \vee \varphi_2(\bar{x})$; $\varphi_1(\bar{x}) \wedge \varphi_2(\bar{x})$; $\mathsf{X}\,\varphi_(\bar{x})$; $\mathsf{F}\,\varphi_1(\bar{x})$; $\mathsf{G}\,\varphi_1(\bar{x})$; $\varphi_1(\bar{x})\,\mathsf{U}\,\varphi_2(\bar{x})$; and $\varphi_1(\bar{x})\,\mathsf{V}\,\varphi_2(\bar{x})$.

Note that in particular, in VLTL we only allow negations of atomic propositions. That is, all VLTL formulas are in Negation Normal Form (NNF).

Semantics Given a (possibly infinite) domain Γ, a quantifier-free formula $\varphi(\bar{x})$, an assignment $\theta : X \to \Gamma$, and a word $w \in (2^{AP \times \Gamma})^\omega$, we denote $w \models_\theta \varphi(\bar{x})$ if $w \models \varphi(\bar{x})_{[\bar{x} \leftarrow \theta(\bar{x})]}$ under the standard semantics of LTL. For example, for $w = \{a.1\}^\omega$ it holds that $w \models_\theta \mathsf{G}\, a.x$ for $\theta(x) = 1$. For $\gamma \in \Gamma$ and $a.\gamma$ we say that γ is the *value of* a. It is important to note that we only allow a single occurrence of $a \in AP$ in every state in the computation, that is, no word may contain both $a.\gamma$ and $a.\gamma'$ for $\gamma \neq \gamma'$ in the same position.

Notice that the semantics of the negation is defined with respect to specific values. That is, if $\theta(x) = 2$ then $\{a.1\}^\omega \models_\theta \mathsf{G} \neg a.x$, but for $\theta'(x) = 1$ we have $\{a.1\}^\omega \not\models_{\theta'} \mathsf{G} \neg a.x$.

We denote $w \models_\theta \exists x \varphi(\bar{x})$ if there exists an assignment $x \leftarrow \gamma$ for some $\gamma \in \Gamma$ such that $w \models_{\theta_{[x \leftarrow \gamma]}} \varphi(\bar{x})$. The assignment $\theta_{[x \leftarrow \gamma]}$ agrees with θ on the values of all variables except the value of x, which is assigned γ, and $\models_{\theta_{[x \leftarrow \gamma]}}$ is as defined before. We denote $w \models_\theta \forall x \varphi(\bar{x})$ if for every assignment $x \leftarrow \gamma$ to the variable x, it holds that $w \models_{\theta_{[x \leftarrow \gamma]}} \varphi(\bar{x})$.

We say that a formula φ is *closed* if every occurrence of a variable in φ is under the scope of a quantifier. Note that the satisfaction of closed formulas is independent of specific assignments. For a closed formula φ over $\bar{x}$, we then write $w \models \varphi(\bar{x})$, instead of $w \models_\theta \varphi(\bar{x})$ for a specific assignment θ.

The logic $\exists^*$-VLTL is the set of all closed VLTL formulas that only use the $\exists$-quantifier. The $\exists$-quantifier may appear anywhere in the formula. The logic $\exists^*_{PNF}$-VLTL is the set of all $\exists^*$-VLTL formulas in prenex normal form, i.e., $\exists$-quantifiers appear only at the beginning of the formula.

The *language* of a formula φ, denoted $\mathcal{L}(\varphi)$, is the set of all computations that satisfy φ.

3.2.2 Automata over infinite words

Non Deterministic Büchi automata

A *non-deterministic Büchi automaton* over infinite words (NBW) [Bue62] is a tuple $\mathcal{B} = \langle \Sigma, Q, q_0, \delta, F \rangle$ where, as in finite automata defined in Chapter 2.1, Σ is a finite alphabet; Q is a finite set of states; $q_0 \in Q$ is the initial state; $F \subseteq Q$ is a set of accepting states; and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

A *run* of $\mathcal{B}$ on a word $w \in \Sigma^\omega$ is an infinite sequence of transitions $\langle q_0, w[1], q_1 \rangle \langle q_1, w[2], q_2 \rangle \ldots$, where for each $i \in \mathbb{N}$, we have $\langle q_i, w[i+1], q_{i+1} \rangle \in \delta$.

A run of $\mathcal{B}$ is *accepting* if it visits some state of F infinitely often. We say that $\mathcal{B}$ *accepts* a word w if there exists an accepting run of $\mathcal{B}$ on w. The *language* of $\mathcal{B}$, denoted $\mathcal{L}(\mathcal{B})$, is the set of words accepted by $\mathcal{B}$.

Alternating Büchi automata

Before defining alternating automata, we define some preliminary notions.

Given a finite set D, a *D-tree* is a set $T \subseteq D^*$ such that T is prefix-closed. That is, if $w \in T$ then $u \in T$ for every prefix u of w. A *node* of T is a word in T, and the *root* of T is the empty word ϵ. A *path* to a node $u = d_1 d_2 \cdots d_n$ in T is the sequence of prefixes $\epsilon, d_1, d_1 d_2, \ldots, d_1 d_2 \cdots d_n$ of u. A *successor* of a node $u \in T$ is of the form $u \cdot d$ where $d \in D$. For $i \geq 0$, the i'th *level* in T is the set of words of length i in T.

Given a set L, an *L-labeled D-tree* is a pair $\langle T, f \rangle$ where T is a D-tree, and $f : T \to L$ is a labeling function that labels each node in T by an element of L.

An *alternating Büchi automaton* over infinite words (ABW) [MS84] is a tuple $\mathcal{B}_A = \langle \Sigma, Q, q_0, \delta, F \rangle$ where Σ, Q, q_0 and F are as in NBW. The transition function is $\delta : Q \times \Sigma \to B^+(Q)$, where $B^+(Q)$ is the set of positive Boolean formulas over the set of states as well as $\{true, false\}$. That is, formulas that include only the Boolean operators $\wedge$ and $\vee$.[1] For example, if $\delta(q, a) = (q_1 \wedge q_2) \vee q_3$, then, by reading a from q, the ABW $\mathcal{B}_A$ moves either to both q_1 and q_2, or to q_3. We assume that δ is given in disjunctive normal form (DNF).[2]

A *run* of $\mathcal{B}_A$ on a word $w \in \Sigma^\omega$ is a Q-labeled Q-tree, in which the i'th level corresponds to the set of states that $\mathcal{B}_A$ reaches after reading $w[i]$. The root is labeled by q_0. For every q-labeled node u in level i, the set of labels of the children of u is a minimal set (with respect to inclusion) that satisfies $\delta(q, w[i])$. For example, if $\delta(q, a) = (q_1 \wedge q_2) \vee q_3$, and $w[i] = a$, and u is a q-labeled node on level $i - 1$, then u has either two children labeled q_1 and q_2, or a single child labeled q_3. Hence, disjunctions in the transitions are equivalent to non-deterministic choices and influence the number of different run trees on a word, and conjunctions induce a split to two or more successors within the run tree.

A run is *accepting* if every infinite path in the run tree visits a state from F infinitely often, and every finite path ends with $true$. That is, the last node on the path is on some level n and is labeled by q such that $true$ satisfies $\delta(q, w[n])$. The notions of acceptance and language are as in NBWs.

We say that an automaton (either NBW or ABW) is a *labeled automaton* if its definition also includes a labeling function $\mathscr{L} : Q \to L$ for its states, where L is a set of labels. We use this notion to conveniently define variable automata.

Non-Deterministic Variable Büchi automata

We now define *non-deterministic variable Büchi automata* over infinite words (NVBWs). Our definition is tailored to model VLTL formulas, and thus is slightly different from the definition in [GKS10]. Specifically, the alphabet consists of subsets of $AP \times X$, where AP is a finite set of parameterized atomic propositions. For ease of presentation we denote the alphabet as $\Sigma = 2^{AP \times X}$. However, we only consider words in which each atomic proposition appears only once at every position of the computation. That is, the set $\{a.x, b.y, c.x\}$ for $a, b, c, \in AP$ can be a letter read by an NVBW, whereas the letter $\{a.x, a.y\}$ is considered only if x and y are assigned with the same concrete value.

[1] In particular, the negation operator is not included.

[2] Note the although δ is a function, this does not imply determinism. We express non-determinism using disjunctions in the Boolean formula that is the output of δ.

An NVBW is a tuple $\mathcal{A} = \langle \mathcal{B}, \Gamma, E \rangle$, where $\mathcal{B} = \langle 2^{AP \times X}, Q, q_0, \delta, reset, F \rangle$ is a labeled NBW such that:

- X is a finite set of variables.
- $reset : Q \to 2^X$ is a labeling function that labels each state q with the set of variables that are reset at q.
- The set $E \subseteq \{x_i \neq x_j \ : \ x_i, x_j \in X\}$ is an *inequality set* over X. This set defines variables that cannot be assigned with the same value at the same point of the computation.
- Γ is an infinite alphabet.

A run of an NVBW $\mathcal{A}$ on a word w assigns a value from Γ to every occurrence of a variable. A variable can "forget" its value only if a reset action occurs. The inequality set E prevents from certain variables to be assigned with the same value at the same state in the computation.

Formally, *a run* of an NVBW $\mathcal{A} = \langle \mathcal{B}, \Gamma, E \rangle$ on a word $w \in (2^{AP \times \Gamma})^\omega$ is a pair $\langle \pi, \theta \rangle$ where $\pi = (q_0, q_1, q_2, \cdots)$, is an infinite sequence of states, and $\theta = (\theta_0, \theta_1, \cdots)$ is an infinite sequence of mappings $\theta_i : X \to \Gamma$ such that:

1. There exists a word $\tilde{w} \in (2^{AP \times X})^\omega$ such that $\theta_i(\tilde{w}[i]) = w[i]$ for every $i \in \mathbb{N}$, and π is a run of $\mathcal{B}$ on $\tilde{w}$. We say that $\tilde{w}$ is a *symbolic word* that is consistent on $\langle \pi, \theta \rangle$ with the *concrete word* w.
2. The run respects the reset actions: for every $i \in \mathbb{N}, x \in X$, if $x \notin reset(q_i)$ then $\theta_i(x) = \theta_{i+1}(x)$.
3. The run respects E: for every $i \in \mathbb{N}$ and for every inequality $(x_m \neq x_l) \in E$ it holds that $\theta_i(x_l) \neq \theta_i(x_m)$. Note that this means that inequalities hold locally at each state.

A run $\langle \pi, \theta \rangle$ on w is *accepting* if π is an accepting run of $\mathcal{B}$ on a symbolic word $\tilde{w}$ that corresponds to w on $\langle \pi, \theta \rangle$, that is, π visits F infinitely often. The notion of acceptance and language are as in NBWs.

We say that an NVBW $\mathcal{A}$ *expresses* a formula φ if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$.

Example 3.2.1. Consider the concrete word

$$w = \{send.1\}(\{send.2, rec.1\}\{send.1, rec.2\})^\omega$$

In an NVBW $\mathcal{A}$, a corresponding symbolic word can be

$$\tilde{w} = \{send.x_1\}(\{send.x_2, rec.x_1\}\{send.x_1, rec.x_2\})^\omega$$

If $\mathcal{A}$ includes reset actions for x_1 and x_2 in every even state in some path of $\mathcal{A}$, then another concrete word consistent with $\tilde{w}$ can be

$$w' = \{send.1\}\{send.2, rec.1\}\{send.3, rec.4\}\{send.4, rec.3\}\{send.5, rec.6\} \ldots$$

since the values of x_1 and x_2 may change at every even state.

Notice that there are several types of infinite sequences discussed in this Chapter. For ease of reading, a summary of their types and notation follows.

- Infinite words over $2^{AP \times \Gamma}$, which we usually denote by w.
- Infinite symbolic words over $2^{AP \times X}$, which we usually denote by $\tilde{w}$.
- Infinite paths in an automaton, which we usually denote by $\pi = (q_0, q_1, \ldots)$.
- Variable assignments, which we usually denote by θ. Sometimes we use θ to refer to a set or a sequence of variable assignments, depending on the context.

3.3 Variable automata: Non-determinism vs. Alternation

In Section 3.6 we show that NVBWs are useful for model checking in our setting, since they have good decidability properties. In particular, there is a polynomial construction for intersection of NVBWs, and their emptiness problem is NLOGSPACE-complete [GKS10]. In Section 3.5.1 we describe fragments of $\exists^*$-VLTL that have a direct translation to NVBWs. We now show that NVBWs are too weak to express all VLTL formulas, or even all $\exists^*$-VLTL formulas. Nevertheless, we use NVBWs for model checking a significant subset of $\exists^*$-VLTL.

Before discussing the properties of variable automata, we first give some motivation for their definition, as presented in Section 3.2.2. In particular, we give motivation for the *reset* labeling function and for E, the inequality set.

Example 3.3.1. We begin with resets. Consider the $\exists^*$-VLTL formula $\varphi_1 = \mathsf{G}\,\exists x(a.x)$. One possible computation that satisfies φ_1 is $w = a.1\ a.2\ a.3 \cdots$. No NVBW with a finite number of variables can read w, unless some variable is reassigned. The reset action allows these reassignments.

Example 3.3.2. To see the necessity of the inequality set E, consider the $\exists^*$-VLTL formula $\varphi_2 = \exists x(G \neg a.x)$. We can use a variable x to store a value that never appears along the computation with a. Imposing inequality restrictions on x with all other variables makes sure that the value assigned to x does not appear along the computation via assignments to other variables. Note that if the logic does not allow negations at all, the inequality set is not needed, since only negations can force values to be different along a computation.

3.3.1 NVBWs are not expressive enough for $\exists^*$-VLTL

As the following Lemma shows, there are $\exists^*$-VLTL formulas that cannot be expressed using an NVBW. This is in contrast to the finite alphabet case, where every LTL formula has an equivalent NBW.

Lemma 3.3.3. *The formula $\varphi_{\mathsf{G}\exists} = \mathsf{G}\,\exists x(a.x \wedge \mathsf{F}\,b.x)$ cannot be expressed by an NVBW.*

Proof. Consider the following word w over $AP = \{a, b\}$ and $\Gamma = \mathbb{N}$, which is defined as follows. For every $i \geq 0$, the letter $w[i]$ is $\{a.(i+1), b.\lfloor \log_2(i+1) \rfloor + 1\}$. Hence,

$$w = \{a.1, b.1\}\{a.2, b.2\}\{a.3, b.2\}\{a.4, b.3\} \cdots \{a.7, b.3\}\{a.8, b.4\} \cdots$$

That is, for every $i > 0$, it holds that $a.(i+1) \in w[i]$, and $b.i$ occurs from $w[2^{i-1} - 1]$ and continues until $w[2^i - 2]$.

w satisfies $\varphi_{\mathsf{G}\exists}$ since for every $t \geq 0$, at every step t, we have that $a.t$ holds, and at some point in the future, specifically at step $2^{t-1} - 1$, the proposition $b.t$ will hold. Thus, at every step t we have $a.t \wedge \mathsf{F}\, b.t$ for some value t.

Intuitively, to see that there exists no NVBW that expresses $\varphi_{\mathsf{G}\exists}$, note that between an occurrence of $a.i$ and $b.i$, the number of different values for a increases as i is increased. Each of these values must be remembered in order to be compared with a future occurrence of b, but an NVBW with finitely many variables cannot handle this requirement.

Formally, assume by way of contradiction that such an NVBW $\mathcal{A}$ does exist, and has m variables. Then when $\mathcal{A}$ runs on a word in which a occurs with more than m values, at least one variable must be reset and used for at least two different values of a during the finite sub-word in which this occurs. Let i_0 be such that $i_0 > \max\{m, 5\}$ and consider the sub-word $w[i_0] \cdots w[i_m]$.[3] Since this is a sub-word of length $m+1$ it holds that some variable x of $\mathcal{A}$ is reset along the run on this sub-word and is used for two different values t, t' of a, such that $t < t'$, and assume that t' is the minimal value satisfying this requirement. In addition, note that since $a.t$ occurs in $w[i_0] \cdots w[i_m]$ it holds that $i_0 \leq t \leq i_m$ and thus $i_m \leq t + m$. Since $b.t$ only occurs at position $2^{t-1} - 1$, and $i_0 > \max\{m, 5\}$, it holds that $2^{t-1} - 1 > t + m$ and thus $b.t$ does not occur in the sub-word $w[i_0] \cdots w[i_m]$.

Now, let w' be the word that is obtained from w above by replacing $a.t$ in $w[t]$ with $a.0$. The word w' does not satisfy $\varphi_{\mathsf{G}\exists}$ since $b.0$ never occurs in w, and thus does not occur in w'. However, we show that since x is reset between position $t-1$ and the next occurrence of $b.t$, the accepting run of $\mathcal{A}$ on w is also an accepting run of $\mathcal{A}$ on w'. This, since $\mathcal{A}$ can no longer check that the occurrence of $a.0$ and the occurrence of $b.t$ do not match; let r be an accepting run of $\mathcal{A}$ on w and let

$$\langle p_0, w[i_0], p_1\rangle \cdots \langle p_t, w[t], p_{t+1}\rangle \cdots \langle p_s, w[s], p_{s+1}\rangle \cdots \langle p_{t'}, w[t'], p_{t'+1}\rangle \cdots \langle p_m, w[i_m], p_{m+1}\rangle$$

the part of r reading the sub-word $w[i_0] \cdots w[i_m]$, where p_s is the state where variable x is reset.

We note that the first time t occurs in w is when reading $a.t$ from state p_t. Thus, all transitions until state p_t remain the same, and the transition $\langle p_t, w'[t], p_{t+1}\rangle$ can be taken since both 0 and t are fresh values, and so they both can be assigned to x. All the transitions that follow up to p_s regard other values and do not impose requirements on x, thus can be taken as well since the rest of the word is unchanged. Since x is reset in p_s, the transition from p_s to p_{s+1} can be taken with any value of x, and in particular with $x = 0$. All other values and all

[3] We choose i_0 to be grater than 5 so that the inequality $2^{t-1} - 1 > t + m$ will hold, as we show below.

atomic propositions remain the same, thus indeed the transition $\langle p_s, w[s], p_{s+1}\rangle$ is valid. Again, all next transitions until $p_{t'}$ read other variable values and thus cannot impose restrictions on x. Now, when reading $w[t']$ the variable x is assigned the new value t', and so from $w[t']$ onwards, the two words w and w' agree on all letters. Therefore, if w is accepted by $\mathcal{A}$ then so is w', and so $\mathcal{A}$ cannot express the property $\varphi_{\mathsf{G}\,\exists}$. ■

Not only $\exists$-quantifiers are problematic for NVBWs. NVBWs cannot handle $\forall$-quantifiers, even in PNF. The proof of the following Lemma is identical to the proof of Lemma 3.3.3.

Lemma 3.3.4. *The formula $\varphi_\forall = \forall x \mathsf{G}(a.x \rightarrow \mathsf{F} b.x)$ cannot be expressed by an NVBW.*

Note that while $\varphi_\forall$ is not expressible by an NVBW, its negation, $\neg\varphi_\forall = \exists x \mathsf{F}\,(a.x \wedge \mathsf{G}\,\neg b.x)$ is expressible using an NVBW (see Section 3.5.1). Since in model-checking we use the negation of the formula we wish to verify, we are able to model-check the property expressed by $\varphi_\forall$. Moreover, we present several techniques for model-checking formulas even if their negations are not expressible by an NVBW. These techniques use AVBWs, and are described in Section 3.3.2.

3.3.2 Alternating Variable Büchi Automata

In Section 3.3.1 we have shown that NVBWs are not expressive enough, even when considering only the fragment of $\exists^*$-VLTL. We now introduce *alternating variable Büchi automata* over infinite words (AVBWs), and show that they can express all of $\exists^*$-VLTL. We study their expressibility and decidability properties.

Definition 3.3.5. An AVBW is a tuple $\mathcal{A} = \langle \mathcal{B}_A, \Gamma, E\,\rangle$ where $\mathcal{B}_A = \langle 2^{AP\times X}, Q, q_0, \delta, \mathit{reset}, F\rangle$ is a labeled ABW, and X, reset, E, and Γ are as in NVBW.

A *run* of an AVBW $\mathcal{A}$ on a word $w \in (2^{AP\times\Gamma})^\omega$ is a pair $\langle T, \theta\rangle$ where T is a Q-labeled Q-tree, and θ associates each node t of T with a function $\theta_t : X \rightarrow \Gamma$ such that:

1. The root of T is labeled with q_0.
2. For every path π of T there exists a symbolic word $\tilde{w}_\pi \in (2^{AP\times X})^\omega$ such that $\theta_{\pi_i}(\tilde{w}_\pi[i]) = w[i]$. That is, for every path π, the word w is obtained from a symbolic word $\tilde{w}_\pi$ that is associated with π, by assigning values to the variables in $\tilde{w}_\pi$ according to θ.
3. The run respects δ: for each node $t \in T$ labeled by q of depth i on path π, the successors of t are labeled by $q_1, \ldots, q_t$ iff $\{q_1, q_2, \ldots, q_t\}$ is a minimal satisfying set of $\delta(q, \tilde{w}_\pi[i])$. That is, the symbolic words that are described in Item 2 can be read following legal transitions in $\mathcal{A}$.
4. The run respects the reset actions: if t' is a child node of t labeled by q and $x \notin \mathit{reset}\,(q)$, then $\theta_t(x) = \theta_{t'}(x)$. That is, along a path in T, as long as x is not reset, it carries the same value.

5. The run respects E: for every $(x_i \neq x_j) \in E$ and for every node $t \in T$ it holds that $\theta_t(x_i) \neq \theta_t(x_j)$.

Intuitively, much like in NVBWs, the variables in every node in every path along the run tree are assigned values in a way that respects the resets along the path, and the inequality set.

A run $\langle T, \theta \rangle$ on w is *accepting* if every infinite path π of T is labeled infinitely often with states in F, and every finite path ends with *true*. The notion of language is as usual.

Note that the same variable can be assigned different values on different paths, even at the same depth of the tree (and also along the same path, provided it has been reset).

Just like ABWs, AVBWs are naturally closed under union and intersection. However, unlike ABWs, they are not closed under complementation. We prove this in Section 3.3.4.

3.3.3 AVBWs can express all of $\exists^*$-VLTL

We now show that AVBWs can express $\exists^*$-VLTL. Together with Lemma 3.3.3, we reach the following surprising theorem.

Theorem 3.1. *AVBWs are strictly more expressive than NVBWs.*

This is in contrast with the finite alphabet case, where there are known algorithms for translating ABWs to NBWs [MH84].

Theorem 3.2. *Every $\exists^*$-VLTL formula φ can be expressed by an AVBW $\mathcal{A}_\varphi$.*

We start with an example AVBW $\mathcal{A}$ for $\varphi_{\mathsf{G}\,\exists} = \mathsf{G}\,\exists x(b.x \wedge \mathsf{F}\,a.x)$ from Lemma 3.3.3. See Figure 3.1 for a graphic representation of $\mathcal{A}$.

Example 3.3.6. Let $\mathcal{A} = \langle \mathcal{B}, \mathbb{N}, \emptyset \rangle$ where $\mathcal{B} = \langle 2^{AP \times \{x_1, x_2, x_3\}}, \{q_0, q_1\}, q_0, \delta, \mathit{reset}, \{q_0\} \rangle$.

- $\mathit{reset}(q_0) = \{x_1, x_2\}$, $\mathit{reset}(q_1) = \{x_2, x_3\}$
- $\delta(q_0, \{b.x_1\}) = \delta(q_0, \{a.x_2, b.x_1\}) = q_0 \wedge q_1$
  $\delta(q_0, \{b.x_1, a.x_1\}) = q_0$
  $\delta(q_1, \{b.x_2\}) = \delta(q_1, \{a.x_2\}) = \delta(q_1, \{a.x_2, b.x_3\}) = q_1$
  $\delta(q_1, \{a.x_1\}) = \delta(q_1, \{a.x_1, b.x_2\}) = \mathit{true}$

Intuitively, q_0 makes sure that at each step there is some value with which b holds. The run then splits to both q_0 and q_1. The state q_1 waits for a with the same value as was seen in q_0 (since x_1 is not reset along this path, it must be the same value), and uses x_2 and x_3 to ignore other values that are attached to a, b. The state q_0 continues to read values of b (which again split the run), while using x_2 to ignore values assigned to a.

We now proceed to the proof of Theorem 3.2.

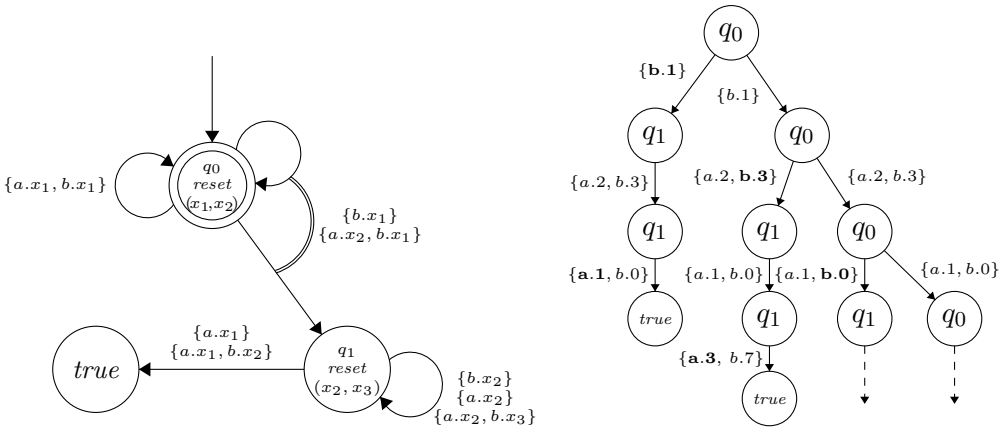


Figure 3.1: The AVBW $\mathcal{A}$ described in Example 3.3.6 and an example of a run. The double arch between transitions represents a conjunction ($\wedge$) in δ.

Proof. Let φ be an $\exists^*$-VLTL formula. We present an explicit construction of $\mathcal{A}_\varphi$, based on the construction of [Var95], using resets to handle the $\exists$-quantifiers, and inequalities to handle negations. First, we rename the variables in φ and get an equivalent formula φ', in which every existential quantifier bounds a variable with a different name. For example, if $\varphi = \exists x(a.x \,\mathsf{U}\, \exists x(b.x))$ then $\varphi' = \exists x_1(a.x_1 \,\mathsf{U}\, \exists x_2(b.x_2))$. In addition, since we can express the temporal operators G and F using U and V, we assume all formulas only contain the temporal operators X, U and V.

Let $sub(\varphi')$ be the set of sub-formulas of φ' and let $var(\varphi')$ be the set of variables that appear in φ'.

The AVBW is $\mathcal{A}_\varphi = \langle \mathcal{B}, \Gamma, E \rangle$ where $\mathcal{B} = \langle 2^{AP \times X}, Q, \varphi', \delta, reset, F \rangle$ and where:

- $X = var(\varphi') \cup \{x_p : \ p \in AP\}$. That is, the variables of $\mathcal{A}_\varphi$ are the set of variables of φ', as well as an additional variable for every atomic proposition. Intuitively, these additional variables allow $\mathcal{A}_\varphi$ to read the values that are carried by atomic propositions, and which are irrelevant to the formula.

- $Q = sub(\varphi')$.

- $\{x_p : \ p \in AP\} \subseteq reset(q)$ for every $q \in Q$, and $\{x_1, \cdots, x_n\} \subseteq reset(q)$ for q of the form $\exists x_1, \cdots, \exists x_n \eta$. That is, x is reset in every state (i.e. sub-formula) in which x is under an un-nested $\exists$ quantifier. Intuitively, the formula states that there exists a value γ that can be assigned to x for which the sub-formula η is true, and resetting x allows it to be assigned γ. In addition, the atomic proposition variables are reset in every state.

- $E = \{x \neq x' : \ x' \in X, (\neg a.x) \in sub(\varphi')\}$. Recall that we only allow negations on atomic propositions. We handle these negations with inequalities. If $\neg a.x$ is a sub-formula of φ, then we do not want the value assigned to x to appear with a when reading a from state $\neg a.x$. Thus, all variables that a can occur with from state $\neg a.x$ must be assigned different values from the value currently assigned to x.

- F consists of all states of the form $\eta \mathsf{V} \psi$, just as in the classic LTL to ABW construction. Intuitively, these states make sure that every U formula is satisfied, and no computation is stuck in the lefthand side of the U without eventually satisfying its righthand side.

As in the classic LTL to ABW construction, the set of states Q consists of all sub-formulas of φ'. In our setting, at every given point in the computation there is an assignment to the variables that may change via resets. If an accepting run of $\mathcal{A}$ on w visits a state ψ, then the suffix of w that is read from ψ satisfies ψ under the current assignment to the variables. The set of variables X consists of all variables in φ', as well as a variable x_p for every atomic proposition $p \in AP$. The additional variables enable the run to read and ignore values that are currently irrelevant. For example, for $\varphi = \exists x \mathsf{F}(b.x \wedge a.x)$, we want to read (and ignore) values of a and b until $a.\gamma \wedge b.\gamma$ occurs with some γ. Along the run, these values can be assigned to the variables x_a, x_b. We proceed to define the transition relation δ. Let A be a subset of $AP \times X$ (recall that $\mathcal{B}$ is defined over the alphabet $2^{AP \times X}$). Then δ is defined as follows.

- $\delta(a.x, A) = \mathit{true}$ if $a.x \in A$ and $\delta(a.x, A) = \mathit{false}$, otherwise.
- $\delta(\neg a.x, A) = \neg\delta(a.x, A)$.[4]
- $\delta(\eta \wedge \psi, A) = \delta(\eta, A) \wedge \delta(\psi, A)$.
- $\delta(\eta \vee \psi, A) = \delta(\eta, A) \vee \delta(\psi, A)$
- $\delta(\mathsf{X}\eta, A) = \eta$
- $\delta(\eta \mathsf{U} \psi, A) = \delta(\psi, A) \vee (\delta(\eta, A) \wedge \eta \mathsf{U} \psi)$
- $\delta(\eta \mathsf{V} \psi, A) = \delta(\eta \wedge \psi, A) \vee (\delta(\psi, A) \wedge \eta \mathsf{V} \psi)$
- $\delta(\exists x \eta, A) = \delta(\eta, A)$

The transition relation δ is then as in the classic LTL to ABW construction, with the additional transitions from states (subformulas) of the type $\exists x\eta$. These serve only to reset x and atomic propositions variables, and the run then proceeds as it would proceed from η. Note that since we only use formulas in NNF, we define δ for both *"and"* and *"or"*, as well as for U (until) and V (release) operators.

Correctness We inductively prove correctness and show that a word w is accepted from a state ψ with a variable assignment θ iff $w \models_\theta \psi$. The correctness of our construction relies on the correctness of the construction in [Var95]. However, we need to take special care in the reset action and existential quantifiers; and in the negations and inequality set.

For a state $a.x$ and a letter $A \in 2^{AP}$, the AVBW accepts and moves to *true* iff $a.x \in A$. Since we only consider closed formulas, we assume x was previously existentially quantified and is assigned with some value. Now, for a state $\neg a.x$, the AVBW moves to *true* iff $a.x \notin A$. However, in that case, if x is assigned with value γ and a occurs with some value $\gamma' \neq \gamma$, then

[4] This can be either *true* or *false*.

the AVBW should allow reading $a.\gamma'$ from state $\neg a.x$. Since the inequality set E is defined to be $E = \{x \neq x' : \ x' \in X, (\neg a.x) \in sub(\varphi')\}$ for $x' \neq x$ and $a.x' \in A$, it holds that $\delta(\neg a.x, A) = true$, and indeed, x' cannot be assigned with the same value as x, so the semantics is preserved.

Inductively, the correctness of δ for temporal and Boolean operators follows from the correctness of the construction of [Var95]. It is left to address existential quantifiers. The $\exists$-quantifier is handled by resetting the variables under its scope. Indeed, according to the semantics of $\exists$, for ψ of the form $\exists x : \psi'$, the suffix of w holds if ψ' holds for some assignment to x. Resetting x allows the run to correctly assign x in a way that satisfies ψ'. Notice also that from this point on, due to the $\exists$ quantifier, the previous value assigned to x may be forgotten.■

3.3.4 AVBWs are not Complementable

As mentioned before, unlike ABWs, AVBWs are not closed under complementation. To prove this, we show that $\forall^*$-VLTL cannot generally be expressed by AVBWs. Since negating an $\exists^*$-VLTL formula produces a $\forall^*$-VLTL formula, the result follows.

Lemma 3.3.7. *There is no AVBW that expresses $\varphi_\forall = \forall x \mathsf{F} a.x$.*

Proof. The formula $\varphi_\forall$ states that all domain values appear somewhere along the computation (with the proposition a). If the alphabet is not countable, then it obviously cannot be enumerated by a computation. However, the claim holds also for countable alphabets. Assume by way of contradiction that there exists an AVBW $\mathcal{A}$ that expresses $\varphi_\forall$ for $\Gamma = \mathbb{N}$. Then $\mathcal{A}$ accepts $w = a.0\ a.1\ a.2 \cdots$. Since the variables are not sensitive to their precise contents but only to inequalities among the values, it holds that the accepting run of $\mathcal{A}$ on w can also be used to read $w^1 = a.1\ a.2 \cdots$, in which the value 0 never occurs. ■

The negation of $\varphi_\forall$ above is in $\exists^*$-VLTL, thus there is an AVBW that expresses $\neg\varphi_\forall$.

Corollary 3.3. *AVBWs are not complementable.*

Corollary 3.4. *$\forall^*$-VLTL is not expressible by AVBWs.*

3.3.5 Variable Automata: From AVBW to NVBW

The emptiness problem for NVBWs is NLOGSPACE-complete [GKS10]. In the context of model checking, this is an important property. We now show that for AVBWs, this problem is undecidable.

Lemma 3.3.8. *The emptiness problem for AVBWs is undecidable.*

Proof. According to [SW], the satisfiability problem for $\exists^*$-VLTL is undecidable. The satisfiability of a formula φ is equivalent to the nonemptiness of an automaton that expresses φ, since a word in the language of the automaton is a satisfying computation of the formula. Since we have shown that every $\exists^*$-VLTL formula can be expressed by an AVBW, the proof follows. ■

Since the emptiness problem for NVBWs is easy, we are motivated to translate AVBWs to NVBWs in order to model-check properties that are expressed by AVBWs. In particular, such a translation will enable us to model-check $\exists^*$-VLTL properties. This, however, is not always possible, since AVBWs are strictly more expressive than NVBWs (Theorem 3.1).

We now present a procedure, which translates an interesting subset of AVBWs to equivalent NVBWs. We later give a structural characterization for AVBWs that can be translated to NVBWs using our procedure.

From AVBW to NVBW

Our procedure is inspired by the construction of [MH84] for translating an ABW $\mathcal{B}$ to an NBW $\mathcal{B}'$. In [MH84] the states of $\mathcal{B}'$ are of the form $\langle S, O\rangle$. Intuitively, the run proceeds in rounds. In every round, every path of the run tree must visit an accepting state at least once. The set S is the set of the states that $\mathcal{B}$ is currently at, and O is the set of states of S that still "owe" a visit to an accepting state. That is, O contains the states from S along paths that have not yet visited an accepting state since the last round. While running $\mathcal{B}'$ on a word w, accepting states are removed from O, until $O = \emptyset$. Thus, when $O = \emptyset$, all paths have visited an accepting state at least once. Now, O is again set to be S, and a new round begins. Accordingly, the accepting states of $\mathcal{B}'$ are states of the form $\langle S, \emptyset\rangle$.

Here, we wish to translate an AVBW $\mathcal{A}$ to an NVBW $\mathcal{A}'$. For simplicity, we assume that $E = \emptyset$. The changes for the case where $E \neq \emptyset$ are described later.

In addition to S and O, we must also remember which variables are currently in use, and might hold values from previous states. In our translation, the states of $\mathcal{A}'$ are tuples containing S, O, and the sets of variables currently in use. Since AVBWs allow different paths to assign different values to the same variable, the translation must allocate a new variable for each such assignment. We also need to release variables that were reset in $\mathcal{A}$, in order to reuse them in $\mathcal{A}'$ to avoid defining infinitely many variables. Since we need to know which variables are in use at each step of a run of $\mathcal{A}$, we dynamically create both the states and δ', the transition function of $\mathcal{A}'$.

Since each path in $\mathcal{A}$ may allocate different values to the same variable, it might be the case that the same variable holds infinitely many values (from different paths). Such a variable induces an unbounded number of variables in $\mathcal{A}'$. The variables make the translation harder, and as stated in Lemma 3.3.3, even impossible in some cases. Our procedure halts when no new states are created, and since the new variables are part of the created states, creating infinitely many such variables causes our procedure not to halt. Therefore, the procedure is incomplete.

