Basic Research in Computer Science

Logics and Automata for Verification: Expressiveness and Decidability Issues

Jesper G. Henriksen

BRICS Dissertation Series **DS-00-6**

ISSN 1396-7002 **May 2000**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/00/6/`

Logics and Automata for Verification: Expressiveness and Decidability Issues

Jesper Gulmann Henriksen

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Logics and Automata for Verification: Expressiveness and Decidability Issues

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
Ph.D. Degree

by
Jesper Gulmann Henriksen
February 29, 2000

Abstract

This dissertation investigates and extends the mathematical foundations of logics and automata for the interleaving and synchronous noninterleaving view of system computations with an emphasis on decision procedures and relative expressive powers, and introduces extensions of these foundations to the emerging domain of noninterleaving asynchronous computations. System computations are described as occurrences of system actions, and tractable collections of such computations can be naturally represented by finite automata upon which one can do formal analysis. Specifications of system properties are usually described in formal logics, and the question whether the system at hand satisfies its specification is then solved by means of automata-theoretic constructions.

Our focus here is on the linear time behaviour of systems, where executions are modeled as sequence-like objects, neither reflecting nondeterminism nor branching choices. We consider a variety of linear time paradigms, such as the classical interleaving view, the synchronous noninterleaving view, and conclude by considering an emerging paradigm of asynchronous noninterleaving computation. Our contributions are mainly theoretical though there is one piece of practical implementation work involving a verification tool. The theoretical work is concerned with a range of logics and automata and the results involve the various associated decision procedures motivated by verification problems, as well as the relative expressive powers of many of the logics that we consider.

Our research contributions, as presented in this dissertation, are as follows. We describe the practical implementation of the verification tool Mona. This tool is basically driven by an engine which translates formulas of monadic second-order logic for finite strings to deterministic finite automata. This translation is known to have a daunting complexity-theoretic lower bound, but surprisingly enough, it turns out to be possible to implement a translation algorithm which often works efficiently in practice; one of the major reasons being that the internal representation of the constituent automata can be maintained symbolically in terms of binary decision diagrams. In effect, our implementation can be used to verify the so-called safety properties because collections of finite strings suffice to capture such properties.

For reactive systems, one must resort to infinite computations to capture the so-called liveness properties. In this setting, the predominant specification mechanism is Pnueli's LTL which turns out to be computationally tractable and, moreover, equal in expressive power to the first-order fragment of monadic

second-order logic. We define an extension of LTL based on the regular programs of PDL to obtain a temporal logic, DLTL, which remains computationally feasible and is yet expressively equivalent to the full monadic second-order logic.

An important class of distributed systems consists of networks of sequential agents that synchronize by performing common actions together. We exhibit a distributed version of DLTL and show that it captures exactly all linear time properties of such systems, while the verification problem, once again, remains tractable.

These systems constitute a subclass of a more general class of systems with a static notion of independence. For such systems the set of computations constitute interleavings of occurrences of causally independent actions. These can be grouped in a natural manner into equivalence classes corresponding to the same partially-ordered behaviour. The equivalence classes of computations of such systems can be canonically represented by restricted labelled partial orders known as (Mazurkiewicz) traces. It has been noted that many properties expressed as LTL-formulas have the "all-or-none" flavour, i.e. either all computations of an equivalence class satisfy the formula or none do. For such properties (e.g. "reaching a deadlocked state") it is possible to take advantage of the noninterleaving nature of computations and apply the so-called partial-order methods for verification to substantially reduce the computational resources needed for the verification task. This leads to the study of linear time temporal logics interpreted directly over traces, as specifications in such logics are guaranteed to have the "all-or-none" property. We provide an elaborate survey of the various distributed linear time temporal logics interpreted over traces.

One such logic is TLC, through which one can directly formulate causality properties of concurrent systems. We strengthen TLC to obtain a natural extended logic TLC* and show that the extension adds nontrivially to the expressive power. In fact, very little is known about the relative expressive power of the various logics for traces. The game-theoretic proof technique that we introduce may lead to new separation results concerning such logics.

In application domains such as telecommunication software, the synchronous communication mechanism that traces are based on is not appropriate. Rather, one would like message-passing to be the basic underlying communication mechanism. Message Sequence Charts (MSCs) are ideally suited for the description of such scenarios, where system executions constitute partially ordered exchanges of messages. This raises the question of what constitutes reasonable collections of MSCs upon which one can hope to do formal analysis. We propose a notion of regularity for collections of MSCs and tie it up with a class of finite-state devices characterizing such regular collections. Furthermore, we identify a monadic second-order logic which also defines exactly these regular collections.

The standard method for the description of multiple scenarios in this setting has been to employ MSC-labelled finite graphs, which might or might not describe collections of MSCs regular in our sense. One common feature is that these collections are finitely generated, and we conclude by exhibiting a subclass of such graphs which describes precisely the collections of MSCs that are both finitely generated and regular.

To my parents
For giving me all the opportunities

Acknowledgments

First and foremost I would like to thank my supervisor Mogens Nielsen. His contagious enthusiasm of logics and concurrency has been a constant source of energy and inspiration. I'm grateful for his guidance, support, and willingness to let me pursue my research interests, while skillfully keeping me on a steady course.

I am very grateful to my advisor and mentor P. S. Thiagarajan without whom this dissertation would never have come into existence. Working under his resourceful and pleasant guidance has been a truly enjoyable and immensely rewarding experience that I wouldn't have missed for the world. It has been a great privilege to get the opportunity to learn from him. I thank him for all his valuable feedback and the many enlightening discussions about both academic and non-academic topics.

Moreover, I would also like to thank the evaluation committee, consisting of Henrik Reif Andersen, Antoine Petit, and Erik Meiniche Schmidt, for interesting discussions and valuable comments at my Ph.D. defense.

I thank all the people at Chennai Mathematical Institute for making my visits to India both fruitful and memorable. I have benefited greatly from working with K. Narayan Kumar and, in particular, Madhavan Mukund whose enjoyable company I've had the pleasure of on many occasions. I have been extremely fortunate in having a very good friend in Deepak D'Souza, whose hospitality and kindness is unsurpassed. I thank him for taking me to the football sessions at Matscience, and his tireless efforts making my stays in Chennai perhaps the most memorable experiences of my life.

I would also like to thank Wolfgang Thomas for letting me spend six months at Rheinisch-Westfälische Technische Hochschule in Aachen. I learned a lot there and value my stay deeply. I thank my friends, Martin Leucker and Oliver Matz, for many interesting discussions and making me feel at home there. Special thanks go to Martin Lange for being a great friend and for our many enjoyable hours conjecturing about life, love, theoretical computer science, football, the Eifel, and various combinations thereof. I admire him for his rare capability of explaining everyday life succinctly in terms of complexity theory and movie quotes.

Thanks are also due to Nils Klarlund for initiating the Mona project and exposing me to the wonders of monadic second-order logic. Writing my first paper with him and Jakob Jensen, Michael Jørgensen, Bob Paige, Theis Rauhe,

and Anders Sandholm was indeed a rewarding experience.

Special thanks go to my good friend and fellow student Anders Sandholm. Since teaming up on the very first day of undergraduate study, we have collaborated on numerous assignments and projects, travelled together, and shared an office for many years. I will always look back at this period of my life with great joy and gratitude.

I would also like to thank everybody at BRICS — and in particular Claus Brabrand, Thomas Hildebrandt, Thomas Hune, Rune Lyngsø, Anders Møller, Rasmus Pagh, Jakob Pagter, Jiří Srba, Christian Storm and numerous other fellow Ph.D. students — for contributing to an outstanding and pleasant research environment that makes everyday life so enjoyable, always making me feel proud of being a BRICS Ph.D. student. At the computer science department I would like to thank the incredibly helpful secretaries and staff for smoothly operating DAIMI.

I am, however, by far most grateful to my parents, Annie and Jørgen Henriksen, for their endless love, encouragement, and support, which made it all possible. They have at every occasion gone beyond what anybody could possibly expect of them, to provide me with excellent opportunities, for which I will be eternally grateful. To them I dedicate this dissertation.

Jesper Gulmann Henriksen,
Århus, February 29th, 2000.

Contents

Chapter 1

Introduction

> Never send a human to do a machine's job.
>
> —Agent Smith, *The Matrix (1999)*

As technological advances in computing lead to increased levels of automation, the need for correctly functioning software and hardware is becoming more crucial. In application domains involving safety critical systems, an unforeseen error can have a devastating impact. To be specific, failures often have unacceptable consequences in applications areas such as air traffic control, medical monitoring equipment, electronic commerce, and power plants.

Of course, in every design of a system a lot of effort is being employed to increase the confidence that the system correctly achieves its designated goal. This may include numerous simulations of an abstraction of the system while in its early design stages, and extensive testing of a concrete realization of (part of) the system. While such techniques are often adequate to detect many design and implementation errors, it is usually impossible to consider all possible scenarios solely by testing.

These considerations have given rise to what is commonly referred to as "Formal Methods". Here, a mathematical model of the system is constructed which abstractly captures all the interesting computations of the system. The model is then rigorously investigated for the presence or absence of certain properties. Based on the results of this analysis, the designer can reach conclusions regarding the functional correctness of the system. The key point here being that the level of confidence attained is comparable to that associated with the (correct!) proof of a mathematical statement.

It turns out that the formal method popularly known as *model checking*, especially when applied to finite-state systems, can be fully automated with the help of software tools. This is in sharp contrast to most other methods such as theorem-proving where a good deal of intervention is required from an expert. In this dissertation, we will concentrate mainly on issues centered around the model checking approach to verification of finite-state systems. Even though systems might be infinite in nature there are many application areas, for instance

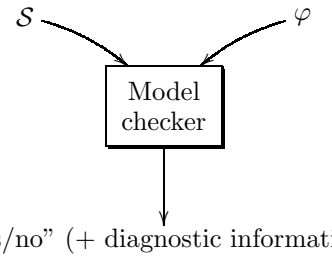


Figure 1.1: Structure of a model checker.

communication protocols and microprocessors, which deal with only finite data domains, and such applications can be expressed as finite-state systems. Moreover, a number of the techniques of this setting extend to model checking of infinite-state systems [13, 85].

1.1 Model Checking

Model checking is a term coined by Clarke and Emerson to denote a particular approach to formal verification. The process of model checking consists of three main stages. Firstly, a formal description or *system model* $\mathcal{S}$ of the intended system design is constructed. Usually, this is achieved either by transforming a graphical representation explicitly identifying states and state changes of the system, or by compilation of a program text written in a specifically designed modeling language. Whichever approach is used, the model checking program builds an internal representation upon which analysis is performed.

Secondly, the properties to be investigated of the system, its *specification* φ, must be precisely described. Sometimes the model checking program has algorithms for checking only a limited number of specific properties such as deadlock or mutual exclusion. However, more sophisticated assertions are formulated within the framework of *formal logics*.

Finally, the *verification* is performed by the model checking program which, by analyzing the internal representation, automatically computes an answer to the question of whether or not the model formally satisfies its specification with some precise semantics. As such, a "no" answer to the correctness question is not very useful. However, model checking programs often provide some diagnostic information in case the answer to the correctness question is negative. Such information helps the designer to identify errors and correct them.

Of course, the value of model checking is limited by the appropriateness of the formal model constructed. A model must represent the concrete system in a such way that it captures at least the system actions that are relevant to the correctness requirements that need to be checked. We shall consider neither the aspects of constructing the model nor assessing its quality any further here.

There are many variations to the concept of model checking. A multitude of

classes of systems models, specification formalisms and verification algorithms have been considered in the literature (see e.g. [20] for a recent exposition), and model checking of infinite systems is becoming a popular field of research [13, 85]. We will however restrict our attention to finite-state systems and temporal logics.

1.2 Logics and Automata

In the model checking area, the prevailing paradigm at present is that of *reactive* systems. These are systems continuously (and possibly infinitely) computing and interacting with other processes or an environment. Here the possible histories or *computation sequences* of the system are of primary interest. The system behaviour is captured by collections of such computation sequences, which in turn can be inspected or transformed in order to investigate whether the system satisfies its specification. Thus finite-state systems can be viewed as finite automata accepting sequences of system actions. The *model checking problem* in this setting is, for a given finite-state automaton $\mathcal{S}$ representing the system and a given specification φ, to decide whether every string accepted by $\mathcal{S}$ satisfies φ.

The connection between logic and automata is very well understood and goes back to the pioneering work of Büchi and Elgot [14, 34] in the early sixties. In fact, they demonstrate an intimate relationship between logic and automata via the basic result that *monadic second-order logic* and finite automata describe the same collections of strings. These collections constitute the so-called *regular languages* of sequences. An equally useful fact is that these translations between logic and automata are constructive, for instance, any sentence of monadic-second order logic can be algorithmically transformed to an equivalent automaton and vice versa.

When such prerequisites are met, the question of whether a finite-state system meets its specification, can then be phrased and settled solely in terms of constructions on finite automata and corresponding language inclusions. Often the model checking problem is solved in terms of the *satisfiability problem* for the specification logic, which is to decide whether there exists any computation sequence satisfying φ for a given specification formula. A common approach is to translate φ to an automaton $\mathcal{A}_\varphi$ accepting the set of strings satisfying φ, which can easily be checked for emptiness. The model checking problem then amounts to checking whether there is a string accepted by $\mathcal{S}$ accepting the negation of the specification, i.e. checking the product of $\mathcal{S}$ and $\mathcal{A}_{\neg\varphi}$ for emptiness. Moreover, by simple investigations of this automaton, diagnostic information can be easily recovered and supplied to the designer. This makes verification by automata a very versatile tool, because the general approach can be used for any specification logic which is algorithmically translatable to finite automata.

Temporal logic was introduced into the area of program verification in a groundbreaking paper by Pnueli [113] more than 20 years ago, and is by now thoroughly investigated and well understood. See e.g. [35, 83] for expositions

on temporal logics. The key point is that correctness properties of systems are usually formulated in terms of relative orderings in time of certain sets of system actions, and the characteristic feature of temporal logic is that it facilitates reasoning about such temporal relationships without introducing time explicitly.

1.3 Outline of Dissertation

This dissertation consists of two parts. Part II contains our research contributions in the form of published papers and technical reports, while Part I provides the broad conceptual and technical context for the results appearing in Part II. In this sense, the overview material presented in Part I, especially in terms of the topics it addresses, is neither complete nor is it meant to be.

Part I consists of three chapters constituting an overview of logics for formal verification with an emphasis on their expressive powers and automata-theoretic decision procedures. The three chapters aim at surveying three different paradigms towards formal verification. Chapter 2 exposes the classical *interleaving* view of computations as being totally ordered sequences of system actions. In this sense it provides a direct realization of the general scheme of automata-based verification as introduced here. In the two chapters following we generalize the notion of computation "sequence" to the richer domains of restricted labelled partial orders. More precisely, we survey in Chapter 3 the extensions of the approach to the *noninterleaving synchronous* view from the perspective of Mazurkiewicz traces [86], where the distributed processes communicate by performing common actions together. Finally, Chapter 4 describes an emerging theory within the *noninterleaving asynchronous* view of communication. Here the distributed processes communicate by means of message-passing specified as message sequence charts [68]. Together these three classes of behavioural models provide a representative overview of the various *linear time* approaches to formal specification and verification of systems. The question whether to use linear time or branching time logics for verification is a subject of intense debate [146], but we will concentrate exclusively on the linear time paradigm here.

Part II summarizes contributions within this field in which I have familiarized myself during my graduate studies. It consists of seven chapters as briefly described below—each constituting a self-contained reprint of a contribution to the area surveyed in Part I.

Interleaving Paradigm: Computation Sequences

Chapter 5 [55]: *"*Mona*: Monadic Second-Order Logic in Practice".* Joint work with Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm. Appears in Proceedings of the 1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95), LNCS 1019. It also appears as BRICS technical

report RS-95-21. The present version is an updated version.

Chapter 6 [60]: *"Dynamic Linear Time Temporal Logic"*. Joint work with P. S. Thiagarajan. Appears in Annals of Pure and Applied Logic 96(1-3). A preliminary and more elaborate version appears as BRICS technical report RS-97-8.

Noninterleaving Synchronous Paradigm: Mazurkiewicz Traces

Chapter 7 [61]: *"A Product Version of Dynamic Linear Time Temporal Logic"*. Joint work with P. S. Thiagarajan. Appears in Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97), LNCS 1243. It also appears as BRICS technical report RS-97-9.

Chapter 8 [138]: *"Distributed Versions of Linear Time Temporal Logic: A Trace Perspective"*. Joint work with P. S. Thiagarajan. A chapter of Reisig and Rozenberg (Eds.): Lectures on Petri Nets I: Basic Models, LNCS 1491. The survey also appears as BRICS technical report RS-98-8.

Chapter 9 [54]: *"An Expressive Extension of TLC"*. Appears in Proceedings of the 5th Asian Computing Science Conference (ASIAN'99), LNCS 1742. Also appears as BRICS technical report RS-99-26. The present version is the full version invited for publication in the ASIAN'99 Special Issue of International Journal of Foundations of Computer Science.

Noninterleaving Asynchronous Paradigm: Message Sequence Charts

Chapter 10 [56, 57]: *"Regular Collections of Message Sequence Charts"*. Joint work with Madhavan Mukund, K. Narayan Kumar, and P. S. Thiagarajan. Appears in Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'00), LNCS 1893. Excerpts appear in BRICS technical report RS-99-52. The present version is an extended version.

Chapter 11 [56, 58]: *"On Message Sequence Graphs and Finitely Generated Regular MSC Languages"*. Joint work with Madhavan Mukund, K. Narayan Kumar, and P. S. Thiagarajan. Appears in Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP'00), LNCS 1853. Excerpts appear in BRICS technical report RS-99-52. The present version is an extended version.

Part I

Overview

Chapter 2

Interleaving Paradigm

In this chapter we consider the verification setting in which computations are strings of system actions. We begin by reviewing the theories of automata over strings in Section 2.1 and define the basic notions of regular languages. In Section 2.2 we define the monadic second-order logic (MSO) of strings and sketch how it captures all regular languages of strings. Following that, we define in Section 2.3 Pnueli's linear time temporal logic (LTL), which constitutes a cornerstone of automated formal verification. We then show how the satisfiability and model checking problems are solved in the approach of Chapter 1. Finally, in Section 2.4 we consider the expressive power of the logics introduced in this chapter. We bring out how LTL corresponds to the first-order fragment of MSO and consider how it can be extended to precisely capture MSO.

This material provides the background for our contributions in Chapters 5 and 6. Moreover, many of the ideas presented here have also played an important role in the developments of the theories in Chapters 7–11.

2.1 Automata over Strings

To bring out the developments of automata over strings, we fix a finite nonempty alphabet of actions Σ throughout the chapter. We let a, b range over Σ and refer to members of Σ as *actions*. Σ^* is the set of finite strings over Σ and Σ^ω is the set of (countably) infinite strings generated by Σ with $\omega = \{0, 1, 2, \ldots\}$. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and denote the null word by ε. We let σ, σ' range over Σ^ω and τ, τ', τ'' range over Σ^*. Moreover, $\preceq$ is the usual prefix ordering defined over Σ^* and for $u \in \Sigma^\infty$, we let $\mathrm{prf}(u)$ be the set of finite prefixes of u. Finally, we let $|\sigma|_a$ denote the number of occurrences of a in the string σ.

Throughout this chapter a *language* is a collection of strings $L \subseteq \Sigma^*$, whereas an *ω-language* is a subset of Σ^ω. Whenever necessary, we will treat both finite and infinite strings on an equal footing and just refer to $L \subseteq \Sigma^\infty$ as being a language. A language can be naturally viewed as a *property*, described explicitly by the set of all strings possessing the given property. This will become

apparent when we consider logical definability later in this chapter. We will use "language" and "property" interchangeably throughout the text.

We describe first automata over finite strings and then their extensions to the infinite setting.

2.1.1 Automata over finite strings

A well-studied notion of classical formal language theory [75] is the collections of strings described by *finite automata*. A (nondeterministic) finite automaton (NFA) over Σ is a quintuple $\mathcal{A} = (Q, \Sigma, \Delta, Q_0, F)$, where Q is a finite set of states , $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.

A run of $\mathcal{A}$ over a string $\tau \in \Sigma^*$ is a map $\rho : \mathrm{prf}(\tau) \to Q$ such that:

- $\rho(\varepsilon) \in Q_0$.
- $\rho(\tau') \xrightarrow{a} \rho(\tau' a)$ for each $\tau' a \in \mathrm{prf}(\tau)$.

The run ρ is said to be *accepting* iff $\rho(\tau) \in F$. To tie everything together, $L(\mathcal{A})$, the *language of finite strings of* Σ^* *accepted by* $\mathcal{A}$, is $L(\mathcal{A}) = \{\tau \in \Sigma^* \mid \exists$ an accepting run of $\mathcal{A}$ over $\tau\}$.

It is well-known that $L \subseteq \Sigma^*$ is accepted by some nondeterministic automaton if and only if it is accepted by some deterministic finite automaton, i.e. where Q_0 is a singleton set and the automaton is equipped with a transition *function* $\delta : Q \times \Sigma \to Q$. Hence, nondeterminism does not increase the distinguishing power of finite automata over finite strings. The class of languages of finite strings over Σ accepted by finite-state automata are the *regular languages*. These languages are also commonly referred to as being *recognizable languages*.

Another description formalism for languages of finite strings are the *regular expressions* of Kleene [74]. Formally, the set of regular expressions over Σ are given by:

$$\mathrm{RE}(\Sigma) ::= a \mid \pi_0 + \pi_1 \mid \pi_0 ; \pi_1 \mid \pi^* \ , \quad a \in \Sigma.$$

With each regular expression we associate a set of finite strings via the map $||\cdot|| : \mathrm{RE}(\Sigma) \longrightarrow 2^{\Sigma^*}$. This map is defined in the obvious fashion. In particular, the semantics of the *Kleene iteration* is given as $||\pi^*|| = \bigcup_{i \in \omega} ||\pi||^i$, where for $L \subseteq \Sigma^*$;

- $L^0 = \{\varepsilon\}$ and
- $L^{i+1} = \{\tau_0 \tau_1 \mid \tau_0 \in L$ and $\tau_1 \in L^i\}$ for every $i \in \omega$.

We will sometimes use π^+ to denote $\pi ; \pi^*$. Occasionally, we will refer to the *star-free regular expressions* as the set of expressions obtained from the regular expressions above by replacing the Kleene-star iteration operator with the operation of complementation $\overline{\pi}$ (with respect to Σ^*).

Kleene's classical theorem can be brought out as follows.

Theorem 2.1.1 (Kleene [74]) *Let $L \subseteq \Sigma^*$. Then L is regular if and only if $L = ||\pi||$ for some regular expression $\pi \in \mathrm{RE}(\Sigma)$.*

Thus both finite automata and regular expressions characterize the class of regular languages of finite strings over Σ.

2.1.2 Automata over infinite strings

Collections of finite strings over some alphabet of actions suffice for capturing the *safety* properties of reactive systems. Such properties essentially express that some property holds of every reachable state of the system, and are usually used to assert that no "bad" states are reached throughout the computations of the system. However, most interesting reactive systems are nonterminating and many important properties of such systems are inherently *liveness* properties, asserting that at any given time of a computation something "good" will eventually happen. (see e.g. [2, 80, 82] for classifications of satisfy and liveness properties.) Such liveness properties, which include various notions of fairness, can only be captured by infinite computations. Hence finite automata over infinite strings and regular languages of such strings are at the center of our concerns.

Guided by the constructions for the finite string case, a stable and coherent theory of regular languages of infinite strings $\sigma \in \Sigma^\omega$ has been developed. It turns out that most characterizations of the finite string case can be carried over to the setting of ω-languages. We start by introducing the notions of finite-state automata over infinite strings.

A finite automaton accepting infinite strings is obtained from the nondeterministic finite automata introduced above by suitably modifying the criterion of acceptance. Hence an automaton accepting infinite strings over Σ is a quintuple $\mathcal{B} = (Q, \Sigma, Q_0, \Delta, Acc)$ where Acc is an *acceptance condition* expressing when a given infinite run is deemed accepting. The notion of run is carried over directly, i.e. a run of $\mathcal{B}$ over a string $\sigma \in \Sigma^\omega$ is a map $\rho : \mathrm{prf}(\sigma) \to Q$ such that:

- $\rho(\varepsilon) \in Q_0$.
- $\rho(\tau) \xrightarrow{a} \rho(\tau a)$ for each $\tau a \in \mathrm{prf}(\sigma)$.

Several acceptance conditions have been proposed in the literature. The most common one is a straightforward generalization of the finite string case called *Büchi acceptance condition*, which is given as $Acc = F$ for a set of final states $F \subseteq Q$. The run ρ is *accepting* iff $\inf(\rho) \cap F \neq \emptyset$, where $\inf(\rho) \subseteq Q$ is given by $q \in \inf(\rho)$ iff $\rho(\tau) = q$ for infinitely many $\tau \in \mathrm{prf}(\sigma)$.

An automaton with a Büchi acceptance condition is called a *Büchi automaton*. In the obvious manner we define the language accepted by $\mathcal{B}$ as $L(\mathcal{B}) = \{\sigma \in \Sigma^\omega \mid \exists \text{ an accepting run of } \mathcal{B} \text{ over } \sigma\}$. We say that $L \subseteq \Sigma^\omega$ is *Büchi recognizable* if it is accepted by some Büchi automaton. An example of a Büchi automaton $\mathcal{B}$ is given in Figure 2.1, where $L(\mathcal{B}) = (a + b)^* b^\omega$. To avoid

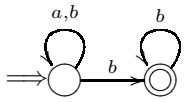


Figure 2.1: Büchi automaton accepting the language $(a+b)^*b^\omega$.

confusion, we will throughout denote automata over finite strings by $\mathcal{A}$ while using $\mathcal{B}$ to denote (Büchi) automata over infinite strings.

By replacing the Kleene-star with an infinite iteration operator π^ω in the regular expressions and furthermore allowing left concatenation by regular expressions of $\mathrm{RE}(\Sigma)$, one obtains the *ω-regular expressions*. With these definitions Büchi [14] showed that Kleene's Theorem carries over to the setting of ω-languages in the straightforward manner. Thus it seems natural to take the *ω-regular languages* to be the Büchi recognizable ω-languages. There is a large body of evidence [140] showing that this is the "right" notion of regularity and we will sketch one such piece of evidence in the form of a characterization in terms of logic in Section 2.2.

While most of the properties are carried over smoothly, deterministic Büchi automata are strictly weaker than their nondeterministic counterparts. More specifically, one can show that the language $L \subseteq \{a,b\}^\omega$ consisting of strings of a's and b's with only a finite number of a's, is *not* accepted by any deterministic Büchi automaton. On the other hand, it is easy to see that L is accepted by the automaton in Figure 2.1.

There is however a number of generalizations of acceptance conditions such that deterministic automata capture the full class of ω-regular languages. One such possibility is *Muller automata*, where the acceptance condition is given as a family of accepting sets $\mathcal{F} = \{F_i\}_{i=1}^n$, where each $F_i \subseteq Q$. An infinite run ρ is accepting in case $\inf(\rho) = F_i$ for some $1 \leq i \leq n$, i.e. the set of states encountered infinitely often along ρ is exactly one of the F_i's. Another possibility is *Rabin automata*, where the acceptance condition is given by a set of accepting pairs $\Omega = \{(L_i, U_i)\}_{i=1}^n$. In this regime, an infinite run ρ is accepting if there exists some $1 \leq i \leq n$ such that $\inf(\rho) \cap L_i = \emptyset$ and $\inf(\rho) \cap U_i \neq \emptyset$, i.e. the states in L_i are only visited a finite number of times whereas some state of U_i occurs infinitely often. *Streett automata* are in some sense dual to the Rabin automata in that the acceptance condition is also given as a set of acceptance pairs $\{(R_i, S_i)\}_{i=1}^n$, but the run is accepting in case for every i, $\inf(\rho) \cap R_i \neq \emptyset$ implies $\inf(\rho) \cap S_i \neq \emptyset$.

It turns out that the class of ω-languages recognized by *deterministic* Muller, Rabin, and Streett automata, respectively, is precisely the class of ω-regular languages. Moreover, *nondeterministic* Muller, Rabin, and Streett automata, respectively, accept the same class of languages. See [122] for an overview of conversions between the various acceptance conditions and complexities of the corresponding *emptiness problems*, which is to determine whether or not the language accepted by a given automaton is empty.

One pleasant advantage of Büchi automata is that it is very easy to solve the emptiness problem. This can be done in time linear in the size of the automaton where the size of a Büchi automaton is the number of states of the automaton. The problem is known to be logspace-complete for NLOGSPACE [128]. We also note that the *intersection problem* for Büchi automata can be easily solved. In other words, let $\mathcal{B}_1, \mathcal{B}_2$ be two Büchi automata both operating over Σ. Then one can effectively construct a Büchi automaton $\mathcal{B}$ over the same alphabet such that the language accepted by $\mathcal{B}$ is the intersection of the languages accepted by $\mathcal{B}_1$ and $\mathcal{B}_2$. Moreover, the size of $\mathcal{B}$ can be assumed to be bounded by $2n_1n_2$ where n_1 is the size of $\mathcal{B}_1$ and n_2 is the size of $\mathcal{B}_2$ [140].

Complementation of Büchi automata can be performed using *Safra's construction*, by which one can effectively complement a Büchi automaton of size n by constructing a deterministic Rabin automaton with $2^{O(n \log n)}$ states and $O(n)$ accepting pairs [122]. This can be translated to a Büchi automaton of size $2^{O(n \log n)}$. An interesting feature of Safra's construction is that it is essentially an optimal one [122].

2.2 Monadic Second-order Logic

Automata over infinite strings were originally introduced as a basis for analysis of restricted arithmetic problems of the second-order theory of one successor (S1S). Büchi [14] showed that the requirements of this restricted system of second-order logic could be translated into acceptance problems of automata over infinite strings. As the logical system permits quantifications over only unary relations (or equivalently, sets), it has become widely known as the *monadic second-order logic* of strings.

In this section, we present the syntax and semantics of the logical system and bring out the close correspondence between the (ω-)regular languages and monadic second-order logic (MSO). We then consider how to use this correspondence in practice to implement a decision procedure for MSO.

2.2.1 Syntax and semantics of MSO

The *monadic second-order theory of (finite or infinite) strings over* Σ is denoted MSO(Σ). Its vocabulary consists of a family of unary predicates $\{Q_a\}_{a \in \Sigma}$, one for each $a \in \Sigma$; a binary predicate $\leq$; a binary predicate $\in$; a countable supply of individual variables $Var = \{x, y, z, \ldots\}$; a countable supply of set variables (i.e. monadic predicate variables) $SVar = \{X, Y, Z, \ldots\}$. The formulas of MSO($\Sigma$) are then given as:

$$\text{MSO}(\Sigma) ::= Q_a(x) \mid x \in X \mid x \leq y \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid (\exists x)\varphi \mid (\exists X)\varphi \ , \quad a \in \Sigma.$$

The set of formulas above can be used for defining both finite and infinite strings. We present here the semantics for the infinite case as follows. A structure for MSO(Σ) is a sequence $\sigma \in \Sigma^\omega$. Let $\mathcal{I}$ be an interpretation of the variables with $\mathcal{I} : Var \longrightarrow \omega$ and $\mathcal{I} : SVar \longrightarrow 2^\omega$. Then the notion of σ being

a model of φ under the interpretation $\mathcal{I}$, denoted $\sigma \models_{\mathcal{I}} \varphi$, is defined in the expected manner. In particular, $\sigma \models_{\mathcal{I}} Q_a(x)$ iff $\sigma(\mathcal{I}(x)) = a$ (note that $\sigma \in \Sigma^\omega$ is here to be viewed as $\sigma : \omega \longrightarrow \Sigma$); $\sigma \models_{\mathcal{I}} x \leq y$ iff $\mathcal{I}(x) \leq \mathcal{I}(y)$ (here $\leq$ is the usual ordering over ω); $\sigma \models_{\mathcal{I}} x \in X$ iff $\mathcal{I}(x) \in \mathcal{I}(X)$.

In a similar fashion, the semantics can be adapted to the setting of finite strings. Most notions are carried over directly with only minor modifications, which we will not go into here. (Chapter 5 provides a definition of the logic for finite strings.) Hence, we can view formulas of monadic second-order logic as also defining languages $L \subseteq \Sigma^\infty$ in the obvious manner.

Apart from the usual derived propositional connectives and universal quantifiers, we will use the following abbreviations:

- $x = y \stackrel{\text{def}}{=} x \leq y \wedge y \leq x$ and thus $x < y \stackrel{\text{def}}{=} x \leq y \wedge \neg(x = y)$.
- $x + 1 \in X \stackrel{\text{def}}{=} (\exists y)(x < y \wedge \neg(\exists z)(x < z \wedge z < y) \wedge y \in X)$. Of course, this can be generalized to $x + k \in X$ for any natural number k.
- $X \subseteq Y \stackrel{\text{def}}{=} (\forall x)(x \in X \Rightarrow x \in Y)$ and hence also (non)equality on set variables in the same manner as above. Similarly, one can construct formulas for $X = \emptyset$ and $X = \omega$ by quantifiers, and the operations of Boolean operations of union, intersection and complementation from the corresponding propositional connectives. For example, $X \cap Y = Z$ can be expressed as $(\forall x)(x \in X \wedge x \in Y \Leftrightarrow x \in Z)$. With the same technique we get:
- $X = Y + 1 \stackrel{\text{def}}{=} (\forall y)(y \in Y \Leftrightarrow y + 1 \in X)$.
- $\text{Single}(X) \stackrel{\text{def}}{=} (\exists Y)(Y \subseteq X \wedge Y \neq X \wedge \neg(\exists Z)(Z \subseteq X \wedge Z \neq X \wedge Z \neq Y))$.

Note that $\text{Single}(X)$ expresses that X is a singleton set by asserting that it has exactly one *proper* subset Y.

As usual, a sentence is a formula with no free variables. Each sentence φ defines a language denoted L_φ where $L_\varphi = \{\sigma \in \Sigma^\infty \mid \sigma \models \varphi\}$. We say that $L \subseteq \Sigma^\infty$ is *definable in* $\text{MSO}(\Sigma)$ iff there exists a sentence $\varphi \in \text{MSO}(\Sigma)$ such that $L = L_\varphi$. Often we will also use $\text{MSO}(\Sigma)$ to denote the class of languages described by sentences of the logic.

The *first-order theory of (finite or infinite) strings over* Σ is denoted $\text{FO}(\Sigma)$ and is obtained from $\text{MSO}(\Sigma)$ by abolishing the monadic second-order quantifications from the logic. The semantics and notions of first-order definability are carried over in the obvious manner.

2.2.2 MSO and the regular languages

The fundamental theorem in this area is Büchi's Theorem [14], which was the original motivation for Büchi automata.

Theorem 2.2.1 (Büchi [14]) *Let $L \subseteq \Sigma^\omega$. Then L is definable in* $\text{MSO}(\Sigma)$ *if and only if L is ω-regular.*

Let $L \subseteq \Sigma^\omega$ be given and suppose L is accepted by the Büchi automaton $\mathcal{B} = (Q, \Sigma, Q_0, \Delta, F)$. To bring out a sentence φ of $\mathrm{MSO}(\Sigma)$ defining L we will express accepting runs ρ of $\mathcal{B}$ over σ by existence of sets of positions X_q for each $q \in Q$. X_q will consist of the positions $i \in \omega$ corresponding to the prefixes $\tau_i \in \sigma$ of length i with $\rho(\tau_i) = q$. More precisely, we define

$$\begin{aligned}
\varphi \quad \stackrel{\text{def}}{=} (\exists X_{q_1}, \ldots, X_{q_n}) \quad & \Big(\bigwedge_{1 \le i \neq j \le n} X_{q_i} \cap X_{q_j} = \emptyset \\
\wedge \quad & \bigvee_{q \in Q_0} (\forall x)(\neg(\exists y)(y < x) \Rightarrow x \in X_q) \\
\wedge \quad & (\forall x)\Big(\bigvee_{(q_i, a, q_j) \in \Delta} x \in X_{q_i} \wedge Q_a(x) \wedge x + 1 \in X_{q_j} \Big) \\
\wedge \quad & \bigvee_{q \in F} (\forall x)(\exists y)(x \le y \wedge y \in X_q)\Big)
\end{aligned}$$

It is now easy to see that $L_\varphi = L$. The first conjunct asserts that the state sets are disjoint, the second one that the beginning state is initial while the third one dictates that the state sets respect the transition relation. Finally, the fourth conjunct expresses the Büchi acceptance condition.

The proof of the other direction of Büchi's Theorem proceeds in two steps. Firstly, the formulas of $\mathrm{MSO}(\Sigma)$ are rewritten into formulas of a small syntactic subset $\mathrm{MSO}_0(\Sigma)$ with only existential quantifications and the sole propositional connectives being negation and conjunction. Furthermore, all individual variables are eliminated from $\mathrm{MSO}_0(\Sigma)$ and expressed in terms of set variables explicitly forced to describe singleton sets via the Single-predicate. Finally, one arrives at a core syntax where the only atomic formulas are of the form $Q_a(X)$ and, as derived on page 14, $X \subseteq Y$ and $X = Y + 1$.

In the second step, formulas of $\mathrm{MSO}_0(\Sigma)$ are inductively translated to Büchi automata. Such formulas have free *set* variables, so a formula φ with free variables $X_1, \ldots, X_n$ can be seen as describing a language of strings over $\Sigma'_n = \Sigma \times \{0,1\}^n$. Here the ith additional "track" of a string $\sigma \in \Sigma'_n$ represents the membership status of positions of $\sigma = a_0 a_1 \ldots$ in X_i, i.e. position j of σ is in the set described by X_i whenever the ith additional component of a_j is 1. In this interpretation, one shows by induction on $\varphi(X_1, \ldots, X_n)$ that there exists a Büchi automaton $\mathcal{B}_{\varphi(X_1,\ldots,X_n)}$ over Σ'_n such that $L_{\varphi(X_1,\ldots,X_n)} = L(\mathcal{B}_{\varphi(X_1,\ldots,X_n)})$.

For the base case, each of the (now very restricted) atomic formulas are translated directly to an equivalent automaton. Figure 2.2 shows the translation step for $X_1 \subseteq X_2$ and $X_1 = X_2 + 1$, respectively, where we have assumed, for notational convenience, that $n = 2$.

The inductive step involves only $\neg, \wedge$, and $\exists$. The translations of $\neg$ and $\wedge$ correspond to the automata-theoretic constructions of complementation and product mentioned in Section 2.1.2. For the existential quantifications we utilize that the (ω-)regular languages are closed under projections of the kind required. As an example, suppose that $\varphi(X_1) = (\exists X_2)(\varphi'(X_1, X_2))$. In the notation of Figure 2.2, this amounts to replacing transition edges of $\mathcal{B}_{\varphi'(X_1,X_2)}$

Figure 2.2: Basic translations for $X_1 \subseteq X_2$ and $X_1 = X_2 + 1$.

labelled $(a, 0, 0)$ or $(a, 0, 1)$ with a transition edge labelled $(a, 0)$ and edges labelled $(a, 1, 0)$ or $(a, 1, 1)$ with $(a, 1)$.

This completes the proof, and as an important corollary it can be seen that the translation from formulas of monadic second-order logic to finite automata is constructive. We will elaborate on this in the next section. With minor modifications to the above proof idea, Büchi's Theorem can be seen to hold for *finite* strings as well.

Theorem 2.2.2 (Büchi, Elgot [34]) *Let $L \subseteq \Sigma^*$. Then L is definable in* MSO(Σ) *if and only if L is regular.*

2.2.3 Deciding monadic second-order logic

The proof of Theorem 2.2.1 yields a decision procedure for the *satisfiability problem* for MSO(Σ), i.e. given a formula φ, does there exist a model $\sigma \in \Sigma^\omega$ such that $\sigma \models \varphi$? Such an algorithm can be obtained by translating φ to $\mathcal{B}_\varphi$ and checking it for emptiness.

For the discussion of the translation, we note that a problem is *nonelementary hard* in case it cannot be solved in time bounded by a tower of exponentials of any fixed height. A basic result of complexity theory is that the satisfiability problem of MSO is nonelementary hard [131], even for the first-order fragment interpreted over finite strings. This nonelementary hardness is witnessed by sequences of quantifier alternations, i.e. formulas of the form $(\exists x_1)(\forall x_2)(\exists x_3)\ldots(\forall x_n)\varphi$. This stems from the fact that, in general, both determinization and complementation incur unavoidable exponential blow-ups in the size of the automaton. It would thus appear impossible to get a satisfactory implementation within any "reasonable" bounds of computational resources, even for very small formulas.

However, Klarlund discovered that the formulas leading to nonelementary blow-ups occur only infrequently, and that by means of techniques from the area of computer-aided verification, an implementation *might* not be impossible. In particular, by internally representing transition functions of automata using *Binary Decision Diagrams (BDDs)* [12], descriptions of the automata could often be kept small.

Chapter 5 describes a successful implementation of the decision procedure over *finite* strings incorporated into the verification tool Mona [73]. Besides the introduction of BDDs for the representation of automata, the key points are to keep the intermediate automata both *deterministic* and *minimal* after each

translation step. The time spent maintaining this representation invariant is more than compensated by the reduction in size. It turns out to be possible to directly employ the necessary automata constructions in terms of algorithms on the symbolic representations as BDDs, hence leading to a very usable and surprisingly efficient implementation.

This was the starting point of still ongoing research efforts at BRICS. See [73] for the current status of the Mona tool, which has been extended, optimized, and improved in a number of ways too substantial to mention here. While Mona works quite well over finite strings, no even comparably efficient implementation exists over infinite strings. Several key issues of the above implementation crucially rely on the fact that the underlying domain consists of finite strings. For instance, there exists no canonical minimal (let alone *deterministic*) Büchi automaton.

2.3 Linear Time Temporal Logic

Monadic second-order logic provides a succinct logical description formalism for computation sequences as seen in the previous section. Another alternative is linear time temporal logic, which allows asserting ordering in time without explicitly introducing time instances as variables of the logic.

Temporal logic was introduced into the area of formal verification of systems in a seminal paper by Pnueli [113] in the late seventies. Since then a large body of work on linear time temporal logic has appeared, and it has become a well established and well understood tool for specifying the dynamic behaviour of reactive systems.

Linear time temporal logic (LTL) was originally introduced in the *state-based* approach as describing properties of systems by means of (infinite) state sequences. We have chosen here the *action-based* approach where computations instead are sequences of actions performed by the system. The two approaches are very similar and the state-based approach can be recovered by taking the system actions as consisting of sets of atomic propositions. Furthermore, the action-based approach is appropriate in the richer setting of noninterleaving computations to be considered in Chapters 3 and 4.

In what follows, automata-theoretic constructions and expressiveness issues will play a considerable role. These topics can be treated in a simpler fashion if we eliminate atomic propositions. Most of the material we present can easily accommodate atomic propositions with some notational overhead. Hence from now on, we will not—except for some passing remarks—deal with atomic propositions. The treatment of LTL in Chapter 8 provides an exposition of these issues where both actions and atomic propositions are treated as first-class objects.

Here, we first define the syntax and semantics of LTL. We then bring out solutions to the satisfiability and model checking problems for the logic in terms of automata as sketched in Chapter 1.

2.3.1 Syntax and semantics of LTL

The set of formulas of LTL(Σ) is given by the syntax:

$$\text{LTL}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a\rangle\alpha \mid \alpha\, U\, \beta\ , \quad a \in \Sigma.$$

Through the rest of this section α, β will range over LTL(Σ).

A model of LTL(Σ) is a sequence $\sigma \in \Sigma^\omega$. Let σ be a model, $\tau \in \text{prf}(\sigma)$ and α be a formula. Then $\sigma, \tau \models \alpha$ will stand for α being satisfied at τ in σ. This notion is defined inductively as follows.

- $\sigma, \tau \models tt$.
- $\sigma, \tau \models \neg\alpha$ iff $\sigma, \tau \not\models \alpha$.
- $\sigma, \tau \models \alpha \vee \beta$ iff $\sigma, \tau \models \alpha$ or $\sigma, \tau \models \beta$.
- $\sigma, \tau \models \langle a\rangle\alpha$ iff $\tau a \in \text{prf}(\sigma)$ and $\sigma, \tau a \models \alpha$.
- $\sigma, \tau \models \alpha\, U\, \beta$ iff there exists τ' such that $\tau\tau' \in \text{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Moreover for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $\sigma, \tau\tau'' \models \alpha$.

Along with the usual derived propositional connectives $\wedge, \Rightarrow$ and $\Leftrightarrow$ we will also use the propositional constant $ff = \neg tt$. Furthermore, some useful derived temporal modalities are $O\alpha = \bigvee_{a\in\Sigma}\langle a\rangle\alpha$, $\Diamond\alpha = tt\, U\, \alpha$, and $\Box\alpha = \neg\Diamond\neg\alpha$.

Formulas of LTL(Σ) can be viewed as defining set of strings over Σ. The *language defined by* α is given by $L_\alpha = \{\sigma \in \Sigma^\omega \mid \sigma, \varepsilon \models \alpha\}$. We will say that L *is definable in* LTL iff there exists some formula α of LTL(Σ) such that $L_\alpha = L$.

Example 2.3.1 A simple LTL($\{a, b\}$)-formula is $\alpha = \Diamond\Box\neg\langle a\rangle tt$. Informally, α asserts that there exists a future point in time, after which it holds continuously that the action a is *not* performed. It can by seen that $L_\alpha = (a + b)^* b^\omega$, which is also the language as accepted by the Büchi automaton on Figure 2.1.

We say that a formula $\alpha \in$ LTL(Σ) is *satisfiable* iff there exist a model $\sigma \in \Sigma^\omega$ and $\tau \in \text{prf}(\sigma)$ such that $\sigma, \tau \models \alpha$. This logic does not refer to the past either in the syntax or in the semantics. Hence the formula α is satisfiable iff there exists a model $\sigma \in \Sigma^\omega$ such that $\sigma, \varepsilon \models \alpha$. The *satisfiability problem* for LTL is to develop a decision procedure which will determine whether a given formula α is satisfiable.

2.3.2 The satisfiability problem for LTL

Both the satisfiability and model checking problems for LTL can be solved elegantly using Büchi automata. We first show how one can effectively construct for each $\alpha \in$ LTL(Σ), a Büchi automaton $\mathcal{B}_\alpha$ such that the language of ω-strings accepted by $\mathcal{B}_\alpha$ is nonempty iff α is satisfiable. This is an action-based version of the classic solution presented by Vardi and Wolper in [147] for LTL.

Through the rest of this section we fix a formula α_0. To construct $\mathcal{B}_{\alpha_0}$ we first define the (Fischer-Ladner) closure of α_0. For convenience we will assume

that the derived next-state modality O is included in the syntax of LTL(Σ). We take $cl(\alpha_0)$ to be the least set of formulas that satisfies:

- $\alpha_0 \in cl(\alpha_0)$.
- If $\neg\beta \in cl(\alpha_0)$ then $\beta \in cl(\alpha_0)$.
- If $\alpha \vee \beta \in cl(\alpha_0)$ then $\alpha, \beta \in cl(\alpha_0)$.
- If $\langle a\rangle\alpha \in cl(\alpha_0)$ then $\alpha \in cl(\alpha_0)$.
- If $O\alpha \in cl(\alpha_0)$ then $\alpha \in cl(\alpha_0)$.
- If $\alpha\, U\, \beta \in cl(\alpha_0)$ then $\alpha, \beta \in cl(\alpha_0)$. In addition, $O(\alpha\, U\, \beta) \in cl(\alpha_0)$.

Now $CL(\alpha_0)$, the *closure* of α_0, is defined to be $CL(\alpha_0) = cl(\alpha_0) \cup \{\neg\beta \mid \beta \in cl(\alpha_0)\}$. In what follows $\neg\neg\beta$ will be identified with β. For convenience, we shall often write CL instead of $CL(\alpha_0)$.

A subset $A \subseteq CL$ is called an *atom* iff it satisfies :

- $tt \in A$.
- $\beta \in A$ iff $\neg\beta \notin A$.
- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- $\alpha\, U\, \beta \in A$ iff $\beta \in A$ or $\alpha, O(\alpha\, U\, \beta) \in A$.
- If $\langle a\rangle\alpha \in A$ and $\langle b\rangle\beta \in A$ then $a = b$.

$AT(\alpha_0)$ is the set of atoms and, again, we shall often write AT instead of $AT(\alpha_0)$. Finally we set U_{α_0}, the set of *until requirements* of α_0, to be the set given by $U_{\alpha_0} = \{\alpha\, U\, \beta \mid \alpha\, U\, \beta \in CL\}$. We will often write U_0 instead of U_{α_0}.

The Büchi automaton $\mathcal{B}_{\alpha_0}$ is now defined as $\mathcal{B}_{\alpha_0} = (Q, \longrightarrow, Q_{in}, F)$, where the various components are specified as follows.

