Generalisation of Alternating Automata over Infinite Alphabets

Xiao Xu

To cite this version:

Xiao Xu. Generalisation of Alternating Automata over Infinite Alphabets. Formal Languages and Automata Theory [cs.FL]. Université Grenoble Alpes, 2020. English. NNT : . tel-02915498

HAL Id: tel-02915498

https://hal.science/tel-02915498

Submitted on 14 Aug 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UGA
Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Mathématiques et Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

« Xiao / XU »

Thèse dirigée par **Radu IOSIF, HdR, CR, CNRS VERIMAG**,
et
codirigée par **Susanne GRAF, DR, CNRS VERIMAG**

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Généralisation d'Automates Alternatifs sur des Alphabets Infinis

Thèse soutenue publiquement le **« 27 Janvier 2020 »**,
devant le jury composé de :

Monsieur, Radu, IOSIF
Chargé de Recherche, CNRS, VERIMAG, Directeur de Thèse
Madame, Susanne, GRAF
Directeur de Recherche, CNRS, VERIMAG, Co-Directeur de Thèse
Monsieur, Tomas, VOJNAR
Professeur, Brno University of Technology, Rapporteur
Monsieur, Andreas, PODELSKI
Professeur, University of Freiburg, Rapporteur
Monsieur, Margus, VEANES
Chercheur Principal, Microsoft Research Lab, Membre
Monsieur, Ahmed, BOUAJJANI
Professeur, Université de Paris Diderot (Paris 7), IRIF, Président

Abstract

The language inclusion problem is recognised as being central to verification in different domains, such as hardware, communication protocols, software systems, etc. There we might face two challenges: non-determinism and infinite alphabets.

We propose two models of alternating automata over infinite alphabets: (i) alternating data automata (ADA) and (ii) first-order alternating data automata (FOADA). They both recognise the data words over infinite alphabets. In ADA model, the control states are Booleans and the transition rules are specified by a set of formulae in a combined first-order theory of states (Booleans) and data that relate past values of variables with current values of variables. But a restriction of the ADA model is that, there is not hidden variable, hence all the data values taken by the variables are visible in the input. But in FOADA model, the arguments of a predicate atom track the values of the internal variables associated with the state, and these values are invisible in the input sequence, which overcomes the restriction of the ADA model.

With these two alternating models, Boolean operations of union, intersection and complement can be done in linear time, thus matching the complexity of performing these operations in the finite-alphabet case. However, the price to be paid here is that the emptiness checking becomes undecidable. For this reason, we provide two efficient semi-algorithms for emptiness checking: (i) lazy predicate abstraction [33] and (ii) IMPACT method [45]. These semi-algorithms are proven to terminate by returning a word from the language of the given automaton if one exists; but if the language of the given automaton is empty, then the termination is not guaranteed.

The main application of our models is checking inclusions between various classes of automata extended with variables ranging over infinite domains that recognise languages over infinite alphabets. The most widely known classes of this kind are timed automata and finite-memory (register) automata. Another application is checking safety (mutual exclusion, absence of deadlocks, etc.) and liveness (termination, lack of starvation, etc.) properties of parameterised concurrent programs.

Besides the theoretical parts, we also have developed a tool - **FOADA Checker** [62], mainly used for checking inclusion between two automata or checking emptiness of an automaton. FOADA Checker is written in Java, via Java-SMT interface [57] and using Z3 SMT solver [53] for spuriousness, coverage queries and interpolant generation. The IMPACT semi-algorithm [45] has been implemented in the tool to check the emptiness of an automaton.

Keywords: *Model Checking, Verification, Emptiness, Language Inclusion, Alternating, Infinite Alphabets, IMPACT*

Résumé

Le problème de l'inclusion linguistique est reconnu comme étant au cœur de la vérification dans différents domaines, tels que le matériel, les protocoles de communication, les systèmes logiciels, etc. Nous pouvons être confrontés à deux défis: le non-déterminisme et les alphabets infinis.

Nous proposons deux modèles d'automates alternatifs sur des alphabets infinis : (i) les automates alternatifs de données (ADA) et (ii) les automates alternatifs de données du premier ordre (FOADA). Ils reconnaissent tous deux les mots de données sur des alphabets infinis. Dans le modèle ADA, les états de contrôle sont des booléens et les règles de transition sont spécifiées par un ensemble de formules combinées dans une théorie des états du premier ordre (booléens) et des données associant les valeurs passées des variables aux valeurs actuelles des variables. Mais le modèle ADA a une restriction : il n'y a pas de variable cachée, ainsi toutes les valeurs de données prises par les variables sont visibles dans l'entrée. Pourtant dans le modèle FOADA, les arguments d'un atome de prédicat tracent les valeurs des variables internes associées à l'état, et ces valeurs sont invisibles dans la séquence d'entrée, ce qui surmonte la restriction du modèle ADA.

Avec ces deux modèles en alternance, les opérations booléennes d'union, d'intersection et de complément peuvent être effectuées en temps linéaire, ce qui correspond à la complexité de l'exécution de ces opérations dans le cas d'un alphabet fini. Cependant, le prix à payer ici est que la vérification du vide devient indécidable. Pour ceci, nous fournissons deux semi-algorithmes efficaces pour la vérification du vide : (i) abstraction de prédicats paresseux [33] et (ii) méthode IMPACT [45]. S'il existe un mot du langage de l'automate donné, il est prouvé que ces semi-algorithmes se terminent en le retournant; mais si la langue de l'automate donné est vide, la terminaison n'est pas garantie.

La principale application de nos modèles est de vérifier l'inclusion entre différentes classes d'automates étendues avec des variables allant de domaines infinis reconnaissant les langues à des alphabets infinis. Les plus connues de ce genre de classes sont les automates temporisés et les automates à mémoire finie (registre). Une autre application est de vérifier les propriétés de sécurité (exclusion mutuelle, absence de blocages, etc.) et de vitalité (résiliation, absence de famine, etc.) des programmes concurrents paramétrés.

Outre les parties théoriques, nous avons également développé un outil - FOADA Checker [62], en général à l'usage de la vérification de l'inclusion entre deux automates ou de la vérification du vide d'un automate. FOADA Checker est écrit en Java, via l'interface Java-SMT [57] et en utilisant le solveur Z3 SMT [53] pour les parasites, les requêtes de couverture et la génération d'interpolation. Le semi-algorithme IMPACT [45] a été implémenté dans l'outil pour vérifier le vide d'un automate.

Mots-Clés: *Vérification de Modèle, Vérification, Vide, Inclusion Linguistique, Alternance, Alphabets Infinis, IMPACT*

Acknowledgements

I must start by thanking my supervisor, Radu Iosif, whose patience, guidance, encouragement, support and trust were key to achieving this thesis. I got precious experiences from both theoretical and practical aspects. I am also indebted to my co-supervisor, Susanne Graf, who spent precious time in helping me.

Besides my supervisors, I would like to thank the rest of my thesis committee: Mr. VOJNAR Thomas, Mr. PODELSKI Andreas, Mr. VEANES Margus and Mr. BOUAJJANI Ahmed, for their encouragement, insightful comments and hard questions.

Last but not the least, I would like to thank my family, especially my dear wife, Taoran YAN, for supporting me.

Contents

1 Introduction **11**

1.1 Motivation 11

1.2 Challenges 14

1.2.1 Non-Determinism 14

1.2.2 Infinite Alphabets 16

1.3 State-of-the-Art 17

1.3.1 Solutions for Non-Determinism 17

1.3.2 Solutions for Infinite Alphabets 17

1.3.3 Solutions for both Non-Determinism and Infinite Alphabets 20

1.3.4 Solutions for Language Inclusion 21

1.4 Contributions 22

1.4.1 Alternating Data Automata (ADA) 22

1.4.2 First-Order Alternating Data Automata (FOADA) 23

1.4.3 FOADA Checker 24

1.4.4 Applications 24

1.5 Organisation 24

1.6 Notations 25

2 Preliminaries **27**

2.1 First-Order Logic 27

2.1.1 Functions and Constants 27

2.1.2 Terms 28

2.1.3 Formulae 28

2.1.4 Interpretation and Valuation 29

2.1.5 Interpretation of Terms 29
2.1.6 Semantics of Formulae 29
2.2 Interpolation 30
2.2.1 Craig's Interpolation 30
2.2.2 Lyndon's Interpolation 30
2.3 Automata on Finite Words 30
2.3.1 Non-Deterministic Finite Automata (NFA) 30
2.3.2 Runs and Languages of NFA 31
2.3.3 Deterministic Finite Automata (DFA) and Determinisation 32
2.3.4 Complementation of NFA 34
2.4 Alternating Finite Automata (AFA) 34
2.4.1 Definition of AFA 34
2.4.2 Languages of AFA 36
2.5 Data Automata (DA) 37
2.5.1 Definition of DA 37
2.5.2 Languages of DA 37
2.5.3 Determinisation 38
2.5.4 Closure Properties 39

3 Alternating Data Automata (ADA) 41
3.1 Introduction of ADA 41
3.1.1 Data Words 41
3.1.2 Definition of ADA 42
3.1.3 Time Stamp and Accepted Words 43
3.2 Closure Properties of ADA 44
3.2.1 Intersection 44
3.2.2 Union 44
3.2.3 Complementation 45
3.2.4 Proofs for Boolean Closures 45
3.3 Antichains and Interpolants for ADA Emptiness 47
3.3.1 Undecidability for Emptiness Problem 47
3.3.2 Post-Images and Acceptance Function 47
3.3.3 Improvement by Anti-Chains 48

3.3.4 Safety Invariants 48
3.3.5 Abstraction and Refinement 49
3.4 Checking Emptiness - Lazy Predicate Abstraction 52
3.4.1 Abstract Reachability Tree (ART) 52
3.4.2 Lazy Predicate Abstraction Semi-Algorithm 53
3.5 Checking Emptiness - IMPACT 57
3.5.1 In-Place Refinement and Coverage 57
3.5.2 IMPACT Semi-Algorithm 57

4 First-Order Alternating Data Automata (FOADA) 63
4.1 Introduction of FOADA 64
4.1.1 Data Words 64
4.1.2 Definition of FOADA 64
4.1.3 Execution Semantic 65
4.2 Symbolic Execution of FOADA 66
4.2.1 Path Formulae 66
4.2.2 Acceptance Formulae 68
4.2.3 Elimination of Path Quantifiers 69
4.2.4 Elimination of Predicate Atoms 71
4.3 Closure Properties of FOADA 73
4.4 Emptiness Problem of FOADA 74
4.4.1 Unfoldings of FOADA 74
4.4.2 IMPACT Semi-Algorithm 75
4.5 Interpolant Generation of FOADA 78
4.5.1 Over-Approximation and Interpolants 78
4.5.2 Unfolding with Non-local Interpolants 84

5 Applications 87
5.1 Application on Timed Automata 87
5.2 Application on Register Automata 89
5.3 Application on Predicate Automata 91

6 FOADA Checker 93
6.1 Brief User Guide 93

6.1.1 Installation 93
6.1.2 Emptiness Checking 94
6.1.3 Inclusion Checking 95
6.2 Input Format - First-Order Alternating Data Automata (FOADA) 96
6.3 Input Format - Alternating Data Automata (ADA) 97
6.4 Input Format - Predicate Automata (PA) 99
6.5 Experimental Results 100

7 Conclusions **103**
7.1 Summary of Contributions 103
7.2 Future Work 104

Chapter 1

Introduction

1.1 Motivation

The growth in complexity of designs increases the importance of system verification techniques in many domains, such as hardware [59], software engineering, transportation, banking, telecommunications, national defence, aerospace and aeronautical engineering, etc. This could be attributed to important safety requirements where errors either have huge commercial significance, or even lead to life-threatening situations such as in the transport systems, power plants and so forth.

The system verification aims at using formal proofs to demonstrate that a system meets a certain specification. According to the needs, we pick up interesting information from the description of a system or a specification, which is usually written in natural languages, and then, we can use this filtered information to re-describe the system or the specification in an abstract way under a certain specific concept, which is called a **model**. A model is an abstraction that helps to explain a system or a specification, and can be used for studying the effects of different components or for making predictions about behaviour.

Finite-state automaton (FSA) is a largely used model for verification. It is a mathematical model of computation in which different states of the system (or the specification) are defined as the **states** of the model. The behaviour of the system (or the specification) is represented by discrete state changes, called **transitions**. The transitions of the model are triggered by actions or events, formally called **input symbols**. A **word** is a sequence of input symbols. Taking one by one the input symbol from a word and starting from the initial state of the model, if there exists an execution that leads to a final state of the model, then the word is **accepted** by the model. The set of all the words accepted by a model $\mathcal{A}$ is called the **language** of $\mathcal{A}$, often denoted as $\mathcal{L}(\mathcal{A})$. In addition, the set of all possible input symbols is called the **alphabet** of the model, usually denoted by Σ, and Σ^* is the set of finite words with symbols from Σ.

Example 1.1 *(Automaton for System) In a chemical production line, starting with an empty bottle, we add chemical product in the bottle, and in the end we put a cap on. There are two types of product: A and B; and there are two types of caps: normal cap and special cap. The*

bottles with different products must use different caps and Table 1.1 shows the type specification of the cap for different products.

Product	**Cap**
A	Normal
B	Special
A + B	Special

Table 1.1: Cap Type Specification for Different Products

The finite-state automaton in Figure 1.1 explains how a complete chemical product is produced in this chemical production line.

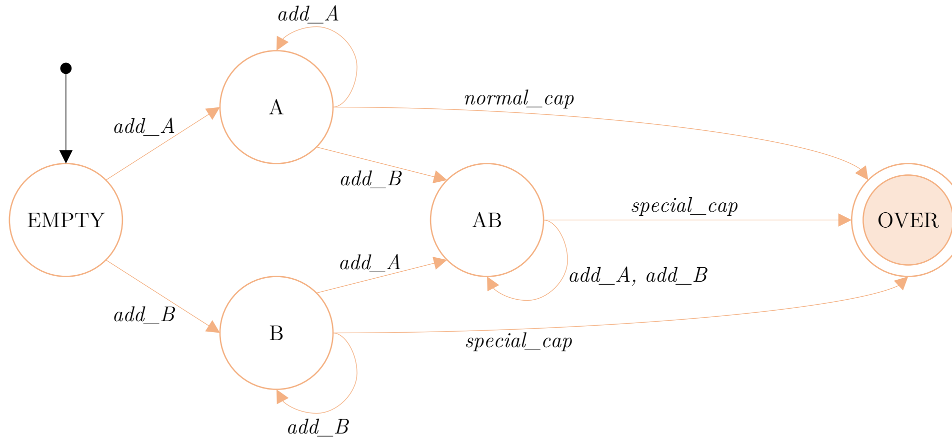


Figure 1.1: An FSA for a Chemical Production System

Given a system that has already been modelled by an automaton, we can build automata for the specifications that we want to check for the given system, over the same alphabet, hence the same actions (or events) for both system and specifications.

Example 1.2 *(Automaton for Specification) For the chemical production line in Example 1.1, a recent study shows that the product B is toxic. For all the bottles containing the product B, the special caps are necessary. Figure 1.2 describes the safe production specification where any bottle containing toxic product (the product B) does have a special cap when the production is over.*

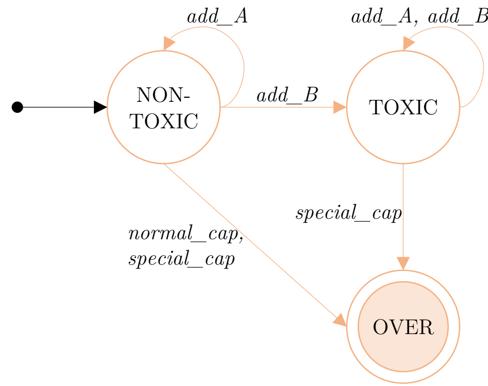


Figure 1.2: An FSA for a Safe Production Specification of the System in *Example 1.1*

If all the words that are accepted by the system model are also accepted by the given specification, then the system meets the given specification. In other words, if we can prove that the language of the system model is included in the language of the specification, then the verification problem is solved. Hence, the verification problem is a **language inclusion** problem.

There exists a classical solution (*Figure 1.3*) to solve the language inclusion problem. Instead of checking language inclusion between two automata A and B over an alphabet Σ, we firstly build a new automaton $\overline{B}$, called the complement of B, whose language is the complement of the language of B over the set of all available words Σ^*, so $\mathcal{L}(\overline{B}) = \Sigma^* - \mathcal{L}(B)$; and then we check whether the intersection between $\mathcal{L}(A)$ and $\mathcal{L}(\overline{B})$ is empty, so check if $\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) = \emptyset$.

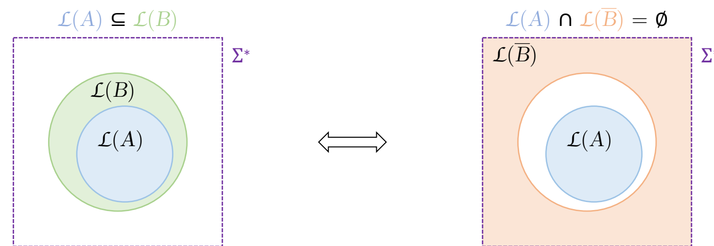


Figure 1.3: A Technique to Solve the Language Inclusion Problem

Hence, the verification problem in which we check whether a system $\mathcal{S}$ meets a given property $\mathcal{P}$ (hence check whether $\mathcal{S} \models \mathcal{P}$), can be transformed into an **emptiness problem** of the intersection between (i) the language of system model $\mathcal{L}(\mathcal{M}_\mathcal{S})$ and (ii) the complement of the language of the property $\mathcal{L}(\overline{\mathcal{M}_\mathcal{P}})$, so check whether $\mathcal{L}(\mathcal{M}_\mathcal{S}) \cap \mathcal{L}(\overline{\mathcal{M}_\mathcal{P}}) = \emptyset$. *Figure 1.4* shows the transformation of the system verification problem into an emptiness problem.

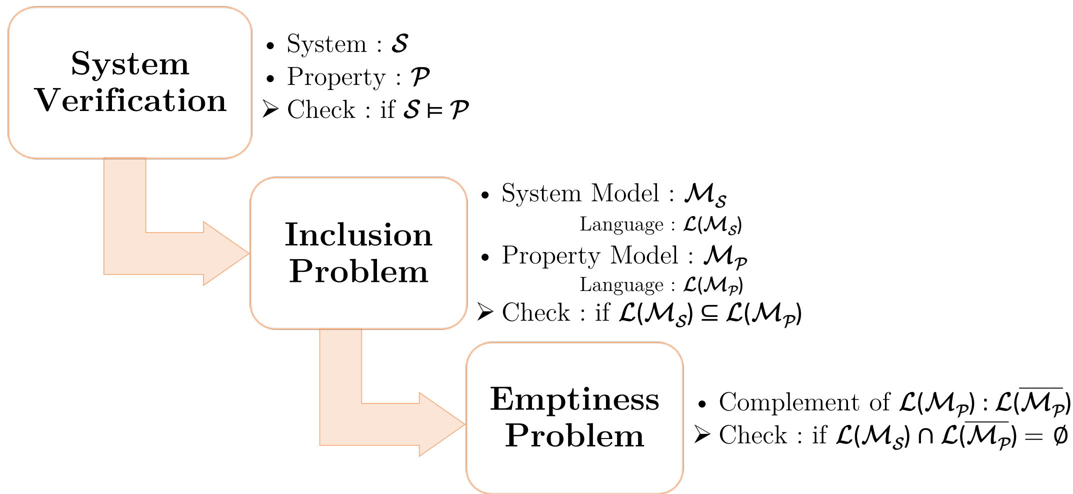


Figure 1.4: From System Verification to Emptiness Problem

1.2 Challenges

1.2.1 Non-Determinism

A finite state automaton is called a **deterministic finite automaton (DFA)** if each of its transitions is uniquely determined by its source state and input symbol, and reading an input symbol is required for each transition. Hence, for any input, the deterministic finite state automaton produces a unique computation[1]. A **non-deterministic finite automaton (NFA)**[2] does not need to obey the restrictions above. In other words, for any non-deterministic finite automaton, from a given state, if we take an input symbol, there can be several possible next states. *Section 2.2* provides more details about NFA and DFA.

If we transform the verification problem into an emptiness problem (*Figure 1.4*), then we have to complement the automaton of the specification. Complementing a DFA can be simply done by just flipping the final states and the non-final states, but if the automaton is non-deterministic, then this method does not work.

Example 1.3 *If we complement the NFA in Figure 1.5.left by just flipping the final states and the non-final states, then we obtain the NFA in Figure 1.5.right. But both of them accept the word "a", hence the complementation in this way is not correct.*

[1]Some input word can block the computation, but the computation is still unique for DFA.

[2]Theoretically, any DFA is also a NFA. But here in this chapter, we use "NFA" in a narrower sense, referring to those NFA who are not DFA.

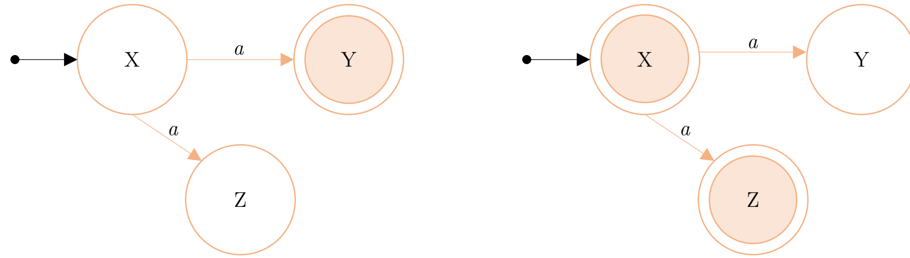


Figure 1.5: Complement an NFA in a Wrong Way

One classical technique to deal with the non-determinism is converting a non-deterministic automaton into a deterministic automaton that recognises the same language [52], by power-set construction (also called subset construction). However, if a non-deterministic automaton has n states, then the resulting deterministic automaton by subset construction may have up to 2^n states, an **exponentially larger** number, which makes the construction impractical for large automata.

Example 1.4 *Figure 1.6.a is an NFA with 3 states. Figure 1.6.b is a DFA that recognises the same language as Figure 1.6.a does, but it has 8 states, which is exponentially larger.*

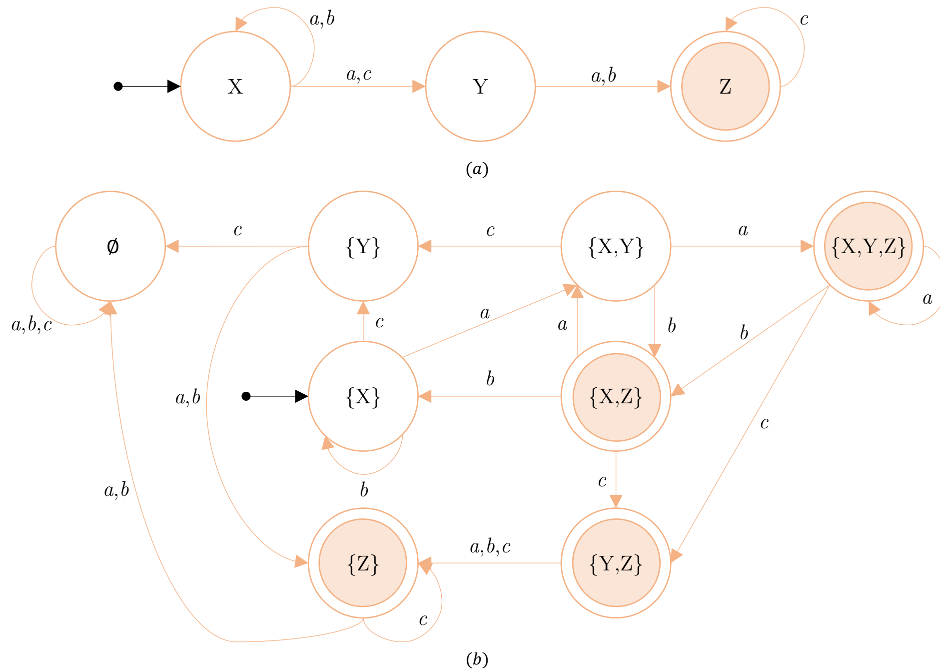


Figure 1.6: A 3-State NFA to an 8-State DFA by Subset Construction

1.2.2 Infinite Alphabets

When dealing with real-life systems, the models usually handle data from very large domains that can be assumed to be infinite, such as 64-bit integers, floating point numbers, strings of characters, etc. The correctness of this kind of systems must be specified in terms of data values. Alternatively, sometimes the systems must respond to strict deadlines, which requires temporal specifications expressed in terms of timed languages [4].

Example 1.5 *Giving two integer arrays X and Y, as in Figure 1.7.left: (i) $X_0 = 1$; (ii) $Y_0 = 1$; (iii) $\forall i > 0 : X_i = Y_{i-1}, Y_i = X_{i-1} + Y_{i-1}$. We want to check whether these two arrays meet the specification, as in Figure 1.7.right: $\forall i > 0 : X_i > X_{i-1}, Y_i > Y_{i-1}$. Here, using finite alphabets is not enough since integer is an infinite data domain.*

	0	1	2	3	...
X	1	1	2	3	...
Y	1	2	3	5	...

	...	i - 1	i	i + 1	...
X	...	$>X_{i-2}$	$>X_{i-1}$	$>X_i$	...
Y	...	$>Y_{i-2}$	$>Y_{i-1}$	$>Y_i$	...

Figure 1.7: Arrays and Specification in *Example 1.5*

Example 1.6 *Figure 1.8 shows a timed system with two clocks x and y, which increase jointly as time elapses. Clocks can be reset to 0 and restart timing at occurrence of a transition (supposed to take no time), and then again, they will increase with time. The clock values are real numbers.*

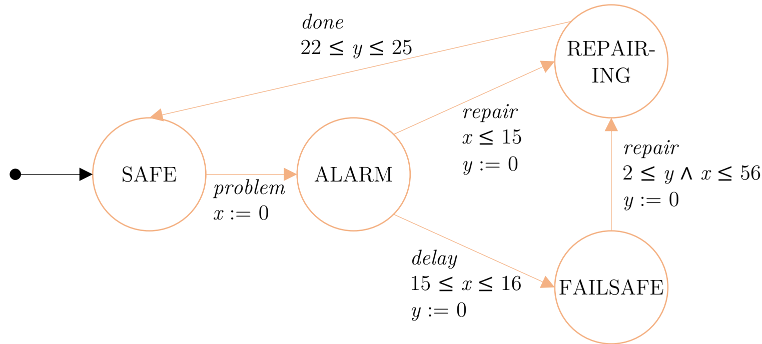


Figure 1.8: A Timed System

There exist some classical automata that can, to some extent, handle infinite alphabets, such as timed automata [4] and finite-memory (register) automata [36]. But they both face the **closure problem** for the complementation due to their infinite alphabets. In other words, for these two kinds of automata, there exist automata for which the complement language cannot

be recognised by an automaton in the same class. This excludes the possibility to transform the inclusion problem into the emptiness problem.

1.3 State-of-the-Art

1.3.1 Solutions for Non-Determinism

As mentioned before, the NFA may need to be determinised in order to be complemented. However, this determinisation may cause an exponential blow-up in the number of states. This is the context in which alternation [12] has been introduced.

One classical alternating model is **alternating finite automata (AFA)** [12], where the transitions are divided into existential (OR-relation, disjunctive branching) transitions and universal (AND-relation, conjunctive branching) transitions. In AFA, we also allow the formulae $true$ and $false$. We will introduce more about AFA in *Section 2.4*, and note that, the complementation of an AFA can be done in linear time, since (i) we flip the final states and the non-final states and this operation is linear; (ii) we flip $\wedge$ and $\vee$ in the transition rules and this operation is linear.

Example 1.7 *The automaton in Figure 1.9 is an alternating finite automaton. The transition rules are: (i)* $X \xrightarrow{a} Y \wedge Z$ *(ii)* $Y \xrightarrow{a} Y$ *(iii)* $Y \xrightarrow{b} Y$ *(iv)* $Z \xrightarrow{b} X \vee Y$*.*

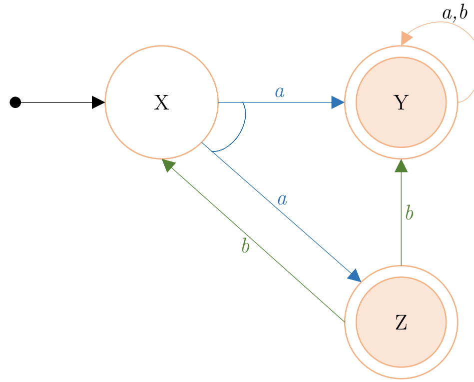


Figure 1.9: An Alternating Finite Automaton

1.3.2 Solutions for Infinite Alphabets

Using finite alphabets for the models and the specifications is very restrictive when dealing with real-life systems. However, there exist some classical models that can, to some extent, handle infinite alphabets.

One classical model is **timed automata** [4], which can capture several interesting aspects of real-time systems, including some qualitative features such as liveness, fairness and non-determinism, as well as some quantitative features such as periodicity, bounded response and timing delays. Timed automata accept timed words - possibly infinite sequences in which a real-valued time of occurrence is associated with each symbol. A timed automaton is a finite automaton with a finite set of real-valued clocks. The clocks can be reset to 0 independently of each other with the transitions of the automaton, and keep track of the time elapsed since the last reset. The transitions of the automaton may impose certain constraints on clock values, such that a transition may be taken only if the current values of the clocks satisfy the associated constraints. Language inclusion is generally undecidable[3] for timed automata. Moreover, the class of timed regular languages is not closed under complementation, this excludes the possibility to transform the inclusion problem into an emptiness problem.

Example 1.8 *The timed automaton in Figure 1.10 accepts the language:*

$$\{(a^\omega, \tau) \mid \exists i \geq 1.\exists j > i.(\tau_j = \tau_i + 1)\}$$

where a^ω *stands for the infinite concatenation of* a *to itself and* τ *is a time sequence (hence a sequence of real numbers). The complement of this language cannot be characterised using a timed automaton. The complement needs to make sure that no pair of* a *is separated by distance* 1*. Since there is no bound on the number of* a *that can happen in a time period of length* 1*, keeping track of the times of all the* a *within the past* 1 *time unit would require an unbounded number of clocks.*

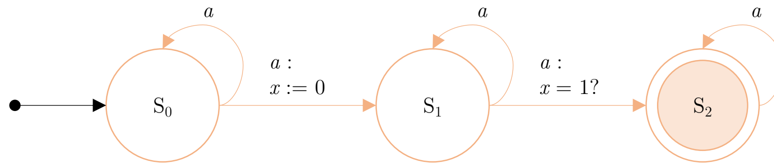


Figure 1.10: A Non-Complementable Timed Automaton

Another model of computation dealing with infinite alphabets, called **finite-memory automata** [36], is a natural generalisation of the classical finite-state automata [52]. This model is also called **register automata**. The basic idea behind this model is to equip the automaton with a finite set of registers, called windows. Each window is capable of being empty or storing a symbol from the infinite alphabet. When the automaton takes the next input symbol: (i) if no window contains the input symbol, then it is copied into a specified window depending on the state; (ii) otherwise the test of equality applies. The language inclusion is undecidable for finite-memory automata (register automata). Moreover, this model is generally not closed under complementation, which makes it impossible to transform the inclusion problem into emptiness problem.

[3] It becomes decidable if restricted to having at most one clock.

Example 1.9 *The finite-memory automaton in Figure 1.11 accepts the language:*

$$\{\sigma_1, \sigma_2, ..., \sigma_n : \exists 1 \leq i \leq j \leq n. \sigma_i = \sigma_j\}$$

The complement of this language cannot be characterised using a finite-memory automaton. Assume to the contrary that there exists a finite-memory automaton $\mathcal{A}$ that accepts the complement of the language above, hence all the words where each symbol appears at most once. Since the alphabet Σ is infinite, there exists a word $\sigma \in \mathcal{L}(\mathcal{A})$ of length $|\mathcal{A}| + 1$. We know that $\mathcal{A}$ accepts a word σ' of length $|\mathcal{A}| + 1$ that contains at most $|\mathcal{A}|$ distinct symbols (see proposition 4 in [36]*), therefore some symbol of Σ must appear in σ' more than once, in contradiction with the assumption.*

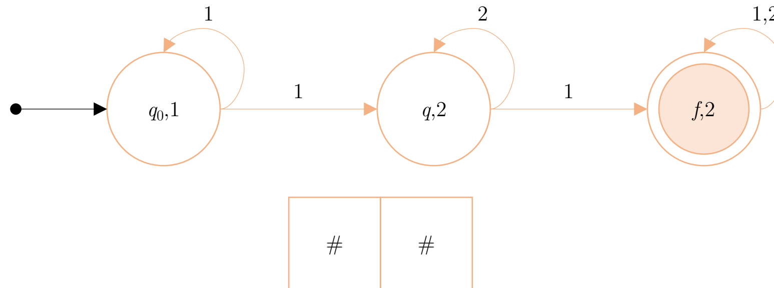


Figure 1.11: A Non-Complementable Finite-Memory Automaton

Symbolic finite automata (s-FA) [19, 58] are a model that can also, to some extent, overcome the limitation of only handling finite and small alphabets. Symbolic automata allow transitions to carry predicates and functions over a specified alphabet theory, such as linear arithmetic, and therefore extend finite automata to operate over infinite alphabets, such as the set of rational numbers. If it is decidable to check whether predicates in the algebra are satisfiable, then (i) symbolic automata are closed under Boolean operations and (ii) emptiness and inclusion are decidable. However, this model loses the previous values after each transition since the values cannot be stored in registers or other forms of memory. This excludes the possibility of comparing the current values with the past values.

Example 1.10 *The symbolic automaton in Figure 1.12 defines the list of odd numbers with length greater than 1.*

Data automata (DA) [34] are extensions of non-deterministic finite automata (NFA) with variables ranging over a possibly infinite data domain, equipped with a first-order theory. DA model recognises the data words over infinite alphabets consisting of pairs (a, v) where a is an input event from a finite set and v is a valuation of a finite set of variables that range over a possibly infinite data domain. Data automata are closed under the Boolean operations of intersection and complement, and these Boolean operations can be done in linear time. However, the inclusion problem for data automata is undecidable. We introduce more details about data automata in *Section 2.5*.

Example 1.11 *Consider the data automaton in Figure 1.13. There are two transition rules: (i)* $P \xrightarrow{a, x'=0 \wedge v'=0} Q$ *and (ii)* $Q \xrightarrow{b, x'=v+1 \wedge v'=x} Q$*, where* x *refers to the current values of* x *and* x' *refer to the next (new) value of* x *(idem for* v *and* v'*).*

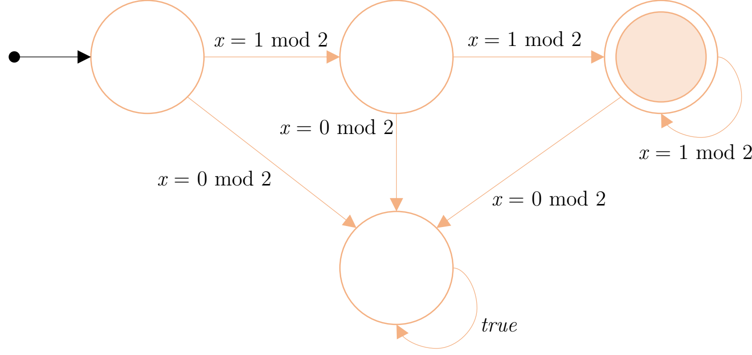


Figure 1.12: A Symbolic Automaton

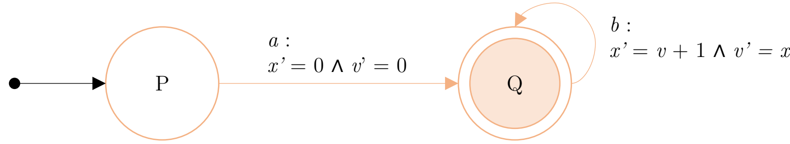


Figure 1.13: A Data Automaton

1.3.3 Solutions for both Non-Determinism and Infinite Alphabets

There exists an alternating model called **predicate automata (PA)** [26, 27, 39], in the class of infinite-state automata which recognise languages over an infinite alphabet. This model is used in some works on verification of parameterised concurrent programs with shared memory. In this model, the alphabet consists of pairs of program statements and thread identifiers, thus being infinite because the number of threads is potentially unbounded. The data theory in PA is the theory of equality because thread identifiers can only be compared for equality or dis-equality. The emptiness problem is undecidable when either (i) the predicates have arity greater than one, or (ii) some transition rule is quantified. Checking emptiness of quantifier-free PA is possible with some semi-algorithms, by explicitly enumerating reachable configurations and checking coverage by looking for permutations of argument values. However, no semi-algorithm exists for quantified PA.