Procedure AVBWtoNVBW: Let $\mathcal{A} = \langle \mathcal{B}_\mathcal{A}, \Gamma, E \rangle$ be an AVBW, where $\mathcal{B}_\mathcal{A} = \langle 2^X, Q, q_0, \delta, reset, F\rangle$. For simplicity of the presentation, we assume that $\mathcal{B}_\mathcal{A}$ is defined over the alphabet 2^X instead of $2^{AP\times X}$. Recall that we assume that $\delta(q, X')$ is in DNF for all $q \in Q, X' \subseteq X$. Let $\mathcal{A}' = \langle \mathcal{B}', \Gamma, E' \rangle$ be an NVBW where $\mathcal{B}' = \langle 2^Z, Q', q'_0, \delta', reset', F'\rangle$, such that[5]:

- $Z = \{z_i : \ 0 \leq i < k\}$ is the set of variables.

[5]Comments are given after each item and are preceded by ▷.

▷ k can be finite or infinite, according to the translation. If the algorithm converges then we have $|Z| < \infty$ and the AVBW is translatable to an NVBW.

- $Q' \subseteq 2^{Q \times 2^{X \times Z}} \times 2^{Q \times 2^{X \times Z}}$. The states of $\mathcal{A}'$ are pairs of the form $\langle S, O \rangle$. Each of S, O is a set of pairs of type $\langle q, f_q \rangle$ where $q \in Q$, and $f_q : X \to Z$ is a mapping from the variables of $\mathcal{A}$ to the variables of $\mathcal{A}'$.

  ▷ At each state we need to know how many different values can be assigned to a variable $x \in X$ by different states of $\mathcal{A}$, and create variables in Z accordingly, in order to keep track of the different values of x.

- $q'_0 = \langle \{(q_0, \emptyset)\}, \emptyset \rangle$.

  ▷ The initial state of $\mathcal{A}'$ is the initial state of $\mathcal{A}$ with no additional mappings.

- $F' = 2^{Q \times 2^{X \times Z}} \times \emptyset$.

  ▷ The accepting states of $\mathcal{A}'$ are states for which $O = \emptyset$, i.e., all paths in $\mathcal{A}$ have visited an accepting state.

1. Preprocessing: For each $q \in Q$: if there is no accepting state or *true* reachable from q then replace q with *false*.

   ▷ This is in order to remove loops that may prevent halting, but, in fact, are redundant since they do not lead to an accepting state.

2. Initialization: set $S := \{\langle q_0, \emptyset \rangle\}$; $O := \emptyset$ if $q_0 \in F$ and $O = \{\langle q_0, \emptyset \rangle\}$ otherwise; $Q_{\text{new}} := \{\langle S, O \rangle\}$; $Q_{\text{old}} := \emptyset$; $vars := \emptyset$; $Z := \emptyset$.

   ▷ The purpose of S and O is as explained above; Q_{new} and Q_{old} keep track of the changes in the states that the procedure creates, in order to halt when no new states are created; $vars$ holds variables of Z that are currently in use.

3. We iteratively define $\delta'(\langle S, O \rangle, X')$ for $\langle S, O \rangle \in Q_{\text{new}}$, and denote it by $\langle S', O' \rangle$. We continue as long as new states are created, i.e. while $Q_{\text{new}} \not\subseteq Q_{\text{old}}$.

   (a) Set: $S' := \emptyset$; $O' := \emptyset$; $Z' := \emptyset$; $Z_{reset} := \emptyset$.

   ▷ Z_{reset} contains the set of variables to be reset; at each step, Z' holds the variables in Z that label the current edge (and are the image of the variables in X that label the corresponding edges in $\mathcal{A}$). The set Z_{reset} is initialized at every iteration of the algorithm.

   (b) $Q_{\text{old}} := Q_{\text{old}} \cup \{\langle S, O \rangle\}$

   (c) $Q_{\text{new}} := Q_{\text{new}} \setminus \{\langle S, O \rangle\}$

   (d) For each $\langle q, f_q \rangle \in S$, let $P_q \subseteq Q$ be a minimal set of states such that $P_q \models \delta(q, X')$.

      i. Create a state $\langle p, f_p \rangle$ for each $p \in P_q$. The function f_p is initialized to $f_p(x) := f_q(x)$ for every $x \notin reset(p)$.

      ▷ That is, every successor state p of q remembers the assignments to variables in q, and releases the assignments to variables that were reset in p.

ii. For $x \in X'$ with $x \in dom(f_q)$, update $Z' := Z' \cup \{f_q(x)\}$

iii. For each $x \in X'$ with $x \notin dom(f_q)$, let $i \in \mathbb{N}$ be the minimal index for which $z_i \notin vars$.

A Add to f_p the mapping $f_p(x) := z_i$ if $x \notin reset(p)$.

B Update $vars := vars \cup \{z_i\}$, $Z' := Z' \cup \{z_i\}$, $Z_{reset} := Z_{reset} \cup \{z_i\}$, $Z := Z \cup \{z_i\}$.

▷ z_i may already be in Z, if it was introduced earlier.

iv. Define $S_{P_q} := \{\langle p, f_p\rangle\}_{p \in P_q}$.

v. If $O \neq \emptyset$, set $O_{P_q} := S_{P_q}$ if $\langle q, f_q\rangle \in O$.

▷ That is, add to O' only successor states of states from O.

vi. If $O = \emptyset$, set $O_{P_q} := S_{P_q}$.

(e) Set $S' := \bigcup_{\langle q,f_q\rangle \in S} S_{P_q}$, $O' := (\bigcup_{\langle q,f_q\rangle \in O} O_{P_q}) \setminus \{\langle p, f_p\rangle\}_{p \in F}$

(f) Add $\{z_i | z_i \in Z_{reset}\}$ to the reset function of previous state, $\langle S, O\rangle$. That is, $reset'(\langle S, O\rangle) := reset'(\langle S, O\rangle) \cup \{z_i | z_i \in Z_{reset}\}$.

(g) Update $\delta'(\langle S, O\rangle, Z') := \delta'(\langle S, O\rangle, Z') \cup \{\langle S', O'\rangle\}$

(h) Update $Q_{new} := Q_{new} \cup \{\langle S', O'\rangle\}$

(i) If for $z_i \in vars$ it holds that for all $\langle S, O\rangle \in Q_{new}$, for all $\langle p, f_p\rangle \in S$ we have $z_i \notin range(f_q)$, then:

i. $vars := vars \setminus \{z_i\}$.

ii. add z_i to $reset'(\langle S', O'\rangle)$.

▷ Here we release variables of Z that are no longer in use, by resetting them. Thus $\mathcal{A}'$ can assign these variables with a new value, and delete them from $vars$ so they can be used in following transitions.

4. Set $Q' := Q_{old}$

To handle cases where $E \neq \emptyset$, mapping a variable x to a new variable is as follows. For every x' with $(x \neq x') \in E$, let $\{z_i\}_{i \in I_{x'}}$ be the set of variables that x' is already mapped to in *previous steps*. Then, we map x to some variable z and add the inequalities $\{z \neq z_i : i \in I_{x'}\}$ to E'.

Figure 3.2 demonstrates a *partial* NVBW $\mathcal{C}$ that is constructed by running AVBWtoNVBW on the AVBW $\mathcal{A}$ shown in Figure 3.1.[6] Recall that $\mathcal{A}$ is an AVBW that expresses the formula $\varphi_{\mathsf{G}\exists} = \mathsf{G}\,\exists x(b.x \wedge \mathsf{F}\,a.x)$. A variable z_i in $\mathcal{C}$ is reset when there is no mapping from the variables of X to z_i. Consider, for example, the transition from state p_2 to state p_3. The edge is labeled with $\{b.z_3, a.z_1\}$. The parameterized proposition $b.z_3$ yields a new mapping from x_1 to z_3, which is released only after $a.z_3$ appears, on the transition from p_5 to p_2. In addition, we can reset the variable z_1 and reuse it in state p_3, since the corresponding transition in the AVBW is from q_1 to $true$ once $a.x_1$ is seen. Thus, $a.z_1$ locally fulfills the requirement for z_1 and the mapping $x_1 \to z_1$ can be forgotten.

[6] Note that AVBWtoNVBW does not halt when given $\mathcal{A}$ as an input.

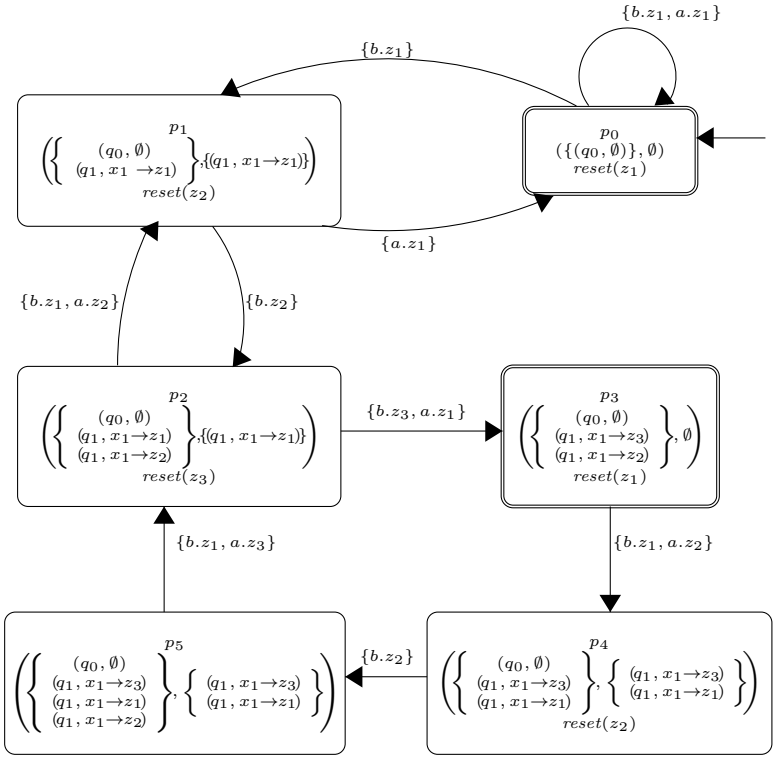


Figure 3.2: The NVBW $\mathcal{C}$

A Structural Characterization of Translatable AVBWs

In order to define a structural characterization of translatable AVBWs, we refer to an AVBW $\mathcal{A}$ as a directed graph $G_{\mathcal{A}}$ whose nodes are the states of $\mathcal{A}$. There is an edge from q to q' iff q' is in $\delta(q, A)$ for some $A \subseteq X$. Edges are labeled with the variables labeling the transition. For example, if $\delta(q, x) = q_1 \vee (q_2 \wedge q_3)$ then there are edges from q to q_1, q_2 and q_3, and each edge is labeled with x.

Definition 3.3.9. *An x-cycle* in an AVBW $\mathcal{A}$ is a cycle in $G_{\mathcal{A}}$ containing an edge labeled x.

Theorem 3.5. *Assume that the preprocessing of stage 1 in the algorithm has been applied, resulting in an AVBW $\mathcal{A}$. Then Procedure AVBWtoNVBW halts on $\mathcal{A}$ and returns an equivalent NVBW iff for every x-cycle $\mathcal{C}_{\mathcal{G}}$ in $G_{\mathcal{A}}$, exactly one of the following holds:*

1. *For every q on $\mathcal{C}_{\mathcal{G}}$ it holds that $x \notin reset(q)$.*
2. *Let q be a state such that q is on a path from the initial state to $\mathcal{C}_{\mathcal{G}}$ with $q_1 \wedge q_2 \in \delta(q, A)$ for some $q_1, q_2 \in Q$ and $x \in A$, such that q_1 is on the cycle $\mathcal{C}_{\mathcal{G}}$ and q_2 leads to an accepting state. Then, every x-cycle $\mathcal{C}'_{\mathcal{G}} \neq \mathcal{C}_{\mathcal{G}}$ on a path from q_2 to an accepting state contains a state q' with $x \in reset(q')$.*[7]

[7] See Figure 3.3 for a graphical demonstration of this condition.

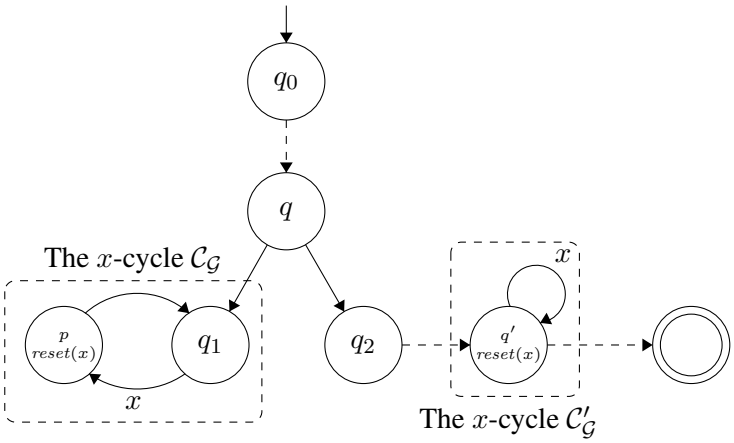


Figure 3.3: A partial graph $G_\mathcal{A}$ for which the second condition of Theorem 3.5 holds. In the transition function of the AVBW $\mathcal{A}$ we have $\delta(q, A) = q_1 \wedge q_2$ for some A. In case $x \notin reset(q')$, the second condition does not hold and our translation algorithm does not halt.

Proof. First, notice that Procedure AVBWtoNVBW halts iff Z is finite, which means that the number of variables it produces is finite.

For the first direction we show that running AVBWtoNVBW on an AVBW $\mathcal{A}$ with the above properties results in an NVBW with a finite set Z. Consider a variable $x \in X$. If 1 holds for every x-cycle, then x is not reset on any cycle in $G_\mathcal{A}$. Thus, there is a bound $k \in \mathbb{N}$ such that on every possible run of $\mathcal{A}$, x is reset most k times, inducing at most k variables in Z. Assume now that 1 does not hold for a variable $x \in X$, and consider an x-cycle $\mathcal{C}_\mathcal{G}$ on which x is reset. From 2 we conclude that x is reset on every other x-cycle $\mathcal{C}'_\mathcal{G}$ that is is reachable in $\mathcal{A}$ while reading the same word. Therefore, we can bound the number of different values assigned to x by the longest simple path between the two x-cycles. Thus, in both cases, x induces finitely many variables in Z.

For the other direction, if 1-2 do not hold, then there exists a state q that leads both to an x-cycle $\mathcal{C}_\mathcal{G}$ on which x is reset, and to an x-cycle $\mathcal{C}'_\mathcal{G}$ with no $reset(x)$, on a path to an accepting state. While running our procedure, a new mapping $x \to z_i$ is introduced after every visit to $q \in Q$ such that $x \in reset(q)$ on $\mathcal{C}_\mathcal{G}$. At the same time, z_i cannot be removed from $vars$, since there always exists a path that visits $\mathcal{C}'_\mathcal{G}$, that does not contain a state that resets x. Therefore, the procedure continuously creates new assignments $x \to z_j$ for $j \neq i$, and so $vars$ does not converge. Recall that in the prepossessing we prune paths that do not lead to an accepting state. Therefore, the fact that there is a path to an accepting state is needed in order for this cycle to "survive" the preprocessing. ■

Completeness and soundness

We now show that no translation algorithm from AVBWs to NVBWs can be both sound and complete, and, that given a general AVBW $\mathcal{A}$, it is possible to decide if the algorithm halts on input $\mathcal{A}$. We first formally define completeness and soundness.

Definition 3.3.10. A procedure $\mathcal{E}$ is *complete* if for every $\mathcal{A}$ that has an equivalent NVBW, $\mathcal{E}(\mathcal{A})$ halts and returns such an equivalent NVBW.

Definition 3.3.11. A procedure $\mathcal{E}$ is *sound* if, whenever $\mathcal{E}(\mathcal{A})$ halts and returns an NVBW $\mathcal{A}'$, it holds that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$.

Theorem 3.6. *There is no algorithm $\mathcal{E}$ that translates AVBWs into NVBWs such that all the following hold.*

1. *$\mathcal{E}$ is complete.*
2. *$\mathcal{E}$ is sound.*
3. *There is a full characterization of AVBWs for which $\mathcal{E}$ halts, i.e., given a general AVBW $\mathcal{A}$, decides if the algorithm halts with input $\mathcal{A}$.*

Proof. We apply a reduction from the emptiness problem for AVBWs, which we have shown in Lemma 3.3.8 to be undecidable. Assume there is a translation algorithm $\mathcal{E}$ as described in Theorem 3.6. Then, consider the following algorithm. Given an AVBW $\mathcal{A}$, if $\mathcal{E}$ halts, check if $\mathcal{E}(\mathcal{A})$ is empty, which implies that $\mathcal{L}(\mathcal{A})$ is empty as well. If $\mathcal{E}$ does not halt on input $\mathcal{A}$, we know it in advance due to the full characterization. Moreover, we know that $\mathcal{L}(\mathcal{A})$ is not empty. Indeed, otherwise, since there is an NVBW for the empty language, and since $\mathcal{E}$ is complete, $\mathcal{E}$ would halt on $\mathcal{A}$. Hence, a translation algorithm as described in Theorem 3.6 induces a procedure that decides the emptiness problem for AVBWs, a contradiction. ■

For our procedure, we have shown a full characterization for halting. We now prove that our algorithm is sound, and demonstrate its incompleteness by an example of an AVBW for the empty language, for which our procedure does not halt.

Theorem 3.7. *Procedure AVBWtoNVBW is sound.*

Proof. We first show that the definition of E' is correct. Let $x, x' \in X$ be variables in the input AVBW $\mathcal{A}$, and let $\{z_{x_i}\}_{x \in X, i \in \mathbb{N}}$ be the set of variables in the constructed NVBW $\mathcal{A}'$, where a variable z_{x_i} in $\mathcal{A}'$ is induced by the variable x in $\mathcal{A}$. Let E and E' be the sets of inequalities in $\mathcal{A}$ and $\mathcal{A}'$, respectively, and $reset$, $reset'$ be the reset functions of $\mathcal{A}$ and $\mathcal{A}'$. Every $(z_{x_i} \neq z_{x'_j}) \in E'$ is derived from $(x \neq x') \in E$, and each z_{x_i} is induced from only one variable $x \in X$. Therefore, E' preserves exactly the inequalities of E . Now, $reset'$ is defined according to $reset$ such that if z_i is induced from x, and x is reset in a state q, then z_i is reset in states that include q. Therefore, $reset'$ allows fresh values only when $reset$ does. The correctness of the rest of the construction follows from the correctness of [MH84] and from the explanations in the body of the algorithm. ■

Example 3.3.12. Incompleteness of the procedure. Consider the AVBW $\mathcal{A}_\emptyset$ of Figure 3.4. Formally, let $\mathcal{A}_\emptyset = \langle \mathcal{B}, \Gamma, \emptyset \rangle$ where $\mathcal{B} = \langle 2^{\{a.x, b.x\}}, \{q_0, q_1\}, q_0, \delta, reset, \{q_0\} \rangle$ and

- $reset(q_0) = \{x\}, reset(q_1) = \emptyset$

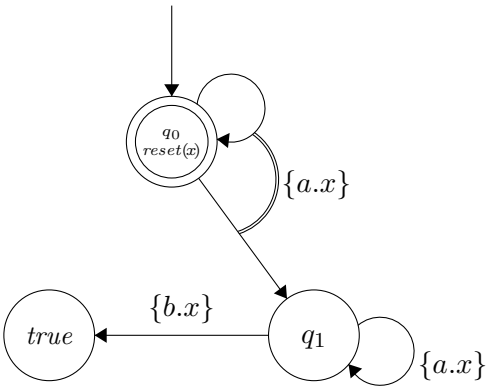


Figure 3.4: The AVBW $\mathcal{A}_\emptyset$

- $\delta(q_0, \{a.x\}) = q_0 \wedge q_1$
  $\delta(q_1, \{a.x\}) = q_1$
  $\delta(q_1, \{b.x\}) = true$

The language of $\mathcal{A}_\emptyset$ is empty, since in order to reach an accepting state on the path from q_1, the input must be exactly $\{b.i\}$ for some $i \in \Gamma$, but the cycle of q_0 can only read $\{a.j\}$, without any $b.i$. Although there is an NVBW for the empty language, our procedure does not halt on $\mathcal{A}_\emptyset$: it keeps allocating new variables to x, thus new states are created and the procedure does not reach a fixed point.

3.4 Bounded Model Checking for Systems over Infinite Data

Bounded model checking [BCCZ99] searches for a counterexample in a bounded part of the system. This approach comes in handy when the search-space is very large. Bounded model checking is usually applied iteratively, where in every iteration the bound is increased. In [BCCZ99, CBRZ01] the authors use SAT-based procedures in order to produce a minimal counterexample. While iterating over k, a CNF formula that describes a computation of length k that ends in a bad state, is checked. If the formula is unsatisfiable, then the system is safe up to k steps. If the formula is satisfiable, then a satisfying assignment produces a counterexample of length k.

In this section we employ the bounded approach in order to search for a witness to the nonemptiness of an AVBW. We rely on the translation procedure AVBWtoNVBW of Section 3.3.5, which translates an AVBW $\mathcal{A}$ to an NVBW $\mathcal{A}'$ by iteratively adding states and variables to $\mathcal{A}'$. We exploit the natural iterative behavior of AVBWtoNVBW in order to seek, after every iteration, a witness to its nonemptiness in the partial construction $\mathcal{C}$ of $\mathcal{A}'$ that has been calculated until that point. The language of $\mathcal{C}$ is a subset of that of $\mathcal{A}'$, which ensures the correctness of the procedure. Notice that here, we bound the number of iterations, and not the length of the witness. This method is particularly appealing in our setting, as AVBWtoNVBW may not converge, yet $\mathcal{A}'$ may include an accepting lasso after finitely many steps.

Since deciding nonemptiness lies at the heart of automata-based model-checking, we can use this general scheme in order to describe a bounded model-checking algorithm for $\exists^*$-VLTL formulas and NVBWs.

3.4.1 A Bounded Model-Checking algorithm for $\exists^*$-VLTL Formulas

Let ψ be an $\exists^*$-VLTL formula that describes a bad behavior. It can be viewed as the negation of an $\forall^*$-VLTL formula, which describes a desired, good behavior of the system. Let $\mathcal{A}_\psi$ be an AVBW that expresses ψ, and let M be a system modeled by an NVBW $\mathcal{A}_M$. Model checking M against ψ amounts to checking the nonemptiness of $\mathcal{A}_\psi \cap \mathcal{A}_M$. If we can construct an NVBW $\mathcal{A}'_\psi$ that is equivalent to $\mathcal{A}_\psi$, we can directly construct an NVBW $\mathcal{A}'_\psi \cap \mathcal{A}_M$ and test it for emptiness. If $\mathcal{A}'_\psi$ does not exist, then bounded model checking is the only hope. Moreover, since $\mathcal{A}'_\psi$, even if exists, is exponential in the size of the formula, a bounded model-checking approach may still be preferable.

Algorithm: $\exists^*$VLTL-BMC

As we have described, we use the partial output of AVBWtoNVBW and test it for emptiness.

1. Create new states according to algorithm AVBWtoNVBW (steps 1-4), where we follow the construction breadth-wise, i.e., instead of following one path until no new states are created, follow all possible successors of the current set of states. This way we scan all paths simultaneously up to a bounded distance.

2. If AVBWtoNVBW closes a cycle containing an existing accepting state, create a candidate NVBW $\mathcal{C}$ from the current set of states and transitions.

3. Construct $\mathcal{C} \cap \mathcal{A}_M$, and test it for emptiness. The emptiness test amounts, as in the case of finite alphabets, to finding an accepting lasso, which can be done on-the-fly with some additional considerations that match the variables of $\mathcal{C}$ and $\mathcal{A}_M$, in a similar fashion to [GKS12]. If the intersection is empty, go to 1 and continue running AVBWtoNVBW. Otherwise, the emptiness test returns a word $\tilde{w} \in (AP \times Z)^\omega$, which is a symbolic counterexample. Assigning values to the variables in Z then produces a concrete counterexample. Since w is lasso-shaped, a concrete calculation can be finitely produced.

As we have mentioned, phases 1-2 comply well with AVBWtoNVBW, due to its iterative behavior, which produces a sub-NVBW of the final result in every step, even if the run never converges.

Example 3.4.1. Let $\varphi_{\mathsf{G}\exists} = \mathsf{G}\,\exists x(b.x \wedge \mathsf{F}\,a.x)$ be the formula discussed in Section 3.3.1, and let $\mathcal{A}$ in Figure 3.1 be an AVBW that expresses $\varphi_{\mathsf{G}\exists}$. Recall that $\varphi_{\mathsf{G}\exists}$ cannot be expressed by an NVBW. The NVBW $\mathcal{C}$ in Figure 3.2 is a partial result constructed by running $\exists^*$VLTL-BMC on $\mathcal{A}$.

Consider the following symbolic word.

$$\tilde{w} = \{b.z_1\}\{b.z_2\}\{b.z_3, a.z_1\}(\{b.z_1, a.z_2\}\{b.z_2\}\{b.z_1, a.z_3\}\{b.z_3, a.z_1\})^\omega$$

$\tilde{w}$ is a symbolic witness to the nonemptiness of $\mathcal{A}$. We can construct a concrete witness over $\mathbb{N}$ as follows.

$$w = \{b.1\}\{b.1\}(\{b.1, a.1\}\{b.1\}\{b.1, a.1\}\{b.1, a.1\})^\omega$$

w is not a very interesting witness for the satisfiability of $\varphi_{\mathsf{G}\,\exists}$, yet it suffices in order to know that $\mathcal{A}$ is nonempty. We can look for more interesting concretizations, such as

$$\{b.1\}\{b.2\}\{b.3, a.1\}(\{b.1, a.2\}\{b.2\}\{b.1, a.3\}\{b.3, a.1\})^\omega$$

Note that the distance between an occurrence of $b.\gamma$ and $a.\gamma$ for $\gamma \in \Gamma$ is bounded by the length of a cycle from state p_3 to itself. Therefore, producing larger automata via $\exists^*$VLTL-BMC may allow producing more interesting concretizations.

3.4.2 Absence of Cycles does not Guarantee Emptiness

In Section 3.4.1 we argued that an accepting cycle in the NVBW $\mathcal{C}$ that is partially constructed by AVBWtoNVBW induces a witness to the nonemptiness of the original AVBW $\mathcal{A}$. We now demonstrate that the absence of cycles in every such partial NVBW $\mathcal{C}$ does not guarantee the emptiness of $\mathcal{A}$.[8] Intuitively, this phenomenon occurs when the language of $\mathcal{A}$ requires infinitely many values to appear in every word in the language, a requirement that cannot be fulfilled by a finite cycle, even if it contains resets.

Example 3.4.2. Consider the AVBW $\mathcal{A}_1$ shown in Figure 3.5 and some unwinding of it $\mathcal{C}_1$ shown in Figure 3.6. The transition function of $\mathcal{A}_1$ is defined as follows:

$$\begin{gathered}\delta(q_0, \{a.x\}) = q_0 \wedge q_1 \wedge q_2 \\ \delta(q_1, \{a.y\}) = q_1 \\ \delta(q_2, \{a.y\}) = q_3 \\ \delta(q_3, \{a.y\}) = \delta(q_3, \{a.x\}) = q_2\end{gathered}$$

The *reset* labeling function is:

$$\begin{aligned}reset(q_0) = reset(q_3) &= \{x\} \\ reset(q_1) = reset(q_2) &= \{y\}\end{aligned}$$

The inequality set is $E = \{x \neq y\}$. The inequality set of $\mathcal{C}_1$ is then $E' = \{z_i \neq z_j : \ i < j\}$.

Since the inequality set of $\mathcal{A}_1$ is $E = \{x \neq y\}$, and the state q_1 does not reset the variable x, a value that appears once and is assigned to x may not appear again. Note that x and y may

[8] Note that absence of cycles means, in particular, that the algorithm AVBWtoNVBW does not halt, as it keeps creating new states.

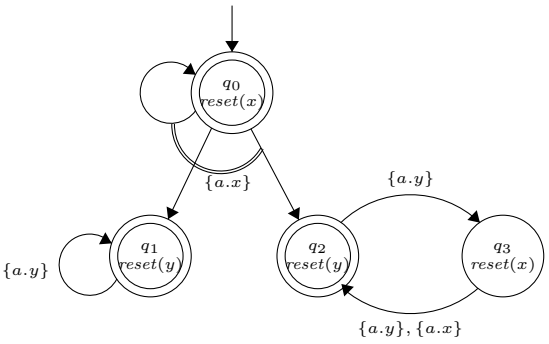


Figure 3.5: The AVBW $\mathcal{A}_1$

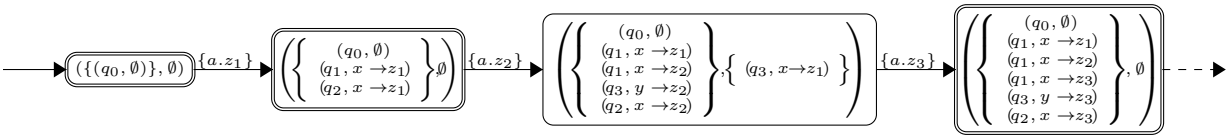


Figure 3.6: $\mathcal{C}_1$, a partial NVBW for the translation of $\mathcal{A}_1$

only be assigned the same value along different paths, since the restriction of inequalities holds for assignments on the same path.

Every state in $\mathcal{C}_1$ contains a new mapping for x in q_1. This is due to the fact that q_1 has an incoming transition labeled $a.x$, and q_1 does not reset x. The transitions of $\mathcal{C}_1$ are labeled at each iteration with a new variable z_i. All previous variables $\{z_j\}_{j<i}$ are still kept (that is, are not reused), in order to make sure that the inequality set $E' = \{z_i \neq z_j : \ i < j\}$ is satisfied.

Since new variables are introduced in every step, no state in $\mathcal{C}_1$ – in particular, no accepting state – is seen twice by AVBWtoNVBW.

For an ω-regular language $\mathcal{L}$ over a finite alphabet, there exists a finite set (of size k for some natural number k) of regular languages $\mathcal{L}_i^1$ and $\mathcal{L}_i^2$, such that $\mathcal{L} = \cup_{i=0}^{k} \mathcal{L}_i^1 \cdot (\mathcal{L}_i^2)^\omega$ [Saf88]. The example above shows that this characterization does not hold for the class of languages accepted by AVBWs. Note that the language of $\mathcal{A}_1$ is not empty. For example, it contains all words of the from $a.i\ a.(i+1)\ a.(i+2) \cdots$, yet there is no finite candidate that indicates nonemptiness. This example shows that a witness word in $\mathcal{L}(\mathcal{C})$ indicates a witness word in $\mathcal{L}(\mathcal{A})$; However, the emptiness of $\mathcal{L}(\mathcal{C})$ does not indicate the emptiness of $\mathcal{L}(\mathcal{A})$.

3.5 Decidable fragments of $\exists^*$-VLTL

As we have mentioned in Section 3.3, the satisfiability problem of $\exists^*$-VLTL formulas is in general undecidable, and $\exists^*$-VLTL formulas cannot in general be expressed by NVBWs. In this section we review fragments of $\exists^*$-VLTL that can be expressed by NVBWs, and a fragment of $\exists^*$-VLTL for which the satisfiability problem is decidable, even though it cannot be expressed by NVBWs.

3.5.1 Fragments of $\exists^*$-VLTL that are Expressible by NVBW

We present several fragments of $\exists^*$-VLTL that can be directly translated to NVBWs. As a result, the satisfiability problem for these fragments is decidable.

$\exists^*_{PNF}$-VLTL

We first consider $\exists^*$-VLTL formulas in prenex normal form (PNF), denoted $\exists^*_{PNF}$-VLTL. An $\exists^*_{PNF}$-VLTL formula is of the form $\varphi = \exists x_1 \exists x_2 \ldots \exists x_k \psi$, where ψ is quantifier-free. Notice that an AVBW $\mathcal{A}_\varphi$ that expresses φ contains no resets, and is similar to an ABW for φ over the symbolic alphabet. Since resets are at the heart of the complexities of the AVBW to NVBW construction, and $\mathcal{A}_\varphi$ is free of them, an NVBW for φ is computable in a similar way to the standard ABW to NBW construction.[9] Moreover, we can use the standard tableau construction for LTL [BCM$^+$92], and so we have a direct construction from $\exists^*_{PNF}$-VLTL to NVBW.

Example 3.5.1. An interesting property expressible in $\exists^*_{PNF}$-VLTL is given by the formula $\exists x(\mathsf{G}\,\mathsf{F}\, send.x \wedge \mathsf{F}\,\mathsf{G}\,\neg recieve.x)$. If x ranges over the messages content, this formula states that there is some message that is sent infinitely often, but is not received starting some point in the computation. This is the semantical negation of the property "every message that is sent will be received in the future". That is, $\exists^*_{PNF}$-VLTL allows us to express interesting "bad" properties, which we can test for satisfiability using our algorithms.

Fully Nested $\exists^*$-VLTL

We say that an $\exists^*$-VLTL formula φ is *fully nested* if every quantifier in φ is either at the head of φ, or adjacent to a parameterized atomic proposition. That is, all of the quantifiers in a fully nested formula are either at the very "outside" of the formula, or at the very "inside" of it. We denote this fragment $\exists^*_{fn}$-VLTL. Notice that $\exists^*_{fn}$-VLTL subsumes $\exists^*_{PNF}$-VLTL.

We show now that $\exists^*_{fn}$-VLTL is translatable to NVBW. Consider an $\exists^*_{fn}$-VLTL formula φ and let $\mathcal{A}_\varphi$ be an AVBW expresses it. We notice that transitions from states that represent atomic propositions in $\mathcal{A}_\varphi$ are either $true$ or $false$, and therefore do not introduce new variables. Hence, despite the fact that quantifiers inside the formula induce resets in these states, these resets do not require adding new variables in the translation of $\mathcal{A}_\varphi$ to an NVBW. Therefore, in this case, our translation algorithm to NVBW always terminates.

As in Section 3.5.1, here too we can use the tableau construction and directly construct an NVBW for the formula φ, by adding resets on states in the tableau that represent atomic propositions.

Example 3.5.2. Consider the formula $\varphi = \mathsf{G}\,\mathsf{F}\,\exists x(fail.x)$, where x ranges over process IDs. We can use φ in order to verify that from some point on, all processes work correctly, since φ is the negation of this property.

[9] In [SW] the authors conjecture without proof that the formula $\mathsf{G}\,\exists x : a.x$ does not have an equivalent in *PNF*. In Lemma 3.3.3 we show that $\mathsf{G}\,\exists x(b.x \wedge \mathsf{F}\,a.x)$ does not have an equivalent NVBW, and therefore does not have an equivalent $\exists^*_{PNF}$-VLTL formula. This is a different formula from $\mathsf{G}\,\exists x a.x$, but the conclusion remains the same.

$\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL

Another fragment that is easily translatable to NVBW is $\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL, which is the fragment of $\exists^*$-VLTL that only uses the temporal operators X and F. A similar fragment for LTL is defined in [EH86].

Notice that the $\exists$ quantifier and the X, F operators are interchangeable. Intuitively, it does not matter whether a variable is reset immediately after committing to its future appearance or before. Therefore, every $\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL has an equivalent $\exists^*_{PNF}$-VLTL formula over X, F only, that can be calculated by renaming variables under the scope of the quantifier, and pushing the quantifiers out. As a conclusion, we have that $\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL can be expressed by NVBW.

This fragment expresses very intuitive, yet important properties, as demonstrated in the following example.

Example 3.5.3. Consider the formula $\exists x \mathsf{F}\, fail.x$, where x ranges over critical process IDs. Clearly, it is enough for one such process to fail in order for the entire system to be unsafe. Thus, The fragment of $\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL allows us to easily model-check systems for such bugs.

Remark. The interchangeability between quantifiers and the X, F operators is not applicable for formulas that include the $\forall$ quantifier. For example, consider the VLTL formulas $\varphi = \mathsf{F}\,\forall x(\neg a.x)$ and $\varphi' = \forall x \mathsf{F}\,\neg a.x$. The word $w = (\{a.1\}\{a.2\})^\omega$ satisfies φ', since for every $x \neq 1$, the formula $\neg a.x$ holds in the first step, and $\neg a.x$ holds at the second step for $x = 1$. However, w does not satisfy φ since there is no step along the computation where a does not hold with some value.

To conclude the discussion so far, we have the following.

Theorem 3.8.