- $Q = AT \times 2^{U_0}$ is the set of states.
- The transition relation $\longrightarrow \subseteq Q \times \Sigma \times Q$ is given by $(A, x) \xrightarrow{a} (B, y)$ iff the following requirements are met:
  - For every $\langle a\rangle\alpha \in CL$, $\langle a\rangle\alpha \in A$ iff $\alpha \in B$ and for every $O(\alpha) \in CL$, $O(\alpha) \in A$ iff $\alpha \in B$.
  - if $\langle b\rangle\beta \in A$ then $b = a$.
  - if $x \neq \emptyset$ then $y = \{\alpha\, U\, \beta \mid \alpha\, U\, \beta \in x \text{ and } \beta \notin B\}$. If $x = \emptyset$ then $y = \{\alpha\, U\, \beta \mid \alpha\, U\, \beta \in B \text{ and } \beta \notin B\}$.
- $Q_{in} \subseteq Q$ is given by $(A, x) \in Q_{in}$ iff $\alpha_0 \in A$ and $x = \emptyset$.
- $F \subseteq Q$ is given by $(A, x) \in F$ iff $x = \emptyset$.

It is not hard to show that $L(\mathcal{B}_{\alpha_0}) \neq \emptyset$ iff α_0 is satisfiable. It is also easy to check that the size of $\mathcal{B}_{\alpha_0}$ is at most exponential in the size of α_0. As observed in Section 2.1.2 the emptiness problem for Büchi automata can be solved in time linear in the size of the automaton. Thus we arrive at:

Theorem 2.3.2 (Vardi, Wolper [147]) *The satisfiability problem for* LTL *is decidable in exponential time.*

It has been shown that the problem is in fact PSPACE-complete [127].

In more recent developments, Vardi [145] shows that the satisfiability problem can also be solved by means of *alternating automata*. This class of automata is an extension, not only allowing existential quantification (as nondeterministic automata), but also universal quantification and combinations of both. A run of an alternating automata is now not a sequence, but a *run tree* witnessing how these quantifications are being satisfied. In turns out that this class of automata also captures the (ω-)regular languages.

By means of alternating automata, the construction above for α_0 can be formulated very nicely, as the state space can now be taken to be only $CL(\alpha_0)$. The bookkeeping is maintained by the alternation of the automaton. However, the emptiness problem for this class of automata requires exponential time, so the running time of the decision procedure remains the same. We will however not go into these issues in more detail.

2.3.3 The model checking problem for LTL

We now formulate the model checking problem for LTL(Σ). In the present context a program is just a finite-state transition system $Pr = (S, \longrightarrow, S_{in})$ over Σ, i.e. S is a finite set of states, $\longrightarrow \subseteq S \times \Sigma \times S$ is a transition relation and $S_{in} \subseteq S$ is a set of initial states of the program. It will often be the case that the set of initial states is a singleton. Moreover, it is easy to arrange matters so that at each reachable state of the program at least one transition can be performed. We will assume that this is indeed the case for all program models we consider. Each such program Pr has the language L_{Pr} associated with it. This is just the language accepted by the Büchi automaton $\mathcal{B}_{Pr} = (S, \longrightarrow, S_{in}, S)$, so programs can be taken as Büchi automata.

Let Pr be a program and α be a formula of LTL(Σ). We say that Pr *meets the specification* α—denoted $Pr \models \alpha$—if for every computation $\sigma \in L_{Pr}$, it is the case that $\sigma, \varepsilon \models \alpha$. The *model checking problem* is to decide for a given program Pr and a given formula α whether or not $Pr \models \alpha$.

Let α_0 be a specification. Then we construct the Büchi automaton $\mathcal{B}_{\neg\alpha_0}$ corresponding to the *negation* of α_0. Now, let $\mathcal{B}$ be the Büchi automaton which accepts the intersection of the languages accepted by $\mathcal{B}_{Pr}$ and $\mathcal{B}_{\neg\alpha_0}$. It is then easy to see that $Pr \models \alpha_0$ iff $L_{Pr} \subseteq L_{\alpha_0}$ iff $L_{Pr} \cap L_{\neg\alpha_0} = \emptyset$. Thus $Pr \models \alpha_0$ iff the language accepted by $\mathcal{B}$ is *empty*.

An easy analysis of the size of $\mathcal{B}$ leads to:

Proposition 2.3.3 *The model checking problem for* LTL *is decidable in time* $O(|Pr| \cdot 2^{|\alpha_0|})$.

This solution to the model checking problem forms the conceptual basis of many verification algorithms. Several tools being employed in industry are built upon this translation from formulas to automata. To improve performance, however, a number of substantial optimizations must be incorporated.

One observation is that the state space of the product automaton needs seldomly to be fully constructed. Often the answer to the verification problem can be established by investigating only a subset of states, and this subset might be considerably smaller than the entire state space. This is the key insight underlying the so-called *on-the-fly* verification techniques [46].

The verification tool SPIN [62] is perhaps the most prominent example of an efficient implementation based on the above automata-theoretic technique incorporating the on-the-fly algorithm of [46]. While the classical technique of [46] employs the state-based approach of labellings by atomic propositions, the method extends smoothly to action-based versions of linear time temporal logics [30], in particular the present LTL(Σ).

2.4 Expressiveness of Logics

Since its inception, LTL has gained a substantial amount of attention due to its practical applicability for verification purposes. Formulas of LTL can be viewed as defining system *properties* in terms of sets of computation sequences. More precisely, the property α is identified with the language L_α of ω-strings satisfying α.

We first bring out that LTL is equal in expressive power to the first-order theory of strings. We consider fragments of both logics and mention how the tight correspondence persists. Then we survey how a temporal logic equivalent to MSO might arise.

2.4.1 LTL is equivalent to FO

An interesting aspect of the study of temporal logics is to characterize and compare the classes of properties definable in such logics. It turns out that the seminal, but perhaps surprising, result is that LTL is *expressively complete* with respect to first-order logic. This result is commonly referred to as *Kamp's Theorem*.

Theorem 2.4.1 (Kamp [70], Gabbay, Pnueli, Shelah, Stavi [41]) *Let* $L \subseteq \Sigma^\omega$. *Then* L *is definable in* LTL(Σ) *if and only if* L *is definable in* FO(Σ).

Traditionally, LTL was introduced with each temporal operator having an additional (symmetric) *past* counterpart. More precisely, the syntax included operators such as "previously α" and "α since β", dual to $\langle a \rangle \alpha$ and $\alpha \, U \, \beta$,

respectively. Kamp showed in his thesis [70] that first-order logic and the past-augmented LTL are expressively equivalent. There is an obvious linear translation from (past-augmented) LTL to FO, but the reverse translation is necessarily quite involved as it must battle the nonelementary blow-up in formula size made unavoidable by the PSPACE-completeness of LTL and nonelementary hardness of FO.

Gabbay, Pnueli, Shelah, and Stavi [41] later strengthened Kamp's Theorem by showing that the past-augmented version of LTL is expressively equivalent to the present formulation of LTL.

The first-order definable languages are broadly accepted as being *the* interesting class of properties of reactive systems, because it corresponds to the "non-counting" fragment of the regular languages. Temporal logic is usually intended for expressing the relative ordering between events and although counting abilities are of interest in certain applications, the "purely temporal" properties are those of FO. Hence LTL in an intuitive fashion exactly captures a very natural class of properties without suffering from the nonelementary complexity of first-order logic, but possibly with a large blow-up in the formula sizes.

2.4.2 Variable-confined subsets of FO

An important corollary of Kamp's Theorem is that not only is first-order logic expressively equivalent to LTL, but a bounded number of variables suffice to express all first-order properties over strings. In particular, letting FO^3 denote first-order logic with only three different variables, Kamp [70] and Immermann and Kozen [66] showed that FO^3 is expressively equivalent to FO. The result of Stockmeyer [131] however shows that the satisfiability problem remains nonelementary hard for FO^3.

It turns out that the tight correspondence between linear time temporal logic and first-order logic persists even when considering variable-confined fragments. More precisely, Etessami, Vardi, and Wilke [36] demonstrated that first-order logic with two variables also captures a natural class of temporal properties. More precisely, FO^2 is expressively equivalent to unary-LTL. Here unary-LTL is past-augmented LTL with the until (and since) modalities replaced by the unary modality $\Diamond$ (and its past counterpart).

Thérien and Wilke provide another beautiful characterization of first-order logic with two variables [133]. By applying Ehrenfeucht-Fraïssé games in the setting of semigroups they show that a property for *finite* strings is definable in FO^2 if and only if it is definable by a (general) first-order formula with just one quantifier alternation. Equivalently, FO^2 expresses exactly those properties that can be defined by a formula of the form $(\exists x_1 \ldots \exists x_m \forall y_1 \ldots \forall y_n)(\varphi)$ and also of the form $(\forall x_1 \ldots \forall x_r \exists y_1 \ldots \exists y_s)(\psi)$, where φ and ψ are quantifier-free formulas of first-order logic. Unfortunately, the result cannot be extended to infinite strings as $(a+b)^*(ab^*)^\omega$ is definable in FO^2 but *not* by an existential formula with one quantifier alternation.

2.4.3 Extensions to MSO

It was observed by Wolper [155] that properties like "a is performed at every even position" are not definable in LTL (and hence FO). With both this fact and the complexity issues in mind, one might set out to find natural extensions of linear time temporal logics expressively equivalent to *full* monadic second-order logic.

One route towards achieving this goal consists of permitting quantification over atomic propositions. The resulting logic called QPTL [126] has the expressive power of MSO but has a decision procedure of nonelementary complexity. The second route consists of augmenting LTL with the so-called automaton connectives of the form $\mathcal{A}(\alpha_1, \ldots, \alpha_n)$. Indeed these infinitely many additional modalities are so powerful that the next and until modalities become derived ones. The resulting logic called *Extended Temporal Logic (ETL)* [156] is equal in expressive power to MSO while admitting an exponential-time decision procedure. Intuitively, accepting runs of the automaton $\mathcal{A}$ operating over the alphabet $\{a_1, a_2, \ldots, a_n\}$ express ways that the modality $\mathcal{A}(\alpha_1, \ldots, \alpha_n)$ can be fulfilled.

The key drawback of ETL(Σ), as we see it, lies in its lack of structuring principles for forming compound formulas. Stated differently, the only mechanism that ETL(Σ) has—apart from the boolean connectives—to form compound formulas is by *nesting* the automaton formulas. Thus a typical compound formula would look like $\mathcal{A}^1(\phi_1^1, \mathcal{A}^2(\phi_1^2, \phi_2^2, \mathcal{A}^3(\phi_1^3, \ldots, \phi_n^3), \phi_4^2, \ldots, \phi_m^2), \phi_3^1, \ldots, \phi_n^1)$. Moreover, ETL, as formulated in [148] has an uncontrolled amount of "external" elements in the sense that the states and the alphabets of the automata which are used to write down the automaton formulas have little to do with the logic under consideration.

Yet another possibility is to extend LTL(Σ) *orthogonally* by indexing the until operator with the regular expressions (or *programs*) of the branching time logic Propositional Dynamic Logic (PDL) [39, 51]. One then arrives at *Dynamic Linear Time Temporal Logic*, denoted DLTL(Σ), where $\alpha\, U\, \beta$ is replaced by the strengthened $\alpha\, U^\pi \beta$ modality.

To satisfy $\alpha\, U^\pi \beta$, one must satisfy $\alpha\, U\, \beta$ along some finite stretch of behaviour which is in the (linear time) behaviour of the regular expression π. Recalling from Section 2.1.1 that there is an obvious way to associate a set of finite strings with every regular expression via a map $||\cdot|| : \mathrm{RE}(\Sigma) \to 2^{\Sigma^*}$, the semantics of the new modality is given by:

- $\sigma, \tau \models \alpha\, U^\pi \beta$ iff there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Moreover, for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $\sigma, \tau\tau'' \models \alpha$.

Note that by replacing the $\alpha\, U^\pi \beta$ modality with the derived operator $\langle\pi\rangle\alpha \stackrel{\mathrm{def}}{=} ttU^\pi\alpha$ one obtains a sublogic which is essentially PDL equipped with a linear time semantics.

The details of the approach are given in Chapter 6. It turns out that DLTL (and also linear time PDL) are indeed expressively equivalent to MSO. As an additional fact, the expressive power of FO can be recovered by restriction to

the fragment of DLTL with *star-free* regular expressions. Moreover, DLTL has an exponential-time decision procedure generalizing the classical decision procedure by Büchi automata sketched in Section 2.3. Finally, it turns out that the axiomatization of PDL [76] can be nicely extended to a finitary axiomatization of DLTL.

DLTL is, in spirit, inspired by ETL and in fact it may appear to be at first sight just a reformulation of ETL with some cosmetic changes. This however has to do with the instinctive identification one makes between finite state automata and regular expressions. In fact DLTL is quite different in terms of the mechanisms it offers for structuring formulas and it seems more transparent and easier to work with. Our approach also leads to smooth generalizations in nonsequential settings of which Chapter 7 is an example.

We conclude this chapter by pointing to another expressive temporal logic known as the *(linear time) μ-calculus* [130]. The new feature is the introduction of fixed points operators. More precisely, the formulas are given as:

$$\Phi_\mu(\Sigma) ::= tt \mid X \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a\rangle\alpha \mid \mu X.\alpha \ , \quad a \in \Sigma.$$

Here X ranges over the set of variables, and all free occurrences of X within formulas are required to lie within the scope of an even number of negations.

We will not go into the semantics in detail, but just note for illustration that the until operator can be defined in this logic via the minimal fixed point operator as follows; $\alpha\, U\, \beta = \mu X.\beta \vee (\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle X)$. In fact, it is this "fixed point characterization" of the until operator which is being exploited in the construction of the automaton $\mathcal{B}_{\alpha_0}$ in Section 2.3.2.

One can prove that the μ-calculus is expressively equivalent to monadic second-order logic over strings.

Chapter 3

Mazurkiewicz Traces

In Chapter 2 we saw how a rich theory is available for deciding whether a finite-state system meets its specification. The systems at hand are almost always distributed in nature, and for such systems the computations will constitute interleavings of the occurrences of causally independent actions. Consequently, the computations can be naturally grouped together into equivalence classes where two computations are equated in case they are two different interleavings of the same partially ordered stretch of behaviour. It turns out that many of the properties expressed as LTL-formulas happen to have the so-called "all-or-none" flavour. Either all members of an equivalence class of computations have the desired property or none do. ("Leads to deadlock" is one such property.) For verifying such properties one has to check the property for just one member of each equivalence class. This is the insight underlying many of the partial-order based verification methods [48, 108, 144]. As may be guessed, the importance of these methods lies in the fact that via these methods the computational resources required for the verification task can often be dramatically reduced.

It is often the case that the equivalence classes of computations generated by a distributed system constitute objects called Mazurkiewicz traces. They can be canonically represented as restricted labelled partial orders. This opens up an alternative way of exploiting the nonsequential nature of the computations of a distributed systems and the attendant partial-order based methods. It consists of developing linear time temporal logics that can be directly interpreted over Mazurkiewicz traces. In these logics, every specification is guaranteed to have the "all-or-none" property and hence can take advantage of the partial-order based reduction methods during the verification process. The study of these logics also exposes the richness of the partial-order settings from a logical standpoint and the complications that can arise as a consequence.

In this chapter, we will show how the theory of the previous chapters has been extended to the richer noninterleaving setting of Mazurkiewicz traces. In the next section we introduce the basic objects of study, and present two equivalent representations of traces. In Section 3.2 we extend the notion of regularity to trace languages and define a corresponding class of distributed finite-state

acceptors known as asynchronous automata. We then carry over logical definability of monadic second-order logic to the setting of traces and sketch how it, as before, coincides with the notion of regularity. In Section 3.4 we survey the various extensions of LTL to the partial-order setting and sketch how their automata-theoretic decision procedures generalize the classical approach of Chapter 2. We then identify a subclass of trace languages with some nice properties before concluding the chapter in Section 3.6 by considering the relative expressive powers of the logics put forth in this chapter.

This material provides the technical and conceptual background for our contributions in Chapters 7–9, of which Chapter 8 contains a survey. Furthermore, the concepts and results of the present chapter serve as guiding principles for the developments in Chapters 10 and 11. Certain constructions to be encountered in those chapters will rely crucially upon the present developments.

3.1 Mazurkiewicz Traces

We first introduce the notion of traces from the standpoint of sequences. This will enable us to define the notion of a trace consistent property. This notion plays an important role in partial order based reducion methods. As pointed out above, it also provides the motivation for studying trace-based linear time temporal logics. Then we show how traces can be viewed as restricted labelled partial orders. For both representations, it is possible to give a uniform definition of both finite and infinite traces.

3.1.1 Traces as sets of sequences

A *(Mazurkiewicz) trace alphabet* is a pair (Σ, I), where Σ, the alphabet, is a finite set and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric *independence relation*. In most applications, Σ consists of the actions performed by a distributed system while I captures a static notion of causal independence between actions. The idea is that contiguous independent actions occur with no causal order between them. Thus, every sequence of actions from Σ corresponds to an interleaved observation of a partially ordered stretch of system behaviour. This leads to a natural equivalence relation over execution sequences: two sequences are equated if and only if they correspond to different interleavings of the same partially ordered stretch of behaviour.

For the rest of the section we fix a trace alphabet (Σ, I) and define $D = (\Sigma \times \Sigma) - I$ to be the *dependency relation*. Note that D is reflexive and symmetric. A set $p \subseteq \Sigma$ is called a *D-clique* iff $p \times p \subseteq D$. The equivalence relation $\approx_I \subseteq \Sigma^\infty \times \Sigma^\infty$ induced by I is given by: $\sigma \approx_I \sigma'$ iff $\sigma \restriction p = \sigma' \restriction p$ for every D-clique p. Here and elsewhere, if Σ is a finite set, $\sigma \in \Sigma^\infty$ and $\Sigma' \subseteq \Sigma$ then $\sigma \restriction \Sigma'$ is the sequence obtained by erasing from σ all occurrences of letters in $\Sigma - \Sigma'$.

Clearly $\approx_I$ is an equivalence relation. Notice that if $\sigma = \tau ab \sigma_1$ and $\sigma' = \tau ba \sigma_1$ with $(a, b) \in I$ then $\sigma \approx_I \sigma'$. Thus σ and σ' are identified if they differ

only in the order of appearance of a pair of adjacent independent actions. In fact, for finite strings, an alternative way to characterize $\approx_I$ is to say that $\sigma \approx_I \sigma'$ iff σ' can be obtained from σ by a finite sequence of permutations of adjacent independent actions. However, the definition of $\approx_I$ in terms of permutations can not be directly transported to infinite strings, which is why we work with the less intuitive definition presented here.

The equivalence classes generated by $\approx_I$ are called *(Mazurkiewicz) traces*. Throughout this chapter, we will use $[\sigma]$ to denote the $\approx_I$-equivalence class containing σ. A set of traces is called a *trace language*. The theory of traces and trace languages is well developed and documented (see e.g. [24, 27] for basic material as well as a substantial number of references to related work).

Example 3.1.1 As an example consider the trace alphabet (Σ, I) where $\Sigma = \{a, b, d\}$ and $I = \{(a, b), (b, a)\}$. One trace over (Σ, I) is $[abdabd] = \{abdabd, abdbad, badabd, badbad\}$.

A variety of models of distributed systems naturally have a trace alphabet associated with them [154]. It also turns out that many interesting properties of distributed systems respect the equivalence relation induced by these trace alphabets. This has important consequences for the practical verification of such properties.

The key notion in this context is that of a trace consistent property. To bring this out, we start with a trace alphabet (Σ, I) and let $L \subseteq \Sigma^\infty$. We say that L is *trace consistent* in case $\sigma \in L$ and $\sigma \approx_I \sigma'$ implies $\sigma' \in L$ for every $\sigma, \sigma' \in \Sigma^\infty$. In other words, either *all* members of a trace are in L or *none* of them are. We say that the formula α of LTL(Σ) is trace consistent in case L_α is trace consistent. It is not hard to see that there is a one-to-one correspondence between trace languages and trace consistent languages of strings.

Now suppose Pr is a program over Σ as defined in Section 2.3.3 which has a trace alphabet (Σ, I) associated with it in some natural manner. Suppose further that L_{Pr}, the linear time behaviour of Pr, is trace consistent (we will see natural models of distributed programs that possess these features in the material to follow). Now consider a specification α which happens to be trace consistent. Then, as described in Section 2.3.3, verifying $Pr \models \alpha$ boils down to verifying $L_{Pr} \subseteq L_\alpha$. Instead of checking $L_{Pr} \subseteq L_\alpha$ we can choose to check $L' \subseteq L_\alpha$ where L' is designed to be such that $L' \subseteq L_{Pr}$ and for every $\sigma \in L_{Pr}$, $[\sigma] \cap L' \neq \emptyset$. The key point is, the finite representation of L' can often be substantially smaller than the representation of Pr. This is the insight underlying many of the so-called partial-order methods deployed in the model checking world [48, 108, 144].

3.1.2 Traces as labelled partial orders

Traces have many equivalent representations. Here we shall view them as restricted Σ-labelled partial orders. Abusing terminology we shall also call these objects *traces*. We will later argue that these objects are in a rather precise sense the same as the objects called traces defined in Section 3.1.1 in terms of equivalence classes of sequences.

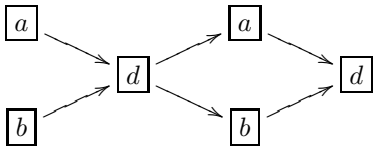


Figure 3.1: Trace over (Σ, I) where $\Sigma = \{a, b, d\}$ and $I = \{(a, b), (b, a)\}$.

Let T be a Σ-labelled poset. In other words, $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $Y \subseteq E$ we define $\downarrow Y = \{x \mid \exists y \in Y : \ x \leq y\}$ and $\uparrow Y = \{x \mid \exists y \in Y : \ y \leq x\}$. In case $Y = \{y\}$ is a singleton we shall write $\downarrow y$ ($\uparrow y$) instead of $\downarrow\{y\}$ ($\uparrow\{y\}$). We also let $\lessdot$ be the *covering relation*; $x \lessdot y$ iff $x < y$ and for all $z \in E$, $x \leq z \leq y$ implies $x = z$ or $z = y$. Moreover, we let the *concurrency relation* be defined as $x \ co \ y$ iff $x \not\leq y$ and $y \not\leq x$.

A *(Mazurkiewicz) trace (over* (Σ, I)*)* is a Σ-labelled poset $T = (E, \leq, \lambda)$ satisfying:

- $\downarrow e$ is a finite set for each $e \in E$.
- For every $e, e' \in E$, $e \lessdot e'$ implies $\lambda(e) \ D \ \lambda(e')$.
- For every $e, e' \in E$, $\lambda(e) \ D \ \lambda(e')$ implies $e \leq e'$ or $e' \leq e$.

We shall refer to members of E as *events*. The trace $T = (E, \leq, \lambda)$ is said to be *finite* if E is a finite set. Otherwise it is an *infinite* trace. Note that E is always a countable set. T is said to be *nonempty* in case $E \neq \emptyset$. An example of a finite trace with six events is depicted in Figure 3.1. We let $\mathbb{TR}^*(\Sigma, I)$ be the set of finite traces and $\mathbb{TR}^\omega(\Sigma, I)$ be the set of infinite traces over (Σ, I) and set $\mathbb{TR}(\Sigma, I) = \mathbb{TR}^*(\Sigma, I) \cup \mathbb{TR}^\omega(\Sigma, I)$. Often we will write $\mathbb{TR}$ instead of $\mathbb{TR}(\Sigma, I)$ etc. As before, a subset of traces $\mathcal{L} \subseteq \mathbb{TR}$ will be called a *trace language*.

Let $T = (E, \leq, \lambda)$ be a trace. The finite prefixes of T, to be called configurations, will play a crucial role in what follows. A *configuration* of T is a finite subset $c \subseteq E$ such that $c = \downarrow c$. We let $\mathcal{C}_T$ be the set of configurations of T and let c, c', c'' range over $\mathcal{C}_T$. Note that $\emptyset$ is always a configuration and $\downarrow e$ is a configuration for every $e \in E$. Finally, the transition relation $\longrightarrow_T \subseteq \mathcal{C}_T \times \Sigma \times \mathcal{C}_T$ is given by: $c \xrightarrow{a}_T c'$ iff there exists $e \in E$ such that $\lambda(e) = a$ and $e \notin c$ and $c' = c \cup \{e\}$. It is easy to see that if $c \xrightarrow{a}_T c'$ and $c \xrightarrow{a}_T c''$ then $c' = c''$.

Note that we have now introduced two different notions of traces; one in terms of equivalence classes of strings in Section 3.1.1 and the other in terms of Σ-labelled partial orders in this section. We now sketch briefly the constructions that show that $\Sigma^\infty/\approx_I$ and $\mathbb{TR}(\Sigma, I)$ represent the same class of objects. We shall construct representation maps $\mathsf{st} : \Sigma^\infty/\approx_I \to \mathbb{TR}(\Sigma, I)$ and $\mathsf{ts} : \mathbb{TR}(\Sigma, I) \to \Sigma^\infty/\approx_I$ and state some results which show that these maps are "inverses" of each other.

Henceforth, we will not distinguish between isomorphic elements in $\mathbb{TR}(\Sigma, I)$. In other words, whenever we write $T = T'$ for traces $T = (E, \leq, \lambda)$ and $T' =$

$(E', \leq', \lambda')$, we mean that there is a label-preserving isomorphism between T and T'.

Recall that for $\sigma \in \Sigma^\infty$, $[\sigma]$ stands for the $\approx_I$-equivalence class containing σ. We now define $\mathsf{st} : \Sigma^\infty \to \mathbb{TR}(\Sigma, I)$. Let $\sigma \in \Sigma^\infty$. Then $\mathsf{st}(\sigma) = (E, \leq, \lambda)$ where:

- $E = \{\tau a \mid \tau a \in \mathrm{prf}(\sigma)\}$. (Thus $E = \mathrm{prf}(\sigma) - \{\varepsilon\}$.)
- $\leq \subseteq E \times E$ is the least partial order which satisfies: For all $\tau a, \tau' b \in E$, if $\tau a \preceq \tau' b$ and $(a, b) \in D$ then $\tau a \leq \tau' b$.
- For $\tau a \in E$, $\lambda(\tau a) = a$.

The map st induces a natural map st' from $\Sigma^\infty/\approx_I$ to $\mathbb{TR}(\Sigma, I)$ defined by $\mathsf{st}'([\sigma]) = \mathsf{st}(\sigma)$. One can show that if $\sigma, \sigma' \in \Sigma^\infty$, then $\sigma \approx_I \sigma'$ iff $\mathsf{st}(\sigma) = \mathsf{st}(\sigma')$. This observation guarantees that st' is well-defined. In fact, henceforth we shall write st to denote both st and st'.

Next, let $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$. Then $\sigma \in \Sigma^\infty$ is a *linearization* of T iff there exists a map $\rho : \mathrm{prf}(\sigma) \to \mathcal{C}_T$, such that the following conditions are met:

- $\rho(\varepsilon) = \emptyset$.
- $\forall \tau a \in \mathrm{prf}(\sigma)$ with $\tau \in \Sigma^*$, $\rho(\tau) \xrightarrow{a}_T \rho(\tau a)$.
- $\forall e \in E\ \exists \tau \in \mathrm{prf}(\sigma),\ e \in \rho(\tau)$.

The function ρ will be called a *run map* of the linearization σ. Note that the run map of a linearization is unique. In what follows, we shall let $lin(T)$ to be the set of linearizations of the trace T.

We can now define the map $\mathsf{ts} : \mathbb{TR}(\Sigma, I) \to \Sigma^\infty/\approx_I$ as: $\mathsf{ts}(T) = lin(T)$ and show that ts is well-defined. We can also show that for every $\sigma \in \Sigma^\infty$, $\mathsf{ts}(\mathsf{st}(\sigma)) = [\sigma]$ and for every $T \in \mathbb{TR}(\Sigma, I)$, $\mathsf{st}(\mathsf{ts}(T)) = T$. This justifies our claim that $\Sigma^\infty/\approx_I$ and $\mathbb{TR}(\Sigma, I)$ are indeed two equivalent ways of talking about the same class of objects. By inspection one easily verifies that the traces of Example 3.1.1 and Figure 3.1 are two different represention of the same trace.

We shall use both representations interchangeably, but as a rule of thumb we will use L to denote trace consistent languages of strings and $\mathcal{L}$ to denote collections of labelled partial orders.

3.2 Automata over Traces

Regular collections of traces and corresponding finite-state automata play an important role as they constitute the basis for extending the model checking techniques of the previous chapter to the noninterleaving setting of traces. As we will see in this section, it is possible to exhibit generalizations of finite automata to the richer domain of traces guided by the interleaving setting of Chapter 2. In this way one can develop elegant and efficient decision procedures for

satisfiability and model checking problems of most linear time temporal logics for traces.

As a starting point for regularity, we note that every trace consistent subset L of Σ^∞ defines a trace language $\mathcal{L}$ given by $\mathcal{L} = \{\mathsf{st}(\sigma) \mid \sigma \in L\}$ which has the property $\mathsf{ts}(L_{Tr}) = L$. Consequently, we say that a trace language $\mathcal{L}$ is *regular* iff $\mathsf{ts}(\mathcal{L})$ is a regular subset of Σ^∞ in the sense of Chapter 2.

In order to recognize regular trace languages one will have to use generalized versions of finite-state automata. These process their inputs in a distributed fashion according to a distribution of actions, which we will introduce first. Following this, we introduce the class of automata called *asynchronous automata* as formulated by Zielonka [157] for recognizing regular languages of finite traces, and then subsequently define their generalization to infinite traces due to Gastin and Petit [43]. We present a specific asynchronous automaton called the gossip automaton which plays an important role in some of the later developments. We then conclude the chapter by mentioning the extensions of Kleene's Theorem to the setting of traces.

3.2.1 Distributed alphabets

The asynchronous automata will process the actions of Σ in a distributed fashion according to some spatial distribution of processes. To bring this out, we say that a *distributed alphabet* is a family $\{\Sigma_p\}_{p\in\mathcal{P}}$ where $\mathcal{P}$ is a finite nonempty set of agents (also referred to as processes in the sequel) and Σ_p is a finite nonempty alphabet for each $p \in \mathcal{P}$. The idea is that whenever an action from Σ_p occurs, the agent p must participate in it. Hence the agents can constrain each other's behaviour, both directly and indirectly.

Trace alphabets and distributed alphabets are closely related to each other. Let $\widetilde{\Sigma} = \{\Sigma_p\}_{p\in\mathcal{P}}$ be a distributed alphabet. Then $\Sigma_{\mathcal{P}}$, the global alphabet associated with $\widetilde{\Sigma}$, is the collection $\bigcup_{p\in\mathcal{P}} \Sigma_p$. The distribution of $\Sigma_{\mathcal{P}}$ over $\mathcal{P}$ can be described using a *location function* $\mathrm{loc}_{\widetilde{\Sigma}} : \Sigma_{\mathcal{P}} \to 2^{\mathcal{P}}$ defined as follows: $\mathrm{loc}_{\widetilde{\Sigma}}(a) = \{p \mid a \in \Sigma_p\}$. This in turn induces the relation $I_{\widetilde{\Sigma}} \subseteq \Sigma_{\mathcal{P}} \times \Sigma_{\mathcal{P}}$ given by: $(a, b) \in I_{\widetilde{\Sigma}}$ iff $\mathrm{loc}_{\widetilde{\Sigma}}(a) \cap \mathrm{loc}_{\widetilde{\Sigma}}(b) = \emptyset$. Clearly $I_{\widetilde{\Sigma}}$ is irreflexive and symmetric and hence $(\Sigma_{\mathcal{P}}, I_{\widetilde{\Sigma}})$ is a trace alphabet. Thus every distributed alphabet canonically induces a trace alphabet. Two actions are independent according to $\widetilde{\Sigma}$ if they are executed by disjoint sets of processes. Henceforth, we write loc for $\mathrm{loc}_{\widetilde{\Sigma}}$ whenever $\widetilde{\Sigma}$ is clear from the context.

Going in the other direction there are, in general, many different ways to implement a trace alphabet as a distributed alphabet. A standard approach is to create a separate agent for each maximal D-clique generated by (Σ, I). Recall that a D-clique of (Σ, I) is a nonempty subset $p \subseteq \Sigma$ such that $p \times p \subseteq D$. Let $\mathcal{P}$ be the set of maximal D-cliques of (Σ, I). This set of processes induces the distributed alphabet $\widetilde{\Sigma} = \{\Sigma_p\}_{p\in\mathcal{P}}$ where $\Sigma_p = p$ for every process p. The alphabet $\widetilde{\Sigma}$ implements (Σ, I) in the sense that the canonical trace alphabet induced by it is exactly (Σ, I). In other words, $\Sigma_{\mathcal{P}} = \Sigma$ and $I_{\widetilde{\Sigma}} = I$.

Example 3.2.1 Consider again the trace alphabet of Example 3.1.1 and Figure 3.1, where $\Sigma = \{a, b, d\}$ and $I = \{(a, b), (b, a)\}$. The canonical D-clique implementation of (Σ, I) yields the distributed alphabet $\widetilde{\Sigma} = \{\{a, d\}, \{d, b\}\}$.

3.2.2 Asynchronous automata

Through the rest of the section we fix a distributed alphabet $\{\Sigma_p\}_{p\in\mathcal{P}}$ and set $\Sigma = \Sigma_{\mathcal{P}}$. It will be convenient to assume that $\mathcal{P} = \{1, 2, \ldots, K\}$. Further, the ith component of a K-tuple $x = (x_1, x_2, \ldots, x_K)$ will be written as $x[i]$. In other words, $x[i] = x_i$.

Let $\widetilde{\Sigma}$ be a distributed alphabet with $\mathcal{P}$ as the associated set of agents. In an asynchronous automaton, each process $p \in \mathcal{P}$ is equipped with a finite nonempty set of local p-states, denoted S_p, which we will assume to be pairwise disjoint. It will be convenient to develop some notations for talking about "more global" states before defining these automata.

First we set $S = \bigcup_{p\in\mathcal{P}} S_p$ and call S the set of *local states*. We let P, Q range over nonempty subsets of $\mathcal{P}$ and let p, q range over $\mathcal{P}$. A Q-state is a map $s : Q \to S$ such that $s(q) \in S_q$ for every $q \in Q$. We let S_Q denote the set Q-states. We call $S_{\mathcal{P}}$ the set of *global states*. We use a to abbreviate $\mathrm{loc}(a)$ when talking about states (recall that $\mathrm{loc}(a) = \{\ p \mid a \in \Sigma_p\ \}$). Thus an a-state is just a $\mathrm{loc}(a)$-state and S_a denotes the set of all $\mathrm{loc}(a)$-states. Henceforth, when dealing with a $\mathcal{P}$-indexed family of sets $\{S_p\}_{p\in\mathcal{P}}$ we will often just write $\{S_p\}$. A similar remark applies for Σ-indexed sets as e.g. $\{\longrightarrow_a\}_{a\in\Sigma}$.

A *distributed transition system* TS over $\widetilde{\Sigma}$ is a structure $(\{S_p\}, \{\longrightarrow_a\}, S_{in})$, where

- S_p is a finite nonempty set of p-states for each process p.
- For $a \in \Sigma$, $\longrightarrow_a \subseteq S_a \times S_a$ is a transition relation between a-states.
- $S_{in} \subseteq S_{\mathcal{P}}$ is a set of initial global states.

The idea is that an a-move by TS involves only the local states of the agents which participate in the execution of a. This is reflected in the global transition relation $\longrightarrow_{TS} \subseteq S_{\mathcal{P}} \times \Sigma \times S_{\mathcal{P}}$ which is defined in the following. Suppose s and s' are two global states and s_a and s'_a are the two corresponding a-states. In other words, $s_a(i) = s(i)$ and $s'_a(i) = s'(i)$ for each i in $\mathrm{loc}(a)$. Then

$$s \xrightarrow{a}_{TS} s' \text{ iff } (s_a, s'_a) \in \longrightarrow_a \text{ and } s(j) = s'(j) \text{ for every } j \notin \mathrm{loc}(a).$$

From the definition of $\longrightarrow_{TS}$, it is clear that actions which are executed by disjoint sets of agents are processed independently by TS.

An *asynchronous automaton* over $\widetilde{\Sigma}$ is then a distributed transition system equipped with a set of global accepting states. More precisely, it is a structure $\mathcal{Z} = (\{S_p\}, \{\longrightarrow_a\}, S_{in}, F)$ where

- $(\{S_p\}, \{\longrightarrow_a\}, S_{in})$ is a distributed transition system.

- $F \subseteq S_{\mathcal{P}}$ is a set of global accepting states.

A *trace run* of $\mathcal{Z}$ over the finite trace $T = (E, \leq, \lambda)$ is a map $\rho : \mathcal{C}_T \to S_{\mathcal{P}}$ such that $\rho(\emptyset) \in S_{in}$ and for every $(c, a, c') \in \longrightarrow_T$, $\rho(c) \xrightarrow{a}_{TS} \rho(c')$. We say that ρ is an *accepting run* whenever $\rho(E) \in F$. The *language of finite traces accepted by* $\mathcal{Z}$ is given by $\mathcal{L}_{Tr}(\mathcal{Z}) = \{\ T \in \mathbb{TR}^* \mid \exists$ an accepting run of $\mathcal{Z}$ over $T\ \}$. We will call the class of languages accepted by some asynchronous automaton the *recognizable trace languages*. Zielonka's fundamental result that asynchronous automata recognize exactly the regular trace languages can now be formulated as

Theorem 3.2.2 (Zielonka [157]) *Let $\mathcal{L} \subseteq \mathbb{TR}^*(\Sigma, I)$. Then $\mathcal{L}$ is regular if and only if $\mathcal{L} = \mathcal{L}_{Tr}(\mathcal{Z})$ for some asynchronous automaton $\mathcal{Z}$ over some $\widetilde{\Sigma}$ where $\widetilde{\Sigma}$ is a distributed alphabet whose induced trace alphabet is (Σ, I).*

Further, one may assume $\mathcal{Z}$ to be deterministic and one may assume $\widetilde{\Sigma}$ to be the distributed alphabet induced by the maximal D-cliques of (Σ, I).

3.2.3 Asynchronous automata over infinite traces

Zielonka's Theorem has been generalized to the set of ω-regular trace languages by Gastin and Petit [43] in terms of asynchronous automata with Büchi acceptance conditions. In bringing this out, we use two types of acceptance conditions for the components in order to be able to handle both finite and infinite behaviours. Even if one is interested only in global infinite behaviours, finite behaviours at the component level must be treated; a component might quit after engaging in a finite number of actions while some components run forever.

Hence our automata are distributed transition systems equipped with *both* finite and infinite accepting states. Gastin and Petit [43] use slightly different acceptance conditions but they can be easily shown equivalent to the present formulation.

An *asynchronous Büchi automaton* over $\widetilde{\Sigma}$ is a structure $\mathcal{Z} = (\{S_p\}, \{\longrightarrow_a\}, S_{in}, \{(F_p, F_p^\omega)\})$, where:

- $(\{S_p\}, \{\longrightarrow_a\}, S_{in})$ is a distributed transition system.
- $F_p \subseteq S_p$ is a set of local finitary accepting states of process p.
- $F_p^\omega \subseteq S_p$ is a set of local infinitary accepting states of process p.

For convenience we will from now on denote this class of automata just "asynchronous[1] automata". We will also use $\mathcal{Z}$ to denote such automata.

To define acceptance we must now compute $\inf_p(\rho)$, the set of p-states that are encountered infinitely often along ρ. When incorporating both finite and

[1]The term "asynchronous" is perhaps slightly misleading in this context, because these automata process input in a distributed fashion by *synchronizing* on common actions. Consequently, a better nomenclature might have been "synchronous automata", but we adhere to the tradition here.

infinite behaviour in this richer domain we have to take care in defining the set of infinitely occuring states of process p. The obvious definition, namely $\inf_p(\rho) = \{s_p \mid \rho(c)(p) = s_p \text{ for infinitely many } c \in \mathcal{C}_T\}$, will not work. The complication arises because some processes may make only finitely many moves, even though the overall trace consists of an infinite number of events. So, we have to define $\inf_p(\rho)$ so as to detect whether or not process p is making progress.

To bring this out, consider a trace $T \in \mathbb{TR}$ with $T = (E, \leq, \lambda)$ and some $p \in \mathcal{P}$. Letting $E_p = \{e \mid e \in E \text{ and } \lambda(e) \in \Sigma_p\}$, we can define the appropriate formulation of acceptance as follows:

- E_p *is finite:* $\inf_p(\rho) = \{s_p\}$, where $\rho(\downarrow E_p) = s$ and $s_p = s(p)$.
- E_p *is an infinite set:* $\inf_p(\rho) = \{s_p \mid \text{for infinitely many } e \in E_p, s_e(p) = s_p, \text{ where } \rho(\downarrow e) = s_e\}$.

A *trace run* of an asynchronous automaton over the (possibly infinite) trace $T = (E, \leq, \lambda) \in \mathbb{TR}$ is now defined in the obvious way. A run ρ of $\mathcal{Z}$ over the (possibly infinite) trace $T = (E, \leq, \lambda)$ is *accepting* iff for each process p the following conditions are met:

- If E_p is *finite* then $\inf_p(\rho) \cap F_p \neq \emptyset$.
- If E_p is *infinite* then $\inf_p(\rho) \cap F_p^\omega \neq \emptyset$.

We then have the following characterization extending Theorem 3.2.2.

Theorem 3.2.3 (Gastin, Petit [43]) *Let $\mathcal{L} \subseteq \mathbb{TR}(\Sigma, I)$ be a trace language. Then $\mathcal{L}$ is regular if and only if $\mathcal{L} = \mathcal{L}_{Tr}(\mathcal{Z})$ for an asynchronous automaton over $\widetilde{\Sigma}$ where $\widetilde{\Sigma}$ is a distributed alphabet whose induced trace alphabet is (Σ, I).*

It should be noted however that deterministic automata no longer suffice for accepting *all* regular languages.

The emptiness problem for this class of asynchronous automata is decidable in time $O(n^{2|\mathcal{P}|})$, where n is the largest of the local state spaces, S_p [97].

We have described here the languages defined by asynchronous automata in terms of traces. We note that these automata can be viewed as automata running over Σ-sequences. Using the global transition relations of these automata one can easily define the string languages accepted by these automata. These languages will be naturally trace consistent with respect to the trace alphabets induced by the associated distributed alphabets. The resulting trace languages will be precisely the trace languages accepted by these automata according to the definitions we have provided here.

3.2.4 The gossip automaton

Zielonka's proof [157] of Theorem 3.2.2 is nontrivial, intricate, and difficult. The reason is that, in the trace setting, one must keep track of the latest information that the agents have about each other when defining the transitions of the asynchronous automaton accepting a given regular trace language.

Mukund and Sohoni [96] define and isolate this phenomenon and show that such "gossip" information can be kept track of by a deterministic asynchronous automaton whose size depends only on $\widetilde{\Sigma}$. With this "gossip automaton" as a constituent subpart, Mukund and Sohoni go on to give—in our opinion—a more structured account [95] of Zielonka's Theorem. Gossip is also present in the original proof by Zielonka [157], but not explicitly. We will also use the gossip automaton as a constituent part of the automata-theoretic decision procedure for TrPTL in Chapter 3.4.1. Furthermore, gossiping will play an important role of the developments in Chapter 10.

We identify the nature of gossiping in the following. Let $T \in \mathbb{TR}$ with $T = (E, \leq, \lambda)$ and let $\widetilde{\Sigma}$ be a distributed alphabet implementing (Σ, I). Recall that $E_i = \{e \mid e \in E \text{ and } \lambda(e) \in \Sigma_i\}$. Let $c \in \mathcal{C}_T$ and $i \in \mathcal{P}$. Then $\downarrow^i(c)$ is *the i-view of c* and it is defined as $\downarrow^i(c) = \downarrow(c \cap E_i)$. Note that $\downarrow^i(c)$ is also a configuration. It is the "best" configuration that the agent i is aware of at c. We say that $\downarrow^i(c)$ is an *i-local configuration*. Let $\mathcal{C}_T^i = \{\downarrow^i(c) \mid c \in \mathcal{C}_T\}$ be the set of i-local configurations. For $Q \subseteq \mathcal{P}$ and $c \in \mathcal{C}_T$, we let $\downarrow^Q(c)$ denote the set $\bigcup\{\downarrow^i(c) \mid i \in Q\}$. Once again, $\downarrow^Q(c)$ is a configuration. It represents the collective knowledge of the processes in Q about the configuration c.

To bring out the relevant properties of the gossip automaton, let $T \in \mathbb{TR}$ with $T = (E, \leq, \lambda)$. For each subset Q of processes, the function $\mathsf{latest}_{T,Q} : \mathcal{C}_T \times \mathcal{P} \to Q$ is given by $\mathsf{latest}_{T,Q}(c, j) = \ell$ iff ℓ is the least member of Q (under the usual ordering over the integers) with the property $\downarrow^j(\downarrow^q(c)) \subseteq \downarrow^j(\downarrow^\ell(c))$ for every $q \in Q$. In other words, among the agents in Q, ℓ has the best information about j at c, with ties being broken by the usual ordering over integers.

Proposition 3.2.4 (Mukund, Sohoni [96]) *There exists an effectively constructible deterministic asynchronous automaton $\mathcal{Z}_\Gamma = (\{\Gamma_i\}, \{\Longrightarrow_a\}, \Gamma_{in}, \{(F_i, F_i^\omega)\})$ such that:*

(1) $\mathcal{L}(\mathcal{Z}_\Gamma) = \mathbb{TR}^\omega$.

(2) For each $Q = \{i_1, i_2, \ldots, i_n\}$, there exists an effectively computable function $\mathsf{gossip}_Q : \Gamma_{i_1} \times \Gamma_{i_2} \times \cdots \times \Gamma_{i_n} \times \mathcal{P} \to Q$ such that for every $T \in \mathbb{TR}^\omega$, every $c \in \mathcal{C}_T$ and every $j \in \mathcal{P}$, $\mathsf{latest}_{T,Q}(c, j) = \mathsf{gossip}_Q(\gamma(i_1), \ldots, \gamma(i_n), j)$ where $\rho_{_T}(c) = \gamma$ and $\rho_{_T}$ is the unique (accepting) run of $\mathcal{Z}_\Gamma$ over T.

Henceforth, we refer to $\mathcal{Z}_\Gamma$ as the *gossip automaton*. Each process in the gossip automaton has $2^{O(K^2 \log K)}$ local states, where $K = |\mathcal{P}|$. Moreover the function gossip_Q can be computed in time which is polynomial in the size of K.

The automaton described in [96] operates over finite traces but it is a trivial task to convert it into an asynchronous automaton having the desired properties.

3.2.5 Concurrent-regular expressions

We have seen that for both finite and infinite traces, the notions of regularity and recognizability coincide. However, it turns out that the class of languages described by regular trace expressions is *different* from the recognizable ones.

To bring this out, consider the trace alphabet (Σ, I) with $\Sigma = \{a, b\}$ and aIb. Then the regular trace expression $(ab)^*$ describes the set of strings over Σ with an equal number of a's and b's, which is *not* regular.

The problem arises when traces consisting of several "independent portions" are iterated. To bring this out, we say that a trace $T = (E, \leq, \lambda)$ is *connected* if $\lambda(E)$ induces a subgraph of (Σ, D) which is connected in the usual graph-theoretic sense. Equivalently, a trace is connected if and only if its Hasse diagram is connected when viewed in the straightforward manner as an undirected graph. It is then easy to see that any trace T can be (uniquely) factored into its *connected components*, i.e. $T = t_1 \cdots t_n$ with each t_i being a connected trace such that $\lambda(t_i) I \lambda(t_j)$ for $1 \leq i \neq j \leq n$. We let $\mathrm{con}(T) = \{t_1, \ldots, t_n\}$ and let $\mathrm{con}(\mathcal{L}) = \bigcup\{\mathrm{con}(T) \mid T \in \mathcal{L}\}$ be the *connected components of* $\mathcal{L}$ for a trace language $\mathcal{L} \in \mathbb{TR}^*$.

This led Ochmański to define a slightly modified iteration operator. In exactly the same manner as in Section 2.1.1, one can equip the regular expressions with an obvious trace semantics. In this notation, we introduce the *connected iteration* $\pi^\dagger$ as follows:

- $||\pi^\dagger|| = \mathrm{con}(||\pi||)^*$.

Intuitively, the iteration is now performed *not* over arbitrary traces of the language, but only over its connected components. With these definitions, $\mathrm{con}(ab) = \{a, b\}$ and $(ab)^\dagger = (a + b)^*$ is regular.

By replacing Kleene's iteration operator with Ochmański's connected iteration operator, one obtains the *concurrent-regular expressions*:

$$\mathrm{RE}(\Sigma, I) ::= a \mid \pi_0 + \pi_1 \mid \pi_0; \pi_1 \mid \pi^\dagger \ , \quad a \in \Sigma.$$

We arrive at the following nontrivial generalization of Kleene's Theorem.