Another alternating model that can, to some extent, handle infinite alphabets, is **symbolic alternating finite automata (s-AFA)** [18]. The two key-features of s-AFA are that: (i) the alphabet is symbolic, as in a symbolic finite automaton (s-FA) [19]; (ii) the automaton may

make use of both existential and universal non-determinism, as in an alternating finite automaton (AFA). For a normal s-AFA, the complementation can be done in linear time, however, for any given s-AFA, the normalisation [18, 44, 61] which aims at converting an s-AFA into an equivalent normal s-AFA, may (in the worst case) cause an exponential blow-up in the number of outgoing transitions of any one state in an s-AFA.

1.3.4 Solutions for Language Inclusion

Antichains [23] algorithms or semi-algorithms have been implemented for automata on finite words [20], on finite trees [9], on infinite words [24, 29], and for other applications where exponential constructions are involved such as model-checking of linear-time logic [21], games of imperfect information [13, 7], and synthesis of linear-time specifications [28].

The idea is always to exploit the special structure of the subset constructions. For example, consider the classical subset construction for the complementation of automata on finite words. States of the complement automaton are sets of states of the original automaton, that we call cells and denote by s_i. Set inclusion between cells is a partial order that turns out to be a simulation relation for the complement automaton: if $s_2 \subseteq s_1$ and there is a transition from s_1 to s_3, then there exists a transition from s_2 to some $s_4 \subseteq s_3$. This structural property carries over to the sets of cells manipulated by reachability algorithms: if $s_2 \subseteq s_1$ and a final cell can be reached from s_1, then a final cell can be reached from s_2. Therefore, in a breadth-first search algorithm with backward state traversal, if s_1 is visited by the algorithm, then s_2 is visited simultaneously; the algorithm manipulates $\subseteq$-downward closed sets of cells that can be canonically and compactly represented by the antichain of their $\subseteq$-maximal elements.

There exists also a semi-algorithm described in [34], which combines the principle of the antichain-based language inclusion algorithm [2] with the interpolant-based abstraction refinement semi-algorithm [45] via a general notion of language-based subsumption relation. This method aims at solving the **trace inclusion** problem (an instance is shown in *Figure 1.14*).

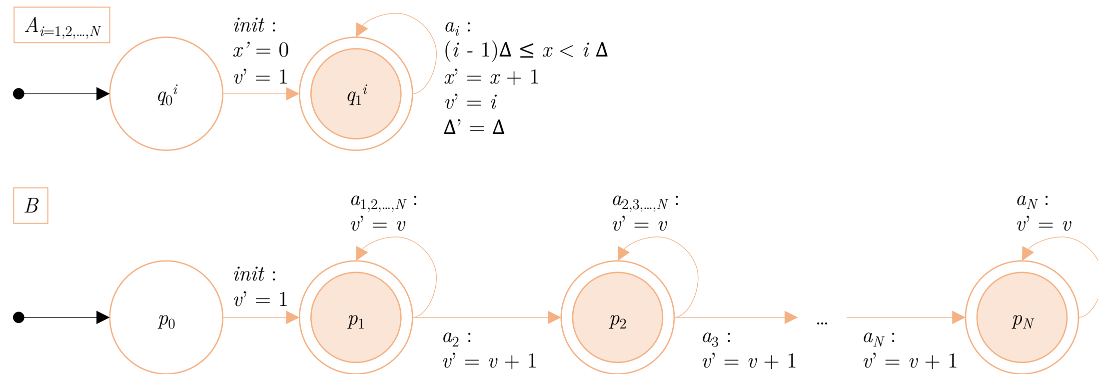


Figure 1.14: An Instance of the Trace Inclusion Problem

1.4 Contributions

1.4.1 Alternating Data Automata (ADA)

One of our contributions is a new model of alternating automata over infinite alphabets, called **alternating data automata (ADA)**. Inspired by the data automata (DA) model [8] and related studies [22, 34], we extend the DA model to the alternating automata model [60] where the control states become Booleans and the transition rules are specified by a set of formulae in a combined first-order theory of states (Booleans) and data that relate past values of variables with current values of variables. As the DA model does, the ADA model recognises the data words over infinite alphabets consisting of pairs (a, v) where a is an input event from a finite set and v is a valuation of a finite set of variables that range over a possibly infinite data domain.

Example 1.12 *Consider the alternating data automaton in Figure 1.15. The transitions are: (i)* $X \xrightarrow{a} Y \wedge x = 0 \wedge Z \wedge y = 0$ *(ii)* $Y \xrightarrow{a} Y$ *(iii)* $Y \xrightarrow{b} Y$ *(iv)* $Z \xrightarrow{b} X \wedge y = \overline{x} + 1 \vee Y \wedge x = \overline{y}$*, where* X*,* Y *and* Z *are Boolean control states,* x *refers to the current value of* x *and* $\overline{x}$ *refers to the past value of* x *(idem for* y *and* $\overline{y}$*).*

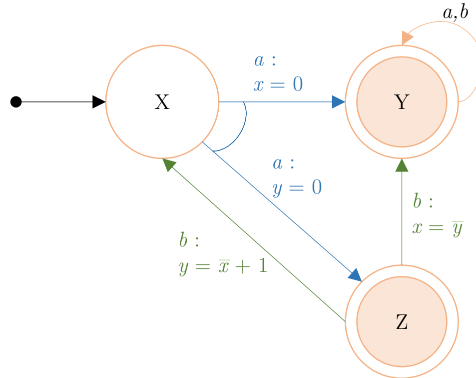


Figure 1.15: An Alternating Data Automata

With the ADA model, Boolean operations of union, intersection and complement can be done in linear time, thus matching the complexity of performing these operations in the finite-alphabet case. The price to be paid here is that emptiness checking becomes undecidable, this is the reason why we provide two efficient semi-algorithms for emptiness checking. One of these two semi-algorithms is based on lazy predicate abstraction [33]; and the other is based on the IMPACT method [45]. These two semi-algorithms are proven to terminate by returning a word from the language of the automaton if one exists. But if the language of the given automaton is empty, then termination is not guaranteed.

A restriction of the ADA model here is that there is no hidden variable, hence all the data values taken by the variables are visible in the input.

More details about ADA are given in *Chapter 3*.

1.4.2 First-Order Alternating Data Automata (FOADA)

Another contribution of this thesis is a generalised alternating automata model, called **first-order alternating data automata (FOADA)**, in which states are predicate symbols, the input is associated with data variables ranging over an infinite data domain and transitions use formulae in the first-order theory of the data domain. As the ADA model does, the FOADA model also recognises the data words over infinite alphabets consisting of pairs (a, v) where a is an input event from a finite set and v is a valuation of a finite set of variables that range over a possibly infinite data domain.

In the FOADA model, the arguments of a predicate atom track the values of the internal variables associated with the state, and these values are invisible in the input sequence. This overcomes the restriction of ADA, and it solves a classical language inclusion problem $\cap_{i=1}^{n}\mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{B})$, between FSA with data variables whose languages are alternating sequences of input events and variable valuations [34], where the variables of the right-hand side automaton $\mathcal{B}$ are also controlled by the left-hand side automaton $\mathcal{A}$, in other words, that $\mathcal{B}$ has no hidden variables.

Example 1.13 *Here is an example of FOADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$:*

- $D = \mathbb{Z}$, $\Sigma = \{a, b\}$, $X = \{x, y\}$, $Q = \{q_0, q_1, q_2\}$, $\iota = q_0(0)$, $F = \{q_2\}$,
- Δ *contains transitions:*

$$q_0(d) \xrightarrow{a(x,y)} q_1(x) \wedge x > d \wedge q_2(x, y) \wedge y > d,$$

$$q_1(d) \xrightarrow{b(x,y)} q_1(x) \wedge y < d \vee q_2(x, y) \wedge y > d,$$

$$q_2(d, e) \xrightarrow{b(x,y)} q_2(x, y) \wedge x > d \wedge y > e.$$

Note that d and e are not visible in the input.

The FOADA model is closed under union, intersection and complementation. Again, Boolean operations are possible in linear time. As the ADA model, the price here to be paid is that the emptiness checking of FOADA is undecidable, even for the simplest data theory of equality [26]. Hence, in this thesis we also introduce an effective emptiness checking semi-algorithm for FOADA model, in the spirit of the IMPACT procedure [45], originally developed for checking safety of non-deterministic integer programs.

More details about FOADA are given in *Chapter 4*.

1.4.3 FOADA Checker

For validation purposes, we also have developed a tool - **FOADA Checker** [62], mainly used for checking inclusion between two automata or checking emptiness of an automaton. The tool is written in Java, via Java-SMT interface [57] and using Z3 SMT solver [53] for checking spuriousness, coverage queries and interpolant generation. The IMPACT semi-algorithm has been implemented in the tool to check the emptiness of an automaton. The automata models supported as input are: (i) predicate automata [26, 27], (ii) alternating data automata and (iii) first-order alternating data automata.

More details about FOADA Checker are given in *Chapter 6*.

1.4.4 Applications

The main application of our models (ADA and FOADA) is checking inclusions between various classes of automata extended with variables ranging over infinite domains that recognise languages over infinite alphabets. The most widely known such classes are (i) **timed automata** [4] and (ii) **finite-memory automata** [36]. In both cases, complementation is not possible inside the class and inclusion is undecidable. Our contribution here is providing a systematic semi-algorithm for these decision problems. In addition, we can extend **generic register automata** [34] inclusion checking framework by allowing monitor (right-hand side) automata to have local (hidden) variables that are not visible in the language.

Another application is checking safety (mutual exclusion, absence of deadlocks, etc.) and liveness (termination, absence of starvation, etc.) properties of parameterised concurrent programs, consisting of an unbounded number of replicated threads that communicate via a fixed set of global variables (locks, counters, etc.). The verification of parametric programs has been reduced to checking the emptiness of a possibly infinite sequence of first-order alternating data automata, called **predicate automata** [26, 27], encoding the inclusion of the set of traces of a parametric concurrent program into increasingly general proof spaces, obtained by generalisation of counter-examples. The program and the proof spaces are first-order alternating data automata over the infinite alphabet of pairs consisting of program statements and thread identifiers.

1.5 Organisation

Chapter 2 presents some preliminaries, including some basics of the first order logic, some brief introductions to automata and alternating automata.

In *Chapter 3* we define **Alternating Data Automata (ADA)**. Then we introduce the closure properties and show how the Boolean operations on ADA can be done in linear time. After that, we introduce the anti-chains and interpolants for the emptiness of ADA. Based on this, we provide two efficient semi-algorithms for emptiness checking, inspired by two state-of-the-art abstraction refinement model checking methods: lazy predicate abstraction [33] and the

IMPACT semi-algorithm [45].

In *Chapter 4* we define **First-Order Alternating Data Automata (FOADA)**. Then we explain symbolic execution of FOADA. After that, we show that FOADA are closed under union, intersection and complementation. The emptiness problem for FOADA is undecidable, therefore we introduce an abstraction refinement semi-algorithm based on lazy annotation [45, 46] of the symbolic execution paths with interpolants obtained by (i) applying quantifier elimination with witness term generation and (ii) Lyndon interpolation in the quantifier-free theory of the data domain with uninterpreted predicate symbols.

The main applications of ADA and FOADA is checking inclusions between various classes of automata extended with variables ranging over infinite domains that recognise languages over infinite alphabets. *Chapter 5* shows the applications of our models for timed automata [4], register automata [36] and predicate automata [26, 27].

Chapter 6 explains our tool, the **FOADA Checker** [62]. The IMPACT semi-algorithm [45] described in previous chapters has been implemented in this tool. Some insightful case-studies can also be found in this chapter.

1.6 Notations

The following notations are frequently used throughout this thesis:

- $\mathbb{N}$: The symbol $\mathbb{N}$ denotes natural numbers;
- $\mathbb{Z}$: The symbol $\mathbb{Z}$ denotes integers;
- $\mathbb{R}$: The symbol $\mathbb{R}$ denotes real numbers;
- $|S|$: Given a set S, $|S|$ denotes the cardinality of S;
- $[a, b]$: Given two integers a and b such that $a \leq b$, $[a, b]$ denotes the integer set $\{i \in \mathbb{Z} \mid a \leq i \leq b\}$;
- A^B : Given two sets A and B, A^B denotes the set of all functions $f : A \to B$.

Chapter 2

Preliminaries

In this chapter, we introduce some basics that are important for the following chapters.

In the first section, we introduce the syntax of the *First-Order Logic (FOL)*, starting by terms and formulae; then we explain its semantics including interpretation and valuation. The second section introduces *Craig interpolation* and *Lyndon interpolation*. In the third section, we introduce some basics of automata on finite words. We start with *Non-Deterministic Finite Automata (NFA)*, then we introduce *Deterministic Finite Automata (DFA)* which is a particular case of NFA, and then we explain the transformation of an NFA into a DFA that accepts the same language, which is called determinisation. We terminate this section by explaining complementation of an NFA. In the forth section, we introduce alternation. We first present *Alternating Finite Automata (AFA)* model, then we show how to complement AFA in linear time.

2.1 First-Order Logic

2.1.1 Functions and Constants

Given a set of sort symbols $\Sigma^{\mathcal{S}}$, a **function symbol** $f^{\sigma_1,\sigma_2,\ldots,\sigma_{\#(f)}:\sigma_f}$ contains the following information:

- $\sigma_1, \sigma_2, ..., \sigma_{\#(f)} : \sigma_f$ is the function signature[1] where:
  - $\sigma_1, \sigma_2, ..., \sigma_{\#(f)} \in \Sigma^{\mathcal{S}}$ are the sorts of the function arguments;
  - $\sigma_f \in \Sigma^{\mathcal{S}}$ is the sort of the function result;
- $\#(f) \geq 0$ is the function arity.

[1] We omit specifying the signature of a function when it is not necessary.

For a function f with result sort σ_f, if $\#(f) = 0$ then f is called a **constant**, and denoted by f^{σ_f} or simply[2] f. For the Boolean sort $Bool = \{\top, \bot\} \in \Sigma^{\mathcal{S}}$, we write $\top$ for the constant $true$ and $\bot$ for the constant $false$.

2.1.2 Terms

Given a set of sort symbols $\Sigma^{\mathcal{S}}$, a set of function symbols $\Sigma^{\mathcal{F}}$ and a countable set of variables VAR where each variable $x^{\sigma_x} \in VAR$ (simply[3] denoted as x) has an associated sort $\sigma_x \in \Sigma^{\mathcal{S}}$, a **term** t of sort $\sigma_t \in \Sigma^{\mathcal{S}}$, denoted[4] by t^{σ_t}, is defined recursively by the grammar:

$t ::= x^{\sigma_t},$	$x^{\sigma_t} \in VAR;$	variable
$\mid c^{\sigma_t},$	$c^{\sigma_t} \in \Sigma^{\mathcal{F}};$	constant
$\mid f^{\sigma_1,\sigma_2,\ldots,\sigma_{\#(f)}:\sigma_t}(t_1^{\sigma_1}, t_2^{\sigma_2}, \ldots, t_{\#(f)}^{\sigma_{\#(f)}}),$	$f^{\sigma_1,\sigma_2,\ldots,\sigma_{\#(f)}:\sigma_t} \in \Sigma^{\mathcal{F}},$	
	$\sigma_1, \sigma_2, \ldots, \sigma_{\#(f)} \in \Sigma^{\mathcal{S}},$	
	$t_1^{\sigma_1}, t_2^{\sigma_2}, \ldots, t_{\#(f)}^{\sigma_{\#(f)}}$ are terms;	function application

2.1.3 Formulae

Given a set of sort symbols $\Sigma^{\mathcal{S}}$, a **first-order formula** ϕ is defined recursively by the grammar:

$\phi ::= t^{Bool},$	t^{Bool} is Boolean term;	Boolean term
$\mid t_1^{\sigma_1} \approx t_2^{\sigma_2},$	$\sigma_1, \sigma_2 \in \Sigma^{\mathcal{S}},$	
	$t_1^{\sigma_1}, t_2^{\sigma_2}$ are terms;	equality
$\mid \neg\psi,$	ψ is first-order formula;	negation
$\mid \psi_1 \wedge \psi_2,$	ψ_1, ψ_2 are first-order formulae;	conjunction
$\mid \exists x.\psi,$	ψ is first-order formula,	
	$x \in FV(\psi);$	existential quantification

In addition, we can write:

$\psi_1 \vee \psi_2$	for	$\neg(\neg\psi_1 \wedge \neg\psi_2),$	ψ_1, ψ_2 are first-order formulae;	disjunction
$\psi_1 \rightarrow \psi_2$	for	$\neg\psi_1 \vee \psi_2,$	ψ_1, ψ_2 are first-order formulae;	implication
$\forall x.\psi$	for	$\neg(\exists x.\neg\psi),$	ψ is first-order formula,	
			$x \in FV(\psi);$	universal quantification

For a formula ϕ, we denote by $FV(\phi)$ the set of variables not occurring under the scope of a quantifier in ϕ. It is also called the set of **free variables**.

[2] We omit specifying the sort of a constant when it is not necessary.
[3] We omit specifying the sort of a variable if it is not necessary.
[4] We omit specifying the sort of a term when it is not necessary.

2.1.4 Interpretation and Valuation

Given a set of sort symbols $\Sigma^{\mathcal{S}}$ and a set of function symbols $\Sigma^{\mathcal{F}}$, an **interpretation** $\mathcal{I}$ for $(\Sigma^{\mathcal{S}}, \Sigma^{\mathcal{F}})$ maps each:

- Sort symbol $\sigma \in \Sigma^{\mathcal{S}}$: to a non-empty set $\sigma^{\mathcal{I}}$;
- Function symbol $f^{\sigma_1,\sigma_2,...,\sigma_{\#(f)}:\sigma_f} \in \Sigma^{\mathcal{F}}$ with $\#(f) > 0$ where $\sigma_1, \sigma_2, ..., \sigma_{\#(f)}, \sigma_f \in \Sigma^{\mathcal{S}}$: to a function $f^{\mathcal{I}} : \sigma_1^{\mathcal{I}} \times \sigma_2^{\mathcal{I}} \times ... \times \sigma_{\#(f)}^{\mathcal{I}} \to \sigma^{\mathcal{I}}$;
- Constant symbol $c^{\sigma} \in \Sigma^{\mathcal{F}}$ where $\sigma \in \Sigma^{\mathcal{S}}$: to an element of $\sigma^{\mathcal{I}}$.

Given an interpretation $\mathcal{I}$, the set of all possible valuations under $\mathcal{I}$ is denoted by $\mathcal{V}_{\mathcal{I}}$. A **valuation** $v \in \mathcal{V}_{\mathcal{I}}$ maps each variable $x^{\sigma_x} \in VAR$ to an element of $\sigma_x^{\mathcal{I}}$. Given in addition a value $\alpha \in \sigma_x^{\mathcal{I}}$, we write $v[x \leftarrow \alpha]$ for a valuation such that: (i) $v[x \leftarrow \alpha](x) = \alpha$, and (ii) $v[x \leftarrow \alpha](y) = v(y)$ for any $y \in VAR$ with $y \neq x$.

2.1.5 Interpretation of Terms

Given an interpretation $\mathcal{I}$ and a valuation $v \in \mathcal{V}_{\mathcal{I}}$, the **interpretation of a term** t, denoted by $t_v^{\mathcal{I}}$, is defined recursively:

$$\begin{aligned} x_v^{\mathcal{I}} &= v(x), && x \in VAR; \\ c_v^{\mathcal{I}} &= c^{\mathcal{I}}, && c \in \Sigma^{\mathcal{F}}; \\ f_v^{\mathcal{I}}(t_1, t_2, ..., t_{\#(f)}) &= f^{\mathcal{I}}({t_1}_v^{\mathcal{I}}, {t_2}_v^{\mathcal{I}}, ..., {t_{\#(f)}}_v^{\mathcal{I}}), && f \in \Sigma^{\mathcal{F}}, t_1, t_2, ..., t_{\#(f)} \text{ are terms;} \end{aligned}$$

2.1.6 Semantics of Formulae

Given an interpretation $\mathcal{I}$ and a valuation $v \in \mathcal{V}_{\mathcal{I}}$, we write $\mathcal{I}, v \models \phi$ if the first-order formula ϕ is interpreted to *true* under $\mathcal{I}$ and v. We have the following recursive definitions:

$$\begin{array}{lll} \mathcal{I}, v \models t^{Bool} & \text{iff } {t^{Bool}}_v^{\mathcal{I}} = \top, & t^{Bool} \text{ is Boolean term;} \\ \mathcal{I}, v \models t_1 \approx t_2 & \text{iff } {t_1}_v^{\mathcal{I}} = {t_2}_v^{\mathcal{I}}, & t_1, t_2 \text{ are terms;} \\ \mathcal{I}, v \models \neg\psi & \text{iff } \mathcal{I}, v \not\models \psi, & \psi \text{ is first-order formula;} \\ \mathcal{I}, v \models \psi_1 \wedge \psi_2 & \text{iff } \mathcal{I}, v \models \psi_1 \text{ and } \mathcal{I}, v \models \psi_2, & \psi_1, \psi_2, \text{ are first-order formulae;} \\ \mathcal{I}, v \models \exists x^{\sigma}.\psi & \text{iff } \mathcal{I}, v[x^{\sigma} \leftarrow \alpha] \models \psi \text{ for some } \alpha \in \sigma^{\mathcal{I}}, & \psi \text{ is first-order formula}, x^{\sigma} \in FV(\psi); \end{array}$$

A first-order formula ϕ is **satisfiable** under the interpretation $\mathcal{I}$ if there exists a valuation v such that $\mathcal{I}, v \models \phi$, otherwise ϕ is **unsatisfiable** under $\mathcal{I}$. If $\mathcal{I}, v \models \phi$ for any v under $\mathcal{I}$, then ϕ is **valid** under $\mathcal{I}$.

Given two formulae ϕ and ψ, we write $\phi \models^{\mathcal{I}} \psi$ and say that ϕ **entails** ψ under the interpretation $\mathcal{I}$, if and only if $\mathcal{I}, v \models \phi$ implies $\mathcal{I}, v \models \psi$ for any valuation v.

2.2 Interpolation

2.2.1 Craig's Interpolation

Given a formula ϕ, the **vocabulary** of ϕ, denoted $V(\phi)$, is the set of predicate symbols and variables occurring in ϕ. For a term t, its vocabulary $V(t)$ is the set of variables that occur in t. Observe that quantified variables and the interpreted function symbols of the data theory do not belong to the vocabulary of a formula.

Definition 2.1 [16, 17] *For two formulae A and B such that $A \models B$, a **Craig interpolant** is a formula I such that: (i) $A \models I$, (ii) $I \models B$ and (iii) $V(I) \subseteq V(A) \cap V(B)$.*

Definition 2.2 *For two formulae A and B, suppose the conjunction $A \wedge B$ is unsatisfiable, a **reverse interpolant** is a formula I such that: (i) $A \models I$, (ii) $I \wedge B$ is unsatisfiable and (iii) $V(I) \subseteq V(A) \cap V(B)$.*

2.2.2 Lyndon's Interpolation

Lyndon's interpolation theorem [43] is a stronger form of Craig's interpolation theorem. By $P^+(\phi)$ we denote the set of predicate symbols that occur in ϕ under an even number of negations and by $P^-(\phi)$ we denote the set of predicate symbols that occur in ϕ under an odd number of negations.

Definition 2.3 [43] *Given two formulae A and B such that $A \wedge B$ is unsatisfiable, a **Lyndon interpolant** is a formula I such that: (i) $A \models I$, (ii) $I \wedge B$ is unsatisfiable and (iii) $V(I) \subseteq V(A) \cap V(B)$, $P^+(I) \subseteq P^+(A) \cap P^+(B)$ and $P^-(I) \subseteq P^-(A) \cap P^-(B)$.*

2.3 Automata on Finite Words

2.3.1 Non-Deterministic Finite Automata (NFA)

A **non-deterministic finite automaton (NFA)** is a tuple $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ where:

- Σ is a finite input alphabet;
- Q is a finite set of states;

- $I \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is the set of final states;
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function.

Σ defines the symbols on which the automaton is defined. The set I defines the states in which the automaton may start, and I is possibly empty. The **transition function** δ can be identified with the relation $\rightarrow \subseteq Q \times \Sigma \times Q$ given by:

$$\text{for } q, p \in Q, a \in \Sigma : q \xrightarrow{a} p \text{ iff } p \in \delta(q, a)$$

Intuitively, $q \xrightarrow{a} p$ denotes that the automaton can move from the state q to the state p when reading the input a.

Example 2.1 *An example of NFA is depicted in Figure 2.1. Here in the AFA* $\mathcal{A} = (\Sigma, Q, I, F, \delta)$*:*

- $\Sigma = \{a, b\}$;
- $Q = \{q_0, q_1, q_2\}$;
- $I = \{q_0\}$;
- $F = \{q_2\}$;
- δ *is defined by:*

  $\delta(q_0, a) = \{q_0\}$, $\delta(q_0, b) = \{q_0, q_1\}$, $\delta(q_1, a) = \{q_2\}$, $\delta(q_1, b) = \{q_2\}$, $\delta(q_2, a) = \delta(q_2, b) = \emptyset$

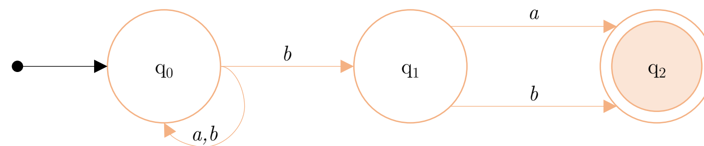


Figure 2.1: A Non-Deterministic Finite Automaton

2.3.2 Runs and Languages of NFA

Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an NFA and $w = a_1, a_2, ..., a_n \in \Sigma^*$ a finite word of length n. A **run** for w in $\mathcal{A}$ is a finite sequence of states $q_0, q_1, ..., q_n$ such that:

- $q_0 \in I$
- $q_i \xrightarrow{a_{i+1}} q_{i+1}$ for all $0 \leq i < n$

In an NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, there can be several runs for a given word $w \in \Sigma^*$ and the set of all possible runs for w in $\mathcal{A}$ is denoted by $\mathcal{R}_{\mathcal{A}}(w)$. A run $r = q_0, q_1, ..., q_n$ in $\mathcal{A}$ is called **accepting** if $q_n \in F$. In addition, a finite word $w \in \Sigma^*$ is called **accepted** by $\mathcal{A}$ if there exists an accepting run for w.

The **accepted language** of an NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of all words in Σ^* accepted by $\mathcal{A}$:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists r \in \mathcal{R}_{\mathcal{A}}(w).r \text{ is accepting}\}$$

For any NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, here we extend the transition function δ to the function $\delta^* : Q \times \Sigma^* \rightarrow 2^Q$ as follows:

- $\delta^*(q) = \{q\}$ for $q \in Q$;
- $\delta^*(q, a) = \delta(q, a)$ for $q \in Q$ and $a \in \Sigma$;
- $\delta^*(q, a_1, a_2, ..., a_n) = \bigcup_{p \in \delta(q, a_1)} \delta^*(p, a_2, ..., a_n)$ for $q, p \in Q$ and $a_1, a_2, ..., a_n \in \Sigma$ and $n \geq 2$.

Stated in words, given a state $q \in Q$ and a word $w \in \Sigma^*$, $\delta^*(q, w)$ is the set of states that are reachable from the state q for the input word w. Here we can represent the accepted language of a NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ by means of the extended transition function δ^*:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q_0 \in I.\delta^*(q_0, w) \cap F \neq \emptyset\}$$

2.3.3 Deterministic Finite Automata (DFA) and Determinisation

Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be a NFA. $\mathcal{A}$ is called a **deterministic finite automaton (ADA)** if $|I| \leq 1$ and $|\delta(q, a)| \leq 1$ for all states $q \in Q$ and all symbols $a \in \Sigma$. In other words, a NFA is a DFA if it has at most one initial state and for each symbol the successor state of each state is either uniquely defined or undefined.

A DFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is **total** if it has exactly one initial state and for each symbol the successor state of each state is uniquely defined, hence $|I| = 1$ and $|\delta(q, a)| = 1$ for all states $q \in Q$ and all symbols $a \in \Sigma$. Total DFA is often written in the form $\mathcal{A} = (\Sigma, Q, \iota, F, \delta)$ where ι stands for the unique initial state, and δ is a total transition function $\delta : Q \times \Sigma \rightarrow Q$. In addition, the extended transition function δ^* of a total DFA can be viewed as a total function $\delta^* : Q \times \Sigma^* \rightarrow Q$, which for a given state $q \in Q$ and a finite word $w \in \Sigma^*$, returns a unique state $p \in Q$ that is reached from the state q for the input word w, hence $\delta^*(q, w) = p$. So here particularly, the accepted language of a total DFA $\mathcal{A} = (\Sigma, Q, \iota, F, \delta)$ is given by:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(\iota, w) \in F\}$$

For a given NFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, we can construct a total DFA $\mathcal{A}_D = (\Sigma_D, Q_D, \iota_D, F_D, \delta_D)$ that accepts the same language, hence $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_D)$, by **power-set construction** (also called

subset construction), in which we simulate $\mathcal{A}$ by moving the prefixes of the given input word to the set of reachable states. This total DFA maybe exponentially larger than the original NFA:

- $\mathcal{A}_D$ starts in the state set I;
- If $\mathcal{A}_D$ is in a state set $Q' \subseteq Q$, then with the input symbol $a \in \Sigma$, $\mathcal{A}_D$ moves to another state set $Q'' = \bigcup_{q \in Q'} \delta(q, a)$;
- If the input word has been consumed and $\mathcal{A}_D$ is in a state set $Q' \subseteq Q$ that contains a state in F, then $\mathcal{A}_D$ accepts the input word.

More formally, we define $\mathcal{A}_D = (\Sigma_D, Q_D, \iota_D, F_D, \delta_D)$ as follows:

- $\Sigma_D = \Sigma$;
- $Q_D = 2^Q$;
- $\iota_D = I$;
- $F_D = \{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$;
- $\delta_D : 2^Q \times \Sigma \to 2^Q$ is defined by: $\delta_D(Q', a) = \bigcup_{q \in Q'} \delta(q, a)$ for $Q' \in Q_D$ and $a \in \Sigma$.

Example 2.2 *The NFA in Example 2.1 (depicted in Figure 2.1) in Page 31 is not deterministic as on input symbol* b *in state* q_0 *the next state is either* q_0 *or* q_1. *We apply power-set construction to obtain a DFA accepting same language and the result is depicted in Figure 2.2.*

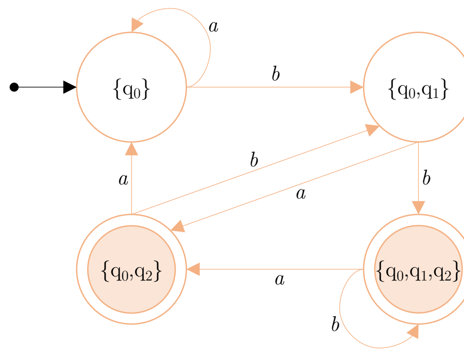


Figure 2.2: Determinisation of NFA in *Example 2.1* in *Page 31*

2.3.4 Complementation of NFA

Since total DFA have exactly one run for each input word, **complementing** a total DFA is simple, by just declaring all the non-final states to be final and all the final states to be non-final. This defines again a total DFA that accepts the complement of the language of the original DFA under the same alphabet. More formally, given a total DFA $\mathcal{A} = (\Sigma, Q, \iota, F, \delta)$, then $\overline{\mathcal{A}} = (\Sigma, Q, \iota, Q \setminus F, \delta)$ is a total DFA with $\mathcal{L}(\overline{\mathcal{A}}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$.

For any given NFA $\mathcal{A}$ over an alphabet Σ, we can first transform it into a total DFA $\mathcal{A}_D$ by power-set construction, and complement $\mathcal{A}_D$ to obtain $\overline{\mathcal{A}_D}$. $\overline{\mathcal{A}_D}$ accepts the complement of the language of $\mathcal{A}$, hence $\mathcal{L}(\overline{\mathcal{A}_D}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$.

Example 2.3 *Considering the total DFA in Figure 2.2, we declare all its non-final states to be final, hence* $\{q_0\}$ *and* $\{q_0, q_1\}$ *become final states; and we declare all its final states to be non-final, hence* $\{q_0, q_2\}$ *and* $\{q_0, q_1, q_2\}$ *become non-final states. Then we obtain a DFA, depicted in Figure 2.3, which is the complement of the DFA in Figure 2.2, hence also the complement of the NFA in Figure 2.1 in Page 31.*

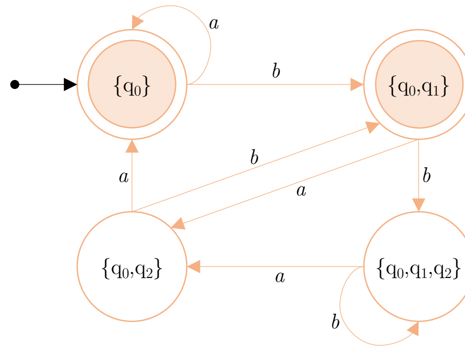


Figure 2.3: Complement of NFA in *Example 2.1* in *Page 31*

2.4 Alternating Finite Automata (AFA)

2.4.1 Definition of AFA

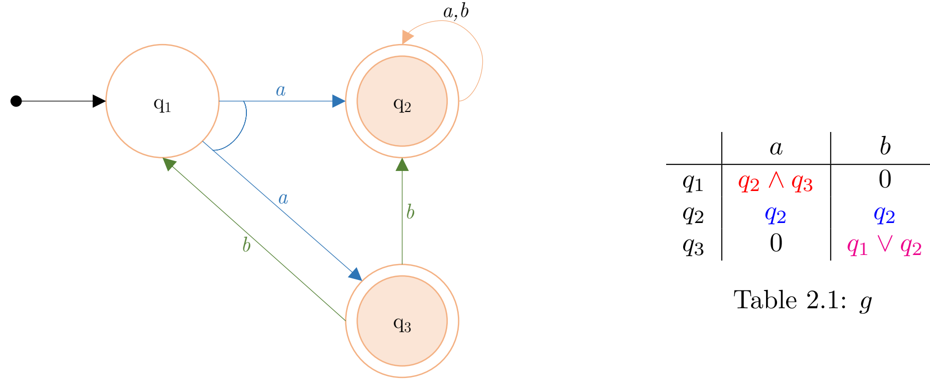
An **alternating finite automaton (AFA)** is a tuple $\mathcal{A} = (\Sigma, Q, \iota, F, g)$ where:

- Σ is a finite input alphabet;
- $Q = \{q_1, q_2, ..., q_{|Q|}\}$ is a finite set of states;
- $\iota \in Q$ is the initial states;
- $F \subseteq Q$ is the set of final states;

- $g : Q \to (\Sigma \times B^{|Q|} \to B)$ is the transition function where B denotes the Boolean set $\{0, 1\}$.

In an AFA $\mathcal{A} = (\Sigma, Q, \iota, F, g)$, the function g associates with each state $q \in Q$ a Boolean function $g(q) : \Sigma \times B^{|Q|} \to B$. Given an input symbol $a \in \Sigma$ and associating a Boolean value u_i with each of the $|Q|$ states q_i where $q_i \in Q$ for $i \in \mathbb{N}$ and $1 \leq i \leq |Q|$, then $g(q)$ computes a Boolean value $g(q)(a)(u_1, u_2, ..., u_{|Q|})$ to be associated with state q.