1. *In terms of expressive power, $\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL $\leq \exists^*_{PNF}$-VLTL $\leq \exists^*_{fn}$-VLTL $< \exists^*$-VLTL.*
2. *$\exists^*_{(\mathsf{X},\mathsf{F})}$-VLTL, $\exists^*_{PNF}$-VLTL, and $\exists^*_{fn}$-VLTL can all be expressed by NVBWs.*

VGR(1)

The logic of GR(1), first presented by [PPS06], is widely used in software verification, particularly in synthesis. GR(1) formulas are of the form $\bigwedge_i B_i \rightarrow \bigwedge_i C_i$ where B_i and C_i are of the form $\mathsf{G}\,\mathsf{F}\, p_i$ for $p_i \in AP$. We consider possible extensions of GR(1) to variable GR(1), or VGR(1), in $\exists^*$-VLTL.

According to Theorem 3.8, fully-nested $\exists^*$-VLTL formula can be expressed by NVBWs. Therefore, VGR(1) formulas of the form $\exists x_1 \exists x_2 \ldots \exists x_k (\bigwedge_i B_i \rightarrow \bigwedge_i C_i)$ where B_i and C_i are of the form $a_i.x_i$ or $\exists x_i (a_i.x_i)$ can be expressed by NVBWs.

We now consider a more complex structure of VGR(1) formulas. Consider formulas of the form $\varphi = \bigwedge_i B_i \rightarrow \bigwedge_i C_i$ where C_i is in one of the forms

$$\exists x_i \mathsf{G}\,\mathsf{F}\, a_i.x_i, \quad \mathsf{G}\,\exists x_j \mathsf{F}\, a_i.x_j, \quad \mathsf{G}\,\mathsf{F}\,\exists x_j (a_i.x_j)$$

and B_i is of the latter two forms.[10]

The formula φ is equivalent to $(\bigvee_i \neg B_i) \vee (\bigwedge_i C_i)$. Thus, constructing an NVBW $\mathcal{A}_\varphi$ for φ amounts to constructing an NVBW $\mathcal{A}_{\neg B_i}$ for every $\neg B_i$, and an NVBW $\mathcal{A}_{C_i}$ for every C_i.

The negation of the formulas B_i is of the form $\mathsf{F}\,\mathsf{G}\,\forall x \neg a.x$. This formula states that from some point of the computation, a does not appear at all. Although this is a $\forall$-VLTL formula, it is easy to construct an NVBW expresses it. Note that the negation of the formula $\psi = \exists x_i \mathsf{G}\,\mathsf{F}\,a_i.x_i$, which is $\forall x_i \mathsf{F}\,\mathsf{G}\,\neg a_i.x_i$ has no equivalent NVBW. Intuitively, this is since it requires keeping track on all domain elements, and for each of them, to make sure that starting from some point in the computation, this values does not appear. However, the position of this point in the computation is unbounded and may be different for different domain elements. Therefore, we do not take B_i to be of the form of ψ.

As for C_i, the formula $\exists x_i \mathsf{G}\,\mathsf{F}\,a_i.x_i$ is an $\exists^*_{PNF}$-VLTL formula and thus has an equivalent NVBW; and the formulas $\mathsf{G}\,\exists x_i \mathsf{F}\,a_i.x_i$ and $\mathsf{G}\,\mathsf{F}\,\exists x_i a_i.x_i$ are equivalent, and can be represented as NVBWs since $\mathsf{G}\,\mathsf{F}\,\exists x_i a_i.x_i$ is a fully nested $\exists^*$-VLTL formula.

Then, $\mathcal{A}_\varphi = (\bigcup_i \mathcal{A}_{\neg B_i}) \cap (\bigcap_i \mathcal{A}_{C_i})$. Since NVBWs are closed under union and intersection [GKS10], $\mathcal{A}_\varphi$ is an NVBW.

3.5.2 Further decidable fragments

We now present a fragment of $\exists^*$-VLTL that cannot in general be expressed by NVBWs, yet its satisfiability is decidable. Consider an $\exists^*$-VLTL formula φ. We assume that every $\exists$ quantifier in φ bounds a different variable.[11] We define the *flattening* of φ, denoted by φ^{fl}, as the $\exists^*$-VLTL formula obtained from φ by removing all the $\exists$ quantifiers, and placing them at the beginning of the formula. For example, the flattening of $\varphi = \mathsf{G}\,\exists x_1 b.x_1 \wedge \exists x_2 \mathsf{F}\,a.x_1 \wedge b.x_2$ is $\varphi^{fl} = \exists x_1 \exists x_2 \mathsf{G}\,b.x_1 \wedge \mathsf{F}\,a.x_1 \wedge b.x_2$. Notice that $\varphi \not\equiv \varphi^{fl}$, and that φ^{fl} is an $\exists^*_{PNF}$-VLTL formula.

As we now show, the satisfiability of φ^{fl} may point to the satisfiability of φ, and in case that φ is negation free, the two are equisatisfiable.

Lemma 3.5.4. *Let φ be an $\exists$-VLTL formula. Then, the following holds.*

1. *$\mathcal{L}(\varphi^{fl}) \subseteq \mathcal{L}(\varphi)$, and therefore if φ^{fl} is satisfiable then φ is satisfiable.*
2. *If φ does not contain negations, then φ is satisfiable iff φ^{fl} is satisfiable.*

Moreover, every witness for the satisfiability of φ^{fl} is a witness for φ as well.

Proof. Let $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi^{fl}}$ be AVBWs that express φ and φ^{fl}, respectively. Since the difference between φ and φ^{fl} is only in the location of the $\exists$ quantifiers, the graph structures of $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi^{fl}}$ are identical, and the only difference between them is in the location of the resets. Indeed, while resets may occur anywhere in $\mathcal{A}_\varphi$, the resets in $\mathcal{A}_{\varphi^{fl}}$ occur only in its initial state (recall the construction of an AVBW from an $\exists^*$-VLTL formula, as described the proof of Theorem 3.2.)

[10] As we show in 3.5.1, these latter two formulas are equivalent.

[11] Every $\exists^*$-VLTL has an equivalent in this form.

Accordingly, in an accepting run-tree T_w of $\mathcal{A}_{\varphi^{fl}}$ on a word w, each variable is assigned a fixed value throughout the run. Notice that due to the similar structure of both AVBWs, the run-tree T_w is also an accepting run-tree of $\mathcal{A}_\varphi$ on w, in which each variable is pre-assigned the values it is assigned in the run of $\mathcal{A}_{\varphi^{fl}}$ on w, and the resets do not change the values of the variables.

Therefore, together with the previous paragraph, we have that $\mathcal{L}(\mathcal{A}_{\varphi^{fl}}) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$. This concludes the proof of 1.

For 2, the first direction is a special case of 1. For the second direction, let φ be a satisfiable $\exists^*$-VLTL formula with no negations, and let w be a computation that is accepted by $\mathcal{A}_\varphi$ with a run-tree T_w. Let w' be the concrete computation that is obtained from w by replacing every value in w with the same value γ. Notice that since φ and φ^{fl} are both negation-free, then both AVBWs have an empty set of inequalities, and so there is no restriction on the values that may be assigned to the variables throughout the run. Also notice that since the values of all the variables remain fixed, w' is accepted by $\mathcal{A}_{\varphi^{fl}}$, and therefore satisfies φ^{fl}. We claim that w' also satisfies φ. Indeed, since there are no inequalities in both AVBWs, then at every reset to a variable x along T_w, the variable x may be assigned γ. Therefore, we can use T_w to produce an accepting run-tree of $\mathcal{A}_\varphi$ on w'. ■

Notice that although φ and φ^{fl} are equisatisfiable in case that φ is negation free, they are not necessarily equivalent. Moreover, φ may not even be expressible by an NVBW. We now present some examples.

Example 3.5.5.

1. Consider the formula $\psi = \mathsf{G}\,\exists x(a.x \wedge \mathsf{X}\,\mathsf{G}\,\neg a.x)$. The formula ψ can be satisfied only by a computation in which a is paired with a different value at every step. The formula ψ^{fl} is $\exists_x(\mathsf{G}\,a.x \wedge \mathsf{X}\,\mathsf{G}\,\neg a.x)$. The formula ψ^{fl} is not satisfiable, even though ψ is satisfiable.[12]

2. Consider the formula $\varphi_{\mathsf{G}\,\exists} = \mathsf{G}\,\exists x(b.x \wedge \mathsf{F}\,a.x)$. We have shown in Section 3.3 that $\varphi_{\mathsf{G}\,\exists}$ cannot be expressed by an NVBW. However, the formula $\varphi^{fl}_{\mathsf{G}\,\exists} = \exists x(\mathsf{G}\,b.x \wedge \mathsf{F}\,a.x)$ is satisfiable, and a witness to its satisfiability, such as $w = (\{a.1, b.1\})^\omega$, can easily be found with the $\exists^*$VLTL-BMC algorithm from Section 3.4.1. As we have shown, w is also a witness to the satisfiability of $\varphi_{\mathsf{G}\,\exists}$.

Note that both ψ and $\varphi_{\mathsf{G}\,\exists}$ do not have an equivalent NVBW. However, while we can find a witness to the satisfiability of $\varphi_{\mathsf{G}\,\exists}$ with $\exists^*$VLTL-BMC using a very small bound, this algorithm cannot find a witness to the satisfiability of ψ, with any bound.

Following the observations that we have discussed here, we can use the flattening of $\exists^*$-VLTL formulas for an incomplete model-checking procedure: Given an NVBW $\mathcal{A}_M$ and an $\exists^*$-VLTL formula φ, check the emptiness of $\mathcal{A}_{\varphi^{fl}} \cap \mathcal{A}_M$. Since φ^{fl} is in PNF, emptiness is decidable. If there exists a word in the intersection, it is a witness to the violation of the specification given to us as $\neg\varphi$. However, if we cannot find a word in the intersection, this does not imply that the program satisfies the specification.

[12] The set of computations satisfy ψ is exactly the language of the AVBW $\mathcal{A}_1$ from Figure 3.5.

3.6 Concluding Remarks

In this chapter we consider the verification of ongoing systems over infinite data domains with respect to VLTL specifications.

We have defined AVBWs, a new model of automata over infinite alphabets, which combines alternation with variable automata over infinite words. As we have demonstrated, AVBWs manage to express VLTL formulas that previous models were unable to express, namely all $\exists^*$-VLTL formulas. We showed that AVBWs are strictly stronger than NVBWs. Nevertheless, we presented a procedure for translating AVBWs to NVBWs when possible. Moreover, we defined a structural characterization of AVBWs that are translatable by our procedure.

$\exists^*$-VLTL formulas can, in many cases, naturally describe "bad" behaviors, and hence, come up naturally in the context of model-checking. Thus, AVBWs become an essential tool in model-checking $\exists^*$-VLTL formulas, as every such formula can be expressed by an AVBW. An example for such an important property is the *response property*, $\varphi_\forall = \forall x \mathsf{G}\,(a.x \rightarrow \mathsf{F}\, b.x)$. The negation of $\varphi_\forall$ is $\exists x \mathsf{F}\,(a.x \wedge \mathsf{G}\, \neg b.x)$, for which we present an easy model-checking algorithm.

When an AVBW can be translated to an NVBW by our procedure, the result can be used in a model-checking procedure that calculates the intersection of the NVBW with the program automaton, and checks the nonemptiness of the intersection. However, even for formulas that are not translatable, we can still use our procedure for a bounded model-checking algorithm. Moreover, we presented fragments of $\exists^*$-VLTL for which there is a direct construction of NVBW, and a fragment ($\exists^*$-VLTL with no negations) whose satisfiability is decidable even though it is not always translatable to NVBW. Thus, we have expanded not only the expressive fragment of VLTL, but also the fragments that can be model-checked.

To conclude, in order to preform model-checking for an $\exists^*$-VLTL formula φ, we can do one of the three: translate φ to an NVBW if it is one of the types of Section 3.5; build an AVBW $\mathcal{A}_\varphi$, and if the structure of $\mathcal{A}\varphi$ agrees with the structural characterization of Theorem 3.5, translate it to an equivalent NVBW according to Section 3.3.5; or use the bounded model-checking algorithm presented in Section 3.4 and look for a partial NVBW that enables searching for a witness for nonemptiness.

Our work presents an incomplete model-checking algorithm for $\exists^*$-VLTL formulas, thus laying the theoretical foundations for a model-checking tool for $\exists^*$-VLTL, which we plan to implement as future work. As a further direction, we plan to use the techniques we have presented here in order to construct suitable algorithms for model checking VCTL [GKS14] formulas as well.

Chapter 4

Compositional Verification and Repair

4.1 Communicating Programs

In this chapter we present the notion of *communicating programs*. These are C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as automata over an *action alphabet* that reflects the program statements. The alphabet includes *constraints*, which are quantifier-free first-order formulas, representing the conditions in *if* and *while* statements. It also includes *assignment statements* and $read$ and $write$ *communication actions*. The automata representation is similar in nature to that of control-flow graph. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as $\mathbf{L}^*$ for its verification [Ang87b].

Given two communicating programs, M_1 and M_2, we wish to prove that the composed system $M_1||M_2$, that is the result of the communication of the two components, is correct. However, the composed system might be too large for the verification to scale well. To address this problem, we turn to compositional verification. We use the Assume-Guarantee rule [MC81, Pnu85] and the $\mathbf{L}^*$ algorithm in order to compositionally prove the correctness of the system. In case an error is found, we repair the system and return to try and verify the repaired system.

We first formally define the alphabet over which communicating programs are defined. Let G be a finite set of communication channels. Let X be a finite set of variables (whose ordered vector is $\bar{x}$) and $\mathbb{D}$ be a (possibly infinite) data domain. For simplicity, we assume that all variables are defined over $\mathbb{D}$. The elements of $\mathbb{D}$ are also used as constants in arithmetic expressions and constraints.

Definition 4.1.1. An *action alphabet* is $\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ where:

1. $\mathcal{G} \subseteq \{\ g?x_1, g!x_1, (g?x_1, g!x_2), (g!x_1, g?x_2)\ :\ g \in G, x_1, x_2 \in X\}$ is a finite set of *communication actions*.

   - $g?x$ is a *read* action of a value to the variable x through channel g.
   - $g!x$ is a *write* action of the value of x on channel g. We use $g * x$ to indicate some action, either *read* or *write*, through g.

- The pairs $(g?x_1, g!x_2)$ and $(g?x_1, g!x_2)$ represent a synchronization of two programs on read-write actions over channel g (defined later).

2. $\mathcal{E} \subseteq \{\ x := e\ :\ e \in E, x \in X\}$ is a finite set of *assignment statements*, where E is a set of expressions over $X \cup \mathbb{D}$.

3. $\mathcal{C}$ is a finite set of constraints over $X \cup \mathbb{D}$.

Definition 4.1.2. A *communicating program* (or, a program) is $M = \langle Q, X, \alpha, \delta, q_0, F \rangle$, where:

1. Q is a finite set of states and $q_0 \in Q$ is the initial state.

2. X is a finite set of variables that range over $\mathbb{D}$.

3. $\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ is the action alphabet of M.

4. $\delta \subseteq Q \times \alpha \times Q$ is the transition relation.

5. $F \subseteq Q$ is the set of accepting states.

The words that are read along a communicating program are a *symbolic representation* of the program behaviors. We refer to such a word as a *trace*. Each such trace induces *concrete runs* of the program, which are formed by concrete assignments to the program variables in a way that conforms with the actions along the word.

Although communicating programs are an extension of finite automata, we investigate them from a different perspective. While in Chapter 3 the automaton takes as input a computation and checks whether the computation satisfies the specification by reading the computation against the specification automaton, here we like to think of the automaton as the generator of the behavior, as it describes the program. Therefore, we begin with a run of the program, and induce traces from the run, and not the other way around. We now formally define these notions.

Definition 4.1.3. As defined in Chapter 2.1, a run in a program automaton M is a finite sequence of states and actions $r = \langle q_0, a_1, q_1 \rangle \ldots \langle q_{n-1}, a_n, q_n \rangle$, starting with the initial state q_0, such that $\forall 0 \leq i < n$ we have $\langle q_i, a_{i+1}, q_{i+1} \rangle \in \delta$. The *induced trace* of r is the sequence $t = (a_1, \ldots, a_n)$ of the actions in r. If q_n is accepting, then t is an *accepted trace* of M.

From now on we assume that every trace we discuss is induced by some run. We turn to define the concrete executions of the program.

Definition 4.1.4. Let $t = (a_1, \ldots, a_n)$ be a trace and let $(\beta_0, \ldots, \beta_n)$ be a sequence of valuations (i.e., assignments to the program variables).[1] Then a sequence $e = (\beta_0, a_1, \beta_1, a_2, \ldots, a_n, \beta_n)$ is an *execution* of t if the following holds.

1. β_0 is an arbitrary valuation.

[1] Such valuations are usually referred to as states. We do not use this terminology here in order not to confuse them with the states of the automaton.

2. If $a_i = g?x$, then $\beta_i(y) = \beta_{i-1}(y)$ for every $y \neq x$. Intuitively, x is arbitrarily assigned by the read action, and the rest of the variables are unchanged.

3. If a_i is an assignment $x := e$, then $\beta_i(x) = e[\bar{x} \leftarrow \beta_{i-1}(\bar{x})]$ and $\beta_i(y) = \beta_{i-1}(y)$ for every $y \neq x$.

4. If $a_i = (g?x, g!y)$ then $\beta_i(x) = \beta_{i-1}(y)$ and $\beta_i(z) = \beta_{i-1}(z)$ for every $z \neq x$. That is, the effect of a synchronous communication on a channel is that of an assignment.

5. If a_i does not involve a read or an assignment, then $\beta_i = \beta_{i-1}$.

6. Finally, if a_i is a constraint in $\mathcal{C}$, then $\beta_i(\bar{x}) \vDash a_i$ (and since a_i does not change the variable assignments, then $\beta_{i-1}(\bar{x}) \vDash a_i$ holds as well).

We say that t is *feasible* if there exists an execution of t.

The *symbolic language* of M, denoted $\mathcal{T}(M)$, is the set of all *accepted* traces induced by runs of M. The *concrete language* of M is the set of all executions of accepted traces in $\mathcal{T}(M)$. We will mostly be interested in feasible traces, which represent (concrete) executions of the program.

Example 4.1.5.

- The trace $(x := 2 \cdot y,\ g?x,\ y := y + 1,\ g!y)$ is feasible, as it has an execution $(x = 1, y = 3), (x = 6, y = 3), (x = 20, y = 3), (x = 20, y = 4), (x = 20, y = 4)$.
- The trace $(g?x,\ x := x^2, x < 0)$ is not feasible since no β can satisfy the constraint $x < 0$ if $x := x^2$ is executed beforehand.

4.1.1 Parallel Composition

We now describe and define the parallel composition of two communicating programs, and the way in which they communicate.

Let M_1 and M_2 be two programs, where $M_i = \langle Q_i, X_i, \alpha_i, \delta_i, {q_0}^i, F_i \rangle$ for $i = 1, 2$. Let G_1, G_2 be the sets of communication channels occurring in actions of M_1, M_2, respectively. We assume that $X_1 \cap X_2 = \emptyset$.

The *interface alphabet* αI of M_1 and M_2 consists of all communication actions on channels that are common to both components. That is, $\alpha I = \{\, g?x,\ g!x \ :\ g \in G_1 \cap G_2,\ x \in X_1 \cup X_2 \}$.

In *parallel composition*, the two components synchronize on their communication interface only when one component writes data through a channel, and the other reads it through the same channel. The two components cannot synchronize if both are trying to read or both are trying to write. We distinguish between communication of the two components with each other (on their common channels), and their communication with their environment. In the former case, the components must "wait" for each other in order to progress together. In the latter case, the communication actions of the two components interleave asynchronously.

Formally, the *parallel composition* of M_1 and M_2, denoted $M_1 || M_2$, is the program $M = \langle Q, x, \alpha, \delta, q_0, F \rangle$, defined as follows.

1. $Q = (Q_1 \times Q_2) \cup (Q'_1 \times Q'_2)$, where Q'_1 and Q'_2 are new copies of Q_1 and Q_2, respectively. The initial state is $q_0 = (q_0^1, q_0^2)$.

2. $X = X_1 \cup X_2$.

3. $\alpha = \{ (g?x_1, g!x_2), (g!x_1, g?x_2) \ : \ g*x_1 \in (\alpha_1 \cap \alpha I)$ and $g*x_2 \in (\alpha_2 \cap \alpha I)\} \cup ((\alpha_1 \cup \alpha_2) \setminus \alpha I)$. That is, the alphabet includes pairs of read-write communication actions on channels that are common to M_1 and M_2. It also includes individual actions of M_1 and M_2, which are not communications on common channels.

4. δ is defined as follows.

   (a) For $(g * x_1, g * x_2) \in \alpha^2$:

      i. $\delta((q_1, q_2), (g * x_1, g * x_2)) = (q'_1, q'_2)$.

      ii. $\delta((q'_1, q'_2), x_1 = x_2) \ = \ (\delta_1(q_1, g * x_1), \delta_2(q_2, g * x_2))$.

      That is, when a communication is performed synchronously in both components, the data is transformed through the channel from the writing component to the reading component. As a result, the values of x_1 and x_2 equalize. This is enforced in M by adding a transition labeled by the constraint $x_1 = x_2$ that immediately follows the synchronous communication.

   (b) For $a \in \alpha_1 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (\delta_1(q_1, a), q_2)$.

   (c) For $a \in \alpha_2 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (q_1, \delta_2(q_2, a))$.

   That is, on actions that are not in the interface alphabet, the two components interleave.

5. $F = F_1 \times F_2$

Figure 4.1 demonstrates the parallel composition of components M_1 and M_2. The program $M = M_1 || M_2$ reads a password from the environment through channel $pass$. The two components synchronize on channel $verify$. Assignments to x are interleaved with reading the value of y from the environment.

4.2 Regular Properties and Their Satisfaction

The specifications we consider in this chapter are also given as some variation of communicating programs. We now define the syntax and semantics of the properties that we consider as specifications. These are properties that can be represented as finite automata, hence the name *regular*. However, the alphabet of such automata includes communication actions and first-order constraints over program variables. Thus, such automata are suitable for specifying the desired and undesired behaviors of communicating programs over time.

[2]Note that according to item 3, one of the actions must be a read action and the other one is a write action.

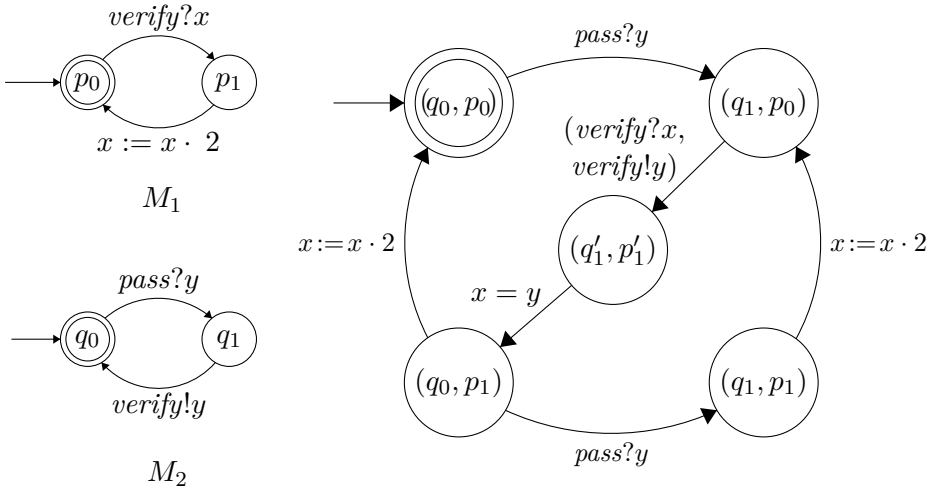


Figure 4.1: Components M_1 and M_2 and their parallel composition $M_1||M_2$.

In order to define our properties, we first need the notion of a *deterministic and complete* program. The definition is somewhat different from the standard definition for finite automata given in Chapter 2.1, since it takes the semantic meaning of constraints into account.

Intuitively, in a deterministic and complete program, every concrete execution has exactly one trace that induces it.

Definition 4.2.1. A communicating program over alphabet α is *deterministic and complete* if for every state q and for every action $a \in \alpha$ the following hold:

1. *Syntactic determinism and completeness.* There is exactly one state q' such that $\langle q, a, q' \rangle$ is in δ.[3]

2. *Semantic determinism.* If $\langle q, c_1, q' \rangle$ and $\langle q, c_2, q'' \rangle$ are in δ for constraints $c_1, c_2 \in \mathcal{C}$ such that $c_1 \neq c_2$ and $q' \neq q''$, then $c_1 \wedge c_2 \equiv false$.

3. *Semantic completeness.* Let C_q be the set of all constraints on transitions leaving q. Then $(\bigvee_{c \in C_q} c) \equiv true$.

A *property* is a deterministic and complete program with no assignment actions, whose language defines the set of desired and undesired behaviors over the alphabet αP.

A trace is accepted by a property P if it reaches a state in F, the set of accepting states of P. Otherwise, it reaches a state in $Q \setminus F$, and is rejected by P.

Next, we define the satisfaction relation $\models$ between a program and a property. Intuitively, a program M satisfies a property P (denoted $M \models P$) if all executions induced by accepted traces of M reach an accepting state in P. Thus, the accepted behaviors of M are also accepted by P.

A property P specifies the behavior of a program M by referring to communication actions of M and imposing constraints over the variables of M. Thus, the set of variables of P is

[3] in our examples we sometimes omit the actions that lead to a rejecting sink for the sake of clarity.

identical to that of M. Let $\mathcal{G}$ be the set of communication actions of M. Then, αP includes a subset of $\mathcal{G}$ as well as constraints over the variables of M. The *interface* of M and P, which consists of the communication actions that occur in P, is defined as $\alpha I = \mathcal{G} \cap \alpha P$.

In order to capture the satisfaction relation between M and P, we define a *conjunctive composition* between M and P, denoted $M \times P$. In conjunctive composition, the two components synchronize on their common communication actions when both read or both write through the same communication channel. They interleave on constraints and on actions of αM that are not in αP.

Definition 4.2.2. Let $M = \langle Q_M, X_M, \alpha M, \delta_M, q_0^M, F_M \rangle$ be a program and $P = \langle Q_P, X_P, \alpha P, \delta_P, q_0^P, F_P \rangle$ be a property, where $X_M \supseteq X_P$. The *conjunctive composition* of M and P is $M \times P = \langle Q, X, \alpha, \delta, q_0, F \rangle$, where:

1. $Q = Q_M \times Q_P$. The initial state is $q_0 = (q_0^M, q_0^P)$.
2. $X = X_M$.
3. $\alpha = \{\, g!x,\ g?x,\ (g?x, g!y),\ (g!x, g?y)\ :\ g{*}x,\ (g{*}x, g{*}y) \in \alpha I\} \cup ((\alpha M \cup \alpha P) \setminus \alpha I))$. Note that communication actions of the form $(g * x, g * y)$ can only appear if M is itself a parallel composition of two programs. That is, the alphabet includes communication actions on channels common to M and P. It also includes individual actions of M and P.
4. δ is defined as follows.
   - For $a = (g * x, g * y)$ in αI, or $a = g * x$ in αI, we define $\delta((q_1, q_2), a) = (\delta_M(q_1, a), \delta_P(q_2, a))$.
   - For $a \in \alpha M \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (\delta_M(q_1, a), q_2)$.
   - For $a \in \alpha P \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (q_1, \delta_P(q_2, a))$.

   That is, on actions that are not common communication actions to M and P, the two components interleave.
5. $F = F_M \times B_P$, where $B_P = Q_P \setminus F_P$.

Note that accepted traces in $M \times P$ are those that are accepted in M and rejected in P. Such traces are called *error traces* and their corresponding executions are called *error executions*. Intuitively, an error execution is an execution along M which violates the properties modeled by P. Such an execution either fails to synchronize on the communication actions, or reaches a point in the computation in which its assignments violate some constraint described by P. These executions are manifested in the traces that are accepted in M but are composed with matching traces that are rejected in P. We can now formally define when a program satisfies a property.

Definition 4.2.3. For a program M and a property P, we define $M \vDash P$ iff $M \times P$ contains no feasible accepted traces.

Thus, a feasible error trace in $M \times P$ is an evidence to $M \not\models P$, since it indicates the existence of an execution that violates P.

Example 4.2.4. Consider the program M, the property P and a partial construction of $M \times P$ presented in Figure 4.2. P requires every verified password y to be of length at least 4. It is easy to see that $M \not\models P$, since the trace $t = (password?y, y > 0, verify!y, y < 1000)$ is a feasible error trace in $M \times P$.

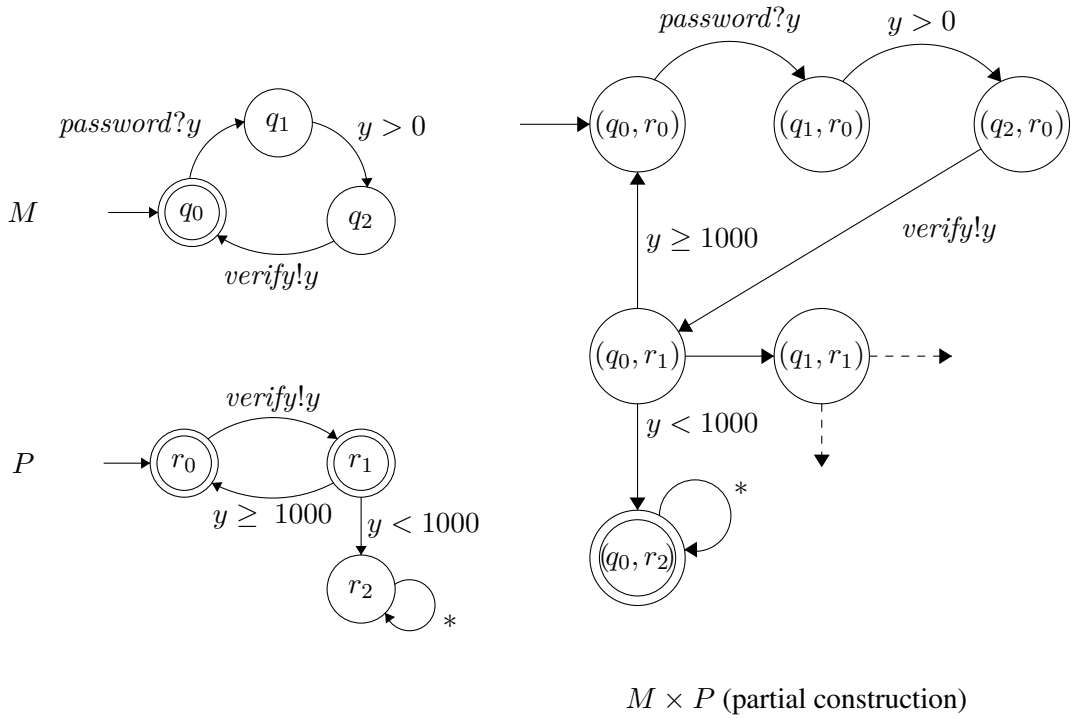


Figure 4.2: Partial conjunctive composition of M and P.

4.3 Traces in the Composed system

Before we discuss our framework for compositional verification and repair of communicating systems, we first prove some properties of traces in the composed system. We later use these properties in order to prove that our framework is sound and complete (Section 4.4.1), and to prove correctness and termination of our algorithm (Section 4.5.2 and Section 4.5.4).

Definition 4.3.1. Let t be a trace over alphabet α, and let $\alpha' \subseteq \alpha$. We denote by $t \downarrow_{\alpha'}$ the *restriction of* t *on* α', which is the trace obtained from t by omitting all letters in t that are not in α'. If α contains a communication action $a = (g * x, g * y)$ and we have $g * x \in \alpha'$ then the restriction $t \downarrow_{\alpha'}$ includes the corresponding communication, $g * x$. For $g * y \in \alpha'$ it is defined similarly.

Example 4.3.2. Let $\alpha = \{g_1!x, x := x+1, x < 10, (g_2?x, g_2!y)\}$, and $\alpha' = \{g_1!x, x := x+1, g_2?x\}$. Then we have $(g_2?x, g_2!y) \downarrow_{\alpha'} = g_2?x$, and for

$$t = ((g_2?x, g_2!y), x := x+1, x := x+1, x < 10, g_1!x)$$

we have

$$t \downarrow_{\alpha'} = (g_2?x, x := x+1, x := x+1, g_1!x)$$

Lemma 4.3.3. *Let M be a communicating program and P be a property, and let t be a trace of $M \times P$. Then $t \downarrow \alpha M$ is a trace of M.*

Proof. Let $M = \langle Q_M, X_M, \alpha M, \delta_M, q_0^M, F_M \rangle$ and $P = \langle Q_P, X_P, \alpha P, \delta_P, q_0^P, F_P \rangle$, and denote $M \times P = \langle Q, X_M, \alpha, \delta, q_0, F \rangle$.[4]

Let $r = \langle q_0, c_1, q_1 \rangle \dots \langle q_{m-1}, c_m, q_m \rangle$ be the run in $M \times P$ such that t is induced from r. Denote by $t_M = t \downarrow_{\alpha M}$ the trace $t_M = (c_{i_1}, \cdots, c_{i_n})$.

We first observe the following. If $(a_1, \cdots, a_k)$ is a trace of $M \times P$ such that $\forall i : a_i \notin \alpha M$, and $q = (q_M, q_P^0)$ is the state in $M \times P$ before reading a_1, then $\forall i \geq 1 : \exists q_P^i : \delta((q_M, q_P^{i-1}), a_i) = (q_M, q_P^i)$, that is, when reading a trace that does not contain letters from αM, the program $M \times P$ only advances on the P component. This is true since by the definition of δ, if a_i is not in αM, then $\delta((q_M, q_P), a_i) = (q_M, \delta_P(q_P, a_i))$.

We now inductively prove that $\forall 1 \leq j \leq n$ it holds that $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M. In particular, for $j = n$ this means that $t_M \in M$.

Let $j := 1$ and denote $k := i_1$. Then $c_1, \dots, c_{k-1} \notin \alpha M$ since k is the first index of t for which $c_k \in \alpha M$. Thus, $\forall 1 \leq i < k : \exists q_i^P : \delta((q_0^M, q_{i-1}^P), c_i) = (q_0^M, q_i^P)$. For $c_{i_1} = c_k \in \alpha M$, by the definition of δ, we have $\delta((q_0^M, q_{k-1}^P), c_{i_1}) = (\delta_M(q_0^M, c_{i_1}), q')$ for some $q' \in Q_P$. Then indeed, $\langle q_0^M, c_{i_1}, \delta_M(q_0^M, c_{i_1}) \rangle$ is a run in M, making (c_{i_1}) a trace of M.

Let $1 < j \leq n$, and assume $t_{j-1} = (c_{i_1}, \cdots, c_{i_{j-1}})$ is a trace of M. Let $\langle q_0, c_{i_1}, q_1 \rangle \dots \langle q_{j-2}, c_{i_{j-1}}, q_{j-1} \rangle$ a run that induces t_{j-1}. Denote $i_{j-1} = k, i_j = k+m$ for some $m > 0$. Then, as before, $c_{k+1}, \dots, c_{k+m-1} \notin \alpha M$, thus $\forall k < l < k+m : \exists q_l^P : \delta(q_{j-1}, q_{l-1}^P), c_l) = (q_{j-1}, q_l^P)$. For c_{i_j} it holds that $\delta(q_{j-1}, q_{k+m-1}^P), c_{i_j}) = (\delta_M(q_{j-1}, c_{i_j}), q'))$ for some $q' \in Q_P$. Thus $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M, as needed. ■

Lemma 4.3.4. *Let M_1, M_2 be two programs, and let t be a trace of $M_1 || M_2$. Then $t \downarrow \alpha M_1$ is a trace of M_1 and $t \downarrow \alpha M_2$ is a trace of M_2.*

The proof is similar to the proof of Lemma 4.3.3, however, in this case we need to take special care of the communication actions.

Proof. Let $M_i = \langle Q_i, X_i, \alpha M_i, \delta_i, q_0^i, F_i \rangle$ for $i = 1, 2$, and denote $M_1 || M_2 = \langle (Q_1 \times Q_2) \cup (Q_1' \times Q_2'), X_1 \cup X_2, \alpha M, \delta, (q_0^1, q_0^2), F_1 \times F_2 \rangle$, as defined in Section 4.1.1 of Chapter 4.

Let $r = \langle q_0, c_1, q_1 \rangle \dots \langle q_{m-1}, c_m, q_m \rangle$ be the run in $M_1 || M_2$ such that t is induced from r. Denote by $t_1 = t \downarrow_{\alpha M_1}$ the trace $t_1 = (c_{i_1}, \cdots, c_{i_n})$. The proof for $t_2 = t \downarrow_{\alpha M_2}$ is the same.