Theorem 3.2.5 (Ochmański [105]) *Let $\mathcal{L} \subseteq \mathbb{TR}^*$. Then $\mathcal{L}$ is regular if and only if $\mathcal{L} = ||\pi||$ for some concurrent-regular expression π.*

This result is commonly referred to as *Ochmański's Theorem*.

In much the same was as the regular expressions was extended from finite to infinite strings, the concurrent-regular expressions can be extended to infinite traces. Ochmański's Theorem has been extended accordingly.

Theorem 3.2.6 (Gastin, Petit, Zielonka [44]) *Let $\mathcal{L} \subseteq \mathbb{TR}^\omega$. Then $\mathcal{L}$ is ω-regular if and only if $\mathcal{L} = ||\pi||$ for some concurrent ω-regular expression π.*

3.3 Monadic Second-order Logic for Traces

We saw in the preceding chapter how monadic second-order via Büchi's Theorem has become a yardstick for regularity in the interleaving setting. There is a very natural generalization to the richer domain of traces which, as it turns out,

captures exactly the class of regular *trace* languages. In this sense it provides further evidence that the notion of regularity is robust.

We first define the syntax and semantics of MSO for traces and then consider its connection to regular trace languages.

3.3.1 Syntax and semantics of MSO for traces

We fix a trace alphabet (Σ, I) for the remainder of this section. $\mathrm{MSO}(\Sigma, I)$, the *monadic second-order theory of (finite or infinite) traces over* (Σ, I), then has the same syntax as $\mathrm{MSO}(\Sigma)$. The structures are either finite or infinite traces of $\mathbb{TR}(\Sigma, I)$, and here we will present both on an equal footing.

Let $T \in \mathbb{TR}(\Sigma, I)$ with $T = (E, \leq, \lambda)$ and let $\mathcal{I} : X \to E$ be an interpretation. Then $T \models_{\mathcal{I}} Q_a(x)$ iff $\lambda(\mathcal{I}(x)) = a$ and $T \models_{\mathcal{I}} x \leq y$ iff $\mathcal{I}(x) \leq \mathcal{I}(y)$. Hence, the essential difference is that the binary predicate symbol is now interpreted as the *causal (partial) order* of the trace. The remaining semantic definitions go along the expected lines. Not surprisingly, each sentence φ defines the trace language $\mathcal{L}_\varphi = \{T \in \mathbb{TR} \mid T \models \varphi\}$. We say that $L \subseteq \mathbb{TR}$ is *definable in* MSO iff there exists a sentence φ in $\mathrm{MSO}(\Sigma, I)$ such that $\mathcal{L} = \mathcal{L}_\varphi$.

One should note that even though there is a close correspondence to $\mathrm{MSO}(\Sigma)$ in terms of syntax, formulas of the logic cannot be trivially carried over while preserving their meanings. As an example, the formula $\varphi = (\exists x)(\exists y)(\neg(x \leq y) \wedge \neg(y \leq x))$ is true of any trace with two concurrent events, but not true of *any* finite nor infinite string.

The fragment of *the first-order theory of (finite or infinite) traces over* (Σ, I), $\mathrm{FO}(\Sigma, I)$, is obtained from the monadic second-order logic of traces by abolishing second-order quantifications in the same way as in the noninterleaving case in Section 2.2.1. Clearly it will be strictly weaker than $\mathrm{MSO}(\Sigma, I)$ as, in particular, $\mathrm{FO}(\Sigma)$ is strictly weaker than $\mathrm{MSO}(\Sigma)$.

3.3.2 MSO and the regular trace languages

Thomas [141] adapted the classical proof of Section 2.2.2 to show that the MSO-definable languages of finite traces are precisely the regular trace languages, i.e. those recognized by asynchronous automata.

It turns out that the extension to infinite traces is not just a simple adaptation, but was shown by Ebinger and Muscholl [32] by utilizing already existing characterizations of regular (string and trace) languages such as Büchi's Theorem and Theorem 3.2.6 in conjunction with a connection between $\mathrm{MSO}(\Sigma)$ and $\mathrm{MSO}(\Sigma, I)$. The observation underlying this connection is that if $\mathcal{L} \subseteq \mathbb{TR}^\omega(\Sigma, I)$ is definable in $\mathrm{MSO}(\Sigma, I)$ then $\mathsf{ts}(\mathcal{L})$ is definable in $\mathrm{MSO}(\Sigma)$.

For languages of finite strings, the opposite result follows using an important technique, which will also be utilized for a similar result in Chapter 4. More precisely, one can show that if L is *trace consistent* with respect to (Σ, I) and definable in $\mathrm{MSO}(\Sigma)$ then $\mathsf{st}(L)$ is definable in $\mathrm{MSO}(\Sigma, I)$. The crucial observation is that by fixing a specific representative of each equivalence class $[\tau] \subseteq L$, one can express the total order $x \leq y$ of this representative by a translation to

$x \leq_{lex} y$, expressed using the partial order of the induced trace. This representative is taken to be $lex(\tau)$, the unique *lexicographically* least element of $[\tau]$ (given some fixed total order on Σ).

By replacing order formulas in this fashion, one translates formulas φ of $\mathrm{MSO}(\Sigma)$ to formulas $\hat{\varphi}$ of $\mathrm{MSO}(\Sigma, I)$ such that $\mathsf{st}(\tau) \models \hat{\varphi}$ if and only if $lex(\tau) \models \varphi$. This is sufficient as L is trace consistent.

The same argument holds true for the first-order fragments by replacing "MSO" with "FO" in the above proof sketch. In this manner, we have in fact revealed a very intimate relationship between MSO (FO) of finite traces and their corresponding fragments over *finite* strings. Unfortunately, the proof technique doesn't readily extend to the infinite setting, because the lexicographically smallest element is in general undefined for traces where the set of actions occurring infinitely often induces an unconnected subgraph of (Σ, D).

However, the result follows for MSO from the equivalence of $\mathrm{MSO}(\Sigma, I)$ and asynchronous (Büchi) automata. Fortunately, the remaining gap for FO in the infinite setting was closed by Ebinger and Muscholl by demonstrating the equivalence between star-free concurrent ω-regular expressions and first-order logic of traces and using the already known characterizations of first-order definability and star-free ω-regular expressions.

These remarks are collected in the following proposition.

Proposition 3.3.1 (Ebinger, Muscholl [32], Thomas [141]) *Let $L \subseteq \Sigma^\infty$. Then L is trace consistent and definable in* $\mathrm{MSO}(\Sigma)$ *(resp.* $\mathrm{FO}(\Sigma)$*) if and only if* $\{\mathsf{st}(\sigma) \mid \sigma \in L\}$ *is definable in* $\mathrm{MSO}(\Sigma, I)$ *(resp.* $\mathrm{FO}(\Sigma, I)$*).*

Interestingly, there are currently no results for variable-confined fragments of FO for traces. In particular, it is still not known whether the classical result mentioned in Section 2.4.2 that three variables suffice for $\mathrm{FO}(\Sigma)$, can be carried over to the setting of $\mathrm{FO}(\Sigma, I)$.

3.4 Linear Time Temporal Logics for Traces

Motivated by the fact that one can generate specifications that are guaranteed to be trace consistent, several linear time temporal logics for traces have been proposed starting with [135]. As mentioned earlier, such properties are precisely those amenable for partial-order reductions. Furthermore, these logics facilitate the specification of concurrency and causality as first-class notions.

There are several routes towards extending LTL to the setting of traces. One approach is based on *locations*, where one reasons explicitly about a distribution of computing agents cooperating through some communication structure given as a distributed alphabet. Another option is to view *events* as the partial order computation points in time, and base the specifications upon the relationship between individual events. Together these paradigms constitute the *local* trace logics. On the other hand, in *global* view of computations *configurations* are seen as instantaneous snapshots of the system at hand. In this sense, a configuration is a global view capturing a collection of simultaneous local views.

In this section we will cover all three approaches in the order introduced above, and survey the most important logics of each category.

3.4.1 TrPTL and location-based logics

We present here the linear time temporal logic over traces called TrPTL, which was the first logic patterned after PTL (i.e. LTL) formulated for traces. It was proposed by Thiagarajan [135] and initiated the developments of linear time temporal logics for traces.

The logic TrPTL is location-based and parameterized by the class of distributed alphabets. Through this section, we fix a distributed alphabet $\widetilde{\Sigma} = \{\Sigma_i\}_{i\in\mathcal{P}}$ with $\mathcal{P} = \{1, 2, \ldots, K\}$ and $K \geq 1$. The trace alphabet induced by $\widetilde{\Sigma}$ is denoted (Σ, I). We adopt the notations from the previous sections; in particular, we will often write just $\{X_i\}$ when dealing with a $\mathcal{P}$-indexed family $\{X_i\}_{i\in\mathcal{P}}$.

Having fixed $\widetilde{\Sigma}$ we shall often almost always write TrPTL to mean TrPTL($\widetilde{\Sigma}$), the logic associated with $\widetilde{\Sigma}$. Then $\Phi_{\mathrm{TrPTL}(\widetilde{\Sigma})}$, the set of formulas of TrPTL($\widetilde{\Sigma}$), is given by:

$$\Phi_{\mathrm{TrPTL}(\widetilde{\Sigma})} ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a\rangle_i\alpha \mid \alpha\mathcal{U}_i\beta \ , \quad a \in \Sigma_i.$$

Throughout we denote $\Phi_{\mathrm{TrPTL}(\widetilde{\Sigma})}$ as just Φ. In the semantics of the logic, which will be based on infinite traces, the i-view of a configuration as defined in Section 3.2.4 will play a crucial role as we shall see shortly. A model is a trace $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$. Let $c \in \mathcal{C}_T$ and $\alpha \in \Phi$. Then $T, c \models \alpha$ denotes that α is satisfied at c in T and it is defined inductively as follows:

- $T, c \models tt$, $T, c \models \neg\alpha$, and $T, c \models \alpha \vee \beta$ are defined as usual.
- $T, c \models \langle a\rangle_i\alpha$ iff there exists $e \in E_i - c$ such that $\lambda(e) = a$ and $T, \downarrow e \models \alpha$. Moreover, for every $e' \in E_i$, $e' < e$ iff $e' \in c$.
- $T, c \models \alpha\mathcal{U}_i\beta$ iff there exists $c' \in \mathcal{C}_T$ such that $c \subseteq c'$ and $T, \downarrow^i(c') \models \beta$. Moreover, for every $c'' \in \mathcal{C}_T$, if $\downarrow^i(c) \subseteq \downarrow^i(c'') \subset \downarrow^i(c')$ then $T, \downarrow^i(c'') \models \alpha$.

A formula $\alpha \in \Phi$ is *satisfiable* if there exists a trace T and a configuration $c \in \mathcal{C}_T$ such that $T, c \models \alpha$. Moreover, T *satisfies* α, denoted $T \models \alpha$, in case $T, \emptyset \models \alpha$. As usual, the *language defined by* α is given as $\mathcal{L}_\alpha = \{T \in \mathbb{TR}(\Sigma, I) \mid T \models \alpha\}$. We will say that $\mathcal{L}$ is *definable in* TrPTL($\widetilde{\Sigma}$) iff there exists some formula α of such that $\mathcal{L}_\alpha = \mathcal{L}$.

Thus TrPTL is an action-based multi-agent version of LTL. Indeed both in terms of its syntax and semantics, LTL(Σ) corresponds to the case where there is only one agent. The semantics of TrPTL when specialized down to this case yields the previous LTL(Σ) semantics as given in Section 2.3.1.

The assertion $\langle a\rangle_i\alpha$ says that the agent i will next participate in an a-event. Moreover, at the resulting i-view, the assertion α will hold. The assertion $\alpha\mathcal{U}_i\beta$ says that there is a future i-view (including the present i-view) at which β

will hold and for all the intermediate i-views (if any) starting from the current i-view, the assertion α will hold.

Atomic propositions can easily be incorporated into the logic in a number of ways. Chapter 8 illustrates this point by means of *local* valuation functions assigning a set of atomic propositions to each i-local configuration. There are other similar approaches, but it is interesting to note that all atomic assertions (that we know of) concerning distributed behaviours are local in nature. Indeed, it is well-known that global atomic propositions will at once lead to an undecidable logic in the current setting [81, 112].

Before considering examples of TrPTL specifications, we will introduce some notation. We let α, β with or without subscripts range over Φ. Abusing notation, we will use loc to denote the map which associates a set of *locations* with each formula.

- $\mathrm{loc}(tt) = \emptyset$.
- $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$.
- $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.
- $\mathrm{loc}(\langle a\rangle_i\alpha) = \mathrm{loc}(\alpha\mathcal{U}_i\beta) = \{i\}$.

In what follows, $\Phi^i = \{\alpha \mid \mathrm{loc}(\alpha) \subseteq \{i\}\}$ is the set of i-type formulas. We note that a TrPTL formula of the form $\langle a\rangle_i\alpha$ could have $j \in \mathrm{loc}(\alpha)$ with $j \neq i$. A similar remark applies to the indexed until-operators.

A basic observation concerning the semantics of TrPTL is that for a trace T and formula α with $\mathrm{loc}(\alpha) \subseteq Q$, it is the case that $T, c \models \alpha$ iff $T, \downarrow^Q(c) \models \alpha$. Thus if $\alpha \in \Phi^i$ then $T, c \models \alpha$ if and only if $T, \downarrow^i(c) \models \alpha$. As a result, the formulas in Φ^i can be used in exactly the same manner as one would use LTL to express properties of the agent i. Boolean combinations of such local assertions can be used to capture various interaction patterns between the agents implied by the logical connectives as well as the coordination enforced by the distributed alphabet $\widetilde{\Sigma}$.

For writing specifications, apart from the usual derived connectives that we already introduced in Section 2.3.1 for LTL, a number of derived operators are available. $[a]_i\alpha = \neg\langle a\rangle_i\neg\alpha$, $\Diamond_i\alpha = tt\mathcal{U}_i\alpha$, and $\Box_i\alpha = \neg\Diamond_i\neg\alpha$ are all local versions of the corresponding derived operator of LTL. Moreover, for $X \subseteq \Sigma_i$ let $\alpha\mathcal{U}_i^X\beta = (\alpha \wedge \bigwedge_{a\in\Sigma_i - X}[a]_i ff)\mathcal{U}_i\beta$. In other words, $\alpha\mathcal{U}_i^X\beta$ is fulfilled using (at most) actions taken from X. We set $\Diamond_i^X\alpha = tt\mathcal{U}_i^X\alpha$ and $\Box_i^X\alpha = \neg\Diamond_i^X\neg\alpha$.

Another interesting operator that we will use later is $\alpha@i \stackrel{\mathrm{def}}{=} \alpha\mathcal{U}_i\alpha$ (or equivalently $ff\mathcal{U}_i\alpha$). $\alpha@i$ is to be read as "α at i". If T is a trace and $c \in \mathcal{C}_T$ then $T, c \models \alpha@i$ iff $T, \downarrow^i(c) \models \alpha$. (It could of course be the case that $\mathrm{loc}(\alpha) \neq \{i\}$.)

A simple but important observation is that every formula is a boolean combination of formulas taken from $\bigcup_{i\in\mathcal{P}}\Phi^i$, which makes TrPTL well suited for describing local properties. However, in TrPTL we can express certain global properties, e.g. that some specific global configuration is reachable from the initial configuration. Let $\{\alpha_i\}_{i\in\mathcal{P}}$ be a family with $\alpha_i \in \Phi^i$ for each i. Then we can

define a derived connective $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ which has the following semantics at the empty configuration. Let T be a trace. Then $T, \emptyset \models \Diamond(\alpha_1, \alpha_2, \ldots, \alpha_k)$ iff there exists $c \in \mathcal{C}_T$ such that $T, c \models \bigwedge_{i=1}^{K} \alpha_i$.

To define this derived connective set $\Sigma'_1 = \Sigma_1$ and, for $1 < i \leq K$, set $\Sigma'_i = \Sigma_i - \cup\{\Sigma_j \mid 1 \leq j < i\}$. Then $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ is the formula:

$$\Diamond_1^{\Sigma'_1}(\alpha_1 \wedge \Diamond_2^{\Sigma'_2}(\alpha_2 \wedge \Diamond_3^{\Sigma'_3}(\alpha_3 \wedge \cdots \Diamond_K^{\Sigma'_K}\alpha_K))\cdots).$$

The idea is that the sequence of actions leading up to the required configuration can be reordered so that one first performs all the actions in Σ_1, then all the actions in $\Sigma_2 - \Sigma_1$ etc. Dually, safety properties that hold at the initial configuration can also be expressed in a similar fashion.

On the other hand, it seems difficult to express nested global and safety properties in TrPTL. It is also the case that due to the local nature of the modalities, information about the past sneaks into the semantics even though there are no explicit past operators in the logic. We make this apparent in an example.

Example 3.4.1 Consider again the distributed alphabet $\widetilde{\Sigma}_0 = \{\Sigma_1, \Sigma_2\}$ with $\Sigma_1 = \{a, d\}$ and $\Sigma_2 = \{b, d\}$. Then the trace T_0 of Figure 3.1 is a trace over the trace alphabet induced by $\widetilde{\Sigma}_0$. Let $c_0 \in \mathcal{C}_{T_0}$ be the configuration, where the first a, the first d, and both b's have occurred. Finally, consider the TrPTL-formula $\alpha_0 = \Box_2\neg\langle a\rangle_2 tt \wedge (\langle b\rangle_2 tt)@1$. Then α_0 asserts that 2 will never be able to perform a b-action again, while *according the 1's view of the current configuration*, 2 will next engage in a b-action. It's then not hard to verify that (perhaps counter-intuitively) $T_0, c_0 \models \alpha_0$. However, there exists no trace $T \in \mathbb{TR}$ such that $T, \emptyset \models \alpha_0$.

A formula α is said to be *root-satisfiable* iff there exists a trace T such that $T, \emptyset \models \alpha$. On the other hand, α is said to be *satisfiable* iff there exists a trace T and $c \in \mathcal{C}_T$ such that $T, c \models \alpha$. It turns out that these two notions are *not* equivalent as brought out by Example 3.4.1. One can however transform every formula α into a formula α' such that α is satisfiable iff α' is root-satisfiable.

This follows from the observation that every α can be expressed as a boolean combination of formulas taken from the set $\bigcup_{i\in\mathcal{P}} \Phi^i$. Hence the given formula α can be assumed to be of the form $\alpha = \bigvee_{j=1}^{m}(\alpha_{j1} \wedge \alpha_{j2} \wedge \cdots \wedge \alpha_{jK})$ where $\alpha_{ji} \in \Phi^i$ for each $j \in \{1, 2, \ldots, m\}$ and each $i \in \mathcal{P}$. Now convert α to the formula α' where $\alpha' = \bigvee_{j=1}^{m} \Diamond(\alpha_{j1}, \alpha_{j2}, \cdots, \alpha_{jK})$. From the semantics of $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ above it follows that α is satisfiable iff α' is root-satisfiable.

Hence, in principle, it suffices to consider only root-satisfiability in developing a decision procedure for TrPTL. There is of course a blow-up involved in converting satisfiable formulas to root-satisfiable formulas. If one wants to avoid this blow-up then the decision procedure for checking root-satisfiability can be suitably modified to yield a direct decision procedure for checking satisfiability as done in [135]. In any case, it is root-satisfiability which is of importance from the standpoint of model checking. Hence here we shall only develop a procedure for deciding if a given formula of TrPTL is root-satisfiable.

As a first step we augment the syntax of our logic by one more construct.

- If α is a formula, so is $O_i\alpha$. In the trace T at the configuration $c \in \mathcal{C}_T$, $T, c \models O_i\alpha$ iff $T, c \models \langle a\rangle_i\alpha$ for some $a \in \Sigma_i$. We also define $\mathrm{loc}(O_i\alpha) = \{i\}$.

Thus $O_i\alpha \equiv \bigvee_{a\in\Sigma_i}\langle a\rangle_i\alpha$ is a valid formula and O_i is expressible in the former syntax. It will be however more efficient to admit O_i as a first class modality as we did in Section 2.3.2.

Fix a formula α_0. Our aim is to effectively associate an asynchronous automaton $\mathcal{Z}_{\alpha_0}$ with α_0 such that α_0 is root-satisfiable iff $L_{Tr}(\mathcal{Z}_{\alpha_0}) \neq \emptyset$. Since the emptiness problem for asynchronous automata is decidable (see Section 3.2.3), this will yield the desired decision procedure. Let $cl(\alpha_0)$ be the least set of formulas containing α_0 which satisfies:

- $\neg\alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.
- $\alpha \vee \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$.
- $\langle a\rangle_i\alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.
- $O_i\alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.
- $\alpha\mathcal{U}_i\beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$. In addition, $O_i(\alpha\mathcal{U}_i\beta) \in cl(\alpha_0)$.

We then define $CL(\alpha_0)$ to be the set $cl(\alpha_0) \cup \{\neg\beta \mid \beta \in cl(\alpha_0)\}$. As usual, $\neg\neg\beta$ is identified with β and we write CL instead of $CL(\alpha_0)$.

$A \subseteq CL$ is called an *i-type atom* iff it satisfies:

- $tt \in A$.
- $\alpha \in A$ iff $\neg\alpha \notin A$.
- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- $\alpha\mathcal{U}_i\beta \in A$ iff $\beta \in A$ or ($\alpha \in A$ and $O_i(\alpha\mathcal{U}_i\beta) \in A$).
- If $\langle a\rangle_i\alpha$, $\langle b\rangle_i\beta \in A_i$ then $a = b$.

AT_i denotes the set of i-type atoms. We now need to define the notion of a formula in CL being a member of a collection of atoms. Let $\alpha \in CL$ and $\{A_i\}_{i\in Q}$ be a family of atoms with $\mathrm{loc}(\alpha) \subseteq Q$ and $A_i \in AT_i$ for each $i \in Q$. We define the predicate Member(α,$\{A_i\}_{i\in Q}$), which for convenience will be denoted by $\alpha \in \{A_i\}_{i\in Q}$. It is defined inductively as:

- If $\mathrm{loc}(\alpha) = \{j\}$ then $\alpha \in \{A_i\}_{i\in Q}$ iff $\alpha \in A_j$.
- If $\alpha = \neg\beta$ then $\alpha \in \{A_i\}_{i\in Q}$ iff $\beta \notin \{A_i\}_{i\in Q}$.
- If $\alpha = \alpha_1 \vee \alpha_2$ then $\alpha_1 \vee \alpha_2 \in \{A_i\}_{i\in Q}$ iff $\alpha_1 \in \{A_i\}_{i\in Q}$ or $\alpha_2 \in \{A_i\}_{i\in Q}$.

The construction of the asynchronous automaton $\mathcal{Z}_{\alpha_0}$ is guided by the construction developed for LTL in Section 2.3.3. However in the much richer setting of traces it turns out that one must make crucial use of the latest information that the agents have about each other when defining the transitions of $\mathcal{Z}_{\alpha_0}$. Hence one needs to incorporate the gossip automaton $\mathcal{Z}_\Gamma = (\{\Gamma_i\}, \{\Longrightarrow_a\}, \Gamma_{in}, \{(F_i, F_i^\omega)\})$ described in Proposition 3.2.4.

Each i-state of the automaton $\mathcal{Z}_{\alpha_0}$ will consist of an i-type atom together with an appropriate i-state of the gossip automaton. Two additional components will be used to check for liveness requirements. One component will take values from the set $N_i = \{0, 1, 2, \ldots, |U_i|\}$ where $U_i = \{\alpha \mathcal{U}_i \beta \mid \alpha \mathcal{U}_i \beta \in CL\}$. This component will be used to ensure that all "until" requirements are met. The other component will take values from the set $\{\mathsf{on}, \mathsf{off}\}$. This will be used to detect when an agent has quit.

The automaton $\mathcal{Z}_{\alpha_0}$ can now be defined as $\mathcal{Z}_{\alpha_0} = (\{S_i\}, \{\longrightarrow_a\}, S_{in}, \{(F_i, F_i^\omega)\})$, where:

- For each i, $S_i = AT_i \times \Gamma_i \times \{0, 1, 2, \ldots, |U_i|\} \times \{\mathsf{on}, \mathsf{off}\}$. (Recall that Γ_i is the set of i-states of the gossip automaton.)
- Let $s_a, s'_a \in S_a$ with $s_a(i) = (A_i, \gamma_i, n_i, v_i)$ and $s'_a(i) = (A'_i, \gamma'_i, n'_i, v'_i)$ for each $i \in \mathrm{loc}(a)$. Then $(s_a, s'_a) \in \longrightarrow_a$ iff the following conditions are met:
  - $(\gamma_a, \gamma'_a) \in \Longrightarrow_a$ (recall that $\{\Longrightarrow_a\}$ is the family of transition relations of the gossip automaton) where $\gamma_a, \gamma'_a \in \Gamma_a$ such that $\gamma_a(i) = \gamma_i$ and $\gamma'_a(i) = \gamma'_i$ for each $i \in \mathrm{loc}(a)$.
  - $\forall i, j \in \mathrm{loc}(a)$, $A'_i = A'_j$.
  - $\forall i \in \mathrm{loc}(a)$ $\forall \langle a \rangle_i \alpha \in CL$. $\langle a \rangle_i \alpha \in A_i$ iff $\alpha \in A'_i$.
  - $\forall i \in \mathrm{loc}(a)$ $\forall O_i \alpha \in CL$. $O_i \alpha \in A$ iff $\alpha \in A'_i$.
  - $\forall i \in \mathrm{loc}(a) \forall \langle b \rangle_i \beta \in CL$. If $\langle b \rangle_i \beta \in A_i$ then $b = a$.
  - Suppose $j \notin \mathrm{loc}(a)$ and $\beta \in CL$ with $\mathrm{loc}(\beta) = \{j\}$. Further suppose that $\mathrm{loc}(a) = \{i_1, i_2, \ldots, i_n\}$. Then $\beta \in A'_i$ iff $\beta \in A_\ell$ where $\ell = \mathsf{gossip}_{\mathrm{loc}(a)}(\gamma_{i_1}, \gamma_{i_2}, \ldots, \gamma_{i_n}, j)$.
  - Let $i \in \mathrm{loc}(a)$, $U_i = \{\alpha_1 \mathcal{U}_i \beta_1, \alpha_2 \mathcal{U}_i \beta_2, \ldots, \alpha_{n_i} \mathcal{U}_i \beta_{n_i}\}$. Then u'_i and u_i are related to each other via:

    $$u'_i = \begin{cases} (u_i{+}1) \bmod (n_i{+}1), & \text{if } u_i = 0 \text{ or } \beta_{u_i} \in A_i \text{ or } \alpha_{u_i} \mathcal{U}_i \beta_{u_i} \notin A_i \\ u_i, & \text{otherwise} \end{cases}$$

  - For each $i \in \mathrm{loc}(a)$, $v_i = \mathsf{on}$. Moreover, if $v'_i = \mathsf{off}$ then $\langle a \rangle_i \alpha \notin A'_i$ for every $i \in \mathrm{loc}(a)$ and every $\langle a \rangle_i \alpha \in CL$.
- Let $s \in S_{\mathcal{P}}$ with $s(i) = (A_i, \gamma_i, u_i, v_i)$ for every i. Then $s \in S_{in}$ iff $\alpha_0 \in \{A_i\}_{i \in \mathcal{P}}$ and $\gamma \in \Gamma_{in}$ where $\gamma \in \Gamma_{\mathcal{P}}$ satisfies $\gamma(i) = \gamma_i$ for every i. Furthermore, $u_i = 0$ for every i. Finally, for every i, $v_i = \mathsf{off}$ implies that $\langle a \rangle_i \alpha \notin A_i$ for every $\langle a \rangle_i \alpha \in CL$.

- For each i, $F_i^\omega \subseteq S_i$ is given by $F_i^\omega = \{(A_i, \gamma_i, u_i, v_i) \mid u_i = 0 \text{ and } v_i = \mathsf{on}\}$ and $F_i \subseteq S_i$ is given by $F_i = \{(A_i, \gamma_i, u_i, v_i) \mid v_i = \mathsf{off}\}$.

Following [134, 135] one now shows that α_0 is root-satisfiable iff $\mathcal{L}_{Tr}(\mathcal{Z}_{\alpha_0}) \neq \emptyset$. Moreover, the number of local states of $\mathcal{Z}_{\alpha_0}$ is bounded by $2^{O(\max(n, m^2 \log m))}$ where $n = |\alpha_0|$ and m is the number of agents mentioned in α_0. Clearly, $m \leq n$. We then have the following result.

Theorem 3.4.2 (Thiagarajan [134, 135]) *The root-satisfiability problem (and in fact the satisfiability problem) for* TrPTL *is solvable in time* $2^{O(\max(n, m^2 \log m) \cdot m)}$.

The number of local states of each process in $\mathcal{Z}_{\alpha_0}$ is determined by two quantities: the length of α_0 and the size of the gossip automaton $\mathcal{Z}_\Gamma$. As far as the size of $\mathcal{Z}_\Gamma$ is concerned, it is easy to verify that we need to consider only those agents in $\mathcal{P}$ that are mentioned in $\mathrm{loc}(\alpha_0)$, rather than all agents in the system.

In certain cases, the satisfiability problem can be solved within more appealing time bounds. Meyer and Petit [89] have shown that whenever the induced trace alphabet is disconnected, it is possible to decompose formulas of TrPTL into disconnected portions in order to reduce the complexity of the satisfiability problem.

The *model checking problem* for TrPTL can be phrased as follows. A finite state distributed program Pr over $\widetilde{\Sigma}$ is an asynchronous automaton $\mathcal{Z}_{Pr} = (\{S_i^{Pr}\}, \{\Longrightarrow_a^{Pr}\}, S_{in}^{Pr}, \{(S_i^{Pr}, S_i^{Pr})\})$ modeling the state space of Pr.

Viewing a formula α_0 as a specification, we say that Pr *meets the specification* α_0—denoted $Pr \models \alpha_0$—if for every $T \in \mathbb{TR}^\omega$, if $\mathcal{Z}_{Pr}$ has a run over T then $T, \emptyset \models \alpha_0$. In a manner identical to Section 2.3.3, the model checking problem for TrPTL can now be solved by "intersecting" the program automaton $\mathcal{Z}_{Pr}$ with the formula automaton $\mathcal{Z}_{\neg\alpha_0}$ to yield an automaton $\mathcal{Z}$ such that $\mathcal{L}_{Tr}(\mathcal{Z}) = \mathcal{L}_{Tr}(\mathcal{Z}_{Pr}) \cap \mathcal{L}_{Tr}(\mathcal{Z}_{\neg\alpha_0})$. As before, $\mathcal{L}_{Tr}(\mathcal{Z}) = \emptyset$ iff $Pr \models \alpha_0$.

It turns out that this model checking problem has time complexity $O(|\mathcal{Z}_{Pr}| \cdot 2^{O(\max(n, m^2 \log m) \cdot m)})$ where $|\mathcal{Z}_{Pr}|$ is the size of the global state space of the asynchronous automaton modeling the behaviour of the given program Pr and, as before, $n = |\alpha_0|$ and m is the number of agents mentioned in α_0, where α_0 is the specification formula.

We now take a brief look at some related location-based linear time temporal logics over traces. The first one is the sublogic of TrPTL denoted TrPTL$^{\mathrm{con}}$ which consists of the so called *connected formulas* of TrPTL. We define $\Phi^{\mathrm{con}}_{\mathrm{TrPTL}}$ (from now on written as Φ^{con}) to be the least subset of Φ satisfying the following conditions:

- $tt \in \Phi^{\mathrm{con}}$.
- If $\alpha \in \Phi^{\mathrm{con}}$ then $\neg\alpha \in \Phi^{\mathrm{con}}$.
- If $\alpha, \beta \in \Phi^{\mathrm{con}}$ then $\alpha \vee \beta \in \Phi^{\mathrm{con}}$.

- If $\alpha \in \Phi^{\mathrm{con}}$ and $a \in \Sigma_i$ such that $\mathrm{loc}(\alpha) \subseteq \mathrm{loc}(a)$ then $\langle a \rangle_i \alpha \in \Phi^{\mathrm{con}}$.
- If $\alpha, \beta \in \Phi^{\mathrm{con}}$ with $\mathrm{loc}(\alpha), \mathrm{loc}(\beta) \subseteq \bigcap\{\mathrm{loc}(a) \mid a \in \Sigma_i\}$ then $\alpha \mathcal{U}_i \beta \in \Phi^{\mathrm{con}}$.

Connected formulas were first identified by Niebert and used by Huhn [65]. They have also been independently identified by Ramanujam [115]. Thanks to the syntactic restrictions imposed on the next-state and until formulas, past information is not allowed to creep in. Indeed one can prove that α is satisfiable if and only if α is root-satisfiable.

Yet another pleasing feature of $\mathrm{TrPTL}^{\mathrm{con}}$ is that the gossip automaton can be eliminated in the construction of the automaton $\mathcal{Z}_{\alpha_0}$ whenever $\alpha_0 \in \Phi^{\mathrm{con}}$. The automaton construction for full TrPTL on page 42 can be modified accordingly to obtain that the satisfiability problem for $\mathrm{TrPTL}^{\mathrm{con}}$ is solvable in time $2^{O(|\alpha_0|)}$. Once again, a suitably modified statement can be made about the associated model checking problem. At present it is not known whether or not TrPTL is strictly more expressive than $\mathrm{TrPTL}^{\mathrm{con}}$.

Niebert also identified another logic related to TrPTL. By extending TrPTL with certain fixed point operators as in Section 2.4.3, one obtains a location-based version [101] of the linear-time μ-calculus. This logic was later revised to become νTrPTL [102], which Niebert shows expressively equivalent to the monadic second-order theory of traces while still elementary-time decidable.

The semantics of TrPTL is based on (local) agents' knowledge of a (global) configuration, as is hence reminiscent of the so-called logics of knowledge [38]. Ramanujam [115] makes this connection explicit by defining four logics of increasing expressive power based on knowledge-like modalities in the trace setting. Two of the four logics considered turn out to be $\mathrm{LTL}^{\otimes}$ (to be considered in Section 3.5.3) and $\mathrm{TrPTL}^{\mathrm{con}}$.

3.4.2 TLC and event-based logics

We now move to the event-based logics over traces. The most prominent of these logics is the *Temporal Logic of Causality (TLC)*, which was introduced by Alur, Peled, and Penczek [4] to formulate causality and concurrency properties in a direct fashion. It is a temporal logic for traces which allows quantification over causal chains. Its set of formulas[2] is given as follows:

$$\begin{aligned} \mathrm{TLC}(\Sigma, I) \quad ::= \quad & p_a \mid \neg\alpha \mid \alpha \vee \beta \mid \mathrm{co}(\alpha) \mid EX(\alpha) \mid EU(\alpha, \beta) \mid EG(\alpha) \mid \\ & EX^{-}(\alpha) \mid EU^{-}(\alpha, \beta) \ , \quad a \in \Sigma. \end{aligned}$$

The syntax that we have chosen here is inspired by the branching time temporal logic CTL [19]. Indeed this is no coincidence, as TLC can be viewed as CTL interpreted over the Hasse diagram of the partial order of the trace at hand. To bring out the semantics, we will need some simple definitions.

Let (Σ, I) be a trace alphabet which we fix throughout this section. For any trace $T = (E, \leq, \lambda)$, we will in the presentation of the event-based logics

[2] The syntax is slightly different from [4], but we have chosen the present syntax for purposes that will become evident when we later in the subsection extend TLC.

assume the existence of a unique least event $\perp \in E$ corresponding to a system initialization event carrying no label, i.e. $\lambda(\perp)$ is undefined and $\perp < e$ for every $e \in E - \{\perp\}$.

A *future causal chain rooted at* $e \in E$ is a (finite or infinite) sequence $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ with $e = e_0$, $e_i \in E$ such that $e_{i-1} \lessdot e_i$ for every $i \geq 1$. The labelling function $\lambda : E \to \Sigma$ is extended to causal chains in the obvious way by $\lambda(\rho) = \lambda(e_0)\lambda(e_1)\cdots\lambda(e_n)\cdots$. We say that a future causal chain ρ is *maximal* in case ρ is either infinite or it is finite and there exists no $e' \in E$ such that $e_{|\rho|} \lessdot e'$. Past causal chains are defined in the obvious manner.

We can now define the semantics of TLC. Let $T \in \mathbb{TR}(\Sigma, I)$ and $e \in E$. The notion of a formula α of TLC being satified at an event e of T is defined inductively in the following manner.

- $T, e \models p_a$ iff $\lambda(e) = a$.
- $T, e \models \neg\alpha$ and $T, e \models \alpha \vee \beta$ are defined as usual.
- $T, e \models \mathrm{co}(\alpha)$ iff there exists an $e' \in E$ with $e \; co \; e'$ and $T, e' \models \alpha$.
- $T, e \models EX(\alpha)$ iff there exists some $e' \in E$ with $e \lessdot e'$ such that $T, e' \models \alpha$.
- $T, e \models EU(\alpha, \beta)$ iff there exists a future causal chain rooted at e, $\rho = (e_0 \lessdot e_1 \lessdot \cdots \lessdot e_n)$, such that $T, e_n \models \beta$ and $T, e_i \models \alpha$ for each $0 \leq i < n$.
- $T, e \models EG(\alpha)$ iff there exists a maximal future causal chain rooted at e, $\rho = (e_0 \lessdot e_1 \lessdot \cdots \lessdot e_n \lessdot \cdots)$, such that $T, e_i \models \alpha$ for each $0 \leq i$.
- $T, e \models EX^-(\alpha)$ iff there exists some $e' \in E$ with $e' \lessdot e$ such that $T, e' \models \alpha$.
- $T, e \models EU^-(\alpha, \beta)$ iff there exists a past causal chain rooted at e, $\rho = (e_n \lessdot e_{n-1} \lessdot \cdots \lessdot e_0)$, such that $T, e_n \models \beta$ and $T, e_i \models \alpha$ for each $0 \leq i < n$.

We say that α of TLC is *satisfiable* in case there exist a trace $T = (E, \leq, \lambda)$ and an event $e \in E$ such that $T, e \models \alpha$. By $T \models \alpha$ we denote that T *satisfies* α, i.e. $T, \perp \models \alpha$. The *language defined by* α is given by $\mathcal{L}_\alpha = \{T \in \mathbb{TR}(\Sigma, I) \mid T \models \alpha\}$. We will say that $\mathcal{L}$ *is definable in* $\mathrm{TLC}(\Sigma, I)$ iff there exists some formula α of $\mathrm{TLC}(\Sigma, I)$ such that $\mathcal{L}_\alpha = \mathcal{L}$.

Note that we can derive the propositional constants *tt* and *ff* as $p_a \vee \neg p_a$ and its negation. We abbreviate $EF(\alpha) = EU(tt, \alpha)$ and $EF^-(\alpha) = EU^-(tt, \alpha)$. Moreover, it is possible to derive EG^- as $EG^-(\alpha) = EU^-(\alpha, \alpha \wedge \neg EX^-(tt))$ and it is hence not included in the syntax.

The satisfiability problem for TLC is solved by once again extending the classical decision procedure of LTL in Section 2.3.2, albeit in a quite different way. In more detail, given a formula α_0 of TLC, one constructs a Streett automaton $\mathcal{S}_{\alpha_0}$ accepting the set of linearizations of traces satisfying α. In other words, $\mathcal{L}(\mathcal{S}) = \mathsf{ts}(\mathcal{L}_{\alpha_0})$. The essential feature here, is that the causal chains can be iteratively recovered as certain subsequences of actions in the linearizations, and hence the modalities on causal chains can be given fixed point formulations over *linearizations*. This leads to the following result.

Theorem 3.4.3 (Alur, Peled, Penczek [4]) *The satisfiability problem for* TLC(Σ, I) *is decidable in exponential time.*

As before, the usual arguments apply to the model checking problem.

TLC is ideally suited to express causality properties such as *serializability* of partially ordered computations as exemplified in [4]:

Example 3.4.4 Let $A, B \subseteq \Sigma$ be sets of actions indicating that stages of the transactions p and q, respectively, are executing. Then

$$\alpha \stackrel{\text{def}}{=} \neg \left(EF(\bigvee_{a \in A} p_a \wedge EF(\bigvee_{b \in B} p_b)) \wedge EF(\bigvee_{a \in A} p_a \wedge EF^{-}(\bigvee_{b \in B} p_b)) \right)$$

asserts that it is not allowed that events of one transaction both causally precede *and* succeed events of the other.

One of the weaknesses of TLC is that it doesn't directly facilitate reasoning about causal relationships of the individual events of the causal chains at hand. As a consequence, a number of interesting properties are not (either easily or at all) expressible within TLC. (This claim will be substantiated later.)

It turns out that one can strengthen the chain quantification of TLC to obtain the logic TLC* [54], which enjoys the same similarity to CTL* [19] as TLC has to CTL. A more elaborate treatment of TLC* can be found in Chapter 9 so we just highlight its main features here.

TLC* consists of three different syntactic entities; event formulas (Φ_{ev}), future chain formulas (Φ^{+}_{ch}) and past chain formulas (Φ^{-}_{ch}) defined by mutual induction as described below:

$$\begin{array}{lcl} \Phi_{ev} & ::= & p_a \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \mathrm{co}(\alpha) \mid E(\phi) \mid E^{-}(\psi)\ , \quad a \in \Sigma. \\ \Phi^{+}_{ch} & ::= & \alpha \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 U \phi_2. \\ \Phi^{-}_{ch} & ::= & \alpha \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid X^{-}\psi \mid \psi_1 U^{-} \psi_2\ , \end{array}$$

where α, ϕ and ψ with or without subscripts here and throughout the rest of the section are formulas of Φ_{ev}, Φ^{+}_{ch} and Φ^{-}_{ch}, respectively. The formulas of TLC$^*(\Sigma, I)$ are the set of *event* formulas Φ_{ev} as defined above.

The semantics of formulas of TLC* is divided into two parts; event formulas and chain formulas. Let $T \in \mathbb{TR}(\Sigma, I)$ and $e \in E$. The notion of an event formula α being satified at an event e of T is defined inductively in a manner very similar to TLC with the following modifications:

- $T, e \models E(\phi)$ iff there exists a future causal chain ρ rooted at e with $T, \rho \models \phi$.
- $T, e \models E^{-}(\psi)$ iff there exists a past causal chain ρ rooted at e with $T, \rho \models \psi$.

Now, assuming $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ is a future causal chain, the notion of $T, \rho \models \phi$ for a future chain formula ϕ is as for LTL in Section 2.3.1 in the obvious way by setting: $T, \rho \models \alpha$ iff $T, e_0 \models \alpha$. The notion of $T, \rho \models \psi$ for a past causal chain ρ and past chain formula ψ is defined in the straightforward manner.

The notions of satisfiability and language definability are carried over from TLC in the obvious manner. It is not hard to show that the satisfiability problem of the extension to TLC* remains decidable because it can be embedded into monadic second-order logic for traces. Chapter 9 investigates this issue in more detail.

The formulas of TLC(Σ, I) are easily seen to be the set of formulas of TLC$^*(\Sigma, I)$ where each of the chain operators X, U, G, X^-, U^- is immediately preceded by a chain quantifier E. Indeed, TLC is a sublogic of TLC*. In Chapter 9 we prove formally that TLC* is strictly more expressive that TLC. Here, we shall content ourselves by bringing out a natural property which is easily definable in TLC* but *not* in TLC.

Example 3.4.5 Suppose that a and b are actions representing the acquisition and release, respectively, of some resource. A relevant property of this system is whether there exists some causal chain in the execution of the system — presumably containing other system actions than $\{a, b\}$—such that the a's and b's alternate strictly until the task is perhaps eventually completed. Via the future chain formula $\phi_{xy} = p_x \Rightarrow X(\neg(p_x \vee p_y)U(p_y))$ we can easily express this property in TLC* by $E(G(\phi_{ab} \wedge \phi_{ba}))$. The point is here that TLC* allows us to investigate each causal chain in mention by a causal chain formula, which is then confined to this very chain. This is not possible in TLC, as the existential quantifications interpreted at some fixed event of the chain would potentially consider *all* causal chains originating at this event—not just the one presently being investigated.

Walukiewicz [150] has taken the idea of causal logics further and has defined several variations of the μ-calculus [130] interpreted over the Hasse diagram of the trace. He shows that the pure causal event-based μ-calculus itself is not sufficient to express all regular trace languages. However, he goes on to demonstrate that one can augment this logic by different operators reminiscent of the present co-operator to obtain event-based fixed point logics expressively complete with respect to monadic second-order logic over traces while maintaining PSPACE-complete satisfiability problems. The novel new feature is that his completeness proof factors through monadic second-order logic of infinite trees, which is being employed to define trees of lexicographically least paths of the corresponding traces. We will not bring out the details here, merely remark that we will later denote this family of logics by $\mu-$co.

3.4.3 LTrL and configuration-based logics

It is not difficult to show that TrPTL is no more expressive than the first-order theory of traces but it is not known whether the converse also holds. Moreover,

it seems difficult to express arbitrary global liveness properties in the location- and event-based temporal logics over traces.

It would be nice to have a linear time temporal logic for traces patterned after LTL which has the same expressive power as the first-order theory of traces, and which would furthermore allow easy formulations of liveness properties. Further motivation is provided by Proposition 3.3.1, because such a logic would capture exactly all the trace consistent properties of LTL.

With this as motivation, a different kind of trace-based linear time temporal logic called LTrL has been proposed in [139] by Thiagarajan and Walukiewicz. This logic works directly with a trace alphabet (i.e. it is not based on agents). However, it is not interpreted over individual events, but instead over *configurations* of a trace. Its syntax is given by:

$$\mathrm{LTrL}(\Sigma, I) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a \rangle \alpha \mid \alpha \, U \, \beta \mid \langle a^{-1} \rangle tt \ , \quad a \in \Sigma.$$

Thus the syntax is very close to LTL except for the addition of a very restricted past-operator. In fact, just a *linear* number of past-operators are present in the logic; one for each action.

A model of $\mathrm{LTrL}(\Sigma, I)$ is a trace $T = (E, \leq, \lambda)$. Let $c \in \mathcal{C}_T$ be a configuration of T. Then $T, c \models \alpha$ will stand for α being satisfied at c in T. This notion is defined inductively as follows:

- $T, c \models tt$, $T, c \models \neg\alpha$, and $T, c \models \alpha \vee \beta$ are defined in the expected manner.
- $T, c \models \langle a \rangle \alpha$ iff there exists $c' \in \mathcal{C}_T$ with $c \xrightarrow{a}_T c'$ with $T, c' \models \alpha$.
- $T, c \models \alpha \, U \, \beta$ iff there exists $c' \in \mathcal{C}_T$ with $c \subseteq c'$ such that $T, c' \models \beta$. Moreover, for every $c'' \in \mathcal{C}_T$, $c \subseteq c'' \subset c'$ implies $T, c'' \models \alpha$.
- $T, c \models \langle a^{-1} \rangle tt$ iff there exists $c' \in \mathcal{C}_T$ with $c' \xrightarrow{a}_T c$.

As usual, we say that a formula $\alpha \in \mathrm{LTrL}(\Sigma, I)$ is *satisfiable* if there exists a trace $T \in \mathbb{TR}$ and $c \in \mathcal{C}_T$ such that $T, c \models \alpha$. Moreover, T *satisfies* α, denoted $T \models \alpha$, in case $T, \emptyset \models \alpha$. The *language defined by* α is given by $\mathcal{L}_\alpha = \{T \in \mathbb{TR} \mid T \models \alpha\}$. We will say that $\mathcal{L}$ is *definable in* $\mathrm{LTrL}(\Sigma, I)$ iff there exists some formula α of such that $\mathcal{L}_\alpha = \mathcal{L}$.

Notice that LTrL is a very natural generalization of LTL to the setting of traces, where configurations are the trace-theoretic equivalents of computation prefixes. Thus it seems like a direct trace-based analogue of LTL. In the obvious manner, we derive $\Diamond\alpha = ttU\alpha$ and its dual $\Box\alpha = \neg\Diamond\neg\alpha$. With these definitions, it's very easy to express global liveness properties.

Example 3.4.6 Let (Σ, I) be any trace alphabet and let $A \subseteq \Sigma$ be a set of pairwise independent events. The existence of infinitely many configurations with the top events labelled exactly with the actions of A is then described by

$$\alpha_A \stackrel{\mathrm{def}}{=} \Box\Diamond\Big(\bigwedge_{a \in A} \langle a^{-1} \rangle tt \wedge \bigwedge_{b \notin A} \neg\langle b^{-1} \rangle tt\Big).$$

Indeed, all global liveness properties *can* be expressed in LTrL; the major result concerning LTrL is the following.

Theorem 3.4.7 (Thiagarajan, Walukiewicz [139]) *Let* $\mathcal{L} \subseteq \mathbb{TR}^{\omega}(\Sigma, I)$. *Then* $\mathcal{L}$ *is definable in* $\mathrm{LTrL}(\Sigma, I)$ *if and only if* $\mathcal{L}$ *is definable in* $\mathrm{FO}(\Sigma, I)$.

Thus, except for the addition of the restricted past-operators, LTrL is a generalization of Kamp's Theorem to the much richer setting of traces.