Example 2.4 *In Figure 2.4 we introduce alternation into FSA, where there are two types of alternating transitions: (i) a universal transition:* $q_1 \xrightarrow{a} q_2 \wedge q_3$*; (ii) an existential transition:* $q_3 \xrightarrow{b} q_1 \vee q_2$*. Formally, we define an AFA* $\mathcal{A} = (\Sigma, Q, \iota, F, g)$*, where* $\Sigma = \{a, b\}$*,* $Q = \{q_1, q_2, q_3\}$*,* $\iota = q_1$*,* $F = \{q_2, q_3\}$ *and* g *is given by Table 2.1. According to the definition of the function* g*, we can also build three* $\Sigma \times B^{|Q|} \to B$ *tables referring respectively to* $g(q_1)$*,* $g(q_2)$ *and* $g(q_3)$*.*



	a	b
q_1	$q_2 \wedge q_3$	0
q_2	q_2	q_2
q_3	0	$q_1 \vee q_2$

Table 2.1: g

Figure 2.4: An Automaton with Alternating Transitions

	a	b
$(0, 0, 0)$	0	0
$(0, 0, 1)$	0	0
$(0, 1, 0)$	0	0
$(0, 1, 1)$	1	0
$(1, 0, 0)$	0	0
$(1, 0, 1)$	0	0
$(1, 1, 0)$	0	0
$(1, 1, 1)$	1	0

Table 2.2: $g(q_1)$

	a	b
$(0, 0, 0)$	0	0
$(0, 0, 1)$	0	0
$(0, 1, 0)$	1	1
$(0, 1, 1)$	1	1
$(1, 0, 0)$	0	0
$(1, 0, 1)$	0	0
$(1, 1, 0)$	1	1
$(1, 1, 1)$	1	1

Table 2.3: $g(q_2)$

	a	b
$(0, 0, 0)$	0	0
$(0, 0, 1)$	0	0
$(0, 1, 0)$	0	1
$(0, 1, 1)$	0	1
$(1, 0, 0)$	0	1
$(1, 0, 1)$	0	1
$(1, 1, 0)$	0	1
$(1, 1, 1)$	0	1

Table 2.4: $g(q_3)$

2.4.2 Languages of AFA

Given an n-tuple $u = \langle u_1, u_2, ..., u_n \rangle$ of Boolean values, we define the projection function $\pi : [1, n] \to (B^n \to B)$ as follows:

$$\pi(i)(u) = u_i \text{ for } i \in \mathbb{N} \text{ and } 1 \le i \le n$$

With this projection, for an AFA $\mathcal{A} = (\Sigma, Q, \iota, F, g)$, we define f, the characteristic vector of F where:

$$\pi(i)(f) = \begin{cases} 1 & \text{if } q_i \in F \\ 0 & \text{if } q_i \notin F \end{cases} \text{ for } i \in \mathbb{N} \text{ and } 1 \le i \le |Q|$$

For any AFA $\mathcal{A} = (\Sigma, Q, \iota, F, g)$, we extend the function g to the function $g^* : Q \to (\Sigma^* \to (B^{|Q|} \to B))$ as follows:

- $g^*(q_i)(\lambda) = \pi(i)$ where $q_i \in Q$ and $\lambda \in \Sigma^*$ is the empty string;

- $g^*(q_i)(ax)(u) = g(q_i)(a, g^*(q_1)(x)(u), g^*(q_2)(x)(u), ..., g^*(q_{|Q|})(x)(u))$ where $q_i \in Q$, $a \in \Sigma$, $x \in \Sigma^*$, $u = \langle u_1, u_2, ..., u_{|Q|} \rangle$ and $u_j \in \{0, 1\}$ for $j \in \mathbb{N}$ and $1 \le j \le |Q|$.

Now with this function g^*, we can define the accepted words of the AFA. Let $\mathcal{A} = (\Sigma, Q, \iota, F, g)$ be an AFA, a word $w \in \Sigma^*$ is **accepted** by $\mathcal{A}$ if and only if:

$$g^*(\iota)(w)(f) = 1 \text{ where } f \text{ is the characteristic vector of } F$$

Example 2.5 *The word "ab" is accepted by the AFA in Example 2.4 and here is the proof:*

$$\begin{aligned}
g^*(q_1)(ab)(0,1,1) &= g(q_1)(a, g^*(q_1)(b)(0,1,1), g^*(q_2)(b)(0,1,1), g^*(q_3)(b)(0,1,1)) \\
&= g(q_1)(a, g(q_1)(b, g^*(q_1)(\lambda)(0,1,1), g^*(q_2)(\lambda)(0,1,1), g^*(q_3)(\lambda)(0,1,1)) \\
&\qquad g(q_2)(b, g^*(q_1)(\lambda)(0,1,1), g^*(q_2)(\lambda)(0,1,1), g^*(q_3)(\lambda)(0,1,1)) \\
&\qquad g(q_3)(b, g^*(q_1)(\lambda)(0,1,1), g^*(q_2)(\lambda)(0,1,1), g^*(q_3)(\lambda)(0,1,1))) \\
&= g(q_1)(a, g(q_1)(b, \pi(1)(0,1,1), \pi(2)(0,1,1), \pi(3)(0,1,1)) \\
&\qquad g(q_2)(b, \pi(1)(0,1,1), \pi(2)(0,1,1), \pi(3)(0,1,1)) \\
&\qquad g(q_3)(b, \pi(1)(0,1,1), \pi(2)(0,1,1), \pi(3)(0,1,1))) \\
&= g(q_1)(a, g(q_1)(b,0,1,1), g(q_2)(b,0,1,1), g(q_3)(b,0,1,1)) \\
&= g(q_1)(a, 0, 1, 1) \\
&= 1
\end{aligned}$$

The language accepted by an AFA is the set of all accepted words, hence for a given AFA $\mathcal{A} = (\Sigma, Q, \iota, F, g)$:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid g^*(\iota)(w)(f) = 1\} \text{ where } f \text{ is the characteristic vector of } F$$

2.5 Data Automata (DA)

2.5.1 Definition of DA

Data Automata (DA) are extensions of NFA with variables ranging over an infinite data domain $\mathcal{D}$, equipped with a first-order theory $\mathbb{T}(\mathcal{D})$.

Formally, a DA is a tuple $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$ where:

- $\mathcal{D}$ is a possibly infinite data domain;
- Σ is a finite alphabet of input events including a special padding symbol $\diamond \in \Sigma$;
- $X = \{x_1, x_2, ..., x_{|X|}\}$ is a set of variables;
- Q is a finite set of states;
- $\iota \in Q$ is the initial state;
- $F \subseteq Q$ is the set of final states;
- Δ is a set of rules of the form $q \xrightarrow{a, \phi(X,X')} q'$ where $a \in \Sigma$ is an input symbol and $\phi(X, X')$ is a formula in $\mathbb{T}(\mathcal{D})$.

A configuration of a DA $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$ is a pair $(q, v) \in Q \times \mathcal{D}^X$ and a configuration (q', v') is called a **successor** of (q, v) if and only if: (i) $\exists a \in \Sigma . q \xrightarrow{a,\phi} q' \in \Delta'$; (ii) $(v, v') \models_{\mathbb{T}(\mathcal{D})} \phi$.

We denote the successor relation by $(q, v) \xrightarrow{a,\phi} (q', v')$ and we omit writing ϕ when no confusion may arise. We denote by $succ(q, v) = \{(q', v') \mid (q, v) \rightarrow (q', v')\}$ the set of successors of a configuration (q, v).

2.5.2 Languages of DA

For a DA $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$, a **trace** is a finite sequence w of pairs (v_i, a_i) taken from $\mathcal{D}^X \times \Sigma$:

$$w = (v_0, a_0), (v_1, a_1), ..., (v_{n-1}, a_{n-1}), (v_n, \diamond)$$

Accordingly, a **run** of $\mathcal{A}$ over the trace $w = (v_0, a_0), (v_1, a_1), ..., (v_{n-1}, a_{n-1}), (v_n, \diamond)$ is a sequence of configurations π:

$$\pi = (q_0, v_0), (q_1, v_1), ..., (q_n, v_n) \text{ for each } i \in \mathbb{N}, 0 \leq i \leq n-1 : (q_i, v_i) \xrightarrow{a_i} (q_{i+1}, v_{i+1})$$

We say that π is **accepting** if and only if $q_n \in F$, in which case $\mathcal{A}$ **accepts** w. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of all traces accepted by $\mathcal{A}$.

2.5.3 Determinisation

Let $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$ be a DA, $\mathcal{A}$ is said to be **deterministic** if and only if, for each trace $w \in \mathcal{L}(\mathcal{A})$, $\mathcal{A}$ has at most one run over w. Any DA can be determinised while preserving its language. The reason why determinisation is possible for automata over an infinite data alphabet $\mathcal{D}^X \times \Sigma$ is that the successive values taken by each variable $x \in X$ are tracked by the language $\mathcal{L}(\mathcal{A}) \subseteq (\mathcal{D}^X \times \Sigma)^*$. But there is an example of classical automata over an infinite alphabet that cannot be determinised - timed automata [4], in which only the elapsed time is reflected in the language but not the values of the clocks.

The determinisation procedure is a generalisation of the classical subset construction for word automata [52] on finite alphabets. Formally, for a DA $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$, the deterministic data automata (DDA) accepting the language $\mathcal{L}(\mathcal{A})$ are defined as $\mathcal{A}_D = (\mathcal{D}, \Sigma, X, Q_D, \iota_D, F_D, \Delta_D)$:

- $\mathcal{D}$ is the same infinite data domain;
- Σ is the same finite alphabet of input events including a special padding symbol $\diamond \in \Sigma$;
- $X = \{x_1, x_2, ..., x_{|X|}\}$ is the same set of variables;
- $Q_D = 2^Q$;
- $\iota_D = \{\iota\}$;
- $F_D = \{P \subseteq Q \mid P \cap F = \emptyset\}$;
- Δ_D is the set of rules $P \xrightarrow{a,\theta} P'$ such that: (i) $\forall p' \in P'.\exists p \in P.p \xrightarrow{a} p' \in \Delta$; (ii) $\theta(X, X') \equiv \bigwedge_{p' \in P'} \bigvee_{p \xrightarrow{a,\psi} p' \in \Delta, p \in P, a \in \Sigma} \psi \wedge \bigwedge_{p' \in Q \setminus P'} \bigwedge_{p \xrightarrow{a,\phi} p' \in \Delta, p \in P, a \in \Sigma} \neg\phi$.

The main difference with the classical subset construction for Rabin-Scott automata is that here we consider all sets P' of states that have a predecessor in P, not just the maximal such set. This refined subset construction takes not only the alphabet symbols in Σ but also the valuations of variables in X. This determinisation can be done for any theory $Th(\mathcal{D})$ closed under conjunction and negation.

Given a DA $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$ and its determinisation $\mathcal{A}_D = (\mathcal{D}, \Sigma, X, Q_D, \iota_D, F_D, \Delta_D)$, we have[5]:

- For any $w \in (\mathcal{D}^X \times \Sigma)^*$ and $P \in Q_D$, $\mathcal{A}_D$ has exactly one run on w that starts in P;
- $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_D)$.

[5] The proof is in [34].

2.5.4 Closure Properties

Given a DA $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q, \iota, F, \Delta)$ and its determinisation $\mathcal{A}_D = (\mathcal{D}, \Sigma, X, Q_D, \iota_D, F_D, \Delta_D)$, we can construct the **complement** of $\mathcal{A}$, denoted by $\overline{\mathcal{A}}$, defined as follow:

$$\overline{\mathcal{A}} = (\mathcal{D}, \Sigma, X, Q_D, \iota_D, Q_D \setminus F_D, \Delta_D)$$

$\overline{\mathcal{A}}$ has the same structure as $\mathcal{A}_D$, and its set of final states consists of those subsets that contain no final state of $\mathcal{A}$, hence $\{P \subseteq Q \mid P \cap F = \emptyset\}$. We have $\mathcal{L}(\overline{\mathcal{A}}) = (\mathcal{D}^X \times \Sigma)^* \setminus \mathcal{L}(\mathcal{A})$.

Given two DA $\mathcal{A} = (\mathcal{D}, \Sigma, X, Q_\mathcal{A}, \iota_\mathcal{A}, F_\mathcal{A}, \Delta_\mathcal{A})$ and $\mathcal{B} = (\mathcal{D}, \Sigma, X, Q_\mathcal{B}, \iota_\mathcal{B}, F_\mathcal{B}, \Delta_\mathcal{B})$, we define the intersection of these two DA: $\mathcal{A} \times \mathcal{B} = (\mathcal{D}, \Sigma, X, Q_\mathcal{A} \times Q_\mathcal{B}, (\iota_\mathcal{A}, \iota_\mathcal{B}), F_\mathcal{A} \times F_\mathcal{B}, \Delta^\times)$ where $(q_\mathcal{A}, q_\mathcal{B}) \xrightarrow{a,\phi} (q'_\mathcal{A}, q'_\mathcal{B}) \in \Delta^\times$ if and only if: (i) $q_\mathcal{A} \xrightarrow{a,\psi} q'_\mathcal{A} \in \Delta_\mathcal{A}$; (ii) $q_\mathcal{B} \xrightarrow{a,\eta} q'_\mathcal{B} \in \Delta_\mathcal{B}$; (iii) $\phi \equiv \psi \wedge \eta$. And we have $\mathcal{L}(\mathcal{A} \times \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$.

Above we show that DA are closed under intersection, now it is easy to show that DA are also closed under union since:

$$\mathcal{L}(\overline{\overline{\mathcal{A}} \times \overline{\mathcal{B}}}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$$

Chapter 3

Alternating Data Automata (ADA)

Alternating automata have been widely used to model and verify systems that handle data from finite domains, such as communication protocols or hardware. The main advantage of the alternating model of computation is that complementation is possible in linear time, thus allowing one to concisely encode trace inclusion problems that occur often in verification.

In this chapter, we consider a model of alternating automata over infinite alphabets, called *alternating data automata (ADA)*, whose transition rules are formulae in a combined theory of Booleans and some infinite data domain, that relate past and current values of the data variables. The data theory is not fixed, but rather it is a parameter of the class.

We also show that union, intersection and complementation are possible in linear time in this model and though the emptiness problem is undecidable, we provide two efficient semi-algorithms, inspired by two state-of-the-art abstraction refinement model checking methods: lazy predicate abstraction [33] and the IMPACT semi-algorithm [45].

3.1 Introduction of ADA

3.1.1 Data Words

Firstly, we fix an interpretation $\mathcal{I}$ and a finite alphabet Σ of input events for the rest of this section. Given a finite set $X \subset VAR$ of variables of sort D, let $X \mapsto D^{\mathcal{I}}$ be the set of data symbols. A **data word** w is a finite sequence:

$$(a_1, v_1), (a_2, v_2), ..., (a_{|w|}, v_{|w|})$$

where $a_1, a_2, ..., a_{|w|} \in \Sigma$ and $v_1, v_2, ..., v_{|w|} : X \to D^{\mathcal{I}}$ are valuations. We denote by ε the empty sequence, by Σ^* the set of finite sequences of input events and by $\Sigma[X]^*$ the set of data words

over X. This definition generalises the classical notion of words from a finite alphabet to a possibly infinite alphabet $\Sigma[X]$. More precisely, when $D^{\mathcal{I}}$ is sufficiently large or infinite, we can map the elements of Σ into designated elements of $D^{\mathcal{I}}$ and use a special variable to encode the input events.

3.1.2 Definition of ADA

Given a finite set $X \subset VAR$ of variables of sort D and a finite set B of Boolean variables, we denote by $FORM(B, X)$ the set of formulae ϕ such that $FV^{Boolean}(\phi) \subseteq B$ and $FV^{D}(\phi) \subseteq X$. In addition, by $FORM^{+}(B, X)$ we denote the set of formulae from $FORM(B, X)$ in which each Boolean variable occurs only under an even number of negations.

An **alternating data automaton (ADA)** is a tuple $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ where:

- D is a possibly infinite data domain;
- Σ is a finite alphabet of input events;
- $X \subset VAR$ is a finite set of variables of sort D;
- $Q \subset VAR$ is a finite set of states which are Boolean;
- $\iota \in FORM^{+}(Q, \emptyset)$ is the initial configuration;
- $F \subseteq Q$ is the set of final states;
- $\Delta : Q \times \Sigma \rightarrow FORM^{+}(Q, \overline{X} \cup X)$ is a transition function where $\overline{X}$ denotes $\{\overline{x} \mid x \in X\}$.

In each formula $\Delta(q, a)$ where $q \in Q$ and $a \in \Sigma$, the variables $\overline{X}$ track the **previous values** and X track the **current values** of variables of $\mathcal{A}$. Observe that the initial configuration does not contain free data variables, hence the initial values of the variables are left unconstrained. The **size** of $\mathcal{A}$ is defined as $|A| = |\iota| + \sum_{(q,a) \in Q \times \Sigma} |\Delta(q, a)|$.

Example 3.1 *Figure 3.1.left depicts an ADA over $D^{\mathcal{I}} = \mathbb{Z}$ with an input alphabet $\Sigma = \{a, b\}$, variables $X = \{x, y\}$, states $Q = \{q_0, q_1, q_2, q_3, q_4\}$, initial configuration $\iota = q_0$, final states $F = \{q_3, q_4\}$ and transitions Δ given in Figure 3.1.right, where missing rules are assumed to be $false$, for example $\Delta(q_0, b) = \bot$. Transition rules $\Delta(q_0, a)$ and $\Delta(q_1, a)$ are universal, and there is no existential non-deterministic rule in this ADA. Transition rule $\Delta(q_2, a) \equiv q_2 \wedge x > \overline{x} \wedge y > \overline{y}$ compares the current value of x (denoted by x) with the past value of x (denoted by $\overline{x}$) and compares the current value of y (denoted by y) with the past value of y (denoted by $\overline{y}$). Transition rule $\Delta(q_0, a) \equiv q_1 \wedge q_2 \wedge x \approx 0 \wedge y \approx 0$ constrains the current value of x and the current value of y.*

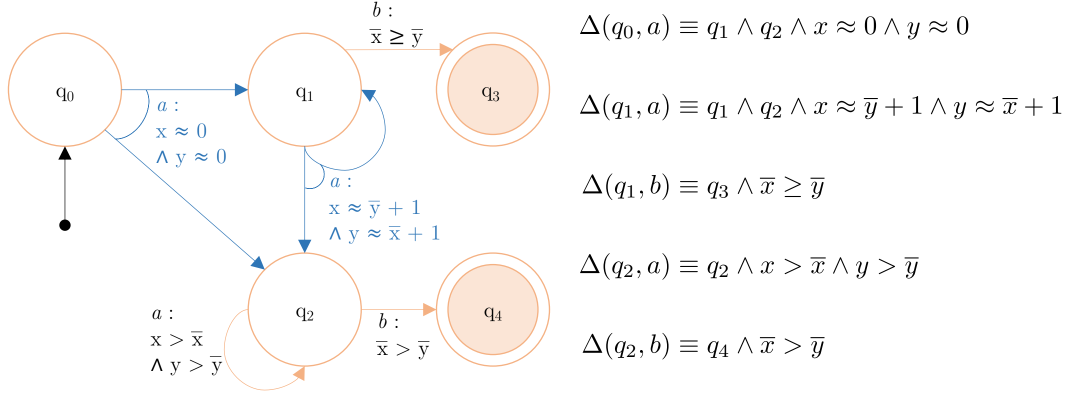


Figure 3.1: An Alternating Data Automaton

3.1.3 Time Stamp and Accepted Words

Given an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, for an input event $a \in \Sigma$ and a formula ϕ, we write $\Delta(\phi, a)$ for the formula obtained from ϕ by simultaneously replacing each state $q \in FV^{Boolean}(\phi)$ by the formula $\Delta(q, a)$. Let $X_k = \{x_k \mid x \in X\}$, for any $k \in \mathbb{N}$, be a set of **time-stamped** variables. We write $\Delta_k(\phi, a)$ for the formula obtained from ϕ by replacing each state $q \in FV^{Boolean}(\phi)$ by the formula $\Delta(q, a)[X_k/\overline{X}, X_{k+1}/X]$.

For any ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, given a word $w = (a_1, v_1), (a_2, v_2), ..., (a_{|w|}, v_{|w|})$ where $a_1, a_2, ..., a_{|w|} \in \Sigma$ and $v_1, v_2, ..., v_{|w|} : X \to D^{\mathcal{I}}$, the **run** of $\mathcal{A}$ over w is the sequence of formulae:

$$\phi_0(Q), \phi_1(Q, X_0 \cup X_1), \phi_2(Q, X_0 \cup X_1 \cup X_2), ..., \phi_{|w|}(Q, X_0 \cup X_1 \cup ... \cup X_{|w|})$$

where $\phi_0 \equiv \iota$ and $\forall k \in [1, |w|].\phi_k \equiv \Delta_k(\phi_{k-1}, a_k)$. Next, let us write $\Delta(\iota, a_1, a_2, ..., a_{|w|})$ for the formula $\phi_{|w|}(X_0, X_1, ..., X_{|w|})$ above. We say that $\mathcal{A}$ **accepts** the word w if and only if $\mathcal{I}, v \models \Delta(\iota, a_1, a_2, ..., a_{|w|})$ for the valuation v that maps:

- each $x \in X_k$ to $v_k(x)$ for all $k \in [1, |w|]$;
- each $q \in FV^{Boolean}(\phi_{|w|}) \cap F$ to $\top$;
- each $q \in FV^{Boolean}(\phi_{|w|}) \setminus F$ to $\bot$;

Example 3.2 *For the ADA in Example 3.1 (in Figure 3.1), where the function symbols have standard arithmetic interpretation, the word $w = (a, 0, 0), (a, 1, 1), (b, 2, 1)$ is not accepted. Here*

is the run of $\mathcal{A}$ on w:

$$
\begin{aligned}
& q_0 && (\phi_0)\\
\xrightarrow{a,0,0}\ & q_1 \wedge q_2 \wedge x_1 \approx 0 \wedge y_1 \approx 0 && (\phi_1)\\
\xrightarrow{a,1,1}\ & \boxed{q_1} \wedge \boxed{q_2} \wedge x_2 \approx y_1 + 1 \wedge y_2 \approx x_1 + 1 \wedge \boxed{q_2} \wedge x_2 > x_1 \wedge y_2 > y_1 \wedge x_1 \approx 0 \wedge y_1 \approx 0 && (\phi_2)\\
\xrightarrow{b,2,1}\ & \boxed{q_3 \wedge x_2 \geq y_2} \wedge \boxed{q_4 \wedge x_2 > y_2} \wedge x_2 \approx y_1 + 1 \wedge y_2 \approx x_1 + 1\\
& \wedge \boxed{q_4 \wedge x_2 > y_2} \wedge x_2 > x_1 \wedge y_2 > y_1 \wedge x_1 \approx 0 \wedge y_1 \approx 0 && (\phi_3)
\end{aligned}
$$

with the valuation v where:

- $v(x_1) = 0, v(y_1) = 0, v(x_2) = 1, v(y_2) = 1, v(x_3) = 2, v(y_3) = 1$;
- $v(q_3) = \top, v(q_4) = \top$;

we can have:

$$
\begin{aligned}
{\phi_3}^{\mathcal{I}}_{v} &= \top \wedge 1 \geq 1 \wedge \top \wedge 1 > 1 \wedge 1 = 0 + 1 \wedge 1 = 0 + 1\\
&\quad \wedge \top \wedge 1 > 1 \wedge 1 > 0 \wedge 1 > 0 \wedge 0 = 0 \wedge 0 = 0\\
&= \top \wedge \top \wedge \top \wedge \bot \wedge \top \wedge \top \wedge \top \wedge \bot \wedge \top \wedge \top \wedge \top \wedge \top\\
&= \bot
\end{aligned}
$$

Hence $\mathcal{I}, v \not\models \Delta(q_0, a, a, b)$, therefore the word $w = (a, 0, 0), (a, 1, 1), (b, 2, 1)$ is not accepted.

3.2 Closure Properties of ADA

3.2.1 Intersection

Given two ADA $\mathcal{A} = (D, \Sigma, X, Q_{\mathcal{A}}, \iota_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (D, \Sigma, X, Q_{\mathcal{B}}, \iota_{\mathcal{B}}, F_{\mathcal{B}}, \Delta_{\mathcal{B}})$, assuming without loss of generality, that $Q_{\mathcal{A}} \cap Q_{\mathcal{B}} = \emptyset$, we define the intersection automaton:

$$\mathcal{A} \cap \mathcal{B} = (D, \Sigma, X, Q_{\mathcal{A}} \cup Q_{\mathcal{B}}, \iota_{\mathcal{A}} \wedge \iota_{\mathcal{B}}, F_{\mathcal{A}} \cup F_{\mathcal{B}}, \Delta_{\mathcal{A}} \cup \Delta_{\mathcal{B}})$$

and we have $\mathcal{L}(\mathcal{A} \cap \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. The intersection can be built in linear time since $|\mathcal{A} \cap \mathcal{B}| = |\mathcal{A}| + |\mathcal{B}|$.

3.2.2 Union

Given two ADA $\mathcal{A} = (D, \Sigma, X, Q_{\mathcal{A}}, \iota_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (D, \Sigma, X, Q_{\mathcal{B}}, \iota_{\mathcal{B}}, F_{\mathcal{B}}, \Delta_{\mathcal{B}})$, assuming without loss of generality, that $Q_{\mathcal{A}} \cap Q_{\mathcal{B}} = \emptyset$, we define the union automaton:

$$\mathcal{A} \cup \mathcal{B} = (D, \Sigma, X, Q_{\mathcal{A}} \cup Q_{\mathcal{B}}, \iota_{\mathcal{A}} \vee \iota_{\mathcal{B}}, F_{\mathcal{A}} \cup F_{\mathcal{B}}, \Delta_{\mathcal{A}} \cup \Delta_{\mathcal{B}})$$

and we have $\mathcal{L}(\mathcal{A} \cup \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$. The union can be built in linear time since $|\mathcal{A} \cup \mathcal{B}| = |\mathcal{A}| + |\mathcal{B}|$.

3.2.3 Complementation

Given a set B of Boolean variables and a set X of variables of sort D, for a formula $\phi \in FORM^+(B, X)$ with no negated occurrences of the Boolean variables, we define its **complement**:

$$\begin{aligned}
\overline{\phi_1 \wedge \phi_2} &\equiv \overline{\phi_1} \vee \overline{\phi_2} \\
\overline{\phi_1 \vee \phi_2} &\equiv \overline{\phi_1} \wedge \overline{\phi_2} \\
\overline{\phi} &\equiv \phi \text{ if } \phi \in B \\
\overline{\phi} &\equiv \neg\phi \text{ if } \phi \notin B \text{ atom} \\
\overline{\neg\phi} &\equiv \neg\overline{\phi} \text{ if } \phi \text{ not atom}
\end{aligned}$$

Given an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, now we define the complement automaton:

$$\overline{A} = (D, \Sigma, X, Q, \overline{\iota}, Q \setminus F, \overline{\Delta})$$

where $\overline{\Delta}(q, a) \equiv \overline{\Delta(q, a)}$ for all $q \in Q$ and $a \in \Sigma$. We have $\mathcal{L}(\overline{\mathcal{A}}) = \Sigma[X]^* \setminus \mathcal{L}(\mathcal{A})$. The operation of complementation can be build in linear time since $|\overline{\mathcal{A}}| = |\mathcal{A}|$.

3.2.4 Proofs for Boolean Closures

We prove $\mathcal{L}(\mathcal{A} \cup \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$ first, and the proof for $\mathcal{L}(\mathcal{A} \cap \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ is analogous. Let $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n)$ be a word, where $n = 0$ corresponds to the empty word. We prove by induction on $n \geq 0$ that $\Delta(\iota_1 \vee \iota_2, a_1, a_2, ..., a_n) \Leftrightarrow \Delta(\iota_1, a_1, a_2, ..., a_n) \vee \Delta(\iota_2, a_1, a_2, ..., a_n)$. The case $n = 0$ follows from the definition of the initial configuration of $\mathcal{A} \cup \mathcal{B}$. For the inductive step $n > 0$, $\Delta(\iota_1 \vee \iota_2, a_1, a_2, ..., a_n)$ is obtained from $\Delta(\iota_1 \vee \iota_2, a_1, a_2, ..., a_{n-1})$ by replacing each variable $q \in FV^{Boolean}(\iota_1 \vee \iota_2, a_1, a_2, ..., a_{n-1})$ with $\Delta(q, a_n)[X_{n-1}/\overline{X}, X_n/X]$, denoted $\Delta^n(\Delta(\iota_1 \vee \iota_2, a_1, a_2, ..., a_{n-1}), a_n)$. Since by induction hypothesis:

$$\Delta(\iota_1 \vee \iota_2, a_1, a_2, ..., a_{n-1}) \Leftrightarrow \Delta(\iota_1, a_1, a_2, ..., a_{n-1}) \vee \Delta(\iota_2, a_1, a_2, ..., a_{n-1})$$

we obtain:

$$\Delta^n(\Delta(\iota_1 \vee \iota_2, a_1, a_2, ..., a_{n-1}), a_n)$$

$$\Leftrightarrow$$

$$\Delta^n(\Delta(\iota_1, a_1, a_2, ..., a_{n-1}), a_n) \vee \Delta^n(\Delta(\iota_2, a_1, a_2, ..., a_{n-1}), a_n)$$

$$\Leftrightarrow$$

$$\Delta(\iota_1, a_1, a_2, ..., a_n) \vee \Delta(\iota_2, a_1, a_2, ..., a_n)$$

Proposition 3.1 *Given a formula $\phi \in FORM^+(Q, X)$ and a valuation v mapping each $q \in Q$ to a value $v(q) \in \mathbb{B}$ and each $x \in X$ to a value $v(x) \in D^{\mathcal{I}}$, let v' be the valuation that assigns each $q \in Q$ the value $\neg v(q)$ and each $x \in X$ the value $v(x)$. Then we have $\mathcal{I}, v \models \phi$ if and only if $\mathcal{I}, v' \not\models \overline{\phi}$. (Can be proved immediately by induction on the structure of ϕ.)*

To prove $\mathcal{L}(\overline{\mathcal{A}}) = \Sigma[X]^* \setminus \mathcal{L}(\mathcal{A})$, let $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n)$ be a word and by induction on $n \geq 0$ show that:

$$\overline{\Delta}(\iota, a_1, a_2, ..., a_n) = \overline{\Delta(\iota, a_1, a_2, ..., a_n)}$$

The case $n = 0$ is immediate, because $FV(\iota) \subseteq Q$ and thus $\bar{\iota} \equiv \iota$. For the case $n > 0$, we compute: $\overline{\Delta}(\bar{\iota}, a_1, a_2, ..., a_n) = \overline{\Delta(\iota, a_1, a_2, ..., a_n)}$ by induction on $n \geq 0$.

In the case $n = 0$, we have $\overline{\Delta}(\bar{\iota}, a_1, a_2, ..., a_n) \equiv \bar{\iota}$. Then ε is accepted by $\mathcal{A}$ if and only if $v_0 \models \iota$, where $v_0(q) = \top$ if $q \in F$ and $v_0(q) = \bot$, otherwise. But $v_0 \models \iota$ if and only if $\overline{v_0} \models \bar{\iota}$, where $\overline{v_0}(q) = \top$ if $q \notin F$ and $\overline{v_0}(q) = \bot$, otherwise. Thus ε is accepted by $\mathcal{A}$ if and only if it is not accepted by $\overline{\mathcal{A}}$.

For the case $n > 0$, we compute:

$$\overline{\Delta}^n(\overline{\Delta}(\iota, a_1, a_2, ..., a_{n-1}), a_n)$$

$$\Leftrightarrow$$

$$\overline{\Delta}^n(\overline{\Delta(\iota, a_1 \dots a_{n-1})}, a_n)$$

$$\Leftrightarrow$$

$$\overline{\Delta(\iota, a_1, a_2, ..., a_n)}$$

Let $v, v' : (Q \cup \bigcup_{i=0}^{n} X_i) \to (\mathbb{B} \cup D^{\mathcal{I}})$ be valuations such that:

- $v(q) = \top$ and $v'(q) = \bot$, for each $q \in F$;
- $v(q) = \bot$ and $v'(q) = \top$, for each $q \in Q \setminus F$;
- $v(x) = v'(x)$, for each $x \in X_0$;
- $v(x) = v'(x) = v_i(x)$, for each $x \in X_i$ and each $i \in [1, n]$.

By *Proposition 3.1*, we have:

$$\mathcal{I}, v \models \Delta(\iota, a_1, a_2, ..., a_n)$$

$$\Leftrightarrow$$

$$\mathcal{I}, v' \not\models \overline{\Delta(\iota, a_1, a_2, ..., a_n)}$$

$$\Leftrightarrow$$

$$\mathcal{I}, v' \not\models \overline{\Delta}(\iota, a_1, a_2, ..., a_n)$$

Thus for all $w \in \Sigma[X]^*$, we have $w \in \mathcal{L}(\mathcal{A})$ if and only if $w \notin \mathcal{L}(\overline{\mathcal{A}})$.

3.3 Antichains and Interpolants for ADA Emptiness

3.3.1 Undecidability for Emptiness Problem

The emptiness problem for ADA is **undecidable**, even in very simple cases. For example, given the set of positive integers as $D^{\mathcal{I}}$, an ADA can simulate an alternating vector addition system with states (AVASS) [15] using only atoms $x \geq k$ and $x = \overline{x} + k$ for $k \in \mathbb{Z}$, with the classical interpretation of the function symbols on integers. Since the reachability of a control state is undecidable for AVASS [42], ADA emptiness is undecidable.

Consequently, given an ADA $\mathcal{A}$, we give up on the guarantee for termination and build semi-algorithms that meet the requirements below:

- if $\mathcal{L}(\mathcal{A}) \neq \emptyset$, the procedure will terminate and return a word $w \in \mathcal{L}(\mathcal{A})$ which is called a **counter-example** of emptiness;
- if the procedure terminates without returning any counter-example, then $\mathcal{L}(\mathcal{A}) = \emptyset$.

3.3.2 Post-Images and Acceptance Function

Let $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ be an ADA, given a formula $\phi \in \mathit{FORM}^+(Q, X)$ and an input event $a \in \Sigma$, we define the **post-image** function $\mathit{POST}_\mathcal{A} : \mathit{FORM}^+(Q, X) \times \Sigma \to \mathit{FORM}^+(Q, X)$ as follows:

$$\mathit{POST}_\mathcal{A}(\phi, a) \equiv \exists \overline{X}.\Delta(\phi[\overline{X}/X], a)$$

mapping each formula in $\mathit{FORM}^+(Q, X)$ to a formula defining the effect of reading the event a.

For any ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, we extend the post-image function to $\mathit{FORM}^+(Q, X) \times \Sigma^* \to \mathit{FORM}^+(Q, X)$ as follows:

- $\mathit{POST}_\mathcal{A}(\phi, \varepsilon) \equiv \phi$;
- $\mathit{POST}_\mathcal{A}(\phi, ua) \equiv \mathit{POST}_\mathcal{A}(\mathit{POST}_\mathcal{A}(\phi, u), a)$ for $a \in \Sigma$ and $u \in \Sigma^*$.

And we define now the **acceptance function** $\mathit{ACC}_\mathcal{A} : \Sigma^* \to \mathit{FORM}^+(Q, X)$ as follows:

$$\mathit{ACC}_\mathcal{A}(u) \equiv \mathit{POST}_\mathcal{A}(\iota, u) \wedge \bigwedge_{q \in Q \setminus F} (q \to \bot) \text{ for } u \in \Sigma^*.$$

The emptiness problem for an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ then becomes "Does there exist a word $u \in \Sigma^*$ such that the formula $\mathit{ACC}_\mathcal{A}(u)$ is satisfiable?". Since we ask a **satisfiability** query, the final states of $\mathcal{A}$ need not be constrained. Because each state occurs positively in $\mathit{ACC}_\mathcal{A}(u)$, this formula has a model if and only if there is a model with every $q \in F$ set to true. A naive semi-algorithm enumerates all finite sequences and checks the satisfiability of $\mathit{ACC}_\mathcal{A}(u)$

for each $u \in \Sigma^*$, using a decision procedure for the theory $\mathbb{T}(\mathcal{S}, \mathcal{I})$[1].

3.3.3 Improvement by Anti-Chains

Given a partial order $\preceq$ over a data domain D, an **antichain** is a set $A \subseteq D$ such that $a \not\preceq b$ for any $a, b \in A$.

For any ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, since no Boolean variable from Q occurs under negation in any formula, it is easy to prove the **monotonicity property**: given $\phi, \psi \in \mathit{FORM}^+(Q, X)$, if $\phi \models \psi$ then $\mathit{POST}_{\mathcal{A}}(\phi, u) \models \mathit{POST}_{\mathcal{A}}(\psi, u)$ for any $u \in \Sigma^*$. This suggests an improvement of the above semi-algorithm that enumerates and stores only a set $U \subseteq \Sigma^*$ for which $\{\mathit{POST}_{\mathcal{A}}(\phi, u) \mid u \in U\}$ forms an antichain with respect to the entailment partial order. This is because, for any $u, v \in \Sigma^*$, if $\mathit{POST}_{\mathcal{A}}(\iota, u) \models \mathit{POST}_{\mathcal{A}}(\iota, v)$ and $\mathit{ACC}_{\mathcal{A}}(uw)$ is satisfiable for some $w \in \Sigma^*$, then $\mathit{POST}_{\mathcal{A}}(\iota, uw) \models \mathit{POST}_{\mathcal{A}}(\iota, vw)$, thus $\mathit{ACC}_{\mathcal{A}}(vw)$ is satisfiable as well, and there is no need to check further for u, since the non-emptiness of $\mathcal{A}$ can be proved using v alone. However, even with this improvement, the enumeration of sequences from Σ^* diverges in many real cases, because infinite antichains exist in many interpretations, such as $q \wedge x \approx 0, q \wedge x \approx 1, ...$ for $D^{\mathcal{I}} = \mathbb{N}$.