[4] Recall that the set of variables X_P is a subset of X_M.

We first observe the following. If $(a_1, \cdots, a_k)$ is a trace of $M_1||M_2$ such that $\forall i : a_i \notin \alpha M_1$, and $a_1 \in \alpha M_2$, and $q = (q_1, q_2^0)$ is the state in $M_1||M_2$ before reading a_1, then $\forall i \geq 1 : \exists q_2^i : \delta((q_1, q_2^{i-1}), a_i) = (q_1, q_2^i)$, that is, when reading a trace that does not contain letters from αM_1, the program $M_1||M_2$ only advances on the M_2 component. This is true since by the definition of δ, if a_i is not in αM_1, then $\delta((q_1, q_2), a_i) = (q_1, \delta_2(q_2, a_i))$. The requirement of $a_1 \in \alpha M_2$ is since if $a_1 \notin \alpha m_2$ and $a_1 \notin \alpha m_1$ then a_1 is an equality constraint of the form $x = y$ after a communication over a common channel and the transition relation is defined differently.

We now inductively prove that $\forall 1 \leq j \leq n$ it holds that $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M_1. In particular, for $j = n$ this means that $t_1 \in M_1$. For some parts of the proof we abuse notations where c is a communication action over M_1 and M_2, and we use it also to denote the restriction to the first component of M_1.

Let $j := 1$ and denote $k := i_1$. Then $c_1, \ldots, c_{k-1} \notin \alpha M_1$ since k is the first index of t for which $c_k \in \alpha M_1$ or $c_k = (c_k^1, c_k^2)$ such that c_k is a communication action and $c_k^1 \in \alpha M_1$. In particular, this means that no common communication action had occurred until c_k. Thus, $\forall 1 \leq i < k : \exists q_i^2 : \delta((q_0^1, q_{i-1}^2), c_i) = (q_0^1, q_i^2)$. For $c_{i_1} = c_k \in \alpha M_1$, one of the following holds:

1. If $c_k \in \alpha M_1$ is not a common communication action, then by the definition of δ, we have $\delta((q_0^1, q_{k-1}^2), c_{i_1}) = (\delta_1(q_0^1, c_{i_1}), q_{k-1}^2)$. Then indeed, $\langle q_0^1, c_{i_1}, \delta_1(q_0^1, c_{i_1})\rangle$ is a run in M_1, making (c_{i_1}) a trace of M_1.

2. If $c_k = (c_k^1, c_k^2)$ is a common communication channel, then it holds that c_{k+1} is an equality constraint. Then, by the definition of δ we have that $\delta((q_0^1, q_{k-1}^2), c_{i_1}) = (q_0^{1\prime}, {q_{k-1}^2}')$ and $\delta((q_0^{1\prime}, {q_{k-1}^2}'), c_{i_1+1}) = (\delta_1(q_0^1, c_{i_1}^1), \delta_2(q_{k-1}^2, c_{i_1}^2))$. Then, again we have that $\langle q_0^1, c_{i_1}, \delta_1(q_0^1, c_{i_1})\rangle$ is a run in M_1, making (c_{i_1}) a trace of M_1.

Let $1 < j \leq n$, and assume $t_{j-1} = (c_{i_1}, \cdots, c_{i_{j-1}})$ is a trace of M_1. Let $\langle q_0, c_{i_1}, q_1\rangle \ldots \langle q_{j-2}, c_{i_{j-1}}, q_{j-1}\rangle$ be a run that induces t_{j-1}. Denote $i_{j-1} = k, i_j = k + m$ for some $m > 0$. Then, as before, $c_{k+1}, \ldots, c_{k+m-1} \notin \alpha M_1$ and is not a communication action as well, thus $\forall k < l < k+m : \exists q_l^2 : \delta(q_{j-1}, q_{l-1}^2), c_l) = (q_{j-1}, q_l^2)$. For c_{i_j} it holds that either $c_{i_j} \in \alpha M_1$ is not a communication action and then $\delta(q_{j-1}, q_{k+m-1}^2), c_{i_j}) = (\delta_1(q_{j-1}, c_{i_j}), q_{k+m-1}^2))$; or that $c_{i_j} = (c_{i_j}^1, c_{i_j}^2)$ is a communication action and then $\delta((q_{j-1}, q_{k+m-1}^2), c_{i_j}) = (q_{j-1}{}', {q_{k+m-1}^2}')$ and $\delta((q_{j-1}{}', {q_{k+m-1}^2}'), c_{i_1+1}) = (\delta_1(q_{j-1}, c_{i_j}^1), \delta_2(q_{k+m-1}^2, c_{i_j}^2))$. In both cases, in M_1 it holds that $\delta_1(q_{j-1}, c_{i_j})$ is defined in M_1 and thus $(c_{i_1}, \cdots, c_{i_j})$ is a trace of M_1, as needed. ■

We now discuss the feasibility of traces in the composed system.

Lemma 4.3.5. *Let M be a program and P be a property, and let t be a **feasible** trace of $M \times P$. Then $t \downarrow \alpha M$ is a **feasible** trace of M.*

Proof. Let $t \in \mathcal{T}(M \times P)$ be a feasible trace. Then, there exists an execution u on t. Denote $t = (b_1, \cdots, b_n)$ and $u = (\beta_0, b_1, \beta_1, \cdots, b_n, \beta_n)$. We inductively build an execution e on $t \downarrow_{\alpha M}$. The existence of such an execution e proves that $t \downarrow_{\alpha M}$ is feasible.

Let $t \downarrow_{\alpha M} = (c_1, \cdots, c_k)$. We set $e = (\gamma_0, c_1, \gamma_1, \cdots, c_k, \gamma_k)$ where $\gamma_0, \ldots, \gamma_k$ are defined as follows.

1. Set $j := 0, i := 0$.
2. Define $\gamma_0 := \beta_0$ and set $j := j + 1$.
3. Repeat until $j = k$:
   - Let $i' > i$ be the minimal index such that $b_{i'} = c_j$.
   - Define $\gamma_j := \beta_{i'}$ and set $j := j + 1, i := i' + 1$.

Note that for each $i < l < i'$ is holds that b_l is a constraint. This is true since by the definition of conjunctive composition (Definition 4.2.2), if b_l is not a constraint, then $b_l \in \alpha M$. But in that case, b_l has to synchronize with some alphabet letter in $t \downarrow_{\alpha M}$, contradicting the fact that i' is the minimal index for which $b_{i'} = c_j$. Thus, since u is an execution, and for all $i < l < i' : b_l$ is a constraint, it holds that $\forall i \leq l < i' : \beta_i = \beta_l$. In particular, it holds that $\beta_{i'-1} = \beta_i = \gamma_{j-1}$. Now, since $b_{i'} = c_j$, we can assign γ_j to be the same as $\beta_{i'}$ and result in a valid assignment. Thus, e is a valid execution on $t \downarrow_{\alpha M}$, making $t \downarrow_{\alpha M}$ feasible as needed. ■

Lemma 4.3.6. *Let M_1, M_2 be two programs, and let t be a **feasible** trace of $M_1||M_2$. Then $t \downarrow \alpha M_i$ is a **feasible** trace of M_i for $i \in 1, 2$.*

The proof of Lemma 4.3.6 is different from the proof of Lemma 4.3.5, since here we can no longer use the exact same assignments as the ones of the run on $M_1||M_2$. In the case of $M \times P$, the variables of $M \times P$ are the same as the variables of M, and the two runs only differ on the constraints that are added to the trace of $M \times P$. In $M_1||M_2$, on the other hand, M_1 and M_2 are defined over two different sets of variables, with empty intersection between them. Nevertheless, The proof is similar to the proof of Lemma 4.3.5.

Proof of Lemma 4.3.6. Denote by X_i the set of variables of M_i for $i \in \{1, 2\}$. Let $t \in M_1||M_2$ be a feasible trace. Then, there exists an execution u on t. Denote $t = (b_1, \cdots, b_n)$ and $u = (\beta_0, b_1, \beta_1, \cdots, b_n, \beta_n)$. We build an execution e on $t \downarrow_{\alpha M_1}$ as follows (in the same way, we can build an execution on $t \downarrow_{\alpha M_2}$). Let $t \downarrow_{\alpha M_1} = (c_1, \cdots, c_k)$. We define $e = (\gamma_0, c_1, \gamma_1, \cdots, c_k, \gamma_k)$ as follows.

1. Set $j := 0, i := 0$.
2. Define $\gamma_0 := \beta_0(X_1)$ and set $j := j + 1$.
3. Repeat until $j = k$:
   - Let $i' > i$ be the minimal index such that $b_{i'} = c_j$ or $b_{i'} = (g * x, g * y)$ for $c_j = g * x \in \alpha M_1$.
   - Define $\gamma_j := \beta_{i'}(X_1)$ and set $j := j + 1, i := i' + 1$.

Note that for each $i < l < i'$ is holds that $b_l \notin \alpha M_1$ and there are no $x \in X_1$ and $y \in X_2$ such that $b_l = (g * x, g * y)$ for $g * x \in \alpha M_1$. Otherwise, b_l is either a synchronization between M_1 and M_2, or b_l is a letter of M_1 that belongs to αM_1 and is part of $t \downarrow_{\alpha M_1}$. Both are contradiction to the fact that i' is the minimal index for which $b_{i'}$ and c_j are either equal or that c_j is a read/write action and $b_{i'}$ is a synchronization on that action.

Since u is an execution, and for all $i < l < i'$ it holds that b_l does not contain variables of X_1, we have that $\beta_i(X_1) = \beta_l(X_1)$ for all $i < l < i'$. This is since an assignment to a variable may only change if the variable is involved in the action alphabet. In particular, it holds that $\beta_{i'-1}(X_1) = \beta_i(X_1) = \gamma_{j-1}(X_1)$. We now can assign γ_j to be the same as $\beta_{i'}(X_1)$ and result in a valid assignment, as needed. ■

4.4 The Assume-Guarantee Rule for Communicating Systems

Let M_1 and M_2 be two programs, and let P be a property. The classical Assume-Guarantee (AG) proof rule [Pnu85] assures that if we find an assumption A (in our case, a communicating program) such that $M_1||A \vDash P$ and $M_2 \vDash A$ both hold, then $M_1||M_2 \vDash P$ holds as well. For labeled transition systems over a finite alphabet (LTSs) [CGP03a], the AG-rule is guaranteed to either prove correctness or return a real (non-spurious) counterexample. The work in [CGP03a] relies on the $\mathbf{L}^*$ algorithm [Ang87b] for learning an assumption A for the AG-rule. In particular, $\mathbf{L}^*$ aims at learning A_w, the weakest assumption for which $M_1||A_w \vDash P$. A crucial point of this method is the fact that A_w is *regular* [GPB02], and thus can be learned by $\mathbf{L}^*$. For communicating programs, this is not the case, as stated in Lemma 4.4.2.

Definition 4.4.1 (Weakest Assumption)**.** Let P be a property and M be a system. The weakest assumption A_w with respect to M and P has the language $\mathcal{L}(A_w) = \{w : M||w \vDash P\}$. That is, A_w is the set of all words that together with M satisfy P.

Lemma 4.4.2. *For infinite-state communicating programs, the weakest assumption A_w is not always regular.*

Proof. Consider the programs M_1 and M_2, and the property P of Figure 4.3. Let $\alpha M_2 = \{x := 0,\ y := 0,\ x := x + 1,\ y := y + 1,\ sync\}$. Note that in order to satisfy P, after the *sync* action, a trace t must pass the test $x = y$. Also note that the weakest assumption A_w does not depend on the behavior of M_2, but only on its alphabet. Assume by way of contradiction that $\mathcal{L}(A_w)$ is a regular language, and consider the language

$$L = \{x := 0\} \cdot \{y := 0\} \cdot \{x := x + 1, y := y + 1\}^* \cdot \{sync\}$$

By closure properties of regular languages, it holds that L is a regular language, and thus following our assumption, we have that $L \cap \mathcal{L}(A_w)$ is regular as an intersection of two regular languages. However $L \cap \mathcal{L}(A_w)$ is the set of all words that after the initialization $\{x := 0\}\{y :=$

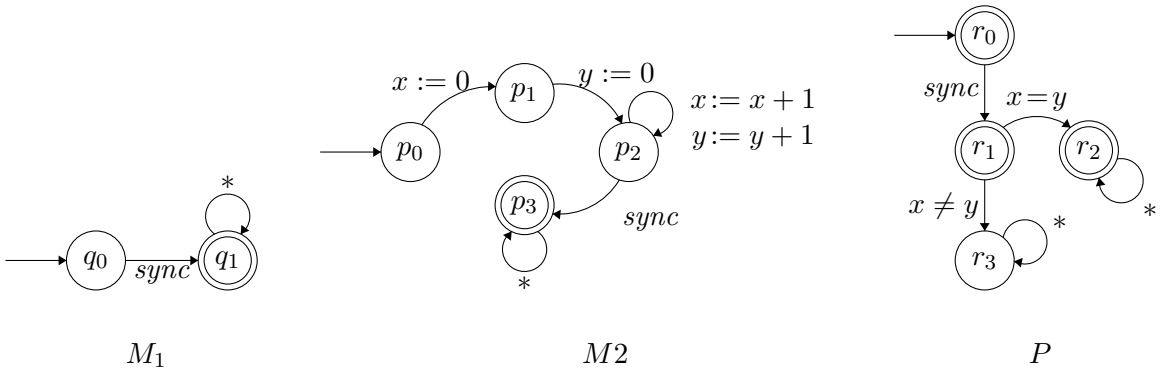
$0\}$, contain equally many actions of the form $x := x + 1$ and $y := y + 1$. That is

$$L \cap \mathcal{L}(A_w) = \{x := 0\} \cdot \{y := 0\} \cdot L_{eq} \cdot \{sync\}$$

where

$$L_{eq} = \{u \in \{x := x{+}1, y := y{+}1\}^* \ : \ \text{num of } x := x{+}1 \text{ in } u \text{ is equal to num of } y := y{+}1 \text{ in } u\}$$

L_{eq} is not regular since the pumping lemma does not hold for it, and for the same reason $L \cap \mathcal{L}(A_w)$ is not regular as well, contradicting our assumption that $\mathcal{L}(A_w)$ is regular. ■



M_1 $M2$ P

Figure 4.3: A system for which the weakest assumption is not regular.

To cope with this difficulty, we change the focus of learning. Instead of learning the (possibly) non-regular language of A_w, we learn $\mathcal{T}(M_2)$, the set of accepted traces of M_2. This language is guaranteed to be regular, as it represented by the automaton M_2.

4.4.1 Soundness and Completeness of the Assume-Guarantee Rule for Communicating Systems

Since we have changed the goal of learning, we first show that in the setting of communicating systems, the assume-guarantee rule is sound and complete.

Theorem 4.1. *For communicating programs, the Assume-Guarantee rule is sound and complete. That is,*

- *Soundness: for every communicating program A such that $\alpha A \subseteq \alpha M_2$, if $M_1||A \vDash P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$ then $M_1||M_2 \vDash P$.*
- *Completeness: If $M_1||M_2 \vDash P$ then there exists an assumption A such that $M_1||A \vDash P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$.*

Proof. Completeness. If $M_1||M_2 \vDash P$, then we can choose $A = M_2$, and then it holds that $M_1||A \vDash P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$.

Soundness. Assume by way of contradiction that there exists an assumption A such that $M_1||A \models P$ and $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, but $M_1||M_2 \not\models P$. Therefore, there exists an error trace $t \in (M_1||M_2) \times P$. By Lemma 4.3.3 and Lemma 4.3.4, it holds that $t_2 = t \downarrow_{\alpha M_2} \in \mathcal{T}(M_2)$ and by Lemma 4.3.5 and Lemma 4.3.6 it holds that t_2 is feasible. Since $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, it holds that $t_2 \in A$ and thus t is an error trace in $(M_1||A) \times P$, contradicting $M_1||A \models P$. ■

4.5 The Assume-Guarantee-Repair (AGR) Framework

In this section we discuss our Assume-Guarantee-Repair (AGR) framework for communicating programs. The framework consists of a learning-based Assume-Guarantee algorithm, called $\mathbf{AG}_{\mathbf{L}^*}$, and a REPAIR procedure, which are tightly joined.

Recall that the goal of $\mathbf{L}^*$ in our case is to learn $\mathcal{T}(M_2)$. The nature of $\mathbf{AG}_{\mathbf{L}^*}$ is such that the assumptions it learns before it reaches M_2 may contain the traces of M_2 and more, but still be represented by a smaller automaton. Therefore, similarly to [CGP03a], $\mathbf{AG}_{\mathbf{L}^*}$ often terminates with an assumption A that is much smaller than M_2. Indeed, our tool often produces very small assumptions (see Section 4.6).

As mentioned before, not only that we determine whether $M_1||M_2 \models P$, but we also repair the program in case it violates the specification. When $M_1||M_2 \not\models P$, the $\mathbf{AG}_{\mathbf{L}^*}$ algorithm returns an error trace t as a witness for the violation. In this case, we initiate the REPAIR procedure, which eliminates t from M_2. REPAIR applies abduction in order to learn a new constraint which, when added to t, makes the counterexample infeasible.[5] The new constraint enriches the alphabet in a way which may eliminate additional counterexamples from M_2, by making them infeasible. We elaborate on our use of abduction in Section 4.5.2. The removal of t and the addition of the new constraint result in a new goal M_2' for $\mathbf{AG}_{\mathbf{L}^*}$ to learn. We now return to $\mathbf{AG}_{\mathbf{L}^*}$ to search for a new assumption A' that allows to verify $M_1||M_2' \models P$.

An important feature of our AGR algorithm is its *incrementality*. When learning an assumption A' for M_2' we can use the membership queries previously asked for M_2, since the answer for them has not been changed. As we show later (Theorem 4.2 in Section 4.5.1), the difference between the languages of M_2 and M_2' lies in words (traces) whose membership has not yet been queried on M_2. This allows the learning of M_2' to start from the point where the previous learning has left off, resulting in a more efficient algorithm.

As opposed to the case where $M_1||M_2 \models P$, we cannot guarantee the termination of the repair process in case $M_1||M_2 \not\models P$. This is because we are only guaranteed to remove one (bad) trace and add one (infeasible) trace in every iteration (although in practice, every iteration may remove a larger set of traces). Thus, we may never converge to a repaired system. Nevertheless, in case of property violation, our algorithm always finds an error trace, thus a progress towards a "less erroneous" program is guaranteed.

It should be noted that the $\mathbf{AG}_{\mathbf{L}^*}$ part of our AGR algorithm deviates from the AG-rule of [CGP03a] in two important ways. First, since the goal of our learning is M_2 rather than

[5] There are also cases in which we do not use abduction, as discussed in Section 4.5.3.

A_w, our membership queries are different in type and order. Second, in order to identify real error traces and send them to REPAIR as early as possible, we add additional queries to the membership phase that reveal such traces. We then send them to REPAIR without ever passing through equivalence queries, which improves the overall efficiency. Indeed, our experiments include several cases in which all repairs were invoked from the membership phase. In these cases, AGR ran an equivalence query only when it has already successfully repaired M_2, and terminated.

4.5.1 The Assume-Guarantee-Repair (AGR) Algorithm

We now describe our AGR algorithm in more detail (see Algorithm 4.1). Figure 4.4 describes the flow of the algorithm. AGR comprises two main parts, namely $\mathbf{AG_{L^*}}$ and REPAIR.

The input to AGR are the components M_1 and M_2, and the property P. While M_1 and P stay unchanged during AGR, M_2 keeps being updated as long as the algorithm recognizes that it needs repair.

The algorithm works in iterations, where in every iteration the next updated M_2^i is calculated, starting with iteration $i = 0$, where $M_2^0 = M_2$. An iteration starts with the membership phase in line 2 of Algorithm 4.1, and ends either when $\mathbf{AG_{L^*}}$ successfully terminates (line 16) or when procedure REPAIR is called (lines 7 and 24). When a new system M_2^i is constructed, $\mathbf{AG_{L^*}}$ does not start from scratch. The information that has been used in previous iterations is still valid for M_2^i. The new iteration is given additional new trace(s) that have been added or removed from the previous M_2^i (lines 9,11,20, 27).

$\mathbf{AG_{L^*}}$ consists of two phases: membership, and equivalence.

The membership phase (lines 2-11) consists of a loop in which the learner constructs the next assumption A_j^i according to answers it gets from the teacher on a sequence of membership queries on various traces. These queries are answered in accordance with traces we allow in A_j^i. There are the traces in M_2^i that in parallel with M_1 satisfy P. If a trace t in M_2^i in parallel with M_1 does not satisfy P, then t is a bad behavior of M_2. Therefore, if such a t is found during the membership phase, REPAIR is invoked.

Once the learner reaches a stable assumption A_j^i, it passes it to the equivalence phase (lines 12-27). A_j^i is a suitable assumption if both $M_1||A_j^i \models P$ and $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A_j^i)$ hold. In this case, AGR terminates and returns M_2^i as a successful repair of M_2. If $M_1||A_j^i \not\models P$, then a counterexample t is returned, that is composed of bad traces in M_1, A_j^i, and P. If the bad trace t_2, the restriction of t to the alphabet of A_j^i, is also in M_2^i, then t_2 is a bad behavior of M_2^i, and here too the REPAIR phase is invoked. Otherwise, AGR returns to the membership phase with t_2 as a trace that should not be in A_j^i, and continues to learn A_j^i.

As we have described, REPAIR is called when a bad trace t is found in $(M_1||M_2^i) \times P$ and should be removed. If t contains no constraints then its sequence of actions is illegal and its restriction $t_2 \in \mathcal{T}(M_2^i)$ should be removed from M_2^i. In this case, REPAIR returns to $\mathbf{AG_{L^*}}$ with a new learning goal $\mathcal{T}(M_2^{i+1}) \subseteq \mathcal{T}(M_2^i) \setminus \{t_2\}$, along with the answer "no" to the membership query on t_2. In Section 4.5.3 we discuss different methods for removing t_2

Algorithm 4.1 AGR

1: **function** $\mathbf{AG}_{\mathbf{L}^*}$
2: //Membership Queries
3: Let $t_2 \in (\alpha M_2^i)^*$.
4: **if** $t_2 \in \mathcal{T}(M_2^i)$ **then**
5: **if** $M_1||t_2 \not\models P$ **then**
6: Let $t \in (M_1||t_2) \times P$ be an error trace. ▷ t is a cex proving $M_1||M_2^i \not\models P$
7: REPAIR(M_2^i, t)
8: **else** ▷ $M_1||t_2 \models P$
9: Return to $\mathbf{AG}_{\mathbf{L}^*}$ in Line 2 with $t_2 \in \mathcal{T}(A_j^i)$.
10: **else** ▷ $t_2 \notin \mathcal{T}(M_2^i)$
11: Return to $\mathbf{AG}_{\mathbf{L}^*}$ in Line 2 with $t_2 \notin \mathcal{T}(A_j^i)$.
12: //Equivalence Queries
13: Let A_j^i be the candidate assumption generated by the learner.
14: **if** $M_1||A_j^i \models P$ **then**
15: **if** $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A_j^i)$ **then**
16: Terminate and return $M_1||M_2^i \models P$.
17: **else**
18: Let $t_2 \in \mathcal{T}(M_2^i) \setminus \mathcal{T}(A_j^i)$.
19: Set $j := j + 1$
20: Return to $\mathbf{AG}_{\mathbf{L}^*}$ in Line 2 with $t_2 \in \mathcal{T}(A_j^i)$.
21: **else** ▷ $M_1||A_j^i \not\models P$
22: let $t \in (M_1||A_j^i) \times P$ be an error trace, and denote $t = (t_1||t_A) \times t_P$.
23: **if** $t_A \in \mathcal{T}(M_2^i)$ **then**
24: REPAIR(M_2^i, t_A) ▷ t_A is a cex proving $M_1||M_2^i \not\models P$
25: **else**
26: Set $j := j + 1$.
27: Return to $\mathbf{AG}_{\mathbf{L}^*}$ in Line 2 with $t_A \notin \mathcal{T}(A_j^i)$.
28: **function** REPAIR(M_2^i, t)
29: Let $t_1 \in M_1, t_2 \in M_2^i, t_p \in P$ such that $t = (t_1||t_2) \times t_p$.
30: **if** t does not contain constraints **then**
31: Return to $\mathbf{AG}_{\mathbf{L}^*}$ in Line 2 with $M_2^{i+1} = \mathcal{T}(M_2^i) \setminus \{t_2\}$ and $t_2 \notin \mathcal{T}(A_0^{i+1})$.
32: **else** ▷ t contains constraints
33: Use *abduction* to eliminate t.
34: Let c be the new constraint learned during abduction.
35: Update $\alpha M_2^{i+1} = \alpha M_2^i \cup \{c\}$.
36: Let $t_2' = t_2 \cdot c$ be the output of the abduction
37: Return to $\mathbf{AG}_{\mathbf{L}^*}$ in Line 2 with $M_2^{i+1} = (\mathcal{T}(M_2^i) \setminus \{t_2\}) \cup \{t_2'\}$,
38: and $t_2 \notin \mathcal{T}(A_0^{i+1}), t_2' \in \mathcal{T}(A_0^{i+1})$

from M_2^i.

The more interesting case is when t contains constraints. In this case, we not only remove the matching t_2 from M_2^i, but also add a new constraint c to the alphabet, which causes t_2 to be infeasible. This way we eliminate t_2, and may also eliminate a family of bad traces that violate the property in the same manner – adding a new constraint can only cause the removal of additional error traces, and cannot add traces to the system. We deduce c using abduction, see Section 4.5.2. As before, REPAIR returns to $\mathbf{AG}_{\mathbf{L}^*}$ with a new goal to be learned, but now also with an extended alphabet. The membership phase is then provided with two new answers to the membership query: t_2 that should *not* be included in the new assumption, and $(t_2 \cdot c)$ that should be included.

Implementing the $\mathbf{L}^*$ Teacher

As explained above, $\mathbf{AG}_{\mathbf{L}^*}$ uses the $\mathbf{L}^*$ algorithm in order to learn each assumption A_j^i, using membership and equivalence queries. We now formally describe how the teacher answers each of the queries. As we have noted, the target of learning is the set of traces of M_2. However, the learning process terminates once a suitable assumption of the AG-rule is found. Therefore, we denote the language the teacher answers according to by U, which informally denotes the desired result of the learning process.

***Membership Queries* (MQ)** Algorithm 4.1, lines 2-11.

Given a trace t, the teacher answers "is $t \in U$?" as follows.

- If $t \notin \mathcal{T}(M_2)$, answer "no".
- If $t \in \mathcal{T}(M_2)$ check if t is an error trace, and if so, turn directly to repair. That is -
  - If $M_1||t \not\models P$, pause learning and turn to repair.
  - If $M1||t \models P$, answer the MQ with "yes".

***Equivalence Queries* (EQ)** Algorithm 4.1 lines 12-27.

Given a candidate A, the teacher answers the EQ "$A \equiv U$" as follows.

- If $M_1||A \models P$, then A is a suitable candidate according to the AG-rule. Then, we check if the second condition of the AG-rule holds, that is, if $\mathcal{T}(M_2) \subseteq \mathcal{T}(A)$, and answer accordingly.
- If $M_1||A \not\models P$, then, check if the error trace that violates P is also a trace of M_2. If yes, them pause learning and turn to repair. If this is not a real error trace, return the answer "no" to the EQ , together with the error trace that needs to be eliminated from A.

Incremental learning

One of the advantages of AGR is that it is *incremental*, in the sense that answers to membership queries from previous iterations remain unchanged for the repaired system. Formally, we have the following.

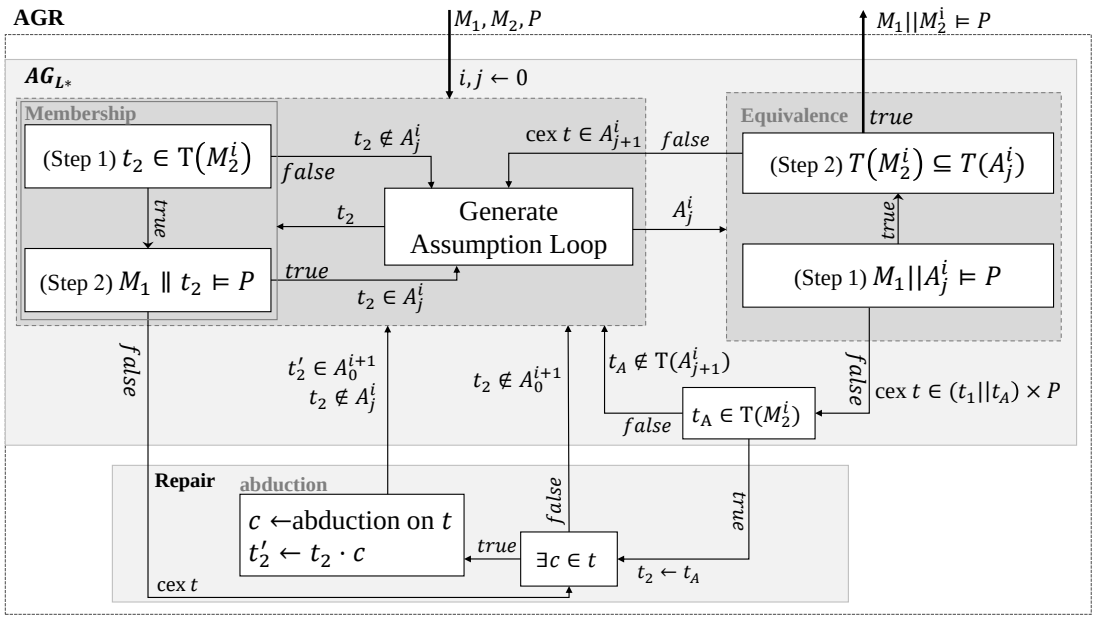


Figure 4.4: The flow of AGR

Theorem 4.2. *Let $\mathcal{T}(M_2^i)$ be the language learned by phase i of the AGR algorithm. Assume that phase i ended with a counterexample t and initiated a call to REPAIR(M_2^i, t). Then:*

1. *For every trace t_2 that was queried before, the answer remains the same for $\mathcal{T}(M_2^{i+1})$.*
2. *AGR did not query $t \downarrow_{\alpha M_2^i}$ before, thus removing it from $\mathcal{T}(M_2^{i+1})$ is consistent with all previous queries.*
3. *If t_2' is a trace learned using abduction, then AGR did not query it before.*

Proof. Item 1. Consider two cases. If t_2 was answered "$t_2 \in \mathcal{T}(A^j)$" for some previous iteration j , then in particular it holds that $M_1||t_2 \vDash P$ (line 9 of AGR algorithm). Since M_1 and P remain unchanged, then in all future iterations the same holds for t_2. Since We only remove a trace t from $\mathcal{T}(M_2^j)$ if it is an error trace, we conclude that for every j, t_2 is never removed from $\mathcal{T}(M_2^j)$. Thus, $t \in \mathcal{T}(M_2^{i+1})$.

If t_2 was queried before and the answer was "$t_2 \notin \mathcal{T}(A^j)$", then one of the following holds: $M_1||t_2 \nvDash P$ and as in the previous case, this remains true for all future iterations; or, $t_2 \notin \mathcal{T}(M_2^j)$ for some previous iteration j. Since we only remove traces, it holds that for every $j < j', \mathcal{T}(M_2^{j'}) \subseteq \mathcal{T}(M_2^j)$. Thus, $t_2 \notin \mathcal{T}(M_2^{i+1})$ as needed.

Item 2. Due to item 1, all answers on previous queries can be used in order to learn $\mathcal{T}(M_2^j)$, thus maintaining information from previous iterations is consistent with the $\mathbf{L}^*$ algorithm. Since $\mathbf{L}^*$ does not query any trace more than once, item 2 follows.

Item 3. Since t_2' contains a new alphabet letter, it for sure was not queried in any of the previous iterations. ■

4.5.2 Semantic Repair by Abduction

We now describe the repair we apply to M_2^i, in case the error trace t contains constraints (see Algorithm 4.1, line 32). Error traces with no constraints are removed from M_2^i syntactically (line 31), while in abduction we *semantically* eliminate t by making it infeasible. The new constraints are then added to the alphabet of M_2^i and may eliminate additional error traces. Note that the constraints added by abduction can only restrict the behavior of M_2, making more traces infeasible. Therefore, we do not add counterexamples to M_2.

The process of inferring new constraints from known facts about the program is called *abduction* [DD13]. We now describe how we apply it. Given a trace t, let φ_t be the first-order formula (a conjunction of constraints), which constitutes the SSA representation of t [AWZ88]. In order to make t infeasible, we look for a formula ψ such that $\psi \wedge \varphi_t \rightarrow \mathit{false}$.[6]

Note that $t \in \mathcal{T}(M_1||M_2^i) \times P$, and so it includes variables both from X_1, the set of variables of M_1, and from X_2, the set of variables of M_2^i. Since we wish to repair M_2^i, the learned ψ is over the variables of X_2 only.

The formula $\psi \wedge \varphi_t \rightarrow \mathit{false}$ is equivalent to $\psi \rightarrow (\varphi_t \rightarrow \mathit{false})$. Then, $\psi = \forall x \in X_1 : (\varphi_t \rightarrow \mathit{false}) = \forall x \in X_1(\neg\varphi_t)$, is such a desired constraint: ψ makes t infeasible and is defined only over the variables of X_2. We now use quantifier elimination [Wei84] to produce a quantifier-free formula over X_2. Computing ψ is similar to the abduction suggested in [DD13], but the focus here is on finding a formula over X_2 rather than over any minimal set of variables as in [DD13]; in addition, in [DD13] they look for ψ such that $\varphi_t \wedge \psi$ is not a contradiction, while we specifically look for ψ that blocks φ_t. We use Z3 [DMB08] to apply quantifier elimination and to generate the new constraint. After generating $\psi(X_2)$, we add it to the alphabet of M_2^i (line 35 of Algorithm 4.1). In addition, we produce a new trace $t_2' = t_2 \cdot \psi(X_2)$. The trace t_2' is returned as the output of the abduction.

We now turn to prove that by making t_2 infeasible, we eliminate the error trace t.

Lemma 4.5.1. *Let $t = (t_1||t_2) \times t_P$. If t_2 is infeasible, then t is infeasible as well.*

Proof. This is due to the fact that t_P can only restrict the behaviors of t_1 and t_2, thus if t_2 is infeasible, t cannot be made feasible. Formally, Lemma 4.5.1 follows from Lemma 4.3.5 and Lemma 4.3.6 given in Section 4.3. By Lemma 4.3.5, if $t = (t_1||t_2) \times t_P$ is feasible, then $t_1||t_2$ is a feasible trace of $M_1||M_2$. By Lemma 4.3.6, if $t_1||t_2$ is feasible, then t_2 is feasible as well. Therefore, if t_2 is infeasible, then t is infeasible, proving Lemma 4.5.1. ■

In order to add $t_2 \cdot \psi(X_2)$ to M_2^i while removing t_2, we split the state q that t_2 reaches in M_2^i into two states q and q', and add a transition labeled $\psi(X_2)$ from q to q', where only q' is now accepting, see Figure 4.5. Thus, we eliminate the violating trace from $M_1||M_2^i$. AGR now returns to $\mathbf{AG}_{\mathbf{L}^*}$ in order to learn an assumption for the repaired component M_2^{i+1}, which now includes t_2' but not t_2. Note that ψ is also added to t' of Figure 4.5. The new constraint then

[6] Usually, in abduction, we look for ψ such that $\psi \wedge \varphi_t$ is not a contradiction. In our case, however, since φ_t is a violation of the specification, we want to infer a formula that makes φ_t unsatisfiable.

blocks assignments of t' that violate P in the same way as t_2, but it allows for other assignments of t' to hold.

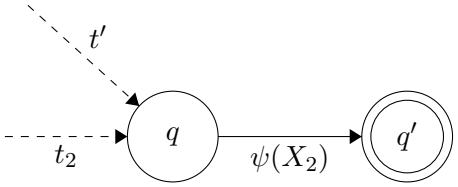


Figure 4.5: Adding the constraint $\psi(X_2)$ to block the error trace t_2

4.5.3 Syntactic Removal of Error Traces

Recall that the goal of REPAIR is to remove a bad trace t from M_2 once it is found by $\mathbf{AG}_{\mathbf{L}^*}$. If t contain constraints, we remove it by using abduction as described in Section 4.5.2. Otherwise, we can remove t by constructing a system whose language is $\mathcal{T}(M_2) \setminus \{t\}$. We call this the *exact* method for repair. However, removing a single trace at a time may lead to slow convergence, and to an exponential blow-up in the repaired systems. Moreover, as we have discussed, in some cases there are infinitely many such traces, in which case AGR may never terminate.