A natural question to ask is whether or not one can get a direct analogue of Kamp's Theorem by removing the restricted past-operators from LTrL to obtain exactly LTL interpreted directly over traces. Meyer and Petit did show that the past-operators can be eliminated without loss of expressive power when the logic is interpreted over *finite* traces, but the proof [90] turned out to contain a mistake, which is not readily correctable.

Later Diekert and Gastin exhibited an expressively complete temporal logic, $\mathrm{LTL}_{\mathrm{f}}$, based on LTrL *without past-operators*, but at the expense of introducing new future *filtering* modalities [25]. However, very recently Diekert and Gastin [26] have shown that these filtering modalities are not necessary to obtain expressive completeness.

Theorem 3.4.8 (Diekert, Gastin [26]) *Let* $\mathcal{L} \subseteq \mathbb{TR}(\Sigma, I)$. *Then* $\mathcal{L}$ *is definable in* $\mathrm{LTL}(\Sigma, I)$ *if and only if* $\mathcal{L}$ *is definable in* $\mathrm{FO}(\Sigma, I)$.

Hence, LTL for traces—interpreted directly over configurations of traces instead of finite prefixes of sequences—is indeed expressively complete with respect to first-order logic for traces. In this sense, Thiagarajan and Walukiewicz, together with Diekert and Gastin, have provided exactly *the* linear time temporal for traces.

Unfortunately, this configuration-based logic does not have a matching time complexity in relation to LTL. Walukiewicz has shown that the satisfiability problem for LTrL (and also LTL for traces) is nonelementary hard [149]. These discouraging news have lead Walukiewicz [151] and others to conjecture that there exists no expressively complete *and* elementary-time decidable temporal logic over traces patterned after LTL.

Gastin, Meyer, and Petit [42] show that one can still give an automata-theoretic decision procedure for LTrL. The procedure does not, however, generalize the classical solution for LTL of Section 2.3.2, but instead constructs the Büchi automaton $\mathcal{B}_{\alpha_0}$ in a modular fashion by induction on α_0.

3.5 Product Languages

We will now exhibit a restricted but useful class of distributed behaviours that we call product behaviours. Such behaviours are generated by a network of sequential agents that coordinate their activities by performing common actions together. It turns out that product behaviours are naturally trace consistent.

They also constitute a clean and yet nontrivial subset of the class of trace behaviours.

We start by defining the regular product languages and sketch how they can be characterized by a subclass of the asynchronous automata. Finally, we solve the satisfiability and model checking problems for product versions of linear time temporal logics via these automata.

3.5.1 Regular product languages

We begin by bringing out the product languages. To this end we fix a distributed alphabet $\widetilde{\Sigma}$ for the remainder of the section and define the K-ary operation $\otimes : 2^{\Sigma_1^\infty} \times 2^{\Sigma_2^\infty} \times \cdots \times 2^{\Sigma_K^\infty} \to 2^{\Sigma^\infty}$ via $\otimes(L_1, \ldots, L_K) = \{\sigma \mid \sigma{\restriction}\Sigma_i \in L_i \text{ for each } i\}$.

In what follows we will write $L = L_1 \otimes L_2 \cdots \otimes L_K$ to denote the fact $\otimes(L_1, \ldots, L_K) = L$. We say that $L \subseteq \Sigma^\infty$ is a *direct product language* over $\widetilde{\Sigma}$ iff there exist $L_i \subseteq \Sigma_i^\infty$ for each i such that $L = L_1 \otimes L_2 \otimes \cdots \otimes L_K$. Here and elsewhere we will say "product language" instead of "product language over $\widetilde{\Sigma}$" etc.

As usual, for an alphabet Σ and $L \subseteq \Sigma^\infty$ we say that L is *regular* iff $L \cap \Sigma^*$ is a regular subset of Σ^* and $L \cap \Sigma^\omega$ is an ω-regular subset of Σ^ω as described in Section 2.3.1.

We can now define the class of regular product languages as follows. $\mathcal{R}_0^\otimes(\widetilde{\Sigma})$ is the subset of 2^{Σ^∞} given by $L \in \mathcal{R}_0^\otimes(\widetilde{\Sigma})$ iff $L = L_1 \otimes L_2 \otimes \cdots \otimes L_K$ with each L_i a regular subset of Σ_i^∞. The class of *regular product languages* over $\widetilde{\Sigma}$, denoted $\mathcal{R}^\otimes(\widetilde{\Sigma})$, is then the least subset of 2^{Σ^∞} which contains $\mathcal{R}_0^\otimes$ and is closed under finite unions. As usual, we shall often write $\mathcal{R}_0^\otimes$ instead of $\mathcal{R}_0^\otimes(\widetilde{\Sigma})$ and write $\mathcal{R}^\otimes$ instead of $\mathcal{R}^\otimes(\widetilde{\Sigma})$. An interesting observation concerning $\mathcal{R}^\otimes$, due to Thiagarajan [136], is that $\mathcal{R}^\otimes$ is closed under boolean operations. Moreover, it's not hard to see that product languages are naturally trace consistent.

3.5.2 Product automata

We will bring out how the regular product languages constitute a nice, nontrivial, but proper subclass of the regular trace languages. It will become clear that the product languages can intuitively be viewed as the "communication free" trace languages.

As a first step, we will characterize the regular product languages in terms of automata. Recall that the transition relation of an asynchronous automaton $\mathcal{Z} = (\{S_p\}, \{\longrightarrow_a\}, S_{in}, \{(F_p, F_p^\omega)\})$ as described in Section 3.2.3) is given as a relation between a-states, i.e. $\longrightarrow_a \subseteq S_{\mathrm{loc}(a)} \times S_{\mathrm{loc}(a)}$ with $a \in \Sigma$. Intuitively, communication between the component automata is performed by enforcing that only some of the possible joint moves might be allowed and that information is shared among the participating processes by a global interrelated update of their resulting local states.

Here, we will restrict attention to the subset of asynchronous automata where the a-transitions are products of *local* component transition relations. In this

sense, these automata correspond to networks of processes *purely synchronizing* without exchanging information. Formally, a *product Büchi automaton* over $\widetilde{\Sigma}$ is a structure $\mathcal{B}^{\otimes} = (\{\mathcal{B}_i\}_{i=1}^{K}, Q_{in})$ where $\mathcal{B}_i = (Q_i, \longrightarrow_i, F_i, F_i^{\omega})$ for each i, such that:

- Q_i is a finite set of i-local states.
- $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$ is the transition relation of the ith component.
- $F_i \subseteq Q_i$ is a set of i-local finitary accepting states.
- $F_i^{\omega} \subseteq Q_i$ is a set of i-local infinitary accepting states.
- $Q_{in} \subseteq Q_1 \times Q_2 \times \cdots \times Q_K$ is a set of global initial states.

Let $\mathcal{B}^{\otimes} = (\{\mathcal{B}_i\}_{i=1}^{K}, Q_{in})$ be a product Büchi automaton over $\widetilde{\Sigma}$. From now on we will say just "product automaton". Also, we shall often suppress mention of $\widetilde{\Sigma}$ and write $\{\mathcal{B}_i\}$ instead of $\{\mathcal{B}_i\}_{i=1}^{K}$. Let $\mathcal{B}_i = (Q_i, \longrightarrow_i, F_i, F_i^{\omega})$. Then we set $Q_G^{\mathcal{B}^{\otimes}} = Q_1 \times Q_2 \times \cdots \times Q_K$. When $\mathcal{B}^{\otimes}$ is clear from the context, we will write Q_G instead of $Q_G^{\mathcal{B}^{\otimes}}$. The global transition relation of $\mathcal{B}^{\otimes}$ is denoted as $\longrightarrow_{\mathcal{B}^{\otimes}}$ and it is the subset of $Q_G \times \Sigma \times Q_G$ given by:

$$q \xrightarrow{a}_{\mathcal{B}^{\otimes}} q' \text{ iff } \forall\, i \in \mathrm{loc}(a) : q[i] \xrightarrow{a}_i q'[i] \text{ and } \forall\, i \notin \mathrm{loc}(a) : q[i] = q'[i].$$

Let $\sigma \in \Sigma^{\infty}$. A *run* of $\mathcal{B}^{\otimes}$ over σ is a map $\rho : \mathrm{prf}(\sigma) \longrightarrow Q_G$ which satisfies:

- $\rho(\varepsilon) \in Q_{in}$.
- $\rho(\tau) \xrightarrow{a}_{\mathcal{B}^{\otimes}} \rho(\tau a)$ for every $\tau a \in \mathrm{prf}(\sigma)$.

A simple but useful property of runs is the following. Suppose ρ is a run of the product automaton $\mathcal{B}^{\otimes}$ over σ. Suppose further that $\tau, \tau' \in \mathrm{prf}(\sigma)$ such that $\tau \restriction \Sigma_i = \tau' \restriction \Sigma_i$ for some i. Then $\rho(\tau)[i] = \rho(\tau')[i]$.

Let ρ be a run of the product automaton $\mathcal{B}^{\otimes}$ over σ. Then ρ is *accepting* iff for each i, the following condition is satisfied:

- If $\sigma \restriction \Sigma_i$ is finite then $\rho(\tau)[i] \in F_i$ where $\tau \in \mathrm{prf}(\sigma)$ such that $\tau \restriction \Sigma_i = \sigma \restriction \Sigma_i$.
- If $\sigma \restriction \Sigma_i$ is infinite then $\rho(\tau a)[i] \in F_i^{\omega}$ for infinitely many $\tau a \in \mathrm{prf}(\sigma)$ with $a \in \Sigma_i$.

We now define $L(\mathcal{B}^{\otimes})$, the *language accepted by the product automaton* $\mathcal{B}^{\otimes}$ as $L(\mathcal{B}^{\otimes}) = \{\sigma \mid \exists \text{ an accepting run of } \mathcal{B}^{\otimes} \text{ over } \sigma\}$. It is now easy to see that product automata can be viewed as a subclass of the asynchronous automata. The following brings out an important characterization.

Theorem 3.5.1 (Thiagarajan [136]) *Let $L \subseteq \Sigma^{\infty}$. Then L is a regular product language if and only if L is accepted by some product automaton.*

Product automata can be naturally applied to settle the satisfiability and model checking problems for the logic $\mathrm{LTL}^{\otimes}$ to be introduced in the next section.

3.5.3 Product logics

We now wish to bring out a product version of LTL denoted $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$. The set of formulas and their *locations* are given by:

- tt is a formula and $\mathrm{loc}(tt) = \emptyset$.
- Suppose α and β are formulas. Then so are $\neg\alpha$ and $\alpha \vee \beta$. Furthermore, $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$ and $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.
- Suppose $a \in \Sigma_i$ and α is a formula with $loc(\alpha) \subseteq \{i\}$. Then $\langle a\rangle_i\alpha$ is a formula and $loc(\langle a\rangle_i\alpha) = \{i\}$.
- Suppose α and β are formulas such that $\mathrm{loc}(\alpha), \mathrm{loc}(\beta) \subseteq \{i\}$. Then $\alpha\mathcal{U}_i\beta$ is a formula. Moreover, $\mathrm{loc}(\alpha\mathcal{U}_i\beta) = \{i\}$.

We note that each formula in $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ is a boolean combination of formulas taken from the set $\bigcup_{i\in Loc} \mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$ where, for each i,

$$\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma}) = \{\alpha \mid \alpha \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma}) \text{ and } \mathrm{loc}(\alpha) \subseteq \{i\}\ \}.$$

It is thus easy to see that at least in terms of syntax, $\mathrm{LTL}^{\otimes}$ is a sublogic of TrPTL (and in fact $\mathrm{TrPTL}^{\mathrm{con}}$). The semantics to be presented below supports this view, and indeed $\mathrm{LTL}^{\otimes}$ was isolated in [137] as a nice trace consistent class of formulas and shown equivalent to a natural subset of TrPTL. For this reason, we present below the semantics in terms of strings and their projections onto components.

As before, we will often suppress the mention of $\widetilde{\Sigma}$. We will also often write τ_i, τ_i' and τ_i'' instead of $\tau \restriction \Sigma_i$, $\tau' \restriction \Sigma_i$ and $\tau'' \restriction \Sigma_i$, respectively, with $\tau, \tau', \tau'' \in \Sigma^*$. A model is a sequence $\sigma \in \Sigma^\infty$ and the semantics of this logic is given, as before, with $\tau \in \mathrm{prf}(\sigma)$.

- $\sigma, \tau \models tt$, $\sigma, \tau \models \neg\alpha$, and $\sigma, \tau \models \alpha \vee \beta$ are as usual.
- $\sigma, \tau \models \langle a\rangle_i\alpha$ iff there exists $\tau' \in \mathrm{prf}(\sigma)$ such that $\sigma, \tau' \models \alpha$ and $\tau_i' = \tau_i a$. (recall that $\tau_i' = \tau' \restriction \Sigma_i$.)
- $\sigma, \tau \models \alpha\mathcal{U}_i\beta$ iff there exists τ' such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Further, for every $\tau'' \in \mathrm{prf}(\tau')$, if $\varepsilon \preceq \tau_i'' \prec \tau_i'$ then $\sigma, \tau\tau'' \models \alpha$.

We will say that a formula $\alpha \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ is *satisfiable* if there exist $\sigma \in \Sigma^\infty$ and $\tau \in \mathrm{prf}(\sigma)$ such that $\sigma, \tau \models \alpha$. The *language defined by* α is given by $L_\alpha = \{\sigma \in \Sigma^\infty \mid \sigma, \varepsilon \models \alpha\}$. We say that a language $L \subseteq \Sigma^\infty$ is *definable in* $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ iff there exists some α such that $L_\alpha = L$. The modalities O_i, $\Diamond_i$ and $\Box_i$ are once again derivable in the standard fashion as seen earlier in this chapter.

Even though $\mathrm{LTL}^{\otimes}$ might seem expressively weak, many interesting properties can still be stated easily.

Example 3.5.2 Consider the distributed alphabet $\widetilde{\Sigma}_0 = (\{a, b\}, \{b, c\}, \{c, d\})$ and the formula $\Box_1 O_1 tt \Rightarrow \Box_3 O_3 tt$. This property asserts that along every computation, if the first component executes infinitely often then so does the third component. The point to note is that the first component and the third component do not have any common events and hence there is no direct communication between them. Nevertheless through the power of the boolean connectives alone, the logic can make assertions about the way components that are "far apart" are required to influence each other's behaviour.

The satisfiability problem for $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ can be solved by effectively constructing a product automaton $\mathcal{B}^{\otimes}_{\alpha_0}$ for each $\alpha_0 \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ such that the language accepted by $\mathcal{B}^{\otimes}_{\alpha_0}$ is nonempty if and only if α_0 is satisfiable [137]. The construction is a generalization of the one for LTL in Section 2.3.2 and can be viewed as a simplification of the construction for TrPTL (and $\mathrm{TrPTL}^{\mathrm{con}}$) presented in Section 3.4.1.

Theorem 3.5.3 (Thiagarajan [137]) *The satisfiability problem for* $\mathrm{LTL}^{\otimes}$ *is decidable in exponential time.*

The solution to the satisfiability problem will at once lead to a solution to the model checking problem for programs modeled as a product of sequential agents. A product program (over $\widetilde{\Sigma}$) is a structure $Pr^{\otimes} = (\{Pr_i\}_{i=1}^{K}, Q_{in}^{Pr})$ where, for each i, $Pr_i = (Q_i, \longrightarrow_i)$ with Q_i a finite set and $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$. As usual, it is not difficult to prove the model checking problem for $\mathrm{LTL}^{\otimes}$ is decidable in time $O(|Pr| \cdot 2^{|\alpha_0|})$. The details can be found in Chapter 8.

We will consider the expressiveness of $\mathrm{LTL}^{\otimes}$ in the next section, but it is easy to see that it cannot express all regular product languages. Hence, one might look for natural extensions of $\mathrm{LTL}^{\otimes}$ to capture exactly the class of regular product languages.

One possibility is to define a product version of DLTL by strengthening the until-operator in a manner similar to the one in Section 2.4.3. More precisely, we replace the defining clause of $\alpha \mathcal{U}_i \beta$ by

- Suppose α and β are formulas such that $\mathrm{loc}(\alpha), \mathrm{loc}(\beta) \subseteq \{i\}$, and suppose that π is a regular expression over Σ_i. Then $\alpha \mathcal{U}_i^{\pi} \beta$ is a formula. Moreover, $\mathrm{loc}(\alpha \mathcal{U}_i^{\pi} \beta) = \{i\}$.

The semantics is then changed accordingly as follows:

- $\sigma, \tau \models \alpha \mathcal{U}_i^{\pi} \beta$ iff there exists τ' such that $\tau\tau' \in \mathrm{prf}(\sigma)$ with $\tau_i \in ||\pi||$ and $\sigma, \tau\tau' \models \beta$. Further, for every $\tau'' \in \mathrm{prf}(\tau')$, if $\varepsilon \preceq \tau_i'' \prec \tau_i'$ then $\sigma, \tau\tau'' \models \alpha$.

The resulting logic, denoted $\mathrm{DLTL}^{\otimes}$, is explored in more detail in Chapter 7. With the obvious notion of definability carried over, it turns out that $\mathrm{DLTL}^{\otimes}$ is indeed expressively complete with respect to $\mathcal{R}^{\otimes}$.

Theorem 3.5.4 *Let* $L \subseteq \Sigma^{\infty}$. *Then* L *is definable in* $\mathrm{DLTL}^{\otimes}$ *if and only if* L *is a regular product language.*

Furthermore, given a formula α_0 of DLTL$^{\otimes}$, we can generalize the constructions for DLTL in Chapter 6 and LTL$^{\otimes}$ [137] to obtain an automaton $\mathcal{B}^{\otimes}_{\alpha_0}$. As before, one can show that α_0 is satisfiable iff $L(\mathcal{B}^{\otimes}_{\alpha_0}) \neq \emptyset$. This leads to:

Theorem 3.5.5 *The satisfiability problem for* DLTL$^{\otimes}$ *is decidable in exponential time.*

As for LTL$^{\otimes}$ above, the model checking problem for DLTL$^{\otimes}$ is decidable in time $O(|Pr| \cdot 2^{|\alpha_0|})$.

3.6 Expressiveness of Logics for Traces

As we have seen, there are very many different ways to extend the classical linear time temporal logic LTL to the setting of Mazurkiewicz traces. None of these extensions have been commonly accepted as *the* right extension of LTL, mainly because the nonelementary lower bound for the configuration-based logics makes them compare unfavourably in relation to the exponential-time decidable LTL for sequences.

The tight relationship between first-order logic for strings and first-order logic for traces, as brought out in Proposition 3.3.1, suggests that one should look for an elementary-time (preferably exponential-time) logic equal in expressive power to the first-order theory of traces. While such a logic has yet to be identified (if it exists!), we will conclude this chapter by summarizing the relative merits of the logics encountered in the chapter and giving an overview of their expressive powers.

Surprisingly, this area has turned out to be far more challenging than its interleaving counterpart, mainly because the logics are parameterized by *trace alphabets* which, even with the same set of underlying system actions Σ, might have very different independence structure dictated by the relation $I \subseteq \Sigma \times \Sigma$.

Hence, for a logic A to be *at least as expressive* as B we will demand that any property $\mathcal{L} \subseteq \mathbb{TR}(\Sigma, I)$ expressible by a formula of $B(\Sigma, I)$ is also expressible by a formula of $A(\Sigma, I)$, for *every* trace alphabet (Σ, I). A is then *more expressive than* B if A is at least as expressive as B and there exists a trace alphabet (Σ, I) and a property $\mathcal{L} \subseteq \mathbb{TR}(\Sigma, I)$ such that $\mathcal{L}$ is definable in $A(\Sigma, I)$ but *not* in $B(\Sigma, I)$. The notions of expressive equivalence and incomparability should now be obvious from these remarks.

A quick overview is displayed in Figure 3.2. A dotted (solid) arrow from A to B indicates that B is at least as expressive as (strictly more expressive than) A. A directed squiggled line from A to B indicates that B does *not* subsume A in terms of expressive power, whereas undirected squiggled lines denote that the logics are incomparable in terms of expressive power.

We first recall that there exist two logics expressively equivalent to monadic second-order logic; Niebert's νTrPTL as mentioned in Section 3.4.1, and the family of $\mu-$co by Walukiewicz from Section 3.4.2. Both are exponential-time decidable. At present there exists no (known) event- or location-based temporal logics expressively complete with respect to first-order logic. As sketched in

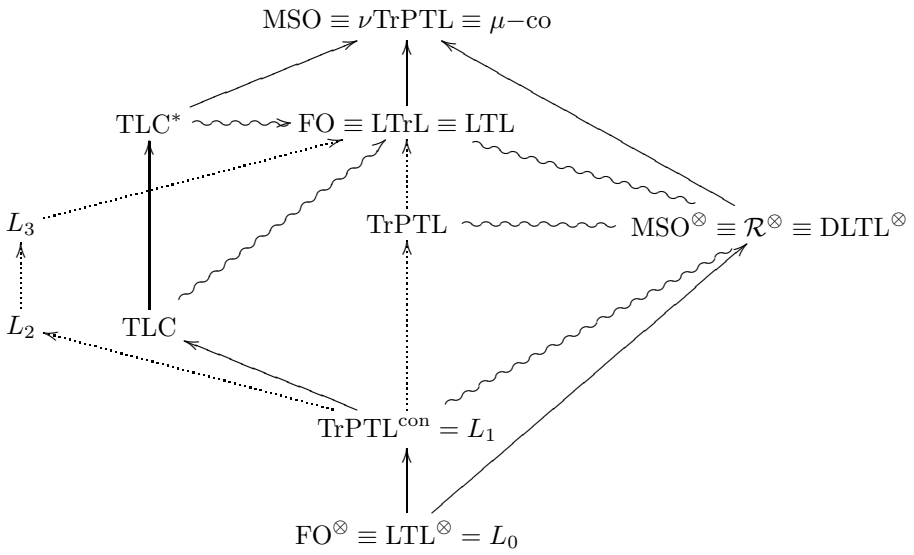


Figure 3.2: Overview of relative expressiveness.

Section 3.4.3, the configuration-based logic LTrL by Thiagarajan and Walukiewicz and its sublogic LTL by Diekert and Gastin are expressively complete with respect to first-order logic, but both of these have nonelementary complexity.

It follows from the sequential case where $I = \emptyset$ that FO is strictly weaker than MSO. One example language separating these logics are "a is performed at every even position" as mentioned in Section 2.4.3. It turns out that this is a recurring phenomenon; virtually all separation results are separations inherited trivially from the sequential setting.

It is not hard to see that TrPTL is expressible within FO [97], but it is not known whether this inclusion is strict. It is widely believed, however, that TrPTL is expressively strictly weaker than FO. Obviously, $\mathrm{TrPTL}^{\mathrm{con}}$ is a syntactic subset of TrPTL and hence expressively no stronger than TrPTL. Again, it is neither known whether $\mathrm{TrPTL}^{\mathrm{con}}$ is strictly weaker than the full TrPTL nor whether it is strictly weaker than first-order logic. It is not hard to translate formulas of $\mathrm{TrPTL}^{\mathrm{con}}$ into equivalent formulas of TLC. This is done by deriving an until-operator of TLC which only operates over locations of some given distributed alphabet, and translating the i-type until operators of $\mathrm{TrPTL}^{\mathrm{con}}$ to this derived operator of TLC. We will, however, not bring out the details here.

The first nontrivial separation result is due to Thiagarajan [136], who showed that $\mathrm{LTL}^{\otimes}$ is strictly weaker that $\mathrm{TrPTL}^{\mathrm{con}}$. (Recall from Section 3.5.3 that $\mathrm{LTL}^{\otimes}$ is a syntactic subset of $\mathrm{TrPTL}^{\mathrm{con}}$.) This can be brought out by considering the property $L = (abd + bad + a'b'd + b'a'd)^{\omega}$ over the distributed alphabet $\widetilde{\Sigma} = (\{a, a', d\}, \{d, b, b'\})$. It is not hard to see that L is trace consistent and

that one can easily exhibit a TrPTL$^{\text{con}}$-formula defining L. However, Thiagarajan shows that for direct product languages L' and $\sigma \in \Sigma^{\infty}$, it is the case that $\sigma \in L'$ if and only if for each i there exist $\sigma_i \in L'$ such that $\sigma \restriction \Sigma_i = \sigma_i \restriction \Sigma_i$. Thus one shows that L cannot be a finite union of such languages, and hence not a product language. Intuitively, "mixed" substrings of the form $ab'd$, $b'ad$, $a'bd$, and $ba'd$ would then inevitably also have to be allowed. In this sense, the separation once again goes back to properties of strings.

This example is in fact also an instance of a regular trace language which is not a regular product language. Hence $\mathcal{R}^{\otimes}$ constitutes a proper subclass of the regular trace (consistent) languages. It is easy in the obvious way to define product versions of first-order and monadic second-order logic, which would then be expressively equivalent to LTL$^{\otimes}$ and DLTL$^{\otimes}$, respectively. Once again, as the sequential setting is recovered by the distributed alphabet $\widetilde{\Sigma} = (\Sigma)$, it follows from the classical theory that FO$^{\otimes}$ is strictly weaker than MSO$^{\otimes}$. The example property L above illustrates in conjunction with the sequential setting the fact that MSO$^{\otimes}$ is incomparable to FO and the expressively weaker TrPTL and TrPTL$^{\text{con}}$.

Ramanujam [115] introduced four logics L_0, L_1, L_2, and L_3 with each L_i being a syntactic subset of L_{i+1} for $i = 0, 1, 2$. As mentioned in Section 3.4.1, L_0 and L_1 can be identified with LTL$^{\otimes}$ and TrPTL$^{\text{con}}$. It can be seen that they're all definable within first-order logic, but their exact expressiveness was not investigated. We conjecture however that all four are expressible with TrPTL.

Finally, we consider TLC and its extension TLC*. It is easy to see that in the sequential case, both TLC and TLC* boil down to past-augmented LTL, which by Kamp's Theorem is known to be expressively complete with respect to FO and strictly weaker than MSO. However, one can formally show that TLC* is expressively strictly stronger than TLC exactly when the dependency relation of the underlying trace alphabet is not transitive. This is done in Chapter 9, where we tailor the Ehrenfeucht-Fraïssé game of Etessami and Wilke [37] to the setting of Mazurkiewicz traces to demonstrate the separation. More precisely, we show that the "counting" property $\mathcal{L}_{abd} = [abdabd]^*$ is undefinable in TLC over trace alphabets with $I = \{(a, b), (b, a)\}$ such as in Example 3.1.1. As an corollary, we obtain that TLC* is not included in FO.

Neither TLC nor TLC* has a completely resolved relationship to FO, but progress has been made very recently [151], as Walukiewicz has identified an example of a property definable in TLC, but *not* in FO. This also witnesses our inclusion of TrPTL$^{\text{con}}$ into TLC to be strict. It is still an open problem, however, whether TLC can express all properties of FO, though it is believed not to be the case.

In effect, the game-theoretic approach provide the only separation results of logics over traces phrased directly over the partial orders, *without* relying on the properties of the sequential case. Moreover, it is also applicable to location- and configuration-based logics, and might illuminate a path towards future separation results within the area of temporal logics over traces.

Chapter 4

Message Sequence Charts

The previous chapter demonstrates how the basic formulation of automata-based verification leads to elegant but nontrivial generalizations in the noninterleaving setting of Mazurkiewicz traces. Such objects are ideally suited for the specification and verification of distributed systems composed of independently computing agents occasionally communicating by handshake mechanisms.

Recently there has been a spurt of activity within the area of formal description and validation of distributed systems such as telecommunication software, where the computing agents communicate by sending messages to each other. However, it turns out that the synchronous view of distributed systems on which traces are based becomes inappropriate when the fundamental communication paradigm is message-passing. Instead the predominant description technique is that of Message Sequence Charts (MSCs), which have an appealing and intuitive visual syntax. These objects are particularly well suited for the description of scenarios for distributed telecommunications [121] and are often used to capture system requirements in the early design stages.

In its basic form, an MSC depicts a single partially-ordered execution of a distributed system which just describes the exchange of messages between the processes of the system. A collection of MSCs is used to capture some set of scenarios that a designer might want the system to exhibit (or avoid). A standard way to generate a collection of MSCs is to use a Hierarchical (or High-level) Message Sequence Chart (HMSC) [84]. An HMSC is a finite directed graph in which each node is labelled, in turn, by an HMSC. The HMSCs labelling the vertices are not refer to each other. The collection of MSCs represented by an HMSC consists of all MSCs obtained by tracing a path in the HMSC from an initial vertex to a terminal vertex and concatenating the MSCs that are encountered along the path.

Because of the restrictions on the labelling of HMSCs, we can derive an equivalent Message Sequence Graph (MSG) by flattening out the hierarchical labelling in an HMSC. In other words, an MSG is a graph where each node is labelled by a simple MSC. Like an HMSC, an MSG defines a collection of MSCs obtained by concatenating the MSCs labelling each path from an initial ver-

tex to a terminal vertex. Though HMSCs provide more succinct specifications than MSGs, they are only as expressive as MSGs. Thus, one often restricts one's attention to characterizing structural properties of MSGs rather than of HMSCs [5, 98, 100].

In this chapter, we consider some foundational steps towards transferring the well-established techniques of the previous chapters to the message-passing setting of MSCs. In order to do so, we first propose notions of regularity, "message-passing" automata, and monadic second-order logic. Unlike Chapters 2 and 3, we will confine our attention to *finite* MSCs only. This is due to the fact that the issues investigated here have at present no counterparts in the infinite setting. We feel however that our results will serve as a good launching pad for a similar account concerning infinite MSCs. This should then lead to the design of appropriate temporal logics and automata-theoretic solutions (based on message-passing automata) to model-checking problems for these logics.

We begin by investigating the basic objects of study in Section 4.1 and present, like for traces, two equivalent representations of MSCs. Guided by Chapter 3 we propose a notion of regularity of collections of MSCs. In Section 4.2 we introduce a class of finite-state acceptors called message-passing automata, which accept exactly the MSC languages that are regular in our sense. We provide another characterization of regular MSC languages in terms of definability of a natural monadic second-order logic in Section 4.3. Then, in Section 4.4, we define MSGs and survey the existing theory of detecting certain specific properties of MSGs. It turns out that not all languages described by MSGs are regular. Conversely, not all regular MSC languages can be defined by MSGs. Following this, in Section 4.5, we proceed by identifying the key notion of finitely generated MSC languages, which is a property shared by all languages of MSGs. We then exhibit a subclass of MSGs which defines precisely those languages that are *both* finitely generated and regular. In this way, we give a precise characterization of collections of MSCs which are both regular and definable by MSGs. We conclude the chapter by briefly discussing various options of regular expressions for MSCs.

This material provides the background for our contributions in Chapters 10 and 11. These very recent developments are guided by the theory and results of Chapters 1–3.

4.1 Message Sequence Charts

We first introduce MSCs by means of its original visual representation, which induces a restricted labelled partial order in the natural manner. Then we show how MSCs can be given an equivalent representation in terms of equivalence classes of certain strings.

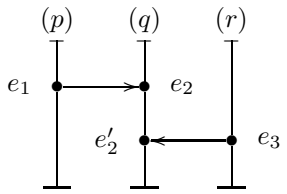


Figure 4.1: An example MSC over $\{p, q, r\}$.

4.1.1 MSCs as labelled partial orders

Through the rest of the chapter, we fix a finite set of processes (or agents) $\mathcal{P}$ and let p, q, r range over $\mathcal{P}$. For each $p \in \mathcal{P}$ we define $\Sigma_p = \{p!q \mid p \neq q\} \cup \{p?q \mid p \neq q\}$ to be the set of communication actions in which p participates. The action $p!q$ is to be read as p *sends to* q and the action $p?q$ is to be read as p *receives from* q. At our level of abstraction, we shall not be concerned with the actual messages that are sent and received. We will also not deal with the internal actions of the agents. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$ and let a, b range over Σ. Also, we denote the set of *channels* by $Ch = \{(p, q) \mid p \neq q\}$ and let c, d range over Ch.

As in the previous chapter, a Σ-labelled poset is a structure $M = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $p \in \mathcal{P}$ and $a \in \Sigma$, we set $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$ and $E_a = \{e \mid \lambda(e) = a\}$, respectively. For each $c \in Ch$, we define the relation $R_c = \{(e, e') \mid \lambda(e) = p!q, \lambda(e') = q?p \text{ and } |{\downarrow}e \cap E_{p!q}| = |{\downarrow}e' \cap E_{q?p}|\}$. Finally, for each $p \in \mathcal{P}$, we define the relation $R_p = (E_p \times E_p) \cap \leq$.

An MSC (over $\mathcal{P}$) is a *finite* Σ-labelled poset $M = (E, \leq, \lambda)$ which satisfies the following conditions:

- Each R_p is a linear order.
- If $p \neq q$ then $|E_{p!q}| = |E_{q?p}|$.
- $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$ where $R_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} R_p$ and $R_{Ch} = \bigcup_{c \in Ch} R_c$.

In diagrams, the events of an MSC are presented in *visual order*. The events of each process are arranged in a vertical line and the members of the relation R_{Ch} are displayed as horizontal or downward-sloping directed edges. We illustrate the idea with the example depicted on Figure 4.1. There $\mathcal{P} = \{p, q, r\}$. For $x \in \mathcal{P}$, the events in E_x are arranged along the line labelled (x) with smaller (relative to $\leq$) events appearing above the larger events. The R_{Ch}-edges across agents are depicted by horizontal edges—for instance $e_3 \; R_{(r,q)} \; e'_2$. The labelling function λ is easy to extract from the diagram—for example, $\lambda(e_1) = p!q$ and $\lambda(e'_2) = q?r$.

Henceforth, we will identify an MSC with its isomorphism class. We let $\mathbb{MSC}(\mathcal{P})$ be the set of MSCs over $\mathcal{P}$. An *MSC language* is a subset $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$. From now on, we will often omit the subscripts and just denote $\mathbb{MSC}(\mathcal{P})$ by $\mathbb{MSC}$ and $\Sigma_{\mathcal{P}}$ by Σ when no confusion arises.

4.1.2 MSCs as sets of strings

We will define regular MSC languages in terms of their linearizations, so we investigate here the relationship between the MSCs and their linearizations.

The notion of linearization, as defined for traces in Section 3.1.1, can be carried over to general Σ-labelled partial orders and, in particular, directly to the setting of MSCs. For an MSC $M \in \mathbb{MSC}$, we let $lin(M)$ denote the set of linearizations of M and set $lin(\mathcal{L}) = \bigcup\{lin(M) \mid M \in \mathcal{L}\}$ for an MSC language $\mathcal{L} \subseteq \mathbb{MSC}$. We will see that it is fruitful to identify MSCs also with their set of linearizations in a manner very similar to the two equivalent representations of traces in Sections 3.1.1 and 3.1.2, respectively. In this way we identify the MSC M in Figure 4.1 with $lin(M) = \{p!q\ q?p\ r!q\ q?r,\ p!q\ r!q\ q?p\ q?r,\ r!q\ p!q\ q?p\ q?r\}$.

In the literature (e.g. [3, 99, 100]) one sometimes considers a more generous notion of linearization where two *adjacent* receive actions in a process corresponding to messages from *different* senders are deemed causally independent. For instance, $p!q\ r!q\ q?r\ q?p$ would also be a valid linearization of the MSC in Figure 4.1. One then refers to the *causal order* of the MSC (as opposed to the visual order). The present results go through with suitable modifications even in the presence of this more generous notion of linearization.

To directly characterize the subsets of Σ^* that correspond to MSC languages, we proceed as follows. Let $Com = \{(p!q, q?p) \mid (p, q) \in Ch\}$. We say that $\sigma \in \Sigma^*$ is *proper* if for every prefix τ of σ and every pair $(a, b) \in Com$, $|\tau|_a \geq |\tau|_b$. We say that σ is *complete* if σ is proper and $|\sigma|_a = |\sigma|_b$ for every $(a, b) \in Com$. Next we define a *context-sensitive* independence relation $I \subseteq \Sigma^* \times (\Sigma \times \Sigma)$ as follows: $(\sigma, a, b) \in I$ if σab is proper, $a \in \Sigma_p$ and $b \in \Sigma_q$ for distinct processes p and q, and if $(a, b) \in Com$ then $|\sigma|_a > |\sigma|_b$. Observe that if $(\sigma, a, b) \in I$ then $(\sigma, b, a) \in I$.

Let $\Sigma^\circ = \{\sigma \mid \sigma \in \Sigma^* \text{ and } \sigma \text{ is complete}\}$. We then define $\sim \subseteq \Sigma^\circ \times \Sigma^\circ$ to be the least equivalence relation such that if $\sigma = \sigma_1 ab \sigma_2$, $\sigma' = \sigma_1 ba \sigma_2$ and $(\sigma_1, a, b) \in I$ then $\sigma \sim \sigma'$. It is important to note that $\sim$ is defined over Σ° (and not Σ^*). It is easy to verify that for each $M \in \mathbb{MSC}$, $lin(M)$ is a subset of Σ° and is in fact a $\sim$-equivalence class over Σ°.

We define $L \subseteq \Sigma^*$ to be a *string MSC language* if there exists an MSC language $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$ such that $L = \bigcup\{lin(M) \mid M \in \mathcal{L}\}$. It is easy to see that $L \subseteq \Sigma^*$ is a string MSC language if and only if L is a subset of Σ^* such that every string in L is complete and L is $\sim$-closed (that is, for each $\sigma \in \Sigma^\circ$, if $\sigma \in L$ and $\sigma \sim \sigma'$ then $\sigma' \in L$).

Clearly MSC languages and string MSC languages represent each other in a precise manner. Below we construct representation maps $\mathsf{sm} : \Sigma^\circ/\sim \to \mathbb{MSC}$ and $\mathsf{ms} : \mathbb{MSC} \to \Sigma^\circ/\sim$ and sketch briefly that these maps are "inverses" of each other.

We first define $\mathsf{sm} : \Sigma^\circ \to \mathbb{MSC}$. Let $\sigma \in \Sigma^\circ$. Then $\mathsf{sm}(\sigma) = (E, \leq, \lambda)$, where

- $E = \{\tau a \mid \tau a \in \mathrm{prf}(\sigma)\}$. (Thus $E = \mathrm{prf}(\sigma) - \{\varepsilon\}$.)
- $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$ where $R_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} R_p$, $R_{Ch} = \bigcup_{c \in Ch} R_c$. The constituent relations are defined as follows. For each $p \in \mathcal{P}$, $(\tau a, \tau' b) \in R_p$

iff $\mathrm{loc}(a) = \mathrm{loc}(b) = p$ and $\tau a \in \mathrm{prf}(\tau' b)$. Moreover, for each $c \in \mathcal{C}h$, $(\tau a, \tau' b) \in R_c$ iff $a = p!q$ and $b = q?p$ for some $p, q \in \mathcal{P}$ and furthermore $|\tau a|_a = |\tau' b|_b$.

- For $\tau a \in E$, $\lambda(\tau a) = a$.

One can show that $\sigma \sim \sigma'$ implies $\mathsf{sm}(\sigma) = \mathsf{sm}(\sigma')$. This observation guarantees that $\mathsf{sm}'([\sigma]_\sim) = \mathsf{sm}(\sigma)$ is well-defined. In fact, we shall henceforth write sm to denote both sm and sm'.

Conversely, we define the map $\mathsf{ms} : \mathbb{MSC} \to \Sigma^\circ/\!\sim$ as: $\mathsf{ms}(M) = \mathit{lin}(M)$ and it is not hard to show that ms is well-defined. We can also show that for every $\tau \in \Sigma^\circ, \mathsf{ms}(\mathsf{sm}(\sigma)) = [\sigma]_\sim$ and for every $M \in \mathbb{MSC}$, $\mathsf{sm}(\mathsf{ms}(M)) = M$. In this sense, this justifies the claim that $\Sigma^\circ/\!\sim$ and $\mathbb{MSC}$ are two equivalent ways of representing the same class of objects. Hence, abusing terminology, we will write "MSC language" to mean "string MSC language". From the context, it should be clear whether we are working with MSCs from $\mathbb{MSC}$ or complete strings over Σ. As a rule of thumb, we will use $\mathcal{L}$ to denote the former and L to denote the latter, but this distinction is not always firm.

We can now finally bring out our notion of regular collections of MSCs. We will say that $\mathcal{L} \subseteq \mathbb{MSC}$ is a *regular MSC language* if the corresponding *string* MSC language is a regular subset of Σ^*. Thus, a language $\mathcal{L}$ of MSCs is regular in case $\mathit{lin}(\mathcal{L})$ is regular in the classical sense. Note that, unlike the settings of strings (or trees or Mazurkiewicz traces), the universe $\mathbb{MSC}$ is itself *not* regular according to our definition. This fact has a strong bearing on the automata-theoretic and logical formulations in our work, as will become evident later in this chapter.

We conclude this section by introducing the notion of B-bounded MSC languages. Let $B \in \mathbb{N}$ be a natural number. We say that a complete string σ is B-bounded if for each prefix τ of σ and for each channel $(p, q) \in \mathcal{C}h$, $|\tau|_{p!q} - |\tau|_{q?p} \leq B$. We say that $L \subseteq \Sigma^\circ$ is B-bounded if every string $\sigma \in L$ is B-bounded. One can show that any regular MSC language is B-bounded for some $B \in \mathbb{N}$. In fact, it is easy to see that a *crude* bound is always given by $|\mathcal{A}_L|$, where $\mathcal{A}_L$ is the minimal DFA accepting L. The optimal bound can however easily be computed from L.

Finally, we shall say that the MSC M is B-bounded if every string in $\mathit{lin}(M)$ is B-bounded. A collection of MSCs is B-bounded if every member of the collection is B-bounded. In the following, we let $\mathbb{MSC}(\mathcal{P}, B)$ be the set of B-bounded MSCs over $\mathcal{P}$.

4.2 Automata over MSCs

In this section we define a class of automata characterizing the class of regular MSC languages proposed in the previous section. In this way we obtain a class of finite-state acceptors which can be used as a basis for carrying over the automata-based verification technique of the previous chapters to the setting of MSC languages.

Our class of automata can be seen as message-passing variations of the asynchronous automata of Section 3.2.2. In fact, recalling the remarks on page 32, the present class of automata perhaps better deserves to be denoted "asynchronous automata".

To bring this out, we say that a *message-passing automaton* over Σ is a structure $\mathcal{M} = (\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \Delta, s_{in}, F)$ where:

- Δ is a finite alphabet of messages.
- Each component $\mathcal{A}_p$ is of the form $(S_p, \longrightarrow_p)$ where
  - S_p is a finite set of p-local states.
  - $\longrightarrow_p \subseteq S_p \times \Sigma_p \times \Delta \times S_p$ is the p-local transition relation.
- $s_{in} \in \prod_{p\in\mathcal{P}} S_p$ is a global initial state.
- $F \subseteq \prod_{p\in\mathcal{P}} S_p$ is a set of global final states.

The local transition relation $\longrightarrow_p$ specifies how the process p sends and receives messages. The transition $(s, p!q, m, s')$ specifies that when p is in the state s, it can send the message m to q (by executing the communication action $p!q$) and go to the state s'. The message m is, as a result, appended to the queue of messages in the channel (p, q). Similarly, the transition $(s, p?q, m, s')$ signifies that at the state s, the process p can receive the message m from q by executing the action $p?q$ and go to the state s'. The message m is removed from the head of the queue of messages in the channel (q, p).

The set of global states of $\mathcal{M}$ is given by $\prod_{p\in\mathcal{P}} S_p$. For a global state s, we let s_p denote the pth component of s. A *configuration* is a pair (s, χ) where s is a global state and $\chi : Ch \to \Delta^*$ is the *channel state* which specifies the queue of messages currently residing in each channel c. The *initial configuration* of $\mathcal{M}$ is $(s_{in}, \chi_\varepsilon)$ where $\chi_\varepsilon(c)$ is the empty string ε for every channel c. The set of *final configurations* of $\mathcal{M}$ is $F \times \{\chi_\varepsilon\}$.

We now define the set of reachable configurations $Conf_{\mathcal{M}}$ and the global transition relation $\Longrightarrow \subseteq Conf_{\mathcal{M}} \times \Sigma \times Conf_{\mathcal{M}}$ inductively as follows:

- $(s_{in}, \chi_\varepsilon) \in Conf_{\mathcal{M}}$.
- Suppose $(s, \chi) \in Conf_{\mathcal{M}}$, (s', χ') is a configuration and $(s_p, p!q, m, s'_p) \in \longrightarrow_p$ such that the following conditions are satisfied:
  - $r \neq p$ implies $s_r = s'_r$ for each $r \in \mathcal{P}$.
  - $\chi'((p, q)) = \chi((p, q)) \cdot m$ and for $c \neq (p, q)$, $\chi'(c) = \chi(c)$.

  Then $(s, \chi) \stackrel{p!q}{\Longrightarrow} (s', \chi')$ and $(s', \chi') \in Conf_{\mathcal{M}}$.
- Suppose $(s, \chi) \in Conf_{\mathcal{M}}$, (s', χ') is a configuration and $(s_p, p?q, m, s'_p) \in \longrightarrow_p$ such that the following conditions are satisfied:
  - $r \neq p$ implies $s_r = s'_r$ for each $r \in \mathcal{P}$.

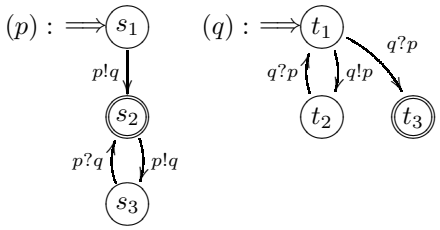


Figure 4.2: A 3-bounded message-passing automaton.

– $\chi((q,p)) = m \cdot \chi'((q,p))$ and for $c \neq (q,p)$, $\chi'(c) = \chi(c)$.

Then $(s,\chi) \stackrel{p?q}{\Longrightarrow} (s',\chi')$ and $(s',\chi') \in \mathit{Conf}_{\mathcal{M}}$.

Let $\sigma \in \Sigma^*$. A run of $\mathcal{M}$ over σ is a map $\rho : \mathrm{prf}(\sigma) \to \mathit{Conf}_{\mathcal{M}}$ such that $\rho(\varepsilon) = (s_{in}, \chi_\varepsilon)$ and for each $\tau a \in \mathrm{prf}(\sigma)$, $\rho(\tau) \stackrel{a}{\Longrightarrow} \rho(\tau a)$. The run ρ is *accepting* if $\rho(\sigma)$ is a final configuration. We define $L(\mathcal{M}) = \{\sigma \mid \mathcal{M} \text{ has an accepting run over } \sigma\}$. It is easy to see that every member of $L(\mathcal{M})$ is complete and $L(\mathcal{M})$ is $\sim$-closed in the sense that if $\sigma \in L(\mathcal{M})$ and $\sigma \sim \sigma'$ then $\sigma' \in L(\mathcal{M})$.

Unfortunately, $L(\mathcal{M})$ need not be regular. Consider, for instance, a message-passing automaton for the canonical producer-consumer system in which the producer p sends an arbitrary number of messages to the consumer q. Since we can reorder all the $p!q$ actions to be performed before all the $q?p$ actions, the queue in channel (p,q) can grow arbitrarily long. Hence, the set of reachable configurations of this system is not bounded and the corresponding language is not regular.

For $B \in \mathbb{N}$, we say that a configuration (s,χ) of the message-passing automaton $\mathcal{M}$ is *B-bounded* if for every channel $c \in \mathit{Ch}$, it is the case that $|\chi(c)| \leq B$. We say that $\mathcal{M}$ is a B-bounded automaton if every reachable configuration $(s,\chi) \in \mathit{Conf}_{\mathcal{M}}$ is B-bounded. It is not difficult to show that given a message-passing automaton $\mathcal{M}$ and a bound $B \in \mathbb{N}$, one can decide whether or not $\mathcal{M}$ is B-bounded.

Figure 4.2 depicts an example of a 3-bounded message-passing automaton with two components, p and q (the message alphabet is a singleton and hence omitted). The automaton accepts the infinite set of MSCs $\mathcal{L} = \{M_i\}_{i=0}^{\omega}$, where M_i is displayed in Figure 4.3 for $i = 2$.

We say that $\mathcal{M}$ is a *bounded* message-passing automaton if $\mathcal{M}$ is B-bounded for some $B \in \mathbb{N}$. We arrive at the following automata-theoretic characterization of the regular MSC languages.

Theorem 4.2.1 *Let $L \subseteq \Sigma^\circ$. Then L is a regular MSC language if and only if there exists a bounded message-passing automaton $\mathcal{M}$ such that $L(\mathcal{M}) = L$.*

The proof of this result goes via an intermediate step by a nontrivial application of Zielonka's Theorem. Furthermore, the construction incorporates a

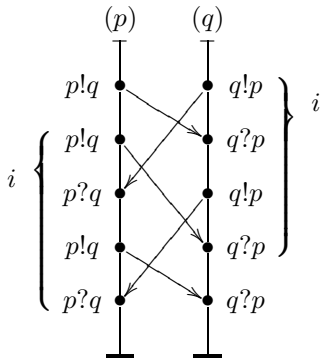


Figure 4.3: The M_i's accepted by the automaton in Figure 4.2.