3.3.4 Safety Invariants

A **safety invariant** for an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ is a function $\mathit{INV} : (Q \mapsto \mathbb{B}) \to 2^{X \mapsto D^{\mathcal{I}}}$ such that, for every Boolean valuation $\beta : Q \to \mathbb{B}$, every valuation $v : X \to D^{\mathcal{I}}$ of the data variables and every finite sequence $u \in \Sigma^*$ of input events, the following hold:

- $\mathcal{I}, \beta \cup v \models \mathit{POST}_{\mathcal{A}}(\iota, u) \Rightarrow v \in \mathit{INV}(\beta)$;
- $v \in \mathit{INV}(\beta) \Rightarrow \mathcal{I}, \beta \cup v \not\models \mathit{ACC}_{\mathcal{A}}(u)$.

If INV satisfies only the first point above, then we call it an **invariant**. Intuitively, a safety invariant maps every Boolean valuation into a set of data valuations, that contains the initial configuration $\iota \equiv \mathit{POST}_{\mathcal{A}}(\iota, \varepsilon)$, whose data variables are unconstrained, over-approximates the set of reachable valuations and excludes the valuations satisfying the acceptance condition.

For an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, a formula $\phi(Q, X)$ is said to define INV if and only if for all $\beta : Q \to \mathbb{B}$ and $v : X \to D^{\mathcal{I}}$, we have $\mathcal{I}, \beta \cup v \models \phi$ if and only if $v \in \mathit{INV}(\beta)$. And in addition, we have following lemma:

Lemma 3.1 *$\mathcal{L}(\mathcal{A}) = \emptyset$ if and only if $\mathcal{A}$ has a safety invariant.*

The proof of *Lemma 3.1* is very simple. Let $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ in the following:

[1] The theory $\mathbb{T}(\mathcal{S}, \mathcal{I})$ is the set of valid formulae written in the signature $\mathcal{S}$, with the interpretation $\mathcal{I}$. A decision procedure for $\mathbb{T}(\mathcal{S}, \mathcal{I})$ is an algorithm that takes a formula ϕ in the signature $\mathcal{S}$ and returns yes if and only if $\phi \in \mathbb{T}(\mathcal{S}, \mathcal{I})$.

$\Leftarrow$ This direction is trivial.

$\Rightarrow$ We define: $INV : (Q \to \mathbb{B}) \to 2^{X \to D^{\mathcal{I}}}$ as follows. For each $\beta : Q \to \mathbb{B}$, let $INV(\beta) = \{v : X \to D^{\mathcal{I}} \mid \exists u \in \Sigma^*. \beta \cup v \models POST_{\mathcal{A}}(\iota, u)\}$. Checking that INV is a safety invariant is straightforward.

3.3.5 Abstraction and Refinement

Turning back to our issue of divergence of language emptiness semi-algorithms in the case $\mathcal{L}(\mathcal{A}) = \emptyset$, we can observe that an enumeration of input sequences $u_1, u_2, ... \in \Sigma^*$ can stop at step k as soon as $\bigvee_{i=1}^{k} POST_{\mathcal{A}}(\iota, u_i)$ defines a safety invariant for $\mathcal{A}$. Although this condition can be effectively checked using a decision procedure for the theory $\mathbb{T}(\mathcal{S}, \mathcal{I})$, there is no guarantee that this check will ever succeed.

The solution we adopt in the sequel is an abstraction to ensure the termination of invariant computations. However, it is worth pointing out from the start that the abstraction alone will only allow us to build invariants that are not necessarily safety invariants. To meet the latter condition, we resort to **counter-example guided abstraction refinement (CEGAR)**.

Formally, for a given ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, we fix $\Pi \subseteq FORM(Q, X)$, a set of formulae such that $\bot \in \Pi$ and refer to these formulae as **predicates**. Given a formula ϕ, we denote by $\phi^{\#} \equiv \bigwedge\{\pi \in \Pi \mid \phi \in \pi\}$ the **abstraction** of ϕ with respect to the predicates in Π. The abstract version of the post-image is defined as follows:

- $POST^{\#}_{\mathcal{A}}(\phi, \varepsilon) \equiv \phi^{\#}$;
- $POST^{\#}_{\mathcal{A}}(\phi, ua) \equiv (POST_{\mathcal{A}}(POST^{\#}_{\mathcal{A}}(\phi, u), a))^{\#}$ for $a \in \Sigma$ and $u \in \Sigma^*$.

With this abstract version of post-image, we can define the abstract version of acceptance function:

$$ACC^{\#}_{\mathcal{A}}(u) \equiv POST^{\#}_{\mathcal{A}}(\iota, u) \wedge \bigwedge_{q \in Q \setminus F} (q \to \bot) \text{ for } u \in \Sigma^*.$$

Lemma 3.2 *For any bijection $\mu : \mathbb{N} \to \Sigma^*$, there exists $k > 0$ such that $\bigvee_{m=0}^{k} POST^{\#}_{\mathcal{A}}(\iota, \mu(m))$ defines an invariant $INV^{\#}$ for $\mathcal{A}$.*

The proof of *Lemma 3.2* is not complicated. It is sufficient to show that there exists $k \geq 0$ such that for all $u \in \Sigma^*$ there exists $i \in [0, k]$ such that $POST_{\mathcal{A}}(\iota, u) \models POST^{\#}_{\mathcal{A}}(\iota, \mu(i))$. We have $POST_{\mathcal{A}}(\iota, u) \models POST^{\#}_{\mathcal{A}}(\iota, u)$ for all $u \in \Sigma^*$. But since Π is a finite set, also the set $\{POST^{\#}_{\mathcal{A}}(\iota, u) \mid u \in \Sigma^*\}$ is finite. Thus there exists $k \geq 0$ such that, for all $u \in \Sigma^*$ there exists $i \in [0, k]$ such that $POST^{\#}_{\mathcal{A}}(\iota, u) \Leftrightarrow POST^{\#}_{\mathcal{A}}(\iota, \mu(i))$, which concludes the proof.

If we look back to the definition of safety invariants in the previous section, we are left with fulfilling the second point from the definition. To this end, suppose that, for a given set Π of predicates, the invariant $INV^{\#}$ defined above meets the first point of the definition of a safety

invariant but not the second point. In other words, there exists a finite sequence $u \in \Sigma^*$ such that $v \in INV^{\#}(\beta)$ and $\mathcal{I}, \beta \cup v \models ACC^{\#}_{\mathcal{A}}(u)$ for some Boolean $\beta : Q \to \mathbb{B}$ and data $v : X \to D^{\mathcal{I}}$ valuations. Such $u \in \Sigma^*$ is called a **counter-example**. Once a counter-example u is discovered, there are two possibilities: either (i) $ACC_{\mathcal{A}}(u)$ is satisfiable, in which case u is **feasible** and $\mathcal{L}(\mathcal{A}) \neq \emptyset$; or (ii) $ACC_{\mathcal{A}}(u)$ is unsatisfiable, in which case u is **spurious**. In the first case, our semi-algorithm stops and returns a witness for non-emptiness (the counter-example), obtained from the satisfying valuation of $ACC_{\mathcal{A}}(u)$. In the second case, we must strengthen the invariant by excluding from $INV^{\#}$ all pairs (β, v) such that $\mathcal{I}, \beta \cup v \models ACC^{\#}_{\mathcal{A}}(u)$. This strengthening is carried out by adding to Π several predicates that are sufficient to exclude the spurious counter-example.

Given an unsatisfiable conjunction of formulae $\psi_1 \wedge \psi_2 \wedge ... \wedge \psi_n$, an **interpolant** is a tuple of formulae $(I_1, I_2, ..., I_n)$ such that $I_n \equiv \bot, I_i \wedge \psi_i \models I_{i+1}$ and I_i contains only variables and function symbols that are common to ψ_i and ψ_{i+1}, for all $i \in [1, n-1]$. Moreover, by Lyndon's Interpolation Theorem [43], we can assume without loss of generality that every Boolean variable with at least one positive/negative occurrence in I_i has at least one positive/negative occurrence in both ψ_i and ψ_{i+1}. In the following, we shall assume the existence of an interpolating decision procedure for $\mathbb{T}(\mathcal{S}, \mathcal{I})$ that meets the requirements of Lyndon's Interpolation Theorem.

A classical method for abstraction refinement is to add the elements of the interpolant obtained from a proof of spuriousness to the set of predicates. This guarantees progress, meaning that the particular spurious counter-example, from which the interpolant was generated, will never be revisited in the future. Though not always, in many practical test cases, this progress property eventually yields a safety invariant.

Given a non-empty spurious counter-example $u = a_1, a_2, ..., a_n$, where $n > 0$, we consider the following interpolation problem:

$$\Theta(u) \equiv \theta_0(Q_0) \wedge \theta_1(Q_0 \cup Q_1, X_0 \cup X_1) \wedge ... \wedge \theta_n(Q_{n-1} \cup Q_n, X_{n-1} \cup X_n) \wedge \theta_{n+1}(Q_n)$$

where $Q_k = \{q_k \mid q \in Q\}$ for $k \in [0, n]$ are time-stamped sets of Boolean variables corresponding to the set Q of states of $\mathcal{A}$. The first conjunct $\theta_0(Q_0) \equiv \iota[Q_0/Q]$ is the initial configuration of $\mathcal{A}$, with every $q \in FV^{Boolean}(\iota)$ replaced by q_0. The definition of θ_k for all $k \in [1, n]$, uses **replacement sets** $R_j \subseteq Q_j, j \in [0, n]$, which are defined inductively below:

- $R_0 = FV^{Boolean}(\theta_0)$;
- $\theta_j \equiv \bigwedge_{q_{j-1} \in R_{j-1}} (q_{j-1} \to \Delta(q, a_j)[Q_j/Q, X_{j-1}/\overline{X}, X_j/X]$ and $R_j = FV^{Boolean}(\theta_j) \cap Q_j$ for each $j \in [1, n]$;
- $\theta_{n+1}(Q_n) \equiv \bigwedge_{q \in Q \setminus F} (q_n \to \bot)$.

The intuition is that $R_0, R_1, ..., R_n$ are the sets of states replaced, $\theta_0, \theta_1, ..., \theta_n$ are the sets of transition rules fired on the run of $\mathcal{A}$ over u and θ_{n+1} is the acceptance condition, which forces the last remaining non-final states to be false. We recall that a run of $\mathcal{A}$ over u is a sequence:

$$\phi_0(Q), \phi_1(Q, X_0 \cup X_1), \phi_2(Q, X_0 \cup X_1 \cup X_2), ..., \phi_n(Q, X_0 \cup X_1 \cup ... \cup X_n)$$

where ϕ_0 is the initial configuration ι and for each $k > 0$, ϕ_k is obtained from ϕ_{k-1} by replacing each state $q \in FV^{Boolean}(\phi_{k-1})$ by the formula $\Delta(q, a_k)[X_{k-1}/\overline{X}, X_k/X]$, given by the transition function of $\mathcal{A}$. Observe that, because the states are replaced with transition formulae when moving one step in a run, these formulae lose track of the control history and are not suitable for producing interpolants that relate states and data.

The main idea behind the above definition of the interpolation problem is that we would like to obtain an interpolant $(\top, I_0(Q), I_1(Q, X), ..., I_n(Q, X), \bot)$ whose formulae combine states with the data constraints that must hold **locally**, whenever the control reaches a certain Boolean configuration. This association of states with data valuations is tantamount to defining efficient semi-algorithms, based on lazy abstraction. Furthermore, the abstraction defined by the interpolants generated in this way can also **over-approximate** the control structure of an automaton, in addition to the sets of data values encountered throughout its run.

The correctness of this interpolation-based abstraction refinement setup is captured by the progress property below, which guarantees that adding the formulae of an interpolant for $\Theta(u)$ to the set Π of predicates suffices to exclude the spurious counter-example u from future searches.

Lemma 3.3 *Let $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ be an ADA, for any sequence $u = a_1, a_2, ..., a_{|u|} \in \Sigma^*$, if $ACC_{\mathcal{A}}(u)$ is unsatisfiable, then:*

- *$\Theta(u)$ is unsatisfiable;*
- *if $(\top, I_0, I_1, ..., I_n, \bot)$ is an interpolant for $\Theta(u)$ such that $\{I_i \mid i \in [0, n]\} \subseteq \Pi$, then $ACC^{\#}_{\mathcal{A}}(u)$ is unsatisfiable.*

In order to prove *Lemma 3.3*, we need to firstly see following proposition.

Proposition 3.2 *Given a formula $\phi \in FORM^+(Q, X)$ and $a \in \Sigma$, we have:*

$$\Delta(\phi, a) \Leftrightarrow \exists Q'.\phi[Q'/Q] \wedge \bigwedge_{q \in Q} (q' \rightarrow \Delta(q, a))$$

Here is the proof of *Proposition 3.2*:

- $\Rightarrow$ If $\mathcal{I}, \beta \cup \overline{v} \cup v \models \Delta(\phi, a)$, for some valuations $\beta : Q \rightarrow \mathbb{B}$ and $\overline{v} : \overline{X} \rightarrow D^{\mathcal{I}}$, $v : X \rightarrow D^{\mathcal{I}}$, then we build a valuation $\beta' : Q' \rightarrow \mathbb{B}$ such that $\mathcal{I}, \beta' \cup \beta \cup \overline{v} \cup v \models \phi[Q'/Q] \wedge \bigwedge_{q \in Q} (q' \rightarrow \Delta(q, a))$. For each occurrence of a formula $\Delta(q, a)$ in $\Delta(\phi, a)$ we set $\beta'(q') = true$ if $\mathcal{I}, \beta \cup \overline{v} \cup v \models \Delta(q, a)$ and $\beta'(q') = false$, otherwise. Since there are no negated occurrences of such sub-formulae, the definition of β' is consistent, and the check $\mathcal{I}, \beta' \cup \beta \cup \overline{v} \cup v \models \phi[Q'/Q] \wedge \bigwedge_{q \in Q} (q' \rightarrow \Delta(q, a))$ is immediate.
- $\Leftarrow$ This direction is an easy check.

Here is the proof of *Lemma 3.3*. Let $\Theta(u) \equiv \theta_0(Q_0) \wedge \theta_1(Q_0 \cup Q_1, X_0 \cup X_1) \wedge ... \wedge \theta_n(Q_{n-1} \cup Q_n, X_{n-1} \cup X_n) \wedge \theta_{n+1}(Q_n)$ in the following:

(1) We apply *Proposition 3.2* recursively and get:

$$POST^{\#}_{\mathcal{A}}(\iota, u)[Q_n/Q, X_n/X] \iff \exists Q_0, \exists Q_1, ..., \exists Q_{n-1}, \exists X_0, \exists X_1, ..., \exists X_{n-1}. \bigwedge_{i=0}^{n} \theta_i$$

Assuming that $\Theta(u)$ is satisfiable, we obtain a model for $ACC_{\mathcal{A}}(u) \equiv POST_{\mathcal{A}}(\iota, u) \wedge \theta_{n+1}[Q/Q_n]$.

(2) if $(\top, I_0, I_1, ..., I_n, \bot)$ is an interpolant for $\Theta(u)$, the following entailments hold:

- $\theta_0 \models I_0[Q_0/Q]$;
- $I_{k-1}[Q_{k-1}/Q, X_{k-1}/X] \wedge \theta_k \models I_k[Q_k/Q, X_k/X]$, for all $k \in [1, n]$;
- $I_n[Q_n/Q] \wedge \theta_{n+1} \models \bot$.

We prove that $POST^{\#}_{\mathcal{A}}(\iota, a_1, a_2, ..., a_n) \models I_n$ by induction on $n \geq 0$. This is sufficient to conclude because $ACC^{\#}_{\mathcal{A}}(a_1, a_2, ..., a_n) \equiv POST^{\#}_{\mathcal{A}}(\iota, a_1, a_2, ..., a_n) \wedge \theta_{n+1}[Q/Q_n] \models I_n \wedge \theta_{n+1}[Q/Q_n] \models \bot$. For the base case $n = 0$, we have $POST^{\#}_{\mathcal{A}}(\iota, \varepsilon) \equiv \iota \equiv \theta_0[Q/Q_0] \models I_0$. For the induction step $n > 0$, we compute:

$$POST_{\mathcal{A}}(\iota, a_1, a_2, ..., a_n)[Q_n/Q] \equiv \text{(by def. of } POST^{\#}_{\mathcal{A}})$$

$$\exists X_{n-1}.\Delta^n(POST^{\#}_{\mathcal{A}}(\iota, a_1, a_2, ..., a_{n-1}), a_n)^{\#}[Q_n/Q] \models \text{(by \textit{Proposition 3.2})}$$

$$\exists Q_{n-1} \exists X_{n-1}.POST^{\#}_{\mathcal{A}}(\iota, a_1, a_2, ..., a_{n-1})[Q_{n-1}/Q] \wedge \theta_n \models \text{(ind. hyp.)}$$

$$\exists Q_{n-1} \exists X_{n-1}.I_{n-1}[Q_{n-1}/Q] \wedge \theta_n \models I_n[Q_n/Q]$$

3.4 Checking Emptiness - Lazy Predicate Abstraction

3.4.1 Abstract Reachability Tree (ART)

In the context of checking emptiness of an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, an **abstract reachability tree (ART)** is a tuple $\mathcal{T} = (N, E, r, \Lambda, R, T, \triangleleft)$ where:

- N is a set of nodes;
- $E \subseteq N \times \Sigma \times N$ is a set of edges;
- $r \in N$ is the root of the directed tree (N, E);
- $\Lambda : N \to FORM(Q, X)$ is a labelling of the nodes with formulae such that $\Lambda(r) = \iota$;
- $R : N \to 2^Q$ is a labelling of nodes with replacement sets such that $R(r) = FV^{Boolean}(\iota)$;

- $T : E \to \bigcup_{i=0}^{\infty} FORM^{+}(Q_i, X_i, Q_{i+1}, X_{i+1})$ is a labelling of edges with time-stamped formulae;
- $\lhd \subseteq N \times N$ is a set of **covering edges**.

Each node $n \in N$ corresponds to a unique path from the root to n, labelled by a sequence $\lambda(n) \in \Sigma^*$ of input events. The **least infeasible suffix** of $\lambda(n)$ is the smallest sequence $v = a_1, a_2, ..., a_k$ such that $\lambda(n) = wv$ for some $w \in \Sigma^*$ and the following formula is unsatisfiable:

$$\Psi(v) \equiv \Lambda(p)[Q_0/Q] \wedge \theta_1(Q_0 \cup Q_1, X_0 \cup X_1) \wedge ... \wedge \theta_{k+1}(Q_k)$$

where $\theta_1, \theta_2, ..., \theta_{n+1}$ are defined as in the interpolation problem and $\theta_0 \equiv \Lambda(p)[Q_0/Q]$. The **pivot** of n is the node p corresponding to the start of the least infeasible suffix. We assume the existence of two functions without detailing their implementation:

- *FindPivot*$(u, \mathcal{T})$ returning the pivot of a sequence $u \in \Sigma^*$ in an ART $\mathcal{T}$;
- *LeastInfeasibleSuffix*$(u, \mathcal{T})$ returning the least infeasible suffix of a sequence $u \in \Sigma^*$ in an ART $\mathcal{T}$;

3.4.2 Lazy Predicate Abstraction Semi-Algorithm

We now have all the ingredients to describe the first emptiness checking semi-algorithm for ADA. *Semi-Algorithm 1* builds an ART whose nodes are labelled with formulae over-approximating the concrete sets of configurations, and a covering relation between nodes in order to ensure that the set of formulae labelling the nodes in the ART forms an antichain. Any spurious counter-example is eliminated by computing an interpolant and adding its formulae to the set of predicates.

Semi-Algorithm 1 Lazy Predicate Abstraction for ADA Emptiness

Input: an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$

Output: $\begin{cases} true & \text{if } \mathcal{L}(\mathcal{A}) = \emptyset \\ \text{a data word } w \in \mathcal{L}(\mathcal{A}) & \text{if } \mathcal{L}(\mathcal{A}) \neq \emptyset \end{cases}$

1: let $\mathcal{T} = (N, E, r, \Lambda, \lhd)$ be an ART
2: let Π be a set
3: let *WorkList* be a list
4: $N := \emptyset$
5: $E := \emptyset$
6: $\Lambda := \{(r, \iota)\}$
7: $\lhd := \emptyset$
8: add $\{\bot\}$ into Π
9: add r into *WorkList*
10: **while** *WorkList* $\neq \emptyset$ **do**
11: dequeue n from *WorkList*
12: $N := N \cup \{n\}$

13: let $\lambda(n) = a_1, a_2, ..., a_k$ be the label of the path from r to n
14: **if** $POST^{\#}_{\mathcal{A}}(\lambda(n))$ is satisfiable **then**
15:   **if** $ACC_{\mathcal{A}}(\lambda(n))$ is satisfiable **then**
16:     get model $(\beta, v_1, v_2, ..., v_k)$ of $ACC^{\#}_{\mathcal{A}}(\lambda(n))$
17:     **return** $w = (a_1, v_1), (a_2, v_2), ..., (a_k, v_k)$
18:   **else**
19:     $p := \mathit{FindPivot}(\lambda(n), \mathcal{T})$
20:     $v := \mathit{LeastInfeasibleSuffix}(\lambda(n), \mathcal{T})$
21:     $\Pi := \Pi \cup \{I_0, I_1, ..., I_k\}$ where $(\top, I_0, I_1, ..., I_k, \bot)$ is an interpolant for $\Psi(v)$
22:     let $\mathcal{S} = (N', E', p, \Lambda', \lhd')$ be the sub-tree of $\mathcal{T}$ rooted at p
23:     **for** $(m, q) \in \lhd$ such that $q \in N'$ **do**
24:       remove m from N and enqueue m into *WorkList*
25:       remove $\mathcal{S}$ from $\mathcal{T}$
26:       enqueue p into *WorkList*
27:     **end for**
28:   **end if**
29: **else**
30:   **for** $a \in \Sigma$ **do**
31:     $\phi := POST^{\#}_{\mathcal{A}}(\Lambda(n), a)$
32:     **if** exist $m \in N$ such that $\phi \models \Lambda(m)$ **then**
33:       $\lhd := \lhd \cup \{(n, m)\}$
34:     **else**
35:       let s be a fresh node
36:       $E := E \cup \{(n, a, s)\}$
37:       $\Lambda := \Lambda \cup \{(s, \phi)\}$
38:       $R := \{m \in \mathit{WorkList} \mid \Lambda(m) \models \phi\}$
39:       **for** $r \in R$ **do**
40:         **for** $m \in N$ such that $(m, b, r) \in E, b \in \Sigma$ **do**
41:           $\lhd := \lhd \cup \{(m, s)\}$
42:         **end for**
43:         **for** $(m, r) \in \lhd$ **do**
44:           $\lhd := \lhd \cup \{(m, s)\}$
45:         **end for**
46:       **end for**
47:       remove R from $\mathcal{T}$
48:       enqueue s into *WorkList*
49:     **end if**
50:   **end for**
51: **end if**
52: **end while**
53: **return** *true*

Semi-Algorithm 1 uses a work-list iteration to build an ART. We keep newly expanded nodes of $\mathcal{T}$ in a queue *WorkList*, thus implementing a breadth-first exploration strategy, which guarantees that the shortest counter-examples are explored first. When the search encounters a counter-example candidate u, it is checked for spuriousness. If the counter-example is feasible,

the procedure returns a data word $w \in \mathcal{L}(\mathcal{A})$, which interleaves the input events of u with the data valuations from the model of $ACC_{\mathcal{A}}(u)$. Since u is feasible, clearly $ACC_{\mathcal{A}}(u)$ is satisfiable. Otherwise, u is spurious and we compute its pivot p, add the interpolants for the least infeasible suffix of u to Π, remove and recompute the sub-tree of $\mathcal{T}$ rooted at p.

Termination of *Semi-Algorithm 1* depends on the ability of a given interpolating decision procedure for the combined Boolean and data theory $\mathbb{T}(\mathcal{S}, \mathcal{I})$ to provide interpolants that yield a safety invariant, whenever $\mathcal{L}(\mathcal{A}) = \emptyset$. In this case, we use the covering relation $\lhd$ to ensure that, when a newly generated node is covered by a node already in N, it is not added to the work-list, thus cutting the current branch of the search.

Formally, for any two nodes $n, m \in N$, we have $n \lhd m$ if and only if $POST^{\#}_{\mathcal{A}}(\Lambda(n), a) \models \Lambda(m)$ for some $a \in \Sigma$. In other words, if n has a successor whose label entails the label of m.

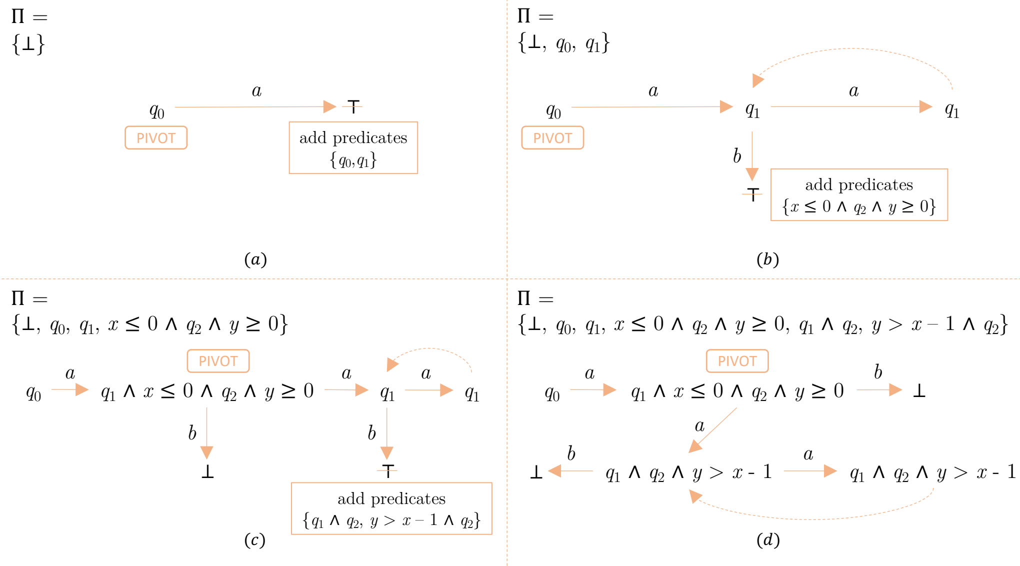


Figure 3.2: Proving Emptiness of the ADA in *Figure 3.1* by *Semi-Algorithm 1*

Example 3.3 *Consider the ADA in Figure 3.1. First, Semi-Algorithm 1 fires the sequence a, and since there are no other formulae than $\bot$ in Π, the successor of $\iota \equiv q_0$ is $\top$ (see Figure 3.2.a). The spuriousness check for a yields the root of the ART as pivot and the interpolant (q_0, q_1), which is added to the set Π. Then the $\top$ node is removed and the next time a is fired, it creates a node labelled q_1. The second sequence aa creates a successor node q_1, which is covered by the first, depicted with a dashed arrow (see Figure 3.2.b). The third sequence is ab, which results in a new uncovered node $\top$ and triggers a spurious check. The new predicate obtained from this check is $x \leq 0 \wedge q_2 \wedge y \geq 0$ and the pivot is again the root. Then the entire ART is rebuilt with the new predicates and the fourth sequence aab yields an uncovered node $\top$ (see Figure 3.2.c). The new pivot is the end-point of a and the newly added predicates are $q_1 \wedge q_2$ and $y > x - 1 \wedge q_2$. Finally, the ART is rebuilt from the pivot node and finally all nodes are covered; thus proving the emptiness of the automaton (see Figure 3.2.d).*

Theorem 3.1 *Given an ADA* $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$*:*

- *if* $\mathcal{L}(\mathcal{A}) \neq \emptyset$*, then Semi-Algorithm 1 terminates by returning a word* $w \in \mathcal{L}(\mathcal{A})$ *(hence the termination is guaranteed when* $\mathcal{A}$ *is not empty);*
- *if Semi-Algorithm 1 terminates by reporting true, then* $\mathcal{L}(\mathcal{A}) = \emptyset$ *(although the termination is not guaranteed when* $\mathcal{A}$ *is empty, if the semi-algorithm terminates by reporting true, then* $\mathcal{A}$ *is surely empty, hence the correctness of the result is guaranteed).*

Here is the proof of *Theorem 3.1*:

We prove the following invariant: each time *Semi-Algorithm 1* reaches *line 10*, the set W of nodes in *WorkList* contains all the frontier nodes in the ART $(N \cup W, E, r, \Lambda, \lhd)$ which are not covered by some node in N, namely that:

$$W = \{n \mid \forall m \in N, \forall a \in \Sigma.(n, a, m) \notin E \wedge (n, m) \notin \lhd\}$$

Initially, this is the case because $W = \{r\}$ and $E = \lhd = \emptyset$. If the invariant holds previously, at *line 10*, it will hold again after *line 26* is executed, because, when the sub-tree rooted at the pivot p is removed, p becomes a member of the set of uncovered frontier nodes, and is added to W at *line 26*. Otherwise, the invariant holds at *line 10* and the control follows the else branch at *line 29*. In this case, the newly created frontier node s is added to W only if it is not covered by an existing node in N (*line 32*).

Next we prove that, if *Semi-Algorithm 1* returns *true*, then $\bigvee_{n \in N} \Lambda(n)$ defines a safety invariant. Suppose that *Semi-Algorithm 1* returns at *line 53*. Then it must be that $W = \emptyset$. Each node in N is either covered by another node in N, or all its successors are in N. We prove first that $\bigvee_{n \in N} \Lambda(n)$ is an invariant: for any $u \in \Sigma^*$, there exists some node $n \in N$ such that $POST_{\mathcal{A}}(\iota, u) \models \Lambda(n)$. Let $u \in \Sigma^*$ be an arbitrary sequence. If u labels the path from r to some $n \in N$, we have $POST_{\mathcal{A}}(\iota, u) \models POST^{\#}_{\mathcal{A}}(\iota, u) \models \Lambda(n)$ and we are done. Otherwise, let v be the (possibly empty) prefix of u which labels the path from r to some $n \in N$, which is covered by another $m \in N$, where $(n, a, m) \in E$, that is $u = vav'$, for some $a \in \Sigma$ and $v' \in \Sigma^*$. Moreover, we have $POST_{\mathcal{A}}(\iota, va) \models POST^{\#}_{\mathcal{A}}(\iota, va) \models \Lambda(m)$, by the construction of the set $\lhd$ of covering edges: lines *line 33*, *line 41* and *line 44*. Continuing this argument recursively from m, since $|v'| < |u|$, we shall eventually discover a node p such that $POST_{\mathcal{A}}(\iota, u) \models \Lambda(p)$.

To prove that $\bigvee_{n \in N} \Lambda(n)$ is, moreover, a safety invariant, suppose, by contradiction, that there exists $u \in \Sigma^*$ such that $ACC_{\mathcal{A}}(u)$ is satisfiable. By the previous point, there exists a node $p \in N$ such that $POST_{\mathcal{A}}(\iota, u) \models \Lambda(p)$. But then we have $ACC_{\mathcal{A}}(\iota, u) \models ACC_{\mathcal{A}}(\iota, \Lambda(p))$, thus $ACC_{\mathcal{A}}(\iota, \Lambda(p))$ is satisfiable as well. However, this cannot be the case, because p has been processed at *line 15* and *Semi-Algorithm 1* would have returned a counter-example, contradicting the assumption that it returns *true*. This concludes the proof that $\bigvee_{n \in N} \Lambda(n)$ is a safety invariant, thus $\mathcal{L}(\mathcal{A}) = \emptyset$, by *Lemma 3.1*. We have then proved the second point of the statement.

For the first point, assume that $\mathcal{L}(\mathcal{A}) \neq 0$ and let $w = (a_1, v_1), (a_2, v_2), ..., (a_k, v_k) \in \mathcal{L}(\mathcal{A})$ be a word. By the above, *Semi-Algorithm 1* cannot return *true*. Suppose, by contradiction that

it does not terminate. Since the sequences from Σ^* are explored in breadth-first order, every sequence of length k is eventually explored, which leads to the discovery of w at *line 15*. Then *Semi-Algorithm 1* terminates returning $w \in \mathcal{L}(\mathcal{A})$.

3.5 Checking Emptiness - IMPACT

3.5.1 In-Place Refinement and Coverage

As pointed out by a number of authors, the bottleneck of predicate abstraction is the high cost of reconstructing parts of the ART, subsequent to the refinement of the set of predicates. The main idea of the IMPACT procedure [45] is that this can be avoided and the refinement (strengthening of the node labels of the ART) can be performed in-place. This refinement step requires an update of the covering relation, because a node that used to cover another node might not cover it anymore after the strengthening of its label.

We consider a total alphabetical order $\prec$ on Σ and lift it to the total lexicographical order $\prec^*$ on Σ^*. A node $n \in N$ is **covered** if $(n, p) \in \lhd$ or it has an ancestor m such that $(m, p) \in \lhd$ for some $p \in N$. A node n is **closed** if it is covered, or $\Lambda(n) \not\models \Lambda(m)$ for all $m \in N$ such that $\lambda(m) \prec^* \lambda(n)$. Observe that we use the coverage relation $\lhd$ here with a different meaning than in *Semi-Algorithm 1*.

3.5.2 IMPACT Semi-Algorithm

The execution of *Semi-Algorithm 2* consists of three phases: **close**, **refine** and **expand**, corresponding to the CLOSE, REFINE and EXPAND in [45].

Semi-Algorithm 2 IMPACT for ADA Emptiness

Input: an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$

Output: $\begin{cases} true & \text{if } \mathcal{L}(\mathcal{A}) = \emptyset \\ \text{a data word } w \in \mathcal{L}(\mathcal{A}) & \text{if } \mathcal{L}(\mathcal{A}) \neq \emptyset \end{cases}$

1: let $\mathcal{T} = (N, E, r, \Lambda, R, T, \lhd)$ be an ART
2: let *WorkList* be a list
3: $N := \emptyset$
4: $E := \emptyset$
5: $\Lambda := \{(r, \iota)\}$
6: $R := FV^{Boolean}(\iota[Q_0/Q])$
7: $T := \emptyset$
8: $\lhd := \emptyset$
9: add r into *WorkList*
10: **while** *WorkList* $\neq \emptyset$ **do**
11: dequeue n from *WorkList*
12: $N := N \cup \{n\}$
13: let $(r, a_1, n_1), (n_1, a_2, n_2), ..., (n_{k-1}, a_k, n)$ be the path from r to n

```
14:     if ACC_A(a_1, a_2, ..., a_k) is satisfiable then
15:        get model (β, v_1, v_2, ..., v_k) of ACC_A(λ(n))
16:        return w = (a_1, v_1), (a_2, v_2), ..., (a_k, v_k)
17:     else
18:        let (⊤, I_0, I_1, ..., I_k, ⊥) be an interpolant for Θ(a_1, a_2, ..., a_k)
19:        b := false
20:        for i ∈ [0, k] do
21:           if Λ(n_i) ⊭ I_i then
22:              ◁ := ◁ \ {(m, n_i) ∈ ◁ | m ∈ N}
23:              Λ(n_i) := Λ(n_i) ∧ I_i
24:              if ¬b then
25:                 b := CLOSE(n_i)
26:              end if
27:           end if
28:        end for
29:     end if
30:     if n is not covered then
31:        for a ∈ Σ do
32:           let s be a fresh node
33:           let e = (n, a, s) be a new edge
34:           E := E ∪ {e}
35:           Λ := Λ ∪ {(s, ⊤)}
36:           T := T ∪ {(e, θ_k)}
37:           R := R ∪ {(s, ⋃_{q∈R(n)} FV^Boolean(Δ(q, a)))}
38:           enqueue s into WorkList
39:        end for
40:     end if
41:  end while
42:  return true
```

Function 1 CLOSE

Input: a node x

Output: $\begin{cases} true & \text{if } x \text{ is closed} \\ false & \text{if } x \text{ is not closed} \end{cases}$

```
1: for y ∈ N such that λ(y) ≺* λ(x) do
2:    if Λ(x) ⊨ Λ(y) then
3:       ◁ := (◁ \ {(p, q) ∈ ◁ | q is x or a successor of x}) ∪ {(x, y)}
4:       return true
5:    end if
6: end for
7: return false
```

Let n be a node removed from the *WorkList*. If $ACC_{\mathcal{A}}(\lambda(n))$ is satisfiable, the counterexample $\lambda(n)$ is feasible, in which case a model of $ACC_{\mathcal{A}}(\lambda(n))$ is obtained and a word $w \in$

$\mathcal{L}(\mathcal{A})$ is returned. Otherwise, $\lambda(n)$ is a spurious counter-example and the procedure enters the refinement phase. The interpolant for $\Theta(\lambda(n))$ is used to strengthen the labels of all the ancestors of n by conjoining the formulae of the interpolant to the existing labels. In this process, the nodes on the path between r and n, including n, might become eligible for coverage, therefore we attempt to close each ancestor of n that is impacted by the refinement. Observe that, in this case the call to CLOSE must uncover each node which is covered by a successor of n. This is required because, due to the over-approximation of the sets of reachable configurations, the covering relation is not transitive, as explained in [45]. If CLOSE adds a covering edge (n_i, m) to $\lhd$, it does not have to be called for the successors of n_i on this path, which is handled via the Boolean flag b. Finally, if n is still uncovered (it has not been previously covered during the refinement phase), we expand n by creating a new node for each successor s via the input event $a \in \Sigma$ and inserting it into the *WorkList*.