For faster convergence, we have implemented two additional heuristics, namely *approximate* and *aggressive*. These heuristics may remove more than a single trace at a time, while keeping the size of the systems small. While "good" traces may be removed as well, the correctness of the repair is maintained, since no bad traces are added. Moreover, an error trace is likely to be in an erroneous part of the system, and in these cases our heuristics manage to remove a set of error traces in a single step.

We survey the three methods.

- *Exact.* To eliminate only t from M_2, we construct a program (an automaton) A_t that accepts only t, and complement it to construct $\bar{A}_t$ that accepts all traces except for t. Finally, we intersect $\bar{A}_t$ with M_2. This way we only eliminate t, and not other (possibly good) traces. On the other hand, this method converges slowly in case there are many error traces, or does not converge at all if there are infinitely many error traces.

- *Approximate.* Similarly to our repair via abduction in Section 4.5.2, we prevent the last transition that t takes from reaching an accepting state. Let q be the state that M_2 reaches to, when reading t. We mark q as non-accepting, and add an accepting state q', to which all in-going transitions to q are diverted, except for the last transition on t. This way, some traces that lead to q are preserved by reaching q' instead, and the traces that share the last transition of t are eliminated along with t. As we have argued, these transitions may also be erroneous.

- *Aggressive.* In this simple method, we remove q, the state that M_2 reaches to when reading t, from the set of accepting states. This way we eliminate t along with all other traces that lead to q. In case that every accepting state is reached by some error trace,

this repair might result in an empty language, creating a trivial repair. However, our experiments show that in most cases, this method quickly leads to a non-trivial repair.

4.5.4 Correctness and Termination

For this discussion, we assume a sound and complete teacher who can answer the membership and equivalence queries in $\mathbf{AG}_{\mathbf{L}^*}$, which require verifying communicating programs and properties with first-order constraints. Our implementation uses Z3 [DMB08] in order to answer satisfiability queries issued in the learning process.

As we have discussed earlier, AGR is not guaranteed to terminate, and there are cases where the REPAIR stage may be called infinitely many times. However, in case that no repair is needed, or if a repaired system is obtained after finitely many calls to REPAIR, then AGR is guaranteed to terminate with a correct answer.

To see why, consider a repaired system M_2^i for which $M_1||M_2^i \models P$. Since the goal of $\mathbf{AG}_{\mathbf{L}^*}$ is to syntactically learn M_2^i, which is regular, this stage will terminate at the latest when $\mathbf{AG}_{\mathbf{L}^*}$ learns exactly M_2^i (it may terminate sooner if a smaller appropriate assumption is found). Notice that, in particular, if $M_1||M_2 \models P$, then AGR terminates with a correct answer in the first iteration of the verify-repair loop.

REPAIR is only invoked when a (real) error trace t is found in M_2^i, in which case a new system M_2^{i+1}, that does not include t, is produced by REPAIR. If $M_1||M_2^i \not\models P$, then an error trace is guaranteed to be found by $\mathbf{AG}_{\mathbf{L}^*}$ either in the membership or equivalence phase. Therefore, also in case that M_2^i violates P, the iteration is guaranteed to terminate.

In particular, since every iteration of AGR finds and removes an error trace t, and no new erroneous traces are introduced in the updated system, then in case that M_2 has finitely many error traces, AGR is guaranteed to terminate with a repaired system, which is correct with respect to P.

To conclude the above discussion, Theorem 4.3 formally states the correctness and termination of the AGR algorithm. The proof of Theorem 4.3 follows from Lemma 4.5.2, Lemma 4.5.3, and Lemma 4.5.4 given bellow.

Theorem 4.3.

1. *If $M_1||M_2 \models P$ then AGR terminates with the correct answer.*
2. *If, after finitely many iterations, a repaired program M_2' is such that $M_1||M_2' \models P$, then AGR terminates with the correct answer (this is a generalization of item 1).*
3. *If an iteration i of AGR ends with an error trace t, then $M_1||M_2^i \not\models P$, where M_2^i is the updated system at iteration i.*
4. *If $M_1||M_2 \not\models P$ then AGR finds an error trace. In addition, M_2', the system post REPAIR, contains less error traces than M_2.*

Lemma 4.5.2. *Every iteration i of the AGR algorithm terminates.*

In addition, answers to MQs *and* EQs *are consistent with $\mathcal{T}(M_2^i)$. That is, whenever AGR returns "$t \in \mathcal{T}(A_j^i)$" to $\boldsymbol{L}^*$ (lines 9, 20) then indeed $t \in \mathcal{T}(M_2^i)$, and whenever AGR returns "$t \notin \mathcal{T}(A_j^i)$" to $\boldsymbol{L}^*$ (lines 11, 27) then indeed $t \notin \mathcal{T}(M_2^i)$.*

Proof. For one iteration of AGR, we show that both membership queries and equivalence queries are consistent with $\mathcal{T}(M_2^i)$. In addition, we show that if $M_1||M_2 \not\models P$ then REPAIR is invoked. Thus, if $M_1||M_2 \models P$, since $\mathbf{L}^*$ is an algorithm for learning a regular language, and since $\mathcal{T}(M_2^i)$ is a regular language, by the termination of $\mathbf{L}^*$ we conclude that each iteration terminates. Otherwise, the iteration terminates by a call to REPAIR.

Membership queries. For $t_2 \in (\alpha M_2^i)^*$, membership queries are of the form "is $t_2 \in \mathcal{T}(M_2^i)$?". The only case in which the algorithm does not return the same answer as the $\mathbf{L}^*$ teacher does, is when $t_2 \in \mathcal{T}(M_2^i)$ and $M_1||t_2 \not\models P$. In this case we conclude that $M_1||M_2^i \not\models P$ and thus REPAIR is called (line 7) and the iteration terminates immediately. Therefore membership queries are consistent with $\mathcal{T}(M_2^i)$.

Equivalence queries. The teacher returns to $\mathbf{L}^*$ with a counterexample in lines 20 and 27. In line 20, AGR returns $t \in \mathcal{T}(A_j^i)$. Indeed, it holds that t is a trace in $\mathcal{T}(M_2^i) \setminus \mathcal{T}(A_j^i)$ thus in $\mathcal{T}(M_2^i)$. In line 27 AGR returns $t \notin \mathcal{T}(A_j^i)$. In that case, the test of line 23 fails, that is, $t \notin \mathcal{T}(M_2^i)$. Therefore, in all cases where AGR returns an answer regarding a trace t, this answer is consistent with $\mathcal{T}(M_2^i)$. In order to prove that each iteration terminates, we consider now the cases where AGR does not return a counterexample for an EQ. This happens in line 16 where the algorithm terminates; and in line 24 where REPAIR is invoked. Thus, every iteration indeed terminates. ■

Lemma 4.5.3. *If $M_1||M_2^i \models P$, the AGR algorithm terminates with the correct answer. Otherwise, if $M_1||M_2 \not\models P$, AGR finds a counterexample witnessing the violation (and continues to repairing M_2^i).*

Proof. Assume that $M_1||M_2 \models P$. By Lemma 4.5.2, the answers to MQs and EQs are consistent with $\mathcal{T}(M_2^i)$, and from the correctness of $\mathbf{L}^*$ algorithm we conclude that the algorithm will eventually learn $\mathcal{T}(M_2^i)$. Note that in case that $M_1||M_2 \models P$ and that AGR learned $\mathcal{T}(M_2^i)$, that is $\mathcal{T}(A_j^i) = \mathcal{T}(M_2^i)$, then the test of line 15 holds and the algorithm terminates. That is, in case $M_1||M_2 \models P$ AGR terminates with the correct answer (line 16).

Assume that $M_1||M_2 \not\models P$. Then there exists an error trace $t \in (M_1||M_2) \times P$. From Lemmas 4.3.5, 4.3.6 it holds that $t_2 = t \downarrow_{\alpha M_2^i}$ is feasible in M_2. In particular, it holds that t is an error trace of $(M_1||t_2) \times P$. Thus, $M_1||t_2 \not\models P$. Since AGR converges towards $\mathcal{T}(M_2^i)$ (by Lemma 4.5.2), either t_2 shows up as a membership query, and then line 5 holds, thus the iteration terminates by a call to REPAIR with t_2 as a witness to the violation; or AGR continues to the equivalence query part. There, again, t_2 (or some other error trace) will come up as an error trace $t_2 \in \mathcal{T}(M_2^i)$, resulting in termination of the iteration in line 24, again, by calling REPAIR. ■

Note that although each phase converges towards $\mathcal{T}(M_2^i)$, it may terminate earlier. We show that in case the algorithm terminates before finding $\mathcal{T}(M_2^i)$, it returns the correct answer.

Lemma 4.5.4. *L^* terminates and returns the correct answer. That is:*

1. *If L^* outputs an assumption A, then $M_1||A \models P$ and there exists i such that $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$, thus we can conclude $M_1||M_2^i \models P$.*

2. *If a phase i of AGR ends with an error trace t, then $M_1||M_2^i \not\models P$.*

Proof. Item 1. Assume AGR returns an assumption A. We can then conclude that the following holds for A: there exists i such that $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A)$ and $M_1||A \models P$, since this is the only scenario in which an assumption A is returned. From the soundness of our AG rule (Theorem 4.1) it holds that $M_1||M_2^i \models P$.

Item 2. Assume now that a phase i of AGR ends with an error trace t. We prove that $M_1||M_2^i \not\models P$. First note that AGR may output such a trace both while making a membership query and while making an equivalence query. If t was found during a membership query (line 5), then there exists $t_2 \in T(M_2^i)$ such that $M_1||t_2 \not\models P$, and $t \in (M_1||t_2) \times P$. Since $t_2 \in \mathcal{T}(M_2^i)$, it holds that t is also an error trace of $(M_1||M_2^i) \times P$, proving $M_1||M_2^i \not\models P$.

If t was found during an equivalence query (line 24), then t is an error trace in $(M_1||A_j^i) \times P$. Moreover, $t \downarrow_{\alpha A_j^i} \in \mathcal{T}(M_2^i)$. This makes t an error trace of $(M_1||M_2^i) \times P$ as well, thus $M_1||M_2^i \not\models P$. This finishes the proof. ■

The proof of Theorem 4.3 follows almost directly from the lemmas above.

Proof of Theorem 4.3. Lemma 4.5.3 states that if $M_1||M_2^i \models P$ then AGR terminates with the correct answer. This implies item 1 and item 2.

In addition, Lemma 4.5.3 states that if $M_1||M_2^i \not\models P$ then AGR finds an error trace witnessing the violation. Once such an error trace is found, REPAIR is invoked (lines 7 and 24 of Algorithm 4.1). Since REPAIR eliminates at least one error trace, the system post REPAIR contains less bad traces, and item 4 follows.

Lemma 4.5.4 states that if an iteration i of AGR ends with an error trace, then $M_1||M_2^i \not\models P$. This implies item 3. ■

4.6 Experimental Results

We implemented our AGR framework in Java, integrating the $\mathbf{L}^*$ learner implementation from the LTSA tool [MK99]. We used Z3 [DMB08] to implement the teacher while answering the satisfaction queries in $\mathbf{AG}_{\mathbf{L}^*}$, and for abduction in REPAIR.

Table 4.1 displays results of running AGR on various examples, varying in their sizes, types of errors – semantic and syntactic – and their amount. The examples are available on [exa]. The *iterations* column indicates the number of iterations of the verify-repair loop, until a repaired M_2 is achieved. Examples with no errors were verified in the first iteration, and are indicated by *verification*. We tested the three repair methods described in Section 4.5.3. Figure 4.6 presents comparisons between the three methods in terms of run-time and the size of the repair and assumptions. Note that the graphs are given in logarithmic scale.

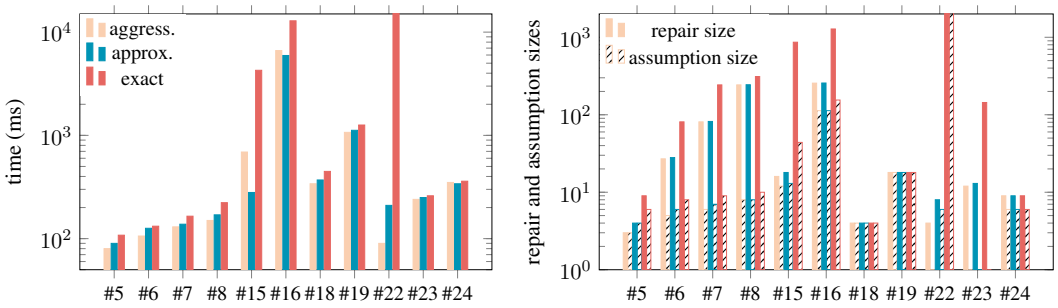


Figure 4.6: Comparing repair methods: time and repair size. Logarithmic scale

Most of our examples model multi-client-server communication protocols, with varying sizes. Our tool managed to repair all those examples that were flawed.

As can be seen in Table 4.1, our tool successfully generates assumptions that are significantly smaller than the repaired and the original M_2.

For the examples that needed repair, in most cases our tool needed 2-5 iterations of verify-repair in order to successfully construct a repaired component. Interestingly, in example $\#15$ the *aggressive* method converged slower than the *approximate* method. This is due to the structure of M_2, in which different error traces lead to different states. Marking these states as non-accepting removed each trace separately. However, some of these traces have a common transition, and preventing this transition from reaching an accepting state, as done in the *approximate* method, managed removing several error traces in a single repair.

Example $\#22$ models a simple structure in which, due to a loop in M_2, the same alphabet sequence can generate infinitely many error traces. The *exact* repair method timed out, since it attempted removing one error trace at a time. On the other hand, the *aggressive* method removed all accepting states, creating an empty program – a trivial (yet valid) repair. However, the *approximate* method created a valid, non-trivial repair.

4.7 Concluding Remarks

In this chapter we present the model of communicating programs that is able to capture program behavior and synchronization between the system components, while exploiting the finite automata representation in order to apply automata learning. We then present the AGR algorithm, that offers a new take on the learning-based approach to assume-guarantee verification, and manages coping with complex properties and repairing infinite-state programs.

Our experimental results show that using existing semantic tools, AGR produces very succinct proofs, and quickly and efficiently repairs flawed communicating programs.

Our algorithm exploits the finite automata-like representation of the systems in order to apply the $\mathbf{L}^*$ algorithm and to learn small proofs of correctness.

Example	M_1 Size	M_2 Size	P Size	Time (sec.)	A size	Repair Size	Repair Method	#Iterations
#1	4	4	3	0.2	3	verification		
#2	16	16	3	1.8	4	verification		
#3	32	32	3	11.1	6	verification		
#4	64	64	3	95	7	verification		
#5	2	3	2	0.08	3	3	aggress.	2
				0.09	4	4	approx.	2
				0.108	6	9	exact	2
#6	2	27	2	0.106	5	27	aggress.	2
				0.126	6	28	approx.	2
				0.132	8	81	exact	2
#7	2	81	2	0.13	6	81	aggress.	2
				0.138	7	82	approx.	2
				0.165	9	243	exact	2
#8	2	243	2	0.15	8	243	aggress.	2
				0.17	8	244	approx.	2
				0.223	10	729	exact	2
#9	2	4	3	0.093	3	verification		
#10	3	16	4	0.29	13	verification		
#11	5	256	6	4.88	92	verification		
#12	2	4	3	0.08	3	verification		
#13	3	16	4	0.22	10	verification		
#14	5	256	6	4.44	109	verification		
#15	3	16	5	0.69	12	16	aggress.	5
				0.28	13	18	approx.	3
				4.27	44	864	exact	5
#16	4	256	8	6.63	113	256	aggress.	2
				5.94	113	257	approx.	2
				12.87	155	1280	exact	2
#17	2	3	4	0.075	3	verification		
#18	2	3	4	0.34	5	4	aggress.	2
				0.37	5	4	approx.	2
				0.488	5	4	exact	2
#19	3	16	5	1.07	18	18	aggress.	3
				1.12	18	18	approx.	3
				1.26	18	18	exact	3
#20	9	6	15	0.1	6	verification		
#21	11	13	17	0.18	11	verification		
#22	2	4	2	0.09	1	4 (trivial)	aggress.	4
				0.21	6	8	approx.	5
				timeout			exact	timeout
#23	11	12	17	0.24	1	12 (trivial)	aggress.	2
				0.25	1	13 (trivial)	approx.	2
				0.26	1	144 (trivial)	exact	2
#24	4	8	5	0.35	6	9	aggress.	2
				0.34	6	9	approx.	2
				0.36	6	9	exact	2
#25	3	5	5	0.13	4	verification		
#26	4	5	5	0.11	4	verification		
#27	4	8	5	0.35	5	verification		

Table 4.1: AGR algorithm results on various examples

Chapter 5

Learning Symbolic Automata

Chapter 4 discusses compositional verification and repair, and demonstrates the use of automata learning for formal verification. There, we adjust the $\mathbf{L}^*$ algorithm for the setting of communicating programs. In this chapter, we discuss more abstract automata type, namely *symbolic automata*. Symbolic finite state automata, SFAs for short, are an automata model in which transitions between states correspond to predicates over a domain of concrete alphabet letters. Their purpose is to cope with situations where the domain of concrete alphabet letters is large or infinite. As opposed to the models we study in Chapters 4 and 3, where we use variables in order to keep track of data values throughout the computation, in the model of SFAs there is no notion of "memory". In SFAs, predicates are used in order to make the automaton transitions more concise, and allow us to express languages over infinite alphabets.

In this chapter we study the learnability of SFAs. The state-of-the-art literature on this topic follows the *query learning* paradigm, which stipulates that the *learner* can interact with an *oracle* (*teacher*) by asking it several types of allowed queries. So far all obtained results for SFAs are positive. We provide a necessary condition for efficient learnability of SFAs in this paradigm, from which we obtain the first negative result regarding the complexity of learnability of SFA over the propositional algebra. Most of this chapter studies learnability of SFAs under the paradigm of *identification in the limit using polynomial time and data*. We provide a necessary condition and a sufficient condition for efficient learnability of SFAs in this paradigm. We provide an efficient learning algorithm for monotonic algebras, and in particular for the interval algebra over the natural number or the reals. In addition, we prove that as in the query learning paradigm, here too the complexity of learnability of SFAs over the propositional algebra is not polynomial.

5.1 Preliminaries

5.1.1 Effective Boolean Algebra

A Boolean Algebra A is a tuple $\langle \mathbb{D}, \mathbb{P}, [\![\cdot]\!], \bot, \top, \vee, \wedge, \neg \rangle$ where $\mathbb{D}$ is a set of domain elements; $\mathbb{P}$ is a set of predicates closed under the Boolean connectives, where $\bot, \top \in \mathbb{P}$; the component

$[\![\cdot]\!] : \mathbb{P} \to 2^{\mathbb{D}}$ is the so-called *semantics function*. $\mathbb{P}$ satisfies the following three requirements: (i) $[\![\bot]\!] = \emptyset$, (ii) $[\![\top]\!] = \mathbb{D}$, and (iii) for all $\varphi, \psi \in \mathbb{P}$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathbb{D} \setminus [\![\varphi]\!]$. A Boolean Algebra is *effective* if all the operations above, as well as satisfiability, are decidable. Henceforth, we implicitly assume Boolean algebras to be effective.

One way to define a Boolean algebra is by defining a set $\mathbb{P}_0$ of *atomic formulas* that includes $\top$ and $\bot$, and obtaining $\mathbb{P}$ by closing $\mathbb{P}_0$ for conjunction, disjunction and negation. For a predicate $\psi \in \mathbb{P}$ we say that ψ is *atomic* if $\psi \in \mathbb{P}_0$. We say that ψ is *basic* if ψ is a conjunction of atomic formulas.

We now introduce two Boolean algebras that are discussed extensively below.

The Interval Algebra is the Boolean algebra in which the domain $\mathbb{D}$ is the set $\mathbb{Z} \cup \{-\infty, \infty\}$ of integers augmented with two special symbols with their standard semantics, and the set of atomic formulas $\mathbb{P}_0$ consists of intervals of the form $[a, b)$ where $a, b \in \mathbb{D}$ and $a < b$. The semantics associated with intervals is the natural one: $[\![[a, b)]\!] = \{z \in \mathbb{D} \ : \ a \leq z \text{ and } z < b\}$.

The Propositional Algebra is defined with respect to a set $AP = \{p_1, p_2, \ldots, p_k\}$ of atomic propositions. The set of *atomic predicates* $\mathbb{P}_0$ consists of the atomic propositions and their negations. The domain $\mathbb{D}$ consists of all the possible valuations for these propositions, thus $\mathbb{D} = \mathbb{B}^k$ where $\mathbb{B} = \{0, 1\}$. The semantics of an atomic predicate p is given by $[\![p_i]\!] = \{v \in \mathbb{B}^k \ : \ v[i] = 1\}$, and similarly $[\![\neg p_i]\!] = \{v \in \mathbb{B}^k \ : \ v[i] = 0\}$. In this case a basic formula is a *monomial*, that is, a conjunction of atomic predicated and their negations. .

5.1.2 Symbolic Automata

A *symbolic finite automaton* (SFA) is a tuple $\mathcal{M} = \langle \mathsf{A}, Q, q_0, \delta, F \rangle$ where A is a Boolean algebra, and as in the definition of finite automata in Section 2.1, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times \mathbb{P}_{\mathsf{A}} \times Q$ is the transition relation, where $\mathbb{P}_{\mathsf{A}}$ is the set of predicates of A.

We use the term *letters* for elements of $\mathbb{D}$ where $\mathbb{D}$ is the domain of A and the term *words* for elements of $\mathbb{D}^*$. A run of $\mathcal{M}$ on a word $\sigma_1\sigma_2 \ldots \sigma_n$ where $\sigma_i \in \mathbb{D}$, is a sequence of transitions $\langle q_0, \psi_1, q_1\rangle\langle q_1, \psi_2, q_2\rangle \ldots \langle q_{n-1}, \psi_n, q_n\rangle$ satisfying that $\sigma_i \in [\![\psi_i]\!]$ and that $\langle q_i, \psi_{i+1}, q_{i+1}\rangle \in \delta$. Such a run is said to be *accepting* if $q_n \in F$. A word $w = \sigma_1\sigma_2 \ldots \sigma_n$ is said to be *accepted* by $\mathcal{M}$ if there exists an accepting run of $\mathcal{M}$ on w. The set of words accepted by an SFA $\mathcal{M}$ is denoted $\mathcal{L}(\mathcal{M})$. Note that while the transitions of an SFA are symbolic, its language consists of concrete words.

An SFA is said to be *deterministic* if for every state $q \in Q$ and every letter $\sigma \in \mathbb{D}$ we have that $|\{\langle q, \psi, q'\rangle \in \delta \ : \ \sigma \in [\![\psi]\!]\}| \leq 1$, namely from every state and every concrete letter there exists at most one transition. It is said to be *complete* if $|\{\langle q, \psi, q'\rangle \in \delta \ : \ \sigma \in [\![\psi]\!]\}| \geq 1$ for every $q \in Q$ and every $\sigma \in \mathbb{D}$. As is the case for finite automata (over concrete alphabets), non-determinism does not add expressive power but does add succinctness [VdHT10]. When $\mathcal{A}$ is deterministic we use $\delta(q, w)$ to denote the state $\mathcal{A}$ reaches on reading the word w from state q. If $\delta(q_0, w) = q$ then w is termed an *access word to state* q.

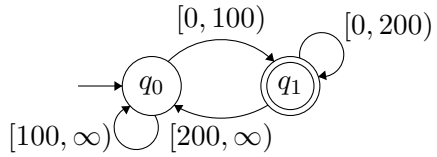


Figure 5.1: The SFA $\mathcal{M}$ over the interval algebra

5.2 Special Forms and their Properties

5.2.1 Types of Symbolic Automata

We turn to define special types of SFAs, which affect the complexity of related procedures.

Neat and Normalized SFAs We note that there is a trade-off between the number of transitions, and the complexity of the transition predicates. The literature defines an SFA as *normalized* if for every two states q and q' there exists at most one transition from q to q'. This definition prefers fewer transitions at the cost of potentially complicated predicates. By contrast, preferring simple transitions at the cost of increasing the number of transitions, leads to *neat* SFAs. We define an SFA to be *neat* if all transition predicates are basic predicates.

Feasibility The second distinction concerns the fact that an SFA can have transitions with unsatisfiable predicates. A symbolic automaton is said to be *feasible* if for every $\langle q, \psi, q' \rangle \in \delta$ we have that $[\![\psi]\!] \neq \emptyset$. Feasibility is an orthogonal property to being neat or normalized. Recall that feasibility checks are local to each transition.

Monotonicity The third distinction we make concerning the nature of a given SFA regards its underlying algebra. A Boolean algebra A over domain $\mathbb{D}$ is said to be *monotonic* if the following hold.

1. There exists a total order $<$ on the elements of $\mathbb{D}$; and
2. There exist two elements d_{inf}, d_{sup} such that $d_{inf} \leq d$ and $d \leq d_{sup}$ for all $d \in \mathbb{D}$; and
3. An atomic predicate $\psi \in \mathbb{P}_0$ can be associated with two concrete values a and b such that $[\![\psi]\!] = \{d \in \mathbb{D} \ : \ a \leq d < b\}$.

The interval algebra (given in Section 5.1.1) is clearly monotonic, as is the similar algebra obtained using $\mathbb{R}$ (the real numbers) instead of $\mathbb{Z}$ (the integers). On the other hand, the propositional algebra is clearly non-monotonic.

Example 5.2.1. Consider the SFA $\mathcal{M}$ given in Fig 5.1. It is defined over the algebra $A_{\mathbb{N}}$ which is the interval algebra restricted to the domain $\mathbb{D} = \mathbb{N} \cup \{\infty\}$. The language of $\mathcal{M}$ is the set of all words over $\mathbb{N}$ that end with a number between 0 and 100 followed by some (possibly empty) sequence of numbers smaller than 200. $\mathcal{M}$ is defined over a monotonic algebra, and is neat, normalized, deterministic and complete.

Example 5.2.2. (ASCII Algebra) The *ASCII Algebra* aims to model regular expressions in modern programming languages. The domain $\mathbb{D}$ is assumed to be the set of all ASCII codes

$\{n : 0 \leq n \leq 127\}$. The predicates contain rich ways to refer to sets of letters, e.g., `[A-Z]` and `[a-z]` denote the set of all ASCII upper and lower case letters, respectively, `[A-Za-z]` denotes their union, and `[^A-Za-z]` denotes the complement of `[A-Za-z]`. The ASCII algebra is defined with respect to a well ordered domain $\{n : 0 \leq n \leq 127\}$, and has predicates such as `[A-Z]` and `[a-z]` that can be defined by the two end points of the respective interval, but this is not the case for all predicates, and therefore it is not monotonic.

5.2.2 Size of an SFA

The size of an automaton (not a symbolic one) is typically measured by its number of states. This is since for DFAs, the size of the alphabet is assumed to be a given constant, and the rest of the parameters, in particular the transition relation, are at most quadratic in the number of states. In the case of SFAs the situation is different, as the size of the predicates labeling the transitions can vary greatly. In fact, if we measure the size of a predicate by the number of nodes in its parse DAG, then the size of a formula can grow unboundedly. The size and structure of the predicates influence the complexity of their satisfiability check, and thus the complexity of corresponding algorithms. On the other hand there might be a trade-off between the size of the transition predicates and the number of transitions; e.g. a predicate of the form $\psi_1 \vee \psi_2 \ldots \vee \psi_k$ can be replaced by k transitions, each one labeled by ψ_i for $1 \leq i \leq k$.

Therefore, we measure the size of an SFA by three parameters: the number of states (n), the maximal out-degree of a state (m) and the size of the most complex predicate (l). In order to analyze the complexity of automata algorithms discussed in Sections 5.2.4 and 5.2.5, for a class $\mathbb{P}$ of predicates over a Boolean algebra A, we also use the following measures: the complexity measure $sat^{\mathbb{P}}(l)$, which is the complexity of satisfiability check for a predicate of length l in $\mathbb{P}$; and the size measure $size^{\mathbb{P}}_{\wedge}(l_1, l_2)$ (or $size^{\mathbb{P}}_{\vee}(l_1, l_2)$), which is the size of the conjunction (disjunction) of two predicates in $\mathbb{P}$. While for the interval algebra $size^{\mathbb{P}}_{\wedge}(l_1, l_2)$ is linear in l_1 and l_2, for the OBDD (ordered binary decision diagrams) algebra Boolean operations on predicates are polynomial [Bry86]. When the algebra is built on a set of atomic predicates $\mathbb{P}_0$ we also use $sat^{\mathbb{P}_0}(l)$, $size^{\mathbb{P}_0}_{\wedge}(l_1, l_2)$ and $size^{\mathbb{P}_0}_{\vee}(l_1, l_2)$, for the respective complexities when restricted to atomic predicates.

5.2.3 Transformations to Special Forms

We now address the task of transforming SFAs into their special forms as presented in Section 5.2.1. Moreover, we discuss the complexity of standard procedures on SFAs of these special forms compared to the complexity on general SFAs. We start with transformations to the special forms *neat*, *normalized* and *feasible* automata, measured as suggested using $\langle n, m, l \rangle$ — the number of states, the maximal out-degree of a state, and the size of the most complex predicate.

Neat Automata

Since each predicate in a neat SFA is a conjunction of atomic predicates, neat automata are very intuitive, and the number of transitions in the SFA reflects the complexity of the different operations, as opposed to the situation with normalized SFAs. This is since most operations depend on satisfiability checking. For the class $\mathbb{P}_0$ of basic formulas, $sat^{\mathbb{P}_0}(l)$ is usually more efficient than $sat^{\mathbb{P}}(l)$, and in particular is polynomial for the algebras we consider here. This is since satisfiability testing can be reduced to checking that for a basic predicate φ that is a conjunction of l atomic predicates, there are no two atomic predicates that contradict each other. Since satisfiability checking directly affects the complexity of various algorithms discussed in Section 5.2.4, neat SFAs allow for efficient automata operations, as we show in Section 5.2.5.

Transforming to Neat Given a general SFA $\mathcal{M}$ of size $\langle n, m, l\rangle$, we can construct a neat SFA $\mathcal{M}'$ of size $\langle n, m \cdot 2^l, l\rangle$, by transforming each transition predicate to a DNF formula, and turning each disjunct into an individual transition. The number of states, n, remains the same. However, the number of transitions can grow exponentially due to the transformation to DNF. In the worst case, the size of the most complex predicate can remain the same after the transformation, resulting in the same l parameter for both automata. Note that there is no unique minimal neat SFA. For instance, a predicate ψ over the propositional algebra with $AP = \{p_1, p_2, p_3\}$, satisfying $[\![\psi]\!] = \{100, 101, 111\}$ can be represented using two basic transitions $(p_1 \wedge \neg p_2)$ and $(p_1 \wedge p_2 \wedge p_3)$; or alternatively using the two basic transitions: $(p_1 \wedge p_3)$ and $(p_1 \wedge \neg p_2 \wedge \neg p_3)$, though it cannot be represented using one basic transition.

Although in the general case, the transformation from normalized to neat SFAs is exponential, for monotonic algebras we have the following lemma, which follows directly from the definition of monotonic algebras and basic predicates.

Lemma 5.2.3. *Over a monotonic algebra, the conjunction of two atomic predicates is also an atomic predicate; inductively, any basic formula that does not contain negations, over a monotonic algebra, is an atomic predicate. In addition, the negation of an atomic predicate is a disjunction of at most 2 atomic predicates.*

Lemma 5.2.4. *Let $\mathcal{M}$ be a normalized SFA over a monotonic algebra A_{mon}. Then, transforming $\mathcal{M}$ into a neat SFA $\mathcal{M}'$ is linear in the size of $\mathcal{M}$.*

Since a DNF formula with m disjunctions is a natural representation of m neat transitions, Lemma 5.2.4 follows from the following property of monotonic algebras.

Lemma 5.2.5. *Let ψ be a general formula over a monotonic algebra A. Then, there exists an equivalent DNF formula ψ_d of size linear in $|\psi|$.*

Proof. First, we transform ψ into a Negation Normal Form formula ψ_{NNF}, pushing negations inside the formula. When transforming to NNF, the number of atomic predicates (under negation) remains the same, and so is the number of conjunctions and disjunctions. Since, by Lemma 5.2.3, a negation of an atomic predicate over a monotonic algebra, namely a negation of

an interval, results in at most two intervals, we get that $|\psi_{NNF}| \leq 2 \cdot |\psi|$. Note that ψ_{NNF} does not contain any negations, as they were applied to the intervals. We now transform ψ_{NNF} into a DNF formula ψ_d recursively, operate on sub-formulas of ψ_{NNF}, distributing conjunctions over disjunctions.

We inductively prove that $|\psi_d| = |\psi_{NNF}|$. For the base case, if ψ_{NNF} is a single interval $[a, b)$, then $[a, b)$ is in DNF and we are done.

For the induction step, consider the two cases.

1. Assume $\psi_{NNF} = \psi_1 \vee \psi_2$. By the induction hypothesis, there exists DNF formulas ψ_{1d} and ψ_{2d} such that $\psi_{id} \equiv \psi_i$ and $|\psi_i d| = |\psi_i|$ for $i = 1, 2$. Then, $\psi_d = \psi_{1d} \vee \psi_{2d}$ is equivalent to ψ_{NNF} and of the same size.

2. Assume $\psi_{NNF} = \psi_1 \wedge \psi_2$. Again, by the induction hypothesis, instead of $\psi_1 \wedge \psi_2$ we can consider $\psi_{1d} \wedge \psi_{2d}$ where ψ_{1d} and ψ_{2d} are in DNF. That is $\psi_{1d} = \bigvee_{i=1}^{k} [a_i, b_i)$ and $\psi_{2d} = \bigvee_{j=1}^{l} [c_j, d_j)$. Now,

$$\psi_{1d} \wedge \psi_{2d} = \left(\bigvee_{i=1}^{k} [a_i, b_i) \right) \wedge \left(\bigvee_{j=1}^{l} [c_j, d_j) \right) = \bigvee_{i=1}^{k} \bigvee_{j=1}^{l} \Big([a_i, b_i) \wedge [c_j, d_j) \Big)$$

   From properties of intervals, each conjunction $[a_i, b_i) \wedge [c_j, d_j)$ is of the form $[\max\{a_i, c_j\}, \min\{b_i, d_j\})$. The intervals in $\{[a_i, b_i) \ : \ 1 \leq i \leq k\}$ do not intersect (otherwise it would have resulted in a longer single interval), and the same for $\{[c_j, d_j) \ : \ 1 \leq j \leq l\}$. Thus, every element a_i or c_j can define at most one interval of the form $[\max\{a_i, c_j\}, \min\{b_i, d_j\})$. That is, the DNF formula $\psi_d = \bigvee_{i=1}^{k} \bigvee_{j=1}^{l} \Big([a_i, b_i) \wedge [c_j, d_j) \Big)$ contains at most $k + l$ intervals, as the others are not proper intervals. Since the size of the original ψ_{NNF} is $k + l$, we have that $|\psi_{NNF}| = |\psi_d|$.

To conclude, since ψ_{NNF} is linear in the size of ψ and ψ_d is of the same size as ψ_{NNF}, we have that the translation of ψ into the DNF formula ψ_d is linear. ■

Normalized Automata

Neat automata stand in contrast to normalized ones. In a normalized SFA, there is at most one transition between every pair of states, which allows for a succinct formulation of the condition to transit from one state to another. On the other hand, this makes the predicates on the transitions structurally more complicated. Given a general SFA $\mathcal{M}$ with parameters $\langle n, m, l \rangle$, we can easily construct a normalized SFA $\mathcal{M}'$ as follows. For every pair of states q and q', construct a single edge labeled with the predicate $\bigvee_{\langle q, \varphi, q' \rangle \in \delta} \varphi$. Then, $\mathcal{M}'$ has size $\langle n, n, size^{\mathbb{P}}_{\vee m}(l) \rangle$, where we use $size^{\mathbb{P}}_{\vee m}(l)$ to denote the size of m disjunctions of predicates of size at most l. Note that there is no unique minimal normalized automaton either, since in general Boolean formulas have multiple representations. However, in Section 5.2.5 we show that over monotonic algebras there is a canonical minimal normalized SFA.

The complexity of $sat^{\mathbb{P}}(l)$ for general formulas (corresponding to normalized SFAs) is usually exponentially higher than for basic predicates (and thus for neat SFAs). In addition, as we saw above, generating a normalized automaton is an easy operation. This motivates working with neat automata, and generating normalized automata as a last step, if desired (e.g., for drawing).