(message-passing) gossip construction [93] very similar to the one presented in Section 3.2.4 for synchronous communication.

The details of the proof can be found in Chapter 10. Our approach constructs a nondeterministic bounded message-passing automaton for any regular MSC language. This was later refined [94] to construct a *deterministic* bounded message-passing automaton.

4.3 Monadic Second-order logic for MSCs

In Chapter 10 we also formulate a natural monadic second-order logic to be interpreted over MSCs. The logic will characterize regular B-bounded MSC languages for each fixed $B \in \mathbb{N}$. Consequently, our logic will be parameterized by a pair $(\mathcal{P}, B)$ in the same way as the monadic second-order logic of traces in Section 3.3 was parameterized by trace alphabets. We will denote this logic by $\mathrm{MSO}(\mathcal{P}, B)$. For convenience, we fix the set of processes $\mathcal{P}$ and the bound $B \in \mathbb{N}$ throughout the rest of the section.

The syntax of formulas of $\mathrm{MSO}(\mathcal{P}, B)$ is, once again, identical to the original formulation of Section 2.2.1. Thus the syntax does not reflect any information about B or the structural features of an MSC. These aspects will be dealt with in the semantics. The formulas of our logic are interpreted over the members of $\mathbb{MSC}(\mathcal{P}, B)$. Let $M = (E, \leq, \lambda)$ be an MSC in $\mathbb{MSC}(\mathcal{P}, B)$ and $\mathcal{I}$ be an interpretation which assigns to each individual variable a member $\mathcal{I}(x)$ in E and to each set variable X a subset $\mathcal{I}(X)$ of E. Then $M \models_{\mathcal{I}} \varphi$ denotes that M satisfies φ under $\mathcal{I}$. These notions are defined exactly as in Sections 2.2.1 and 3.3.1, with the only change being the way in which the order predicate is interpreted. Formally, $M \models_{\mathcal{I}} x \leq y$ if $\mathcal{I}(x) \leq \mathcal{I}(y)$. Once again, we have used $\leq$ to denote both the predicate symbol in the logic and the corresponding causality relation in the model M.

All other notions are carried over from $\mathrm{MSO}(\Sigma)$ and $\mathrm{MSO}(\Sigma, I)$. In the obvious manner, we can now with each sentence φ associate an MSC language $\mathcal{L}_\varphi = \{M \in \mathbb{MSC}(\mathcal{P}, B) \mid M \models \varphi\}$. We say that $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P}, B)$ is *definable in* $\mathrm{MSO}(\mathcal{P}, B)$ if there exists a sentence φ such that $\mathcal{L}_\varphi = \mathcal{L}$.

It turns out that the techniques used for extending Büchi's Theorem to the setting of Mazurkiewicz traces sketched in Section 3.3.2, can be suitably modified to derive an analogous result in the present setting. The following important result generalizes Büchi's Theorem to the setting of MSC languages.

Theorem 4.3.1 *Let $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$. Then $\mathcal{L}$ is definable in $\mathrm{MSO}(\mathcal{P}, B)$ for some $B \in \mathbb{N}$ if and only if $\mathcal{L}$ is a regular MSC language.*

Once again, the details can be found in Chapter 10. An important corollary of Theorem 4.3.1 is that the translations between $\mathrm{MSO}(\Sigma_\mathcal{P})$ and $\mathrm{MSO}(\mathcal{P}, B)$ are constructive, so the satisfiability problem for $\mathrm{MSO}(\mathcal{P}, B)$ is decidable (albeit only in nonelementary time). Additionally, one can implement a set of macro definitions so that $\mathrm{MSO}(\mathcal{P}, B)$ can be decided directly by the Mona tool described in Chapter 5.

It is not hard to see that Theorem 4.3.1 holds true even if we replace the causal order $\leq$ with the family of communication relations $\{R_c\}_{c \in Ch}$ together with the family of component relations $\{R_p\}_{p \in \mathcal{P}}$. Indeed, these two order notions are interdefinable within $\mathrm{MSO}(\mathcal{P}, B)$. However, the corresponding first-order fragments are not readily expressively equivalent, and this leaves open the question of what constitutes a plausible definition of $\mathrm{FO}(\mathcal{P}, B)$.

Our notion of regularity seems quite natural and, as we have seen, has an accompanying automata-theoretic characterization. Moreover, our notion coincides with definability in the natural monadic second-order logic. Consequently, it is quite robust.

At present, languages of infinite MSCs have not been considered in this context, but our results can be seen as a guideline towards generalizations to the infinite setting. One starting point is to say that a language of infinite MSCs is *regular* in case its linearizations constitute an ω-regular subset of Σ^ω as defined in Section 2.1.2. It is our hope that an extension of Theorem 4.2.1 is possible using message-passing automata with Büchi acceptance conditions. Moreover, we feel confident that this class obviously coincides with definability in (the trivial extension of) monadic second-order logic over infinite MSCs. This paves the way for linear time temporal logics to be interpreted over infinite MSCs with automata-theoretic decision procedures based on message-passing (Büchi) automata, thus generalizing the constructions of Section 2.3.2 and Section 3.4.1.

These issues will be addressed in future research.

4.4 Message Sequence Graphs

The standard method to describe multiple communication scenarios is to generate collections of MSCs by means of Hierarchical Message Sequence Charts (HMSCs). As described in the introduction, to analyze HMSCs, it suffices to

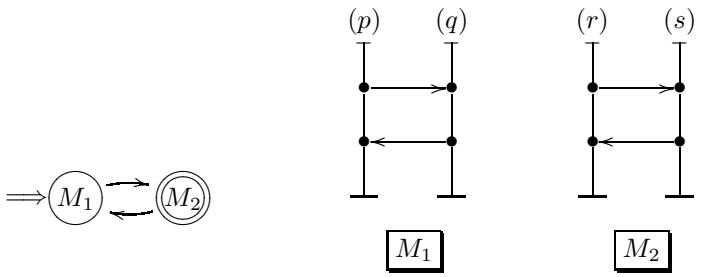


Figure 4.4: An example MSG.

flatten out them out to obtain Message Sequence Graphs (MSGs). As a consequence, henceforth we concentrate on MSGs rather than HMSCs.

An MSG allows the protocol designer to write a finite specification which combines MSCs using basic operations such as branching choice, composition and iteration. Such MSGs are finite directed graphs with designated initial and terminal vertices. Each vertex in an MSG is labelled by an MSC. The edges represent the natural operation of MSC concatenation. The collection of MSCs represented by an MSG consists of all those MSCs obtained by tracing a path in the MSG from an initial vertex to a terminal vertex and concatenating the MSCs that are encountered along the path.

A *Message Sequence Graph (MSG)* is a structure $\mathcal{G} = (Q, \longrightarrow, Q_{in}, F)$, where:

- Q is a finite and nonempty set of states.
- $\longrightarrow \subseteq Q \times Q$.
- $Q_{in} \subseteq Q$ is a set of initial states.
- $F \subseteq Q$ is a set of final states.
- $\Phi : Q \to \mathbb{MSC}(\mathcal{P})$ is a (state-)labelling function.

A *path* π through an MSG $\mathcal{G}$ is a sequence $q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_n$ such that $(q_{i-1}, q_i) \in \longrightarrow$ for $i \in \{1, 2, \ldots, n\}$. The MSC generated by π is $M(\pi) = M_0 \circ M_1 \circ M_2 \circ \cdots \circ M_n$, where $M_i = \Phi(q_i)$. A path $\pi = q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_n$ is a *run* if $q_0 \in Q_{in}$ and $q_n \in F$. The language of MSCs accepted by $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \{M(\pi) \in \mathbb{MSC}(\mathcal{P}) \mid \pi \text{ is a run through } \mathcal{G}\}$.

An example of an MSG is depicted in Figure 4.4. It's not hard to see that the language $\mathcal{L}$ defined is *not* regular. To see this, we note that $\mathcal{L}$ projected to $\{p!q, r!s\}^*$ is $\{\sigma \in \{p!q, r!s\}^* \mid |\sigma|_{p!q} = |\sigma|_{r!s}\}$, which is not a regular string language. (Recall that the regular languages are closed under arbitrary projections.)

A number of studies are available which are concerned with individual MSCs in terms of their semantics and properties [3, 78]. The rest of the work within the area consists of checking specific properties of communication scenarios specified as MSGs. We will briefly survey these results here.

Muscholl, Peled, and Su [100] investigate various problems of matching of both individual MSCs and MSGs and combinations. Here, matching refers to embeddings between the partial orders. More specifically, they show that given MSGs $\mathcal{G}_1$ and $\mathcal{G}_2$, it is NP-complete to decide whether there exist MSCs $M_i \in \mathcal{L}(\mathcal{G}_i)$ such that M_1 matches M_2. Similarly, it is also NP-complete to determine the universal counterpart; does there exists some $M_1 \in \mathcal{L}(\mathcal{G}_1)$ such that M_1 matches every $M_2 \in \mathcal{L}(\mathcal{G}_2)$.

Muscholl [98] goes on to define "and-or" versions of MSGs reminiscent of alternating automata, and shows that the matching problems of deciding, given an "and-or" MSG $\mathcal{G}_1$ and a conventional MSG $\mathcal{G}_2$, whether there exists a run-tree of $\mathcal{G}_1$ such that the MSC of every path matches some $M_2 \in \mathcal{L}(\mathcal{G}_2)$. She shows that this problem in PSPACE-complete, and moreover that a similar problem of MSGs and properties of LTL (which turns out to be PSPACE-complete as well) can essentially be solved by matchings between alternating and Büchi automata.

In [8] Ben-Abdallah and Leue identify and characterize two harmful problems in MSG-specifications and give algorithms to detect such anomalies. The first of them is that of *process divergence* signifying that the specification allows some process to have an unbounded number of unreceived messages in its buffer. This can be detected in time linear in the total number of messages in the specification. We will note here that though related, the notion of divergence-freeness is implied by our notion of regularity, but does *not* coincide with it. Figure 4.4 provides a simple counter-example.

The second underspecification detected by Ben-Abdallah and Leue is that of *nonlocal choice*. Intuitively, this denotes the existence of branching choices where different processes have the possibility of taking conflicting routes in the MSG-specification. To prevent such consistency problems, additional messages or history variables have to be introduced into the system, whence absence of nonlocal choice is a very desirable property of MSGs. Once again, Ben-Abdallah and Leue give an algorithm to detect this which runs in time linear in the total number of messages in the specification.

Alur and Yannakakis [5] consider model checking problems under various semantics for systems modeled as MSGs. Specifications are given as automata accepting the undesireable linearizations, thus describing the *complement* of the intended behaviour. In this manner the decision problems essentially boil down to emptiness checking of products of automata, as the constituent automata need not be complemented. They show that for synchronous concatenations of MSCs on the paths of the MSG, the problem is coNP-complete, while the problem is undecidable in general for asynchronous concatenation.

Following this negative result, they then define the notion[1] of *local syn-*

[1]This notion was introduced as "bounded" in [5], but we prefer the terminology "locally synchronized" of [99] to avoid unnecessary overloading of nomenclature.

chronicity of MSGs. For an MSC $M = (E, \leq, \lambda)$, let CG_M, *the communication graph of* M, be the directed graph $(\mathcal{P}, \mapsto)$ where $(p, q) \in \mapsto$ iff there exists an $e \in E$ with $\lambda(e) = p!q$. M is said to be *connected* if CG_M consists of one nontrivial strongly connected component and isolated vertices. A *loop* in $\mathcal{G}$ is a sequence of edges that starts and ends at the same node. We say that $\mathcal{G}$ is *locally synchronized* if for every loop $\pi = q \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q$, the MSC $M(\pi)$ is connected. An MSC language $\mathcal{L}$ is a *locally synchronized MSG-language* if there exists a locally synchronized MSG $\mathcal{G}$ with $\mathcal{L} = \mathcal{L}(\mathcal{G})$.

Alur and Yannakakis [5] then show that, interestingly, the asynchronous model checking problem becomes PSPACE-complete for *locally synchronized* MSGs. Clearly, the MSG of Figure 4.4 is *not* locally synchronized. This is no coincidence, as it follows as a corollary of their proof sketch that every locally synchronized MSG-language is indeed regular.

We conclude this section by pointing out that Muscholl and Peled [99] consider two decision problems also related to locally synchronized MSGs. The first such problem is that of *race conditions* which can essentially be formulated as the question whether the causal order allows more linearizations than the visual order. Recalling our discussion of this issue in Sections 4.1.1 and 4.1.2 we see that on Figure 4.1 there is a race on process q between the receive events for the messages from p and r, respectively.

The other problem is to detect *confluence* of MSG specifications. An MSG $\mathcal{G}$ is said to be confluent in case for any two prefixes M_1, M_2 of MSCs in $\mathcal{L}(\mathcal{G})$ that are consistent (in the sense that both are prefixes of some common MSC), there does indeed exist such a completed MSC M in $\mathcal{L}(\mathcal{G})$ of which both M_1 and M_2 is a prefix. Muscholl and Peled show that both problems of deciding whether an MSG has race conditions or is confluent are undecidable for general MSGs. However, they emphasize the importance of local synchronicity by additionally proving that both problems are EXPSPACE-complete for *locally synchronized* MSGs.

In the next section we will look at the closely related notion of finitely generated MSC languages.

4.5 Finitely Generated MSC Languages

A key feature of MSG languages is that for each such language there is a fixed finite set $\mathcal{X}$ of MSCs such that each MSC in the language can be expressed as a concatenation of MSCs (with multiple copies) taken from $\mathcal{X}$. We say that they are finitely generated. In this section we investigate the important connection between MSGs and finitely generated regular MSC languages.

To this end, we let $M_1 = (E_1, \leq_1, \lambda_1)$ and $M_2 = (E_2, \leq_2, \lambda_2)$ be a pair for MSCs such that E_1 and E_2 are disjoint. For $i \in \{1, 2\}$, let R_c^i and $\{R_p^i\}_{p \in \mathcal{P}}$ denote the underlying communication and process causality relations in M_i. The *(asynchronous) concatenation* of M_1 and M_2 yields the MSC $M_1 \circ M_2 = (E, \leq, \lambda)$ where $E = E_1 \cup E_2$, $\lambda(e) = \lambda_i(e)$ if $e \in E_i$, $i \in \{1, 2\}$, and $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$, where $R_p = R_p^1 \cup R_p^2 \cup \{(e_1, e_2) \mid e_1 \in E_1, e_2 \in E_2, \lambda(e_1) \in$

$\Sigma_p, \lambda(e_2) \in \Sigma_p\}$ for $p \in \mathcal{P}$, and $R_c = R_c^1 \cup R_c^2$ for $c \in Com$.

Let $\mathcal{L}_1, \mathcal{L}_2 \subseteq \mathbb{MSC}$ be two sets of MSCs. As usual, $\mathcal{L}_1 \circ \mathcal{L}_2$ denotes the pointwise concatenation of $\mathcal{L}_1$ and $\mathcal{L}_2$ given by $\{M \mid \exists M_1 \in \mathcal{L}_1, M_2 \in \mathcal{L}_2 : M = M_1 \circ M_2\}$. For $\mathcal{X} \subseteq \mathbb{MSC}$, we define $\mathcal{X}^0 = \{\varepsilon\}$, where ε denotes the empty MSC, and for $i \geq 0$, $\mathcal{X}^{i+1} = \mathcal{X} \circ \mathcal{X}^i$. The *asynchronous iteration of* $\mathcal{X}$ is then defined by $\mathcal{X}^{\circledast} = \bigcup_{i \geq 0} \mathcal{X}^i$. Now, let $\mathcal{L} \subseteq \mathbb{MSC}$. We say that $\mathcal{L}$ is *finitely generated* if there is a finite set of MSCs $\mathcal{X} \subseteq \mathbb{MSC}$ such that $\mathcal{L} \subseteq \mathcal{X}^{\circledast}$.

We first observe that not every regular MSC language is finitely generated. As an example, the automaton in Figure 4.2 accepts a regular language which is *not* finitely generated. By inspection of Figure 4.3 one readily verifies that none of the MSCs in this language can be expressed as the concatenation of two or more nontrivial MSCs. Hence, this language is *not* finitely generated.

Our interest in finitely generated languages stems from the fact that these arise naturally from standard high-level descriptions of MSC languages such as message sequence graphs. However, as we saw earlier, Figure 4.4 provides an example showing that, conversely, not all finitely generated languages are regular.

Chapter 11 investigates this correspondence in more detail. We show that it is decidable whether a given regular MSC language $\mathcal{L}$ is finitely generated. Conversely, we show that it is undecidable whether a given MSG generates a regular MSC language.

Consequently, it would be nice to have a characterization of the finitely generated regular MSC languages. Alur and Yannakakis' notion of local synchronicity of MSGs supplies a sufficient, but not necessary condition for a language of an MSG to be regular. However, the main result of Chapter 11 is the following theorem.

Theorem 4.5.1 *Let $\mathcal{L}$ be an MSC language. Then $\mathcal{L}$ is a finitely generated regular MSC language if and only if $\mathcal{L}$ is a locally synchronized MSG-language.*

Another important way to phrase this main characterization result is:

Corollary 4.5.2 *Let $\mathcal{L}$ be a regular MSC language. Then $\mathcal{L}$ can be described by an MSG if and only if $\mathcal{L}$ is finitely generated.*

4.6 Regular MSC Expressions

We have defined a robust rotion of regularity of collections of MSCs with characterizations in terms of both bounded message-passing automata and a natural monadic second-order logic. The only remaining characterization is to extend the celebrated theorems of Kleene and Ochmański (Theorems 2.1.1 and 3.2.5, respectively) to the setting of MSCs.

Carrying over directly the original definition of Kleene in Section 2.1.1,

$$\mathrm{RE}(\mathcal{P}) ::= M \mid \pi_0 + \pi_1 \mid \pi_0 ; \pi_1 \mid \pi^{\circledast} \ , \quad M \in \mathbb{MSC}(\mathcal{P}),$$

unfortunately goes beyond the regular languages. In fact, it is easy to see that the (nonregular!) language described by the MSG in Figure 4.4 can also be defined by the expression $(M_1 \circ M_2)^{\circledast}$. Indeed, one can verify that this class of regular expressions defines precisely the set of languages corresponding to HMSCs (or MSGs).

Once again, as for Mazurkiewicz traces, the iteration-operator is too powerful. An obvious approach would then be to mimic the "connected" iteration-operator defined for traces by Ochmański, and restrict $^{\circledast}$ to languages connected in the sense of Section 4.4. While the collections of languages definable by this class of regular expressions remain regular, one can show that such expressions are expressively equivalent to locally synchronized MSGs. As we have seen already, it follows from Figures 4.2 and 4.3 and Theorem 4.5.1 that this class is *too restrictive* to capture the regular MSC languages.

In conclusion, it remains a challenging open problem to exhibit a class of regular expressions characterizing our notion of regularity. As all other operations preserve finitely generatedness, the right iteration-operator must be able to construct infinitely many atoms from only finitely many MSCs. A natural such iteration-operator has yet to be defined...

Part II

Papers

Chapter 5

Mona: Monadic Second-order Logic in Practice

Contents

We introduce monadic second-order logic of finite strings as a practical means of specifying regularity. The logic is a highly succinct alternative to the use of regular expressions. We have built a tool Mona, which acts as a decision procedure and as a translator to finite-state automata. The tool is based on new algorithms for minimizing finite-state automata that use binary decision diagrams (BDDs) to represent transition functions in compressed form. A byproduct of this work is an algorithm that matches the time but improves the space of Sieling and Wegener's algorithm to reduce OBDDs in linear time.

The potential applications are numerous. We discuss text processing, Boolean circuits, and distributed systems. Our main example is an automatic proof of properties for the "Dining Philosophers with Encyclopedia" example by Kurshan and MacMillan. We establish these properties for the parameterized case *without* the use of induction.

Our results show that, contrary to common beliefs, high computational complexity may be a desired feature of a specification formalism.

5.1 Introduction

In computer science, *regularity* amounts to the concept that a class of structures is recognized by a finite-state device. Often phenomena are so complicated that their regularity either

- may be overlooked, as in the case of parameterized verification of distributed finite-state systems with a regular communication topology; or
- may not be exploited, as in the case when a search pattern in a text editor is known to be regular, but in practice inexpressible as a regular expression.

We argue that the *Monadic Second-Order Logic* or *MSO* can help in practice to identify and to use regularity. In MSO, one can directly mention positions and subsets of positions in the string. This feature distinguishes the logic from regular expressions or automata. Together with quantification and Boolean connectives, an extraordinarily succinct formalism arises.

Although it has been known for thirty-five years that MSO defines regular languages (see [140]), the translator from formulas to automata that we describe in this article appears to be one of the first implementations.

The reason such projects have not been pursued may be the staggering theoretical lower-bound: any decision procedure is bound to sometimes require as much time as a stack of exponentials that has height proportional to the length of the formula.

It is often believed that the lower the computational complexity of a formalism is, the more useful it may be in practice. We want to counter such beliefs in this article — at least for logics on finite strings.

5.1.1 Why use logic?

Some simple finite-state languages easily described in English call for convoluted regular expressions. For example, the language L_{2a2b} of all strings over $\Sigma = \{a, b, c\}$ containing at least two occurrences of a *and* at least two occurrences of b seems to require a voluminous expression, such as

$$\begin{array}{ll} & \Sigma^*a\Sigma^*a\Sigma^*b\Sigma^*b\Sigma^* \\ \cup & \Sigma^*a\Sigma^*b\Sigma^*a\Sigma^*b\Sigma^* \\ \cup & \Sigma^*a\Sigma^*b\Sigma^*b\Sigma^*a\Sigma^* \\ \cup & \Sigma^*b\Sigma^*b\Sigma^*a\Sigma^*a\Sigma^* \\ \cup & \Sigma^*b\Sigma^*a\Sigma^*b\Sigma^*a\Sigma^* \\ \cup & \Sigma^*b\Sigma^*a\Sigma^*a\Sigma^*b\Sigma^*. \end{array}$$

If we added $\cap$ to the operators for forming regular expressions, then the language L_{2a2b} could be expressed more concisely as $(\Sigma^*a\Sigma^*a\Sigma^*) \cap (\Sigma^*b\Sigma^*b\Sigma^*)$. Even

with this extended set of operators, it is often more convenient to express regular languages in terms of positions and corresponding letters. For example, to express the set $L_{a\text{after}b}$ of strings in which every b is followed by an a, we would like a formal language allowing us to write something like

> "for every position p, if there is a b in p then for some position q after p, there is an a in q."

The extended regular languages do not seem to allow an expression that very closely reflects this description — although upon some reflection a small regular expression can be found. But in MSO we can express $L_{a\text{after}b}$ by a formula

$$\forall p : 'b'(p) \;\Rightarrow\; \exists q : \; p < q \;\wedge\; 'a'(q)$$

(Here the predicate $'b'(p)$ means "there is a b in position p".) In general, we believe that many errors can be avoided if logic is used when the description in English does not lend itself to a direct translation into regular expressions or automata. However, the logic can easily be combined with other methods of specifying regularity since almost any such formalism can be translated with only a linear blow-up into MSO.

Often regularity is identified by means of *projections*. For example, if L_{trans} is regular on a cross-product alphabet $\Sigma \times \Sigma$ (e.g. describing a parameterized transition relation, see Section 5.5 and L_{start} is a regular language on Σ describing a set of start strings, then the set of strings that can be reached by a transition from a start string is $\pi_2(L_{trans} \cap \pi_1^{-1}(L_{start}))$, where π_1 and π_2 are the projections from $(\Sigma \times \Sigma)^*$ to the first and second component. Such language-theoretic operations can be very elegantly expressed in MSO.

5.1.2 Our results

In this article, we discuss applications of MSO to text processesing and the description of parameterized Boolean circuits. Our principal application is a new proof technique for establishing properties about parameterized, distributed finite-state systems with regular communication topology. We illustrate our method by showing safety and liveness properties for a non-trivial version of the Dining Philosophers' problem as proposed in [77] by Kurshan and MacMillan.

We present Mona, which is our tool that translates formulas in MSO to finite-state machines. We show how BDDs can be used to overcome an otherwise inherent problem of exponential explosion. Our minimization algorithm works very fast in practice thanks to a simple generalization of the unary apply operation of BDDs.

5.1.3 Comparisons to other work

Parameterized circuits are described using BDDs in [49]. This method relies on formulating inductive steps as finite-state devices and does not provide a single specification language. The work in [120] is closer in spirit to our method

in that languages of finite strings are used although not as part of a logical framework. In [6], another approach is given based on iterating abstractions. The parameterized Dining Philosopher's problem is solved in [77] by a finite-state induction principle.

A tool for MSO on finite, binary trees has been developed at the University of Kiel [129]. Apparently, this tool has only been used for very simple examples.

In [22], a programming language for finite domains based on a fixed point logic is described and used for verification of non-parameterized finite systems.

5.1.4 Contents

In Section 5.2 we explain the syntax and semantics of MSO for finite strings. We recall the correspondence to automata theory in Section 5.3. We give several applications of MSO and the tool in Section 5.4: text patterns, parameterized circuits, and equivalence testing. Our main example of parameterized verification is discussed in Section 5.5. We give an overview of our implementation in Section 5.6.

5.2 The Monadic Second-order Logic on Strings

Let Σ be an alphabet and let $w \in \Sigma^*$ be a string over Σ. The semantics of the logic determines whether a closed MSO formula φ holds on w. The language $L(\varphi)$ denoted by φ is the set of strings that make φ hold. Assume now that w has length n and consists of letters $a_0 a_1 \cdots a_{n-1}$. The *positions* in w are then $0, \ldots, n-1$.

The syntax of MSO, to be presented here, is slightly different from the exposition in Part I, but the present syntax is derivable from the minimalistic syntax of Section 2.2.1.

We can now describe the three syntactic categories of MSO on strings. A *position term* t is either

- the constant 0 (which denotes the position 0);
- the constant \$ (which denotes the last position, that is $n-1$);
- a position variable p (which denotes a position i);
- of the form $t \oplus i$ (which denotes the position $j + i \bmod n$, where j is the interpretation of t); or
- of the form $t \ominus i$ (which denotes the position $j - i \bmod n$, where j is the interpretation of t);

(Position terms are only interpreted for non-empty strings).
A *position set term* T is either

- the constant $\emptyset$ (which denotes the empty set);

- the constant **all** (which denotes the set $\{0, ..., n-1\}$);
- a position set variable P (which denotes a subset of positions);
- of the form $T_1 \cup T_2$, $T_1 \cap T_2$, or $\complement T_1$ (which are interpreted in the natural way);
- of the form $T + i$ (which denotes the set of positions in T shifted right by an amount of i); or
- of the form $T - i$ (which denotes the set of positions in T shifted left by an amount of i);

A *formula* φ is either of the form

- $'a'(t)$ (which holds if letter a_i in $w = a_0 a_1 \cdots$ is a, where i is the interpretation of t);
- $t_1 = t_2$, $t_1 < t_2$ or $t_1 \leq t_2$ (which are interpreted in the natural way);
- $T_1 = T_2$, $T_1 \subseteq T_2$, or $t \in T$ (which are interpreted in the natural way);
- $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, or $\varphi_1 \Leftrightarrow \varphi_2$ (where φ_1 and φ_2 are formulas, and which are interpreted in the natural way);
- $\exists p : \varphi$ (which is true, if there is a position i such that φ holds when i is substituted for p);
- $\forall p : \varphi$ (which is true, if for all positions i, φ holds when i is substituted for p);
- $\exists P : \varphi$ (which is true, if there is a subset of positions I such that φ holds when I is substituted for P); or
- $\forall P : \varphi$ (which is true, if for all subsets of positions I, φ holds when I is substituted for P);

5.3 From MSO to Automata

In this section, we recall the method for translating a formula in MSO to an equivalent finite-state automaton (see [140] for more details). Note that any formula φ can be interpreted, given a string w and a *value assignment* $\mathcal{I}$ that fixes values of the free variables. If φ then holds, we write $w, \mathcal{I} \models \varphi$. The key idea is that a value assignment and the string may be described together as a word over an extended alphabet consisting of Σ and extra binary tracks, one for each variable. By structural induction, we then define for each formula an automaton that exactly recognizes the words in the extended alphabet corresponding to pairs consisting of a string and an assignment that satisfy the formula.

Example 5.3.1 Assume that the free variables are $\mathcal{P} = \{P_1, P_2\}$ and that $\Sigma = \{a, b\}$. Let us consider the string $w = abaa$ and value assignment

$$\mathcal{I} = [P_1 \mapsto \{0, 2\}, P_2 \mapsto \emptyset].$$

The set $\mathcal{I}(P_1) = \{0, 2\}$ can be represented by the bit pattern 1010, since the numbered sequence

1	0	1	0
0	1	2	3

defines that 0 is in the set (the bit in position 0 is 1), 1 is not in the set (the bit in position 1 is 0), etc. Similarly, the bit pattern 0000 describes $\mathcal{I}(P_2) = \emptyset$.

If these patterns are laid down as extra "tracks" along w, we obtain an *extended word* α, which may be depicted as:

a	b	a	a
1	0	1	0
0	0	0	0

Technically, we define $\alpha = \alpha_0 \cdots \alpha_3$ as the word $(a,1,0)(b,0,0)(a,1,0)\,(a,0,0)$ over the alphabet $\Sigma \times \mathbb{B} \times \mathbb{B}$ of *extended letters*, where $\mathbb{B} = \{0, 1\}$ is the set of truth values.

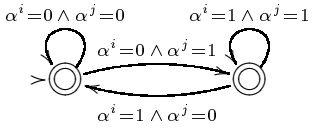
This correspondence can be generalized to any w and any value assignment for a set of variables $\mathcal{P}$ (which can all be assumed to be second-order).

By structural induction on formulas, we construct automata $\mathcal{A}^{\varphi,\mathcal{P}}$ over alphabet $\Sigma \times \mathbb{B}^k$—where $\mathcal{P} = \{P_1, \cdots, P_k\}$ is any set of variables containing the free variables in φ—satisfying the *fundamental correspondence*:

$$w, \mathcal{I} \models \varphi \text{ iff } (w, \mathcal{I}) \in L(\mathcal{A}^{\varphi,\mathcal{P}}).$$

Thus $\mathcal{A}^{\varphi,\mathcal{P}}$ accepts exactly the pairs $(w, \mathcal{I})$ that make φ true.

Example 5.3.2 Let φ be the formula $P_i = P_j + 1$. Thus when φ holds, P_i is represented by the same bit pattern as that of P_j but shifted right by one position. This can be expressed by the automaton $\mathcal{A}^{\varphi,\mathcal{P}}$:



In this drawing, α^i refers to the ith extra track. Thus, the automaton checks that the ith track holds the same bit as the jth track the instant before.

5.4 Applications

5.4.1 Text patterns

The language L_{2a2b} of strings containing at least two occurrences of a and two occurrences of b can be described in MSO by the formula

$$(\exists p_1, p_2 : 'a'(p_1) \ \wedge \ 'a'(p_2) \ \wedge \ p_1 \neq p_2) \wedge \\ (\exists p_1, p_2 : 'b'(p_1) \ \wedge \ 'b'(p_2) \ \wedge \ p_1 \neq p_2)$$

Our translator yields the minimal automaton, which contains nine states, in a fraction of a second.

The language $L_{a\text{after}b}$ given by the formula

$$\forall p : 'b'(p) \ \Rightarrow \ \exists q : \ p < q \ \wedge \ 'a'(q)$$

is translated to the minimal automaton, which has two states, in .3 seconds.

A far more complicated language to express is $L_{<1\text{apart}}$ consisting of every string over $\{a, b\}$ such that for any prefix the number of a's and b's are at most one apart. When using regular expressions or MSO, one needs to struggle a bit, but in MSO there is a strategy for describing the functioning of the finite-state machine that comes to mind.

We observe that a position p may be used to designate a prefix; for example, 0 denotes the prefix consisting of the first letter and \$ (the last position) denotes the whole input string. We may now recognize a string in $L_{<1\text{apart}}$ by identifying three sets of positions: the set P_0 corresponding to prefixes with an equal number of a's and b's, the set P_{+1} corresponding to prefixes where the number of a's is one greater than the number of b's, and the set P_{-1} corresponding to prefixes where the number of a's is one less than the number of b's:

$$\begin{aligned} \exists P_0, P_{+1}, P_{-1} : P_0 \cup P_{+1} \cup P_{-1} = \mathbf{all} \\ \wedge \ 0 \notin P_0 \\ \wedge \ 0 \in P_{+1} \Leftrightarrow 'a'(0) \\ \wedge \ 0 \in P_{-1} \Leftrightarrow 'b'(0) \\ \wedge \ \forall p : (p > 0 \ \Rightarrow \\ \qquad p \in P_0 \ \Leftrightarrow ('a'(p) \ \wedge \ p \ominus 1 \in P_{-1}) \\ \qquad\qquad \vee \ ('b'(p) \ \wedge \ p \ominus 1 \in P_{+1}) \\ \qquad \wedge \ p \in P_{+1} \ \Leftrightarrow \ 'a'(p) \ \wedge \ p \ominus 1 \in P_0 \\ \qquad \wedge \ p \in P_{-1} \ \Leftrightarrow \ 'b'(p) \ \wedge \ p \ominus 1 \in P_0) \end{aligned}$$

The resulting four-state automaton is calculated in a fraction of a second.

Text editors and operating systems often allow the user to the select pieces of text according to a *pattern*. For example, in Emacs we may specify text that contains two words separated by a "," using a regular expression

```
\w+ , \w+
```

where \w matches a character in a word, + makes the preceeding expression match one or more times and "," matches a comma.

Unfortunately, it is often exceedingly cumbersome to specify search patterns using regular expressions. Let us consider an example of modifying all declarations `int* ch` within structure declarations except if the structure declaration is within a procedure declaration. Thus in

```
struct {
  int* ch;
  int   x;
} inf;

void foo() {
  int y;
  struct {int* ch} inf;
  . . .
}
```

we want the first occurrence only of `int* ch` to be replaced. Let us express the problem in logic. Say that an *interval* I is a set of contiguous positions in the text. An interval can be designated by the lowest and highest position it contains, so intervals can easily be treated in MSO. Assume we have already defined predicates *struct_decl*, *proc_decl*, and *int_ch* such that

- *struct_decl* (I) is satisfied if and only if the interval I holds a structure declaration,
- *proc_decl* (I) is satisfied if and only if the interval I holds a procedure declaration, and
- *int_ch* (I) is satisfied if and only if the interval I holds the string "`int* ch`".

We can write *struct_decl* and *proc_decl* in MSO under the assumption that the nesting of braces is bounded. Then an interval I holds a declaration to be replaced if and only if the formula

$$\mathit{int\_ch}(I) \wedge \exists J : J \supseteq I \wedge \mathit{struct\_decl}(J) \wedge \neg(\exists K : K \supsetneq J \wedge \mathit{proc\_decl}(K)) \quad (*)$$

is satisfied. This formula states that I must hold an `int* ch` declaration and that there must be some interval J containing I such that J holds a structure declaration but is not properly contained in any interval K containing a procedure declaration.

5.4.2 Parameterized circuits

Assume that we are given a drawing as in Figure 5.1 denoting a parameterized Boolean function.

How do we describe the language $L_{ex} \subseteq \mathbb{B}^*$ of input bit patterns that make the output true? From the drawing, no immediate description as a regular

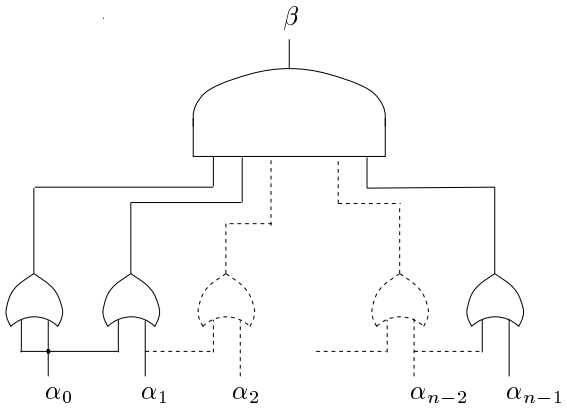


Figure 5.1: A parameterized circuit.

expression or finite-state automaton is apparent. In MSO, however, it is easy to model the outputs of the n or-gates as a second-order variable Q, which allows the language to be described from a direct interpretation of the drawing.

Note that the or-gate at position $p > 0$ is true if either there is a 1 at $p-1$ or p, or in other words: $p \in Q \Leftrightarrow {}'1'(p \ominus 1) \vee {}'1'(p)$. Since the output is 1 if and only if all or-gates are 1, that is if $Q = \mathbf{all}$, the language L_{ex} is given by the formula

$$\begin{aligned}\exists Q : (\forall p : \quad & (p = 0 \Rightarrow p \in Q \Leftrightarrow {}'1'(p)) \wedge \\ & (p > 0 \Rightarrow (p \in Q \Leftrightarrow {}'1'(p \ominus 1) \vee {}'1'(p))) \wedge Q = \mathbf{all})\end{aligned}$$

The resulting automaton has three states and accepts the language $(1 \cup 10)^*$, which is the regular expression that one would obtain by reasoning about the circuit. For more advanced applications to hardware verification, see [7].

5.4.3 Equivalence testing

A closed formula φ is a *tautology* if $L(\varphi) = L(\Sigma^*)$, that is if all strings over Σ satisfy φ. The equivalence of formulas φ and ψ then amounts to whether $\varphi \Leftrightarrow \psi$ is a tautology.

Example 5.4.1 That a set P contains exactly the even positions in a non-empty input string may be expressed in MSO by the following two rather different approaches: either by the formula $\mathit{even1}(P) \stackrel{\text{def}}{=}$

$$\begin{aligned}0 \in P \ \wedge \ \forall p : ((p \in P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \notin P) \\ \wedge (p \notin P \ \wedge \ p < \$ \Rightarrow p \oplus 1 \in P)),\end{aligned}$$

or as a formula $\mathit{even2}(P) \stackrel{\text{def}}{=}$

$$P \cup (P+1) = \mathbf{all} \ \wedge \ P \cap (P+1) = \emptyset \ \wedge \ P \neq \emptyset$$

To show the equivalence of the two formulas, we check the truth value of the bi-implication:

$$\forall P : even1(P) \Leftrightarrow even2(P)$$

The translation of this formula does indeed produce an automaton accepting Σ^*, and thus verifies our claim.

5.5 Dining Philosophers with Encyclopedia

A distributed system is *parameterized* when the number n of processes is not fixed a priori. For such systems the state space is unbounded, and thus traditional finite-state verification methods cannot be used. Instead, one often fixes n to be, say two or three. This yields a finite state space amenable to state exploration methods. However, the validity of a property for $n = 2$, 3 does not necessarily imply that the property holds for all n.

A central problem in verification is automatically to validate parameterized systems. One way to attack the problem is to formulate induction principles such that the base case and the inductive steps can be formulated as finite-state problems. Kurshan and MacMillan [77] used such a method to verify safety and liveness properties of a non-trivial version of the Dining Philosophers example.

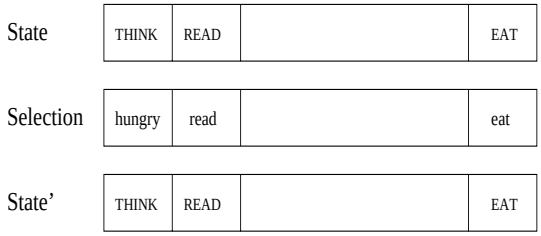


Figure 5.2: Dining Philosophers with Encyclopedia.

In this system, symmetry is broken by an encyclopedia that circulates among the philosophers. Thus each philosopher is in one of three states: EAT, THINK, or READ. The global state can be described as a string *State* of length n over the alphabet $\Sigma_{\text{State}} = \{\text{EAT}, \text{THINK}, \text{READ}\}$, see Figure 5.2.

The system makes a transition according to external events that constitute a *selection* . Each process is presented with an event in the alphabet

$$\Sigma_{\text{Selection}} = \{\text{eat}, \text{think}, \text{read}, \text{hungry}\}.$$

Thus the selection can be viewed as a string *Selection* over $\Sigma_{\text{Selection}}$, see Figure 5.2. As shown, all processes make a synchronous transition to a new global $State'$ on a selection according to a transition relation

$$\text{trans}(State, State', Selection),$$

which is shown in Figure 5.3 (We use '#' in the beginning of a line to indicate that this line is a comment.) together with an auxiliary predicate blocking (*Selection*) used in its definition. Thus the new state of each process is dependent on its old state and on the selection events presented to itself and its neighbors. The transition relation is so complicated that it is hard to grasp the functioning of the system.

Fortunately, the parameterized transition relation can be translated into basic MSO on strings. For example, we encode *State* using two second-order variables P and Q with the convention that

$$\begin{aligned}
&\mathrm{EAT}_p(State) \stackrel{\text{def}}{=} p \in P \ \wedge \ p \in Q \\
&\mathrm{READ}_p(State) \stackrel{\text{def}}{=} p \notin P \ \wedge \ p \in Q \\
&\mathrm{THINK}_p(State) \stackrel{\text{def}}{=} p \notin P \ \wedge \ p \notin Q
\end{aligned}$$

Similarly, $State'$ and *Selection* can also each be encoded using two second-order variables. Thus, the predicate $\mathrm{trans}(State, State', Selection)$ becomes a formula with six free second-order variables.

For this distributed system there are two important properties to verify:

- *Safety Property*: The encyclopedia is neither lost nor replicated. Thus there is always exactly one process in state READ.
- *Liveness Property*: If no process remains in state EAT forever, then the encyclopedia is passed around over and over.

In [77] both properties are proved in terms of a complicated induction hypothesis. This hypothesis is itself a distributed system, where each process has four states. (The Liveness Property in [77] is technically different since it is modeled in terms of selections.)

Our strategy is fundamentally different. We cannot directly verify liveness properties. But we can easily verify properties about the transition relation in the parameterized case and *without* induction as follows.

Let φ be an MSO formula about the global state. For example, we might consider the property that if a philosopher eats, then his neighbors do not:

$$\varphi_{\mathrm{mutex}}(State) \stackrel{\text{def}}{=} \forall p : \mathrm{EAT}_p(State) \Rightarrow \neg\mathrm{EAT}_{p\ominus 1}(State) \ \wedge \ \neg\mathrm{EAT}_{p\oplus 1}(State)$$

A property given as a formula φ can be verified using the *invariance principle*:

$$\begin{aligned}
&\forall State, State', Selection : \\
&\quad \varphi(State) \ \wedge \ \mathrm{trans}(State, State', Selection) \Rightarrow \varphi(State'),
\end{aligned}$$

which is also a formula in MSO. In this way, we have verified for the parameterized case that both φ_{mutex} and the Safety Property that exactly one philosopher reads, that is $\exists! p : \mathrm{READ}_p(State)$, are invariant. Mona verifies such a formula in approximately 3 seconds on a Sparc 20.

Note that this method does not rely on a state space exploration (which is impossible since the state space is unbounded). Instead, it is based on the

$$\begin{array}{l}
\text{blocking}(\mathit{Selection}) \stackrel{\text{def}}{=} \\
\text{eat}_{p\oplus 1}(\mathit{Selection}) \ \vee \ \text{hungry}_{p\ominus 1}(\mathit{Selection}) \\
\vee \ \text{eat}_{p\ominus 1}(\mathit{Selection}) \\
\\
\text{trans}(\mathit{State}, \mathit{State}', \mathit{Selection}) \stackrel{\text{def}}{=} \\
\forall p : \\
\\
\#\text{THINK} \rightarrow \text{THINK} : \\
(\text{THINK}_p(\mathit{State}) \ \wedge \ \text{THINK}_p(\mathit{State}') \Rightarrow \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \neg(\text{read}_{p\ominus 1}(\mathit{Selection})) \\
\vee \\
\text{hungry}_p(\mathit{Selection}) \ \wedge \ \text{blocking}(\mathit{Selection})) \\
\\
\wedge \\
\#\text{THINK} \rightarrow \text{EAT} : \\
(\text{THINK}_p(\mathit{State}) \ \wedge \ \text{EAT}_p(\mathit{State}') \Rightarrow \\
\text{hungry}_p(\mathit{Selection}) \ \wedge \ \neg(\text{blocking}(\mathit{Selection}))) \\
\\
\wedge \\
\#\text{THINK} \rightarrow \text{READ} : \\
(\text{THINK}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \text{read}_{p\ominus 1}(\mathit{Selection})) \\
\\
\wedge \\
\#\text{EAT} \rightarrow \text{THINK} : \\
(\text{EAT}_p(\mathit{State}) \ \wedge \ \text{THINK}_p(\mathit{State}') \Rightarrow \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \neg(\text{read}_{p\ominus 1}(\mathit{Selection}))) \\
\\
\wedge \\
\#\text{EAT} \rightarrow \text{EAT} : \\
(\text{EAT}_p(\mathit{State}) \ \wedge \ \text{EAT}_p(\mathit{State}') \Rightarrow \\
\text{eat}_p(\mathit{Selection})) \\
\\
\wedge \\
\#\text{EAT} \rightarrow \text{READ} : \\
(\text{EAT}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
\text{think}_p(\mathit{Selection}) \ \wedge \ \text{read}_{p\ominus 1}(\mathit{Selection})) \\
\\
\wedge \\
\#\text{READ} \rightarrow \text{THINK} : \\
(\text{READ}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
\text{read}_p(\mathit{Selection}) \ \wedge \ \text{think}_{p\oplus 1}(\mathit{Selection})) \\
\\
\wedge \\
\#\text{READ} \rightarrow \text{EAT} : \\
(\text{READ}_p(\mathit{State}) \ \wedge \ \text{EAT}_p(\mathit{State}') \Rightarrow \\
\text{false}) \\
\\
\wedge \\
\#\text{READ} \rightarrow \text{READ} : \\
(\text{READ}_p(\mathit{State}) \ \wedge \ \text{READ}_p(\mathit{State}') \Rightarrow \\
\text{read}_p(\mathit{Selection}) \ \wedge \ \neg(\text{think}_{p\oplus 1}(\mathit{Selection})))
\end{array}$$

Figure 5.3: The transition relation.

Invariance Principle: to show that a property holds for all reachable states, it is sufficient to show that it holds for the initial state and is preserved under any transition.

5.5.1 Establishing the liveness property

The Liveness Property can be expressed in Temporal Logic as

$$\Box(\text{READ}_{p\ominus 1} \ \Rightarrow \ \Diamond\text{READ}_p), \qquad (\dagger)$$

that is, it always holds that if philosopher $p\ominus 1$ reads, then eventually philosopher p reads. We must prove this property under the assumption that no philosopher eats forever:

$$\Box(\text{EAT}_p \ \Rightarrow \ \Diamond\neg\text{EAT}_p). \qquad (\ddagger)$$

So assume that $\mathrm{READ}_{p\ominus 1}$ holds. We must prove that $\Diamond\mathrm{READ}_p$ holds. There are two cases as follows.

- Case EAT_p holds. By assumption (‡), there is an instant when $\mathrm{EAT}_p \wedge \neg \bigcirc \mathrm{EAT}_p$ holds. Thus if

$$\mathrm{READ}_{p\ominus 1} \wedge \mathrm{EAT}_p \wedge \neg \bigcirc \mathrm{EAT}_p \Rightarrow \bigcirc \mathrm{READ}_p \qquad (\S)$$

  is a valid property of the transition system, $\Diamond\mathrm{EAT}_p$ holds. In fact, we verified using Mona that (§) indeed holds.

- Case $\neg\mathrm{EAT}_p$ holds. If EAT_p becomes true, then use the previous case. Otherwise, $\neg\mathrm{EAT}_p$ continues to hold. Now, by the assumption (‡) at some point $\neg\mathrm{EAT}_{p\oplus 1}$ will hold. We then use the property

$$\mathrm{READ}_{p\ominus 1} \wedge \neg\mathrm{EAT}_p \wedge \neg \bigcirc \mathrm{EAT}_{p\oplus 1} \Rightarrow \bigcirc \mathrm{READ}_p \vee \bigcirc \mathrm{EAT}_p, \quad (\P)$$

  which we have also verified using Mona, to show that eventually READ_p holds (or eventually EAT_p holds, which contradicts the assumption that $\neg\mathrm{EAT}_p$ continues to hold).

5.6 Implementation

Mona is our implementation of the decision procedure, which translates formulas of MSO to finite-state automata as outlined in Section 5.3. Our tool is implemented in Standard ML of New Jersey. A previous version of Mona was written in C with explicit garbage collection and based on representing transition functions in a conjunctive normal form. Our present tool runs up to 50 times faster due to improved algorithms.

5.6.1 Representation of automata

Since the size of the extended alphabet grows exponentially with the number of variables, a straightforward implementation based on explicitly representing the alphabet would only work for very simple examples. Instead, we represent the transition relation using Binary Decision Diagrams (BDDs) [11, 12]. In this way, the alphabet is never explicitly represented. For the external alphabet of ASCII-characters, we choose an encoding based on seven extra tracks holding the binary representation. Thus, character classes such as `[a-zA-Z]` become represented as very simple BDDs.