Theorem 3.2 *Given an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$:*

- *if $\mathcal{L}(\mathcal{A}) \neq \emptyset$, then Semi-Algorithm 2 terminates by returning a word $w \in \mathcal{L}(\mathcal{A})$ (hence the termination is guaranteed when $\mathcal{A}$ is not empty);*
- *if Semi-Algorithm 2 terminates by reporting true, then $\mathcal{L}(\mathcal{A}) = \emptyset$ (although the termination is not guaranteed when $\mathcal{A}$ is empty, if the semi-algorithm terminates by reporting true, then $\mathcal{A}$ is surely empty, hence the correctness of the result is guaranteed).*

In order to prove *Theorem 3.2*, we firstly introduce the following lemma:

Lemma 3.4 *Given an ART $\mathcal{T} = (N, E, r, \Lambda, R, T, \lhd)$ built by Semi-Algorithm 2, we have:*

$$POST_{\mathcal{A}}(\Lambda(n), a) \models \Lambda(m), \text{ for all } (n, a, m) \in E$$

Here is the proof of *Lemma 3.4*. We distinguish two cases. First, if (n, a, m) occurs on a path in $\mathcal{T}$ that has never been refined, then $\Lambda(m) = \top$ and the entailment holds trivially. Otherwise, let Ω be the set of paths $\omega = (n_0, a_1, n_1), (n_1, a_2, n_2), ..., (n_{k-1}, a_k, n_k)$, where $n_0 = r$ and $(n, a, m) = (n_{i-1}, a_i, n_i)$, for some $i \in [1, k]$ and, moreover, $a_1, a_2, ..., a_k$ was found, at some point, to be a spurious counter-example. Let $(\top, I_0^\omega, I_1^\omega, ..., I_k^\omega, \bot)$ be an interpolant for $\Phi(a_1, a_2, ..., a_k) \equiv \Lambda(r) \wedge \bigwedge_{i=1}^{k} \theta_i \wedge \bigwedge_{q \in R(n_k)} (q_k \rightarrow \bot)$, such that $I_i^\omega \in FORM^+(Q, X)$, for all $i \in [0, k]$. According to Lyndon's Interpolation Theorem, it is possible to build such an interpolant, when $\Phi(a_1, a_2, ..., a_k)$ is unsatisfiable. By *Proposition 3.2*, we obtain $\Delta^i(I_{i-1}^\omega, a_i)[Q_i/Q] \Leftrightarrow \exists Q_{i-1}.I_{i-1}^\omega[Q_{i-1}/Q, X_{i-1}/X] \wedge \theta_i$ and, since $I_{i-1}^\omega[Q_{i-1}/Q, X_{i-1}/X] \wedge \theta_i \models I_i^\omega[Q_i/Q, X_i/X]$, we obtain that $\Delta^i(I_{i-1}^\omega, a_i)[Q_i/Q] \models I_i^\omega[Q_i/Q, X_i/X]$. Since $\Lambda(n_{i-1}) = \bigwedge_{\omega \in \Omega} I_{i-1}^\omega$ and $\Lambda(n_i) = \bigwedge_{\omega \in \Omega} I_i^\omega$, we obtain $POST_{\mathcal{A}}(\Lambda(n_{i-1}), a_i) \models \Lambda(n_i)$.

Now we can prove *Theorem 3.2*. We prove first that, each time *Semi-Algorithm 2* reaches the *line 10*, we have:

$$W = \{n \mid n \text{ uncovered}, \exists a \in \Sigma\ \forall s \in N.(n, a, s) \notin E\} \tag{1}$$

Initially, $W = \{r\}$ and $E = \lhd = \emptyset$, thus (1) holds trivially. Suppose that (1) holds at when reaching *line 10* and some node n was removed from W and inserted into N. We distinguish two cases, either:

- n is covered, in which case W becomes $W \setminus \{n\}$ and (1) holds,

or

- n is not covered, in which case W becomes $(W \setminus \{n\} \cup S)$, where $S = \{s \notin N \mid (n, a, s) \in E, a \in \Sigma\}$ is the set of fresh successors of n. But then no node $s \in S$ is covered and has successors in E, thus (1) holds.

Then the condition (1) holds next time *line 10* is reached, thus it is invariant.

Suppose first that *Semi-Algorithm 2* returns *true*, thus $W = \emptyset$ and, by (1), for each node in $n \in N$ one of the following hold:

- n is covered,

or

- for each $a \in \Sigma$ there exists $s \in N$ such that $(n, a, s) \in E$.

We prove that, in this case, $\bigvee_{n \in N} \Lambda(n)$ defines a safety invariant and conclude that $\mathcal{L}(\mathcal{A}) = \emptyset$, by *Lemma 3.2*. To this end, let $u = a_1, a_2, ..., a_k \in \Sigma^*$ be an arbitrary sequence and let v_1 be the largest prefix of u that labels a path from r to some node $n_1 \in N$. If $v_1 = u$ we are done. Otherwise, by the choice of v_1, it must be the case that a successor of n_1 is missing from (N, E), thus n_1 must be covered, by (1) and the fact that $W = \emptyset$. Let n_1' be the closest ancestor of n_1 such that $(n_1', n_1'') \in \lhd$, for some $n_1'' \in N$, and let v_1' be the prefix of v_1 leading to n_1'. By the construction of $\lhd$, we have $\Lambda(n_1') \models \Lambda(n_1'')$. Applying *Lemma 3.4* inductively on v_1', we obtain that $POST_{\mathcal{A}}(\iota, v_1') \models \Lambda(n_1')$, thus $POST_{\mathcal{A}}(\iota, v_1') \models \Lambda(n_1'')$. Continuing inductively from n_1'', we exhibit a sequence of strings $v_1', v_2', ..., v_l' \in \Sigma^*$ and nodes $r = m_0, m_1, ..., m_l$ such that, for all $i \in [1, l]$:

- v_i' labels the path between m_{i-1} and m_1 in (N, E);
- $POST_{\mathcal{A}}(\iota, v_1', v_2', ..., v_i') \models \Lambda(m_i)$.

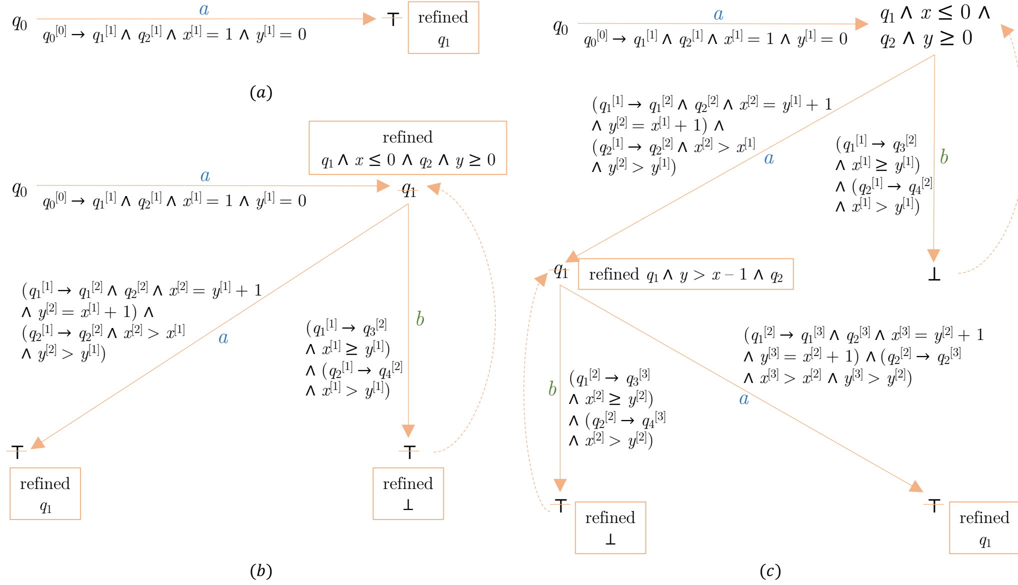
Moreover, we have $u = v_1', v_2', ..., v_k'$, thus $POST_{\mathcal{A}}(\iota, u) \models \Lambda(m_k)$ and we are done showing that $\bigvee_{n \in N} \Lambda(n)$ is an invariant.

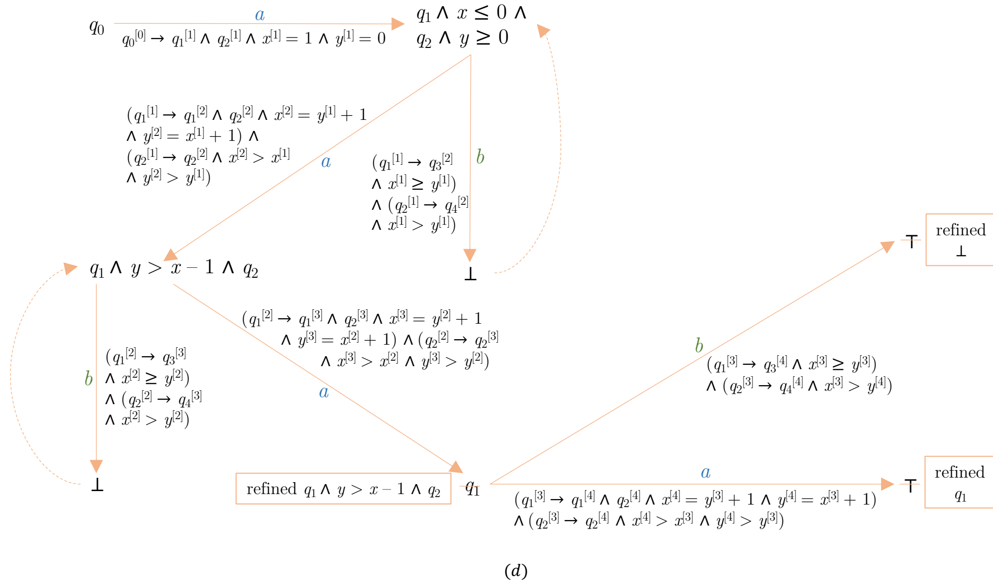
To prove that $\bigvee_{n \in N} \Lambda(n)$ is, moreover, a safety invariant, suppose that $ACC_{\mathcal{A}}(u)$ is satisfiable, for some $u \in \Sigma^*$ and let $n \in N$ be a node such that $POST_A(\iota, u) \models \Lambda(n)$. By the previous point,

such a node must exist. But then $ACC_{\mathcal{A}}(u) \models ACC_{\mathcal{A}}(\lambda(n))$, thus $ACC_{\mathcal{A}}(\lambda(n))$ is satisfiable, and *Semi-Algorithm 2* returns at *line 16*, upon encountering $\lambda(n)$. But this contradicts the assumption that *Semi-Algorithm 2* returns *true*, hence we have proved that $\bigvee_{n \in N} \Lambda(n)$ is a safety invariant, and $\mathcal{L}(\mathcal{A}) = \emptyset$ follows, by *Lemma 3.2*. We have then proved the second point of the statement.

To prove the first point, assume that $\mathcal{L}(\mathcal{A}) \neq \emptyset$. By the previous point, *Semi-Algorithm 2* does not return *true*. Suppose, by contradiction, that it does not terminate and conclude using the breadth-first argument from the proof of *Theorem 3.1*.

Example 3.4 *We show the execution of Semi-Algorithm 2 on the ADA in Figure 3.1. Initially, the procedure fires the sequence a, whose end-point is labelled with* $\top$ *(see Figure 3.3.a). Since this node is uncovered, we check the spuriousness of the counter-example and refine the label of the node to* q_1*. Since the node is still uncovered, two successors labelled with* $\top$ *are computed, corresponding to the sequences aa and ab (see Figure 3.3.b). The spuriousness check for aa yields the interpolant* $(q_0, x \leq 0 \wedge q_2 \wedge y \geq 0))$ *which strengthens the label of the end-point of a from* q_1 *to* $q_1 \wedge x \leq 0 \wedge q_2 \wedge y \geq 0$*. The sequence ab is also found to be spurious, which changes the label of its end-point from* $\top$ *to* $\bot$*, and also covers it (depicted with a dashed edge). Since the end-point of aa is not covered, it is expanded to aaa and aab (see Figure 3.3.c). Both sequences aaa and aab are found to be spurious, and the end-point of aab, whose label has changed from* $\top$ *to* $\bot$*, is now covered. In the process, the label of aa has also changed from* q_1 *to* $q_1 \wedge y > x-1 \wedge q_2$*, due to the strengthening with the interpolant from aab. Finally, the only uncovered node aaa is expanded to aaaa and aaab, both found to be spurious (see Figure 3.3.d). The refinement of aaab causes the label of aaa to change from* q_1 *to* $q_1 \wedge y > x-1 \wedge q_2$ *and this node is now covered by aa. Since its successors are also covered, there are no uncovered nodes and the procedure returns true.*





(d)

Figure 3.3: Proving Emptiness of the ADA in *Figure 3.1* by *Semi-Algorithm 2*

Chapter 4

First-Order Alternating Data Automata (FOADA)

Many results in formal language theory rely on the assumption that languages are defined over finite alphabets. In practice, this assumption is problematic when attempting to use automata as models of real-time systems or even simple programs, whose input and observable output require taking into account data values, ranging over very large domains, better viewed as infinite mathematical abstractions.

Alternating automata are a generalisation of non-deterministic automata with universal transitions, that create several copies of the automaton, which synchronise on the same input word. Alternating automata are appealing for verification because they allow encoding of problems such as temporal logic model checking in linear time, as opposed to the exponential time required by non-deterministic automata [60]. A finite-alphabet alternating automaton is typically described by a set of transition rules $q \xrightarrow{a} \phi$, where q is a state, a is an input symbol and ϕ is a positive Boolean combinations of states, viewed as propositional variables.

In this chapter, we introduce a generalisation of Boolean alternating automata, called *First-Order Alternating Data Automata (FOADA)*, in which transition rules are described by multi-sorted first order formulae, with states and internal variables given by uninterpreted predicate terms. The model is closed under union, intersection and complement, and its emptiness problem is undecidable, even for the simplest data theory of equality. To cope with this limitation, we develop an abstraction refinement semi-algorithm based on lazy annotation of the symbolic execution paths with interpolants, obtained by applying: (i) quantifier elimination with witness term generation and (ii) Lyndon interpolation in the quantifier-free data theory with uninterpreted predicate symbols. This provides a method for checking inclusion of timed and finite-memory register automata, and emptiness of quantified predicate automata, previously used in the verification of parameterised concurrent programs, composed of replicated threads, with a shared-memory communication model.

4.1 Introduction of FOADA

4.1.1 Data Words

Let Σ be a finite alphabet of input events. Given a finite set of variables $X \subseteq \mathit{VAR}$, we denote by $X \mapsto D$ the set of valuations of the variables X and $\Sigma[X] = \Sigma \times (X \mapsto D)$ be the possibly infinite set of data symbols (a, v), where a is an input symbol and v is a valuation.

A **data word** is a finite sequence $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n)$ of data symbols. Given a word w, we denote by $w_\Sigma \stackrel{def}{=\!=} a_1, a_2, ..., a_n$ its sequence of input events and by w_D the valuation associating each time-stamped variable $x^{[i]}$ the value $v_i(x)$, for all $x \in \mathit{VAR}$ and $i \in [1, n]$. We denote by ε the empty sequence, by Σ^* the set of finite sequences of input events and by $\Sigma[X]^*$ the set of data words over the variables X.

4.1.2 Definition of FOADA

A **first-order alternating data automaton (FOADA)** is a tuple $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ where:

- D is a possibly infinite data domain;
- Σ is a finite alphabet of input events;
- $X \subset \mathit{VAR}$ is a finite set of variables of sort D;
- Q is a finite set of predicates denoting control states;
- $\iota \in \mathit{FORM}^+(Q, \emptyset)$ is a sentence defining initial configurations;
- $F \subseteq Q$ is the set of predicates denoting final states;
- Δ is a set of transition rules of the form $q(y_1, y_2, ..., y_{\#(q)}) \xrightarrow{a(X)} \psi$ where $q \in Q$ is predicate, $a \in \Sigma$ is an input event and $\psi \in \mathit{FORM}^+(Q, X \cup \{y_1, y_2, ..., y_{\#(q)}\})$ is a positive formula, where $X \cap \{y_1, y_2, ..., y_{\#(q)}\} = \emptyset$.

The intuition of a transition rule $q(y_1, y_2, ..., y_{\#(q)}) \xrightarrow{a(X)} \psi$ is the following: a is the input event and X are the input data values that trigger the transition, whereas q and $y_1, y_2, ..., y_{\#(q)}$ are the current control state and data values in that state, respectively. Without loss of generality, we consider, for each predicate $q \in Q$ and each input event $a \in \Sigma$, at most one such rule, as two or more rules can be joined using disjunction.

The quantifiers occurring in the right-hand side formula of a transition rule are referred to as **transition quantifiers**. The **size** of $\mathcal{A}$ is defined as $|\mathcal{A}| = |\iota| + \sum_{q(y) \xrightarrow{a(X)} \psi \in \Delta} |\psi|$.

4.1.3 Execution Semantic

The execution semantics of FOADA is given in close analogy with the case of Boolean alternating automata, with transition rules of the form $q \xrightarrow{a} \phi$, where q is a Boolean constant and ϕ a positive Boolean combination of such constants. For instance, $q_0 \xrightarrow{a} q_1 \wedge q_2 \vee q_3$ means that the automaton can choose to transition in either, both q_1 and q_2, or, in q_3 alone. This intuition leads to saying that the steps of the automaton are defined by the minimal Boolean models of the transition formulae. In this case, both $\{q_1 \leftarrow \top, q_2 \leftarrow \top, q_3 \leftarrow \bot\}$ and $\{q_1 \leftarrow \bot, q_2 \leftarrow \bot, q_3 \leftarrow \top\}$ are minimal models, however, $\{q_1 \leftarrow \top, q_2 \leftarrow \top, q_3 \leftarrow \top\}$ is also a model but is not minimal. The original definition of alternating finite-state automata [12] works around this problem by considering Boolean valuations (models) instead of formulae. However, describing FOADA using interpretations instead of formulae would be rather hard to follow.

For a formula ϕ and a valuation v, we define $[[\phi]]_v \stackrel{def}{=\!=} \{\mathcal{I} \mid \mathcal{I}, v \models \phi\}$ and drop the v subscript for sentences. A sentence ϕ is satisfiable if $[[\phi]] \neq \emptyset$ and ϕ is unsatisfiable if $[[\phi]] = \emptyset$. An element of $[[\phi]]$ is called a **model** of ϕ. A formula ϕ is valid if $\mathcal{I}, v \models \phi$ for every interpretation $\mathcal{I}$ and every valuation v. For two formulae ϕ and ψ, we write $\phi \models \psi$ for $[[\phi]] \subseteq [[\psi]]$, in which case we say that ϕ **entails** ψ.

Interpretations are partially ordered by the point-wise subset order, defined as $\mathcal{I}_1 \subseteq \mathcal{I}_2$ if and only if $p^{\mathcal{I}_1} \subseteq p^{\mathcal{I}_2}$ for each predicate $p \in \mathit{PRED}$. Given a set $\mathcal{S}$ of interpretations, a minimal element $\mathcal{I} \in \mathcal{S}$ is an interpretation such that for no other interpretation $\mathcal{I}' \in \mathcal{S} \setminus \{\mathcal{I}\}$ do we have $\mathcal{I}' \subseteq \mathcal{I}$. For a formula ϕ and a valuation v, we denote by $[[\phi]]^\mu_v$ and $[[\phi]]^\mu$ the set of minimal interpretations from $[[\phi]]_v$ and $[[\phi]]$, respectively.

Let $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ be a FOADA. Given a predicate $q \in Q$ and a tuple of data values $d_1, d_2, ..., d_{\#(q)}$, then $q(d_1, d_2, ..., d_{\#(q)})$ is called a **configuration**. To formalise the execution semantics of automata, we relate sets of configurations to models of first-order sentences. Each first-order interpretation $\mathcal{I}$ corresponds to a set of configurations $C(\mathcal{I}) \stackrel{def}{=\!=} \{q(d_1, d_2, ..., d_{\#(q)}) \mid q \in Q, (d_1, d_2, ..., d_{\#(q)}) \in q^{\mathcal{I}}\}$, called a **cube**. For a set $\mathcal{S}$ of interpretations, we define $C(\mathcal{S}) \stackrel{def}{=\!=} \{C(\mathcal{I}) \mid \mathcal{I} \in \mathcal{S}\}$.

Definition 4.1 *Given a word $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n) \in \Sigma[X]^*$ and a cube c, an execution of a FOADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ over w, starting with c, is a possibly infinite forest $\mathcal{T} = \{T_1, T_2, ...\}$, where each T_i is a tree labelled with configurations, such that:*

- *$c = \{T(\epsilon) \mid T \in \mathcal{T}\}$ is the set of configurations labelling the roots of $T_1, T_2, ...$;*
- *if $q(d_1, d_2, ..., d_{\#(q)})$ labels a node on the level $j \in [1, n-1]$ in T_i, then the labels of its children form a cube from $C([[\psi]]^\mu_\eta)$, where $\eta = v_{j+1}[y_1 \leftarrow d_1, y_2 \leftarrow d_2, ..., y_{\#(q)} \leftarrow d_{\#(q)}]$ and $q(y_1, y_2, ..., y_{\#(q)}) \xrightarrow{a_{j+1}(X)} \psi \in \Delta$ is a transition rule of $\mathcal{A}$.*

Definition 4.2 *And an execution $\mathcal{T}$ over w, starting with c, is accepting if and only if:*

- *all paths in $\mathcal{T}$ have the same length n;*

- *the frontier of each tree* $T \in \mathcal{T}$ *is labelled with final configurations* $q(d_1, d_2, ..., d_{\#(q)})$*, where* $q \in F$*.*

If $\mathcal{A}$ *has an accepting execution over* w *starting with a cube* $c \in C([[\iota]]^\mu)$*, then* $\mathcal{A}$ *accepts* w *and let* $\mathcal{L}(\mathcal{A})$ *be the set of words accepted by* $\mathcal{A}$*.*

4.2 Symbolic Execution of FOADA

4.2.1 Path Formulae

In the upcoming developments, it is sometimes more convenient to work with logical formulae defining executions of automata, than with low-level execution forests. For this reason, we first introduce **path formulae** $\Theta(\alpha)$, which are formulae defining the executions of an automaton, over words that share a given sequence α of input events.

Let $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ be a FOADA. For any $i \in \mathbb{N}$, we denote by:

- $Q^{[i]} = \{q^{[i]} \mid q \in Q\}$
- $X^{[i]} = \{x^{[i]} \mid x \in X\}$

the sets of time-stamped predicates and variables, respectively. As a shorthand, we write $Q^{[\leq n]}$ (resp. $X^{[\leq n]}$) for the set $\{q^{[i]} \mid q \in Q, i \in [1, n]\}$ (resp. $\{x^{[i]} \mid x \in X, i \in [1, n]\}$). For a formula ψ and $i \in \mathbb{N}$, we define $\psi^{[i]} \stackrel{def}{=\!=} \psi[X^{[i]}/X, Q^{[i]}/Q]$ the formula in which all input variables and state predicates (and only those symbols) are replaced by their time-stamped counterparts. As a shorthand, we shall write $q(y)$ for $q(y_1, y_2, ..., y_{\#(q)})$ when no confusion arises. Given a sequence of input events $\alpha = a_1, a_2, ..., a_n \in \Sigma^*$, the path formula of α is:

$$\Theta(\alpha) \stackrel{def}{=\!=} \iota^{[0]} \wedge \bigwedge_{i=1}^{n} \bigwedge_{q(y) \xrightarrow{a_i(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[i-1]}(y) \rightarrow \psi^{[i]}$$

The automaton $\mathcal{A}$, to which $\Theta(\alpha)$ refers, will always be clear from the context. To formalise the relation between the low-level configuration-based execution semantics and the symbolic path formulae, consider a word $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n) \in \Sigma[X]^*$. Any execution forest $\mathcal{T}$ of $\mathcal{A}$ over w is associated an interpretation $\mathcal{I}_\mathcal{T}$ of the set of time-stamped predicates $Q^{[\leq n]}$, defined as:

$$\mathcal{I}_\mathcal{T}(q^{[i]}) \stackrel{def}{=\!=} \{(d_1, d_2, ..., d_{\#(q)}) \mid q(d_1, d_2, ..., d_{\#(q)}) \text{ labels a node on } \mathcal{T}^i\}, \forall q \in Q, \forall i \in [1, n]$$

where $\mathcal{T}^i$ refers to the level i in $\mathcal{T}$.

Lemma 4.1 *Given a first-order alternating data automaton* $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$*, for any word* $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n) \in \Sigma[X]^*$*, we have:*

$$[[\Theta(w_\Sigma)]]^\mu_{w_\mathbb{D}} = \{\mathcal{I}_\mathcal{T} \mid \mathcal{T} \text{ is an execution of } \mathcal{A} \text{ over } w\}.$$

Here is the proof of *Lemma 4.1*:

$\subseteq$: Let $\mathcal{I}$ be a minimal interpretation such that $\mathcal{I}, w_D \models \Theta(w_\Sigma)$. We show that there exists an execution $\mathcal{T}$ of $\mathcal{A}$ over w such that $\mathcal{I} = \mathcal{I}_\mathcal{T}$, by induction on $n \geq 0$. For $n = 0$, we have $w = \varepsilon$ and $\Theta(w_\Sigma) = \iota^{[0]}$. Because ι is a sentence, the valuation w_D is not important in $\mathcal{I}, \Theta(w_\Sigma) \models \iota^{[0]}$ and, moreover, since $\mathcal{I}$ is minimal, we have $\mathcal{I} \in [[\iota^{[0]}]]^\mu$. We define the interpretation $\mathcal{J}(q) = \mathcal{I}(q^{[0]})$, for all $q \in Q$. Then $C(\mathcal{J})$ is an execution of $\mathcal{A}$ over ε and $\mathcal{I} = \mathcal{I}_{C(\mathcal{J})}$ is immediate. For the inductive case $n > 0$, we assume that $w = u \cdot (a_n, v_n)$ for a word u. Let $\mathcal{J}$ be the interpretation defined as $\mathcal{I}$ for all $q^{[i]}$, with $q \in Q$ and $i \in [1, n-1]$, and $\emptyset$ everywhere else. Then $\mathcal{J}, u_D \models \Theta(u_\Sigma)$ and $\mathcal{J}$ is moreover minimal. By the induction hypothesis, there exists an execution $\mathcal{G}$ of $\mathcal{A}$ over u, such that $\mathcal{J} = \mathcal{I}_\mathcal{G}$. Consider a leaf of a tree $T \in \mathcal{G}$, labelled with a configuration $q(d_1, d_2, ..., d_{\#(q)})$ and let $\forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[n-1]}(y) \rightarrow \psi^{[n]}$ be the sub-formula of $\Theta(w_\Sigma)$ corresponding to the application(s) of the transition rule $q(y) \xrightarrow{a_n} \psi$ at the $(n-1)$-th step. Let $v = w_D[y_1 \leftarrow d_1, y_2 \leftarrow d_2, ..., y_{\#(q)} \leftarrow d_{\#(q)}]$. Because $\mathcal{I}, w_D \models \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[n-1]}(y) \rightarrow \psi^{[n]}$, we have $\mathcal{I} \in [[\psi^{[n]}]]_v$ and let $\mathcal{K}$ be one of the minimal interpretations such that $\mathcal{K} \subseteq \mathcal{I}$ and $\mathcal{K} \in [[\psi^{[n]}]]_v$. It is not hard to see that $\mathcal{K}$ exists and is unique, otherwise we could take the point-wise intersection of two or more such interpretations. We define the interpretation $\overline{\mathcal{K}}(q) = \overline{\mathcal{K}}(q^{[n]})$ for all $q \in Q$. We have that $\overline{\mathcal{K}} \in [[\psi]]^\mu_v$ if $\overline{\mathcal{K}}$ was not minimal, $\mathcal{K}$ was not minimal to start with, contradiction. Then we extend the execution $\mathcal{G}$ by appending to each node labelled with a configuration $q(d_1, d_2, ..., d_{\#(q)})$ the cube $C(\overline{\mathcal{K}})$. By repeating this step for all leaves of a tree in $\mathcal{G}$, we obtain an execution of $\mathcal{A}$ over w.

$\supseteq$: Let $\mathcal{T}$ be an execution of $\mathcal{A}$ over w. We show that $\mathcal{I}_\mathcal{T}$ is a minimal interpretation such that $\mathcal{I}_\mathcal{T}, w_\mathbb{D} \models \Theta(w_\Sigma)$, by induction on $n \geq 0$. For $n = 0$, $\mathcal{T}$ is a cube from $C([[\iota]]^\mu)$, by definition. Then $\mathcal{I}_\mathcal{T} \models \iota^{[0]}$ and moreover, it is a minimal such interpretation. For the inductive case $n > 0$, let $w = u \cdot (a_n, v_n)$ for a word u. Let $\mathcal{G}$ be the restriction of $\mathcal{T}$ to u. Consequently, $\mathcal{I}_\mathcal{G}$ is the restriction of $\mathcal{I}_\mathcal{T}$ to $Q^{[\leq n-1]}$. By the inductive hypothesis, $\mathcal{I}_\mathcal{G}$ is a minimal interpretation such that $\mathcal{I}_\mathcal{G}, u_\mathbb{D} \models \Theta(u_\Sigma)$. Since $\mathcal{I}_\mathcal{T}(q^{[n]}) = \{(d_1, d_2, ..., d_{\#(q)}) \mid q(d_1, d_2, ..., d_{\#(q)}) \text{ labels a node on the } n\text{-th level in } \mathcal{T}\}$, we have $\mathcal{I}_\mathcal{T}, w_\mathbb{D} \models \varphi$, for each sub-formula $\varphi = \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[n-1]}(y) \rightarrow \psi^{[n]}$ of $\Theta(w_\Sigma)$, by the execution semantics of $\mathcal{A}$. This is the case because the children of each node labelled with $q(d_1, d_2, ..., d_{\#(q)})$ on the $(n-1)$-th level of $\mathcal{T}$ form a cube from $C([[\psi]]^\mu_v)$, where v is a valuation that assigns each y_i the value d_i and behaves like $w_\mathbb{D}$, otherwise. Now suppose, for a contradiction, that $\mathcal{I}_\mathcal{T}$ is not minimal and let $\mathcal{J} \subsetneq \mathcal{I}_\mathcal{T}$ be an interpretation such that $\mathcal{J}, w_\mathbb{D} \models \Theta(w_\Sigma)$. First, we show that the restriction $\mathcal{J}'$ of $\mathcal{J}$ to $\bigcup_{i=0}^{n-1} Q^{[i]}$ must coincide with $\mathcal{I}_\mathcal{G}$. Assuming this is not the case, i.e. $\mathcal{J}' \subsetneq \mathcal{I}_\mathcal{G}$, contradicts the minimality of $\mathcal{I}_\mathcal{G}$. Then the only possibility is that $\mathcal{J}(q^{[n]}) \subsetneq \mathcal{I}_\mathcal{T}(q^{[n]})$, for some $q \in Q$. Let $p_1(y_1, y_2, ..., y_{\#(p_1)}) \xrightarrow{a_n} \psi_1, p_2(y_1, y_2, ..., y_{\#(p_2)}) \xrightarrow{a_n} \psi_2, ..., p_k(y_1, y_2, ..., y_{\#(p_k)}) \xrightarrow{a_n} \psi_k$ be the set of transition rules in which the predicate symbol q occurs on the right-hand side. Then it must be the case that, for some node on the $(n-1)$-th level of $\mathcal{G}$, labelled with a configuration $p_i(d_1, d_2, ..., d_{\#(p_i)})$, the set of children does not form a minimal cube from $C([[\psi_i^{[n]}]]^\mu)$, which contradicts the execution semantics of $\mathcal{A}$.

4.2.2 Acceptance Formulae

Now we give a logical characterisation of acceptance, relative to given sequence of input events $\alpha \in \Sigma^*$. To this end, we constrain the path formula $\Theta(\alpha)$ by requiring that only final states of $\mathcal{A}$ occur on the last level of the execution. The result is the **acceptance formula** for α:

$$\Upsilon(\alpha) \stackrel{def}{=\!=} \Theta(\alpha) \wedge \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q^{[n]}(y) \rightarrow \bot$$

The top-level universal quantifiers from a sub-formula $\forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q^{[i]}(y) \rightarrow \psi$ of $\Upsilon(\alpha)$ will be referred to as **path quantifiers**, in the following. Notice that path quantifiers are distinct from the transition quantifiers that occur within a formula ψ of a transition rule $q(y_1, y_2, ...y_{\#(q)}) \xrightarrow{a(X)} \psi$ of $\mathcal{A}$.

The acceptance formula $\Upsilon(\mathcal{A})$ is false in every interpretation of the predicates that assigns a non-empty set to a non-final predicate occurring on the last level in the execution forest. The relation between the words accepted by $\mathcal{A}$ and the acceptance formula above, is formally captured by the lemma below.

Lemma 4.2 *Given an automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, for every word $w \in \Sigma[X]^*$, the following are equivalent:*

(1) there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}, w_{\mathbb{D}} \models \Upsilon(w_\Sigma)$;

(2) $w \in \mathcal{L}(\mathcal{A})$.

Here is the proof of *Lemma 4.2*:

1⇒2 Let $\mathcal{I}$ be an interpretation such that $\mathcal{I}, w_{\mathbb{D}} \models \Upsilon(w_\Sigma)$. We know that $\mathcal{A}$ has an execution $\mathcal{T}$ over w such that $\mathcal{I} = \mathcal{I}_{\mathcal{T}}$. To prove that $\mathcal{T}$ is accepting, we show that (i) all paths in $\mathcal{T}$ have length n and that (ii) the frontier of $\mathcal{T}$ is labelled with final configurations only. First, assume that (i) there exists a path in $\mathcal{T}$ of length $0 \leq m < n$. Then there exists a node on the m-th level, labelled with some configuration $q(d_1, d_2, ..., d_{\#(q)})$, that has no children. By the definition of the execution semantics of $\mathcal{A}$, we have $C([[\psi]]^{\mu}_{\eta}) = \emptyset$, where $q(y) \xrightarrow{a_{m+1}(X)} \psi$ is the transition rule of $\mathcal{A}$ that applies for q and a_{m+1} and $\eta = w_{\mathbb{D}}[y_1 \leftarrow d_1, y_2 \leftarrow d_2, ..., y_{\#(q)} \leftarrow d_{\#(q)}]$. Hence $[[\psi]]_\eta = \emptyset$, and because $\mathcal{I}, w_{\mathbb{D}} \models \Upsilon(\alpha)$, we obtain that $\mathcal{I}, \eta \models q(y) \rightarrow \psi^{[m+1]}$, thus $(d_1, d_2, ..., d_{\#(q)}) \notin \mathcal{I}(q)$. However, this contradicts the fact that $\mathcal{I} = \mathcal{I}_{\mathcal{T}}$ and that $q(d_1, d_2, ..., d_{\#(q)})$ labels a node of $\mathcal{T}$. Second, assume that (ii), there exists a frontier node of $\mathcal{T}$ labelled with a configuration $q(d_1, d_2, ..., d_{\#(q)})$ such that $q \in Q \setminus F$. Since $\mathcal{I}, w_{\mathbb{D}} \models \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q(y) \rightarrow \bot$, by a similar reasoning as in the above case, we obtain that $(d_1, d_2, ..., d_{\#(q)}) \notin \mathcal{I}(q)$, contradiction.