Feasible Automata

The motivation for feasible automata is clear; if the automaton contains unsatisfiable transitions, then its size is larger than necessary, and the redundancy of transitions makes it less interpretable. Thus, infeasible SFAs add complexity both algorithmically and for the user, as they are more difficult to understand. In order to generate a feasible SFA from a given SFA $\mathcal{M}$, we need to traverse the transitions of $\mathcal{M}$ and test the satisfiability of each transition. The parameters $\langle n, m, l\rangle$ of the SFA remain the same since there is no change in the set of states, and there might be no change in transitions as well (if they are all satisfiable).

In the following, we usually assume that the automata are feasible, and when applying algorithms, we require the output to be feasible as well.

5.2.4 Complexity of standard automata procedures on general SFAs

We turn to discuss Boolean operations, determinization and minimization, and decision procedures (such as emptiness and equivalence) for the different types of SFAs. For intersection and union, the product construction of SFAs was studied in [VdHT10, HV11]. There, the authors assume a normalized SFAs as input, and do not delve on the effect of the construction on the number of transitions and the complexity of the resulting predicates. Determinization of SFAs was studied in [VdHT10], and [DV14] study minimization of SFAs, assuming the given SFA is normalized.

Table 5.1 shows the sizes of the SFAs resulting from the mentioned operations, in terms of $\langle n, m, l\rangle$. The analysis applies to all types of SFAs, not just normalized ones. The time complexity for each operation is given in terms of the parameters $\langle n, m, l\rangle$ and the complexity of feasibility tests for the resulting SFA, as discussed in Section 5.2.3. Table 5.2 summarizes the time complexity of decision procedures for SFAs: emptiness, inclusion, and membership. Again, the analysis applies to all types of SFAs. We note that in many applications of learning in verification, the challenging part is implementing the teacher, for example in Chapter 4 of this thesis, as well as in [PGB$^+$08, CKKS20]. In such cases the complexity of membership and equivalence queries as well as standard automata operations plays a major role.

In both tables we consider two SFAs $\mathcal{M}_1$ and $\mathcal{M}_2$ with parameters $\langle n_i, m_i, l_i\rangle$ for $i = 1, 2$, over algebra A with predicates $\mathbb{P}$. We use $size^{\mathbb{P}}_{\wedge^m}(l)$ for an upper bound on the size of m conjunctions of predicates of size at most l. All SFAs are assumed to be deterministic, except of course for the input for determinization.

[1] For complementation, no feasibility check is needed, since we assume a feasible input.

[2] To determinize transitions, conjunction may be applied $n_1 \times m_1$ times, according to the number of states that

Operation	$\langle \mathbf{n}, \mathbf{m}, \mathbf{l} \rangle$
product construction $\mathcal{M}_1, \mathcal{M}_2$	$\langle n_1 \times n_2,\ m_1 \times m_2,\ size^{\mathbb{P}}_{\wedge}(l_1, l_2) \rangle$
complementation of deterministic $\mathcal{M}_1$ [1]	$\langle n_1 + 1,\ m_1 + 1,\ size^{\mathbb{P}}_{\vee^{m_1}}(l_1) \rangle$
determinization of $\mathcal{M}_1$	$\langle 2^{n_1},\ 2^{m_1},\ size^{\mathbb{P}}_{\wedge^{n_1 \times m_1}}(l_1) \rangle$ [2]
minimization of $\mathcal{M}_1$	$\langle n_1, m_1,\ size^{\mathbb{P}}_{\wedge^{m_1}}(l_1) \rangle$

Table 5.1: Analysis of standard automata procedures on SFAs.

Decision Procedures	Time Complexity
emptiness	linear in n, m
emptiness + feasibility	$n \times m \times sat^{\mathbb{P}}(l)$
membership of $\gamma_1 \cdots \gamma_t \in \mathbb{D}^*$	$\sum_{i=1}^{t} sat^{\mathbb{P}}(size^{\mathbb{P}}_{\wedge}(l, \lvert\psi_{\gamma_i}\rvert))$ [3]
inclusion $\mathcal{M}_1 \subseteq \mathcal{M}_2$	$((n_1 \times n_2) \times (m_1 \times m_2) \times sat^{\mathbb{P}}(size^{\mathbb{P}}_{\wedge}(l_1, l_2)))$

Table 5.2: Analysis of times complexity of decision procedures for SFAs

We now briefly describe the algorithms we analyze in both tables.

Product Construction [VdHT10, HV11] The product construction for SFAs is similar to the product of DFAs – the set of states is the product of the states of $\mathcal{M}_1$ and $\mathcal{M}_2$; and a transition is a synchronization of transitions of $\mathcal{M}_1$ and $\mathcal{M}_2$. That is, a transition from $\langle q_1, q_2 \rangle$ to $\langle p_1, p_2 \rangle$ can be made while reading a concrete letter γ, iff $\langle q_1, \psi_1, p_1 \rangle \in \delta_1$ and $\langle q_2, \psi_2, p_2 \rangle \in \delta_2$ and γ satisfies both ψ_1 and ψ_2. Therefore, the predicates labeling transitions in the product construction are conjunctions of predicates from the two SFAs $\mathcal{M}_1$ and $\mathcal{M}_2$.

Complementation In order to complement a deterministic SFA $\mathcal{M}_1$, we first need to make $\mathcal{M}_1$ complete. In order to do so, we add one state which is a non-accepting sink, and from each state we add at most one transition which is the negation of all other transitions from that state. In case that $\mathcal{M}_1$ is complete, then complementation only negates the set of accepting states, resulting in the same parameters $\langle n_1, m_1, l_1 \rangle$.

Determinization [VdHT10] In order to make an SFA deterministic, the algorithm of [VdHT10] uses the subset construction for DFAs, resulting in an exponential blowup in the number of states. However, in the case of SFAs this is not enough, and the predicates require special care. Let $P = \{q_1, \cdots, q_t\}$ be a state in the deterministic SFA, where $q_1, \ldots, q_t$ are states of the original SFA $\mathcal{M}_1$, and let $\psi_1, \ldots, \psi_t$ be some predicates labelling outgoing transitions from $q_1, \ldots q_t$, correspondingly. Then, in order to determinize transitions, the algorithm of [VdHT10] computes the conjunction $\bigwedge_{i=1}^{t} \psi_1$, which labels a single transition from the state P.

Minimization [DV14] Given a deterministic SFA $\mathcal{M}_1$, the output of minimization is an equivalent deterministic SFA with a minimal number of states. When constructing such an SFA, the number of states and transitions cannot grow. However, as in determinization, if two states of $\mathcal{M}_1$ are replaced with one state, then outgoing transitions might overlap, resulting in a non-deterministic SFA. Therefore, to make sure that transitions do not overlap, all algo-

correspond to a new deterministic state.

[3] Where ψ_{γ_i} is a predicate describing γ_i.

rithms described in [DV14] compute *minterms*, which are the smallest conjunctions of outgoing transitions. Minterms then do not intersect, and thus the output is deterministic.

Emptiness If we assume a feasible SFA $\mathcal{M}$ as an input, then in order to check for emptiness we need to find an accepting state which is reachable from the initial state (as in DFAs). If we do not assume a feasible input, we need to test the satisfiability of each transition, thus the complexity depends on the complexity measure $sat^{\mathbb{P}}(l)$.[4]

Membership Similarly to emptiness, in order to check if a concrete word $\gamma_1 \cdots \gamma_n$ is in $\mathcal{L}(\mathcal{M})$, we need to locally consider the satisfiability of each transition. In the case of membership, we need to check whether the letter γ_i satisfies the predicate on the corresponding transition.

Inclusion Deciding inclusion amounts to checking emptiness and feasibility of $\mathcal{M}_1 \cap \overline{\mathcal{M}_2}$. We assume here that both $\mathcal{M}_1$ and $\mathcal{M}_2$ are deterministic and complete.

5.2.5 Complexity of standard automata procedures on special SFAs

We now discuss the advantages of neat SFAs and of monotonic algebras, in the context of the algorithms presented in the tables, and show that, in general, they are more efficient to handle compared to other SFAs.

Neat SFAs

As can be observed from Table 5.2, almost all decision procedures regarding SFAs depend on $sat^{\mathbb{P}}(l)$. For neat SFAs it is more precise to say that they depend on $sat^{\mathbb{P}_0}(l)$, and since $sat^{\mathbb{P}_0}(l)$ is usually less costly than $sat^{\mathbb{P}}(l)$, most decision procedures are more efficient on neat automata. Here, we claim that applying automata algorithms on neat SFAs preserves their neatness, thus suggesting that neat SFAs may be preferable in many applications.

Lemma 5.2.6. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be neat SFAs. Then: $\mathcal{M}_1 \cap \mathcal{M}_2$, $\mathcal{M}_1 \cup \mathcal{M}_2$, $\overline{\mathcal{M}_1}$, and determinization / minimization of $\mathcal{M}_1$, are all neat SFAs as well.*

Proof. The proof follows from the product construction [VdHT10, HV11] and the determinization [VdHT10] and minimization [DV14] constructions. All of these use only conjunctions in order to construct the predicates on the output SFAs. Thus, if the predicates on the input SFAs are basic, then so are the output predicates. ■

Monotonic Algebras

We now consider the class $\mathbb{M}_{A_m}$ of SFAs over a monotonic algebra A_m with predicates $\mathbb{P}$. We first discuss $size^{\mathbb{P}}_{\wedge}(l_1, l_2)$ and $sat^{\mathbb{P}}(l)$, as they are essential measures in automata operations. Then we show that for $\mathcal{M}_1$ and $\mathcal{M}_2$ in the class $\mathbb{M}_{A_m}$, the product construction is linear in the number of transitions, adding to the efficiency of SFAs over monotonic algebras.

[4] Note that the feasibility check is local, and depends only on the satisfiability of the predicates labeling transitions. This is in contrast to the model of Chapter 4 where we check the feasibility of the whole word.

Lemma 5.2.7. *Let ψ_1 and ψ_2 be formulas over a monotonic algebra A_m. Then: $size^{\mathbb{P}}_{\wedge}(|\psi_1|, |\psi_2|)$ is linear in $|\psi_1| + |\psi_2|$ and $sat^{\mathbb{P}}(|\psi_1|)$ is linear in $|\psi_1|$.*

Proof. Transforming to DNF is linear, as we show in Lemma 5.2.5. There, we show that the conjunction of two DNF formulas of sizes k and l has size $k + l$, which implies that the conjunction of general formulas has linear size. In addition, $sat^{\mathbb{P}}(l)$ is trivial for a single interval, and following Lemma 5.2.5, is linear for general formulas. The satisfiability of a single interval is trivial, since we define intervals as predicates of the form $[a, b)$ for $a < b$, and thus every interval is satisfiable. Even if we allow unsatisfiable intervals, satisfiability check will amount to the question "is $a < b$?". ■

Lemma 5.2.8. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be deterministic SFAs over a monotonic algebra A_m. Then the out-degree of their product SFA $\mathcal{M}$ is at most $m = 2 \cdot (m_1 + m_2)$.*

Proof. From Lemma 5.2.4, we can construct neat SFAs $\mathcal{M}'_1$ and $\mathcal{M}'_2$ of sizes $\langle n_i, 2m_i, l_i \rangle$ for $i \in \{1, 2\}$. For a state q in an SFA, the set $\{\langle q, [a_i, b_i), p^i \rangle\}$ of q's outgoing transitions can be defined using the set $\{a_i\}$ of the minimal element of each transition. Similarly to the proof of Lemma 5.2.5, each transition $\langle \langle q_1, q_2 \rangle, [a, b) \wedge [c, d), \langle p_1, p_2 \rangle \rangle$ in the product SFA results in a formula $[\max\{a, c\}, \min\{b, d\})$. Then, for $q_1 \in Q_1$, every minimal element in the set of q_1's outgoing transitions can define at most one transition in $\mathcal{M}$, and the same for a state $q_2 \in Q_2$, and so the number of transitions from $\langle q_1, q_2 \rangle$ is at most $m_1 + m_2$, as required. ■

Lemma 5.2.9. *Let $\mathcal{M}$ be a neat SFA over a monotonic algebra. Then, transforming $\mathcal{M}$ into a complete SFA $\mathcal{M}'$ is polynomial in the size of $\mathcal{M}$.*

Proof. In order to complete $\mathcal{M}$, we add a non-accepting sink r in case it does not already exist, and at most m transitions from each state q to r, when m is the out-degree of the SFA, resulting in at most $|Q| \times m$ new transitions. ■

Definition 5.2.10. For predicates over a monotonic algebra, we define a *canonical representation* of a predicate ψ as the simplified DNF formula which is the disjunction of all intervals satisfying ψ.

Note that every predicate ψ over a monotonic algebra defines a unique partition of the domain into disjoint intervals. This unique partition corresponds to a simplified DNF formula, which is exactly the canonical representation of ψ.

Example 5.2.11. The canonical representation of $\psi = [0, 100) \wedge ([50, 150) \vee [20, 40))$ is $[20, 40) \vee [50, 100)$.

Lemma 5.2.12. *Let $\mathcal{M}$ be an SFA over a monotonic algebra. Then:*

1. *There is a unique minimal-state neat SFA $\mathcal{M}'$ such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.*
2. *There is a canonical minimal-state normalized SFA $\mathcal{M}''$ such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}'')$.*

Proof. First, we note that for a language $\mathcal{L} = \mathcal{L}(\mathcal{M})$ for some SFA $\mathcal{M}$, there exists a minimal-state SFA. This is since as, similarly to DFAs, when considering the equivalence relation N defined by $(u, v) \in N \iff \forall z \in \mathbb{D}^* : (uz \in \mathcal{L} \Leftrightarrow vz \in \mathcal{L})$, the equivalence classes of the relation N correspond to the states in the minimal-state SFA.

As for transitions, we have the following.

1. Let ψ be a general predicate labeling a transition in $\mathcal{M}$. Then ψ defines a unique partition of the domain into disjoint intervals, which are exactly the transitions in a neat SFA. Then, the minimal state neat SFA is unique.

2. For normalized transitions, we can use Lemma 5.2.5 to transform a general predicate labeling a transition to a DNF predicate one in linear time. A DNF predicate over a monotonic algebra is in-fact a disjunction of disjoint intervals. Then, to obtain a canonical representation, we order these intervals by order of their minimal elements. ■

5.3 Query Learning

Recall that the paradigm of *query learning* stipulates that the *learner* can interact with an *oracle* (*teacher*) by asking it several types of allowed queries. Angluin showed, on the negative side, that regular languages cannot be learned (in the exact model) from only *membership queries* (MQ) [Ang81] or only *equivalence queries* (EQ) [Ang90]. On the positive side, Angluin [Ang87a] showed that regular languages, represented as DFAs, can be learned using both MQ and EQ. The celebrated algorithm, termed $\mathbf{L}^*$, was extended to learning many other classes of languages and representations, e.g. [Sak90, BV96, AV10, BHKL09, AEF15, MP95, AF16], see the survey [Fis18] for more references.

In particular, an extension of $\mathbf{L}^*$, termed $\mathbf{MAT}^*$, to learn SFAs was provided in [AD18], which proved that SFAs over an algebra A can be efficiently learned using $\mathbf{MAT}^*$ if and only if the underlying algebra is efficiently learnable (see Definition 5.3.1), and the size of the disjunctions of k predicates does not grow exponentially in k.[5] From this it was concluded that SFAs over the following underlying algebras are efficiently learnable: Boolean algebras over finite domains, equality algebra, tree automata algebra, and SFAs algebra. Efficient learning of SFAs over a monotonic algebra using MQ and EQ was established in [CDYS17], which improved on the results of [MM14, MM17] by using a binary search instead of a helpful teacher.

The result of [AD18] provides means to establish new positive results on learning classes of SFAs using MQ and EQ, but it does not provide means for obtaining negative results for query learning of SFAs using MQ and EQ. We strengthen this result by providing a learnability result that is independent of the use of a specific learning algorithm. In particular, we show that efficient learnability of a Boolean algebra A using MQ and EQ is a necessary condition for the learnability of the class of SFAs over A, as we state in Theorem 5.1.

[5] As is the case, for instance, in the OBDD algebra [Bry86].

Definition 5.3.1. We say that an algebra $\mathcal{A}$ is *polynomially learnable using* MQs *and* EQs if there exists a query learning algorithm that given a predicate ψ over $\mathcal{A}$ can learn ψ using $poly(|\psi|)$ MQs and EQs.

Theorem 5.1. *If a class of SFAs $\mathbb{M}$ over a Boolean algebra $\mathcal{A}$ is polynomially learnable using* MQ *and* EQs*, then $\mathcal{A}$ is polynomially learnable using* MQ *and* EQs*.*

Proof. Assume that $\mathbb{M}$ is polynomially learnable using MQ and EQs, using an algorithm $\mathbf{Q}_{\mathbb{M}}$. We show that there exists a polynomial learning algorithm $\mathbf{Q}_{\mathcal{A}}$ for the algebra $\mathcal{A}$ using MQ and EQs. The algorithm $\mathbf{Q}_{\mathcal{A}}$ uses $\mathbf{Q}_{\mathbb{M}}$ as a subroutine, and behaves as a teacher for $\mathbf{Q}_{\mathbb{M}}$. Whenever $\mathbf{Q}_{\mathbb{M}}$ asks a $\mathbb{M}$-MQ on a word $\gamma_1 \dots \gamma_k$, if $k > 1$ then $\mathbf{Q}_{\mathcal{A}}$ answers "no". If $k=1$ then the $\mathbb{M}$-MQ is essentially an $\mathcal{A}$-MQ, thus $\mathbf{Q}_{\mathcal{A}}$ issues this query and passes the answer to $\mathbf{Q}_{\mathbb{M}}$. Whenever $\mathbf{Q}_{\mathbb{M}}$ asks a $\mathbb{M}$-EQ on SFA $\mathcal{M}$, if $\mathcal{M}$ is not a two state SFA with a single transition labeled by some predicate ψ, then $\mathbf{Q}_{\mathcal{A}}$ answers "no" to the $\mathbb{M}$-EQ and returns some word $w \in \mathcal{L}(\mathcal{M})$ s.t. $|w| > 1$ and w was not provided before, as a counterexample. Otherwise (if the SFA is of the above form) then $\mathbf{Q}_{\mathcal{A}}$ asks an $\mathcal{A}$-EQ on ψ. If the answer is "yes" then $\mathbf{Q}_{\mathcal{A}}$ terminates and returns ψ as the result of the learning algorithm; if the answer to the $\mathcal{A}$-EQ on ψ is "no", then the provided counterexample $\langle \gamma, b_\gamma \rangle$ is passed back to $\mathbf{Q}_{\mathbb{M}}$ together with the answer "no" to the $\mathbb{M}$-EQ.

The algorithm described above is sound, and the teacher is consistent, and thus, since $\mathbb{M}$ is learnable using $\mathbf{Q}_{\mathbb{M}}$, the algorithm eventually terminates. Note that $Q_{\mathbb{M}}$ learns exactly the language of a one-letter word which consists of the predicate to be learned by $Q_{\mathcal{A}}$. This is since it only answers MQs positively on one-letter words, and an EQ is answered positively only if the SFA consists of two states and one transition. Therefore the algorithm learns one predicate over the algebra, and thus if $\mathbb{M}$ over $\mathcal{A}$ is learnable, then so is $\mathcal{A}$. ■

From Theorem 5.1 we derive what we believe to be the first negative result on learning SFAs from MQ and EQ, as we show that SFAs over the propositional algebra are not polynomially learnable using MQ and EQ. Polynomiality is measured with respect to the parameters $\langle n, m, l \rangle$ representing the size of the SFA and the number k of atomic propositions. We achieve this by showing that no learning algorithm $\mathbf{A}$ for the Boolean algebra using MQ and EQ can do better than asking 2^k MQ/EQs, where k is the number of atomic propositions.[6]

Proposition 5.3.2. *Let $\mathbf{A}$ be a sound learning algorithm for the propositional algebra over $\mathbb{B}^k$. Then, there exists a target predicate ψ, for which $\mathbf{A}$ will be forced to ask at least $2^k - 1$ queries (either* MQ *or* EQ*).*

Proof. Since $\mathbf{A}$ is sound, at stage $i+1$ we have $S^+_{i+1} \supseteq S^+_i$ and $S^-_{i+1} \supseteq S^-_i$ and at least one inclusion is strict. Since the size of the concrete alphabet is 2^k, for every round $i < 2^k$, an adversarial teacher can answer both MQs and EQs negatively. In the case of EQ there must be an element in $\mathbb{B}^k \setminus (S^-_i \cup S^+_i)$ with which the provided automaton disagrees. The adversary will return one such element as a counterexample. This forces $\mathbf{A}$ to ask at least $2^k - 1$ queries. ■

[6] In [Nak00] Boolean formulas represented using OBDDs are claimed to be polynomially learnable with MQ and EQ. However, [Nak00] measures the size of an OBDD by its number of nodes, which can be exponential in the number of propositions.

Corollary 5.2. *SFAs over the propositional algebra with k propositions cannot be learned in $poly(k)$ time using* MQ *and* EQ.

The propositional algebra is a special case of the n-dimensional boxes algebra. Learning n-dimensional boxes was studied using MQ and EQ [GGM94, BGGM98, BK98], as well as in the PAC setting [BK00]. The algorithms presented in [GGM94, BGGM98, BK98, BK00] are mostly exponential in n. Alternatively, [GGM94, BGGM98] suggest algorithms that are exponential in the number of boxes in the union. In [BK98] a linear query learning algorithm for unions of disjoint boxes is presented. Since n-dimensional boxes subsume the propositional algebra, Corollary 5.2 implies the following.

Corollary 5.3. *The class of SFAs over n-dimensional boxes algebra cannot be learned in $poly(n)$ time using* MQ *and* EQ.

In case the minimal normalized SFA and the minimal neat SFA are isomorphic, then in essence, we only need to learn basic predicates. We call such automata *purely neat*. A motivation for purely neat SFAs can be found, for example, in [BJR06], where purely neat transitions augmented to finite alphabet letters are used to model communication protocols. The algorithm of [BJR06] is exponential in the number of propositions appearing in some conjunction. Here, we prove that this is a lower bound on the learnability of purely neat SFAs using MQs and EQs. We observe that the proof of Theorem 5.1 applies also for purely neat SFAs, thus we can strengthen the theorem as follows.

Theorem 5.4. *Any class of SFAs $\mathbb{M}$ over a Boolean algebra A, that subsumes all two-state SFAs with a single transition between them, labeled by an arbitrary predicate from A, is polynomially learnable using* MQ *and* EQ *only if the algebra A is polynomially learnable using* MQ *and* EQ.

Corollary 5.5. *Purely neat SFAs over the propositional algebra with k propositions cannot be learned in $poly(k)$ time using* MQ *and* EQ.

5.4 Identification in the Limit

The model of *identification in the limit using polynomial time and data* was proposed by Gold [Gol78] who showed that regular languages represented by DFAs are learnable in this model. We follow de la Higuera's more general definition [dlH97a].

This learning paradigm has a somewhat information theoretic perspective, in that it asks whether a class of languages can be learned from a polynomial set of correctly labeled words. A *sample* for a language L is a finite set $\mathcal{S}$ consisting of labeled examples, that is, pairs of the form $\langle w, b_w \rangle$ where w is a word and $b_w \in \{0, 1\}$ is its label, satisfying that $b_w = 1$ if and only if $w \in L$. Given two words w and w', we say that w and w' are *not equivalent* with respect to $\mathcal{S}$, denoted $w \not\sim_{\mathcal{S}} w'$, iff there exists z such that $\langle wz, b \rangle, \langle w'z, b' \rangle \in \mathcal{S}$ and $b \neq b'$. Otherwise we say that w and w' are *equivalent* wrt. $\mathcal{S}$, and denote $w \sim_{\mathcal{S}} w'$.

The sample is considered polynomial if it is of size polynomial in the smallest representation for L. In addition, it is required that given such a so-called *characteristic sample* $\mathcal{S}_L$ (formally defined in Definition 5.4.1), a learning algorithm can efficiently learn the target L (i.e. in time polynomial in $\mathcal{S}_L$), and more generally, that there exists an efficient learning algorithm, that outputs a representation agreeing with a given sample. Finally, it is required that the learning algorithm is not diverted from its correct conjecture when seeing additional information on top of the characteristic sample, in the sense that it correctly learns the target language L given any sample $\mathcal{S}'$ for L that is a superset of the characteristic sample $\mathcal{S}_L$. The formal definition follows.

Definition 5.4.1 (identification in the limit using polynomial time and data)**.** A class of languages $\mathbb{L}$ is said to be *identified in the limit using polynomial time and data* via representations in a class $\mathbb{C}$ if there exists a learning algorithm **Alg** such that the following two requirements are met.

1. Given a finite sample $\mathcal{S}$ of labeled examples, **Alg** returns in polynomial time a hypothesis $\mathcal{C} \in \mathbb{C}$ that agrees with $\mathcal{S}$.

2. For a language $L \in \mathbb{L}$, let $\mathcal{C} \in \mathbb{C}$ be the minimal representation for L. Then, there exists a sample $\mathcal{S}_L$, termed *a characteristic sample*, of size polynomial in the size of $\mathcal{C}$, such that **Alg** returns an hypothesis $\mathcal{C}'$ that is equivalent to $\mathcal{C}$, when run on any sample that subsumes $\mathcal{S}_L$.

Note that the first condition ensures polynomial time and the second polynomial data. If given arbitrary large finite sets, that do not have a representation in class $\mathbb{C}$, the algorithm is promised to return some representation that agrees with the sample; the algorithm is then allowed to fail on the second condition. De la Higuera's notion of characteristic sample is a core concept in grammatical inference, for various reasons. Firstly, it addresses shortcomings of several other attempts to formulate polynomial-time learning in the limit [Ang88, Pit89]. Secondly, this notion has inspired the design of popular algorithms for learning formal languages such as, for example, the RPNI algorithm [OG92]. Thirdly, it was shown to bear strong relations to a classical notion of machine teaching [GM96]; models of the latter kind are currently experiencing increased attention in the machine learning community [ZSZR18].

In this chapter, since we are interested in symbolic automata learning, we consider the representation class $\mathbb{C}$ to be a class of certain SFAs. The goal of an identification in the limit algorithm for SFAs is two-folded. The algorithm needs to construct the SFA in terms of structural representation, that is, its states and transitions; and to learn the predicates labeling the transitions. Identification in the limit of SFAs then requires first finding a polynomial size characteristic sample, over the *concrete* domain elements, and second, devising an SFA inference algorithm meeting the requirements of Definition 5.4.1. This problem appears to be hard, and has no solution in the general case. Indeed, we show that SFAs over the propositional algebra cannot be learned in this paradigm. On the other hand we show that the class of SFAs over the interval algebra can be learned in this paradigm.

5.4.1 Identification in the limit of DFAs using polynomial time and data

It was shown by [Gol78, OG92] that DFAs are identifiable in the limit using polynomial time and data. Our results rely on some properties of the algorithms for identification in the limit for DFA. Therefore, for completeness of the presentation, we first provide a complete description of the procedures showing that DFAs are identifiable in the limit using polynomial time and data, and that they satisfy the required properties.

Theorem 5.6 ([OG92])**.** *The class of DFAs is identifiable in the limit using polynomial time and data via procedures **CharDFA** and **InferDFA** satisfying that if $\mathcal{D}$ is a minimal and complete DFA and **CharDFA**$(\mathcal{D}) = \mathcal{S}_\mathcal{D}$ then the following holds:*

1. *$\mathcal{S}_\mathcal{D}$ contains a prefix-closed set A of words. The words in A are termed* access words.[7] *Moreover, A can be chosen to contain only lex-access words, which are the smallest access words in the lexicographic order.*
2. *For every $u_1, u_2 \in A$ it holds that $u_1 \not\sim_{\mathcal{S}_\mathcal{D}} u_2$.*
3. *For every $u, v \in A$ and $\sigma \in \Sigma$, if $\delta(q_0, u\sigma) \neq \delta(q_0, v)$ then $u\sigma \not\sim_{\mathcal{S}_\mathcal{D}} v$.*

We later use these properties in order to construct a DFA out of a given set $\mathcal{S}$. Therefore, we need to relate between the words of the sample and the transition relation of the automaton. This is the motivation for properties (2) and (3). As for the first property, given a set $\mathcal{S} \supseteq \mathcal{S}_\mathcal{D}$, property (1) helps us recognize the words in $\mathcal{S}$ that are also in $\mathcal{S}_\mathcal{D}$. For SFAs, this property allows us to understand which of the alphabet letters are relevant in order to identify the predicates labeling transitions.

To prove Theorem 5.6 we first show that given a DFA $\mathcal{D} = \langle \Sigma, Q, q_0, F, \delta \rangle$, we can construct a polynomial-sized sample of words $\mathcal{S}_\mathcal{D}$ that agrees with $\mathcal{D}$ and satisfies the required properties. We show an algorithm that (i) can infer in polynomial time from a given sample $\mathcal{S}$ a DFA that agrees with $\mathcal{S}$, and (ii) if it is given the set $\mathcal{S}_\mathcal{D}$, or any set $\mathcal{S} \supseteq \mathcal{S}_\mathcal{D}$ that agrees with $\mathcal{D}$, then it infers a DFA that is equivalent to $\mathcal{D}$. All this together proves Theorem 5.6 (and explains why we can refer to $\mathcal{S}_\mathcal{D}$ as the *characteristic sample*).

Constructing a characteristic set

Given a minimal and complete DFA, the algorithm **CharDFA** works as follows. It first creates a prefix-closed set of access words to states. This can be done by considering the graph of the automaton and running an algorithm for finding a spanning tree T from the initial state. Choosing one of the letters on each edge, the access word for a state is obtained by concatenating the labels on the unique path of T that reaches that state. If we wish to work with lex-access words, we can use a depth-first search algorithm that spans branches according to the order of letters in Σ, starting from the smallest. The labels on the paths of the spanning tree constructed this way will form the set of lex-access words. Let S be the set of access words (or lex-access words).

Next, the algorithm turns to find a distinguishing word $v_{i,j}$ for every pair of state $s_i, s_j \in S$ (where $s_i \neq s_j$). Lemma 5.4.2 below states that any pair of states of the minimal DFA

[7] As we later show, the words of A intuitively correspond to the states of $\mathcal{D}$.

has a distinguishing word of size quadratic in the size of the DFA. Let E be the set of all such distinguishing words. We may assume $\epsilon \in E$.[8] Then the algorithm returns the set $\mathcal{S}_\mathcal{D} = \{\langle w, \mathcal{D}(w)\rangle \ : \ w \in (S \cdot E) \cup (S \cdot \Sigma \cdot E)\}$ where $\mathcal{D}(w)$ is the label $\mathcal{D}$ gives w (i.e. 1 if $\mathcal{D}$ accepts w, and 0 otherwise).

It is easy to see that $\mathcal{S}_\mathcal{D}$ satisfies the properties of Theorem 5.6.

Lemma 5.4.2. *Let $\mathcal{D} = \langle \Sigma, Q, q_0, F, \delta\rangle$ be a minimal DFA, and let $q_1, q_2 \in Q$ s.t. $q_1 \neq q_2$. Then there exists a polynomial time procedure that returns a word v of size at most $|Q|^2$ such that $\delta(q_1, v)$ is accepting iff $\delta(q_2, v)$ is rejecting.*

Proof. We can apply the product construction to $\mathcal{D}_i = \langle \Sigma, Q, q_i, F, \delta\rangle$ for $i \in \{1, 2\}$ and search for a path from the initial state (q_1, q_2) to a state in $F \times (Q \setminus F)$ or $(Q \setminus F) \times F$ to find a word that leads to an accepting state when read from q_1 and a rejecting state when read from q_2 or vice versa. Since a shortest simple path in a graph is bounded by the number of nodes, the shortest such word is of length at most $|Q|^2$. The shortest path can be found using breadth-first search algorithms that run in time linear in the number of vertices and edges, thus polynomial in the size of the DFA. ■

Since computing a spanning tree (in particular via DFS) and finding shortest paths can be done in polynomial time, this shows that for DFAs we can construct the characteristic set in polynomial time. That is, while Definition 5.4.1 only requires that the characteristic set be of polynomial size, for DFAs we can show that it can also be computed in polynomial time.

Inferring a DFA

Next, we describe algorithm **InferDFA** that given a set of sample words $\mathcal{S}$, infers from it in polynomial time a DFA that agrees with $\mathcal{S}$. Moreover, if $\mathcal{S} \supseteq \mathcal{S}_\mathcal{D}$ where $\mathcal{S}_\mathcal{D}$ is a characteristic sample set of a DFA $\mathcal{D}$, and $\mathcal{S}$ agrees with $\mathcal{L}(\mathcal{D})$, then **InferDFA** returns an equivalent DFA to $\mathcal{D}$.

Let W be the set of words in the given sample $\mathcal{S}$ (without their labels). Let R be the set of prefixes of W, and C be the set of suffixes of W. Note that $\epsilon \in R$ and $\epsilon \in C$. Let $r_0, r_1, \ldots$ be some enumeration of R and $c_0, c_1, \ldots$ some enumeration of C where $r_0 = c_0 = \epsilon$. In the sequel, we often use i_w for the index of w in R. The algorithm builds a matrix M of size $|R| \times |C|$ whose entries take values in $\{0, 1, ?\}$. The algorithm sets the value of entry (i, j) as follows. If $r_i c_j$ is not in W, it is set to $?$. Otherwise it is set to 1 iff the word $r_i c_j$ is labeled 1 in $\mathcal{S}$. We get that $r_i \sim_\mathcal{S} r_j$ iff for every k such that both $M(i, k)$ and $M(j, k)$ are different than $?$, we have that $M(i, k) = M(j, k)$.

The algorithm sets $R_0 = \{\epsilon\}$. Once R_i is constructed, the algorithm tries to establish whether $r\sigma$ for $r \in R_i$ and $\sigma \in \Sigma$ is distinguished from all words in R_i. It does so by considering all other words $r' \in R_i$ and checking whether $r\sigma \sim_\mathcal{S} r'$. If $r\sigma$ is found to be

[8] Unless $\mathcal{D}$ accepts all words or rejects all words, it has at an accepting state and a rejecting state, and ϵ is the shortest word distinguishing these states. If all states of $\mathcal{D}$ are accepting (or all rejecting) the algorithm returns $\mathcal{S}_\mathcal{D} = \{\langle \epsilon, 1\rangle\}$ (resp. $\mathcal{S}_\mathcal{D} = \{\langle \epsilon, 0\rangle\}$).

distinct from all words in R_i, then R_{i+1} is set to $R_i \cup \{r\sigma\}$. The algorithm proceeds until no new words are distinguished. Let $R = R_k$ where k is the iteration of convergence.

If not all words in R are in W (that is $M(i,0) =?$ for some $r_i \in R$), the algorithm returns the prefix-tree automaton.[9] Otherwise, the states of the constructed DFA are set to be the words in R. The initial state is ϵ and a state r_i is classified as accepting iff $M(i,0) = 1$ (recall that the entry $M(i,0)$ stands for the value of $r_i \cdot \epsilon$ in $\mathcal{S}$). To determine the transitions, for every $r \in R$ and $\sigma \in \Sigma$, recall that there exists at least one state $r' \in R$ that cannot be distinguished from $r\sigma$. The algorithm then adds a transition from r on σ to r'.

Proposition 5.4.3.

1. *Algorithm **InferDFA** runs in polynomial time and returns a DFA that agrees with the given sample $\mathcal{S}$.*

2. *Let $\mathcal{S}_\mathcal{D}$ be the sample constructed for a DFA $\mathcal{D}$ by algorithm **CharDFA**, and let $\mathcal{S} \supseteq \mathcal{S}_\mathcal{D}$. Then algorithm **InferDFA** returns a DFA that recognizes the same language as $\mathcal{D}$.*

Proof.

1. The number of prefixes (or suffixes) of a set of words is bounded by the size of the longest word times the size of the set. Thus M is of polynomial size, and so is its construction. The number of iterations required for converging the R_i sets is bounded by $|W|$. The prefix-tree automaton can be computed in polynomial time. Determining acceptance is polynomial in $|R|$, and determining the transitions is polynomial in $|R| \times |\Sigma|$. Therefore the overall running time of the algorithm is polynomial.

   Clearly, if **InferDFA** returns the prefix-tree automaton then it agrees with the given sample $\mathcal{S}$. We claim that it agrees with the given sample also in the second case. We show, by induction on the length of the word, that for every $w \in W$, if w reaches state r of the constructed DFA, then $w \sim_\mathcal{S} r$. Then, since w is in the sample, and r is in the sample (otherwise the algorithm would return the prefix-tree automaton), it follows that $M(i_w, 0) = M(i_r, 0)$, hence the DFA agrees with the sample on w.