A deterministic automaton $\mathcal{A}$ is represented as follows. The state space is $Q = \{0, 1, \ldots, n-1\}$, where n is size of the state space; $\mathbb{B}^k$ is the extended alphabet; $i_0 \in Q$ is the initial state; $\delta : Q \times \mathbb{B}^k \to Q$ is the transition function; and $F \subseteq Q$ is the set of accepting states. We use a bit vector of size n to represent F and an array containing n pointers to roots of multi-terminal BDDs representing δ. A leaf of a BDD holds the integer designating the next state. An internal node v is called a *decision node* and contains an *index* denoted $v.index$,

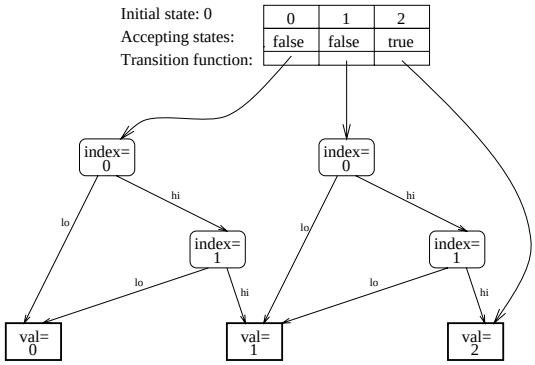
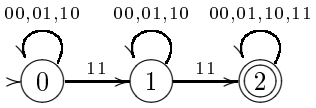


Figure 5.4: BDD automaton representation.

where $0 \leq v.index < k$, and high and low successors $v.hi$ and $v.lo$. If b is a sequence of k bits, that is $b \in \mathbb{B}^k$, then $\delta(q, b)$ is found by looking up the qth entry in the array and following the decision nodes according to b until a leaf is reached (node v is followed by selecting the high successor if the $v.index$th component of b is 1 and the low successor if it is 0).

For example, the following finite automaton accepting all strings over $\mathbb{B}^2$ with at least two occurrences of the letter "11"



could be represented as in Figure 5.4.

The use of BDDs makes the representation very succinct in comparison to our earlier attempt to handle automata with large alphabets [69]. In most cases, we avoid the exponential blow-up associated with an explicit representation of the alphabet. We shall see that all operations on automata needed can be performed by means of simple BDD operations.

Another possibility would have been to use a two-dimensional array of ordinary BDDs. But that would complicate the operations on automata, because many more BDD operations would be needed.

5.6.2 Rewriting formulas

The first step in the translation consists of rewriting formulas so as to eliminate nested terms. Then all terms are variables and all formulas are among a small number of basic formulas.

5.6.3 Translating formulas

The translation is inductive. All automata corresponding to basic formulas have a small number of states (less than five!).

The composite formulas are translated by use of operations on automata. For $\neg\varphi$, $\varphi_1 \wedge \varphi_2$ and $\exists P : \varphi$, which are the ones left after rewriting, we need the operations of complement, product, projection, and determinization.

Complement Complementation is done by simply negating the bit vector representing the set of final states.

Product The product automaton $\mathcal{A}$ of two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is

$$(Q_1 \times Q_2, \mathbb{B}^k, (i_1, i_2), \delta, F_1 \times F_2),$$

where $\delta((q_1, q_2), b) = (\delta_1(q_1, b), \delta_2(q_2, b))$. We are careful, however, to consider only those states of $\mathcal{A}$ that are reachable from (i_1, i_2).

When considering a new state (q_1, q_2), we need to construct the BDD representing the corresponding part of the transition function δ. We use the binary apply operation on the BDDs corresponding to q_1 and q_2. For each pair of states (q', q'') encountered in a pair of leaves, we associate a unique integer in the range $\{0, 1, \ldots N-1\}$, where N is the number of different pairs considered so far. In this way, the new BDDs created conform with the standard representation.

Projection and determinization Projection is the conversion of an automaton over $\mathbb{B}^{k+1}$ to a nondeterministic automaton over $\mathbb{B}^k$ necessary for translating a formula of the form $\exists P : \varphi$. On any letter $b \in \mathbb{B}^k$, there are two transitions possible in the nondeterministic automaton corresponding to whether the P-track is 0 or 1. Therefore this automaton is not hard to construct using the projection (restriction) operation of BDDs.

Determinization is done according to the subset construction. The use of the apply operation is similar to that of the product construction except that leaves hold subsets of states.

5.6.4 Minimizing

Minimization seems essential in order to obtain an effective decision procedure. For example, if a tautology occurs during calculations, then it is obviously a good idea to represent it using a one-state automaton instead of an automaton with e.g. 10,000 states.

The difficulty in obtaining an efficient minimization algorithm stems from the requirement to keep our shared BDDs in reduced form. Recall that a reduced BDD has no duplicate terminals or nonterminals. Such a BDD is just a specialized form of directed acyclic graphs that has been compressed by combining structurally isomorphic nodes (see Aho, Hopcroft, and Ullman [1] or Cai and Paige [17, Section 3.4]). In addition, a reduced BDD has no redundant

tests [11]. Such a BDD is obtained by repeatedly pruning every internal vertex v that has both outedges leading to the same vertex w, and redirecting all of v's incoming edges to w.

Suppose that the shared BDD had all duplicate terminals and nonterminals eliminated, but did not have any of its redundant tests eliminated. Then it would be easy to treat the deterministic finite automaton combined with its BDD machinery as a single automaton whose states were the union of the BDD nodes and the original automaton states, and whose alphabet were zero and one. If this derived automaton had n states, then it could be minimized in $O(n \log n)$ steps using Hopcroft's algorithm [63]. Unfortunately, such an automaton would be too big.

For our purposes, the space savings due to redundant test removal is of crucial importance. But the important 'skip' states that arise from redundant test removal complicates minimization. Our algorithm combines techniques based on [1] with new methods adapted for use with the shared BDD representation of the transition function. For a finite automaton with n states and a transition function represented by m BDD nodes, the algorithm presented here achieves worst-case running time $O(\max(n, m)n)$.

Terminology A *partition* $\mathcal{P}$ of a finite set U is a set of disjoint nonempty subsets of U such that the union of these sets is all of U. The elements of $\mathcal{P}$ are called its *blocks*. A *refinement* $\mathcal{Q}$ of $\mathcal{P}$ is a partition of U such that any block of $\mathcal{Q}$ is a subset of a block of $\mathcal{P}$. If $q \in U$, then $[q]_{\mathcal{P}}$ denotes the block of partition $\mathcal{P}$ containing the element q, and when no confusion arises, we drop the subscript.

Let $\mathcal{A} = (Q, \mathbb{B}^k, i_0, \delta, F)$ denote a deterministic finite automaton, and let $\mathcal{P}$ be a partition of Q, and $\mathcal{Q}$ a refinement of $\mathcal{P}$. A block B of $\mathcal{Q}$ *respects* the partition $\mathcal{P}$ if for all $q, q' \in B$ and for all $b \in \mathbb{B}^k$, $[\delta(q, b)]_{\mathcal{P}} = [\delta(q', b)]_{\mathcal{P}}$. Thus, δ cannot distinguish between the elements in B relative to the partition $\mathcal{P}$. A partition $\mathcal{Q}$ *respects* $\mathcal{P}$ if every block of $\mathcal{Q}$ respects $\mathcal{P}$. A partition is *stable* if it respects itself. The *coarsest, stable partition* $\mathcal{Q}$ *respecting* $\mathcal{P}$ is a unique partition such that any other stable partition respecting $\mathcal{P}$ is a refinement of $\mathcal{Q}$.

The refinement algorithm The minimal automaton $\mathcal{A}'$ recognizing $L(\mathcal{A})$ is isomorphic to the automaton defined by the coarsest stable partition $\mathcal{Q}^{\mathcal{A}}$ of Q respecting the partition $\{F, Q \setminus F\}$. The states of $\mathcal{A}'$ are $\mathcal{Q}^{\mathcal{A}}$, the transition function δ' is defined by $\delta'([p], b) = [\delta(p, b)]$, the initial state is $[i_0]$, and the set of final states is $F' = \{[f] | f \in F\}$.

Now we are ready to sketch our minimizing algorithm, which works by gradually refining a current partition.

- First split Q into an initial partition $\mathcal{Q} = \{F, Q \setminus F\}$. Note that $\mathcal{Q}^{\mathcal{A}}$ is a refinement of this partition.

- Now let $\mathcal{P}$ be the current partition. We construct the new current partition $\mathcal{Q}$ so that it respects $\mathcal{P}$ while $\mathcal{Q}^{\mathcal{A}}$ remains a refinement of $\mathcal{Q}$.

For each state q in Q consider the functions $f_q : \mathbb{B}^k \to \mathcal{P}$ defined by $f_q(b) = [\delta(q,b)]_{\mathcal{P}}$ for all q and b. Now let the equivalence relation $\equiv$ be defined as $q \equiv q' \Leftrightarrow (f_q = f_{q'} \wedge [q]_{\mathcal{P}} = [q']_{\mathcal{P}})$. The new partition $\mathcal{Q}$ then consists of the equivalence classes of $\equiv$. By definition of the f_q's, $\mathcal{Q}$ respects $\mathcal{P}$ and is the coarsest such partition implying the invariant.

We repeat this process until $\mathcal{P} = \mathcal{Q}$.

It can be shown that the final partition $\mathcal{Q}$ is obtained in at most n iterations and equals $\mathcal{Q}^{\mathcal{A}}$. The preceding algorithm is an abstraction of the initial naive algorithm presented in [1, Section 4.13].

The difficult step in the above algorithm is the splitting according to the functions f_q. However, we can here elegantly take advantage of the shared BDD representation. The idea is to construct a BDD representing the functions f_q for each state. We represent a partition of the states Q, by associating with each state $q \in Q$ a *block id* identifying its block. The BDD for f_q is calculated by performing a unary apply on the collection of shared BDDs, where the value calculated in a leaf is the block id. By a suitable generalization of the standard algorithm, it is possible to carry out these calculations while visiting each node at most once (assuming that hashing takes constant time). Thus the split operation requires time $O(\max(n,m))$. Since we use shared BDDs, we may use the results of the apply operations directly as new block ids.

The splitting step without hashing An alternative implementation of the splitting step is possible that achieves the same worst case time bound $O(\max(n,m))$ without hashing. It is instructive to first consider the case in which the shared BDDs are reduced only by eliminating redundant nodes but not by eliminating redundant tests. In this case the BDD may be regarded as an acyclic deterministic automaton D whose states are the BDD nodes, and whose alphabet is zero and one. Consider a partition $\mathcal{P}'$ of the BDD nodes defined by equivalence classes of the following relation. Two BDD leaves are equivalent iff their next states belong to the same block of partition $\mathcal{P}$. All decision nodes of the BDD are equivalent. The coarsest stable partition $\mathcal{Q}'$ that respects $\mathcal{P}'$ for automaton D can be solved in $O(m)$ worst case time by Revuz [119] and Cai and Paige [17, Section 3.4]. Finding the equivalence classes of states in Q that point to BDD roots belonging to the same block of $\mathcal{Q}'$ (that is, finding the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$) solves the splitting step in the original automaton in time $O(n)$.

In the case of fully reduced BDDs, the splitting step is somewhat harder, and a closer look at the BDD structure is needed. For each decision node v, $v.index$ represents a position in a string of length k such that $v.index < (v.lo).index \wedge v.index < (v.hi).index$. For each BDD leaf v we have $v.index = k$, and let $v.lo = v.hi$ be an automaton state belonging to Q. For each BDD node v we define function $f_v : \mathbb{B}^k \to \mathcal{P}$ much like the way functions f_q were defined earlier on automaton states. For each nonleaf v, f_v is defined by the rule $f_v(b) = f_{v.lo}(b)$ if $b_{v.index} = 0$; $f_v(b) = f_{v.hi}(b)$ if $b_{v.index} = 1$. For each

leaf v, f_v is a constant function that maps every argument into an element (that is, a block) of partition $\mathcal{P}$.

If $q \in Q$ is an automaton state that points to a BDD root v, then, clearly, $f_q = f_v$. It is also not hard to see that for any two nonleaf BDD nodes v and v', $f_v = f_{v'}$ iff either of the following two conditions hold:

(1) $v.index = v'.index \wedge f_{v.hi} = f_{v'.hi} \wedge f_{v.lo} = f_{v'.lo}$, or

(2) $f_{v.hi} = f_{v.lo} = f_v \wedge v.hi = v'$.

This leads to the more concrete equivalence relation $\equiv$ on BDD nodes defined as $v \equiv v'$ iff $f_v = f_{v'}$ iff either,

(1) $v.index = v'.index = k \wedge [v.lo]_{\mathcal{P}} = [v'.lo]_{\mathcal{P}}$, or

(2) $v.index = v'.index < k \wedge v.hi \equiv v'.hi \wedge v.lo \equiv v'.lo$, or

(3) $v.index < k \wedge v.lo \equiv v.hi \equiv v'$.

Note that two BDD nodes of different index can be equivalent only by condition (3). Note also, that we can strengthen condition (2) with the additional constraint $v.hi \not\equiv v.lo$ without modifying the equivalence relation. These two observations allow us to construct the equivalence classes inductively using a bottom-up algorithm that processes all BDD nodes of the same index in descending order, proceeding from leaves to roots. The steps are sketched just below.

(1) In a linear time pass through all of the BDD nodes, place each node in a bucket according to its index. An array of $k+1$ buckets can be used for this purpose.

(2) Next, distribute the BDD leaves (contained in the bucket associated with index k) into blocks whose nodes all have lo successors that belong to the same block of $\mathcal{P}$. This takes time proportional to the number of leaves.

(3) For $j = k-1, ..., 0$ examine each node v with $v.index = j$. Both nodes $v.lo$ and $v.hi$ have already been examined, and have been placed into blocks. Hence, a streamlined form of multiset sequence discrimination [17] can be used to place v either in an old block (according to condition (3)) or a new block (according to condition (2)) for nodes whose children belong pair-wise to the same old block.

The preceding algorithm computes the equivalence classes as the final set of blocks in $O(m)$ time. As before, we can use these equivalence classes to find the coarsest partition $\mathcal{Q}$ that respects $\mathcal{P}$, which solves the splitting step in the original automaton, in time $O(n)$. Thus, the total worst-case time to solve the splitting step is $O(\max(n, m))$ (without hashing).

In an efficient implementation of finite-state automaton minimization, when the splitting algorithm above is is performed repeatedly, we only need to perform the first step of that algorithm (that is, sorting BDD nodes according to index) once. Thus, the full DFA minimization algorithm runs in worst case time $O(\max(n, m)n)$ without hashing.

BDD reduction without hashing Sieling and Wegener[125] were the first to compress an arbitrary BDD into fully reduced form in linear time. Their result depended on a radix sort, which is closely related to the multiset discrimination technique that we use. However, their algorithm needs to maintain integer representations of BDD nodes, and it utilizes two arrays of size m. We can show how our algorithm just described can be modified to fully reduce an arbitrary BDD in worst case time linear in the number of BDD nodes (without hashing), but with expected auxiliary space k times smaller than Sieling and Wegener's algorithm.

Let $\mathcal{Q}'$ be the partition of BDD nodes produced by the algorithm. The states of the reduced BDD are the blocks in $\mathcal{Q}'$. For each block $B \in \mathcal{Q}'$, $B.index$ is the largest index of any BDD node contained in B. Let v' be any node belonging to B of maximum index. If v' is a BDD leaf, then B is a leaf in the reduced BDD (that is, $B.index = k$), and $B.lo = B.hi = v'.lo$. Otherwise, $B.lo = [v'.lo]_{\mathcal{Q}'}$ and $B.hi = [v'.hi]_{\mathcal{Q}'}$. The hi and lo successor blocks can be determined during the multiset sequence discrimination pass when a new block is first created. The index of the first node placed in a newly created block is the index for that block.

What distinguishes our algorithm from that of Sieling and Wegener is that our buckets in steps (2) and (3) are associated with actual BDD nodes (inside the main BDD data structure). Their buckets are associated with components of two auxiliary arrays of size m each. If we replaced each equivalence class by a single witness (as they do) each iteration of step (3), then our auxiliary space would be bounded by the maximum number of BDD nodes that have the same index. If BDD nodes were uniformly distributed among indexes, then this number is m/k, which would give us a k-fold advantage in auxiliary space over their algorithm. We expect a minor constant factor advantage in time as well, because our BDD nodes are represented by their locations instead of by computed integer values, and because we avoid array access in favor of less expensive list and pointer processing.

Work is in progress for exploring the "processing the smaller half" idea found in e.g. [106]. We should mention, however, that the current implementation of the minimization algorithm in practice seems to run faster than the procedures for constructing product and subset automata.

5.7 Current version of Mona

Mona is a continuing research project at BRICS and has since this publication been improved substantially in many ways. The basic ideas, such as the BDD representation of automata are still the same. For an update on the capabilities of the Mona tool we refer to the user manual [72] and [73].

Chapter 6

Dynamic Linear Time Temporal Logic

Contents

A simple extension of the propositional temporal logic of linear time is proposed. The extension consists of strengthening the until operator by indexing it with the regular programs of propositional dynamic logic. It is shown that DLTL, the resulting logic, is expressively equivalent to the monadic second-order theory of ω-sequences. In fact, a sublogic of DLTL which corresponds to propositional dynamic logic with a linear time semantics is already expressively complete. We show that DLTL has an exponential time decision procedure and admits a finitary axiomatization. We also point to a natural extension of the approach presented here to a distributed setting.

6.1 Introduction

We present here a simple extension of the propositional temporal logic of linear time. The basic idea is to strengthen the until modality by indexing it with the regular programs of propositional dynamic logic. The resulting logic, called

dynamic linear time temporal logic (DLTL), is easy to handle. It has the full expressive power of the monadic second-order theory of ω-sequences. Indeed a sublogic of DLTL is already expressively complete. A pleasant feature of this sublogic is that it is just propositional dynamic logic operating in a linear time framework.

In addition to our expressiveness results we show that DLTL has an exponential time decision procedure. We also extend the well known axiomatization of propositional dynamic logic [76] to obtain an axiomatization of DLTL.

Our work may be viewed from two different perspectives. The first one is from the standpoint of process logics [53, 104, 114] which attempt a rapprochement between dynamic and temporal logics. However the study of process logics is committed to viewing dynamic logic as a restricted kind of a *branching time* temporal logic. One then attempts to bring in some additional mechanisms for talking about computational paths. Our point of departure consists of merging, in a very simple way, dynamic logic and temporal logic in a *linear time* setting.

The second perspective has to do with attempts to augment the expressive power of linear time temporal logic. One route consists of permitting quantification over atomic propositions. The resulting logic called QPTL [126] is as expressive as MSO, the monadic second-order theory of sequences but its decision procedure has non-elementary time complexity. The second route consists of augmenting linear time temporal logic with the so called automaton connectives. The resulting logic called ETL [156] is equal in expressive power to MSO while admitting an exponential time decision procedure.

Our logic is, in spirit, inspired by ETL and it can be easily translated into ETL. It may appear to be at first sight to be a mere reformulation of ETL with some cosmetic changes. This however has to do with the instinctive identification one makes between finite state automata and regular expressions. In fact DLTL is quite different in terms of the mechanisms it offers for structuring formulas and we feel that it is more transparent and easier to work with. The results and the proofs we present here are designed to support this claim. Our approach also leads to smooth generalizations in non-sequential settings where similar extensions in terms of ETL will be hard to cope with.

In the next section we start with an action-based version of of linear time temporal logic in order to fix terminology. In Section 6.3 we present DLTL and its semantics. This is then followed by a more detailed assessment of the similarities and the differences between ETL and DLTL.

In Section 6.4 we prove the decidability of DLTL by reducing it to the emptiness problem for Büchi automata. In Section 6.5 we show that DLTL$^-$, a sublogic of DLTL, has the same expressive power as MSO, the monadic second-order theory of sequences. We then establish similar results for the first-order fragment of MSO with the help of the "star-free" fragments of DLTL and DLTL$^-$.

In Section 6.6, we extend the axiomatization of PDL (propositional dynamic logic) and the completeness proof in [76] to obtain finitary axiomatizations of DLTL and DLTL$^-$. In the final section we point to a natural generalization in the setting of distributed systems. This generalization is eminently accessible

and offers additional support to our belief that the synthesis of dynamic and temporal logics in a linear time framework as pursued here is a fruitful one.

6.2 Linear Time Temporal Logic

One key feature of the syntax and semantics of our temporal logic is the treatment of *actions* as first class objects. To bring this out we formulate a version of LTL (linear time temporal logic) in which the next-state modality is indexed by actions taken from a fixed alphabet set.

Through the rest of the chapter we fix a finite non-empty alphabet Σ. We let a, b range over Σ and refer to members of Σ as actions. Σ^* is the set of finite words and Σ^ω is the set of infinite words generated by Σ with $\omega = \{0, 1, 2, \ldots\}$. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and denote the null word by ε. We let σ, σ' range over Σ^ω and τ, τ', τ'' range over Σ^*. Finally $\preceq$ is the usual prefix ordering defined over Σ^* and for $u \in \Sigma^\infty$, we let $\mathrm{prf}(u)$ be the set of finite prefixes of u.

Next we fix a countable set of atomic propositions $P = \{p_1, p_2, \ldots\}$ and let p, q range over P. The set of formulas of $\mathrm{LTL}(\Sigma)$ is then given by the syntax:

$$\mathrm{LTL}(\Sigma) ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a\rangle\alpha \mid \alpha \,\mathcal{U}\, \beta.$$

Through the rest of this section α, β will range over $\mathrm{LTL}(\Sigma)$.

A model of $\mathrm{LTL}(\Sigma)$ is a pair $M = (\sigma, V)$ where $\sigma \in \Sigma^\omega$ and $V : \mathrm{prf}(\sigma) \longrightarrow 2^P$ is a valuation function. Let $M = (\sigma, V)$ be a model, $\tau \in \mathrm{prf}(\sigma)$ and α be a formula. Then $M, \tau \models \alpha$ will stand for α being satisfied at τ in M. This notion is defined inductively in the expected manner.

- $M, \tau \models p$ iff $p \in V(\tau)$.
- $M, \tau \models \neg\alpha$ iff $M, \tau \not\models \alpha$.
- $M, \tau \models \alpha \vee \beta$ iff $M, \tau \models \alpha$ or $M, \tau \models \beta$.
- $M, \tau \models \langle a\rangle\alpha$ iff $\tau a \in \mathrm{prf}(\sigma)$ and $M, \tau a \models \alpha$.
- $M, \tau \models \alpha \,\mathcal{U}\, \beta$ iff there exists τ' such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $M, \tau\tau' \models \beta$. Moreover for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $M, \tau\tau'' \models \alpha$.

We note that the next-state modality of LTL is definable via $O\alpha \stackrel{\mathrm{def}}{=} \bigvee_{a\in\Sigma}\langle a\rangle\alpha$. It is well known [41, 70] that $\mathrm{LTL}(\Sigma)$ is expressively equivalent to the first-order theory of sequences. Hence this temporal logic, relative to MSO, has limited expressive power. For instance, as pointed out by Wolper [155], the property "p holds at every even position" is not definable in this logic.

6.3 Dynamic Linear Time Temporal Logic

Our extension of $\mathrm{LTL}(\Sigma)$ basically consists of indexing the until operator with the programs of PDL (e.g. [39, 51]). We start by defining the set of programs

(regular expressions) generated by Σ. This set is denoted by $\mathrm{Prg}(\Sigma)$ and is given by:

$$\mathrm{Prg}(\Sigma) ::= a \mid \pi_0 + \pi_1 \mid \pi_0; \pi_1 \mid \pi^*.$$

Here and elsewhere, π, π' with or without subscripts will range over $\mathrm{Prg}(\Sigma)$. With each program we associate a set of finite words via the map $||\cdot|| : \mathrm{Prg}(\Sigma) \longrightarrow 2^{\Sigma^*}$. This map is defined in the standard fashion. As before, we fix a countable set of atomic propositions $P = \{p_1, p_2, \dots\}$ and let p, q range over P. The set of formulas of $\mathrm{DLTL}(\Sigma)$ is then given by the following syntax:

$$\mathrm{DLTL}(\Sigma) ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha\, \mathcal{U}^\pi \beta.$$

Here and throughout the rest of the chapter we let α, β range over $\mathrm{DLTL}(\Sigma)$. The notion of a model is as in the case of $\mathrm{LTL}(\Sigma)$. So let $M = (\sigma, V)$ be a model, $\tau \in \mathrm{prf}(\sigma)$ and $\alpha \in \mathrm{DLTL}(\Sigma)$. Then $M, \tau \models \alpha$ is defined inductively. The base case and the boolean connectives are handled as before. The semantics of the augmented until operator is given by :

- $M, \tau \models \alpha\ \mathcal{U}^\pi \beta$ iff there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $M, \tau\tau' \models \beta$. Moreover, for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $M, \tau\tau'' \models \alpha$.

Thus $\mathrm{DLTL}(\Sigma)$ is obtained form $\mathrm{LTL}(\Sigma)$ by strengthening the until operator. To satisfy $\alpha\, \mathcal{U}^\pi \beta$, one must satisfy $\alpha \mathcal{U} \beta$ along some finite stretch of behaviour which is in the (linear time) behaviour of the program π.

As usual, $\alpha \in \mathrm{DLTL}(\Sigma)$ is *satisfiable* iff there exist a model $M = (\sigma, V)$ and $\tau \in \mathrm{prf}(\sigma)$ such that $M, \tau \models \alpha$.

Apart from the conventional derived propositional connectives such as $\wedge, \Rightarrow$ and $\Leftrightarrow$ the derived modality $\langle\pi\rangle$ and its dual $[\pi]$ will play an important role in the sequel.

- $tt \stackrel{\mathrm{def}}{=} p_1 \vee \neg p_1$. Recall that $P = \{p_1, p_2, \dots\}$.
- $\langle\pi\rangle\alpha \stackrel{\mathrm{def}}{=} tt\, \mathcal{U}^\pi \alpha$.
- $[\pi]\alpha \stackrel{\mathrm{def}}{=} \neg\langle\pi\rangle\neg\alpha$.

Suppose $M = (\sigma, V)$ is a model and $\tau \in \mathrm{prf}(\sigma)$. It is easy to see that $\sigma, \tau \models \langle\pi\rangle\alpha$ iff there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \alpha$. It is also easy to see that $\sigma, \tau \models [\pi]\alpha$ iff for every $\tau' \in ||\pi||$, if $\tau\tau' \in \mathrm{prf}(\sigma)$ then $\sigma, \tau\tau' \models \alpha$. In this sense, the program modalities of PDL acquire a linear time semantics in the present setting.

Note that $a \in \Sigma$ is a member of $\mathrm{Prg}(\Sigma)$ and hence $\langle a\rangle\alpha$ is a derived modality. Letting $\Sigma = \{a_1, a_2, \dots, a_n\}$, it is also easy to see that the until operator of $\mathrm{LTL}(\Sigma)$ can be obtained via: $\alpha\mathcal{U}\beta \stackrel{\mathrm{def}}{=} \alpha\, \mathcal{U}^{\Sigma^*} \beta$ with Σ as a shorthand for the program $a_1 + a_2 + \cdots + a_n$. Thus $\mathrm{LTL}(\Sigma)$ is a fragment of $\mathrm{DLTL}(\Sigma)$ both in terms of syntax and semantics. To see that $\mathrm{DLTL}(\Sigma)$ is strictly more expressive than

LTL(Σ), let $\pi_{ev} = (\Sigma;\Sigma)^*$. It is easy to see that $\alpha_{ev} = [\pi_{ev}]p$ is a specification of the property "p holds at every even position".

We shall close out the section by briefly discussing the key differences between DLTL(Σ) and ETL, the extension of LTL proposed by Wolper [155]. We shall present a simplified form of ETL so as to stay close to DLTL. First we fix an enumeration of $\Sigma = \{a_1, a_2, \ldots, a_n\}$. The syntax of the logic that we shall name as ETL(Σ) is given by:

$$\text{ETL}(\Sigma) ::= p \mid \neg\phi \mid \phi \vee \phi' \mid \mathcal{A}(\phi_0, \phi_1, \ldots, \phi_n).$$

Here $\mathcal{A}$ is a finite state automaton of the form $\mathcal{A} = (Q, \longrightarrow, Q_{in}, F)$ with $\longrightarrow \subseteq Q \times \Sigma \times Q$ as the transition relation, $Q_{in} \subseteq Q$ as the initial states and $F \subseteq Q$ as the accepting states. Let $L(\mathcal{A})$ be the language of finite words accepted by $\mathcal{A}$. We shall assume for the sake of convenience that $\varepsilon \notin L(\mathcal{A})$ for each formula of the form $\mathcal{A}(\phi_0, \phi_1, \ldots, \phi_n)$.

A model for ETL(Σ) is, as before, a pair $M = (\sigma, V)$ with $V : \text{prf}(\sigma) \longrightarrow 2^P$. Let $\tau \in \text{prf}(\sigma)$. Then $M, \tau \models \phi$ is defined for the cases of atomic propositions and the boolean connectives in the expected manner. The automaton connective is interpreted as follows.

- $M, \tau \models \mathcal{A}(\phi_0, \phi_1, \ldots, \phi_n)$ iff there exists $a_{i_1} a_{i_2} \ldots a_{i_m} \in L(\mathcal{A})$ such that the following conditions are satisfied:
  - $i_1, i_2, \ldots, i_m \in \{1, 2, \ldots, n\}$. (recall that $\Sigma = \{a_1, a_2, \ldots, a_n\}$).
  - $\tau a_{i_1} a_{i_2} \cdots a_{i_m} \in \text{prf}(\sigma)$.
  - $M, \tau \models \phi_0$ and $M, \tau a_{i_1} \cdots a_{i_j} \models \phi_{i_j}$ for $1 \leq j \leq m$.

Though the technical details are somewhat different, ETL(Σ) captures the spirit of the logic presented in [148]. The key drawback of ETL(Σ), as we see it, lies in its lack of structuring principles for forming compound formulas. The only mechanism that ETL(Σ) has — apart from the boolean connectives — to form compound formulas is *by nesting* the automaton formulas. Thus a typical compound formula would look like:

$$\mathcal{A}^1(\phi_0^1, \mathcal{A}^2(\phi_0^2, \phi_1^2, \mathcal{A}^3(\phi_0^3, \ldots, \phi_n^3), \phi_3^2, \ldots, \phi_n^2), \phi_2^1, \ldots, \phi_n^1).$$

In contrast, DLTL(Σ) adds to the familiar mechanisms of LTL an *orthogonal* and well-understood component; namely, the language of regular expressions. Equally important, this orthogonal component is formulated purely in terms of Σ and not in terms of arbitrary formulas as is the case of ETL. In fact, ETL, as formulated in [148] has an uncontrolled amount of "external" elements in the sense that the states and the alphabets of the automata which are used to write down the automaton formulas have little to do with the logic under consideration.

It is an easy exercise to translate DLTL into ETL with only a linear blow-up in the size of the formulas. It will however be more productive and illuminating to give an independent treatment of DLTL as we shall do here.

6.4 A Decision Procedure for DLTL

The goal here is to show that the satisfiability problem for DLTL(Σ) can be solved in deterministic exponential time. This will be achieved by effectively constructing for each $\alpha \in \text{DLTL}(\Sigma)$, a Büchi automaton $\mathcal{B}_\alpha$ such that the language of ω-words accepted by $\mathcal{B}_\alpha$ is non-empty iff α is satisfiable.

A Büchi automaton over Σ is a tuple $\mathcal{B} = (Q, \longrightarrow, Q_{in}, F)$ where:

- Q is a finite non-empty set of states.
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation.
- $Q_{in} \subseteq Q$ is a set of initial states.
- $F \subseteq Q$ is a set of accepting states.

Let $\sigma \in \Sigma^\omega$. Then a run of $\mathcal{B}$ over σ is a map $\rho : \text{prf}(\sigma) \longrightarrow Q$ such that:

- $\rho(\varepsilon) \in Q_{in}$.
- $\rho(\tau) \xrightarrow{a} \rho(\tau a)$ for each $\tau a \in \text{prf}(\sigma)$.

The run ρ is accepting iff $\inf(\rho) \cap F \neq \emptyset$ where $\inf(\rho) \subseteq Q$ is given by $q \in \inf(\rho)$ iff $\rho(\tau) = q$ for infinitely many $\tau \in \text{prf}(\sigma)$. Finally $L(\mathcal{B})$, the language of ω-words accepted by $\mathcal{B}$, is:

$$L(\mathcal{B}) = \{\sigma \mid \exists \text{ an accepting run of } \mathcal{B} \text{ over } \sigma\}.$$

Through the rest of the section we fix a formula α_0. To construct $\mathcal{B}_{\alpha_0}$ we first define the (Fischer-Ladner) closure of α_0 as follows. $cl(\alpha_0)$ is the least set of formulas that satisfies:

- $\alpha_0 \in cl(\alpha_0)$.
- If $\neg\beta \in cl(\alpha_0)$ then $\beta \in cl(\alpha_0)$.
- If $\alpha \vee \beta \in cl(\alpha_0)$ then $\alpha, \beta \in cl(\alpha_0)$.
- If $\alpha\, \mathcal{U}^\tau \beta \in cl(\alpha_0)$ then $\alpha, \beta \in cl(\alpha_0)$.

Now $CL(\alpha_0)$, the *closure* of α_0, is defined to be:

$$CL(\alpha_0) = cl(\alpha_0) \cup \{\neg\beta \mid \beta \in cl(\alpha_0)\}.$$

In what follows $\neg\neg\beta$ will be identified with β. Moreover, throughout the section, all the formulas that we encounter — unless stated otherwise — will be assumed to be members of $CL(\alpha_0)$. For convenience, we shall often write CL instead of $CL(\alpha_0)$.

$A \subseteq CL$ is called an *atom* iff it is a subset of CL satisfying:

- $\beta \in A$ iff $\neg\beta \notin A$.

- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- If $\beta \in A$ and $\varepsilon \in ||\pi||$ then $\alpha\, \mathcal{U}^{\pi} \beta \in A$.

$AT(\alpha_0)$ is the set of atoms and again we shall often write AT instead of $AT(\alpha_0)$. Next we define $Req(\alpha_0)$, the set of until requirements of α_0, to be the subset of CL given by:

$$Req(\alpha_0) = \{\alpha\, \mathcal{U}^{\pi} \beta \mid \alpha\, \mathcal{U}^{\pi} \beta \in CL\}.$$

We shall write Req instead $Req(\alpha_0)$ and take ξ, ξ' to range over Req. For each $\xi = \alpha\, \mathcal{U}^{\pi} \beta \in Req$ we fix a finite state automaton $\mathcal{A}_\xi$ such that $L(\mathcal{A}_\xi) = ||\pi||$ where $L(\mathcal{A}_\xi)$ is the language of finite words accepted by $\mathcal{A}_\xi$. We shall assume each such $\mathcal{A}_\xi$ is of the form $\mathcal{A}_\xi = (Q_\xi, \longrightarrow_\xi, I_\xi, F_\xi)$ where Q_ξ is the set of states, $\longrightarrow_\xi \subseteq Q_\xi \times \Sigma \times Q_\xi$ is the transition relation, $I_\xi \subseteq Q_\xi$ is the set of initial states and $F_\xi \subseteq Q_\xi$ is the set of final states. Without loss of generality, we shall assume that $\xi \neq \xi'$ implies $Q_\xi \cap Q_{\xi'} = \emptyset$ for every $\xi, \xi' \in Req$. We set $Q = \bigcup_{\xi \in Req} Q_\xi$ and $\widehat{Q} = Q \times \{0, 1\}$.

The Büchi automaton $\mathcal{B}_{\alpha_0}$ associated with α_0 (from now on denoted as $\mathcal{B}$) can now be defined as

$$\mathcal{B} = (S, \Longrightarrow, S_{in}, F),$$

where the various components of $\mathcal{B}$ are specified as follows. We provide explanatory remarks immediately after the definition.

(1) $S \subseteq AT \times 2^{Q} \times 2^{\widehat{Q}} \times \{0, 1\} \times \{\downarrow, \checkmark\}$ such that $(A, X, \widehat{X}, x, f) \in S$ iff the following conditions are satisfied for each $\xi = \alpha\, \mathcal{U}^{\pi} \beta$:

   (i) If $\beta \in A$ then $F_\xi \subseteq X$. (Recall that $\mathcal{A}_\xi = (Q_\xi, \longrightarrow_\xi, I_\xi, F_\xi)$).

   (ii) If $\alpha \in A$ and $q \in X$ for some $q \in I_\xi$ then $\alpha\, \mathcal{U}^{\pi} \beta \in A$.

   (iii) If $\alpha\, \mathcal{U}^{\pi} \beta \in A$ then either $\beta \in A$ and $\varepsilon \in ||\pi||$ or $(q, 1-x) \in \widehat{X}$ for some $q \in I_\xi$. (Note that we are considering the candidate $(A, X, \widehat{X}, x, f)$ for membership in S).

   (iv) If $(q, z) \in \widehat{X}$ with $q \in Q_\xi$ and $q \notin F_\xi$ or $\beta \notin A$ then $\alpha \in A$.

(2) The transition relation $\Longrightarrow \subseteq S \times \Sigma \times S$ is defined as follows:

$$(A, X, \widehat{X}, x, f) \stackrel{a}{\Longrightarrow} (B, Y, \widehat{Y}, y, g)$$

iff the following conditions are satisfied for each $\xi = \alpha\, \mathcal{U}^{\pi} \beta$:

   (i) Suppose $q' \in Q_\xi \cap Y$ and $q \stackrel{a}{\longrightarrow}_\xi q'$ and $\alpha \in A$. Then $q \in X$.

   (ii) Suppose $(q, z) \in \widehat{X}$ with $q \in Q_\xi$. Suppose further that $q \notin F_\xi$ or $\beta \notin A$. Then $(q', z) \in \widehat{Y}$ for some q' with $q \stackrel{a}{\longrightarrow}_\xi q'$.

   (iii) If $f = \checkmark$ then $(y, g) = (1 - x, \downarrow)$. If $f = \downarrow$ then,

$$(y, g) = \begin{cases} (x, \downarrow), & \text{if there exists } (q, x) \in \widehat{X} \text{ such that} \\ & q \notin F_\xi \text{ or } \beta \notin A. \\ (x, \checkmark), & \text{otherwise.} \end{cases}$$

(3) $S_{in} = \{(A, X, \widehat{X}, x, f) \mid \alpha_0 \in A \text{ and } (x, f) = (0, \checkmark)\}$.

(4) $F = \{(A, X, \widehat{X}, x, f) \mid f = \checkmark\}$

To understand the functioning of the automaton $\mathcal{B}$, let (σ, V) be a model and ρ a run of $\mathcal{B}$ over σ. Assume further that $\tau \in \text{prf}(\sigma)$ and that $\rho(\tau) = (A, X, \widehat{X}, x, f)$. The role of the atom A, as usual, is to assert that the formulas in A will be satisfied at τ. To check this, the automaton should verify that all the until requirements are being satisfied. This work is divided into two phases; a 0-phase and a 1-phase. The value of the boolean variable x indicates the current phase of the automaton. The last component is used to signal the successful completion of one phase. The automaton will not toggle to the next phase until successful completion of the current phase. The component X corresponds to the so-called safety automaton in [147]. The point is that the automaton must assert $\alpha\, \mathcal{U}^\pi \beta$ at τ in case there is *some* possibility of satisfying this assertion in the unknown future. The component X, in combination with the transition relation, is designed to ensure this. The component $\widehat{X}$ is used to check the liveness requirements. The complication here is that while requirements of the form (q, x) are being checked, new requirements may come up. These will be tagged with the value $1 - x$ but will have to be simultaneously checked. They cannot be ignored while working towards discharging the requirements in the current phase. The definition of the state set of the automaton as well as the transition relation have been guided by these considerations. It might be that this information could be maintained in a more compact form but it is a pointless optimization at this stage.

We wish to first prove that α_0 is satisfiable iff $L(\mathcal{B}) \neq \emptyset$. Afterwards we will argue that the size of $\mathcal{B}$ can be chosen to be at most exponential in the size of α_0.

Lemma 6.4.1 *Suppose $L(\mathcal{B}) \neq \emptyset$. Then α_0 is satisfiable.*

Proof: Let $\sigma \in L(\mathcal{B})$ and $\rho : \text{prf}(\sigma) \longrightarrow S$ be an accepting run. For each $\tau \in \text{prf}(\sigma)$, let $\rho(\tau) = (A_\tau, X_\tau, \widehat{X}_\tau, x_\tau, f_\tau)$. Define the model $M = (\sigma, V)$ via:

$$V(\tau) = A_\tau \cap P \text{ for all } \tau \in \text{prf}(\sigma).$$

Claim: For all $\tau \in \text{prf}(\sigma)$ and $\delta \in CL$,

$$M, \tau \models \delta \textit{ iff } \delta \in A_\tau.$$

First note that if the claim is true then Lemma 6.4.1 follows at once. This is so because ρ is a run of $\mathcal{B}$ and hence $\rho(\varepsilon) \in S_{in}$. But from (3), in the definition of $\mathcal{B}$, it follows that $\alpha_0 \in A_\varepsilon$.

In proving the claim we will repeatedly refer to various clauses in the definition of the Büchi automaton $\mathcal{B}$. We proceed by structural induction on δ. For the base case and the boolean connectives the claim is obvious. Hence assume that $\delta = \alpha\, \mathcal{U}^\pi \beta$.

Suppose that $M, \tau \models \alpha\, \mathcal{U}^{\pi} \beta$. Since $M, \tau \models \alpha\, \mathcal{U}^{\pi} \beta$ there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $M, \tau\tau' \models \beta$. Moreover, $M, \tau\tau'' \models \alpha$ for every $\tau'' \in \Sigma^*$ such that $\varepsilon \preceq \tau'' \prec \tau'$.

Suppose $\tau' = \varepsilon$. Then $\varepsilon \in ||\pi||$ and $M, \tau \models \beta$. By the induction hypothesis $\beta \in A_\tau$. From the definition of an atom it follows that $\alpha\, \mathcal{U}^{\pi} \beta \in A_\tau$.

So assume that $\tau' \neq \varepsilon$. Let $\xi = \alpha\, \mathcal{U}^{\pi} \beta$ and R be an accepting run of $\mathcal{A}_\xi$ over $\tau' = a_1 a_2 \ldots a_n$ with $R(\varepsilon) = q_0 \in I_\xi$ and $R(a_1 a_2 \ldots a_i) = q_i$ for $1 \leq i \leq n$ and $q_n \in F_\xi$. Since $M, \tau\tau' \models \beta$ we have from the induction hypothesis that $\beta \in A_{\tau\tau'}$. Hence by (1.i), $F_\xi \subseteq X_{\tau\tau'}$. Now by the definition of R we are assured that $q_{n-1} \xrightarrow{a_n}_{\xi} q_n$. On the other hand, the fact that $M, \tau \models \alpha\, \mathcal{U}^{\pi} \beta$ and the choice of τ' guarantee that $M, \tau a_1 \ldots a_{n-1} \models \alpha$ (with the convention that $\varepsilon = a_1 \ldots a_{n-1}$ in case $n = 1$). By the induction hypothesis $\alpha \in A_{\tau a_1 \ldots a_{n-1}}$, so by (2.i) and the fact that $q_n \in X_{\tau a_1 \ldots a_n}$, we have that $q_{n-1} \in X_{\tau a_1 \ldots a_{n-1}}$. In case $n \geq 2$ we repeat the above argument at q_{n-1} to conclude that $q_{n-2} \in X_{\tau a_1 \ldots a_{n-2}}$. Continuing this way we can finally arrive at $q_0 \in X_\tau$ and $\alpha \in A_\tau$. But $q_0 \in I_\xi$ and hence by (1.ii) we are assured that $\alpha\, \mathcal{U}^{\pi} \beta \in A_\tau$.

For the converse direction assume that $\alpha\, \mathcal{U}^{\pi} \beta \in A_\tau$. There are four cases to consider depending on the values of x_τ and f_τ. We will only prove one case. The remaining cases can be resolved by similar arguments.

So assume that $x_\tau = 0$ and $f_\tau = \downarrow$. Suppose first that $\beta \in A_\tau$ and $\varepsilon \in ||\pi||$. Then by the induction hypothesis $M, \tau \models \beta$ and hence we at once have $M, \tau \models \alpha\, \mathcal{U}^{\pi} \beta$. So assume that $\beta \notin A_\tau$ or $\varepsilon \notin ||\pi||$. Then by (1.iii), $(q_0, 1) \in \widehat{X}_\tau$ for some $q_0 \in I_\xi$. Suppose $q_0 \in F_\xi$. Then $\varepsilon \in ||\pi||$ and thus $\beta \notin A_\tau$. This implies, by (1.iv), that $\alpha \in A_\tau$, and by the induction hypothesis we have that $M, \tau \models \alpha$.

Now with ρ being an accepting run of $\mathcal{B}$ over σ there must exist τ_1 and τ_2 in Σ^* such that the following conditions are satisfied:

- $\tau_1 \neq \varepsilon$ and $\tau_2 \neq \varepsilon$ and $\tau\tau_1\tau_2 \in \mathrm{prf}(\sigma)$.
- $x_{\tau\tau_1} = 0$ and $x_{\tau\tau_1\tau_2} = 1$. (Recall the notational convention that $\rho(u) = (A_u, X_u, \widehat{X}_u, x_u, f_u)$ for each $u \in \mathrm{prf}(\sigma)$).
- $f_{\tau\tau_1} = \checkmark$ and $f_{\tau\tau_1\tau_2} = \checkmark$.
- For each τ_1'' and τ_2'' in Σ^*, if $\varepsilon \preceq \tau_1'' \prec \tau_1$ then $f(\tau\tau_1'') \neq \checkmark$ and if $\varepsilon \prec \tau_2'' \prec \tau_2$ then $f(\tau\tau_1\tau_2'') \neq \checkmark$.

Let $\tau_1 = a_1 a_2 \ldots a_n$ and $\tau_2 = b_1 b_2 \ldots b_m$. Now $\rho(\tau) \overset{a_1}{\Longrightarrow} \rho(\tau a_1)$, $\alpha\, \mathcal{U}^{\pi} \beta \in A_\tau$ and $(q_0, 1) \in \widehat{X}_\tau$. Moreover, we have that $q_0 \notin F_\xi$ (if $\varepsilon \notin ||\pi||$) or $\beta \notin A_\tau$. Thus by (2.ii), there exists $q_1 \in Q_\xi$ such that $q_0 \xrightarrow{a_1}_{\xi} q_1$ and $(q_1, 1) \in \widehat{X}_{\tau a_1}$.

Now suppose $q_1 \in F_\xi$ and $\beta \in A_{\tau a_1}$. Then $a_1 \in ||\pi||$ and by the induction hypothesis $M, \tau a_1 \models \beta$. Since $M, \tau \models \alpha$ has already been deduced we have $M, \tau \models \alpha\, \mathcal{U}^{\pi} \beta$. So assume that $q_1 \notin F_\xi$ or $\beta \notin A_{\tau a_1}$. Then by repeating the arguments we had above for q_0 at q_1 we can arrive at $\alpha \in A_{\tau a_1}$, and hence by the induction hypothesis $M, \tau a_1 \models \alpha$. Moreover, we can conclude that there exists $q_2 \in Q_\xi$ such that $q_1 \xrightarrow{a_2}_{\xi} q_2$ and $(q_2, 1) \in \widehat{X}_{\tau a_1 a_2}$. Marching down τ_1 using this sequence of arguments we will either terminate with the conclusion $M, \tau \models$

$\alpha\,\mathcal{U}^\pi\beta$ or we will exhaust all of τ_1 while being able to conclude that there must exist states $q_0, q_1, \ldots, q_n \in Q_\xi$ such that $q_0 \xrightarrow{a_1}_\xi q_1 \xrightarrow{a_2}_\xi q_2 \ldots q_{n-1} \xrightarrow{a_n}_\xi q_n$. Furthermore, we will be able to conclude that $M, \tau\tau_1'' \models \alpha$ for every τ_1'' such that $\varepsilon \preceq \tau_1'' \prec \tau_1$. Finally, we will also be assured that $(q_n, 1) \in \widehat{X}_{\tau\tau_1}$.