2⇒1 Let $\mathcal{T}$ be an accepting execution of $\mathcal{A}$ over w. We can prove that $\mathcal{I}_{\mathcal{T}}, w_{\mathbb{D}} \models \Upsilon(w_\Sigma)$. By *Lemma 4.1*, we obtain $\mathcal{I}_{\mathcal{T}}, w_{\mathbb{D}} \models \Theta(w_\Sigma)$. Since every path in $\mathcal{T}$ is of length n and all nodes on the n-th level of $\mathcal{T}$ are labelled by final configurations, we can here obtain that $\mathcal{I}_{\mathcal{T}}, w_{\mathbb{D}} \models \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q^{[n]}(y) \rightarrow \bot$, trivially.

As an immediate consequence, one can decide whether $\mathcal{A}$ accepts some word w with a given input sequence $w_\Sigma = \alpha$, by checking whether $\Upsilon(\alpha)$ is satisfiable. However, unlike non-alternating infinite-state models of computation, such as counter automata (non-deterministic programs with integer variables), the satisfiability query for an acceptance (path) formula falls outside of known decidable theories, supported by standard SMT solvers. There are basically two reasons for this, namely (i) the presence of predicate symbols, and (ii) the non-trivial alternation of quantifiers. To understand this point, consider for example, the decidable theory of Presburger arithmetic [51]. Adding even only one monadic predicate symbol to it yields undecidability in the presence of non-trivial quantifier alternation [32]. However the quantifier-free fragment of Presburger arithmetic extended with predicate symbols can be shown to be decidable, using a Nelson-Oppen style congruence closure argument [47].

To tackle this problem, we start from the observation that acceptance formulae have a particular form, which allows the elimination of path quantifiers and of predicates, by a couple of satisfiability-preserving transformations. The result of applying these transformations is a formula with no predicate symbols, whose only quantifiers are those introduced by the transition rules of the automaton, referred to as transition quantifiers. We shall further assume that the first order theory of the data sort $\mathcal{D}$ has quantifier elimination, which allows to effectively decide the satisfiability of such formulae. The next two sections introduce the elimination of path quantifiers and predicates.

4.2.3 Elimination of Path Quantifiers

Consider a given sequence of input events $\alpha = a_1, a_2, ..., a_n$ and denote by α_i the prefix $a_1, a_2, ..., a_i$ of α for $i \in [1, n]$ where $\alpha_0 = \varepsilon$.

Definition 4.3 *Let $\widehat{\Theta}(\alpha_0), \widehat{\Theta}(\alpha_1), ..., \widehat{\Theta}(\alpha_n)$ be the sequence of formulae defined by:*

- $\widehat{\Theta}(\alpha_0) \stackrel{def}{=\!=} \iota^{[0]}$;
- $\widehat{\Theta}(\alpha_i) \stackrel{def}{=\!=} \widehat{\Theta}(\alpha_{i-1}) \wedge \bigwedge_{cond1, cond2} q^{[i-1]}(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi^{[i]}[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$
  for $i \in [1, n]$
  where cond1: $q^{[i-1]}(t_1, t_2, ..., t_{\#(q)})$ occurs in $\widehat{\Theta}(\alpha_{i-1})$
  and cond2: $q(y_1, y_2, ..., y_{\#(q)}) \xrightarrow{a_i(X)} \psi \in \Delta$

We write $\widehat{\Upsilon}(\alpha)$ for the prenex normal form of the formula:

$$\widehat{\Theta}(\alpha_n) \wedge \bigwedge_{q^{[n]}(t_1, t_2, ..., t_{\#(q)}) \text{ occurs in } \widehat{\Theta}(\alpha_n), q \in Q \setminus F} q^{[n]}(t_1, t_2, ..., t_{\#(q)}) \rightarrow \bot$$

Observe that $\widehat{\Upsilon}(\alpha)$ contains no path quantifiers, as required. On the other hand, the scope of the transition quantifiers in $\widehat{\Upsilon}(\alpha)$ exceeds the right-hand side formulae from the transition rules, as shown by the following example.

Example 4.1 *Consider an automaton* $\mathcal{A} = (\mathbb{N}, \{a_1, a_2\}, \{x\}, \{q, q_f\}, \iota, \{q_f\}, \Delta)$ *where:*

- $\iota = \exists z.z \geq 0 \wedge q(z)$;
- $\Delta = \{q(y) \xrightarrow{a_1(x)} x \geq 0 \wedge \forall z.z \leq y \rightarrow q(x+z), q(y) \xrightarrow{a_2(x)} y < 0 \wedge q_f(x+y)\}$.

For the input event sequence $\alpha = a_1 a_2$, *the acceptance formula is:*

$$\begin{aligned}\Upsilon(\alpha) =& \exists z.z \geq 0 \wedge q^{[0]}(z) \wedge \\ & \forall y.q^{[0]}(y) \rightarrow [x^{[1]} \geq 0 \wedge \forall z.z \geq y \rightarrow q^{[1]}(x^{[1]} + z)] \wedge \\ & \forall y.q^{[1]}(y) \rightarrow [y < 0 \wedge q_f^{[2]}(x^{[2]} + y)]\end{aligned}$$

The result of eliminating the path quantifiers, in prenex normal form, is shown below:

$$\begin{aligned}\widehat{\Upsilon}(\alpha) =& \exists z_1 \forall z_2.z_1 \geq 0 \wedge q^{[0]}(z_1) \\ & [q^{[0]}(z_1) \rightarrow x^{[1]} \geq 0 \wedge (z_2 \geq z_1 \rightarrow q^{[1]}(x^{[1]} + z_2))] \wedge \\ & [q^{[1]}(x^{[1]} + z_2) \rightarrow x^{[1]} + z_2 < 0 \wedge q_f^{[2]}(x^{[2]} + x^{[1]} + z_2)]\end{aligned}$$

Now we show a formal relation between the satisfiability of an acceptance formula $\Upsilon(\alpha)$ and that of the formula $\widehat{\Upsilon}(\alpha)$, obtained by eliminating the path quantifiers from $\Upsilon(\alpha)$.

Lemma 4.3 *For any input event sequence* $\alpha = a_1, a_2, ..., a_n$ *and each valuation* $v : X^{[\leq n]} \rightarrow \mathbb{D}$, *the following hold:*

- *for all interpretations* $\mathcal{I}$, *if* $\mathcal{I}, v \models \Upsilon(\alpha)$ *then* $\mathcal{I}, v \models \widehat{\Upsilon}(\alpha)$;
- *if there exists an interpretation* $\mathcal{I}$ *such that* $\mathcal{I}, v \models \widehat{\Upsilon}(\alpha)$ *then there exists an interpretation* $\mathcal{J} \subseteq \mathcal{I}$ *such that* $\mathcal{J}, v \models \Upsilon(\alpha)$.

Here is the proof of *Lemma 4.3*:

(1) Trivial, since every sub-formula $q(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$ of $\widehat{\Upsilon}$ is entailed by a sub-formula $\forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q(y_1, y_2, ..., y_{\#(q)}) \rightarrow \psi$ of $\Upsilon(\alpha)$.

(2) By repeated applications of *Fact 4.1*.

Fact 4.1 *Given formulae* ϕ *and* ψ, *such that no predicate atom with predicate symbol* q *occurs in* $\psi(y_1, y_2, ..., y_{\#(q)})$, *for each valuation* v, *if there exists an interpretation* $\mathcal{I}$ *such that* $\mathcal{I}, v \models \phi \wedge \bigwedge_{q(t_1, t_2, ..., t_{\#(q)}) \text{ occurs in } \phi} q(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$ *then there exists a valuation* $\mathcal{J}$ *such that* $\mathcal{J}(q) \subseteq \mathcal{I}(q)$ *and* $\mathcal{J}(q') \subseteq \mathcal{I}(q')$ *for all* $q' \in Q \setminus \{q\}$ *and* $\mathcal{J}, v \models \phi \wedge \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q(y_1, y_2, ..., y_{\#(q)}) \rightarrow \psi$.

Here is the proof of *Fact 4.1*:

Assume w.l.o.g. that ϕ is quantifier free. The proof can be easily generalised to the case where ϕ has quantifiers. Let $\mathcal{J}(q) = \{(t_1^v, t_2^v, ..., t_{\#(q)}^v) \in \mathcal{I}(q) \mid q(t_1, t_2, ..., t_{\#(q)}) \text{ occurs in } \phi\}$ and $\mathcal{J}(q') \subseteq \mathcal{I}(q')$ for all $q' \in Q \setminus \{q\}$. Since $\mathcal{I}, v \models \phi$, we obtain also that $\mathcal{J}, v \models \phi$ because the tuples of values in $\mathcal{I}(q) \setminus \mathcal{J}(q)$ are not interpretations of terms that occur within sub-formulae $q(t_1, t_2, ..., t_{\#(q)})$ of ϕ. Moreover, two formulae:

(1) $\bigwedge_{q(t_1,t_2,...,t_{\#(q)}) \text{ occurs in } \phi} q(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$

(2) $\forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q(y_1, y_2, ..., y_{\#(q)}) \rightarrow \psi$

(1) and (2) are equivalent under $\mathcal{J}$, thus $\mathcal{J}, v \models \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q(y_1, y_2, ..., y_{\#(q)}) \rightarrow \psi$, as required. This concludes the proof.

4.2.4 Elimination of Predicate Atoms

We proceed with the elimination of predicate atoms from $\widehat{\Upsilon}(\alpha)$ defined below.

Definition 4.4 *Let $\widetilde{\Theta}(\alpha_0), \widetilde{\Theta}(\alpha_1), ..., \widetilde{\Theta}(\alpha_n)$ be the sequence of formulae defined by $\widetilde{\Theta}(\alpha_0) \overset{def}{=\!=} \iota^{[0]}$ and, for all $i \in [1, n]$, $\widetilde{\Theta}(\alpha_i)$ is obtained by replacing each occurrence of a predicate atom $q^{[i-1]}(t_1, t_2, ..., t_{\#(q)})$ in $\widetilde{\Theta}(\alpha_{i-1})$ by the formula $\psi^{[i]}[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$, where $q(y) \xrightarrow{a_i(X)} \psi \in \Delta$. We write $\widetilde{\Upsilon}(\alpha)$ for the formula obtained by replacing, in $\widetilde{\Theta}(\alpha)$, each occurrence of a predicate $q^{[n]}$, such that $q \in Q \setminus F$ (resp. $q \in F$), by $\bot$ (resp. $\top$).*

Example 4.2 *The result of the elimination of predicate atoms from the acceptance formula in Example 4.1 is shown below:*

$$\widetilde{\Upsilon}(\alpha) = \exists z_1 \forall z_2.z_1 \geq 0 \wedge [x^{[1]} \geq 0 \wedge (z_2 \geq z_1 \rightarrow x^{[1]} + z_2 < 0)]$$

Since this formula is unsatisfiable, no word w with input event sequence $w_\Sigma = a_1 a_2$ is accepted by the automaton $\mathcal{A}$ from Example 4.1.

At this point, we prove the formal relation between the satisfiability of the formulae $\widehat{\Upsilon}(\alpha)$ and $\widetilde{\Upsilon}(\alpha)$. Since there are no occurrences of predicates in $\widetilde{\Upsilon}(\alpha)$, for each valuation $v : X^{[\leq n]} \rightarrow \mathbb{D}$, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}, v \models \widehat{\Upsilon}(\alpha)$ if and only if $\mathcal{J}, v \models \widetilde{\Upsilon}(\alpha)$, for every interpretation $\mathcal{J}$. In this case we omit $\mathcal{I}$ and simply write $v \models \widetilde{\Upsilon}(\alpha)$.

Lemma 4.4 *For any input event sequence $\alpha = a_1, a_2, ..., a_n$ and each valuation $v : X^{[\leq n]} \rightarrow \mathbb{D}$, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}, v \models \widehat{\Upsilon}(\alpha)$ if and only if $v \models \widetilde{\Upsilon}(\alpha)$.*

Here is the proof of *Lemma 4.4* by induction on $n \geq 0$:

- The base case $n = 0$ is trivial, since $\widehat{\Upsilon}(A) = \widetilde{\Upsilon}(A) = \iota^{[0]}$.

- For the induction step, we rely on *Fact 4.2*.

Fact 4.2 *Given formulae ϕ and ψ, such that ϕ is positive, $q(t_1, t_2, ..., t_{\#(q)})$ is the only one occurrence of the predicate symbol q in ϕ and no predicate atom with predicate symbol q occurs in $\psi(y_1, y_2, ..., y_{\#(q)})$, for each interpretation $\mathcal{I}$ and each valuation v, we have:*

$$\mathcal{I}, v \models \phi \wedge q(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$$

$$\Leftrightarrow$$

$$v \models \phi[\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]/q(t_1, t_2, ..., t_{\#(q)})].$$

Here is the proof of *Fact 4.2*. We assume w.l.o.g. that ϕ is quantifier-free. The proof can be easily generalised to the case ϕ has quantifiers:

$\Rightarrow$ We distinguish two cases:

- if $(t_1^v, t_2^v, ..., t_{\#(q)}^v) \in \mathcal{I}(q)$ then $\mathcal{I}, v \models \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$. Since ϕ is positive, replacing $q(t_1, t_2, ..., t_{\#(q)})$ with $\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$ does not change the truth value of ϕ under v, thus:
$$v \models \phi[\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]/q(t_1, t_2, ..., t_{\#(q)})];$$
- else, $(t_1^v, t_2^v, ..., t_{\#(q)}^v) \notin \mathcal{I}(q)$, thus $v \models \phi[\perp/q(t_1, t_2, ..., t_{\#(q)})]$. Since ϕ is positive and $\perp$ entails $\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$, we obtain:
$$v \models \phi[\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]/q(t_1, t_2, ..., t_{\#(q)})]$$
by monotonicity.

$\Leftarrow$ Let $\mathcal{I}(q) = \{(t_1^v, t_2^v, ..., t_{\#(q)}^v) \mid v \models \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]\}$. We distinguish two cases:

- if $\mathcal{I}(q) \neq \emptyset$, then $\mathcal{I}, v \models q(t_1, t_2, ..., t_{\#(q)})$ and $v \models \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$. Thus replacing $\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$ by $q(t_1, t_2, ..., t_{\#(q)})$ does not change the truth value of ϕ under $\mathcal{I}$ and v, and we obtain $\mathcal{I}, v \models \phi$. Moreover, $\mathcal{I}, v \models \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$ implies $\mathcal{I}, v \models q(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$.
- else $\mathcal{I}(q) = \emptyset$, hence $v \not\models \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$, thus $v \models \phi[\perp/q(t_1, t_2, ..., t_{\#(q)})]$. Because ϕ is positive, we obtain $\mathcal{I}, v \models \phi$ by monotonicity. But $\mathcal{I}, v \models q(t_1, t_2, ..., t_{\#(q)}) \rightarrow \psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]$ trivially, because $\mathcal{I}, v \not\models q(t_1, t_2, ..., t_{\#(q)})$.

Finally, we define the acceptance of a word with a given input event sequence by means of a formula in which no predicate atom occurs. As previously discussed, several decidable theories, such as Presburger arithmetic, become undecidable if predicate atoms are added to them. Therefore, the result below makes a step forward towards deciding whether the automaton accepts a word with a given input sequence, by reducing this problem to the satisfiability of a quantified formula without predicates.

Lemma 4.5 *Given an automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$, for every word $w \in \Sigma[X]^*$, we have $w_D \models \widetilde{\Upsilon}(w_\Sigma)$ if and only if $w \in \mathcal{L}(\mathcal{A})$.*

Here is the proof of *Lemma 4.5*:

- By *Lemma 4.2*, $w \in \mathcal{L}(\mathcal{A})$ if and only if $\mathcal{I}, w_D \models \Upsilon(w_\Sigma)$, for some interpretation $\mathcal{I}$;
- By *Lemma 4.3* there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}, w_D \models \Upsilon(w_\Sigma)$ if and only if there exists an interpretation $\mathcal{J}$ such that $\mathcal{J}, v \models \widehat{\Upsilon}(w_\Sigma)$;
- By *Lemma 4.4* there exists an interpretation $\mathcal{J}$ such that $\mathcal{J}, v \models \widehat{\Upsilon}(w_\Sigma)$ if and only if $v \models \widetilde{\Upsilon}(w_\Sigma)$.

4.3 Closure Properties of FOADA

Given a positive formula ϕ, we define the dual formula $\overline{\phi}$ recursively as follows:

- $\overline{\phi_1 \vee \phi_2} = \overline{\phi_1} \wedge \overline{\phi_2}$
- $\overline{\phi_1 \wedge \phi_2} = \overline{\phi_1} \vee \overline{\phi_2}$
- $\overline{t \approx s} = \neg(t \approx s)$
- $\overline{\neg(t \approx s)} = (t \approx s)$
- $\overline{\exists x.\phi_1} = \forall x.\overline{\phi_1}$
- $\overline{\forall x.\phi_1} = \exists x.\overline{\phi_1}$
- $\overline{q(x_1, x_2, ..., x_{\#(q)})} = q(x_1, x_2, ..., x_{\#(q)})$

Observe that, because predicate atoms do not occur negated in ϕ, there is no need to define dualisation for formulae of the form $\neg q(x_1, x_2, ..., x_{\#(q)})$. The following theorem shows closure of automata under all Boolean operations.

Theorem 4.1 *Given two automata $\mathcal{A}_1 = (D, \Sigma, X, Q_1, \iota_1, F_1, \Delta_1)$ and $\mathcal{A}_2 = (D, \Sigma, X, Q_2, \iota_2, F_2, \Delta_2)$, such that $Q_1 \cap Q_2 = \emptyset$, the following hold:*

- *$\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, where $A_\cap = (D, \Sigma, X, Q_1 \cup Q_2, \iota_1 \wedge \iota_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$;*
- *$\mathcal{L}(\overline{\mathcal{A}_i}) = \Sigma[X]^* \setminus \mathcal{L}(\mathcal{A}_i)$, where $\overline{\mathcal{A}_i} = (D, \Sigma, X, Q_i, \overline{\iota_i}, Q_i \setminus F_i, \overline{\Delta_i})$ and for $i = 1, 2$: $\overline{\Delta_i} = \{q(y) \xrightarrow{a(X)} \overline{\psi} \mid q(y) \xrightarrow{a(X)} \psi \in \Delta_i\}$.*

Moreover, $|\mathcal{A}_\cap| = \mathcal{O}(|\mathcal{A}_1| + |\mathcal{A}_2|)$ and $|\overline{\mathcal{A}_i}| = \mathcal{O}(|\mathcal{A}_i|)$ for all $i = 1, 2$.

Here is the proof of *Theorem 4.1*:

(1) $\subseteq$ Let $w \in \mathcal{L}(\mathcal{A}_\cap)$ be a word and $\mathcal{T}$ be an execution of $\mathcal{A}_\cap$ over w. Since $Q_1 \cap Q_2 = \emptyset$, it is possible to partition $\mathcal{T}$ into $\mathcal{T}_1$ and $\mathcal{T}_2$ such that the roots of $\mathcal{T}_i$ form a cube from $C([[\iota_i]]^\mu)$, for all $i = 1, 2$. Because $\Delta_1 \cap \Delta_2 = \emptyset$, by induction on $|w| \geq 0$, one shows that $\mathcal{T}_i$ is an execution of $\mathcal{A}_i$ over w, for all $i = 1, 2$. Finally, because $\mathcal{T}$ is accepting, we obtain that $\mathcal{T}_i$ and $\mathcal{T}_2$ are accepting, respectively, hence $w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

$\supseteq$ Let $w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ and let $\mathcal{T}_i$ an accepting execution of $\mathcal{A}_i$ over w, for all $i = 1, 2$. We show that $\mathcal{T}_1 \cup \mathcal{T}_2$ is an execution of $\mathcal{A}_\cap$ over w, by induction on $|w| \geq 0$. For the base case $|w| = 0$, we have $\mathcal{T}_i \in C([[\iota_i]]^\mu)$ for all $i = 1, 2$ and since $Q_1 \cap Q_2 = \emptyset$, we have $\mathcal{T}_i \cup \mathcal{T}_2 \in C([[\iota_1 \wedge \iota_2]]^\mu))$. The induction step follows as a consequence of the fact that $\Delta_1 \cup \Delta_2$ is the set of transition rules of $\mathcal{A}_\cap$. Finally, since both $\mathcal{T}_1$ and $\mathcal{T}_2$ are accepting, $\mathcal{T}_i \cup \mathcal{T}_2$ is accepting as well. Moreover, we have:

$$|\mathcal{A}_\cap| = |\iota_1 \wedge \iota_2| + \sum_{q(y) \xrightarrow{a(X)} \psi \in \Delta_1 \cup \Delta_2} |\psi| = 1 + |\iota_1| + |\iota_2| + \sum_{q(y) \xrightarrow{a(X)} \psi \in \Delta_1} |\psi| + \sum_{q(y) \xrightarrow{a(X)} \psi \in \Delta_2} |\psi|$$

(2) Let $w \in \Sigma[X]^*$ be a word. We denote by $\Upsilon_{\mathcal{A}_1}(w_\Sigma)$ and $\widetilde{\Upsilon}_{\mathcal{A}_1}(w_\Sigma)$ (resp. $\Upsilon_{\overline{\mathcal{A}_1}}(w_\Sigma)$ and $\widetilde{\Upsilon}_{\overline{\mathcal{A}_1}}(w_\Sigma)$) the formulae $\Upsilon(w_\Sigma)$ and $\overline{\Upsilon}(w_\Sigma)$ for $\mathcal{A}_1$ and $\overline{\mathcal{A}_1}$, respectively. It is enough to show that $\widetilde{\Upsilon}_{\overline{\mathcal{A}_1}}(w_\Sigma) = \neg\Upsilon_{\mathcal{A}_1}(w_\Sigma)$ and apply *Lemma 4.5* to prove that $w \in \mathcal{L}(\mathcal{A}_1) \Leftrightarrow w \notin \mathcal{L}(\overline{\mathcal{A}_1})$. Since the choice of w was arbitrary, this proves $\mathcal{L}(\overline{\mathcal{A}_1}) = \Sigma[X]^* \setminus \mathcal{L}(\mathcal{A}_1)$. By induction on the number of predicate atoms in $\Upsilon_{\mathcal{A}_1}(w_\Sigma)$ that are replaced during the generation of $\widetilde{\Upsilon}_{\mathcal{A}_1}(w_\Sigma)$. The proof relies on the following fact:

Fact 4.3 *Let ϕ be a positive formula and let $q(t_1, t_2, ..., t_{\#(q)})$ be the only occurrence of a predicate symbol within ϕ. Then, every formula ϕ with no predicate occurrences:*

$$\neg\phi[\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]/q(t_1, t_2, ..., t_{\#(q)})]$$

$$\equiv$$

$$\overline{\phi}[\neg\psi[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]/q(t_1, t_2, ..., t_{\#(q)})]$$

The proof of *Fact 4.3* can be done by induction on the structure of ϕ.

4.4 Emptiness Problem of FOADA

4.4.1 Unfoldings of FOADA

Given a finite input event alphabet Σ, for two sequences $\alpha, \beta \in \Sigma^*$, we say that α is a prefix of β, written $\alpha \preceq \beta$, if $\alpha = \beta\gamma$ for some sequence $\gamma \in \Sigma^*$. A set S of sequences is:

- **prefix-closed** if for each $\alpha \in S$, if $\beta \preceq \alpha$ then $\beta \in S$;

- **complete** if for each $\alpha \in S$, there exists $\alpha \in \Sigma$ such that $\alpha a \in S$ if and only if $\alpha b \in S$ for all $b \in \Sigma$.

Definition 4.5 *An **unfolding** of a first-order alternating data automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ is a finite partial mapping $U : \Sigma^* \rightarrow_{fin} FORM^+(Q, \emptyset)$, such that:*

- *$DOM(U)$ is a finite prefix-closed complete set;*
- *$U(\varepsilon) = \iota$;*
- *for each sequence $\alpha a \in DOM(U)$, such that $\alpha \in \Sigma^*$ and $a \in \Sigma$:*

$$U(\alpha)^{[0]} \wedge \bigwedge_{q(y) \xrightarrow{a(X)} \psi} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[0]}(y) \rightarrow \psi^{[1]} \models U(\alpha a)^{[1]}$$

Moreover, U is safe if for each $\alpha \in DOM(U)$, the formula $U(\alpha) \wedge \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q(y) \rightarrow \bot$ is unsatisfiable.

4.4.2 IMPACT Semi-Algorithm

The problem of checking emptiness of a given automaton is undecidable, even for automata with predicates of arity two, whose transition rules use only equality and dis-equality, having no transition quantifiers [26]. Since even such simple classes of alternating automata have no general decision procedure for emptiness, we use an abstraction refinement semi-algorithm based on lazy annotation [45, 46].

In a nutshell, a lazy annotation procedure systematically explores the set of execution paths (in our case, sequences of input events) in search of an accepting execution. Each path has a corresponding path formula that defines all words accepted along that path. If the path formula is satisfiable, the automaton accepts a word. Otherwise, the path is said to be spurious. When a spurious path is encountered, the search backtracks and the path is annotated with a set of learned facts, that marks this path as infeasible. The semi-algorithm uses moreover a coverage relation between paths, ensuring that the continuations of already covered paths are never explored. Sometimes this coverage relation provides a sound termination argument, when the automaton is empty.

We check emptiness of first order alternating automata using a version of the IMPACT lazy annotation semi-algorithm [45].

Semi-Algorithm 3 IMPACT for FOADA Emptiness

Input: a FOADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$

Output: $\begin{cases} true & \text{if } \mathcal{L}(\mathcal{A}) = \emptyset \\ \text{a data word } w \in \mathcal{L}(\mathcal{A}) & \text{if } \mathcal{L}(\mathcal{A}) \neq \emptyset \end{cases}$

1: let $\mathcal{U} = (N, E, r, U, \lhd)$ be an unfolding tree
2: let *WorkList* be a list

3: $N := \emptyset$
4: $E := \emptyset$
5: $U := \emptyset$
6: $\lhd := \emptyset$
7: add r into *WorkList*
8: **while** $\mathit{WorkList} \neq \emptyset$ **do**
9: dequeue n from *WorkList*
10: $N := N \cup \{n\}$
11: let $\alpha(n)$ be $a_1, a_2, ..., a_k$
12: **if** $\widetilde{\Upsilon}(\alpha)(X^{[1]}, X^{[2]}, ..., X^{[k]})$ is satisfiable **then**
13: get model v of $\widetilde{\Upsilon}(\alpha)(X^{[1]}, X^{[2]}, ..., X^{[k]})$
14: **return** $w = (a_1, v(X^{[1]})), (a_2, v(X^{[2]})), ..., (a_k, v(X^{[k]}))$
15: **else**
16: let $(I_0, I_1, ..., Ik)$ be a GLI (generalised Lyndon interpolant) for α
17: $b := \bot$
18: **for** $i \in [0, k]$ **do**
19: **if** $U(n_i) \not\models I_i$ **then**
20: $\mathit{Uncover} := \{m \in N \mid (m, n_i) \in \lhd\}$
21: $\lhd := \lhd \setminus \{(m, n_i) \in \lhd \mid m \in \mathit{Uncover}\}$
22: **for** $m \in \mathit{Uncover}$ such that m is a leaf of $\mathcal{U}$ **do**
23: enqueue m into *WorkList*
24: **end for**
25: $U(n_i) := U(n_i) \wedge J_i$ % see Section - Interpolant Generation of FOADA
26: **if** $\neg b$ **then**
27: $b := CLOSE(n_i)$
28: **end if**
29: **end if**
30: **end for**
31: **end if**
32: **if** n is not covered **then**
33: **for** $a \in \Sigma$ **do**
34: let s be a fresh node
35: let $e = (n, a, s)$ be a new edge
36: $E := E \cup \{e\}$
37: $U := U \cup \{(s, \top)\}$
38: enqueue s into *WorkList*
39: **end for**
40: **end if**
41: **end while**
42: **return** *true*

Function 2 CLOSE

Input: a node x

Output: $\begin{cases} true & \text{if } x \text{ is closed} \\ false & \text{if } x \text{ is not closed} \end{cases}$

1: **for** $y \in N$ such that $\alpha(y) \prec^* \alpha(x)$ **do**
2: **if** $U(x) \models U(y)$ **then**
3: $\lhd := (\lhd \setminus \{(p,q) \in \lhd \mid q \text{ is } x \text{ or a successor of } x\}) \cup \{(x,y)\}$
4: **return** $true$
5: **end if**
6: **end for**
7: **return** $false$

Lazy annotation semi-algorithms [45, 46] build unfoldings of automata trying to discover counter-examples for emptiness. If the automaton $\mathcal{A}$ in question is non-empty, a systematic enumeration of the input event sequences (for instance, using breadth-first search) from Σ^* will suffice to discover a word $w \in \mathcal{L}(\mathcal{A})$, provided that the first order theory of the data domain D is decidable (*Lemma 4.2*). However, if $\mathcal{L}(\mathcal{A}) = \emptyset$, the enumeration of input event sequences may, in principle, run forever. The typical way of fighting this divergence problem is to define a **coverage** relation between the nodes of the unfolding tree.

Definition 4.6 *Given an unfolding U of an automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ a node $\alpha \in DOM(U)$ is covered by another node $\beta \in DOM(U)$, denoted $\alpha \sqsubseteq \beta$ if and only if there exists a node $\alpha' \preceq \alpha$ such that $U(\alpha') \models U(\beta)$. Moreover, U is closed if and only if every leaf from $DOM(U)$ is covered by an uncovered node.*

A lazy annotation semi-algorithm will stop and report emptiness provided that it succeeds in building a closed and safe unfolding of the automaton. Notice that, for any three nodes of an unfolding U, say $\alpha, \beta, \gamma \in DOM(U)$, if $\alpha \prec \beta$ and $\alpha \sqsubseteq \gamma$, then $\beta \sqsubseteq \gamma$ as well. There is no need to expand covered nodes, because, intuitively, there exists a word $w \in \mathcal{L}(\mathcal{A})$ such that $\alpha \preceq w_\Sigma$ and $\alpha \sqsubseteq \gamma$ only if there exists another word $u \in \mathcal{L}(\mathcal{A})$ such that $\gamma \preceq u_\Sigma$. Hence, exploring only those input event sequences that are continuations of γ (and ignoring those of α) suffices in order to find a counter-example for emptiness, if one exists.

An unfolding node $\alpha \in DOM(U)$ is said to be **spurious** if and only if $\Upsilon(\alpha)$ is unsatisfiable. In this case, we change (refine) the labels of (some of the) prefixes of α (and that of α), such that $U(\alpha)$ becomes $\perp$, thus indicating that there is no real execution of the automaton along that input event sequence. As a result of the change of labels, if a node $\gamma \preceq \alpha$ used to cover another node from $DOM(U)$, it might not cover it with the new label. Therefore, the coverage relation has to be recomputed after each refinement of the labelling. The semi-algorithm stops when (and if) a safe complete unfolding has been found.

Theorem 4.2 *If an automaton $\mathcal{A}$ has a non-empty safe closed unfolding then $\mathcal{L}(\mathcal{A}) = \emptyset$.*

The proof is not that complicated. Let U be a safe and complete unfolding of $\mathcal{A}$, such that $DOM(U) \neq \emptyset$. Suppose, by contradiction, that there exists a word $w \in \mathcal{L}(\mathcal{A})$ and let $\alpha \stackrel{def}{=\!=} w_\Sigma$.

Since $w \in \mathcal{L}(\mathcal{A})$, by *Lemma 4.2*, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}, w_{\mathbb{D}} \models \Upsilon(\alpha)$. Assume first that $\alpha \in DOM(U)$. In this case, one can show, by induction on the length $n \geq 0$ of w, that $\Theta(\alpha) \models U(\alpha)^{[n]}$, thus $\mathcal{I}, w_{\mathbb{D}} \models U(\alpha)^{[n]}$. Since $\mathcal{I}, w_{\mathbb{D}} \models \Upsilon(\alpha)$, we have $\mathcal{I}, w_{\mathbb{D}} \models \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q^{[n]}(y) \rightarrow \bot$, hence $U(\alpha)^{[n]} \wedge \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q^{[n]}(y) \rightarrow \bot$. By renaming $q^{[n]}$ with q in the previous formula, we obtain $U(\alpha) \wedge \forall y_1, \forall y_2, ..., \forall y_{\#(q)}.q(y) \rightarrow \bot$ is satisfiable, thus U is not safe, contradiction.

We proceed thus under the assumption that $\alpha \notin DOM(U)$. Since $DOM(U)$ is a non-empty prefix-closed set, there exists a strict prefix α' of α that is a leaf of $DOM(U)$. Since U is closed, the leaf α' must be covered and let $\alpha_1 \preceq \alpha' \preceq \alpha$ be a node such that $U(\alpha_1) \models U(\beta_1)$, for some uncovered node $\beta_1 \in DOM(U)$. Let γ_1 be the unique sequence such that $\alpha_1\gamma_1 = \alpha$. Since $\alpha_1 \sqsubseteq \beta_1$ and $w_\Sigma = \alpha_1\gamma_1 \in \mathcal{L}(\mathcal{A})$, there exists a word w_1 and a cube $c_1 \in C([[U(\alpha_1)]]) \subseteq C([[U(\beta_1)]])$, such that $w_{1\Sigma} = \gamma_1$ and $\mathcal{A}$ accepts w_1 starting with c_1. If $\beta_1\gamma_1 \in DOM(U)$, we obtain a contradiction by a similar argument as above. Hence $\beta_1\gamma_1 \notin DOM(U)$ and there exists a leaf of $DOM(U)$ which is also a prefix of $\beta_1\gamma_1$. Since U is closed, this leaf is covered by an uncovered node $\beta_2 \in DOM(U)$ and let $\alpha_2 \in DOM(U)$ be the minimal (in the prefix partial order) node such that $\beta_1 \preceq \alpha_2 \preceq \beta_1\gamma_1$ and $\alpha_2 \sqsubseteq \beta_2$. Let γ_2 be the unique sequence such that $\alpha_2\gamma_2 = \beta_1\gamma_1$. Since β_1 is uncovered, we have $\beta_1 \neq \alpha_2$ and thus $|\gamma_1| > |\gamma_2|$. By repeating the above reasoning for α_2, β_2 and γ_2, we obtain an infinite sequence $|\gamma_1| > |\gamma_2| > ...$, which is again a contradiction.

As mentioned above, we check emptiness of first order alternating automata using the same method previously used to check emptiness of a simpler model of alternating automata, which uses Boolean constants for control states and whose transition rules have no quantifiers [35]. The higher complexity of the automata model considered here, manifests itself within the interpolant generation procedure, used to refine the labelling of the unfolding. We discuss generation of interpolants in the next section.

4.5 Interpolant Generation of FOADA

4.5.1 Over-Approximation and Interpolants

Typically, when checking the unreachability of a set of program configurations [45], the interpolants used to annotate the unfolded control structure are assertions about the values of the program variables in a given control state, at a certain step of an execution. However, in an alternating model of computation, it is useful to distinguish between (i) locality of interpolants w.r.t. a given control state (control locality) and (ii) locality w.r.t. a given time stamp (time locality). In logical terms, control-local interpolants are defined by formulae involving a single predicate symbol, whereas time-local interpolants involve only predicates $q^{[i]}$ and variables $x^{[i]}$, for a single $i \geq 0$.

When considering an alternating model of computation, control-local interpolants are not always enough to prove emptiness, because of the synchronisation of several branches of the computation on the same sequence of input values.

Example 4.3 *Consider, a FOADA with the following transition rules and final state q_f:*

- $q_0(y) \xrightarrow{a(x)} q_1(y+x) \wedge q_2(y-x)$
- $q_1(y) \xrightarrow{a(x)} y+x > 0 \wedge q_f$
- $q_1(y) \xrightarrow{a(x)} q_1(y+x)$
- $q_2(y) \xrightarrow{a(x)} y-x > 0 \wedge q_f$
- $q_2(y) \xrightarrow{a(x)} q_2(y-x)$

Started in an initial configuration $q_0(0)$ with an input word $(a, v_1), (a, v_2), ..., (a, v_{n-1}), (a, v_n)$, such that $v_i(x) = k_i$, the automaton executes as follows:

$$q_0(0) \xrightarrow{(a,v_1)} \{q_1(k_1), q_2(-k_1)\} \xrightarrow{(a,v_2)} ... \xrightarrow{(a,v_{n-1})} \{q_1(\sum_{i=1}^{n-1} k_i), q_2(-\sum_{i=1}^{n-1} k_i)\} \xrightarrow{(a,v_n)} \emptyset$$

An over-approximation of the set of cubes generated after one or more steps is defined by the formula $\exists x_1 \exists x_2 . q_1(x_1) \wedge q_2(x_2) \wedge x_1 + x_2 \approx 0$. Observe that a control-local formula using one occurrence of a predicate would give a too rough over-approximation of this set, unable to prove the emptiness of the automaton.

In the rest of this section, let us fix an automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$. Due to the above observation, none of the interpolants considered will be control-local and we shall use the term **local** to denote time-local interpolants, with no free variables.