   For the base case, we have that $|w| = 0$ then the DFA accepts if r_0 is accepting, which holds iff $M(0,0) = 1$. Indeed, this entry is filled with the label of ϵ in $\mathcal{S}$. Consider now $w = v\sigma$ for some $v \in \Sigma^*$ and $\sigma \in \Sigma$. Assume that the DFA reaches the state s_ℓ on reading v and the state s_m on reading w. By the induction hypothesis, we know that $r_\ell \sim_\mathcal{S} v$. From the construction of the algorithm it follows that $r_\ell\sigma \sim_\mathcal{S} s_m$ as otherwise, reading σ from r_ℓ would lead to a different state. If $r_m \not\sim_\mathcal{S} w$ then exists a suffix $c_i \in C$ s.t. $M(m,i) \neq M(i_w, i)$. But then σc_i is also in C. Then $M(\ell, j) \neq M(i_v, j)$, contradicting that $r_\ell \sim_\mathcal{S} v$.

[9] The prefix-tree automaton is the automaton obtained by placing all words in a tree data structure (sharing common prefixes) and labeling a state accepting iff the unique word reaching that state is in the sample and is labeled 1.

2. Next, we show that if $\mathcal{S}$ subsumes $\mathcal{S}_\mathcal{D}$ then the returned DFA agrees with $\mathcal{D}$. Let $w_1, \ldots, w_n$ be the set of accessible words chosen by **CharDFA**. Since $\mathcal{S}$ consists of a distinguished word for every pair of access words w_i and w_j of $\mathcal{D}$, algorithm **InferDFA** will determine $w_i \not\sim_\mathcal{S} w_j$ and R will consist of at least n states. It may not consist of more states, since the sample has to agree with the language of $\mathcal{D}$ and every word agrees with some state of $\mathcal{D}$ on all possible suffixes, thus cannot be determined distinct by **InferDFA**. Since $S \cdot \Sigma \cdot E$ was placed in $\mathcal{S}$, for every distinguished state w and every $\sigma \in \Sigma$ the alg. **InferDFA** can determine the transition from w upon reading σ. Since $S \cdot \epsilon$ is placed in $\mathcal{S}$, alg. **InferDFA** can correctly label the set of accepting states. Therefore, the obtained DFA is isomorphic to the original DFA. ■

We can thus conclude that the class $\mathbb{C}_\mathrm{D}$ of DFAs is identifiable in the limit using polynomial time and data. Furthermore, $\mathbb{C}_\mathrm{D}$ satisfies the properties of Theorem 5.6.

5.4.2 Conditions for identification in the limit of SFAs

We now turn to discuss our results regarding identification in the limit of symbolic automata. We provide a necessary condition, and a sufficient condition for deciding whether a class of SFAs is identifiable in the limit using polynomial time and data.

A necessary condition for identification in the limit of SFAs

We make use of the following definitions. A sequence $\langle \Gamma_1, \ldots, \Gamma_m \rangle$ consisting of sets of concrete letters $\Gamma_i \subseteq \mathbb{D}$ is referred to as a *concrete partition* of $\mathbb{D}$ if the sets are pairwise disjoint (namely $\Gamma_i \cap \Gamma_j = \emptyset$ for every $i \neq j$). Similarly, a sequence of predicates $\langle \psi_1, \ldots, \psi_m \rangle$ over a Boolean algebra A is referred to as a *predicate partition* if $[\![\psi_i]\!] \cap [\![\psi_j]\!] = \emptyset$ for every $i \neq j$.

Definition 5.4.4.

- A function f_g from a concrete partition to a predicate partition is termed *generalizing* if $f_g(\langle \Gamma_1, \ldots, \Gamma_m \rangle) = \langle \psi_1, \ldots, \psi_k \rangle$ implies $k = m$ and $[\![\psi_i]\!] \supseteq \Gamma_i$ for all $1 \leq i \leq m$.
- A function f_c from a predicate partition to a concrete partition is termed *concretizing* if $f_c(\langle \psi_1, \ldots, \psi_m \rangle) = \langle \Gamma_1, \ldots, \Gamma_k \rangle$ implies $k = m$ and $\Gamma_i \subseteq [\![\psi_i]\!]$ for all $1 \leq i \leq m$.

We say that f_g (resp. f_c) is *efficient* if it can be computed in polynomial time. Note that if f_c is efficient then the sets Γ_i in the constructed concrete partition are of polynomial size.

Theorem 5.7. *A necessary condition for a class of SFAs $\mathbb{M}$ to be identified in the limit using polynomial time and data is that there exist efficient concretizing and generalizing functions, Concretize$_\mathbb{M}$ and Generalize$_\mathbb{M}$, satisfying that if Concretize$_\mathbb{M}(\langle \psi_1, \ldots, \psi_m \rangle) = \langle \Gamma_1, \ldots, \Gamma_m \rangle$ and Generalize$_\mathbb{M}(\langle \Gamma'_1, \ldots, \Gamma'_m \rangle) = \langle \varphi_1, \ldots, \varphi_m \rangle$ where $\Gamma_i \subseteq \Gamma'_i$ for every $1 \leq i \leq m$, then $[\![\varphi_i]\!] = [\![\psi_i]\!]$ for every $1 \leq i \leq m$.*

Proof. Assume one of the requirements fails. That is, either there exists no efficient concretizing or generalizing function, or the generalization fails if the concrete partition is enhanced with more words. Let $\mathcal{M} \in \mathbb{M}$ be an SFA whose initial state, q_ι, has outgoing transitions labeled by $\psi_1, \ldots, \psi_m$. Then, from the definition of identification in the limit, we cannot learn the outgoing transitions of q_ι. Thus we cannot learn $\mathcal{M}$, and the class $\mathbb{M}$ cannot be identified in the limit using polynomial time and data. ■

Example 5.4.5. Consider the class $\mathbb{M}_{\mathsf{A}_\mathbb{N}}$ of SFAs over the algebra $\mathsf{A}_\mathbb{N}$ of Example 5.2.1, and consider the functions

$$\mathsf{Concretize}_{\mathbb{M}_{\mathsf{A}_\mathbb{N}}}(\langle [d_1 = 0, d_1'), [d_2, d_2'), \ldots, [d_m, d_m' = \infty) \rangle) = \langle \{d_1\}, \ldots, \{d_m\} \rangle$$

$$\mathsf{Generalize}_{\mathbb{M}_I}(\langle \Gamma_1, \ldots, \Gamma_n \rangle) = \langle [\min \Gamma_1, \min \Gamma_2), \ldots, [\min \Gamma_n, \infty) \rangle$$

Then, $\mathsf{Concretize}_{\mathbb{M}_{\mathsf{A}_\mathbb{N}}}$ and $\mathsf{Generalize}_{\mathbb{M}_{\mathsf{A}_\mathbb{N}}}$ are efficient and satisfy the conditions of Theorem 5.7, since every element added to a set Γ_i has to be inside the corresponding interval, and thus cannot affect the result of generalization.

We say that a Boolean algebra A with predicates $\mathbb{P}$ over domain $\mathbb{D}$ is *efficiently identifiable* if there exist polynomially computable functions $f_c : \mathbb{P} \to \mathbb{D}$ and $f_g : \mathbb{D} \to \mathbb{P}$ such that $f_c(\psi) \subseteq [\![\psi]\!]$ and $[\![f_g(\Gamma)]\!] \supseteq \Gamma$ for every $\psi \in \mathbb{P}$ and $\Gamma \subseteq \mathbb{D}$, and moreover if $f_c(\psi) = \Gamma$, $\Gamma' \supseteq \Gamma$ and $f_g(\Gamma') = \psi'$, then $[\![\psi']\!] = [\![\psi]\!]$. Using this terminology we can state the following corollary.

Corollary 5.8. *Efficient identifiability of the underlying algebra is a necessary condition for identification in the limit using polynomial time and data of the respective class of SFAs.*

This corollary holds since the condition of Theorem 5.7 is violated when the underlying algebra is not efficiently identifiable (when considering partitions of size one).

A sufficient condition for identification in the limit of SFAs

The result regarding the sufficient condition on identification in the limit of SFAs using polynomial time and data relies on the respective result for DFAs [Gol78, OG92], as stated in Theorem 5.9 that follows from Section 5.4.1.

Theorem 5.9 ([Gol78, OG92])**.** *There exist procedures **CharDFA** and **InferDFA** witnessing that the class of DFAs is identified in the limit using polynomial time and data.*

To state the sufficient condition we need the following definitions:

Definition 5.4.6.

- For an SFA $\mathcal{M}$, the alphabet Σ_{conc}, as defined in line 4 of Alg. 5.1, is the set of all concretizations of predicates labeling transitions in $\mathcal{M}$.

Algorithm 5.1 CharSFA – Build a characteristic sample for a given SFA

Input an SFA $\mathcal{M}$, algorithm **CharDFA**, function $\mathsf{Concretize}_{\mathbb{M}}$
Output a concrete characteristic set $\mathcal{S}_{\mathcal{M}} \subseteq \mathbb{D}^* \times \{0,1\}$

1: **function** CHARSFA($\mathcal{M} = \langle A, Q, q_\iota, F, \delta \rangle$)
2: **for all** $q \in Q$ **do** let $\langle \psi_1^q, \cdots, \psi_m^q \rangle$ be the predicates labeling
3: outgoing transitions from q
4: $\Sigma_{\mathsf{conc}} := \bigcup_{q \in Q} \mathsf{Concretize}_{\mathbb{M}}(\langle \psi_1^q, \cdots, \psi_m^q \rangle)$
5: $\delta_{\mathcal{D}} := \emptyset$
6: **for all** $q, q' \in Q, d \in \Sigma_{\mathsf{conc}}$ **do**
7: **if** $\langle q, \psi, q' \rangle \in \delta$ and $d \in [\![\psi]\!]$ **then**
8: $\delta_{\mathcal{D}} := \delta_{\mathcal{D}} \cup \langle q, d, q' \rangle$
9: $\mathcal{D}_{\mathcal{M}} := \langle \Sigma_{\mathsf{conc}}, Q, q_\iota, F, \delta_{\mathcal{D}} \rangle$
10: $\mathcal{S}_{\mathcal{M}} = \mathbf{CharDFA}(\mathcal{D}_{\mathcal{M}})$
11: **return** $\mathcal{S}_{\mathcal{M}}$

- A function f_d that takes as input a sample set $\mathcal{S}$ and outputs a sample set $\mathcal{S}' \subseteq \mathcal{S}$, is termed *decontaminating* if given a characteristic sample $\mathcal{S}_{\mathcal{M}}$ over the alphabet Σ_{conc}, if $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$ is over an alphabet $\Sigma \supseteq \Sigma_{\mathsf{conc}}$, then $f_d(\mathcal{S}) = \mathcal{S}'$ for some $\mathcal{S}'$ over the alphabet Σ_{conc} satisfying that $\mathcal{S}_{\mathcal{M}} \subseteq \mathcal{S}' \subseteq \mathcal{S}$.

Theorem 5.10. *For a class $\mathbb{M}$ of SFAs, if there exist functions $\mathsf{Concretize}_{\mathbb{M}}$ and $\mathsf{Generalize}_{\mathbb{M}}$ satisfying the criteria of Theorem 5.7, and in addition there exists an efficient decontaminating function $\mathsf{Decontaminate}_{\mathbb{M}}$, then the class $\mathbb{M}$ is identified in the limit using polynomial time and data.*

Given functions $\mathsf{Concretize}_{\mathbb{M}}$, $\mathsf{Decontaminate}_{\mathbb{M}}$ and $\mathsf{Generalize}_{\mathbb{M}}$ for a class $\mathbb{M}$ of SFAs meeting the criteria of Theorem 5.10, we show that the class $\mathbb{M}$ can be identified in the limit using polynomial time and data.

The procedures we provide, **CharSFA** and **InferSFA**, described in Alg. 5.1 and Alg. 5.2, respectively, use the algorithms **CharDFA** and **InferDFA**, respectively, as well as the methods $\mathsf{Concretize}_{\mathbb{M}}$, $\mathsf{Generalize}_{\mathbb{M}}$ and $\mathsf{Decontaminate}_{\mathbb{M}}$. We briefly describe these two algorithms, and then turn to prove Theorem. 5.10.

The algorithm **CharSFA** (Alg. 5.1), receives an SFA $\mathcal{M} \in \mathbb{M}$, and returns a characteristic sample for it. It does so by constructing a DFA $\mathcal{D}_{\mathcal{M}}$ over the alphabet Σ_{conc} with the same structure as $\mathcal{M}$ (i.e. same states and edges), where a transition in $\mathcal{D}_{\mathcal{M}}$ is labeled by a concrete letter d iff d satisfies the predicate on the corresponding transition in $\mathcal{M}$ (Alg. 5.1 lines 6-8). Recall that since $\mathsf{Concretize}_{\mathbb{M}}$ is an efficient concretizing function, its output is finite, and thus Σ_{conc} is finite as well. Hence $\mathcal{D}_{\mathcal{M}}$ is indeed a DFA (over a finite and concrete alphabet). Then, Alg. 5.1 generates and returns the sample $\mathcal{S}_{\mathcal{M}}$ using the algorithm **CharDFA** applied on the DFA $\mathcal{D}_{\mathcal{M}}$ (line 10).

Algorithm **InferSFA** (Alg. 5.2), given a sample set $\mathcal{S}$, returns an SFA $\mathcal{M}_{\mathcal{S}}$ for it. First, **InferSFA** finds a subset $\mathcal{S}' \subseteq \mathcal{S}$ over the alphabet Σ_{conc} by calling $\mathsf{Decontaminate}_{\mathbb{M}}(\mathcal{S})$ (line 2). Then it uses $\mathcal{S}'$ to construct a DFA by applying the inference algorithm **InferDFA** on $\mathcal{S}'$

Algorithm 5.2 InferSFA – Infer an SFA from a sample

Input sample $\mathcal{S} \subseteq \mathbb{D}^* \times \{0,1\}$, algorithm **InferDFA**,
functions $\mathsf{Generalize}_{\mathbb{M}}$, $\mathsf{Decontaminate}_{\mathbb{M}}$
Output An SFA that agrees with the given sample $\mathcal{S}$

1: **function** INFERSFA($\mathcal{S}$)
2: $\quad\mathcal{S}' := \mathsf{Decontaminate}_{\mathbb{M}}(\mathcal{S})$
3: $\quad\langle \Sigma, Q, q_\iota, F, \delta_{\mathcal{D}} \rangle := \textbf{InferDFA}(\mathcal{S}')$
4: $\quad\delta_{\mathcal{M}} := \emptyset$
5: $\quad$**for all** $q \in Q$ **do**
6: $\quad\quad$**for all** $q_i \in Q$ **do** $\Gamma_i := \{\gamma : \langle q, \gamma, q_i \rangle \in \delta_{\mathcal{D}}\}$
7: $\quad\quad\langle \psi_1, \ldots, \psi_n \rangle := \mathsf{Generalize}_{\mathbb{M}}(\langle \Gamma_1, \ldots, \Gamma_n \rangle)$
8: $\quad\quad$**for all** $q_i \in Q$ **do** $\delta_{\mathcal{M}} := \delta_{\mathcal{M}} \cup \langle q, \psi_i, q_i \rangle$
9: $\quad\mathcal{M}_{\mathcal{S}} := \langle \Sigma, Q, q_\iota, F, \delta_{\mathcal{M}} \rangle$
10: $\quad$**if** $\exists w$ in $\mathcal{S}$ that does not agree with $\mathcal{M}_{\mathcal{S}}$ **then**
11: $\quad\quad$**return** the prefix tree automaton of $\mathcal{S}$
12: $\quad$**else**
13: $\quad\quad$**return** $\mathcal{M}_{\mathcal{S}}$

(line 3). From this DFA, **InferSFA** constructs an SFA, $\mathcal{M}_{\mathcal{S}}$, by applying $\mathsf{Generalize}_{\mathbb{M}}$ on each state q to get the symbolic transitions from the set of concrete transitions exiting q (lines 5-8).

In Section 5.4.3 we provide methods $\mathsf{Concretize}_{\mathbb{M}}$, $\mathsf{Decontaminate}_{\mathbb{M}}$ and $\mathsf{Generalize}_{\mathbb{M}}$ for SFAs over monotonic algebras, deriving their identification in the limit result. We now prove Theorem 5.10.

Proof of Theorem 5.10. To prove sufficiency, we show that if there exist functions $\mathsf{Concretize}_{\mathbb{M}}$, $\mathsf{Decontaminate}_{\mathbb{M}}$ and $\mathsf{Generalize}_{\mathbb{M}}$ meeting the required criteria for a class of SFAs $\mathbb{M}$, then $\mathbb{M}$ is identified in the limit using polynomial time and data. To this end, we show that the two conditions of Definition 5.4.1 are met.

For the first condition, given that **InferDFA**, $\mathsf{Decontaminate}_{\mathbb{M}}$ and $\mathsf{Generalize}_{\mathbb{M}}$ run in polynomial time, and that the prefix-tree automaton can be constructed in polynomial time, it is clear that so does **InferSFA**. In addition, the test of line 10 in Alg. 5.2 ensures the output agrees with the sample. If the SFA $\mathcal{M}_{\mathcal{S}}$ does not agree with the sample $\mathcal{S}$, then the prefix tree automaton of $\mathcal{S}$ is returned.

For the second condition, note that the sample generated by **CharSFA** is polynomial in the size of $\mathcal{D}_{\mathcal{M}}$, from the correctness of **CharDFA**. In addition, since $\mathsf{Concretize}_{\mathbb{M}}$ is efficient, $\mathcal{D}_{\mathcal{M}}$ is polynomial in the size of $\mathcal{M}$, and thus $\mathcal{S}_{\mathcal{M}}$ generated by **CharSFA** is polynomial in $\mathcal{M}$ as well. It is left to show that given that $\mathcal{S}_{\mathcal{M}}$ is the concrete sample produced by **CharSFA** when running on an SFA $\mathcal{M}$, then when **InferSFA** runs on any sample $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$ such that $\mathcal{S}$ agrees with $\mathcal{M}$, **InferSFA** returns an SFA for $\mathcal{L}(\mathcal{M})$. Since $\mathsf{Decontaminate}_{\mathbb{M}}$ is a decontaminating function, and $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$, it holds that the set $\mathcal{S}' = \mathsf{Decontaminate}_{\mathbb{M}}(\mathcal{S})$ is such that $\mathcal{S}' \supseteq \mathcal{S}_{\mathcal{M}}$ and is only over the alphabet Σ_{conc}, which is exactly the alphabet of the DFA $\mathcal{D}_{\mathcal{M}}$ generated in Alg. 5.1 (line 9).

From the correctness of **InferDFA**, given $\mathcal{S}' \supseteq \mathcal{S}_{\mathcal{M}}$, the inferred DFA $\mathcal{D}$ (Alg. 5.2 line 3)

is equivalent to $\mathcal{D}_{\mathcal{M}}$ constructed in Alg.5.1. Since $\mathcal{D}_{\mathcal{M}}$ is complete, for a state q of $\mathcal{D}$, the concrete partition $\langle \Gamma_1, \ldots, \Gamma_n \rangle$ generated in Alg. 5.2 line 6, covers Σ_{conc} and subsumes the output of $\mathsf{Concretize}_{\mathbb{M}}$ on the outgoing symbolic transitions from the state equivalent to q in $\mathcal{D}_{\mathcal{M}}$. Thus, since $\mathsf{Generalize}_{\mathbb{M}}$ and $\mathsf{Concretize}_{\mathbb{M}}$ satisfy the criteria of Theorem 5.7, it holds that the generated predicates agree with the original predicates. In addition, since $\mathcal{S}$, and therefore $\mathcal{S}'$, agrees with $\mathcal{M}$, the test of line 10 fails and the returned SFA is equivalent to $\mathcal{M}$.■

5.4.3 Identification in the limit for certain classes of SFAs

We first discuss monotonic algebras, presenting the following positive result. Then we show that this does not hold for non-monotonic algebras, as stated in Proposition 5.4.12 regarding the propositional algebra.

Theorem 5.11. *Let $\mathbb{M}_{A_m}$ be the set of SFAs over a monotonic Boolean algebra A_m. Then $\mathbb{M}_{A_m}$ is identified in the limit using polynomial time and data.*

In order to prove Theorem 5.11, we show that the sufficient condition holds for the case of monotonic algebras. Example 5.4.11 demonstrates how to apply **CharSFA** and **InferSFA** in order to learn an SFA over the algebra $A_{\mathbb{N}}$. In order to prove the sufficient condition of Theorem 5.10, we first show that the necessary condition of Theorem 5.7 holds for monotonic algebras.

Proposition 5.4.7. *There exist functions $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ and $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ for the class $\mathbb{M}_{A_m}$ of SFAs over a monotonic Boolean algebra, satisfying the criteria of Theorem 5.7.*

Proof. Let $\mathbb{D}$ be the domain of A_m. Recall that in Section 5.2.1, a monotonic algebra is defined over its domain $\mathbb{D}$ together with the two elements d_{inf} and d_{sup}. We provide the functions $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ and $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ for $\mathbb{M}_{A_m}$, and prove that the criteria of Theorem 5.7 hold for them. For ease of presentation, we consider the class of neat and complete SFAs. The result holds for general SFAs as well since, by Lemma 5.2.4 and Lemma 5.2.9, the transformation from a general SFA to a neat and complete SFA over a monotonic algebra is polynomial. The definitions of $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ and $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ are generalizations of the functions $\mathsf{Concretize}_{\mathbb{M}_{A_{\mathbb{N}}}}$ and $\mathsf{Generalize}_{\mathbb{M}_{A_{\mathbb{N}}}}$ given in Example 5.4.5.

$\mathsf{Concretize}_{\mathbb{M}_{A_m}}(\langle \psi_1, \ldots \psi_m \rangle) = \langle \Gamma_1, \ldots, \Gamma_m \rangle$ where we set $\Gamma_i = \{infimum\{d \in \mathbb{D} : d \in [\![\psi_i]\!]\}\}$ for $1 \leq i \leq m$. Since A_m is monotonic, Γ_i is well defined and contains a single element, thus $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ is an efficient concretizing function.

$\mathsf{Generalize}_{\mathbb{M}_{A_m}}(\langle \Gamma_1, \ldots, \Gamma_m \rangle) = \langle \psi_1, \ldots, \psi_m \rangle$, where ψ_i is defined as follows. Let $\Gamma = \bigcup_{1 \leq i \leq m} \Gamma_i$. First, for all $1 \leq i \leq m$ we set $\psi_i = \bot$. Then, we iteratively look for the minimal element $\gamma \in \Gamma$. Let i be such that $\gamma \in \Gamma_i$, and let γ' be the minimal element in Γ satisfying $\gamma' \notin \Gamma_i$. We then set $\psi_i = \psi_i \vee [\gamma, \gamma')$, and remove all elements $\gamma \leq \gamma'' < \gamma'$ from Γ. We repeat the process until for the found $\gamma \in \Gamma_j$, there is no $\gamma' > \gamma$ such that $\gamma' \notin \Gamma_j$. In this case, we define $\psi_j = \psi_j \vee [\gamma, d_{sup})$. In order to construct a neat SFA we can later split each ψ_i that contains m disjunctions, into m transitions. Then, $\Gamma_i \subseteq [\![\psi_i]\!]$ and the predicates are disjoint, thus $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ is an efficient generalizing function.

Now, let $\langle \Gamma_1, \ldots, \Gamma_m \rangle$ be the concrete partition obtained from $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ when applied on the predicate partition $\langle \psi_1, \ldots, \psi_m \rangle$. Assume further that the concrete partition $\langle \Gamma'_1, \ldots, \Gamma'_m \rangle$ satisfies that $\Gamma_i \subseteq \Gamma'_i \subseteq [\![\psi_i]\!]$ for $1 \leq i \leq m$. In particular, $\min(\Gamma'_i) = \min(\Gamma_i)$, since Γ_i contains the minimal elements in $[\![\psi_i]\!]$, and $\Gamma_i \subseteq \Gamma'_i \subseteq [\![\psi_i]\!]$. Therefore, applying $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ will result in the same interval, satisfying the criterion of Theorem 5.7. ■

Example 5.4.8. Let $\Gamma_1 = \{0, 100, 400, 500\}$ and $\Gamma_2 = \{150, 200\}$ over the algebra $A_{\mathbb{N}}$ with domain $\mathbb{N} \cup \{\infty\}$. Then, $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ sets $\Gamma = \{0, 100, 150, 200, 400, 500\}$, and finds the minimal element $0 \in \Gamma_1$. It then looks for the minimal element $\gamma \in \Gamma$ such that $\gamma \notin \Gamma_1$, and finds $150 \in \Gamma_2$. Therefore $\psi_1 = [0, 150)$ and Γ is updated to be $\Gamma = \{150, 200, 400, 500\}$. Next, it finds the minimal element $150 \in \Gamma_2$, and the minimal element that is not in Γ_2 is 400. Then, ψ_2 is set to be $\psi_2 = [150, 400)$ and $\Gamma = \{400, 500\}$. Now, $\psi_1 = [0, 150) \vee [400, \infty)$ since $400 \in \Gamma_1$ and there is no greater element that is not in Γ_1.

Procedure $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$: **Input:** set $\mathcal{S}$ over alphabet Σ
Output: set $\mathcal{S}'$ over alphabet Σ'

1. Let $A_w = \{\epsilon\}$. The set A_w stands for the lex-access words discovered so far.
2. Set $\Sigma' = \{d_{inf}\}$ and set $max = d_{inf}$.[a]
3. Traverse A_w according to the lexicographic order, and for every $u \in A_w$, do:
   (a) Add σ to Σ' iff $\sigma > max$ and $u \cdot \sigma \not\sim_{\mathcal{S}} u \cdot max$ and σ is the minimal in Σ satisfying this property.
   (b) If σ was added to Σ', add $u\sigma$ to A_w iff there is no $u' \in A_w$ such that $u\sigma \sim_{\mathcal{S}} u'$.[b]
   (c) Set $max = \sigma$ and repeat item 3 until no such σ is found.
4. Set $max = d_{inf}$ and repeat item 3 until Σ' is remained unchanged.
5. Return $\mathcal{S}' = \mathcal{S} \cap \Sigma'^*$.

[a] Recall that for a monotonic algebra, as defined in Section 5.2.1, there exists an element d_{inf} such that for every $d \in \mathbb{D}$ it holds that $d_{inf} \leq d$.

[b] Note that we add σ to Σ' in item $3a$ even if there is a u' such that $u\sigma \sim_{\mathcal{S}} u'$, as long as u' is not of the form of $u \cdot max$. Intuitively, this is since we are looking for letters labeling outgoing transitions from the state that is represented by u.

Algorithm 5.3: $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$

To prove that the sufficient condition holds, we build upon some properties of procedures **CharDFA** and **InferDFA** as stated in Theorem 5.6 in Section 5.4.1.

Proposition 5.4.9. *The sufficient condition of Theorem 5.10 holds for the class $\mathbb{M}_{A_m}$ of SFAs over a monotonic Boolean algebra.*

Proof. In Alg. 5.3 we provide pseudo-code for the procedure $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}(\mathcal{S})$.
Given the functions $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ and $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ defined in proof of Proposition 5.4.7, assume that $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$ is such that $\mathcal{S}_{\mathcal{M}}$ is the characteristic sample of a DFA $\mathcal{D}_{\mathcal{M}}$ over alphabet

Σ_{conc}, constructed from some SFA $\mathcal{M}$ as described in Alg. 5.1. In Lemma 5.4.10 we show that under these assumptions it holds that $\Sigma' = \Sigma_{\mathsf{conc}}$. Then, for the set $\mathcal{S}'$ returned in item 5 of Alg. 5.3, it holds that $\mathcal{S}' = \mathcal{S} \cap \Sigma^*_{\mathsf{conc}}$. Since $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$ and $\Sigma^*_{\mathsf{conc}} \supseteq \mathcal{S}_{\mathcal{M}}$, it holds that $\mathcal{S}' \supseteq \mathcal{S}_{\mathcal{M}}$ and is defined over the alphabet Σ_{conc}. Therefore, $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$ is a decontaminating function. In addition, $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$ runs in time polynomial in the size of $\mathcal{S}$, thus the conditions of Theorem 5.10 are met. ■

Lemma 5.4.10. *Assume that the input to $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$ is $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$ as described in the proof of Proposition 5.4.9. Then, for Σ' constructed in function $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$ it holds that $\Sigma' = \Sigma_{\mathsf{conc}}$.*

Proof. Given the functions $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ and $\mathsf{Generalize}_{\mathbb{M}_{A_m}}$ defined in proof of Proposition 5.4.7, we prove that if $\mathcal{S} \supseteq \mathcal{S}_{\mathcal{M}}$ such that $\mathcal{S}_{\mathcal{M}}$ is the characteristic sample of a DFA $\mathcal{D}_{\mathcal{M}}$ over alphabet Σ_{conc}, constructed from some SFA $\mathcal{M}$ as described in Alg. 5.1, then for $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$ given in Proposition 5.4.9 we have $\Sigma' = \Sigma_{\mathsf{conc}}$. To this end, we show that the set A_w is exactly the set of all lex-access words of states in $\mathcal{D}_{\mathcal{M}}$, and that $\Sigma' = \Sigma_{\mathsf{conc}}$.
First, we show that every $u \in A_w$ is a lex-access word and that $\Sigma' \subseteq \Sigma_{\mathsf{conc}}$. We inductively prove that every word that is added to A_w is a lex-access word; and that every letter that was added to Σ' in some iteration of the procedure $\mathsf{Decontaminate}_{\mathbb{M}_{A_m}}$ is in Σ_{conc}.
For the Base case, we consider $A_w = \{\epsilon\}$ and $\Sigma' = \{d_{\mathit{inf}}\}$. From item 1 of Theorem 5.6, we can assume access words are minimal according to the lexicographic order. Thus, $\epsilon \in A_w$ is indeed a lex-access word (of the state q_0). For $d_{\mathit{inf}} \in \Sigma'$, it holds that Σ_{conc} contains the minimal element of $\mathbb{D}$ since it contains all concretizations of intervals, the SFA is complete and $\mathsf{Concretize}_{\mathbb{M}_{A_m}}$ returns the minimal element of each interval. Therefore $d_{\mathit{inf}} \in \Sigma_{\mathsf{conc}}$.
For the induction step, assume that A_w contains only lex-access words and that the current Σ' is a subset of Σ_{conc}. Then, when considering $u \in A_w$ in item 3, it holds that u is a lex-access word of some state q. Then, σ is added to Σ' only if $u\sigma \not\sim_{\mathcal{S}} ud_{\mathit{inf}}$. Since $\mathcal{S}$ agrees with $\mathcal{M}$, it holds that $\delta_{\mathcal{M}}(q_0, u\sigma) \neq \delta_{\mathcal{M}}(q_0, ud_{\mathit{inf}})$ and σ is a minimal element with that property. Then, σ must be a minimal element of an interval labeling an outgoing transition from q, therefore is in Σ_{conc}. Inductively this holds for all elements added to Σ' in the current iteration. This proves that $\Sigma' \subseteq \Sigma_{\mathsf{conc}}$. Assume now that A_w contains only lex-access words and let $u\sigma$ be a word added to A_w in item 3. Then, for all $u' \in A_w$ it holds that $u\sigma \not\sim_{\mathcal{S}} u'$.

> **Claim.** In this setting, $u\sigma \not\sim_{\mathcal{S}} u'$ implies $u\sigma \not\sim_{\mathcal{S}_{\mathcal{M}}} u'$.
>
> **Proof.** Since we assume all words already in A_w are lex-access words, then in particular u is a lex-access word. In addition, $\sigma \in \Sigma'$ and thus $\sigma \in \Sigma_{\mathsf{conc}}$. Since $u\sigma \not\sim_{\mathcal{S}} u'$ and since $\mathcal{S}$ agrees with $\mathcal{M}$, it holds that $\delta_{\mathcal{M}}(q_0, u\sigma) \neq \delta_{\mathcal{M}}(q_0, u')$. Now, $u\sigma$ and u' are both in Σ_{conc} since from item 3 we have $A_w \subseteq \Sigma'^*$, and thus $\delta_{\mathcal{D}_{\mathcal{M}}}(q_0, u\sigma) \neq \delta_{\mathcal{D}_{\mathcal{M}}}(q_0, v)$, and from item 3 of Theorem 5.6 it holds that $u\sigma \not\sim_{\mathcal{S}_{\mathcal{M}}} u'$. This proofs the claim.

Then, for all $u' \in A_w$ we have $\delta_{\mathcal{D}_{\mathcal{M}}}(q_0, u\sigma) \neq \delta_{\mathcal{D}_{\mathcal{M}}}(q_0, u')$. Then, since we traverse words and letters in lexicographic order, $u\sigma$ is a lex-access word for the state $\delta_{\mathcal{D}_{\mathcal{M}}}(q_0, u\sigma)$. We have shown that every $u \in A_w$ is a lex-access word and that $\Sigma' \subseteq \Sigma_{\mathsf{conc}}$. This concludes the first direction.