Now suppose $q_n \in F_\xi$ and $\beta \in A_{\tau\tau_1}$. Then $\tau_1 \in ||\pi||$ and $M, \tau\tau_1 \models \beta$ by the induction hypothesis. Consequently $M, \tau \models \alpha\,\mathcal{U}^\pi\beta$. So assume that $q_n \notin F_\xi$ or $\beta \notin A_{\tau\tau_1}$. Then $\alpha \in A_{\tau\tau_1}$ (by (1.iv)) and hence $M, \tau\tau_1 \models \alpha$ by the induction hypothesis. Now by the choice of τ_1, we know that $(x_{\tau\tau_1}, f_{\tau\tau_1}) = (0, \checkmark)$ and hence $(x_{\tau\tau_1 b_1}, f_{\tau\tau_1 b_1}) = (1, \downarrow)$ by (2.iii). On the other hand, $\rho(\tau\tau_1) \overset{b_1}{\Longrightarrow} \rho(\tau\tau_1 b_1)$ implies that there exists $q_1' \in Q_\xi$ such that $q_n \xrightarrow{b_1}_\xi q_1'$ and $(q_1', 1) \in \widehat{X}_{\tau\tau_1 b_1}$. Again $q_1' \in F_\xi$ and $\beta \in A_{\tau\tau_1 b_1}$ will lead to the desired conclusion $M, \tau \models \alpha\,\mathcal{U}^\pi\beta$.

So suppose $q_1' \notin F_\xi$ or $\beta \notin A_{\tau\tau_1 b_1}$. Then as before, $\alpha \in A_{\tau\tau_1 b_1}$ and hence $M, \tau\tau_1 b_1 \models \alpha$ by induction hypothesis. By the choice of τ_2 we are assured that $m \geq 2$ because $f_{\tau\tau_1 b_1} = \downarrow$. So consider $\rho(\tau\tau_1 b_1) \overset{b_2}{\Longrightarrow} \rho(\tau\tau_1 b_1 b_2)$. Then again it follows easily that there must exist $q_2' \in Q_\xi$ such that $q_1' \xrightarrow{b_2}_\xi q_2'$ and $(q_2', 1) \in \widehat{X}_{\tau\tau_1 b_1 b_2}$. If $q_2' \in F_\xi$ and $\beta \in A_{\tau\tau_1 b_1 b_2}$ then we will at once obtain $M, \tau \models \alpha\,\mathcal{U}^\pi\beta$. If not, the facts that $(q_1', 1) \in \widehat{X}_{\tau\tau_1 b_1}$ and that $q_1' \notin F_\xi$ or $\beta \notin A_{\tau\tau_1 b_1}$ holds, guarantee us that $f_{\tau\tau_1 b_1 b_2} = \downarrow$ by (2.iii). Hence $m \geq 3$. Carrying on this way we will eventually exhaust all of τ_2 and while doing so, reach the desired conclusion $M, \tau \models \alpha\,\mathcal{U}^\pi\beta$. □

Lemma 6.4.2 *Suppose α_0 is satisfiable. Then $L(\mathcal{B}) \neq \emptyset$.*

Proof: Since our logic has no past modalities it is easy to see that if α_0 is satisfiable then there exists a model $M = (\sigma, V)$ such that $M, \varepsilon \models \alpha_0$. We shall show that $\sigma \in L(\mathcal{B})$ by constructing a map $\rho : \mathrm{prf}(\sigma) \longrightarrow S$ so that ρ is an accepting run of $\mathcal{B}$ over σ. For each $\tau \in \mathrm{prf}(\sigma)$ we set $\rho(\tau) = (A_\tau, X_\tau, \widehat{X}_\tau, x_\tau, f_\tau)$ and define ρ in a componentwise manner.

For each $\tau \in \mathrm{prf}(\sigma)$ define A_τ via:

$$A_\tau = \{\alpha \mid M, \tau \models \alpha\}.$$

For each $\tau \in \mathrm{prf}(\sigma)$ define X_τ as follows. Suppose $\xi = \alpha\,\mathcal{U}^\pi\beta$ and $q \in Q_\xi$. Then $q \in X_\tau$ iff there exists a pair (τ', R') such that:

- $\tau\tau' \in \mathrm{prf}(\sigma)$ and $M, \tau\tau' \models \beta$.
- For every τ'', if $\varepsilon \preceq \tau'' \prec \tau'$ then $M, \tau\tau'' \models \alpha$.
- $R' : \mathrm{prf}(\tau') \longrightarrow Q_\xi$ such that $R'(\varepsilon) = q$ and $R'(\tau') \in F_\xi$ and $R'(\tau'') \xrightarrow{a}_\xi R'(\tau''a)$ for every $\tau''a \in \mathrm{prf}(\tau')$.

To define the remaining three components we will first define the fourth and fifth components by mutual induction. To this end we shall make use of some terminology.

We shall call the pair (τ, ξ) an *obligation in M* if $\tau \in \mathrm{prf}(\sigma)$ and $\xi = \alpha\,\mathcal{U}^\pi\beta \in Req$ such that $M, \tau \models \alpha\,\mathcal{U}^\pi\beta$ but $M, \tau \not\models \beta$ or $\varepsilon \notin ||\pi||$. Let (τ, ξ) be

an obligation in M. We shall say that the pair (τ', R') is a *witness* for (τ, ξ) iff the following conditions are satisfied:

- $\tau\tau' \in \mathrm{prf}(\sigma)$ and $M, \tau\tau' \models \beta$ and for every τ'', $\varepsilon \preceq \tau'' \prec \tau'$ implies $M, \tau\tau'' \models \alpha$.
- $\tau' \in ||\pi||$ and $R' : \mathrm{prf}(\tau') \longrightarrow Q_\xi$ such that $R'(\varepsilon) \in I_\xi$, $R'(\tau') \in F_\xi$ and $R'(\tau'') \xrightarrow{a}_\xi R'(\tau''a)$ for every $\tau''a \in \mathrm{prf}(\tau')$.

Note that if (τ', R') is a witness for the obligation (τ, ξ) then $\tau' \neq \varepsilon$. We shall fix a *chronicle set* CH for M. It is a set of quadruples which satisfies the following conditions:

- If $(\tau, \xi, \tau', R') \in CH$ then (τ, ξ) is an obligation in M and (τ', R') is witness for (τ, ξ).
- If (τ, ξ) is an obligation in M then $(\tau, \xi, \tau', R') \in CH$ for some witness (τ', R') for (τ, ξ).
- If $(\tau, \xi, \tau', R'), (\tau, \xi, \tau'', R'') \in CH$ then $(\tau', R') = (\tau'', R'')$.

It is easy to check that CH exists. (In fact it can be chosen in a canonical manner by fixing a lexicographic order on Q_ξ for each $\xi \in Req$).

With these definitions in place, we are now prepared to define the fourth and the fifth components of ρ by induction on τ. For the base case, we set $(x_\varepsilon, f_\varepsilon) = (0, \checkmark)$. Now consider the induction step where $\tau = \tau_0 a$ and assume that $(x_{\tau'}, f_{\tau'})$ is defined for every $\tau' \in \mathrm{prf}(\tau_0)$. If $f_{\tau_0} = \checkmark$ then $(x_\tau, f_\tau) = (1 - x_{\tau_0}, \downarrow)$. Suppose $f_{\tau_0} = \downarrow$. Then $(x_\tau, f_\tau) = (x_{\tau_0}, \downarrow)$ if there exists $(\tau_1, \xi_1, \tau_1', R_1') \in CH$ such that $\tau_1 \preceq \tau_0 \prec \tau_1\tau_1'$ and $x_{\tau_1} = 1 - x_{\tau_0}$. Otherwise, $f_\tau = \checkmark$ and $x_\tau = x_{\tau_0}$.

Finally, the third component of ρ can now be defined. For each $\tau \in \mathrm{prf}(\sigma)$, we define $\widehat{X}_\tau$ as follows. Suppose $\xi \in Req$ and $q \in Q_\xi$ and $z \in \{0, 1\}$. Then $(q, z) \in \widehat{X}_\tau$ iff there exists $(\tau_1, \xi, \tau_1', R_1') \in CH$ such that for some $\tau_1'' \in \mathrm{prf}(\tau_1')$, $\tau_1 \preceq \tau = \tau_1\tau_1''$. Moreover, $R_1'(\tau_1'') = q$ and $x_{\tau_1} = 1 - z$.

We now wish to argue that $\rho : \mathrm{prf}(\sigma) \longrightarrow S$ is an accepting run of $\mathcal{B}$ over σ. First we shall show that ρ is well defined. Let $\tau \in \mathrm{prf}(\sigma)$ be given. We must show that $\rho(\tau) \in S$. It is easy to see that A_τ is an atom, $X_\tau \subseteq Q$, $\widehat{X}_\tau \subseteq \widehat{Q}$, $x_\tau \in \{0, 1\}$ and $f_\tau \in \{\downarrow, \checkmark\}$. We will show that $\rho(\tau)$ satisfies all the clauses of the definition of $\mathcal{B}$.

So fix some $\alpha\, \mathcal{U}^\pi \beta = \xi$. Assume initially that $\beta \in A_\tau$ and $q \in F_\xi$. Then $M, \tau \models \beta$ by definition of A_τ. Now consider the pair (τ', R') where $\tau' = \varepsilon$ and $R'(\varepsilon) = q$. From the definition of X_τ it now follows that $q \in X_\tau$. Thus $F_\xi \subseteq X_\tau$ as required by (1.i).

Next assume that $\alpha \in A_\tau$ and $q \in X_\tau$ for some $q \in I_\xi$. From the definition of X_τ it follows that there exists a pair (τ', R') such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $M, \tau\tau' \models \beta$ and $M, \tau\tau'' \models \alpha$ for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$. Furthermore, $R' : \mathrm{prf}(\tau') \longrightarrow Q_\xi$ such that $R'(\varepsilon) = q$ and $R'(\tau') \in F_\xi$ and $R'(\tau'') \xrightarrow{a}_\xi R'(\tau''a)$ for every $\tau''a \in \mathrm{prf}(\tau')$. But from the assumption that $q \in I_\xi$ we have

that $\tau' \in ||\pi||$, because R' is an accepting run of $\mathcal{A}_\xi$ over τ'. Consequently $M, \tau \models \alpha\, \mathcal{U}^\pi \beta$ and this leads to the conclusion that $\alpha\, \mathcal{U}^\pi \beta \in A_\tau$ as required by (1.ii).

Next assume that $\alpha\, \mathcal{U}^\pi \beta \in A_\tau$ and $\beta \notin A_\tau$ or $\varepsilon \notin ||\pi||$. Then (τ, ξ) is an obligation in M since by the definition of A_τ, $M, \tau \models \alpha\, \mathcal{U}^\pi \beta$ but $M, \tau \not\models \beta$ or $\varepsilon \notin ||\pi||$. Hence there exists $(\tau, \xi, \tau', R') \in CH$. Let $R'(\varepsilon) = q$. From the fact that (τ', R') is a witness for (τ, ξ) we have that $q \in I_\xi$. Moreover, by the definition of $\widehat{X}_\tau$ and from $\tau \preceq \tau = \tau$ (i.e. $\tau_1 = \tau$ and $\tau_1'' = \varepsilon$), it follows that $(q, 1 - x_\tau) \in \widehat{X}_\tau$ as required by (1.iii).

Finally suppose that $(q, z) \in \widehat{X}_\tau$ with $q \in Q_\xi$ such that $q \notin F_\xi$ or $\beta \notin A$. Now $(q, z) \in \widehat{X}_\tau$ implies, by the definition of $\widehat{X}_\tau$, that there exists $(\tau_1, \xi, \tau_1', R_1') \in CH$ such that for some $\tau_1'' \in \mathrm{prf}(\tau_1')$, $\tau_1 \preceq \tau = \tau_1 \tau_1''$ and $R_1'(\tau_1'') = q$ and $x_{\tau_1} = 1 - z$. But (τ_1', R_1') is a witness for the obligation (τ_1, ξ) and hence $R_1'(\tau_1') \in F_\xi$ and $M, \tau_1 \tau_1' \models \beta$. Since $\beta \notin A_\tau$ or $q \notin F_\xi$ it must be the case that $\tau_1'' \prec \tau_1'$ and hence $M, \tau_1 \tau_1'' \models \alpha$. But then $\tau = \tau_1 \tau_1''$ now leads to $\alpha \in A_\tau$ as required by (1.iv).

We have now shown that ρ is well defined. Next we wish to show that ρ is a run of $\mathcal{B}$ over σ. Since $M, \varepsilon \models \alpha_0$ we have $\alpha_0 \in A_\varepsilon$. By definition, $(x_\varepsilon, f_\varepsilon) = (0, \checkmark)$. Hence $\rho(\varepsilon) \in S_{in}$.

Now suppose $\tau a \in \mathrm{prf}(\sigma)$. We must show that $\rho(\tau) \stackrel{a}{\Longrightarrow} \rho(\tau a)$. For this purpose we fix $\alpha\, \mathcal{U}^\pi \beta = \xi \in Req$. Suppose $q, q' \in Q_\xi$ with $q' \in X_{\tau a}$ such that $q \stackrel{a}{\longrightarrow}_\xi q'$. Further suppose $\alpha \in A_\tau$. Now $q' \in X_{\tau a}$ implies that there exists a pair (τ', R') such that $R'(\varepsilon) = q'$ and $R'(\tau') \in F_\xi$ and $R'(\tau'') \stackrel{b}{\longrightarrow}_\xi R'(\tau'' b)$ for every $\tau'' b \in \mathrm{prf}(\tau')$. Furthermore, $M, \tau a \tau' \models \beta$ and $M, \tau a \tau'' \models \alpha$ for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$. Now consider the pair $(a\tau', R_a')$ where $R_a' : \mathrm{prf}(a\tau') \longrightarrow Q_\xi$ is given as $R_a'(\varepsilon) = q$ and for every $\tau'' \in \mathrm{prf}(\tau')$, $R_a'(a\tau'') = R'(\tau'')$. From $M, \tau \models \alpha$ (as $\alpha \in A_\tau$ by assumption) it now follows at once that $q \in X_\tau$ as required by (2.i).

Suppose now that $q \in Q_\xi$ and $(q, z) \in \widehat{X}_\tau$ but $q \notin F_\xi$ or $\beta \notin A_\tau$. Since $(q, z) \in \widehat{X}_\tau$ there must exist $(\tau_1, \xi, \tau_1', R_1') \in CH$ and $\tau_1'' \in \mathrm{prf}(\tau_1')$ such that $\tau_1 \preceq \tau = \tau_1 \tau_1''$ and $x_{\tau_1} = 1 - z$ and $R_1'(\tau_1'') = q$. But (τ_1', R_1') is a witness for (τ_1, ξ) and hence $R_1'(\tau_1') \in F_\xi$ and $M, \tau_1 \tau_1' \models \beta$. Consequently $\tau_1'' \prec \tau_1'$ and thus $\tau_1'' a \in \mathrm{prf}(\tau_1')$ for the unique a. This implies that $R_1'(\tau_1'') \stackrel{a}{\longrightarrow}_\xi R_1'(\tau_1'' a)$. Let $R_1'(\tau_1'' a) = q'$. Then $q \stackrel{a}{\longrightarrow}_\xi q'$. But then it follows directly from the definition of $X_{\tau a}$, that $(q', 1 - z) \in \widehat{X}_{\tau a}$ as required by (2.ii).

Next suppose that $f_\tau = \checkmark$. Then clearly $(x_{\tau a}, f_{\tau a}) = (1 - x_\tau, \downarrow)$ by the definition of ρ. So assume that $f_\tau = \downarrow$. Supposing there exists $\alpha\, \mathcal{U}^\pi \beta = \xi$ in Req and there exists $q \in Q_\xi$ such that $(q, z) \in \widehat{X}_\tau$ where $z = x_\tau$. Further suppose $q \notin F_\xi$ or $\beta \notin A_\tau$. Now $(q, z) \in \widehat{X}_\tau$ implies that there exists $(\tau_1, \xi, \tau_1', R1') \in CH$ such that $\tau_1 \preceq \tau = \tau_1 \tau_1''$ for some $\tau_1'' \in \mathrm{prf}(\tau_1')$ with the further property that $x_{\tau_1} = 1 - z$. From the definitions and the fact that $q \notin F_\xi$ or $\beta \notin A_\tau$ it follows that $\tau_1 \preceq \tau \prec \tau_1 \tau_1'$. Hence by the definition of ρ it follows that $(x_{\tau a}, f_{\tau a}) = (x_\tau, \downarrow)$ as required by (2.iii). On the other hand, if such a $(q, z) \in \widehat{X}_\tau$ does not exist, then it follows directly from the definition that $(x_{\tau a}, f_{\tau a}) = (x_\tau, \checkmark)$ as

required by (2.iii).

We have now verified that ρ is a run of $\mathcal{B}$ over σ. To show that ρ is accepting it suffices to prove that for any $\tau \in \mathrm{prf}(\sigma)$ there exists τ' such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $f_{\tau\tau'} = \checkmark$.

Case 1: $(x_\tau, f_\tau) = (0, \checkmark)$. By picking $\tau' = \varepsilon$ the desired conclusion follows trivially.

Case 2: $(x_\tau, f_\tau) = (0, \downarrow)$. Define the set $\Gamma_\tau \subseteq CH$ as follows. Let (τ, ξ, τ', R') be a member of the chronicle set CH. Then $(\tau_1, \xi_1, \tau_1', R_1') \in \Gamma_\tau$ iff $\tau_1 \preceq \tau \prec \tau_1\tau_1'$ and $x_{\tau_1} = 1$. Now if $\Gamma_\tau = \emptyset$ then it is easy to see that with $\tau' = a$ where $\tau a \in \mathrm{prf}(\sigma)$ we must have $f_{\tau\tau'} = \checkmark$ as required.

So suppose $\Gamma_\tau \neq \emptyset$. Define, for each $ch = (\tau_1, \xi_1, \tau_1', R_1') \in \Gamma_\tau$, $k_{ch} = |\tau_1\tau_1'| - |\tau|$ and set $k_\tau = \max(\{k_{ch}\}_{ch \in \Gamma_\tau})$. Let $\tau a \in \mathrm{prf}(\sigma)$. Then it is easy to see that $(x_{\tau a}, f_{\tau a}) = (0, \downarrow)$. But it is also easy to verify $\Gamma_{\tau a} = \emptyset$ or $k_{\tau a} < k_\tau$. Proceeding in this way the required conclusion can be drawn eventually.

The two other cases can be resolved by similar arguments. □

It is now straightforward to establish the main result of this section. To start with we define the size of a formula α, denoted $|\alpha|$, via:

- $|p| = 1$, $|\neg\alpha| = 1 + |\alpha|$ and $|\alpha \vee \beta| = 1 + |\alpha| + |\beta|$.
- $|\alpha\, \mathcal{U}^\pi \beta| = 1 + |\alpha| + |\pi| + |\beta|$,

where $|\pi|$ is given by $|a| = 1$, $|\pi + \pi'| = |\pi; \pi'| = 1 + |\pi| + |\pi'|$ and $|\pi^*| = 1 + |\pi|$.

Theorem 6.4.3 *For each $\alpha \in \mathrm{DLTL}(\Sigma)$ the question whether or not α is satisfiable can be decided in time $2^{O(|\alpha|)}$.*

Proof: Let $\alpha_0 \in \mathrm{DLTL}(\Sigma)$. Then α_0 is satisfiable iff $L(\mathcal{B}_{\alpha_0}) \neq \emptyset$ where α_0 is the Büchi automaton constructed above. The emptiness problem for $\mathcal{B}_{\alpha_0}$ can be settled in time $O(|S|)$ where S is the set of states of $\mathcal{B}$ [140].

Clearly $CL(\alpha_0)$ is linear in the size of α_0 and hence $|AT| = 2^{O(|\alpha_0|)}$. Let $\alpha\, \mathcal{U}^\pi \beta \in Req$. It is known that for $\pi \in \mathrm{Prg}(\Sigma)$, we can construct in polynomial time a non-deterministic finite state automaton $\mathcal{A}_\xi$ with $L(\mathcal{A}) = ||\pi||$ such that $|Q_\xi|$ is linear in the size of π (see [64] for a recent account on converting regular expression to small finite state automata).

Let $Req = \{\alpha_1\, \mathcal{U}^{\pi_1} \beta_1, \ldots, \alpha_m\, \mathcal{U}^{\pi_m} \beta_m\}$. Then $|\pi_1| + |\pi_2| + \cdots + |\pi_m| \leq |\alpha_0|$. Consequently both Q and $\widehat{Q}$ are linear in the size of α_0. It is now easy to see that $|S| = 2^{O(|\alpha_0|)}$. □

As usual, the decision procedure can be applied to solve the associated model checking problem but we will not enter into details here.

6.5 Some Expressiveness Results

Our main goal here is to show that $\mathrm{DLTL}(\Sigma)$ has the same expressive power as the monadic second-order theory of infinite sequences over Σ. Towards the end

of the section we will also establish that a natural sublogic of DLTL(Σ) captures the first-order theory of infinite sequences over Σ.

In order to obtain clean formulations of the expressiveness results, we shall banish atomic propositions through the rest of the chapter. Instead, we will just work with the constant tt and its negation $\neg tt \stackrel{\text{def}}{=} ff$. To be precise, the syntax of DLTL(Σ) will be from now on assumed to be:

$$\text{DLTL}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha\, \mathcal{U}^{\pi} \beta,$$

where $\pi \in \text{Prg}(\Sigma)$ with Prg(Σ) defined as before.

A model is now just a ω-sequence $\sigma \in \Sigma^{\omega}$. For $\tau \in \text{prf}(\sigma)$ we define $\sigma, \tau \models \alpha$ via:

- $\sigma, \tau \models tt$.
- All the other clauses are filled in exactly as in Section 6.3 while replacing M by σ in the appropriate places.

Each formula α now defines a ω-language $L_\alpha \subseteq \Sigma^{\omega}$ given by:

$$L_\alpha = \{\sigma \mid \sigma, \varepsilon \models \alpha\}.$$

We say that $L \subseteq \Sigma^{\omega}$ is DLTL(Σ)-*definable* iff there exists some $\alpha \in \text{DLTL}(\Sigma)$ such that $L = L_\alpha$.

The monadic second-order theory of infinite sequences over Σ is denoted MSO(Σ). Its vocabulary consists of a family of unary predicates $\{R_a\}_{a \in \Sigma}$, one for each $a \in \Sigma$; a binary predicate $\leq$; a binary predicate $\in$; a countable supply of individual variables $Var = \{x, y, z, \ldots\}$; a countable supply of set variables (i.e. monadic predicate variables) $SVar = \{X, Y, Z, \ldots\}$. The formulas of MSO($\Sigma$) are then built up by:

- $R_a(x)$, $x \leq y$ and $x \in X$ are atomic formulas.
- If ϕ and ϕ' are formulas then so are $\neg\phi$, $\phi \vee \phi'$, $(\exists x)\phi$ and $(\exists X)\phi$.

A structure for MSO(Σ) is a ω-sequence $\sigma \in \Sigma^{\omega}$. Let $\mathcal{I}$ be an interpretation of the variables with $\mathcal{I} : Var \longrightarrow \omega$ and $\mathcal{I} : SVar \longrightarrow 2^{\omega}$. Then the notion of σ being a model of ϕ under the interpretation $\mathcal{I}$, denoted $\sigma \models_{\mathcal{I}} \phi$, is defined in the expected manner. In particular, $\sigma \models_{\mathcal{I}} R_a(x)$ iff $\sigma(\mathcal{I}(x)) = a$ (note that $\sigma \in \Sigma^{\omega}$ is viewed as $\sigma : \omega \longrightarrow \Sigma$); $\sigma \models_{\mathcal{I}} x \leq y$ iff $\mathcal{I}(x) \leq \mathcal{I}(y)$ (here $\leq$ is the usual ordering over ω); $\sigma \models_{\mathcal{I}} x \in X$ iff $\mathcal{I}(x) \in \mathcal{I}(X)$.

As usual, a sentence is a formula with no free variables. Each sentence ϕ defines a ω-language denoted L_ϕ where:

$$L_\phi = \{\sigma \mid \sigma \models \phi\}.$$

We say that $L \subseteq \Sigma^{\omega}$ is MSO(Σ)-*definable* iff there exists a sentence $\phi \in \text{MSO}(\Sigma)$ such that $L = L_\phi$.

Lemma 6.5.1 *Let $L \subseteq \Sigma^\omega$. If L is* DLTL(Σ)*-definable then L is* MSO(Σ)*-definable.*

Proof: Consider the construction from the previous section which associates a Büchi automaton $\mathcal{B}_{\alpha_0}$ with each formula $\alpha_0 \in \mathrm{DLTL}(\Sigma)$. Suppose we apply this construction to formulas arising from the restricted syntax assumed in the present section. Then it is easy to see that, in the absence of atomic propositions, $L_{\alpha_0} = L(\mathcal{B}_{\alpha_0})$. But then the classic result of Büchi [14] asserts that $L \subseteq \Sigma^\omega$ is MSO(Σ)-definable iff there exists a Büchi automaton $\mathcal{B}$ operating over Σ such that $L = L(\mathcal{B})$. □

Next we wish to show that if $L \subseteq \Sigma^\omega$ is MSO(Σ)-definable then L is DLTL(Σ)-definable. In fact, it turns out that it suffices to consider a natural fragment of DLTL(Σ) denoted $\mathrm{DLTL}^-(\Sigma)$ whose syntax is given by:

$$\mathrm{DLTL}^-(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle\pi\rangle\alpha,$$

where $\pi \in \mathrm{Prg}(\Sigma)$.

Here $\langle\pi\rangle\alpha$ is interpreted as $tt\,\mathcal{U}^\pi\,\alpha$ with the resulting semantics. Thus DLTL^- is PDL equipped with a linear time semantics. As before $L \subseteq \Sigma^\omega$ is said to be $\mathrm{DLTL}^-(\Sigma)$*-definable* iff there exists $\alpha \in \mathrm{DLTL}^-(\Sigma)$ such that $L = L_\alpha$, where L_α is defined as for DLTL(Σ). To get at the result we are after we need to work with Muller automata operating over Σ of the form $\mathcal{M} = (Q, \longrightarrow, Q_{in}, \mathcal{F})$ where:

- $Q, \longrightarrow$ and Q_{in} are as in the case of a Büchi automaton.
- $\mathcal{F} \subseteq 2^Q$ is a family of accepting sets of states.

Let $\sigma \in \Sigma^\omega$. Then the notion of a run $\rho : \mathrm{prf}(\sigma) \longrightarrow Q$ of $\mathcal{M}$ over σ is as in the case of a Büchi automaton. The definition of $\inf(\rho)$ is also as before. The run ρ is said to be accepting iff $\inf(\rho) \in \mathcal{F}$. Naturally $L(\mathcal{M})$, the ω-language accepted by $\mathcal{M}$, is given by : $\sigma \in L(\mathcal{M})$ iff there exists an accepting run of $\mathcal{M}$ over σ.

The Muller automaton $\mathcal{M} = (Q, \longrightarrow, Q_{in}, \mathcal{F})$ is deterministic iff $|Q_{in}| = 1$ and whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$, we have $q' = q''$. The well-known theorem of McNaughton [87] guarantees that $L \subseteq \Sigma^\omega$ is MSO(Σ)-definable iff there exists a deterministic Muller automaton operating over Σ such that $L = L(\mathcal{M})$. This fact will be the basis for the proof of the next result.

Lemma 6.5.2 *Let $L \subseteq \Sigma^\omega$. If L is* MSO(Σ)*-definable then L is* $\mathrm{DLTL}^-(\Sigma)$*-definable.*

Proof: As remarked above, L is MSO(Σ)-definable implies that there exists a *deterministic* Muller automaton $\mathcal{M} = (Q, \longrightarrow, \{q_{in}\}, \mathcal{F})$ operating over Σ such that $L = L(\mathcal{M})$. We will exhibit a formula $\alpha_{\mathcal{M}} \in \mathrm{DLTL}^-(\Sigma)$ such that $L_{\alpha_{\mathcal{M}}} = L(\mathcal{M})$.

An easy argument shows that it involves no loss of generality to assume that $\mathcal{M}$ — apart from determinacy — has two additional properties:

(i) $\emptyset \notin \mathcal{F}$.

(ii) $\forall q \in Q\ \forall a \in \Sigma. \quad \exists q'.\ q \xrightarrow{a} q'$.

Determinacy and (ii) ensure that for *every* $\sigma \in \Sigma^\omega$ the Muller automaton $\mathcal{M}$ has a *unique* run over σ. This fact will be crucial in what follows.

If $\mathcal{F} = \emptyset$ we have that $L = \emptyset$, so we set $\alpha_\mathcal{M} = ff$. So suppose that $\mathcal{F} \neq \emptyset$. For each $F \in \mathcal{F}$ we shall construct a formula α_F expressing acceptance by F. The required formula $\alpha_\mathcal{M}$ defining L will then be the disjunction of all such α_F.

First we extend $\longrightarrow \subseteq Q \times \Sigma \times Q$ to $\longrightarrow_*$, where $\longrightarrow_*$ is the least subset of $Q \times \Sigma^* \times Q$ satisfying:

- $q \xrightarrow{\varepsilon}_* q$ for every $q \in Q$.
- If $q \xrightarrow{\tau}_* q'$ and $q' \xrightarrow{a} q''$ then $q \xrightarrow{\tau a}_* q''$.

Next define, for each $q, q' \in Q$, the language of finite words $L_{q,q'} \subseteq \Sigma^*$ by:

$$L_{q,q'} = \{\tau \mid q \xrightarrow{\tau}_* q'\}.$$

It is easy to see that each $L_{q,q'}$ is a regular subset of Σ^*. Hence we can fix a regular expression $\pi_{q,q'} \in \mathrm{Prg}(\Sigma)$ such that $L_{q,q'} = ||\pi_{q,q'}||$. Due to the determinacy of $\mathcal{M}$ it follows at once that if $q, q', q'' \in Q$ such that $L_{q,q'} \cap L_{q,q''} \neq \emptyset$ then $q' = q''$.

Now let $F = \{q_0, q_1, \ldots, q_{n-1}\}$ with $n \geq 1$. Then the formula α_F is given by:

$$\alpha_F = \bigvee_{q \in F} \langle \pi_{q_{in},q} \rangle \left(\bigwedge_{q' \notin F} [\pi_{q,q'}] ff \ \wedge \ \bigwedge_{j=0}^{n-1} [\pi_{q,q_j}] \langle \pi_{q_j,q_{j\oplus 1}} \rangle tt \right),$$

where $\oplus$ denotes addition modulo n.

The required formula $\alpha_\mathcal{M}$ describing $L(\mathcal{M})$ is then defined as:

$$\alpha_\mathcal{M} = \bigvee_{F \in \mathcal{F}} \alpha_F$$

Clearly $\alpha_\mathcal{M} \in \mathrm{DLTL}^-(\Sigma)$. It is easy to check that $L_{\alpha_\mathcal{M}} = L(\mathcal{M})$. □

Theorem 6.5.3 *Let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent:*

(i) *L is* MSO(Σ)*-definable.*

(ii) *L is* DLTL(Σ)*-definable.*

(iii) *L is* DLTL$^-$(Σ)*-definable.*

Proof: Follows immediately from Lemmas 6.5.1 and 6.5.2 and the fact that DLTL$^-(\Sigma)$ is a sublogic of DLTL(Σ). □

At present we do not know of a direct translation of DLTL(Σ)-formulas into DLTL$^-$(Σ)-formulas. Although these two logics have the same expressive power in the sense of Theorem 6.5.3, it appears that DLTL(Σ) will admit more natural specifications. In addition, it is a conservative extension of LTL(Σ) even from a syntactic standpoint and hence conventional LTL specifications can be brought in with no overhead translation costs.

We shall conclude this section by pointing out that star-free programs can be used to capture the first-order definable subsets of Σ^ω. Admittedly this is not a big surprise, but it illustrates once more that our method of augmenting the expressive power of LTL is a natural one.

FO(Σ) will denote the first-order theory of ω-sequences generated by Σ. It is the fragment of MSO(Σ) obtained by eliminating set variables from the syntax. We shall say that $L \subseteq \Sigma^\omega$ is FO(Σ)-*definable* iff there exists a sentence ϕ in FO(Σ) such that $L = L_\phi$.

The set of *star-free regular programs* over Σ is denoted $\mathrm{Prg}_{\mathbb{SF}}(\Sigma)$ and its syntax is given by:

$$\mathrm{Prg}_{\mathbb{SF}}(\Sigma) ::= 0 \mid a \mid \pi + \pi' \mid \pi;\pi' \mid \overline{\pi}.$$

The set of finite words denoted by each star-free program is obtained via the map $||\cdot|| : \mathrm{Prg}_{\mathbb{SF}}(\Sigma) \longrightarrow 2^{\Sigma^*}$ which is defined as follows: $||\overline{\pi}|| = \Sigma^* - ||\pi||$ and $||0|| = \emptyset$. The remaining cases are handled as before.

The star-free version of DLTL(Σ) will be denoted — for want of a better notation — by $\mathrm{DLTL}_{\mathbb{SF}}(\Sigma)$ and its syntax is given by:

$$\mathrm{DLTL}_{\mathbb{SF}}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha\, \mathcal{U}^\pi \beta \quad (\pi \in \mathrm{Prg}_{\mathbb{SF}}(\Sigma)).$$

Thus the only difference is that the programs that are used to build up the until-formulas are required to be star-free programs. The fragment of $\mathrm{DLTL}_{\mathbb{SF}}(\Sigma)$ which corresponds to DLTL$^-$(Σ) has the syntax:

$$\mathrm{DLTL}^-_{\mathbb{SF}}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle\pi\rangle\alpha \quad (\pi \in \mathrm{Prg}_{\mathbb{SF}}(\Sigma)).$$

Theorem 6.5.4 *Let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent:*

(i) *L is* FO(Σ)*-definable.*

(ii) *L is* $\mathrm{DLTL}_{\mathbb{SF}}(\Sigma)$*-definable.*

(iii) *L is* $\mathrm{DLTL}^-_{\mathbb{SF}}(\Sigma)$*-definable.*

Proof: Trivially (iii) implies (ii). The proof that (ii) implies (i) utilizes the well-known fact [88] that FO(Σ)-definable languages over finite strings and the languages described by star-free regular expressions coincide. It is then straightforward to exhibit a syntactic translation of formulas of $\mathrm{DLTL}_{\mathbb{SF}}(\Sigma)$ to FO(Σ) essentially re-expressing the semantics by relativizing the formulas arising from the star-free expressions. The details can be found in [59].

That (i) implies (iii) is a consequence of the fact that the abovementioned characterization of FO(Σ) and star-free regular expressions can be extended

to languages of ω-sequences [140]. A linear translation from the star-free ω-regular expressions to $\mathrm{DLTL}^{-}_{\mathbb{SF}}(\Sigma)$ is then obtained by inductively translating the boolean operations to their logical counterparts, while left concatenation with a star-free language of finite strings is handled by the $\langle\pi\rangle$-modality. Once again, the details can be found in [59]. □

6.6 Axiomatizations

Our axiomatization of the set of valid formulas of DLTL is an extension of Segerberg's axiomatization of PDL [124]. Moreover, our completeness argument is based on the elegant proof of completeness of Segerberg's axioms due to Kozen and Parikh [76]. It will be convenient to first axiomatize DLTL^{-}.

We begin by augmenting the set of regular programs with the atomic program 1. We set $||1|| = \{\varepsilon\}$. By abuse of notation this augmented set of programs will also be denoted as $\mathrm{Prg}(\Sigma)$. Next we define the transition relation $\longrightarrow_{\mathrm{Prg}(\Sigma)}$ (from now on written as just $\longrightarrow$) to be the least subset of $\mathrm{Prg}(\Sigma) \times \Sigma \times \mathrm{Prg}(\Sigma)$ yielded by the following rules:

- $\dfrac{}{a \stackrel{a}{\longrightarrow} 1}$
- $\dfrac{\pi \stackrel{a}{\longrightarrow} \pi_1}{\pi + \pi' \stackrel{a}{\longrightarrow} \pi_1} \qquad \dfrac{\pi \stackrel{a}{\longrightarrow} \pi_1}{\pi' + \pi \stackrel{a}{\longrightarrow} \pi_1}$
- $\dfrac{\pi \stackrel{a}{\longrightarrow} \pi_1}{\pi;\pi' \stackrel{a}{\longrightarrow} \pi_1;\pi'}$ if $\pi_1 \neq 1$
- $\dfrac{\pi \stackrel{a}{\longrightarrow} 1}{\pi;\pi' \stackrel{a}{\longrightarrow} \pi'}$
- $\dfrac{\pi' \stackrel{a}{\longrightarrow} \pi''}{\pi;\pi' \stackrel{a}{\longrightarrow} \pi''}$ if $\varepsilon \in ||\pi||$
- $\dfrac{\pi \stackrel{a}{\longrightarrow} \pi'}{\pi^* \stackrel{a}{\longrightarrow} \pi';\pi^*}$.

This transition relation is extended to the relation $\longrightarrow_* \subseteq \mathrm{Prg}(\Sigma) \times \Sigma^* \times \mathrm{Prg}(\Sigma)$ via:

- $\pi \stackrel{\varepsilon}{\longrightarrow}_* \pi$
- If $\pi \stackrel{\tau}{\longrightarrow}_* \pi'$ and $\pi' \stackrel{a}{\longrightarrow} \pi''$ then $\pi \stackrel{\tau a}{\longrightarrow}_* \pi''$.

Finally the sets of programs $\delta_a(\pi)$ and $\delta_*(\pi)$ for each π and each a are defined as follows:

- $\delta_a(\pi) = \{\pi' \mid \pi \stackrel{a}{\longrightarrow} \pi'\}$.

- $\delta_*(\pi) = \{\pi' \mid \exists \tau.\ \pi \xrightarrow{\tau}_* \pi'\}$.

Proposition 6.6.1 *For each π and each a, both $\delta_a(\pi)$ and $\delta_*(\pi)$ are finite sets.*

Proof: The proof follows easily by structural induction on π. □

We are now ready to present an axiomatization of DLTL$^-$ (Recall that $O\alpha \stackrel{\text{def}}{=} \bigvee_{a\in\Sigma}\langle a\rangle\alpha$). The logical system $\mathcal{DLTL}^-$ is given as follows.

Axiom schemes

(A0) All the tautologies of propositional calculus.
(A1) $[\pi]\ (\alpha \Rightarrow \beta) \Rightarrow ([\pi]\alpha \Rightarrow [\pi]\beta)$.
(A2) $\langle \pi + \pi'\rangle\alpha \Leftrightarrow \langle\pi\rangle\alpha \vee \langle\pi'\rangle\alpha$.
(A3) $\langle \pi;\pi'\rangle\alpha \Leftrightarrow \langle\pi\rangle\langle\pi'\rangle\alpha$.
(A4) $\langle \pi^*\rangle\alpha \Leftrightarrow \alpha \vee \langle\pi\rangle\langle\pi^*\rangle\alpha$.
(A5) $[\pi^*](\alpha \Rightarrow [\pi]\alpha) \Rightarrow (\alpha \Rightarrow [\pi^*]\alpha)$.
(A6) $\alpha \Leftrightarrow \langle 1\rangle\alpha$.
(A7) Ott.
(A8) $\langle a\rangle tt \Rightarrow \bigwedge_{b\neq a}[b]ff$.
(A9) $\langle a\rangle\alpha \Rightarrow [a]\alpha$.
(A10) $\langle\pi\rangle\alpha \Leftrightarrow \alpha \vee \left(\bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{\pi'\in\delta_a(\pi)}\langle\pi'\rangle\alpha\right),\quad (\varepsilon \in ||\pi||)$.
(A11) $\langle\pi\rangle\alpha \Leftrightarrow \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{\pi'\in\delta_a(\pi)}\langle\pi'\rangle\alpha,\quad (\varepsilon \notin ||\pi||)$.

Inference rules

(MP) $\dfrac{\alpha \quad \alpha \Rightarrow \beta}{\beta}$.

(TG) $\dfrac{\alpha}{[\pi]\alpha}$.

(A0) through (A5) and the inference rules together constitute an axiomatization of PDL. The behaviour of 1 is captured by (A6). The remaining axiom schemes describe the linear time semantics provided for regular programs in the setting of DLTL$^-$. Due to Proposition 6.6.1 both (A10) and (A11) are well-defined. It is easy to see that the axioms are valid and that the inference rules preserve validity.

We shall say, as usual, that a formula α is ($\mathcal{DLTL}^-$-)consistent in case $\neg\alpha$ is not a thesis derivable from the system $\mathcal{DLTL}^-$. We shall prove that every consistent formula is satisfiable. To this end, fix a consistent formula α_0. Define $\widehat{cl}(\alpha_0)$ just as we defined $cl(\alpha_0)$ in Section 6.4. In addition, the following conditions are required to be satisfied:

- If $\langle\pi\rangle\alpha \in \widehat{cl}(\alpha_0)$ and $\pi' \in \delta_a(\pi)$ then$\langle\pi'\rangle\alpha, \langle a\rangle\langle\pi'\rangle\alpha \in \widehat{cl}(\alpha_0)$.
- If $\langle 1\rangle\alpha \in \widehat{cl}(\alpha_0)$ then $\alpha \in \widehat{cl}(\alpha_0)$.
- $\langle a\rangle tt \in \widehat{cl}(\alpha_0)$ for every $a \in \Sigma$.

Next define $\widehat{CL}(\alpha_0)$ as $\widehat{CL}(\alpha_0) = \widehat{cl}(\alpha_0) \cup \{\neg\beta \mid \beta \in \widehat{cl}(\alpha_0)\}$. As usual, we will identify $\neg\neg\beta$ with β in what follows.

Proposition 6.6.2 *$\widehat{CL}(\alpha_0)$ is a finite set.*

Proof: Follows at once from Proposition 6.6.1. □

In this section, an atom is a maximal consistent subset of $\widehat{CL}(\alpha_0)$. If A is an atom then $\widehat{A}$ will be the conjunction of all the formulas in A. Let AT_0 be the set of all atoms. We now define the transition system $TS_0 = (AT_0, \Longrightarrow)$ where $\Longrightarrow \subseteq AT_0 \times \Sigma \times AT_0$ is given by $A \stackrel{a}{\Longrightarrow} B$ iff $\widehat{A} \wedge \langle a \rangle \widehat{B}$ is consistent. As before, the transition relation $\Longrightarrow$ is extended to $\Longrightarrow_* \subseteq AT_0 \times \Sigma^* \times AT_0$ in the obvious way.

Lemma 6.6.3

(i) *Suppose $A, B \in AT_0$ and $\pi \in \mathrm{Prg}(\Sigma)$ such that $\widehat{A} \wedge \langle \pi \rangle \widehat{B}$ is consistent. Then there exists $\tau \in ||\pi||$ such that $A \stackrel{\tau}{\Longrightarrow}_* B$.*

(ii) *Suppose $\langle \pi \rangle \alpha \in A \in AT_0$. Then there exists $B \in AT_0$ and $\tau \in ||\pi||$ such that $\alpha \in B$ and $A \stackrel{\tau}{\Longrightarrow}_* B$.*

Proof: Part (i) can be established by just repeating the proof of [76, Lemma 1]. Now part (ii) follows easily from part (i) with the help of a few tautologies of propositional calculus. □

We are now ready to extract a model of α_0 from TS_0. We shall do so by inductively defining a map $\widehat{\rho} : \omega \longrightarrow AT_0$ and an ascending chain of sequences $\tau_0 \prec \tau_1 \prec \ldots$ where each τ_i is in Σ^*. In what follows we will denote $\widehat{\rho}(i)$ by A_i for each $i \in \omega$. We shall also assume that we have fixed an enumeration of the countable set $\widehat{CL}(\alpha_0) \times \Sigma^*$.

- $\widehat{\rho}(0) = A_0$ where $A_0 \in AT_0$ such that $\alpha_0 \in A_0$. Further, $\tau_0 = \varepsilon$.
- Assume $\widehat{\rho}(i)$ and τ_i are defined. We say that the pair $(\langle \pi \rangle \alpha, \tau)$ is a requirement at stage i provided the following conditions are satisfied:
  - $\tau \preceq \tau_i$ and $\langle \pi \rangle \alpha \in A_j$ where $|\tau| = j$.
  - For every $\tau' \in \Sigma^*$, if $\tau\tau' \preceq \tau_i$ then $\tau' \notin ||\pi||$ or $\alpha \notin A_k$ where $|\tau\tau'| = k$.

Let RQ_i be the set of requirements at stage i. Suppose that $RQ_i = \emptyset$. Let $a \in \Sigma$ such that $\langle a \rangle tt \in A_i$. The fact that such an a exists and is unique is guaranteed by (A7) and (A8). Since $\bigvee_{A \in AT_0} \widehat{A}$ is a thesis, it follows from simple propositional reasoning that $\widehat{A} \wedge \langle a \rangle \widehat{B}$ is consistent for some $B \in AT_0$. Consequently $A \stackrel{a}{\Longrightarrow} B$. Now let $\widehat{\rho}(i+1) = B$ and $\tau_{i+1} = \tau_i a$. The construction now proceeds from stage $i+1$.

Suppose now that $RQ_i \neq \emptyset$. Let $(\langle \pi \rangle \alpha, \tau)$ be the least member of RQ_i in the enumeration we have fixed for $\widehat{CL}(\alpha_0) \times \Sigma^*$. Let $j = |\tau|$ and $\tau\tau' = \tau_i$. Then using (A10) and (A11) it is easy to show that there exists π' such that $\pi \stackrel{\tau'}{\longrightarrow}_* \pi'$

and $\langle\pi'\rangle\alpha \in A_i$. Moreover $\alpha \notin A_i$ or $\varepsilon \notin ||\pi'||$. By part (ii) of Lemma 6.6.3, there exists $B \in AT_0$ and $\tau'' \in ||\pi'||$ such that $A_i \stackrel{\tau''}{\Longrightarrow}_* B$ and $\alpha \in B$. Let $\tau'' = b_1 b_2 \dots b_m$. Then we can find $B_0, B_1, \cdots, B_m \in AT_0$ such that $A_i = B_0$ and $B_m = B$ and $B_k \stackrel{b_k}{\Longrightarrow} B_{k+1}$ for $0 \leq k < m$. We now extend $\widehat{\rho}$ by:

$$\widehat{\rho}(i+k) = B_k \quad \text{for} \quad 1 \leq k \leq m.$$

Further we define $\tau_{i+k} = \tau_i b_1 b_2 \dots b_k$ for $1 \leq k \leq m$. The construction now proceeds from stage $i+m$.

Now consider the model $M_0 = (\sigma, V_0)$ where $\sigma \in \Sigma^\omega$ is the sequence satisfying that $\tau_i \preceq \sigma$ for every $i \in \omega$. Further, $V_0(\tau) = A_{|\tau|} \cap P$ for each $\tau \in \mathrm{prf}(\sigma)$. It is a routine exercise to establish that for all $\tau \in \mathrm{prf}(\sigma)$ and $\alpha \in \widehat{CL}(\alpha_0)$, $M_0, \tau \models \alpha$ iff $\alpha \in A_{|\tau|}$. Hence $M_0, \varepsilon \models \alpha_0$ as required.

The system $\mathcal{DLTL}$ is obtained by *replacing* (A10) and (A11) with the following axiom schemes:

(A12) $\alpha\, \mathcal{U}^\pi \beta \Rightarrow \langle\pi\rangle\beta$.

(A13) $\alpha\, \mathcal{U}^\pi \beta \Leftrightarrow \beta \vee \left(\alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{\pi'\in\delta_a(\pi)} \alpha\, \mathcal{U}^{\pi'}\beta\right), \quad (\varepsilon \in ||\pi||)$.

(A14) $\alpha\, \mathcal{U}^\pi \beta \Leftrightarrow \alpha \wedge \bigvee_{a\in\Sigma}\langle a\rangle \bigvee_{\pi'\in\delta_a(\pi)} \alpha\, \mathcal{U}^{\pi'}\beta, \quad (\varepsilon \notin ||\pi||)$.

It is an easy exercise to extend the completeness argument for $\mathcal{DLTL}^-$ to $\mathcal{DLTL}$. Thus we have:

Theorem 6.6.4

(i) *$\mathcal{DLTL}^-$ is a sound and complete axiomatization of the set of valid formulas of* $\mathrm{DLTL}^-(\Sigma)$.

(ii) *$\mathcal{DLTL}$ is a sound and complete axiomatization of the set of valid formulas of* $\mathrm{DLTL}(\Sigma)$.

6.7 Conclusion

We have presented here an enriched version of LTL called DLTL. The extension is obtained by indexing the until operator of LTL with regular programs. We have shown that in terms of the complexity of the decision procedure and expressiveness, DLTL compares very favourably with ETL. It is worth pointing out here that the decision procedure for DLTL is carried out directly in terms of Büchi automata whereas for ETL it is carried out in terms of the so called set-subword automata, which are then translated to Büchi automata [148]. Two additional results that are available for DLTL are: A characterization of the first-order fragment of MSO in terms of the sublogics $\mathrm{DLTL}^-_{\mathbb{SF}}$ and $\mathrm{DLTL}_{\mathbb{SF}}$; and a relatively clean axiomatization of DLTL^- and DLTL. All these results demonstrate that our means of bringing together propositional dynamic and temporal logics in a linear time setting is natural.