Definition 4.7 *Given a non-empty sequence of input events $\alpha = a_1, a_2, ..., a_n \in \Sigma^*$, a **generalised Lyndon interpolant (GLI)** is a sequence $(I_0, I_1, ..., I_n)$ of formulae such that, for all $k \in [1, n-1]$:*

- $P^-(I_k) = \emptyset$*;*
- $\iota^{[0]} \models I_0$ *and* $I_k \wedge (\bigwedge_{q(y) \xrightarrow{a_i(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[k]}(y) \rightarrow \psi^{[k+1]}) \models I_{k+1}$*;*
- $I_n \wedge \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q(y)$ *is unsatisfiable.*

Moreover, the GLI is local if and only if $V(I_k) \subseteq Q^{[k]}$, for all $k \in [1, n]$.

The following proposition states the existence of local GLI for the theories in which Lyndon's Interpolation Theorem holds. If there exists a Lyndon interpolant for any two formulae ϕ and ψ, such that $\phi \wedge \psi$ is unsatisfiable, then any sequence of input events $\alpha = a_1, a_2, ..., a_n \in \Sigma^*$, such that $\Upsilon(\alpha)$ is unsatisfiable, has a local GLI $(I_0, I_1, ..., I_n)$. Here is the proof.

By definition, $\Upsilon(\alpha)$ is the formula:

$$\iota^{[0]} \wedge \bigwedge_{i=1}^{n} \bigwedge_{q(y) \xrightarrow{a_i(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[i-1]}(y) \rightarrow \psi^{[i]} \wedge \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[n]}(y) \rightarrow \bot$$

We define the formulae:

- $\varphi_i \stackrel{def}{=\!=} \bigwedge_{q(y) \xrightarrow{a_i(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[i-1]}(y) \rightarrow \psi^{[i]}$, for all $i \in [1, n]$
- $\psi \stackrel{def}{=\!=} \bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)} . q^{[n]}(y) \rightarrow \bot$

Observe that $V(\iota^{[0]}) \subseteq Q^{[0]}$, $V(\varphi_i) \subseteq Q^{[i-1]} \cup Q^{[i]} \cup X^{[i]}$, for all $i \in [1, n]$, and $V(\psi) \subseteq Q^{[n]}$. We apply Lyndon's Interpolation Theorem for the formulae $\iota^{[0]}$ and $\bigwedge_{i=1}^{n} \varphi_i \wedge \psi$ and obtain a formula I_0, such that $\iota^{[0]} \models I_0$, $I_0 \wedge \bigwedge_{i=1}^{n} \varphi_i \wedge \psi$ is unsatisfiable, $V(I_0) \subseteq V(\iota^{[0]}) \cap (\bigcup_{i=1}^{n} V(\varphi_i) \cup V(\psi)) \subseteq Q^{[0]}$ and $P^-(I_0) \subseteq P^-(\iota^{[0]}) \cap (\bigcup_{i=1}^{n} P^-(\varphi_i) \cup P^-(\psi)) = \emptyset$. Repeating the reasoning for the formulae $I_0 \wedge \varphi_1$ and $\bigwedge_{i=2}^{n} \varphi_i \wedge \psi$, we obtain I_1, such that $I_0 \wedge \varphi_1 \models I_1$, $I_1 \wedge \bigwedge_{i=2}^{n} \varphi_i \wedge \psi$ is unsatisfiable, $V(I_1) \subseteq (V(I_0) \cup V(\varphi_1)) \cap (\bigcup_{i=2}^{n} V(\varphi_i) \cup V(\psi)) \subseteq Q^{[1]}$ and $P^-(I_1) \subseteq (P^-(I_0) \cup P^-(\varphi_1)) \cap (\bigcup_{i=2}^{n} P^-(\varphi_i) \cup P^-(\psi)) = \emptyset$. Continuing in this way, we obtain formulae $(I_0, I_1, ..., I_n)$ as required.

The main problem with the local GLI construction described in the proof of above is that the existence of Lyndon interpolants is guaranteed in principle, but the proof is non-constructive. Building an interpolant for an unsatisfiable conjunction of formulae $\phi \wedge \psi$ is typically the job of the decision procedure that proves the unsatisfiability and, in general, there is no such procedure, when ϕ and ψ contain predicates and have non-trivial quantifier alternation. In this case, some provers use instantiation heuristics for the universal quantifiers that are sufficient for proving unsatisfiability, however these heuristics are not always suitable for interpolant generation. Consequently, from now on, we assume the existence of an effective Lyndon interpolation procedure only for decidable theories, such as the quantifier-free linear (integer) arithmetic with uninterpreted functions (UFLIA, UFLRA, etc.) [56].

This is where the predicate-free path formulae come into play. For a given event sequence α, the automaton $\mathcal{A}$ accepts a word w such that $w_\Sigma = \alpha$ if and only if $\widetilde{\Upsilon}(\alpha)$ is satisfiable. Assuming further that the equality atoms in the transition rules of $\mathcal{A}$ are written in the language of a decidable first order theory, such as Presburger arithmetic, *Lemma 4.5* gives us an effective way of checking emptiness of $\mathcal{A}$, relative to a given event sequence. However, this method does not cope well with lazy annotation, because there is no way to extract, from the unsatisfiability proof of $\widetilde{\Upsilon}(\alpha)$, the interpolants needed to annotate α. This is because (i) the formula $\widetilde{\Upsilon}(\alpha)$, obtained by repeated substitutions loses track of the steps of the execution, and (ii) quantifiers that occur nested in $\widetilde{\Upsilon}(\alpha)$ make it difficult to write $\widetilde{\Upsilon}(\alpha)$ as an unsatisfiable conjunction of formulae from which interpolants are extracted.

The solution we adopt for the first issue (i) consists in partially recovering the time-stamped structure of the acceptance formula $\Upsilon(\alpha)$ using the formula $\widehat{\Upsilon}(\alpha)$ in which only transition quantifiers occur. The second issue (ii) is solved under the additional assumption that the theory of the data domain D has witness-producing quantifier elimination. More precisely, we

assume that, for each formula $\exists x.\phi(x)$, there exists an effectively computable term τ, in which x does not occur, such that $\exists x.\phi(x)$ and $\phi[\tau/x]$ are equisatisfiable. These terms, called **witness terms** in the following, are actual definitions of the Skolem function symbols from the following folklore theorem.

Theorem 4.3 [11] *Given $Q_1x_1Q_2x_2...Q_nx_n.\phi$ a first-order sentence, where $Q_1, Q_2, ..., Q_n \in \{\exists, \forall\}$ and ϕ is quantifier-free, let $\eta_i \stackrel{def}{=\!=} f_i(y_1, y_2, ..., y_{k_i})$ if $Q_i = \forall$ and $\eta_i \stackrel{def}{=\!=} x_i$ if $Q_i = \exists$, where f_i is a fresh function symbol and $\{y_1, y_2, ..., y_{k_i}\} = \{x_j \mid j < i, Q_j = \exists\}$. Then the entailment $Q_1x_1Q_2x_2...Q_nx_n.\phi \models \phi[\eta_1/x_1, \eta_2/x_2, ..., \eta_n/x_n]$ holds.*

See *Theorem 2.1.8* and *Lemma 2.1.9* in [11] for the proof of *Theorem 4.3*.

Examples of witness-producing quantifier elimination procedures can be found in the literature for e.g. linear integer (real) arithmetic (LIA,LRA), Presburger arithmetic and Boolean algebra of sets and Presburger cardinality constraints (BAPA) [41].

Under the assumption that witness terms can be effectively built, let us describe the generation of a non-local GLI for a given input event sequence $\alpha = a_1, a_2, ..., a_n$. First, we generate successively the acceptance formula $\Upsilon(\alpha)$ and its equisatisfiable forms $\widehat{\Upsilon}(\alpha) = Q_1x_1Q_2x_2...Q_mx_m.\widehat{\Phi}$ and $\widehat{\Upsilon}(\alpha) = Q_1x_1Q_2x_2...Q_mx_m.\widetilde{\Phi}$, both written in prenex form, with matrices $\widehat{\Phi}$ and $\widetilde{\Phi}$, respectively. Because we assumed that the first order theory of D has quantifier elimination, the satisfiability problem for $\Upsilon(\alpha)$ is decidable. If $\Upsilon(\alpha)$ is satisfiable, we build a counter-example for emptiness w such that $w_\Sigma = \alpha$ and w_D is a satisfying assignment for $\Upsilon(\alpha)$. Otherwise, $\Upsilon(\alpha)$ is unsatisfiable and there exist witness terms $\tau_{i_1}, \tau_{i_2}, ..., \tau_{i_l}$, where $\{i_1, i_2, ..., i_l\} = \{j \in [1, m] \mid Q_j = \forall\}$, such that $\widetilde{\Phi}[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]$ is unsatisfiable. Then it turns out that the formula $\widehat{\Phi}[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]$, obtained analogously from the matrix of $\widehat{\Upsilon}(\alpha)$, is unsatisfiable as well. Because this latter formula is structured as a conjunction of formulae $\iota^{[0]} \wedge \phi_1 \wedge \phi_2 \wedge ... \wedge \phi_n \wedge \psi$, where $V(\phi_k) \cap Q^{[\leq n]} \subseteq Q^{[k-1]} \cup Q^{[k]}$ and $V(\psi) \cap Q^{[\leq n]} \subseteq Q^{[n]}$, it is now possible to use an existing interpolation procedure for the quantifier-free theory of D, extended with uninterpreted function symbols, to compute a sequence of non-local GLI $(I_0, I_1, ..., I_n)$ such that $V(I_k) \cap Q^{[\leq n]} \subseteq Q^{[k]}$ for all $k \in [1, n]$.

Example 4.4 *The formula $\widetilde{\Upsilon}(\alpha)$ in Example 4.2 is unsatisfiable and let $\tau_2 = z_1$ be the witness term for the universally quantified variable z_2. Replacing z_2 with τ_2 in the matrix of $\widehat{\Upsilon}(\alpha)$ in Example 4.1 yields the unsatisfiable conjunction:*

$$z_1 \geq 0 \wedge q^{[0]}(z_1) \wedge q^{[0]}(z_1) \rightarrow x^{[1]} \geq 0 \wedge (z_1 \geq z_1 \rightarrow q^{[1]}(x^{[1]} + z_1)) \wedge \\ q^{[1]}(x^{[1]} + z_1) \rightarrow x^{[1]} + z_1 < 0 \wedge q_f^{[2]}(x^{[2]} + x^{[1]} + z_1)$$

A non-local GLI for the above is:

$$(q^{[0]}(z_1) \wedge z_1 \geq 0, x^{[1]} \geq 0 \wedge q^{[1]}(x^{[1]} + z_1) \wedge z_1 \geq 0, \bot)$$

A function $\xi : \mathbb{N} \rightarrow \mathbb{N}$ is (i) [strictly] **monotonic** if and only if for each $n < m$ we have $\xi(n) \leq \xi(m)$ [$\xi(n) < \xi(m)$] and (ii) **finite-range** if and only if for each $n \in \mathbb{N}$ the set $\{m \mid \xi(m) = n\}$ is

finite. If ξ is finite-range, we denote by $\xi^{-1}_{max}(n) \in \mathbb{N}$ the maximal value m such that $\xi(m) = n$. The lemma below gives the proof of correctness for the construction of non-local GLI.

Lemma 4.6 *Given a non-empty input event sequence $\alpha = a_1, a_2, ..., a_n \in \Sigma^*$, such that $\Upsilon(\alpha)$ is unsatisfiable, let $Q_1x_1Q_2x_2...Q_mx_m.\widehat{\Phi}$ be a prenex form of $\widehat{\Upsilon}(\alpha)$ and let $\xi : [1, m] \to [1, n]$ be a monotonic function mapping each transition quantifier to the minimal index from the sequence $\widehat{\Theta}(\alpha_0), \widehat{\Theta}(\alpha_1), ..., \widehat{\Theta}(\alpha_n)$ where it occurs. Then one can effectively build:*

- *witness terms $\tau_{i_1}, \tau_{i_2}, ..., \tau_{i_l}$, where $\{i_1, i_2, ..., i_l\} = \{j \in [1, m] \mid Q_j = \forall\}$ and $V(\tau_{i_j}) \subseteq X^{[\leq \xi(i_j)]} \cup \{x_k \mid k < i_j, Q_k = \exists\}, \forall j \in [1, l]$ such that $\widehat{\Phi}[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]$ is unsatisfiable;*
- *a GLI $(I_0, I_1, ..., I_n)$ for α, such that $V(I_k) \subseteq Q^{[k]} \cup X^{[\leq k]} \cup \{x_j \mid j < \xi^{-1}(k), Q_j = \exists\}$, for all $k \in [1, n]$.*

Here is the proof of *Lemma 4.6*:

(1) If $\Upsilon(\alpha)$ is unsatisfiable, by *Lemma 4.3* and *Lemma 4.4*, we obtain that, successively $\widehat{\Upsilon}(\alpha)$ and $\Upsilon(\alpha)$ are unsatisfiable. Let $Q_1x_1Q_2x_2...Q_mx_m.\widehat{\Phi}$ and $Q_1x_1Q_2x_2...Q_mx_m.\widetilde{\Phi}$ be prenex forms for $\widehat{\Upsilon}(\alpha)$ and $\Upsilon(\alpha)$, respectively. Since we assumed that the first order theory of the data domain has witness-producing quantifier elimination, one can effectively build witness terms $\tau_{i_1}, \tau_{i_2}, ..., \tau_{i_l}$, where $\{i_1, i_2, ..., i_l\} = \{i \in [1, m] \mid Q_i = \forall\}$ and:

- $V(\tau_{i_j}) \subseteq X^{[\leq \xi(i_j)]} \cup \{x_k \mid k < i_j, Q_k = \exists\}$, for all $j \in [1, l]$;
- $\widetilde{\Phi}[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]$ is unsatisfiable.

Let $\widehat{\Phi}_0, \widehat{\Phi}_1, ..., \widehat{\Phi}_n$ be the sequence of quantifier-free formulae, defined as follows:

- $\widehat{\Phi}_0$ is the matrix of some prenex form of $\iota^{[0]}$;
- for all $i = 1, 2, ..., n$, let $\widehat{\Phi}_i$ be the matrix of some prenex form of:

$$\widehat{\Phi}_i \overset{def}{=\!=} \widehat{\Phi}_{i-1} \wedge \underbrace{\bigwedge_{cond1, cond2} q^{[i-1]}(t_1, t_2, ..., t_{\#(q)}) \to \psi^{[i]}[t_1/y_1, t_2/y_2, ..., t_{\#(q)}/y_{\#(q)}]}_{\overset{def}{=\!=} \phi_i}$$

where *cond1*: $q^{[i-1]}(t_1, t_2, ..., t_{\#(q)})$ occurs in $\widehat{\Phi}_{i-1}$
and *cond2*: $q(y_1, y_2, ..., y_{\#(q)}) \xrightarrow{a_i(X)} \psi \in \Delta$

It is easy to see that $\widehat{\Phi}$ is the matrix of some prenex form of:

$$\widehat{\Phi}_n \wedge \underbrace{\bigwedge_{q^{[n]}(t_1, t_2, ..., t_{\#(q)}) \text{ occurs in } \widehat{\Phi}_n, q \in Q \setminus F} q^{[n]}(t_1, t_2, ..., t_{\#(q)}) \to \bot}_{\overset{def}{=\!=} \psi}$$

We can obtain a sequence of quantifier-free formulae $\widetilde{\Phi}_0, \widetilde{\Phi}_1, ..., \widetilde{\Phi}_n$ such that $\widetilde{\Phi}_i \equiv \widehat{\Phi}_i$, for all $i \in [1, n]$ and $\widetilde{\Phi}$ is obtained from $\widetilde{\Phi}_n$ by replacing each occurrence of a predicate atom $q(t_1, t_2, ..., t_{\#(q)})$ in $\widetilde{\Phi}_n$ by $\perp$ if $q \in Q \setminus F$ and by $\top$ if $q \in F$. Clearly $\widetilde{\Phi} \equiv \widehat{\Phi}$, thus $\widehat{\Phi}[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}] \equiv \widetilde{\Phi}[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}] \equiv \perp$.

(2) With the notation introduced at (1), we have $\widehat{\Phi} = \widehat{\Phi}_0 \wedge \bigwedge_{i=1}^{n} \phi_i \wedge \psi$. Consider the sequence of witness terms $\tau_{i_1}, \tau_{i_2}, ..., \tau_{i_l}$, whose existence is provided by (1). Because $V(\tau_{i_j}) \subseteq X^{[\leq \xi(i_j)]} \cup \{x_k \mid k < i_j, Q_k = \exists\}$ for all $j \in [1, l]$, and moreover ξ^{-1} is strictly monotonic, we obtain:

- $V(\widehat{\Phi}_0[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]) \subseteq Q^{[0]} \cup X^{[0]} \cup \{x_j \mid j < \xi^{-1}_{max}(0), Q_j = \exists\}$;
- $V(\phi_i[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]) \subseteq Q^{[i-1]} \cup Q^{[i]} \cup X^{[\leq i]} \cup \{x_j \mid j < \xi^{-1}_{max}(i), Q_j = \exists\}$ for all $i \in [1, n]$;
- $V(\psi[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]) \subseteq Q^{[n]} \cup X^{[\leq n]} \cup \{x_j \mid j \in [1, m], Q_j = \exists\}$.

By repeatedly applying Lyndon's Interpolation Theorem, we obtain a sequence of formulae $(I_0, I_1, ..., I_n)$ such that:

- $\widehat{\Phi}_0[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}] \models I_0$ and $V(I_0) \subseteq Q^{[0]} \cup X^{[0]} \cup \{x_j \mid j < \xi^{-1}_{max}(0), Q_j = \exists\}$;
- $I_{k-1} \wedge \phi_i[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}] \models I_k$ and $V(I_k) \subseteq Q^{[k]} \cup X^{[\leq k]} \cup \{x_j \mid j < \xi^{-1}_{max}(k), Q_j = \exists\}$ for all $k \in [1, n]$;
- $I_n \wedge \psi[\tau_{i_1}/x_{i_1}, \tau_{i_2}/x_{i_2}, ..., \tau_{i_l}/x_{i_l}]$ is unsatisfiable.

To show that $(I_0, I_1, ..., I_n)$ is a GLI for $a_1, a_2, ..., a_n$, it is sufficient to notice that:

$$\bigwedge_{q(y) \xrightarrow{a_k(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q^{[k]}(y) \rightarrow \psi^{[k+1]} \models \phi_k$$

for all $k \in [1, n]$. Consequently, we obtain:

- $\iota^{[0]} \models \widehat{\Phi}_0 \models I_0$, by *Theorem 4.3*;
- $I_{k-1} \wedge (\bigwedge_{q(y) \xrightarrow{a_k(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q^{[k-1]}(y) \rightarrow \psi^{[k]}) \models I_{k-1} \wedge \phi_k \models I_k$;
- $I_n \wedge (\bigwedge_{q \in Q \setminus F} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q(y) \rightarrow \perp) \models I_n \wedge \psi \models \perp$.

In conclusion, under two assumptions about the first order theory of the data domain, namely the (i) witness-producing quantifier elimination, and (ii) Lyndon interpolation for the quantifier-free fragment with uninterpreted functions, we developed a rather generic method that produces generalised Lyndon interpolants for infeasible input event sequences. Moreover, each formula I_k in the interpolant refers only to the current predicate symbols $Q^{[I_k]}$, the current and past input variables $X^{[\leq k]}$ and the existentially quantified transition variables introduced at the previous steps $\{x_j \mid j < \xi^{-1}_{max}(k), Q_j = \exists\}$. The remaining question is how to use such non-local interpolants to label the unfolding of an automaton and to compute the coverage between nodes of the unfolding.

4.5.2 Unfolding with Non-local Interpolants

The unfolding U of an automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ is labelled by formulae $U(\alpha) \in FORM^+(Q, \emptyset)$, with no free symbols, other than predicate symbols, such that the labelling is compatible with the transition relation of the automaton. The following lemma describes the refinement of the labelling of an input sequence α of length n by a non-local GLI $(I_0, I_1, ..., I_n)$, such that $V(I_k) \subseteq Q^{[k]} \cup X^{[\leq k]} \cup X_k$ where X_k are the existentially quantified variables from the prenex normal form of $\widehat{\Upsilon}(\alpha_k)$.

Lemma 4.7 *Let U be an unfolding of an automaton $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ such that $\alpha = a_1, a_2, ..., a_n \in DOM(U)$ and $(I_0, I_1, ..., I_n)$ be a GLI for α. The mapping $U' : DOM(U) \rightarrow FORM^+(Q, \emptyset)$ defined as:*

- *$U'(\alpha_k) = U(\alpha_k) \wedge J_k$, for all $k \in [1, n]$, where J_k is the formula obtained from I_k by replacing each time-stamped predicate symbol $q(k)$ by q and existentially quantifying each free variable in I_k;*
- *$U'(\beta) = U(\beta)$ if $\beta \in DOM(U)$ and $\beta \npreceq \alpha$;*

is an unfolding of $\mathcal{A}$.

The proof of *Lemma 4.7* is not complicated. The new set of formulae $U'(\alpha_0), U'(\alpha_1), ..., U'(\alpha_n)$ complies with *Definition 4.5*, because:

- $U'(\alpha_0) \equiv \iota$, since, by point 2 of *Definition 4.7*, we have $\iota^{[0]} \models I_0$, thus $\iota \models J_0$ and $U'(\alpha_0) = U(\alpha_0) \wedge J_0 \equiv \iota \wedge J_0 \equiv \iota$;
- by point 3 of *Definition 4.7*, we have, for all $k \in [1, n-1]$:

$$I_k \wedge \bigwedge_{q(y) \xrightarrow{a_k(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q^{[k]}(y) \rightarrow \psi^{[k+1]} \models I_{k+1}$$

  We write $I_k^{\langle j \rangle}$ for the formula in which each predicate symbol $q^{[k]}$ is replaced by $q^{[j]}$. Then the following entailment holds:

$$I_k^{\langle 0 \rangle} \wedge \bigwedge_{q(y) \xrightarrow{a_k(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q^{[0]}(y) \rightarrow \psi^{[1]} \models I_{k+1}^{\langle 1 \rangle}$$

  Because J_k is obtained by removing the time stamps from the predicate symbols and existentially quantifying all the free variables of I_k, we also obtain, by applying *Fact 4.4* below:

$$J_k^{[0]} \wedge \bigwedge_{q(y) \xrightarrow{a_k(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q^{[0]}(y) \rightarrow \psi^{[1]} \models J_{k+1}^{[1]}$$

  Since U satisfies the labelling condition of *Definition 4.5* and $U'(\alpha_k) = U(\alpha_k) \wedge J_k$, we obtain, as required:

$$U'(\alpha_k)^{[0]} \wedge \bigwedge_{q(y) \xrightarrow{a_k(X)} \psi \in \Delta} \forall y_1, \forall y_2, ..., \forall y_{\#(q)}. q^{[0]}(y) \rightarrow \psi^{[1]} \models U'(\alpha_{k+1})^{[1]}$$

Fact 4.4 *Given formulae $\phi(x, y)$ and $\psi(x)$ such that $\phi(x, y) \models \psi(x)$, we also have $\exists x.\phi(x, y) \models \exists x.\psi(x)$.*

The proof of *Fact 4.4* is quite simple. For each choice of a valuation for the existentially quantified variables on the left-hand side, we chose the same valuation for the variables on the right-hand side.

Observe that, by *Lemma 4.6*, the set of free variables of a GLI formula I_k consists of (i) variables $X^{[\leq k]}$ keeping track of data values seen in the input at some earlier moment in time, and (ii) variables that track past choices made within the transition rules. Basically, it is not important when exactly in the past a certain input has been read or when a choice has been made, as only the value of the variable determines the future behaviour. Intuitively, existential quantification of these variables does the job of ignoring when in the past these values have been seen.

The last ingredient of the lazy annotation semi-algorithm based on unfoldings consist in the implementation of the coverage check, when the unfolding of an automaton is labelled with conjunctions of existentially quantified formulae with predicate symbols, obtained from interpolation. By *Definition 4.6*, checking whether a given node $\alpha \in DOM(U)$ is covered amounts to finding a prefix $\alpha' \preceq \alpha$ and a node $\beta \in DOM(U)$ such that $U(\alpha') \models U(\beta)$, or equivalently, the formula $U(\alpha') \wedge \neg U(\beta)$ is unsatisfiable. However, the latter formula, in prenex form, has quantifier prefix in the language $\exists^*\forall^*$ and, as previously mentioned, the satisfiability problem for such formulae becomes undecidable when the data theory subsumes Presburger arithmetic [32].

Nevertheless, if we require just a yes/no answer (i.e. not an interpolant) recently developed quantifier instantiation heuristics [54] perform rather well in answering a large number of queries in this class. Observe, moreover, that coverage does not need to rely on a complete decision procedure. If the prover fails in answering the above satisfiability query, then the semi-algorithm assumes that the node is not covered and continues exploring its successors. Failure to compute complete coverage may lead to divergence (non-termination) and ultimately, to failure to prove emptiness, but does not affect the soundness of the semi-algorithm (real counter-examples will still be found).

Chapter 5

Applications

The main application of first-order alternating data automata (FOADA) is checking inclusions between various classes of automata extended with variables ranging over infinite domains that recognise languages over infinite alphabets. The most widely known such classes are timed automata [4] and finite-memory automata [36] (also called register automata). In both cases, complementation is not possible inside the class and inclusion is undecidable. Our contribution is providing a systematic semi-algorithm for these decision problems. In addition, the method described in *Section 4.4* can extend generic register automata inclusion checking framework [34], by allowing monitor (right-hand side) automata to have local variables, that are not visible in the language.

Another application is checking safety (mutual exclusion, absence of deadlocks, etc.) and liveness (termination, lack of starvation, etc.) properties of parameterised concurrent programs, consisting of an unbounded number of replicated threads that communicate via a fixed set of global variables (locks, counters, etc.). The verification of parametric programs has been reduced to checking the emptiness of a (possibly infinite) sequence of first order alternating automata, called predicate automata [26, 27], encoding the inclusion of the set of traces of a parametric concurrent program into increasingly general proof spaces, obtained by generalisation of counter-examples. The program and the proof spaces are first order alternating automata over the infinite alphabet of pairs consisting of program statements and thread identifiers.

5.1 Application on Timed Automata

The standard definition of a finite timed word is a sequence of pairs $(a_1, \tau_1), (a_2, \tau_2), ..., (a_n, \tau_n) \in (\Sigma \times \mathbb{R})^*$, where $\mathbb{R}$ is the set of real numbers, such that $0 \leq \tau_i < \tau_{i+1}$, for all $i \in [1, n-1]$. Intuitively, τ_i is the moment in time where the input event a_i occurs. Given a set C of clocks, the set $\Phi(C)$ of clock constraints is defined inductively as the set of formulae $x \leq c$, $x \geq c$, $\neg\delta$, $\delta_1 \wedge \delta_2$, where $x \in C$, $c \in Q$ is a rational constant and $\delta, \delta_1, \delta_2 \in \Phi(X)$.

A timed automaton is a tuple $\mathcal{T} = (\Sigma, S, S_0, F, C, E)$ where:

- Σ is a finite set of input events;
- S is a finite set of states;
- $S_0 \subseteq S$ is the set of initial states;
- $F \subseteq S$ is the set of final states;
- C is a finite set of clocks;
- $E \subseteq S \times \Sigma \times S \times 2^C \times \Phi(C)$ is the set of transitions $(s, a, s', \lambda, \delta)$ from state s to state s' with symbol a, where λ is the set of clocks to be reset and δ is a clock constraint.

A run of $\mathcal{T}$ over a timed word $w = (a_1, \tau_1), (a_2, \tau_2), ..., (a_n, \tau_n)$ is a sequence $(s_0, \gamma_0), (s_1, \gamma_1), ..., (s_n, \gamma_n)$, where $s_i \in S, \gamma_i : C \to \mathbb{R}$ are clocks valuations, for all $i \in [1, n]$ and:

- $s_0 \in S_0$ and $\gamma_0(x) = 0$ for all $x \in C$;
- for all $i \in [1, n]$, there exists a transition $(s_i, a_i, s_{i+1}, \lambda_i, \delta_i) \in E$ such that $\gamma_i + \tau_{i+1} - \tau_i \models \delta_i$, and for all $x \in C$, $\gamma_{i+1}(x) = 0$ if $x \in \lambda_i$ and $\gamma_{i+1}(x) = \gamma_i(x) + \tau_{i+1} - \tau_i$, otherwise.

Here $\tau_0 \stackrel{def}{=\!=} 0$ and $\gamma_i + \tau_{i+1} - \tau_i$ is the valuation mapping each $x \in C$ to $\gamma_i(x) + \tau_{i+1} - \tau_i$. The run is accepting if and only if $s_n \in F$, in which case $\mathcal{T}$ accepts w. As usual, we denote by $\mathcal{L}(\mathcal{T})$ the set of finite words accepted by $\mathcal{T}$. It is well-known that, in general, there is no timed automaton accepting the complement language $(\Sigma \times \mathbb{R})^* \setminus \mathcal{L}(\mathcal{T})$ and, moreover, the language inclusion problem is undecidable [4].

Given a timed automaton $\mathcal{T} = (\Sigma, S, S_0, F, C, E)$, we define a first-order alternating automaton (FOADA) $\mathcal{A}_\mathcal{T} = (\mathbb{R}, \Sigma, \{t\}, Q_\mathcal{T}, \iota_\mathcal{T}, F_\mathcal{T}, \Delta_\mathcal{T})$, with a single input variable t, ranging over $\mathbb{R}$, such that each timed word $w = (a_1, \tau_1), (a_2, \tau_2), ..., (a_n, \tau_n)$ corresponds to a unique data word $d(w) = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n)$ such that $v_i(t) = \tau_i$ for all $i \in [1, n]$ and $\mathcal{L}(\mathcal{A}_\mathcal{T}) = \{d(w) \mid w \in \mathcal{L}(\mathcal{T})\}$. The only difficulty here is capturing the fact that all the clocks of $\mathcal{T}$ evolve at the same pace, which is easily done using a technique from [30], which replaces each clock x_i of $\mathcal{T}$ by a variable y_i tracking the difference between the values of t and x_i.

Formally, if $C = \{x_1, x_2, ..., x_k\}$ and $S = \{s_1, s_2, ..., s_m\}$, we define $Q_\mathcal{T} \stackrel{def}{=\!=} \{q_1, q_2, ..., q_m\}$ where $\#(q_i) = k + 1$ for all $i \in [1, m]$, $\iota_\mathcal{T} \stackrel{def}{=\!=} \bigvee_{s_i \in S_0} q_i(0, 0, ..., 0)$, $F_\mathcal{T} \stackrel{def}{=\!=} \{q_i \mid s_i \in F\}$ and, for each transition $(s_i, a, s_j, \lambda, \delta) \in E$, $\Delta_\mathcal{T}$ contains the rule:

$$q_i(y_1, y_2, ..., y_k, z) \xrightarrow{a(t)} t > z \wedge \delta(z - y_1, z - y_2, ..., z - y_k) \wedge q_j(y'_1, y'_2, ..., y'_k, t)$$

where y'_i stands for z if $x_i \in \lambda$ and for y_i, otherwise. Moreover, nothing else is in $\Delta_\mathcal{T}$. We establish the following connection between a timed automaton and its corresponding first order alternating automaton.

Proposition 5.1 *Given a timed automaton $\mathcal{T} = (\Sigma, S, S_0, F, C, E)$, the first-order alternating data automaton (FOADA) $\mathcal{A}_\mathcal{T} = (\mathbb{R}, \Sigma, \{t\}, Q_\mathcal{T}, \iota_\mathcal{T}, F_\mathcal{T}, \Delta_\mathcal{T})$ recognises the language $\mathcal{L}(\mathcal{A}_\mathcal{T}) = \{d(w) \mid w \in \mathcal{L}(\mathcal{T})\}$.*

Here is the proof of *Proposition 5.1*:

- $\subseteq$ Let $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n) \in \mathcal{L}(\mathcal{A}_\mathcal{T})$ be a data word. We show the existence of a timed word $(a_1, \tau_1), (a_2, \tau_2), ..., (a_n, \tau_n) \in \mathcal{L}(\mathcal{T})$ such that $v_i(t) = \tau_i$, for all $i \in [1, n]$, by induction on $n \geq 0$. In fact we shall prove the following stronger statements:

  (1) each execution of $\mathcal{A}_\mathcal{T}$ over w starting with a cube $c \in C([[\iota_\mathcal{T}]]^\mu)$ is a linear tree, in which each node has at most one child;

  (2) for each execution $q_{i_0}(d_1^0, d_2^0, ..., d_k^0, \tau_0), q_{i_1}(d_1^1, d_2^1, ..., d_k^1, \tau_1), ..., q_{i_n}(d_1^n, d_2^n, ..., d_k^n, \tau_n)$ of $\mathcal{A}_\mathcal{T}$, $\mathcal{T}$ has an execution $(s_{i_0}, \gamma_0), (s_{i_1}, \gamma_1), ..., (s_{i_n}, \gamma_n)$ over the timed word (a_1, τ_1), $(a_2, \tau_2), ..., (a_n, \tau_n)$, such that, for all $i \in [1, n]$ and all $l \in [1, k]$, we have $\gamma_i(x_l) = \tau_{i-1} - d_l^i$.

  The first point above is by inspection of $\iota_\mathcal{T} = \bigvee_{s_i \in S_0} q_i(0, 0, ..., 0)$ and of the rules from $\Delta_\mathcal{T}$. Indeed, each minimal model of $\iota_\mathcal{T}$ corresponds to a cube $q(0, ..., 0)$ and each rule has exactly one predicate atom on its right-hand side, thus each node of the execution will have at most one successor. The second point is by induction on $n \geq 0$.

- $\supseteq$ Let $w = (a_1, \tau_1), (a_2, \tau_2), ..., (a_n, \tau_n) \in \mathcal{L}(\mathcal{T})$ be a time word. By induction on $n \geq 0$, we show that for each run $(s_{i_0}, \gamma_0), (s_{i_1}, \gamma_1), ..., (s_{i_n}, \gamma_n)$ of $\mathcal{T}$ over w, $\mathcal{A}_\mathcal{T}$ has a linear execution $q_{i_0}(d_1^0, d_2^0, ..., d_k^0, \tau_0), q_{i_1}(d_1^1, d_2^1, ..., d_k^1, \tau_1), ..., q_{i_n}(d_1^n, d_2^n, ..., d_k^n, \tau_n)$ such that, for all $i \in [1, n]$ and all $l \in [1, k]$, we have $\gamma_i(x_l) = \tau_{i-1} - d_l^i$.

An easy consequence is that the timed language inclusion problem "given timed automata $\mathcal{T}_1$ and $\mathcal{T}_2$, does $\mathcal{L}(\mathcal{T}_1) \subseteq \mathcal{L}(\mathcal{T}_2)$?" is reduced in polynomial time to the emptiness problem $\mathcal{L}(\mathcal{A}_{\mathcal{T}_1}) \cap \mathcal{L}(\overline{\mathcal{A}_{\mathcal{T}_2}}) = \emptyset$, for which *Section 4.4* provides a semi-algorithm. Observe, moreover, that no transition quantifiers are needed to encode timed automata as first-order alternating data automata (FOADA).

5.2 Application on Register Automata

Finite-memory automata, most commonly referred to as register automata [36] are among the first attempts at lifting the finite alphabet restriction of classical automata. In a nutshell, a register automaton is a finite-state automaton (FSA) equipped with a finite set of registers $x_1, x_2, ..., x_r$ able to copy input values and compare them with subsequent input. Consequently, basic results from classical automata theory, such as the pumping lemma or the closure under complement do not hold in this model and, moreover, inclusion of languages recognised by register automata is undecidable [48].

Let Σ be an infinite alphabet, # be a symbol not in Σ and $r > 0$ be an integer constant, denoting the number of registers. An assignment is a word $V = v_1, v_2, ..., v_r$ such that if $v_i = v_j$ and $i \neq j$ then $v_i = \#$, for all $i, j \in [1, r]$. We write $[V]$ for the set $\{v_i \mid i \in [1, r]\}$ of values in the assignment V. A finite-memory (register) automaton is a tuple $\mathcal{R} = (S, q_0, U, \rho, \mu, F)$ where:

- S is a finite set of states;

- $q_0 \in S$ is the initial state;
- $U = u_1, u_2, ..., u_r$ is the initial assignment;
- $\rho : S \to [1, r]$ is the re-assignment partial function;
- $\mu \subseteq S \times [1, r] \times S$ is the transition relation;
- $F \subseteq S$ is the set of final states.