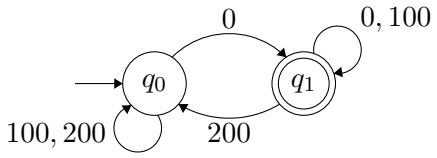


Figure 5.2: The DFA $\mathcal{D}_{\mathcal{M}}$ constructed in Alg. 5.1

We turn to prove the second direction, that is, we show that every lex-access word is in A_w and that $\Sigma_{\mathsf{conc}} \subseteq \Sigma'$.
We start by proving that every lex-access word is in A_w. First note that for ϵ, it holds that $\epsilon \in A_w$. Next, let $u\sigma$ be a lex-access word. For all lex-access words u' found in previous iterations it holds that $u\sigma \not\sim_{\mathcal{S}_{\mathcal{M}}} u'$ from item 2 of Theorem 5.6, and thus $u\sigma \not\sim_{\mathcal{S}} u'$ since $\mathcal{S}_{\mathcal{M}} \subseteq \mathcal{S}$. Then, $u\sigma$ satisfies the condition of item 3 of procedure $\mathsf{Decontaminate}_{\mathbb{M}_{\mathcal{A}_m}}$, and is added to A_w.
To prove $\Sigma_{\mathsf{conc}} \subseteq \Sigma'$, we inductively prove that every letter in Σ_{conc} is add to Σ' in some iteration of $\mathsf{Decontaminate}_{\mathbb{M}_{\mathcal{A}_m}}$. For the base case, let $\sigma \in \Sigma_{\mathsf{conc}}$ be the letter that was added in item 3 with the access word $u = \epsilon$. Note that for every SFA, ϵ is the lex-access word for the state q_0. From the construction of $\mathsf{Concretize}_{\mathbb{M}_{\mathcal{A}_m}}$ it holds that σ is the left endpoint of some interval that is an outgoing transition from q_0. Then, indeed σ is found in the first iteration of item 3. Inductively, let σ label an outgoing transition of q for some $q \in Q$, and let u_q be the lex-access word of q. Since A_w contains all lex-access words, it holds that $u_q \in A_w$, and then the outgoing transitions of q will be considered in some following iteration. Thus, all minimal letters indicating new intervals are added to Σ' and we have that $\Sigma_{\mathsf{conc}} \subseteq \Sigma'$. We conclude that $\Sigma' = \Sigma_{\mathsf{conc}}$. ■

Example 5.4.11. Continuing Example 5.4.5, let $\mathcal{M}$ be the SFA from Figure 5.1 and consider the class $\mathbb{M}_{\mathcal{A}_{\mathbb{N}}}$ of SFAs over the interval algebra. Algorithm **CharSFA** computes the set Σ_{conc} using the function $\mathsf{Concretize}_{\mathbb{M}_{\mathcal{A}_{\mathbb{N}}}}$ given in Example 5.4.5. That is, for the predicates labeling outgoing transitions from q_0 we have $\mathsf{Concretize}_{\mathcal{A}_{\mathbb{N}}}(\langle [0, 100), [100, \infty)\rangle) = \langle \{0\}, \{100\}\rangle$; and for outgoing transitions from q_1, it holds that $\mathsf{Concretize}_{\mathcal{A}_{\mathbb{N}}}(\langle [0, 200), [200, \infty)\rangle) = \langle \{0\}, \{200\}\rangle$. Then, $\Sigma_{\mathsf{conc}} = \{0, 100, 200\}$, and **CharSFA** constructs the DFA over Σ_{conc}, where concrete transitions agree with symbolic transitions of the original SFA. See Figure 5.2 for the resulting DFA $\mathcal{D}_{\mathcal{M}}$. Note the transition $q_0 \xrightarrow{200} q_0$ in $\mathcal{D}_{\mathcal{M}}$. Even-though 200 is not the end-point of any interval labeling the outgoing transitions of q_0 in the SFA $\mathcal{M}$, this is a transition in the DFA $\mathcal{D}_{\mathcal{M}}$ since $200 \in \Sigma_{\mathsf{conc}}$ and since $\mathcal{D}_{\mathcal{M}}$ is complete. In addition, 200 is in the interval $[100, \infty)$ labeling the transition $q_0 \xrightarrow{[100,\infty)} q_0$ in $\mathcal{M}$, and therefore 200 labels the corresponding transition in $\mathcal{D}_{\mathcal{M}}$. After constructing $\mathcal{D}_{\mathcal{M}}$, algorithm **CharSFA** applies **CharDFA** on $\mathcal{D}_{\mathcal{M}}$, and returns the sample set

$$\mathcal{S}_{\mathcal{M}} = \{\langle \epsilon, \bot\rangle, \langle 0, \top\rangle, \langle 100, \bot\rangle, \langle 200, \bot\rangle, \langle 0 \cdot 0, \top\rangle, \langle 0 \cdot 100, \top\rangle, \langle 0 \cdot 200, \bot\rangle\}$$

Now, assume algorithm **InferSFA** is given the set

$$\mathcal{S} = \{\langle \epsilon, \bot\rangle, \langle 0, \top\rangle, \langle 100, \bot\rangle, \langle 150, \bot\rangle, \langle 200, \bot\rangle, \langle 0 \cdot 0, \top\rangle, \langle 0 \cdot 100, \top\rangle, \langle 0 \cdot 200, \bot\rangle\}$$

that subsumes $\mathcal{S}_{\mathcal{M}}$, over the alphabet $\Sigma = \{0, 100, 150, 200\}$. The algorithm **InferSFA** applies $\mathsf{Decontaminate}_{\mathbb{M}_{A_{\mathbb{N}}}}$ that generates the set $\mathcal{S}'$ over Σ_{conc}, where $\mathcal{S}'$ is calculated as follows. It first finds the set Σ_{conc} of all elements that are a minimal left point of some interval, and then chooses from $\mathcal{S}$ the words over Σ_{conc}. First, note that $100 \sim_{\mathcal{S}} 150$, $100 \sim_{\mathcal{S}} 200$ and $150 \sim_{\mathcal{S}} 200$, while $0 \not\sim_{\mathcal{S}} 100, 150, 200$. Since 0 is the minimal element it has to be in Σ_{conc}; and since 100 is the minimal element that is not equivalent to 0 it has to define a new interval and thus is in Σ_{conc} as well. Then, 0 and 100 define the left end-points of all intervals labeling the outgoing transitions of q_0. Next, we consider the suffixes of 0, which is a lex-access word to the state q_1. These are $0 \cdot 0$ and $0 \cdot 100$ that are equivalent, and $0 \cdot 200$ that is not equivalent to the former. Since $0 \cdot 100$ is equivalent to $0 \cdot 0$, it holds that 100 does not define a new interval now, but 200 does as it is the minimal (and only) suffix that is not equivalent to 0 when considering suffixes of 0. Then, we deduce that $\Sigma_{\mathsf{conc}} = \{0, 100, 200\}$ and thus $\mathcal{S}' = \{\langle \epsilon, \bot\rangle, \langle 0, \top\rangle, \langle 100, \bot\rangle, \langle 200, \bot\rangle, \langle 0 \cdot 0, \top\rangle, \langle 0 \cdot 100, \top\rangle, \langle 0 \cdot 200, \bot\rangle\}$.
Algorithm **InferSFA** now applies **InferDFA** on the set $\mathcal{S}'$ and the resulting DFA would be the DFA $\mathcal{D}_{\mathcal{M}}$ of Figure 5.2. Then it applies $\mathsf{Generalize}_{\mathbb{M}_{A_{\mathbb{N}}}}$ described in Example 5.4.5 and the result will be the original SFA of Figure 5.1. That is, for outgoing transitions of q_0 it applies $\mathsf{Generalize}_{\mathbb{M}_{A_{\mathbb{N}}}}(\langle\{0\}, \{100, 200\}\rangle) = \langle[0, 100), [100, \infty)\rangle$ and for outgoing transitions of q_1 it applies $\mathsf{Generalize}_{\mathbb{M}_{A_{\mathbb{N}}}}(\langle\{0, 100\}, \{200\}\rangle) = \langle[0, 200), [200, \infty)\rangle$ and uses these predicates to label the corresponding transitions in the SFA.

Unfortunately, the result of Theorem 5.11 does not extend to the non-monotonic case, as stated in Proposition 5.4.12 regarding SFAs over the propositional algebra. Note that the number of different predicates over $\mathbb{B}^k$ is unbounded. Since the concrete alphabet size is 2^k, we can learn any SFA of size $\Omega(2^k)$ using a characteristic sample for the concrete DFA. Therefore, we consider SFAs which are *useful* in the sense that they are significantly smaller than the corresponding DFA. In particular, the out-degree of the SFA is $O(k)$ rather than $O(2^k)$ as of the DFA.

Proposition 5.4.12. *The class $\mathbb{M}_{Pk}$ of SFAs over the propositional algebra on $\mathbb{B}^k$ with out-degree that is bounded by $O(k)$ is not efficiently identifiable.*

Proof. Assume by way of contradiction that there exist functions $\mathsf{Concretize}_{\mathbb{M}_{Pk}}$ and $\mathsf{Generalize}_{\mathbb{M}_{Pk}}$ satisfying the criteria from Theorem 5.7. Let Ψ be the set of *semantic functions*, over the set of k propositions, i.e., functions that differ in the satisfying assignments (rather than the formula representing them). Then $|\Psi| = 2^{2^k}$. Let Υ be the set of concrete subsets of $\mathbb{B}^k$. Then $|\Upsilon| = 2^{2^k}$ as well. Recall that $\mathsf{Concretize}_{\mathbb{M}_{Pk}}$ should produce a polynomial sized partition. Let Υ_{poly} be the set of polynomially-sized concrete subsets of $\mathbb{B}^k$. Hence, even if we consider singleton concrete/predicate partitions, we obtain $\mathsf{Concretize}_{\mathbb{M}_{Pk}} : \Psi \to \Upsilon_{poly}$ and $\mathsf{Generalize}_{\mathbb{M}_{Pk}} : \Upsilon_{poly} \to \Psi$. Since $|\Upsilon_{poly}| < |\Upsilon| = |\Psi|$, $\mathsf{Concretize}_{\mathbb{M}_{Pk}}$ cannot be one-to-one and $\mathsf{Generalize}_{\mathbb{M}_{Pk}}$ cannot be onto. Thus, for some $\psi \in \Psi$ and some φ for which $[\![\varphi]\!] \neq [\![\psi]\!]$, we have that $\mathsf{Generalize}_{\mathbb{M}_{Pk}}(\mathsf{Concretize}_{\mathbb{M}_{Pk}}(\psi)) = \varphi$, violating the criterion of Theorem 5.7. ■

This argument holds also if we restrict attention to arbitrary DNF (or arbitrary CNF) functions. We remark that there is no exponential gap between the number of all subsets of 2^k and the number of all polynomial-size subsets of 2^k, which may be viewed as an intuition of why the problem of learning a general Boolean function in the PAC (*probably approximately correct*) setting [Val84, KLV94] is hard, and still extensively studied.

5.5 Concluding Remarks

We provide a necessary condition and a sufficient condition for identification of SFAs in the limit using polynomial time and data, as well as a necessary condition for efficient learning of SFAs using MQ and EQ. These imply that SFAs over the propositional algebra are not efficiently identifiable in the limit, and cannot be efficiently learned in the query learning paradigm, either. We show that the sufficient condition for identification of SFAs in the limit using polynomial time and data applies to SFAs over monotonic algebras. We observe that this class of SFAs is also efficiently learnable using MQ and EQ.

We hope that these sufficient or necessary conditions will help to obtain more positive and negative results for learning of SFAs, and spark an interest in investigating characteristic samples in other automata models used in verification.

Chapter 6

Conclusions and Discussion

Program verification aims to formally prove that a system is correct, with respect to a given specification. In program verification we aim to prove that the system is correct for all inputs and for all possible behaviors – with respect to the specification – instead of looking for errors using testing. In this PhD thesis, we address the verification of systems over large and infinite data domains.

Many problems for systems over infinite data domains do not scale well, have high complexity, and are even undecidable. We approach the verification of systems over infinite data domains by suggesting different ways to model them in a finite manner, depending on the type of the system. Such modeling then allows us to find scalable algorithms for verification of such systems (Chapter 4); characterize systems for which there exist efficient algorithms, and in particular, systems that can be learned in polynomial time (Chapter 5); and find specifications for which the model-checking problem is decidable, or suggest bounded model checking algorithms for the undecidable fragments (Chapter 3).

In Chapter 3 we suggest a new model of automata, namely *alternating variable Büchi automata* (AVBWs) that are able to express temporal specifications over infinite data domains, and in particular are able to express the whole fragment of $\exists^*$-VLTL. The existential fragment of VLTL is most suitable for error detection, since it allows us to express the existence of erroneous computations. However, since the model-checking problem for $\exists^*$-VLTL specifications is undecidable [SW], we first characterize decidable fragments of the logic. For these fragments, we suggest a model checking algorithm, based on known methods taken from the world of finite state systems. In particular, we suggest an algorithm for translating AVBWs to non-deterministic variable Buchi automata (NVBWs). Over finite domains, these two automata models have the same expressive power. However, we prove that over infinite data domains, AVBWs are strictly more expressive than NVBWs. Therefore, our translation algorithm is not complete. We use this translation in order to suggest a semi-algorithm for model-checking of $\exists^*$-VLTL specifications. When running without a bound, this semi-algorithm may not halt. Given a bound k set by the user, we can use our algorithm to guarantee that systems of size bounded by k do not violate the given specification. This method is also known as bounded model checking, and is widely used in program verification, see [BCCZ99, CBRZ01].

In Chapter 4 we introduce *communicating systems*, which are C-like programs that are able to communicate data values between one another. Communicating systems naturally model communication and security protocols. Due to the communication between different components, the size of the composed system can grow exponentially in the number of components, and the verification process does not scale well. Thus, in Chapter 4 we suggest a modular verification algorithm that takes advantage of the decomposition of the systems into smaller components. We make use of a known algorithm for compositional verification of systems over finite data domains [CGP03b]. However, while in [CGP03b] the authors rely on the fact that the system is defined over a finite data domain in order to prove the correctness of their algorithm, this is not the case in our setting. We thus first adjust the algorithm to our setting, allowing to find small proofs of correctness for the setting of communicating programs, that are able to model more real-life systems. In addition, in cases where the system is not correct, we suggest a novel repair algorithm, that iteratively eliminates errors and proceeds to verify the repaired system. Our algorithm is incremental and uses the information it has collected in previous verification iterations in order to verify repaired components.
Our work regarding compositional verification and repair makes an important use of the $\mathbf{L}^*$ algorithm for learning regular languages [Ang87b] in order to learn small proofs of correctness. In more detail, in Chapter 4 we follow the techniques of [CGP03b], by looking for an abstraction of one of the components, such that together with the second component, guarantees the correctness of the system. This way, we never need to compute the full composition of the system, but only the composition of the abstraction, which is usually much smaller, allowing the verification to scale. In order to find such abstractions we make use of $\mathbf{L}^*$, exploiting the finite representation of the systems as finite automata.
In Chapter 5 we explore foundational aspects of the learnability of automata over infinite data domains, under different paradigms of automata learning. We analyse the complexity of the $\mathbf{L}^*$ algorithm for automata over infinite data domains, presenting automata classes and domains for which learning is efficient. In addition, we consider a different learning paradigm, from a more information theoretic perspective, namely *learnability in the limit* [dlH97b, Gol78], which tries to learn a finite model for a system using characteristic sample sets. These are sets of polynomial size of allowed and erroneous behaviors of the system. This is the first time that learning in this paradigm is studied in the context of systems over infinite data. We thus present a novel learning algorithm for systems over infinite data in the paradigm of learnability in the limit, which allows learning models for such systems using only a small amount of data.

6.1 Future Work

6.1.1 Automata Learning

Continuing our work of Chapter 5, we note that learnability of n-dimensional box algebra was studied extensively in different contexts [GGM94, BGGM98, BK98, BK00]. As we show in Chapter 5.3, even the class of purely neat SFAs over the n-dimensional box algebra is not

efficiently learnable using MQs and EQs. However, we are yet to determine what is the case for the identification in the limit paradigm. In preliminary results we were able to establish that the class of purely neat SFA satisfies the necessary condition of Chapter 5.4.2. However we were not able to determine whether the sufficient condition holds as well, for purely neat SFAs over the box algebra.
We hope to be able to use these conditions in order to come up with more complexity results for different classes of symbolic automata and Boolean algebras.
In addition, we hope to use the results regarding learnability of symbolic automata in program verification. In Chapter 4 we use $\mathbf{L}^*$ algorithm to learn assumptions for compositional verification. In the case of SFAs, the $\mathbf{MAT}^*$ algorithm is polynomial in the size of the SFA, and terminates given that the underlying algebra is learnable. For monotonic algebras, for example the interval algebra, both $\mathbf{MAT}^*$ and our proposed algorithm for identification in the limit are polynomial in the size of the learned SFA. We hope to find more applications for the learnability of symbolic automata in different aspects of program verification.

6.1.2 Program Synthesis

After studying different aspects of program verification for systems over infinite data, a natural next step is to investigate program synthesis. The problem of program synthesis is to construct a program that is correct-by-construction. That is, given a specification, to automatically construct a correct program with respect to this specification. Program synthesis is extensively studied in the verification community, examples are [PR89, Kup12, FS13]. We consider synthesis of systems over infinite data domains, from different types of temporal logic specifications, as we describe below. The systems we consider are similar to the ones studied in this thesis, allowing us to make use of methods from this thesis in order to synthesize such systems.

Synthesis of Universal Properties In the context of program synthesis we consider universal properties, such as: "all processes are eventually logged-in", or "every process that is logged-in is eventually logged-out". Universal properties are natural as specifications for the synthesis problem, as we usually aim to construct programs that are correct *for all* input values. We suggest to study the synthesis problem of universal VLTL properties. Our preliminary study shows that this problem is too, in general, undecidable. Moreover, we prove that current automata models that are used to model LTL and $\exists^*$-VLTL, are unable to express universal properties. Therefore, identification of fragments that are expressible using existing models (using similar techniques as we used for VLTL presented in Chapter 3) is a natural first step for understanding these specifications. In the long term, we aim to explore the notion of *vacuous synthesis* of universal specifications. For example, consider the specification "every process that is logged-in is eventually logged-out". A program with no processes at all, or no logged-in processes, satisfies this specification. We call such behaviors *vacuous satisfaction*. We wish to construct programs that require at least one process to be logged-in, or even require infinitely many processes to be able to log-in to the system. We aim to look for ways to construct programs for which we can prove this kind of infinite non-vacuous satisfaction.

Synthesis of First Order Specifications Continuing with LTL, we consider LTL specification with first order constraints (FO-LTL). Such specification can reason about the change in program variables over time. For example, we can say "for every point in the computation in which the variable x is equal 0, there is a later point in the computation in which $x > 0$". This is an extension of the specifications discussed in Chapter 4 to the setting of ongoing systems, whose computations are modeled as infinite words. In LTL synthesis [PR89], the propositions that appear in the formula are used as the alphabet of the synthesized program. The program is then able to trigger events that are described using the propositions from the specification. However, in the case of FO-LTL, we wish not only to trigger events, but to construct programs that can manipulate the variable values. The manipulations applied on the values of the program variables do not necessarily appear in the specification. In fact, they are usually not part of the specification as the specification specifies the desired behavior but not how to achieve it. For example, the specification might contain the requirement $x > 0$, but usually it does not include assignments such as $x := x + 1$. In order to infer these variable manipulations, we need not only to learn the structure of the program, but also to deduce the program statements.
In order to construct the desired program, we aim to use automata learning methods, derived from the $\mathbf{L}^*$ algorithm. The output is a finite automaton, that can be viewed as the control-flow graph of the program. While in Chapter 4 that discusses compositional verification, the program statements for the learned abstraction were taken from the components of the system, and thus known in advance, in this proposed line of research, one of the greatest challenges is to automatically deduce program statements from the specification. In Chapter 4 we were able to deduce statements in order to repair the system, and we hope that we will be able to use similar methods in order to learn program statements in the context of synthesis.

Bibliography

[AD18] G. Argyros and L. D'Antoni. The learnability of symbolic automata. In *Computer Aided Verification - 30th Int. Conf., CAV*, volume 10981 of *LNCS*, pages 427–445. Springer, 2018.

[ADG16] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

[AEF15] D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *Proc. of the Twenty-Fourth Int. Joint Conf. on Artificial Intelligence, IJCAI*, pages 3308–3314, 2015.

[AF16] D. Angluin and D. Fisman. Learning regular omega languages. *Theor. Comput. Sci.*, 650:57–72, 2016.

[AFH99] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.

[Ang81] D. Angluin. A note on the number of queries needed to identify regular languages. *Inf. Control.*, 51(1):76–87, 1981.

[Ang87a] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[Ang87b] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[Ang90] D. Angluin. Negative results for equivalence queries. *Mach. Learn.*, 5:121–150, 1990.

[ASJ+16] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security*, pages 1690–1701. ACM, 2016.

[ASKK16] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis. Back in black: Towards formal, black box analysis of sanitizers and filters. In *IEEE Symposium on Security and Privacy, SP*, pages 91–109, 2016.

[AV10] F. Aarts and F. Vaandrager. Learning I/O automata. In *Concurrency Theory, 21th Int. Conf., CONCUR 2010*, pages 71–85, 2010.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL*, 1988.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 1992.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.

[BGGM98] N. H. Bshouty, P. W. Goldberg, S. A. Goldman, and H. D. Mathias. Exact learning of discretized geometric concepts. *SIAM J. Comput.*, 28(2):674–699, 1998.

[BGJ+05] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *Fundamental Approaches to Software Engineering, 8th Int. Conf., FASE*, volume 3442, pages 175–189. Springer, 2005.

[BHJS07] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. Rewriting systems with data. In *Fundamentals of Computation Theory, FCT*, 2007.

[BHKL09] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *IJCAI*, pages 1004–1009, 2009.

[BJR06] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *Fundamental Approaches to Software Engineering, 9th Int. Conf., FASE*, volume 3922 of *LNCS*, pages 107–121. Springer, 2006.

[BK98] A. Beimel and E. Kushilevitz. Learning boxes in high dimension. *Algorithmica*, 22(1/2):76–90, 1998.

[BK00] A. Beimel and E. Kushilevitz. Learning unions of high-dimensional boxes over the reals. *Inf. Process. Lett.*, 73(5-6):213–220, 2000.

[BKMZ15] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

[BKV13] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2013.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[BMS+06] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 7–16. IEEE Computer Society, 2006.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Bue62] Julius R. Buechi. On a decision method in restricted second-order arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[BV96] F. Bergadano and S. Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM J. Comput.*, 25(6):1268–1280, 1996.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CCF+10] Yu-Fang Chen, Edmund M. Clarke, Azadeh Farzan, Ming-Hsien Tsai, Yih-Kuen Tsay, and Bow-Yaw Wang. Automated assume-guarantee reasoning through implicit learning. In *CAV*, 2010.

[CDYS17] K. Chubachi, Diptarama, R. Yoshinaka, and A. Shinohara. Query learning of regular languages over large ordered alphabets. In *The 1st workshop on Learning and Automata (LearnAut)*, 2017.

[CFC+09] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating DFA's for compositional verification. In *TACAS*, 2009.

[CGP01] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2001.

[CGP03a] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, 2003.

[CGP03b] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[CHYS19] K. Chubachi, D. Hendrian, R. Yoshinaka, and A. Shinohara. Query learning algorithm for residual symbolic finite automata. In *Proc. Tenth Int. Symposium on Games, Automata, Logics, and Formal Verification, GandALF*, pages 140–153, 2019.

[CKKS20] Hana Chockler, Pascal Kesseli, Daniel Kroening, and Ofer Strichman. Learning the language of software errors. *J. Artif. Intell. Res.*, 67:881–903, 2020.

[CM04] Séverine Colin and Leonardo Mariani. Run-time verification. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[CS] Sagar Chaki and Ofer Strichman. Optimized L*-based assume-guarantee reasoning, TACAS, 2007.

[CT04] Alexander Clark and Franck Thollard. Pac-learnability of probabilistic deterministic finite state automata. *J. Mach. Learn. Res.*, 5:473–497, 2004.

[DD13] Isil Dillig and Thomas Dillig. Explain: A tool for performing abductive inference. In *CAV*, 2013.

[DD17] S. Drews and L. D'Antoni. Learning symbolic automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd Int. Conf., TACAS*, pages 173–189, 2017.

[dlH97a] C. de la Higuera. Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27(2):125–138, 1997.

[dlH97b] C. de la Higuera. Characteristic sets for polynomial grammatical inference. *Mach. Learn.*, 27(2):125–138, 1997.

[DLT16] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *STTT*, 18(2):205–225, 2016.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[DV14] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 541–554. ACM, 2014.

[DV16] L. D'Antoni and M. Veanes. Minimization of symbolic tree automata. In *Proc. of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 873–882. ACM, 2016.

[DVLM14] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: a transducer-based language for tree manipulation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI*, pages 384–394. ACM, 2014.

[EGPS15] Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. Automated circular assume-guarantee reasoning. In *FM*, 2015.

[EGPS16] Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement. In *CAV*, 2016.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[exa] https://www.dropbox.com/home/tacas2020%20examples.

[FGPS20] Hadar Frenkel, Orna Grumberg, Corina S. Pasareanu, and Sarai Sheinvald. Assume, guarantee or repair. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin,*

Ireland, April 25-30, 2020, Proceedings, Part I, volume 12078 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2020.

[FGS] Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to modeling systems and specifications over infinite data. In *NASA Formal Methods, 2017*.

[FGS19] Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to model-checking systems and specifications over infinite data domains. *J. Autom. Reasoning*, 63(4):1077–1101, 2019.

[Fis18] D. Fisman. Inferring regular languages and ω-languages. *J. Log. Algebraic Methods Program.*, 98:27–49, 2018.

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):519–539, 2013.

[GGM94] P. W. Goldberg, S. A. Goldman, and H. D. Mathias. Learning unions of boxes with membership and equivalence queries. In *Proc. of the Seventh Annual ACM Conf. on Computational Learning Theory, COLT*, pages 198–207. ACM, 1994.

[GGP07] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007.

[GJL10] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. *Theor. Comput. Sci.*, 411(47):4029–4054, 2010.

[GKS10] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In *Language and Automata Theory and Applications, LATA*, 2010.

[GKS12] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Model checking systems and specifications with parameterized atomic propositions. In *Automated Technology for Verification and Analysis, ATVA*, 2012.

[GKS14] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. A game-theoretic approach to simulation of data-parameterized systems. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2014.

[GM96] S. A. Goldman and H. D. Mathias. Teaching a smarter learner. *J. Comput. Syst. Sci.*, 52(2):255–267, 1996.

[GMF08] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301, 2008.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Inf. Control.*, 37(3):302–320, 1978.

[GP16] Hadrien Glaude and Olivier Pietquin. PAC learning of probabilistic automaton based on the method of moments. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 820–829. JMLR.org, 2016.

[GPB02] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 3–12. IEEE Computer Society, 2002.

[GPB05] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.

[HD17] Q. Hu and L. D'Antoni. Automatic program inversion using symbolic transducers. In *Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI*, pages 376–389. ACM, 2017.

[HLM+11] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proc.* USENIX Association, 2011.

[HPU17] Klaus Havelund, Doron Peled, and Dogan Ulus. First order temporal logic monitoring with bdds. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 116–123. IEEE, 2017.

[HSM11] F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *Verification, Model Checking, and Abstract Interpretation - 12th Int. Conf., VMCAI*, volume 6538 of *LNCS*, pages 263–277. Springer, 2011.

[HV11] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Verification, Model Checking, and Abstract Interpretation*

- *12th Int. Conf., VMCAI*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[KLV94] M. J. Kearns, M. Li, and L. G. Valiant. Learning boolean formulas. *J. ACM*, 41(6):1298–1328, 1994.

[KT14] M. Keil and P. Thiemann. Symbolic solving of extended regular expression inequalities. In *34th Int. Conf. on Foundation of Software Technology and Theoretical Computer Science, FSTTCS*, pages 175–186, 2014.

[Kup12] Orna Kupferman. Recent challenges and ideas in temporal synthesis. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings*, volume 7147 of *Lecture Notes in Computer Science*, pages 88–98. Springer, 2012.

[LDD+13] Boyang Li, Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. In *TACAS*, 2013.

[LH14] Shang-Wei Lin and Pao-Ann Hsiung. Compositional synthesis of concurrent systems through causal model checking and learning. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 416–431, 2014.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[McM99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, 1999.

[MH84] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984.

[MJG+12] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs*. Wiley, 1999.

[MM14] O. Maler and I. Mens. Learning regular languages over large alphabets. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th Int. Conf., TACAS*, volume 8413 of *LNCS*, pages 485–499. Springer, 2014.

[MM17] O. Maler and I. Mens. A generic algorithm for learning symbolic automata from membership queries. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pages 146–169, 2017.

[MP95] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118(2):316–326, 1995.

[MRA+17] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna. StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In *Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI*, pages 693–708. ACM, 2017.

[MS84] David E. Muller and Paul E. Schupp. Alternating automata on infinite objects, determinacy and rabin's theorem. In Maurice Nivat and Dominique Perrin, editors, *Automata on Infinite Words, Ecole de Printemps d'Informatique Théorique, Le Mont Dore, May 14-18, 1984*, volume 192 of *Lecture Notes in Computer Science*, pages 100–107. Springer, 1984.

[Nak00] A. Nakamura. Query learning of bounded-width obdds. *Theor. Comput. Sci.*, 241(1-2):83–114, 2000.

[NSV01] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. In Jirí Sgall, Ales Pultr, and Petr Kolman, editors, *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings*, volume 2136 of *Lecture Notes in Computer Science*, pages 560–572. Springer, 2001.

[NT00] Kedar S. Namjoshi and Richard J. Trefler. On the competeness of compositional reasoning. In *CAV*, 2000.

[OG92] J. Oncina and P. GarcÃa. Inferring regular languages in polynomial update time. *World Scientific*, 01 1992.

[PGB+08] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[PGLM15] M. D. Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 329–341. ACM, 2015.

[PH32] C.S. Peirce and C. Hartshorne. *Collected Papers of Charles Sanders Peirce*. Belknap Press, 1932.

[Pit89] L. Pitt. Inductive inference, dfas, and computational complexity. In *Proc. Int. Workshop on Analogical and Inductive Inference*, pages 18–44, 1989.

[Pnu79] Amir Pnueli. The temporal semantics of concurrent programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1979.

[Pnu85] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, 1985.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190, 1989.

[RV11] Kristin Y. Rozier and Moshe Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 417–431, 2011.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327. IEEE Computer Society, 1988.

[Sak90] Y. Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theor. Comput. Sci.*, 76(2-3):223–242, 1990.

[SGP10] Rishabh Singh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning component interfaces with may and must abstractions. In *Computer Aided*

Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, pages 527–542, 2010.

[She19] S. Sheinvald. Learning deterministic variable automata over infinite alphabets. In *Formal Methods - The Next 30 Years - Third World Congress, FM*, volume 11800 of *LNCS*, pages 633–650. Springer, 2019.

[SV17] O. Saarikivi and M. Veanes. Translating c# to branching symbolic transducers. In *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations*, volume 1 of *Kalpa Publications in Computing*. EasyChair, 2017.

[SW] Fu Song and Zhilin Wu. Extending temporal logics with data variable quantifications. In *FSTTCS 2014*.

[Vaa17] F. W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017.

[Val84] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[Var95] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata*, 1995.

[VdHT10] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Third Int. Comf, on Software Testing, Verification and Validation, ICST*, pages 498–507. IEEE Computer Society, 2010.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86)*, 1986.

[Wei84] V. Weispfenning. Quantifier elimination and decision procedures for valued fields. *Models and Sets.Lecture Notes in Mathematics (LNM)*, 1103:419––472, 1984.

[ZSZR18] X. Zhu, A. Singla, S. Zilles, and A. N. Rafferty. An overview of machine teaching. *CoRR ArXiv*, abs/1801.05927, 2018.

מערכות אלה כוללות מספר רכיבים שיכולים לתקשר ביניהם נתונים, ולבצע פעולות על נתונים אלה. מערכות מתקשרות הן מודל טבעי עבור פרוטוקולי תקשורת ואבטחת מידע. אנו מנצלים את החלוקה של המערכת לרכיבים שונים על מנת להציע אלגוריתם מודולרי לאימות פורמלי של המערכת כולה. לצורך זה, אנחנו משתמשים בלמידת אוטומטים, ומציעים אלגוריתם לאימות ולתיקון של מערכות מתקשרות בעזרת אלגוריתמים ידועים ללמידת אוטומטים. תיקון המערכת מתבצע על ידי מציאת שגיאות במערכת והסרת החישובים הגורמים לשגיאה.

לבסוף, אנחנו חוקרים למידת אוטומטים גם בהקשר של אוטומטים סימבוליים. אוטומט סימבולי הוא אוטומט מעל תחום ערכים אינסופי, כך שכל מעבר שלו מיוצג על ידי פרדיקט מעל אלגברה בוליאנית נתונה. למשל, האי"ב של אוטמטים סימבוליים יכול להיות אוסף כל האינטרוולים מעל המספרים הטבעיים. כך האוטומט יכול לקרוא רצפים של מספרים טבעיים ולהתאים אותם לפרדיקטים המייצגים מעברים באוטומט. אנו חוקרים מספר פרדיגמות למידה בהקשר של אוטומטים סימבוליים ומציגים אלגוריתמים יעילים של למידת אוטמטים, בפרט עבור אוטומטים מעל המספרים הטבעיים והממשיים. למידת אוטומטים היא כלי עזר חשוב בתחום אימות התוכנה, כפי שקורה למשל במודל המערכות המתקשרות, ומכאן המוטיבציה לחקור אלגוריתמי למידה עבור מערכות מעל תחומי ערכים אינסופיים.

תקציר

עבודת המחקר המוצגת בתזה זו עוסקת באימות פורמלי של מערכות מעל תחומי ערכים גדולים מאוד או אינסופיים. דוגמאות למערכות כאלה הן פרוטוקולי תקשורת, פרוטוקולי אבטחת מידע, מסדי נתונים גדולים, ועוד. באימות פורמלי, אנו מעוניינים לוודא בצורה אוטומטית שתכנית נתונה היא נכונה ביחס למפרט נתון. כך, בהינתן מפרט, אנחנו יכולים להוכיח שהתכנית מספקת את המפרט עבור כל קלט אפשרי לתכנית. במקרה שהתכנית לא מספקת את המפרט, ותהליך האימות נכשל, אנחנו מעוניינים לקבל חישוב ספציפי של התכנית המציג את השגיאה, כך שנוכל לתקן את התכנית בהמשך.

בתזה זו אנחנו עוסקים בגישה מבוססת אוטומטים לאימות פורמלי של מערכות. בגישה זו, המערכת והמפרט, ממודלים שניהם בצורה סופית על ידי אוטומטים סופיים, וכך ניתן להשתמש באלגוריתמים מעל אוטומטים סופיים על מנת לבדוק האם המערכת מספקת את המפרט. עם זאת, מערכות רבות מחיי היום־יום מכילות תחומי ערכים אינסופיים. כך למשל, במערכת של שרת המתקשר עם לקוחות, מספר הלקוחות במערכת אינו חסום והם יכולים להתחבר ולהתנתק מהמערכת כרצונם. לכן, אחד האתגרים הגדולים כאשר בוחנים מערכות כאלה, הוא למדל אותן בצורה סופית המאפשרת בדיקה ואימות של המערכות. בפרט, לא ניתן למדל מערכות אלה באמצעות אוטומטים סופיים מעל א"ב סופי, כפי שנעשה בדרך כלל באימות פורמלי של מערכות מעל תחומי ערכים סופיים. אתגר נוסף הוא למצוא אלגוריתמים יעילים לאימות פורמלי של מערכות מעל תחומי ערכים אינסופיים, שכן האימות של מערכות כאלה יכול להיות לא יעיל, והבעיה במקרה הכללי היא בלתי כריעה. על מנת להתמודד עם אתגרים אלה, אנו מציעים בתזה זו מודלים שונים עבור מערכות מעל תחומי ערכים אינסופיים. מודלים אלה משמשים אותנו לבניית אלגוריתמים לאימות יעיל של מערכות מעל תחומי ערכים אינסופיים.

תחילה, אנחנו בוחנים מערכות להן יש חישובים מתמשכים, כגון שרת הפועל תמיד ומתקשר עם לקוחות המתחברים ומתנתקים מהמערכת. אנו משתמשים במפרטים טמפורליים על מנת להביע את הדרישות מהמערכת. מפרטים טמפורליים הם מפרטים המתייחסים להתנהגות של המערכת לאורך זמן. אנו מראים כי המודלים הקיימים לא מסוגלים למדל מפרטים טמפורליים מעל תחומי ערכים אינסופיים, ולכן אנחנו מציעים מודל חדש של אוטומטים המסוגל להביע מפרטים אלו. אנו משתמשים במודל האוטמט החדש על מנת להציע אלגוריתם לאימות פורמלי של מערכות מעל תחומי ערכים אינסופיים, אל מול מפרטים טמפורליים.

בהמשך, אנחנו בוחנים סוג נוסף של מערכות, להן אנו קוראים "מערכות מתקשרות".

המחקר בוצע בהנחייתן של פרופסור ארנה גרימברג וד"ר שרי שינולד בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי־עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Hadar Frenkel, Orna Grumberg, Corina S. Pasareanu, and Sarai Sheinvald. Assume, guarantee or repair. In *TACAS Part I*, pages 211–227. Springer, 2020.

Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to modeling systems and specifications over infinite data. In *NFM*, pages 1–18. Springer, 2017.

Hadar Frenkel, Orna Grumberg, and Sarai Sheinvald. An automata-theoretic approach to model-checking systems and specifications over infinite data domains. In *J. Automated Reasoning 63*, pages 1077–1101. Springer, 2019.

תודות

התזה הזאת לא הייתה נכתבת בלי עזרתן והנחייתן של ארנה גרימברג ושרי שינולד. לא יכולתי לבקש מנחות טובות יותר מכן. הייתן לי הרבה יותר ממנחות אקדמיות. הייתן לי בית ומשפחה בשבע השנים האחרונות, ליוויתן אותי גם בפן האקדמי וגם בפן האישי, ותמיד עזרתן לי לעלות למעלה. למדתי מכן כל כך הרבה. אין לי מילים חוץ מתודה.
אני חייבת את תודתי הרבה גם לדנה פיסמן. ראית בי שותפה למרות הפער העצום בינינו, למדתי ממך המון, ונהניתי מכל רגע. בנוסף, אני מודה לסנדרה זילס ולקורינה פסרינו ־ תודה על ההזדמנות לעבוד איתכן, על האמונה בי ועל שיתוף הפעולה.
ההורים שלי, עדין ומיכל. אמא ־ האמונה שלך בי, הדוגמא האישית שלך, לימדת אותי מהי חריצות, מהי עבודה קשה ומהי התמדה. אבא, שנטעת בי את אהבת המתמטיקה, אהבת הידע, ותמיד רצית שאכוון הכי גבוה שאפשר.
חמי וחמותי, יוני ונחמה ־ הלימודים שלנו היו המשימה שלכם. בלי העזרה הרבה שלכם עם הילדים ובלי התמיכה האינסופית לעולם לא היה לי זמן לסיים את הדוקטורט.
כל החברים הרבים שאספתי בדרך. התברכתי בכם. באמת.
ובעיקר, מי שהיה שם תמיד תמיד בשבילי. ידעת מתי לוותר לי, ומתי למשוך אותי למעלה. האמנת בי בכל ליבך אפילו כשאני ממש לא. אתה החבר הכי טוב שלי. תודה שאתה.
התזה הזאת מוקדשת לילדים שלי, רועי־אהרון, אוריה־שמואל וסיני־אליה. אתם העולם שלי.

הכרת תודה מסורה לטכניון, למרכז המחקר לאבטחת סייבר ע"ש הירושי פוג'יוארה ולמערך הסייבר הלאומי על מימון מחקר זה.

אוטומטים מעל תחומי ערכים אינסופיים: למידות ויישומים באימות ותיקון תוכנה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

הדר פרנקל

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
תמוז התשפ"א חיפה יוני 2021

אוטומטים מעל תחומי ערכים אינסופיים: למידות ויישומים באימות ותיקון תוכנה

הדר פרנקל