A run of $\mathcal{R}$ over an input word $a_1, a_2, ..., a_n \in \Sigma^*$ is a sequence $(s_0, V_0), (s_1, V_1), ..., (s_n, V_n)$ such that $V_0 = U$ and, for all $i \in [1, n]$, exactly one of the following holds:

- if there exists $k \in [1, r]$ such that $a_i = (V_{i-1})_k$ then $V_i = V_{i-1}$ and $(s_{i-1}, k, s_i) \in \mu$;
- otherwise $a_i \notin [V_{i-1}], \rho(s_{i-1})$ is defined, $(V_i)_{\rho(s_{i-1})} = a_i$, for each $k \in [1, r] \setminus \{\rho(s_{i-1})\}$, we have $(V_i)_k = (V_{i-1})_k$ and $(s_{i-1}, \rho(s_{i-1}), s_i) \in \mu$.

Intuitively, if the input symbol is already stored in some register, the automaton moves to the next state if, moreover, the transition relation allows it, otherwise it copies the input to the register indicated by the re-assignment, erasing the previous value, and moves according to the transition relation.

The translation of register automata to first order alternating automata is quite natural, because registers can be encoded as arguments of predicate atoms. Formally, given a register automaton $\mathcal{R} = (S, s_0, U, \rho, \mu, F)$ over a data domain D, such that $S = \{s_0, s_1, ..., s_m\}$, we define the first-order alternating data automaton (FOADA) $\mathcal{A}_{\mathcal{R}} = (D, \{\alpha\}, \{x\}, Q_{\mathcal{R}}, \iota_{\mathcal{R}}, F_{\mathcal{R}}, \Delta_{\mathcal{R}})$ where:

- $\alpha \notin \Sigma$;
- $Q_R \stackrel{def}{=\!=} \{q_0, q_1, ..., q_m\}$;
- $\#(q_i) = r$ for all $i \in [1, m]$;
- $\iota_{\mathcal{R}} \stackrel{def}{=\!=} q_0(U)$;
- $F_R \stackrel{def}{=\!=} \{q_i \mid s_i \in F\}$;
- for each transition $(s_i, k, s_j) \in \mu$, $\Delta_{\mathcal{R}}$ contains the rule:

$$q_i(y_1, y_2, ..., y_t) \xrightarrow{\alpha(x)} y_k = x \wedge q_j(y_1, y_2, ..., y_r) \vee \bigwedge_{i=1}^{r} x \neq y_i \wedge q_j(y_1, y_2, ..., y_{k-1}, x, y_{k+1}, y_{k+2}, ..., y_r)$$

Moreover, nothing else is in $\Delta_{\mathcal{R}}$. The connection between register automata and first order alternating data automata (FOADA) is stated below.

Proposition 5.2 *Given a register automaton $\mathcal{R} = (S, s_0, U, \rho, \mu, F)$ over a data domain D, the first-order alternating data automaton (FOADA) $\mathcal{A}_\mathcal{R} = (D, \{\alpha\}, \{x\}, Q_\mathcal{R}, \iota_\mathcal{R}, F_\mathcal{R}, \Delta_\mathcal{R})$ recognises the language:*

$$\mathcal{L}(\mathcal{A}_\mathcal{R}) = \{(\alpha, a_1), (\alpha, a_2), ..., (\alpha, a_n) \mid a_1, a_2, ..., a_n \in \mathcal{L}(\mathcal{R})\}$$

Here is the proof of *Proposition 5.2*:

- $\subseteq$ Let $w = (\alpha, a_1), (\alpha, a_2), ..., (\alpha, a_n) \in \mathcal{L}(\mathcal{A}_\mathcal{R})$. First, it is easy to show that each execution of $\mathcal{A}_\mathcal{R}$, that starts in some cube $c \in C([[\iota_\mathcal{R}]]^\mu)$, is a linear tree with labels $q_0(V_0), q_1(V_0), ..., q_n(V_0)$ such that $V_0 = U$. Second by induction on $n \geq 0$, we prove that $\mathcal{A}_\mathcal{R}$ has a run as above over w only if $\mathcal{R}$ has a run $(q_0, V_0), (q_1, V_1), ..., (q_n, V_n)$ over $a_1, a_2, ..., a_n$.
- $\supseteq$ Let $w = a_1, a_2, ..., a_n \in \mathcal{L}(\mathcal{R})$ and $q_0(V_0), q_1(V_0), ..., q_n(V_0)$ be a run of $\mathcal{R}$ over w, such that $V_0 = U$. By induction on $n \geq 0$, we can build an execution of $\mathcal{A}_\mathcal{R}$ over $(\alpha, a_1), (\alpha, a_2), ..., (\alpha, a_n)$ that is a linear tree with labels $q_0(V_0), q_1(V_1), ..., q_n(V_n)$.

Consequently, the language inclusion problem "given register automata $\mathcal{R}_1$ and $\mathcal{R}_2$, does $\mathcal{L}(\mathcal{R}_1) \subseteq \mathcal{L}(\mathcal{R}_2)$?" is reduced in polynomial time to emptiness problem $\mathcal{L}(\mathcal{A}_{\mathcal{R}_1}) \cap \mathcal{L}(\overline{\mathcal{A}_{\mathcal{R}_2}}) = \emptyset$, for which *Section 4.4* provides a semi-algorithm. Notice further that the encoding of register automata as first-order alternating data automata (FAODA) uses no transition quantifiers.

5.3 Application on Predicate Automata

The model of predicate automata [26, 27] has emerged recently as a tool for checking safety and liveness properties of parameterised concurrent programs, in which there is an unbounded number of replicated threads that communicate via global variables. Predicate automata recognise finite sequences of actions that are pairs (σ, i) where σ is from a finite set Σ of program statements and $i \in N$ ranges over an unbounded set of thread identifiers. To avoid clutter, we shall view a pair (σ, i) as a data symbol (σ, v) where $v(x) = i$, for a designated input variable x.

Since thread identifiers can only be compared for equality, the data theory of predicate automata is the first order theory of equality. Moreover, transition quantifiers are only needed for checking termination and, generally, liveness properties [27].

However, the execution semantics of predicate automata differs from that of first order automata with respect to the following detail: initial configurations and successors of predicate automata are defined using the entire sets of models of the initial sentence and transition rules, not just the minimal ones, as in our case.

Formally, a run of a predicate automaton $\mathcal{P} = (\Sigma, \{x\}, Q, \iota, F, \Delta)$ over a word $(a_1, v_1), (a_2, v_2)$, ..., (a_n, v_n) is a sequence of interpretations $I_0, I_1, ..., I_n$ such that $I_0 \in [[\iota]]$ and for each $i \in [1, n]$, each $q \in Q$ and each tuple $(d_1, d_2, ..., d_{\#(q)}) \in I_{i-1}(q)$, we have $I_i \in [[\psi]]_v$, for each rule

$q(y_1, y_2, ..., y_{\#(q)}) \xrightarrow{a_i(x)} \psi \in \Delta$, where $v = v_i[y_1 \leftarrow d_1, y_2 \leftarrow d_2, ..., y_{\#(q)} \leftarrow d_{\#(q)}]$. The run is accepting if and only if $I(q) \neq \emptyset$ for all $q \in Q \setminus F$.

In fact, as shown next, this more simple execution semantics is equivalent, from the language point of view, with the semantics given by *Definition 4.1* and *Definition 4.2*. We believe that the semantics of first-order alternating data automata based on minimal models is important for its relation to the textbook semantics of Boolean alternating automata [12].

Proposition 5.3 *Given a predicate automaton $\mathcal{P} = (\Sigma, \{x\}, Q, \iota, F, \Delta)$, let $\mathcal{A}_{\mathcal{P}}$ be the first-order alternating automaton that has the same description as $\mathcal{P}$. Then $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{A}_{\mathcal{P}})$.*

Here is the proof of *Proposition 5.3*:

- $\subseteq$ Let $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n) \in \mathcal{L}(\mathcal{P})$ be a word and $I_0, I_1, ..., I_n$ be an accepting execution of $\mathcal{P}$ over w. Let $I_j^{[i]}$ be the interpretation that associates each predicate $q^{[i]}$ the set $I_j(q)$, for $i, j \in [1, n]$. Then one builds, by induction on $n \geq 0$, an execution $\mathcal{T}$ of $\mathcal{A}_{\mathcal{P}}$ such that $I_{\mathcal{T}} \subseteq \bigcup_{i=0}^{n} I_i^{[i]}$, where $I_{\mathcal{T}}$ is the unique interpretation associated with $\mathcal{T}$. Since $I_0, I_1, ..., I_n$ is accepting, we have $I_n^{[n]}(q^{[n])} = \emptyset$, for all $q \in Q \setminus F$ and hence $I_{\mathcal{T}}(q^{[n]}) = \emptyset$, for all $q \in Q \setminus F$ and, consequently $w \in \mathcal{L}(\mathcal{A}_{\mathcal{P}})$.
- $\supseteq$ Let $w = (a_1, v_1), (a_2, v_2), ..., (a_n, v_n) \in \mathcal{L}(\mathcal{A}_{\mathcal{P}})$ be a word and $\mathcal{T}$ be an accepting execution of $\mathcal{A}_{\mathcal{P}}$ over w. We define the sequence of interpretations $I_0, I_1, ..., I_n$ as $I_i(q) = I_{\mathcal{T}}(q^{[i]})$, for each $i \in [1, n]$ and each $q \in Q$. By induction on $n \geq 0$ one shows that $I_0, I_1, ..., I_n$ is an execution of $\mathcal{P}$. Moreover, since $\mathcal{T}$ is accepting, we have $I_n(q) = I_{\mathcal{T}}(q^{[n]}) = \emptyset$, for each $q \in Q \setminus F$, thus $w \in \mathcal{L}(\mathcal{P})$.

As before, this result enables using the semi-algorithm from *Section 4.4* for checking emptiness of predicate automata. We point out that, although quantifier-free predicate automata with predicates of arity one are decidable for emptiness [26], currently there is no method for checking emptiness of predicate automata with predicates of arity greater than one, other than the explicit enumeration of cubes. Moreover, no method for dealing with emptiness in the presence of transition quantifiers is known to exist.

Chapter 6

FOADA Checker

Besides the theoretical parts, we also have developed a tool - *FOADA Checker* [62], mainly used for checking inclusion between two automata or checking emptiness of an automaton. The tool is written in Java, via Java-SMT interface [57] and using Z3 SMT solver [53] for spuriousness, coverage queries and interpolant generation. The IMPACT semi-algorithm has been implemented in the tool to check the emptiness of an automaton. The supported input automata (can be parsed by our own parser written in ANTLR4 [50]) are: (i) predicate automata [26, 27], (ii) alternating data automata and (iii) first-order alternating data automata.

In the first section, we show how to install the tool and use it to check inclusion of two automata or emptiness of an automaton. And then, we show the input format of first-order alternating data automata, which is the default data structure of the tool. After that, we introduce the input formats of alternating data automata and predicate automata, and explain how to transform them into the default data structure of the tool, hence first-order alternating data automata. In the end, we show some experimental results.

6.1 Brief User Guide

6.1.1 Installation

FOADA Checker can be downloaded via [62] and it only supports two operating systems: (i) Mac-OS and (ii) Linux. Once it has been downloaded, the installation can be simply done by typing following command in terminal under the downloaded folder:

```
sudo make install
```

After the installation, we can verify whether all the solvers (SMT Interpol [49], Z3 SMT Solver [53], MathSAT 5 [37] and Princess [55]) are successfully integrated with JavaSMT, by simply typing following command:

```
foada -c
```

```
Xiaos-MacBook:FOADA cathiec$ sudo make install
Architecture : Mac OS X
Installing FOADA...
-----> Building the main executable program...
[ 15%] The main executable program has been built.
-----> Adding Microsoft Z3 dynamic library to the default dynamic library folder...
[ 30%] Microsoft Z3 dynamic library has been added to the default dynamic library folder.
-----> Creating FOADA library folder...
[ 40%] FOADA library folder has been created.
-----> Adding Microsoft Z3 dynamic library to FOADA library folder...
[ 55%] Microsoft Z3 dynamic library has been added to FOADA library folder.
-----> Adding Microsoft Z3 Java dynamic library to FOADA library folder...
[ 70%] Microsoft Z3 Java dynamic library has been added to FOADA library folder.
-----> Adding MathSAT5 Java dynamic library to FOADA library folder...
[ 85%] MathSAT5 Java dynamic library has been added to FOADA library folder.
-----> Adding the runnable jar file to FOADA library folder...
[ 100%] The runnable jar file has been added to FOADA library folder.
FOADA has been successfully installed.
Xiaos-MacBook:FOADA cathiec$ foada -c
FOADA > Start checking all the solvers...
JavaSMT > The solver SMTINTERPOL succeeded.
JavaSMT > The solver Z3 succeeded.
JavaSMT > The solver MATHSAT5 succeeded.
JavaSMT > The solver PRINCESS succeeded.
FOADA > End of session.
```

Figure 6.1: Screenshot of a Successful Installation on Mac-OS

6.1.2 Emptiness Checking

FOADA Checker is able to check whether the language of a given automaton is empty, by simply typing following command:

```
foada -e example.foada
```

We have implemented a version of IMPACT semi-algorithm [45] in the tool for emptiness checking. We have two cases:

- if the given automaton is empty, then the termination is not guaranteed; but if the tool terminates by reporting "empty" (*Figure 6.2*), then the given automaton is truly empty, hence the correctness of the result is guaranteed;
- if the given automaton is not empty, then the termination is guaranteed algorithmically[1], and the tool reports "not empty" together with a counter-example (*Figure 6.3*), which is a word from the language of the given automaton.

```
Xiaos-MacBook:examples cathiec$ foada -e running2.foada
FOADA > Type of the input file is < *.foada >.
ANTLR4 > Parsing and checking the syntax in the input...
ANTLR4 > Syntax checking succeeded...
FOADA > Start checking emptiness (BFS / Universally Quantify Arguments) ...
------------------------------
FOADA > Nodes Created : 13
FOADA > Nodes Visited : 13
FOADA > Time Used : 698 ms
FOADA > The automaton is empty...
FOADA > End of session.
```

Figure 6.2: Screenshot of an Empty Automaton

[1] Any error of the solver might break the program. But besides that, the termination is guaranteed.

```
Xiaos-MacBook:examples cathiec$ foada -e Philo2.foada
FOADA > Type of the input file is < *.foada >.
ANTLR4 > Parsing and checking the syntax in the input...
ANTLR4 > Syntax checking succeeded...
FOADA > Start checking emptiness (BFS / Universally Quantify Arguments) ...
-----------------------------
a       ::::: DATA ::::: { any any any any any -4357 4642 4260 6054 5464 }
a       ::::: DATA ::::: { any any any any any 2073 2663 8140 15852 8089 }
a       ::::: DATA ::::: { any any any any any 7723 11 9754 755 8140 }
-----------------------------
FOADA > Nodes Created : 4
FOADA > Nodes Visited : 4
FOADA > Time Used : 171 ms
FOADA > The automaton is not empty...
FOADA > End of session.
```

Figure 6.3: Screenshot of a Non-Empty Automaton

The counter-example reported by the tool consists of different lines, where each line contains:

- an event symbol (for example, "a" in *Figure 6.3*)
- a valuation of variables (for example, "*any any any any any* -4357 4642 4260 6054 5464" in *Figure 6.3*)

6.1.3 Inclusion Checking

FOADA Checker can also check the language inclusion between two given automata, by simply typing following command:

```
foada -i example1.foada example2.foada
```

The inclusion checking implemented in the tool is also based on IMPACT semi-algorithm [45]. We have two cases:

- if the inclusion holds, then the termination is not guaranteed; but if the tool terminates by reporting "inclusion holds", then the inclusion truly holds, hence the correctness of the result is guaranteed;
- if the inclusion does not hold, then the termination is guaranteed algorithmically[2], and the tool reports "inclusion does not hold" together with a counter-example, which is a word from the language of the first given automaton that is not accepted by the second given automaton.

We have implemented Boolean operations of intersection and complement, so $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ is transformed into $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}}) = \emptyset$, which is an emptiness problem. The inclusion checking of FOADA Checker calls the emptiness checking function. Hence, the counter-example of inclusion checking reported by the tool is in the same format as the one for emptiness checking, consisting of different lines, where each line contains an event symbol and a valuation of variables.

[2]Any error of the solver might break the program. But besides that, the termination is guaranteed.

6.2 Input Format - First-Order Alternating Data Automata (FOADA)

A FOADA input file describing a FOADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ contains:

- declaration of states (Q): (pred (q_0 q_1 ... $q_{|Q|}$))
- declaration of event symbols (Σ): (event (a_1 a_2 ... $a_{|\Sigma|}$))
- declaration of initial configuration (ι): (initial ι)
- declaration of final states (F): (final (f_1 f_2 ... f_k))
- declaration of transition rules (Δ) in the format:

  (trans
      (q_i ((d_1 $Sort_{d_1}$) (d_2 $Sort_{d_2}$) ... ($d_{\#(q_i)}$ $Sort_{d_{\#(q_i)}}$)))
      (a_j ((x_1 $Sort_{x_1}$) (x_2 $Sort_{x_2}$) ... ($x_{|X|}$ $Sort_{x_{|X|}}$)))
      (ψ)
)

  where:

  - $q_i \in Q$;
  - $d_1, d_2, ..., d_{\#(q_i)}$ are the arguments of q_i;
  - $Sort_m$ is the sort of m;
  - $a_j \in \Sigma$;
  - $x_1, x_2, ..., x_{|X|} \in X$;
  - $\psi \in FORM^+(Q, X \cup \{d_1, d_2, ..., d_{\#(q_i)}\})$ is a positive formula in SMT2 format [1], where $X \cap \{d_1, d_2, ..., d_{\#(q_i)}\} = \emptyset$.

Example 6.1 *The source code below describes a FOADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ where:*

- $D = \mathbb{Z}$, $\Sigma = \{a\}$, $X = \{x\}$, $Q = \{p, q\}$, $\iota = p(0)$, $F = \{q\}$,
- *and there are two transitions:*

$$p(d) \xrightarrow{a(x)} q(x) \wedge x \geq 0$$

$$q(d) \xrightarrow{a(x)} q(x) \wedge d \geq 0$$

```
1 (pred (p q))
2 (event (a))
3 (initial (p 0))
4 (final (q))
5
6 (trans (p ((d Int))) (a ((x Int))) (and (q x) (>= x 0)))
7 (trans (q ((d Int))) (a ((x Int))) (and (q x) (>= d 0)))
```

6.3 Input Format - Alternating Data Automata (ADA)

An ADA input file describing an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ contains:

- declaration of states (Q):

  STATES

  $q_0\ q_1\ \dots\ q_{|Q|}$

- declaration of initial configuration (ι):

  INITIAL

  ι

- declaration of final states (F):

  FINAL

  $f_1\ f_2\ \dots\ f_k$

- declaration of event symbols (Σ):

  SYMBOLS

  $a_1\ a_2\ \dots\ a_{|\Sigma|}$

- declaration of variables (X):

  VARIABLES

  $x_1\ x_2\ \dots\ x_{|X|}$

- declaration of transition rules (Δ):

  TRANSITIONS

  $a_{i_1}\ q_{j_1}$

  ψ_{k_1}

  #

  $a_{i_2}\ q_{j_2}$

  ψ_{k_2}

  #

  ...

  #

  $a_{i_{|\Delta|}}\ q_{j_{|\Delta|}}$

  $\psi_{k_{|\Delta|}}$

  #

  where:

– $a_{i_1}, a_{i_2}, ..., a_{i_{|\Delta|}} \in \Sigma$;
– $q_{j_1}, q_{j_2}, ..., q_{j_{|\Delta|}} \in Q$;
– $\psi_{k_1}, \psi_{k_2}, ..., \psi_{k_{|\Delta|}} \in FORM^+(Q, \overline{X} \cup X)$ are formulae in SMT2 format [1], where $\overline{X}$ denotes $\{\overline{x} \mid x \in X\}$.

Example 6.2 *The source code below describes an ADA $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ where:*

- $D = \mathbb{Z}$, $\Sigma = \{a\}$, $X = \{x, y\}$, $Q = \{q_0, q_1, q_2\}$, $\iota = q_0 \wedge q_2$, $F = \{q_1\}$,
- *and there are three transitions:*

$$\Delta(q_0, a) \equiv q_1 \wedge x_1 = 0 \wedge y_1 = 0$$
$$\Delta(q_1, a) \equiv q_1 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1$$
$$\Delta(q_2, a) \equiv q_2 \vee \neg(x_1 = y_1)$$

```
1  STATES
2  q0 q1 q2
3
4  INITIAL
5  (and q0 q2)
6
7  FINAL
8  q1
9
10 SYMBOLS
11 a
12
13 VARIABLES
14 x y
15
16 TRANSITIONS
17 a q0
18 (and q1 (= 0 x1) (= y1 0))
19 #
20 a q1
21 (and q1 (= x1 (+ x0 1)) (= y1 (+ y0 1)))
22 #
23 a q2
24 (or q2 (not (= x1 y1)))
25 #
```

Once an ADA has been read by FOADA Checker as the input, it is stored as a FOADA that is equivalent to the original ADA, hence recognising the same language.

Example 6.3 *The ADA $\mathcal{A}$ in Example 6.2 is transformed into a FOADA $\mathcal{A}' = (D', \Sigma', X', Q', \iota', F', \Delta')$ that is equivalent to $\mathcal{A}$ once it has been read as the input, where:*

- $D' = D = \mathbb{Z}$, $\Sigma = \Sigma' = \{a\}$, $X' = X = \{x, y\}$, $Q' = Q = \{q_0, q_1, q_2\}$, $F' = F = \{q_1\}$,
- $\iota' = q_0(0, 0) \wedge q_2(0, 0)$,

- *and there are three transitions in* Δ'*:*
  - $q_0(x_0, y_0) \xrightarrow{a(x_1,y_1)} q_1(x_1, y_1) \wedge x_1 = 0 \wedge y_1 = 0$
  - $q_1(x_0, y_0) \xrightarrow{a(x_1,y_1)} q_1(x_1, y_1) \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1$
  - $q_2(x_0, y_0) \xrightarrow{a(x_1,y_1)} q_2(x_1, y_1) \vee \neg(x_1 = y_1)$

6.4 Input Format - Predicate Automata (PA)

FOADA Checker supports predicate automata [26, 27] as inputs. Once a PA has been read by the tool as the input, it is stored as a FOADA that is equivalent to the original PA, hence recognising the same language.

Example 6.4 *The source code below is a predicate automaton from an example set* [40]*. It is an example for the tool "Duet"* [38]*, which is a static analysis tool designed for analysing concurrent programs.*

```
1 start: {a}() /\ {b}().
2 final: none.
3
4 {a}() --( a1 : i )-> {c}(i).
5 {b}() --( a1 : i )-> {d}(i).
6 {c}(i) --( a2 : j )-> {e}(i).
7 {d}(i) --( a2 : j )-> {e}(j).
8 {e}(i) --( a3 : j )-> true.
```

Example 6.5 *The PA in the source code above is stored as a FOADA* $\mathcal{A} = (D, \Sigma, X, Q, \iota, F, \Delta)$ *accepting the same language once it has been read by FOADA Checker as the input, where:*

- $D = \mathbb{Z}$, $\Sigma = \{a1, a2, a3\}$, $X = \{x\}$, $Q = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}\}$, $\iota = \{a\}() \wedge \{b\}()$, $F = \emptyset$,
- *and there are five transitions in* Δ*:*
  - $\{a\}() \xrightarrow{a1(x)} \{c\}(x)$
  - $\{b\}() \xrightarrow{a1(x)} \{d\}(x)$
  - $\{c\}(i) \xrightarrow{a2(x)} \{e\}(i)$
  - $\{d\}(i) \xrightarrow{a2(x)} \{e\}(x)$
  - $\{e\}(i) \xrightarrow{a3(x)} true$

6.5 Experimental Results

We have done experiments with several sources:

- **predicate automata models** [26, 27, 40]:
  - incdec.pa
  - localdec.pa
  - ticket.pa
  - count_thread0.pa
  - count_thread1.pa
  - local0.pa
  - local1.pa
- **timed automata inclusion problems**:
  - abp.ada
  - train.ada
  - rr-crossing.foada
- **array logic entailments**:
  - array_rotation.ada
  - array_simple.ada
  - array_shift.ada
- **hardware circuit verification** [34]:
  - hw1.ada
  - hw2.ada
- **parametric verification problems** checking inclusions of the form $\bigcap_{i=1}^{N} \mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{B})$:
  - train-simple1.foada
  - train-simple2.foada
  - train-simple3.foada
  - fischer-mutex2.foada
  - fischer-mutex3.foada

The experiments were carried out on a Mac-OS x64 - 1.3 GHz Intel Core i5 - 8 GB 1867 MHz LPDDR3 machine, and the experimental results are reported in *Table 6.1*.

Example	$\lvert\mathcal{A}\rvert$ (bytes)	$\mathcal{L}(\mathcal{A}) = \emptyset$?	Nodes Expanded	Nodes Visited	Time (ms)
incdec.pa	499	no	21	17	779
localdec.pa	678	no	49	35	1814
ticket.pa	4250	no	229	91	9543
count_thread0.pa	9767	no	154	128	8553
count_thread1.pa	10925	no	766	692	76771
local0.pa	10595	no	73	27	1431
local1.pa	11385	no	1135	858	101042
array_rotation.ada	1834	yes	9	8	1543
array_simple.ada	3440	yes	11	10	6787
array_shift.ada	874	yes	6	5	413
abp.ada	6909	no	52	47	4788
train.ada	1823	yes	68	67	7319
hw1.ada	322	Solver Error	/	/	/
hw2.ada	674	yes	20	22	4974
rr-crossing.foada	1780	yes	67	67	7574
train-simple1.foada	5421	yes	43	44	2893
train-simple2.foada	10177	yes	111	113	8386
train-simple3.foada	15961	yes	196	200	15041
fischer-mutex2.foada	3000	yes	23	23	808
fischer-mutex3.foada	4452	yes	33	33	1154

Table 6.1: Experiments with First-Order Alternating Data Automata

The advantage of using FOADA Checker over the INCLUDER [3] tool from [34] is the possibility of having infinite alphabet automata with hidden (local) variables, whose values are not visible in the input. In particular, this is essential for checking inclusion of timed automata that use internal clocks to control the computation.

Chapter 7

Conclusions

7.1 Summary of Contributions

In order to face the two challenges mentioned in the beginning of this thesis: (i) non-determinism and (ii) infinite alphabets, we propose two models of alternating automata over infinite alphabets: (i) alternating data automata (ADA) and (ii) first-order alternating data automata (FOADA). They both recognise the data words over infinite alphabets consisting of pairs (a, v) where a is an input event from a finite set and v is a valuation of a finite set of variables that range over a possibly infinite data domain.

In ADA model, the control states are Booleans and the transition rules are specified by a set of formulae in a combined first-order theory of states (Booleans) and data that relate past values of variables with current values of variables. But a restriction of the ADA model is that, there is not hidden variable, hence all the data values taken by the variables are visible in the input. But in FOADA model, the arguments of a predicate atom track the values of the internal variables associated with the state, and these values are invisible in the input sequence, which overcomes the restriction of the ADA model.

With these two alternating models, Boolean operations of union, intersection and complement can be done in linear time, thus matching the complexity of performing these operations in the finite-alphabet case.

However, the price to be paid here is that the emptiness checking becomes undecidable. For this reason, we provide two efficient semi-algorithms for emptiness checking: (i) lazy predicate abstraction [33] and (ii) IMPACT method [45]. These semi-algorithms are proven to terminate by returning a word from the language of the given automaton if one exists; but if the language of the given automaton is empty, then the termination is not guaranteed.

The main application of these two models is checking inclusions between various classes of automata extended with variables ranging over infinite domains that recognise languages over infinite alphabets. The most widely known such classes are (i) **timed automata** [4] and (ii) **finite-memory automata** [36]. In both cases, complementation is not possible inside the class and inclusion is undecidable. Our contribution here is providing a systematic semi-algorithm for

these decision problems. In addition, we can extend **generic register automata** [34] inclusion checking framework by allowing monitor (right-hand side) automata to have local (hidden) variables that are not visible in the language.

Another application is checking safety (mutual exclusion, absence of deadlocks, etc.) and liveness (termination, lack of starvation, etc.) properties of parameterised concurrent programs, consisting of an unbounded number of replicated threads that communicate via a fixed set of global variables (locks, counters, etc.). The verification of parametric programs has been reduced to checking the emptiness of a possibly infinite sequence of first-order alternating data automata, called **predicate automata** [26, 27], encoding the inclusion of the set of traces of a parametric concurrent program into increasingly general proof spaces, obtained by generalisation of counter-examples. The program and the proof spaces are first-order alternating data automata over the infinite alphabet of pairs consisting of program statements and thread identifiers.

Besides the theoretical parts, we also have developed a tool - **FOADA Checker** [62], mainly used for checking inclusion between two automata or checking emptiness of an automaton. FOADA Checker is written in Java, via Java-SMT interface [57] and using Z3 SMT solver [53] for spuriousness, coverage queries and interpolant generation. The IMPACT semi-algorithm [45] has been implemented in the tool to check the emptiness of an automaton. The supported input automata are: (i) predicate automata [26, 27], (ii) alternating data automata and (iii) first-order alternating data automata. These input automata can be parsed by our own parser written in ANTLR4 [50], and they are all stored as FOADA once they have been parsed by the tool.

The advantage of using FOADA Checker over the INCLUDER [3] tool from [34] is the possibility of having infinite alphabet automata with hidden (local) variables, whose values are not visible in the input. In particular, this is essential for checking inclusion of timed automata that use internal clocks to control the computation.

7.2 Future Work

For the moment, the examples of alternating data automata (ADA) and first-order alternating data automata (FOADA) for timed automata (TA) inclusion problems, array logic entailments, hardware circuit verification problems and parametric verification problems are produced manually from some existing classical examples written in C/C++, such as abp.ada, train.ada, rr-crossing.foada, array_rotation.ada, train-simple1.foada, etc. (see *Section 6.5*). This excludes the possibility of using huge classical examples as the inputs of our tool (FOADA Checker) since the manual transformation requires too much work and some errors might occur during the manual transformation. We are thinking of studying those C/C++ examples, and extending the parser in our tool so that the tool can directly parse those examples and generate corresponding ADA or FOADA.

We are also going to apply our models and tool to more kinds of verification problems, such as parametric system verification [10], which asks whether a system composed of n replicated processes is safe, for all $n \geq 2$. By safety we mean that every execution of the system stays clear of a set of global error configurations, such as deadlocks or mutual exclusion violations. Even if

we assume each process to be finite-state and every interaction to be a synchronization of actions without data exchange, the problem remains challenging because we want a general proof of safety, that works for any number of processes. In general, parametric verification is undecidable if unbounded data is exchanged [6], while various restrictions of communication (rendez-vous) and architecture (ring, clique) define decidable sub-problems [5, 14, 25, 31]. Seminal works consider rendez-vous communication, allowing a fixed number of participants [14, 25, 31], placed in a ring [14, 25] or a clique [31]. Recently, MSO-definable graphs (with bounded tree and clique-width) and point-to-point rendez-vous communication were considered in [5].

Bibliography

[1] Smt2 format. http://smtlib.cs.uiowa.edu/language.shtml.

[2] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–174, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] Radu Iosif Adam Rogalewicz, Tomas Vojnar. Includer. http://www.fit.vutbr.cz/research/groups/verifit/tools/includer/.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[5] Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. Parameterized model checking of rendezvous systems. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 109–124, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[6] K R Apt and D C Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, May 1986.

[7] Dietmar Berwanger, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Sangram Raje. Strategy construction for parity games with imperfect information. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, pages 325–339, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12:27, 07 2011.

[9] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In Oscar H. Ibarra and Bala Ravikumar, editors, *Implementation and Applications of Automata*, pages 57–67, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[10] Marius Bozga, Radu Iosif, and Joseph Sifakis. Structural invariants for parametric verification of systems with almost linear architectures, 2019.

[11] Egon Börger, Erich Graedel, and Yuri Gurevich. *The Classical Decision Problem.* 01 1997.

[12] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[13] Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Algorithms for omega-regular games with imperfect information. In Zoltán Ésik, editor, *Computer Science Logic*, pages 287–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[14] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 240–248, New York, NY, USA, 1986. ACM.

[15] Jean-Baptiste Courtois and Sylvain Schmitz. Alternating Vector Addition Systems with States. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *39th International Symposium on Mathematical Foundations of Computer Science*, volume 8634 of *Lecture Notes in Computer Science*, pages 220–231, Budapest, Bulgaria, August 2014. Springer.

[16] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, 1957.

[17] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[18] Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. *Electronic Notes in Theoretical Computer Science*, 336, 10 2016.

[19] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV'17)*, pages 47–67, 07 2017.

[20] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 17–30, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[21] M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. Antichains: Alternative algorithms for ltl satisfiability and model-checking. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 63–77, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[22] Normann Decker, Peter Habermehl, Martin Leucker, and Daniel Thoma. Ordered navigation on multi-attributed data words. *CoRR*, abs/1404.6064, 2014.

[23] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 2–22, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[24] Laurent Doyen and Jean-François Raskin. Antichains for the automata-based approach to model-checking. *Logical Methods in Computer Science*, 5, 2009.

[25] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 85–94, New York, NY, USA, 1995. ACM.

[26] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proof spaces for unbounded parallelism. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 407–420, New York, NY, USA, 2015. ACM.

[27] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proving liveness of parameterized programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 185–196, New York, NY, USA, 2016. ACM.

[28] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for ltl realizability. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 263–277, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] Seth Fogarty and Moshe Y. Vardi. Büchi complementation and size-change termination. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 16–30, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[30] Laurent Fribourg. A closed-form evaluation for extended timed automata. Technical report, CNRS & ECOLE NORMALE SUP'ERIEURE DE CACHAN, 1998.

[31] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, July 1992.

[32] Joseph Y. Halpern. Presburger arithmetic with unary predicates is π11 complete. *The Journal of Symbolic Logic*, 56(2):637–642, 1991.

[33] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, January 2002.

[34] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Abstraction refinement and antichains for trace inclusion of infinite state systems. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*, pages 71–89, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[35] Radu Iosif and Xiao Xu. Abstraction refinement for emptiness checking of alternating data automata. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–111, Cham, 2018. Springer International Publishing.

[36] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, November 1994.

[37] Fondazione Bruno Kessler and DISI-University of Trento. Mathsat5. http://mathsat.fbk.eu.

[38] Z. Kincaid. Duet. https://github.com/zkincaid/duet.

[39] Z. Kincaid. Parallel proofs for parallel programs. PhD thesis, University of Toronto, 2016.

[40] Z. Kincaid. Predicate automata. https://github.com/zkincaid/duet/tree/ark2/regression/predicateAutomata.

[41] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Commun. ACM*, 55(2):103–111, February 2012.

[42] P. Lincoln, J. Michell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 662–671 vol.2, Oct 1990.

[43] Roger C. Lyndon. An interpolation theorem in the predicate calculus. *Journal of Symbolic Logic*, 25(3):273–274, 1960.

[44] Ken L. McMillan. Applying sat methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 250–264, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[45] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 123–136, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[46] Kenneth L. McMillan. Lazy annotation revisited. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 243–259, Cham, 2014. Springer International Publishing.

[47] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27:356–364, 1980.

[48] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5, 04 2002.

[49] University of Freiburg. Smtinterpol. https://ultimate.informatik.uni-freiburg.de/smtinterpol/.

[50] Terence Parr and Sam Harwell. Antlr4. https://www.antlr.org.

[51] M. Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt.* 1931.

[52] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[53] Microsoft Research. Z3 smt solver. https://github.com/Z3Prover/z3.

[54] Andrew Reynolds, Tim King, and Viktor Kuncak. Solving quantified linear arithmetic by counterexample-guided instantiation. *Form. Methods Syst. Des.*, 51(3):500–532, December 2017.

[55] Philipp Ruemmer. Princess. http://www.philipp.ruemmer.org/princess.shtml.

[56] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 346–362, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[57] SoSy-Lab. Java smt. https://github.com/sosy-lab/java-smt.

[58] Hellis Tamm and Margus Veanes. Theoretical aspects of symbolic automata. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, pages 428–441, Cham, 2018. Springer International Publishing.

[59] Zerksis D. Umrigar and Vijay Pitchumani. Formal verification of a real-time hardware design. In *Proceedings of the 20th Design Automation Conference*, DAC '83, pages 221–227, Piscataway, NJ, USA, 1983. IEEE Press.

[60] Moshe Y. Vardi. *Alternating Automata and Program Verification*, pages 471–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[61] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 640–654, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[62] X. XU. Foada checker. https://github.com/cathiec/FOADA.