It turns out that our idea extends smoothly to richer domains. In particular, we can obtain similar results concerning the so called ω-regular product languages [136] in terms of the product version of DLTL [61]. Roughly

speaking, a ω-regular product language is a ω-regular language $L \subseteq \Sigma^\omega$ generated by a distributed alphabet $\{\Sigma_i\}_{i=1}^K$ with $\Sigma = \bigcup_{i=1}^K \Sigma_i$. The language L is a product language in the sense it is a finite union languages of the form $L_1 \otimes L_2 \otimes \cdots \otimes L_K$ with each L_i a regular subset of finite and infinite strings over Σ_i and $\otimes$ standing for the synchronized product operation. In other words $\sigma \in \Sigma^\omega$ is in $L_1 \otimes L_2 \otimes \cdots \otimes L_K$ iff $\sigma \restriction \Sigma_i$ (i.e. the sequence obtained by erasing all symbols from σ that are not in Σ_i) is in L_i for each i. The interesting distributed alphabets are of course those in which the component alphabets are *not* pairwise disjoint. The ω-regular product languages can be used to capture the linear time behaviour of a widely used model of distributed programs. These programs consist of a fixed set of finite state sequential programs that coordinate their behaviours by performing common actions together. Our logical characterization of ω-regular product languages is obtained by taking boolean combinations of formulas in $\bigcup_{i=1}^K \mathrm{DLTL}(\Sigma_i)$. More details can be found in [61]. It seems likely that one can find a nice generalization of this distributed version of DLTL to capture the full class of ω-regular trace languages.

Chapter 7

A Product Version of Dynamic Linear Time Temporal Logic

Contents

We present here a linear time temporal logic which simultaneously extends LTL, the propositional temporal logic of linear time, along two dimensions. Firstly, the until operator is strengthened by indexing it with the regular programs of propositional dynamic logic (PDL). Secondly, the core formulas of the logic are decorated with names of sequential agents drawn from fixed finite set. The resulting logic has a natural semantics in terms of the runs of a distributed program consisting of a finite set of sequential programs that communicate by performing common actions together. We show that our logic, denoted DLTL$^{\otimes}$, admits an exponential time decision procedure. We also show that DLTL$^{\otimes}$ is expressively equivalent to the so called regular product languages. Roughly speaking, this class of languages is obtained by starting with synchronized products of (ω-)regular languages and closing under boolean operations. We also sketch how the behaviours captured by our temporal logic fit into the framework of labelled partial orders known as Mazurkiewicz traces.

7.1 Introduction

We present a linear time temporal logic which extends LTL, the propositional temporal logic of linear time [83, 113] along two dimensions. Firstly, we strengthen the until modality by indexing it with the regular programs of PDL, propositional dynamic logic [39, 51]. Secondly, we consider networks of sequential agents that communicate by performing common actions together. We then reflect this in the logic by decorating the "core" formulas with the names of the agents. The resulting logic, denoted DLTL$^{\otimes}$, is a smooth generalization of the logic called product LTL [137] and the logic called dynamic linear time temporal logic [60].

DLTL$^{\otimes}$ admits a pleasant theory and our technical goal here is to sketch the main results of this theory. We believe that these results provide additional evidence — in a non-sequential setting — suggesting that our technique of combining dynamic and temporal logic as initiated in [60] is a fruitful one.

In the next section we introduce dynamic linear time temporal logic. We then state two main results concerning this logic. In Section 7.3 we define regular product languages. These are basically boolean combinations of synchronized products of (ω-)regular languages. We then present a characterization of this class of languages in terms of networks of Büchi automata that coordinate their activities by synchronizing on common letters.

In Section 7.4 we formulate the temporal logic DLTL$^{\otimes}$, the main object of study in this chapter. In Section 7.5 we establish an exponential time decision procedure for this logic by exploiting the Büchi automata networks presented in Section 7.3. In the subsequent section we show that DLTL$^{\otimes}$ captures exactly the class of regular product languages. It is worth noting that this is the first temporal logical characterization of this important class of distributed behaviours. In the final section we sketch how the behaviours described by our temporal logic (i.e. regular product languages) lie naturally within the domain of regular Mazurkiewicz trace languages.

7.2 Dynamic Linear Time Temporal Logic

One key feature of the syntax and semantics of our temporal logic is that *actions* will be treated as first class objects. The usual presentation of LTL [83, 113] is based on *states*; they are represented as subsets of a finite set of atomic propositions. We wish to bring in actions explicitly because it is awkward, if not difficult, to define synchronized products of sequential components in a purely state-based setting. This method of forming distributed systems is a common and useful one. Moreover, it is the main focus of attention in this chapter. Hence it will be handy to work with logics in which both states and actions can be treated on an equal footing. As a vehicle for introducing some terminology we shall first introduce an action-based version of LTL denoted LTL(Σ). We begin with some notations.

Through the rest of the chapter we fix a finite non-empty alphabet Σ. We

let a, b range over Σ and refer to members of Σ as actions. Σ^* is the set of finite words and Σ^ω is the set of infinite words generated by Σ with $\omega = \{0, 1, \ldots\}$. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and denote the null word by ε. We let σ, σ' range over Σ^∞ and τ, τ', τ'' range over Σ^*. Finally, $\preceq$ is the usual prefix ordering defined over Σ^* and $\mathrm{prf}(\sigma)$ is the set of finite prefixes of σ.

The set of formulas of LTL(Σ) is then given by the syntax:

$$\mathrm{LTL}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a \rangle \alpha \mid \alpha \,\mathcal{U}\, \beta.$$

For convenience we have avoided introducing atomic propositions and instead just deal with the constant tt and its negation $\neg tt \stackrel{\mathrm{def}}{=} ff$. Through the rest of this section α, β will range over LTL(Σ). The modality $\langle a \rangle$ is an action-indexed version of the next-state modality of LTL. A model is a ω-sequence $\sigma \in \Sigma^\omega$. For $\tau \in \mathrm{prf}(\sigma)$ we define $\sigma, \tau \models \alpha$ via:

- $\sigma, \tau \models tt$.
- $\sigma, \tau \models \neg\alpha$ iff $\sigma, \tau \not\models \alpha$.
- $\sigma, \tau \models \alpha \vee \beta$ iff $\sigma, \tau \models \alpha$ or $\sigma, \tau \models \beta$.
- $\sigma, \tau \models \langle a \rangle \alpha$ iff $\tau a \in \mathrm{prf}(\sigma)$ and $\sigma, \tau a \models \alpha$.
- $\sigma, \tau \models \alpha \,\mathcal{U}\, \beta$ iff there exists τ' such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Further, for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $\sigma, \tau\tau'' \models \alpha$.

It is well known that LTL(Σ) is equal in expressive power to the first order theory of sequences [41, 70]. Consequently this temporal logic is quite limited in terms of what it can not say. As an example, let $\Sigma = \{a, b\}$. Then the property "at every even position the action b is executed" is not definable in LTL(Σ). This observation, made in a state-based setting by Wolper, is the starting point for the extension of LTL called ETL [155, 156]. The route that we have taken to augment the expressive power of LTL(Σ) is similar in spirit but quite different in terms of the structuring mechanisms made available for constructing compound formulas. A more detailed assessment of the similarities and the differences between the two approaches is given in [60].

The extension that we have proposed is called DLTL(Σ). It basically consists of indexing the until operator with the programs of PDL (e.g. [39]). We start by defining the set of regular programs (expressions) generated by Σ. This set is denoted by $\mathrm{Prg}(\Sigma)$ and its syntax is given by:

$$\mathrm{Prg}(\Sigma) ::= a \mid \pi_0 + \pi_1 \mid \pi_0; \pi_1 \mid \pi^*.$$

With each program we associate a set of finite words via the map $||\cdot|| : \mathrm{Prg}(\Sigma) \longrightarrow 2^{\Sigma^*}$. This map is defined in the standard fashion:

- $||a|| = \{a\}$.
- $||\pi_0 + \pi_1|| = ||\pi_0|| \cup ||\pi_1||$.

- $||\pi_0;\pi_1|| = \{\tau_0\tau_1 \mid \tau_0 \in ||\pi_0|| \text{ and } \tau_1 \in ||\pi_1||\}$.
- $||\pi^*|| = \bigcup_{i\in\omega} ||\pi^i||$, where
  - $||\pi^0|| = \{\varepsilon\}$ and
  - $||\pi^{i+1}|| = \{\tau_0\tau_1 \mid \tau_0 \in ||\pi|| \text{ and } \tau_1 \in ||\pi^i||\}$ for every $i \in \omega$.

The set of formulas of DLTL(Σ) is given by the following syntax.

$$\text{DLTL}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha\, \mathcal{U}^\pi \beta, \quad \pi \in \text{Prg}(\Sigma)$$

A model is a ω-sequence $\sigma \in \Sigma^\omega$. For $\tau \in \text{prf}(\sigma)$ we define $\sigma, \tau \models \alpha$ just as we did for LTL(Σ) in the case of the first three clauses. As for the last one,

- $\sigma, \tau \models \alpha\, \mathcal{U}^\pi \beta$ iff there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \text{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Moreover, for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $\sigma, \tau\tau'' \models \alpha$.

Thus DLTL(Σ) adds to LTL(Σ) by strengthening the until operator. To satisfy $\alpha\ \mathcal{U}^\pi\beta$, one must satisfy $\alpha\mathcal{U}\beta$ along some finite stretch of behaviour which is required to be in the (linear time) behaviour of the program π. We now wish to state two of the main results of [60]. To do so, we first say that a formula $\alpha \in \text{DLTL}(\Sigma)$ is *satisfiable* if there exist $\sigma \in \Sigma^\omega$ and $\tau \in \text{prf}(\sigma)$ such that $\sigma, \tau \models \alpha$. Secondly, we associate with a formula α the ω-language L_α via:

$$L_\alpha \stackrel{\text{def}}{=} \{\sigma \in \Sigma^\omega \mid \sigma, \varepsilon \models \alpha\}.$$

A language $L \subseteq \Sigma^\omega$ is said to be DLTL(Σ)-definable if there exists some $\alpha \in$ DLTL(Σ) such that $L = L_\alpha$. Finally, we assume the notions of Büchi and Muller automata and ω-regular languages as formulated in [140].

Theorem 7.2.1

(i) *Given an $\alpha_0 \in$ DLTL(Σ) one can effectively construct a Büchi automaton $\mathcal{B}_{\alpha_0}$ of size $2^{O(|\alpha_0|)}$ such that $\mathcal{L}(\mathcal{B}_{\alpha_0}) \neq \emptyset$ iff α_0 is satisfiable. Thus the satisfiability problem for* DLTL(Σ) *is decidable in exponential time.*

(ii) *$L \subseteq \Sigma^\omega$ is ω-regular iff L is* DLTL(Σ)*-definable.*

It is also easy to formulate and solve a natural model checking problem for DLTL(Σ) where finite state programs are modelled as Büchi automata. But we shall not enter into details here.

To close out the section we shall point to two useful derived operators of DLTL(Σ):

- $\langle\pi\rangle\alpha \stackrel{\text{def}}{=} tt\, \mathcal{U}^\pi \alpha$.
- $[\pi]\alpha \stackrel{\text{def}}{=} \neg\langle\pi\rangle\neg\alpha$.

Suppose $\sigma \in \Sigma^\omega$ is a model and $\tau \in \mathrm{prf}(\sigma)$. It is easy to see that $\sigma, \tau \models \langle\pi\rangle\alpha$ iff there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \alpha$. It is also easy to see that $\sigma, \tau \models [\pi]\alpha$ iff for every $\tau' \in ||\pi||$, $\tau\tau' \in \mathrm{prf}(\sigma)$ implies $\sigma, \tau\tau' \models \alpha$. In this sense, the program modalities of PDL acquire a linear time semantics in the present setting. As shown in [60] the second part of Theorem 7.2.1 goes through even for the the sublogic of DLTL(Σ) obtained by banishing the until operator and instead using $\langle\pi\rangle\alpha$ and the boolean connectives. For an example of what can be said in this sublogic, assume $\Sigma = \{a, b\}$ and define π_{ev} to be the program $((a + b); (a + b))^*$. Then the formula $[\pi_{ev}]\langle b\rangle tt$ says "at every even position the action b is executed".

Next we note that $a \in \Sigma$ is a member of $\mathrm{Prg}(\Sigma)$ and the until operator of LTL(Σ) can be obtained via: $\alpha\mathcal{U}\beta \stackrel{\mathrm{def}}{=} \alpha\mathcal{U}^{\Sigma^*}\beta$. Due to second part of Theorem 7.2.1 we now have that both syntactically and semantically, LTL(Σ) is a *proper* fragment of DLTL(Σ).

To conclude the section, we note that the material presented here can be easily extended to include finite sequences over Σ as well. We shall assume from now on that this extension has indeed been carried out.

7.3 Regular Product Languages

A restricted but very useful model of finite state concurrent programs consists of a fixed number of finite state *sequential* programs that coordinate their activities by performing common actions together. A regular product language is an abstract specification of the linear time behaviour of such concurrent programs. The idea is to start with synchronized products of regular languages and close under boolean operations. Formally, we start with a *distributed alphabet* $\widetilde{\Sigma} = \{\Sigma_i\}_{i=1}^K$, a family of alphabets with each Σ_i a non-empty finite set of actions. One key point is that the component alphabets are not necessarily disjoint. Intuitively, $\mathrm{loc} = \{1, \ldots, K\}$ is the set of names of communicating sequential processes synchronizing on common actions, where Σ_i is the set of actions which require the participation of the agent i. Through the rest of the chapter we fix a distributed alphabet $\widetilde{\Sigma} = \{\Sigma_i\}_{i=1}^K$ and set $\Sigma = \bigcup_{i=1}^K \Sigma_i$. We carry over the terminology developed in the previous section for dealing with finite and infinite sequences over Σ. In addition, for $\sigma \in \Sigma^\infty$ and $i \in \mathrm{loc}$ we denote by $\sigma \restriction i$ the projection of σ down to Σ_i. In other words, it is the sequence obtained by erasing from σ all occurrences of symbols that are not in Σ_i. We let i, j, k range over $\mathrm{loc} = \{1, \ldots, K\}$ and define $\mathrm{loc}(a) = \{i \mid a \in \Sigma_i\}$. We note that $\mathrm{loc}(a)$ is the set of processes that participate in each occurrence of the action a.

Next we define the K-ary operator $\otimes : 2^{\Sigma_1^\infty} \times \cdots \times 2^{\Sigma_K^\infty} \to 2^{\Sigma^\infty}$ by

$$\otimes(L_1, \ldots, L_K) = \{\sigma \in \Sigma^\infty \mid \sigma \restriction i \in L_i \text{ for each } i \in \mathrm{loc}\}.$$

Usually we will write $\otimes(L_1, \ldots, L_K)$ as $L_1 \otimes L_2 \otimes \cdots \otimes L_K$. Finally, we will say that the language $L \subseteq \Sigma^\infty$ is regular iff $L \cap \Sigma^*$ is a regular subset of Σ^* and $L \cap \Sigma^\omega$ is an ω-regular subset of Σ^ω. Regular product languages can be built up as follows.

Definition 7.3.1 $L \subseteq \Sigma^\infty$ is a *direct* regular product language over $\widetilde{\Sigma}$ iff $L = L_1 \otimes \cdots \otimes L_K$ with L_i a regular subset of Σ_i^∞ for each $i \in \text{loc}$.

We let $\mathcal{R}_0^\otimes(\widetilde{\Sigma})$ be the class of direct regular product languages over $\widetilde{\Sigma}$.

Definition 7.3.2 The class of regular product languages over $\widetilde{\Sigma}$ is denoted $\mathcal{R}^\otimes(\widetilde{\Sigma})$ and is the least class of languages containing $\mathcal{R}_0^\otimes(\widetilde{\Sigma})$ and satisfying:

- If $L_1, L_2 \in \mathcal{R}^\otimes(\widetilde{\Sigma})$ then $L_1 \cup L_2 \in \mathcal{R}^\otimes(\widetilde{\Sigma})$.

In what follows we will often suppress the mention of the distributed alphabet $\widetilde{\Sigma}$. It is easy to prove that $\mathcal{R}^\otimes$ is closed under boolean operations. The proof of this result as well as other results mentioned in this section can be found in [136]. Just as ω-regular languages are captured by Büchi automata, we can capture regular product languages with the help of networks of Büchi automata. For convenience such automata will be termed product automata.

Definition 7.3.3 A *product automaton over* $\widetilde{\Sigma}$ is a structure

$$\mathcal{A} = (\{\mathcal{A}_i\}_{i \in \text{loc}}, Q_{in}),$$

where each $\mathcal{A}_i = (Q_i, \longrightarrow_i, F_i, F_i^\omega)$ satisfies:

- Q_i is a non-empty finite set of i-local states.
- $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$ is the transition relation of the ith component.
- $F_i \subseteq Q_i$ is a set of finitary accepting states of the ith component.
- $F_i^\omega \subseteq Q_i$ is a set of infinitary accepting states of the ith component.

Moreover, $Q_{in} \subseteq Q_1 \times \cdots \times Q_K$ is a set of global initial states.

Thus, a product automaton is a network of local automata with a global set of initial states. It is necessary to have global initial states in order to obtain the required expressive power. Each local automaton is equipped to cope with both finite and infinite behaviours using the finitary and infinitary accepting states. The infinitary accepting states are to be interpreted as defining a Büchi acceptance condition. This will become clear once we define the language accepted by a product automaton. We choose to deal with both finite and infinite component behaviours because the global behaviour can always induce finite local behaviours. In other words, even if ω-behaviour is the main focus of interest, a global infinite run will consist of one or more components running forever but with some other components, in general, quitting after making a finite number of moves. The notational complications involved in artificially making *all* components to run forever do not seem to be worth the trouble.

Let $\mathcal{A}$ be a product automaton over $\widetilde{\Sigma}$. Then $Q_G^\mathcal{A} = Q_1 \times \cdots \times Q_K$ is the set of global states of $\mathcal{A}$. The i-local transition relations induce a global transition relation $\longrightarrow_\mathcal{A} \subseteq Q_G^\mathcal{A} \times \Sigma \times Q_G^\mathcal{A}$ as follows:

$$q \xrightarrow{a}_\mathcal{A} q' \quad \text{iff} \quad \begin{array}{l} q[i] \xrightarrow{a}_i q'[i] \text{ for each } i \in \text{loc}(a) \text{ and} \\ q[i] = q'[i] \text{ for each } i \notin \text{loc}(a), \end{array}$$

where $q[i]$ denotes the ith component of $q = (q_1, \ldots, q_K)$. A run of $\mathcal{A}$ over $\sigma \in \Sigma^\infty$ is a mapping $\rho : \mathrm{prf}(\sigma) \to Q_G^{\mathcal{A}}$ satisfying that $\rho(\varepsilon) \in Q_{in}$ and $\rho(\tau) \stackrel{a}{\longrightarrow}_{\mathcal{A}} \rho(\tau a)$ for all $\tau a \in \mathrm{prf}(\sigma)$. The run is *accepting* iff the following conditions are satisfied for each i:

- If $\sigma \restriction i$ is finite then $\rho(\tau)[i] \in F_i$ for some $\tau \in \mathrm{prf}(\sigma)$ with $\tau \restriction i = \sigma \restriction i$.
- If $\sigma \restriction i$ is infinite then $\rho(\tau)[i] \in F_i^\omega$ for infinitely many $\tau \in \mathrm{prf}(\sigma)$.

We next define

$$\mathcal{L}(\mathcal{A}) = \{\sigma \in \Sigma^\infty \mid \text{there exists an accepting run of } \mathcal{A} \text{ over } \sigma\}.$$

The next result established relates regular product languages to product automata.

Theorem 7.3.4 *$L \in \mathcal{R}^\otimes(\widetilde{\Sigma})$ iff $L = \mathcal{L}(\mathcal{A})$ for some product automaton $\mathcal{A}$ over $\widetilde{\Sigma}$.*

We will later give solutions to the satisfiability problem for a product version of DLTL with the help of product automata. The following results will be useful in this context. In stating these results we take the *size* of the product automaton $\mathcal{A}$ to be $|Q_G^{\mathcal{A}}|$.

Lemma 7.3.5

- *Let $\mathcal{A}$ be a product automaton. The question $\mathcal{L}(\mathcal{A}) \stackrel{?}{\neq} \emptyset$ can be effectively decided in time $O(n^2)$, where n is the size of $\mathcal{A}$.*
- *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be product automata of sizes n_1 and n_2, respectively. Then a product automaton $\mathcal{A}$ of size $O(n_1 n_2)$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ can be effectively constructed.*

7.4 A Product Version of DLTL

We now wish to design a product version of DLTL denoted $\mathrm{DLTL}^\otimes(\widetilde{\Sigma})$. It will turn out to have the expressive power of regular product languages over $\widetilde{\Sigma}$. The set of formulas and their *locations* are given by:

- tt is a formula and $\mathrm{loc}(tt) = \emptyset$.
- Suppose α and β are formulas. Then so are $\neg\alpha$ and $\alpha \vee \beta$. Furthermore, $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$ and $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.
- Suppose α and β are formulas such that $\mathrm{loc}(\alpha), \mathrm{loc}(\beta) \subseteq \{i\}$ and suppose $\pi \in \mathrm{Prg}(\Sigma_i)$. Then $\alpha\, \mathcal{U}_i^\pi\, \beta$ is a formula. Moreover, $\mathrm{loc}(\alpha\, \mathcal{U}_i^\pi\, \beta) = \{i\}$.

We note that each formula in $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ is a boolean combination of formulas taken from the set $\bigcup_{i\in \mathrm{loc}} \mathrm{DLTL}_i^{\otimes}(\widetilde{\Sigma})$ where, for each i,

$$\mathrm{DLTL}_i^{\otimes}(\widetilde{\Sigma}) = \{\alpha \mid \alpha \in \mathrm{DLTL}^{\otimes}(\widetilde{\Sigma}) \text{ and } \mathrm{loc}(\alpha) \subseteq \{i\}\ \}.$$

Once again, we have chosen to avoid dealing with atomic propositions for the sake of convenience. They can be introduced in a local fashion as done in [136]. The decidability result to be presented will go through with minor notational overheads.

As before, we will often suppress the mention of $\widetilde{\Sigma}$. We will also often write τ_i, τ_i' and τ_i'' instead of $\tau \restriction i$, $\tau' \restriction i$ and $\tau'' \restriction i$, respectively with $\tau, \tau', \tau'' \in \Sigma^*$.

A model is a sequence $\sigma \in \Sigma^{\infty}$ and the semantics of this logic is given as before, with $\tau \in \mathrm{prf}(\sigma)$.

- $\sigma, \tau \models tt$.
- $\sigma, \tau \models \neg\alpha$ iff $\sigma, \tau \not\models \alpha$.
- $\sigma, \tau \models \alpha \vee \beta$ iff $\sigma, \tau \models \alpha$ or $\sigma, \tau \models \beta$.
- $\sigma, \tau \models \alpha\, \mathcal{U}_i^{\pi}\, \beta$ iff there exists τ' such that $\tau_i' \in ||\pi||$ (recall that $\tau_i' = \tau' \restriction i$) and $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Further, for every $\tau'' \in \mathrm{prf}(\tau')$, if $\varepsilon \preceq \tau_i'' \prec \tau_i'$ then $\sigma, \tau\tau'' \models \alpha$.

We will say that a formula $\alpha \in \mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ is *satisfiable* if there exist $\sigma \in \Sigma^{\infty}$ and $\tau \in \mathrm{prf}(\sigma)$ such that $\sigma, \tau \models \alpha$. The language defined by α is given by

$$L_\alpha \stackrel{\mathrm{def}}{=} \{\sigma \in \Sigma^{\infty} \mid \sigma, \varepsilon \models \alpha\}.$$

We say that $L \subseteq \Sigma^{\infty}$ is $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$-definable if there exists some $\alpha \in \mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ with $L_\alpha = L$.

7.5 A Decision Procedure for DLTL$^{\otimes}$

We will show the satisfiability problem for $\mathrm{DLTL}(\Sigma)$ is solvable in deterministic exponential time. This will be achieved by effectively constructing a product automaton $\mathcal{A}_\alpha$ for each $\alpha \in \mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ such that the language accepted by $\mathcal{A}_\alpha$ is non-empty iff α is satisfiable. Our construction is a common generalization of the one for product LTL in [137] and the one for $\mathrm{DLTL}(\Sigma)$ in [60]. The solution to the satisfiability problem will at once lead to a solution to the model checking problem for programs modelled as synchronizing sequential agents.

Through the rest of the section we fix a formula $\alpha_0 \in \mathrm{DLTL}^{\otimes}$. In order to construct $\mathcal{A}_{\alpha_0}$ we first define the (Fischer-Ladner) closure of α_0. As a first step let $cl(\alpha_0)$ be the least set of formulas satisfying:

- $\alpha_0 \in cl(\alpha_0)$.
- $\neg\alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.

- $\alpha \vee \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$.
- $\alpha\, \mathcal{U}_i^{\pi} \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$.

We will now take the *closure* of α_0 to be $CL(\alpha_0) = cl(\alpha_0) \cup \{\neg\alpha \mid \alpha \in cl(\alpha_0)\}$. From now on we shall identify $\neg\neg\alpha$ with α. Set $CL_i(\alpha_0) = CL(\alpha_0) \cap \mathrm{DLTL}_i^{\otimes}(\widetilde{\Sigma})$ for each i. We will often write CL instead of $CL(\alpha_0)$ and CL_i instead of $CL_i(\alpha_0)$. All formulas considered from now on will be assumed to belong to CL_i unless otherwise stated.

An *i-type atom* is a subset $A \subseteq CL_i$ which satisfies:

- $tt \in A$.
- $\alpha \in A$ iff $\neg\alpha \notin A$.
- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- $\beta \in A$ and $\varepsilon \in ||\pi||$ implies $\alpha\, \mathcal{U}_i^{\pi} \beta \in A$.

The set of i-type atoms is denoted AT_i. Following this, we define the predicate $\mathrm{Member}(\alpha, (A_1, \ldots, A_K))$ for each $\alpha \in CL(\alpha_0)$ and $(A_1, \ldots, A_K) \in AT_1 \times \cdots \times AT_K$. For convenience this predicate will be denoted as $\alpha \in (A_1, \ldots, A_K)$ and is given inductively by:

- Let $\alpha \in CL_i$. Then $\alpha \in (A_1, \ldots, A_K)$ iff $\alpha \in A_i$.
- Let $\alpha = \neg\beta$. Then $\alpha \in (A_1, \ldots, A_K)$ iff $\beta \notin (A_1, \ldots, A_K)$.
- Let $\alpha = \beta \vee \gamma$. Then $\alpha \in (A_1, \ldots, A_K)$ iff $\beta \in (A_1, \ldots, A_K)$ or $\gamma \in (A_1, \ldots, A_K)$.

The set of *i-type until requirements* is the subset of CL_i given by

$$Req_i = \{\alpha\, \mathcal{U}_i^{\pi} \beta \mid \alpha\, \mathcal{U}_i^{\pi} \beta \in CL_i\}.$$

We shall let ξ, ξ' range over Req_i. For each $\xi = \alpha\, \mathcal{U}_i^{\pi} \beta \in Req_i$ we fix a finite state automaton $\mathcal{A}_\xi$ such that $\mathcal{L}(\mathcal{A}_\xi) = ||\pi||$ where $\mathcal{L}(\mathcal{A}_\xi)$ is the language of finite words accepted by $\mathcal{A}_\xi$. We shall assume each such $\mathcal{A}_\xi$ is of the form $\mathcal{A}_\xi = (Q_\xi, \longrightarrow_\xi, I_\xi, F_\xi)$ where Q_ξ is the set of states, $\longrightarrow_\xi \subseteq Q_\xi \times \Sigma \times Q_\xi$ is the transition relation, $I_\xi \subseteq Q_\xi$ is the set of initial states and $F_\xi \subseteq Q_\xi$ is the set of final states. Without loss of generality, we shall assume that $\xi \neq \xi'$ implies $Q_\xi \cap Q_{\xi'} = \emptyset$ for every $\xi, \xi' \in Req_i$. We set $Q_i = \bigcup_{\xi \in Req_i} Q_\xi$ and $\widehat{Q}_i = Q_i \times \{0, 1\}$.

The product automaton $\mathcal{A}_{\alpha_0}$ associated with α_0 is now defined to be $\mathcal{A}_{\alpha_0} = (\{\mathcal{A}_i\}_{i \in \mathrm{loc}}, Q_{in})$ where for each i, $\mathcal{A}_i = (S_i, \Longrightarrow_i, F_i, F_i^{\omega})$ is specified as

(1) $S_i \subseteq AT_i \times 2^{Q_i} \times 2^{\widehat{Q}_i} \times \{\mathsf{stop}, \mathsf{go}\} \times \{0, 1\} \times \{\downarrow, \checkmark\}$ such that

$$(A, X, \widehat{X}, s, x, f) \in S_i$$

iff the following conditions are satisfied for each $\xi = \alpha\, \mathcal{U}_i^{\pi} \beta$:

(i) If $\beta \in A$ then $F_\xi \subseteq X$. (Recall that $\mathcal{A}_\xi = (Q_\xi, \longrightarrow_\xi, I_\xi, F_\xi)$).

(ii) If $\alpha \in A$ and $q \in X$ for some $q \in I_\xi$ then $\alpha\, \mathcal{U}_i^\pi\, \beta \in A$.

(iii) If $\alpha\, \mathcal{U}_i^\pi\, \beta \in A$ then either $\beta \in A$ and $\varepsilon \in ||\pi||$ or $(q, 1-x) \in \widehat{X}$ for some $q \in I_\xi$. (Note that we are considering the candidate $(A, X, \widehat{X}, s, x, f)$ for membership in S_i).

(iv) If $(q, z) \in \widehat{X}$ with $q \notin F_\xi$ or $\beta \notin A$ then $\alpha \in A$.

(2) The transition relation $\Longrightarrow_i \subseteq S_i \times \Sigma_i \times S_i$ is defined as follows:

$$(A, X, \widehat{X}, s, x, f) \stackrel{a}{\Longrightarrow}_i (B, Y, \widehat{Y}, t, y, g)$$

iff the following conditions are satisfied for each $\xi = \alpha\, \mathcal{U}_i^\pi\, \beta \in Req_i$:

(i) $s = \mathsf{go}$.

(ii) Suppose $q' \in Q_\xi \cap Y$ and $q \stackrel{a}{\longrightarrow}_\xi q'$ and $\alpha \in A$. Then $q \in X$.

(iii) Suppose $(q, z) \in \widehat{X}$ with $q \in Q_\xi$. Suppose further that $q \notin F_\xi$ or $\beta \notin A$. Then $(q', z) \in \widehat{Y}$ for some q' with $q \stackrel{a}{\longrightarrow}_\xi q'$.

(iv) If $f = \checkmark$ then $(y, g) = (1 - x, \downarrow)$. If $f = \downarrow$ then,

$$(y, g) = \begin{cases} (x, \downarrow), & \text{if there exists } (q, x) \in \widehat{X} \text{ such that} \\ & q \notin F_\xi \text{ or } \beta \notin A \\ (x, \checkmark), & \text{otherwise.} \end{cases}$$

(3) $F_i = \{(A, X, \widehat{X}, s, x, f) \mid s = \mathsf{stop} \text{ and } \widehat{X} = \emptyset\}$.

(4) $F_i^\omega = \{(A, X, \widehat{X}, s, x, f) \mid f = \checkmark\}$.

Finally, $Q_{in} \subseteq Q_1 \times \cdots \times Q_K$ is specified as

$$((A_1, X_1, \widehat{X}_1, s_1, x_1, f_1), \ldots, (A_K, X_K, \widehat{X}_K, s_K, x_K, f_K)) \in Q_{in}$$

iff $\alpha_0 \in (A_1, \ldots, A_K)$ and $(x_i, f_i) = (0, \checkmark)$ for every $i \in \mathrm{loc}$.

The main result of this section can now be formulated.

Theorem 7.5.1 *$\mathcal{L}(\mathcal{A}_{\alpha_0}) = L_{\alpha_0}$ where $\mathcal{A}_{\alpha_0}$ is as defined above. Hence α_0 is satisfiable iff $\mathcal{L}(\mathcal{A}_{\alpha_0}) \neq \emptyset$. Moreover, the size of $\mathcal{A}_{\alpha_0}$ is $2^{O(|\alpha_0|)}$ and consequently the satisfiability problem for $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ is decidable in exponential time.*

Proof: Let $\sigma \in \mathcal{L}(\mathcal{A}_{\alpha_0})$ by the accepting run $\rho : \mathrm{prf}(\sigma) \to Q_G^{\mathcal{A}}$. For each $\tau \in \mathrm{prf}(\sigma)$ let $\rho(\tau)[i] = (A_{\tau_i}, X_{\tau_i}, \widehat{X}_{\tau_i}, s_{\tau_i}, x_{\tau_i}, f_{\tau_i})$. Then a detailed examination of the above construction reveals that for all $\tau \in \mathrm{prf}(\sigma)$ and $\delta \in CL_i$,

$$\sigma, \tau \models \delta \text{ iff } \delta \in A_{\tau_i}.$$

By definition of Q_{in} we are assured that $\alpha_0 \in (\rho(\varepsilon)[1], \ldots, \rho(\varepsilon)[K])$. Hence a simple induction on the structure of α_0 will show that $\sigma, \varepsilon \models \alpha_0$.

Conversely, if α_0 is satisfiable we may assume that $\sigma, \varepsilon \models \alpha_0$ for some σ. We will construct an accepting run $\rho : \mathrm{prf}(\sigma) \to Q_G^{\mathcal{A}}$. For each $\tau \in \mathrm{prf}(\sigma)$ and each i, we set $\rho(\tau)[i] = (A_{\tau_i}, X_{\tau_i}, \widehat{X}_{\tau_i}, s_{\tau_i}, x_{\tau_i}, f_{\tau_i})$ and define the various components of this tuple as follows. First we define A_{τ_i} by $A_{\tau_i} = \{\alpha \in CL_i \mid \sigma, \tau \models \alpha\}$. Next s_{τ_i} is defined as $s_{\tau_i} = \mathsf{stop}$ iff $\sigma \restriction i = \tau_i$ (recall the convention $\tau \restriction i = \tau_i$). In defining the other components it will be convenient to adopt the following terminology.

Let $\xi = \alpha\, \mathcal{U}_i^{\pi} \beta$ and $q \in Q_\xi$ and $\tau_i \in \Sigma_i^*$. Then an *accepting run of* $\mathcal{A}_\xi$ *over* τ_i *starting from* q is a map $R : \mathrm{prf}(\tau_i) \longrightarrow Q_\xi$ such that $R(\varepsilon) = q$, $R(\tau_i) \in F_\xi$ and $R(\tau_i'') \xrightarrow{a}_\xi R(\tau_i'' a)$ for every $\tau_i'' a \in \mathrm{prf}(\tau_i)$. In case $q \in I_\xi$ we shall just say that R is an *accepting run of* $\mathcal{A}_\xi$ *over* τ_i.

Let $\xi = \alpha\, \mathcal{U}_i^{\pi} \beta$ and $q \in Q_\xi$. Then $q \in X_{\tau_i}$ iff there exist τ' and R' such that $\tau\tau' \in \mathrm{prf}(\sigma)$, $\sigma, \tau\tau' \models \beta$, and for every $\tau'' \in \mathrm{prf}(\tau')$, if $\varepsilon \preceq \tau_i'' \prec \tau_i'$ then $\sigma, \tau\tau'' \models \alpha$. Furthermore, R' should be an accepting run of $\mathcal{A}_\xi$ over τ_i' starting from q.

To specify the remaining three components we shall make use of a chronicle of obligations.

We'll say that (τ, ξ) is an *obligation* if $\tau \in \mathrm{prf}(\sigma)$ and $\xi = \alpha\, \mathcal{U}_i^{\pi} \beta \in Req_i$ such that $\sigma, \tau \models \alpha\, \mathcal{U}_i^{\pi} \beta$ but $\sigma, \tau \not\models \beta$ or $\varepsilon \notin ||\pi||$. Let (τ, ξ) be an obligation. We shall say that the pair (τ', R') is a *witness* for (τ, ξ) iff $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$ and for every $\tau'' \in \mathrm{prf}(\tau')$, if $\varepsilon \preceq \tau_i'' \prec \tau_i'$ then $\sigma, \tau\tau'' \models \alpha$. Furthermore, $\tau_i' \in ||\pi||$ and R' is an accepting run of $\mathcal{A}_\xi$ over τ_i'. A *chronicle set* CH is a set of quadruples satisfying that if $(\tau, \xi, \tau', R') \in CH$ then (τ, ξ) is an obligation and (τ', R') is witness for (τ, ξ). Moreover, for every obligation (τ, ξ) there is a unique element of the form (τ, ξ, τ', R') in CH. We fix such a set CH which clearly exists.

Now x_{τ_i} and f_{τ_i} are defined by mutual induction as follows. For the base case, $(x_\varepsilon, f_\varepsilon) = (0, \checkmark)$. For the induction step, let $\tau = \tau' a$. Suppose $a \notin \Sigma_i$. Then $(x_{\tau_i}, f_{\tau_i}) = (x_{\tau_i'}, f_{\tau_i'})$. So assume that $a \in \Sigma_i$. If $f_{\tau_i'} = \checkmark$ then $(x_{\tau_i}, f_{\tau_i}) = (1 - x_{\tau_i'}, \downarrow)$. Suppose $f_{\tau_i'} = \downarrow$. Then $(x_{\tau_i}, f_{\tau_i}) = (x_{\tau_i'}, \downarrow)$ if there exists $(\tau'', \xi_1, \tau''', R_1') \in CH$ such that $\tau'' \preceq \tau' \prec \tau''\tau'''$ and $x_{\tau_i''} = 1 - x_{\tau_i'}$. Otherwise, $f_{\tau_i} = \checkmark$ and $x_{\tau_i} = x_{\tau_i'}$.

The only remaining component to be dealt with is $\widehat{X}_{\tau_i}$. This is now defined via: $(q, z) \in \widehat{X}_{\tau_i}$ iff there exists $(\tau', \xi, \tau'', R_1') \in CH$ such that for some $\tau''' \in \mathrm{prf}(\tau'')$, $\tau_i' \preceq \tau_i = \tau_i'\tau_i'''$ and furthermore $R_1'(\tau_i''') = q$ and $x_{\tau_i'} = 1 - z$. Using these definitions it is not difficult to show that ρ is an accepting run.

Finally, by Lemma 7.3.5 it suffices to show that our construction yields a product automaton of exponential size. Clearly, $CL(\alpha_0)$ is linear in α_0, and surely then $|AT_1| + \cdots + |AT_K| = 2^{O(|\alpha_0|)}$. Moreover, it is well-known that each $\pi \in \mathrm{Prg}(\Sigma_i)$ in polynomial time can be converted to a finite (non-deterministic) automaton with a linear state space (see [64] for a recent account of such conversions). Then both $Q_1 + \cdots + Q_K$ and $\widehat{Q}_1 + \cdots + \widehat{Q}_K$ are of size $O(|\alpha_0|)$. Consequently, $|Q_G^{\mathcal{A}}| = 2^{O(|\alpha_0|)}$ as required. □

The procedure outlined above also lends itself to a solution to the *model*

checking problem, which is defined as for DLTL except that a finite-state program is now simply a product automaton $\mathcal{P}$. Once again, we do not wish to enter into details.

7.6 An Expressiveness Result

We now wish to show that our logic is expressively complete with respect to the regular product languages. In fact we will identify a natural sublogic — to be denoted $\mathrm{DLTL}^{\otimes}_{-}$ — which also enjoys this property.

The syntax of the formulas of $\mathrm{DLTL}^{\otimes}_{-}(\widetilde{\Sigma})$ remains as for $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$, but the until modality is to be restricted to the derived operator $\langle \_ \rangle_i$. Formally, the set of formulas and locations of this sublogic is obtained via:

- tt is a formula and $\mathrm{loc}(tt) = \emptyset$.
- Suppose α and β are formulas so are $\neg\alpha$ and $\alpha \vee \beta$. Moreover $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$ and $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.
- Suppose α is formula such that $\mathrm{loc}(\alpha) \subseteq \{i\}$ and $\pi \in \mathrm{Prg}(\Sigma_i)$. Then $\langle \pi \rangle_i \alpha$ is formula. Moreover, $\mathrm{loc}(\langle \pi \rangle_i \alpha) = \{i\}$.

Proposition 7.6.1 *If $L \in \mathcal{R}^{\otimes}(\widetilde{\Sigma})$ then L is $\mathrm{DLTL}^{\otimes}_{-}(\widetilde{\Sigma})$-definable.*

Proof: It suffices to show that the claim holds for $L \in \mathcal{R}^{\otimes}_0(\widetilde{\Sigma})$ because each member of $\mathcal{R}^{\otimes}(\widetilde{\Sigma})$ is a finite union of languages in $\mathcal{R}^{\otimes}_0(\widetilde{\Sigma})$.

Let $L = L_1 \otimes \cdots \otimes L_K \in \mathcal{R}^{\otimes}_0(\widetilde{\Sigma})$. Then each $L_i \cap \Sigma_i^*$ is regular. Clearly $L_i \cap \Sigma_i^* = ||\pi_i||$ for some $\pi_i \in \mathrm{Prg}(\Sigma_i)$. Now define $\alpha^i_* = \langle \pi_i \rangle_i [\pi'_i]_i ff$ where $\pi'_i = (a_1 + \ldots + a_n)$ with $\Sigma_i = \{a_1, \ldots, a_n\}$.

Next, $L_i \cap \Sigma_i^\omega$ is ω-regular. Hence it is accepted, due to McNaughton's theorem [87], by a *deterministic* Muller automaton. Choose such an automaton $\mathcal{M} = (Q, q_{in}, \longrightarrow, \mathcal{F})$, which we, without loss of generality, assume to be complete. (See [60] for the formal details). For $q, q' \in Q$ we set $L_{q,q'} = \{\tau \mid q \xrightarrow{\tau} q'\}$, which is obviously a regular subset of Σ_i^*. So we can fix $\pi_{q,q'} \in \mathrm{Prg}(\Sigma_i)$ such that $L_{q,q'} = ||\pi_{q,q'}||$. Moreover, by the determinacy of $\mathcal{M}$ it follows that $L_{q,q'} \cap L_{q,q''} \neq \emptyset$ implies $q' = q''$. We now define

$$\alpha^i_\omega = \bigvee_{F \in \mathcal{F}} \bigvee_{q \in F} \langle \pi_{q_{in},q} \rangle_i \left(\bigwedge_{q' \notin F} [\pi_{q,q'}]_i ff \wedge \bigwedge_{j=0}^{n-1} [\pi_{q,q_j}]_i \langle \pi_{q_j,q_{j\oplus 1}} \rangle_i tt \right)$$

with the assumption $\{q_0, q_1, \ldots, q_{n-1}\}$ is an enumeration of the $F \in \mathcal{F}$ under consideration and "$\oplus$" denotes addition modulo n. It is easy to show that $\sigma \restriction i \in L_i \cap \Sigma_i^\omega$ iff $\sigma, \varepsilon \models \alpha^i_\omega$.

The required formula α is given by $\alpha = \bigwedge_{i \in \mathrm{loc}} \alpha^i$ where $\alpha^i = \alpha^i_* \vee \alpha^i_\omega$ for each i. It is a routine exercise to establish $L_\alpha = L_1 \otimes \cdots \otimes L_K$.

□

On the other hand, Theorem 7.3.4 together with Theorem 7.5.1 states that L_{α_0} is a product language over $\widetilde{\Sigma}$ for any $\alpha_0 \in \mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$. Since $\mathrm{DLTL}^{\otimes}_{-}$ is a sublogic of $\mathrm{DLTL}^{\otimes}$ we have the following expressiveness result.

Corollary 7.6.2 *Let $L \subseteq \Sigma^{\infty}$. Then the following statements are equivalent:*

(i) $L \in \mathcal{R}^{\otimes}(\widetilde{\Sigma})$.

(ii) L *is* $\mathrm{DLTL}^{\otimes}_{-}(\widetilde{\Sigma})$*-definable.*

(iii) L *is* $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$*-definable.*

7.7 Discussion

We shall conclude this section by placing regular product languages in the broader context of regular Mazurkiewicz trace languages. For an introduction to (Mazurkiewicz) traces related to the concerns of the present chapter, we refer the reader to [97]. We shall assume the bare minimum of the background material on traces.

We begin by noting that the distributed alphabet $\widetilde{\Sigma} = \{\Sigma_i\}_{i=1}^{K}$ induces the trace alphabet $(\Sigma, I_{\widetilde{\Sigma}})$ where the irreflexive and symmetric independence relation $I_{\widetilde{\Sigma}} \subseteq \Sigma \times \Sigma$ is given by:

$$a \; I_{\widetilde{\Sigma}} \; b \;\; \text{iff} \;\; \mathrm{loc}(a) \cap \mathrm{loc}(b) = \emptyset.$$

Recall that $\mathrm{loc}(x) = \{i \mid x \in \Sigma_i\}$ for $x \in \Sigma$. We shall write I instead of $I_{\widetilde{\Sigma}}$ from now on. This independence relation in turn induces the equivalence relation $\approx_I \subseteq \Sigma^{\infty} \times \Sigma^{\infty}$ (from now on written as $\approx$) given by:

$$\sigma \approx \sigma' \;\; \text{iff} \;\; \sigma \restriction i = \sigma' \restriction i \;\; \text{for every} \;\; i \in \mathrm{loc}.$$

The $\approx$-equivalence classes of Σ^{∞} constitute the set of finite and infinite traces generated by the trace alphabet (Σ, I). Traces can be — upto isomorphisms — uniquely represented as certain Σ-labelled posets where the labelling functions respect I in a natural manner. A trace language is just a subset of $\Sigma^{\infty}/\approx$.

A language $L \subseteq \Sigma^{\infty}$ is trace consistent in case $\sigma \in L$ and $\sigma \approx \sigma'$ implies $\sigma' \in L$, for every σ, σ'. The point is, a trace consistent language L canonically represents the trace language $\{[\sigma]_{\approx} \mid \sigma \in L\}$. We extend this idea to logical formulas by saying that $\alpha \in \mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ is trace consistent iff L_{α} is trace consistent. It is easy to show that *every* formula of $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ is trace consistent. An important feature of properties defined by trace consistent formulas is that they can often be verified efficiently using partial order based reduction techniques [48, 108, 144]. Consequently, $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ provides a flexible and powerful means for specifying trace consistent properties of distributed programs. As it turns out, every formula of $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ defines — via the canonical representation — a regular trace language contained in $\Sigma^{\infty}/\approx$. Hence by Corollary 7.6.2, every regular product language corresponds to a regular trace language.

The converse however is not true. To bring this out, consider the distributed alphabet $\widetilde{\Sigma} = \{\{a, a', c\}, \{b, b', c\}\}$ and the language $L = \{cab, cba, ca'b', cb'a'\}^\omega$. Then it is easy to check that L is trace consistent and ω-regular and that it is *not* a regular product language. In a forthcoming paper we shall deal with the problem of extending $\mathrm{DLTL}^{\otimes}(\widetilde{\Sigma})$ so as to capture *all* of the regular trace languages.

Chapter 8

Distributed Versions of Linear Time Temporal Logic: A Trace Perspective

Contents

8.1 Introduction

Linear time Temporal Logic (LTL) as proposed by Pnueli [113] has become a well established tool for specifying the dynamic behaviour of distributed systems. A basic feature of LTL is that its formulas are interpreted over sequences. Typically, such a sequence will model a computation of a system; a sequence of states visited by the system or a sequence of actions executed by the system during the course of the computation. A system is said to satisfy a specification expressed as an LTL formula in case every computation of the system is a

model of the formula. A rich theory of LTL is now available using which one can effectively verify whether a finite state system meets its specification [147]. Indeed, the verfication task can be automated (for instance using the software packages SPIN [62] and FormalCheck [16]) to handle large systems of practical interest.

In many applications the computations of a distributed system will constitute interleavings of the occurrences of causally independent actions. Consequently, the computations can be naturally grouped together into equivalence classes where two computations are equated in case they are two different interleavings of the same partially ordered stretch of behaviour. It turns out that many of the properties expressed as LTL-formulas happen to have the so called "all-or-none" property. Either all members of an equivalence class of computations will have the desired property or none will do ("leads to deadlock" is one such property). For verifying such properties one has to check the property for just one member of each equivalence class. This is the insight underlying many of the partial-order based verification methods [48, 108, 144]. As may be guessed, the importance of these methods lies in the fact that via these methods the computational resources required for the verification task can often be dramatically reduced.

It is often the case that the equivalence classes of computations generated by a distributed system constitute objects called Mazurkiewicz traces. They can be canonically represented as restricted labelled partial orders. This opens up an alternative way of exploiting the non-sequential nature of the computations of a distributed systems and the attendant partial-order based methods. It consists of developing linear time temporal logics that can be directly interpreted over Mazurkiewicz traces. In these logics, every specification is guaranteed to have the "all-or-none" property and hence can take advantage of the partial-order based reduction methods during the verification process. The study of these logics also exposes the richness of the partial-order settings from a logical standpoint and the complications that can arise as a consequence.

Our aim here is to present an overview of linear time temporal logics whose models can be viewed as Mazurkiewicz traces. The presentation is, in principle, self-contained though previous exposure to temporal logics [35] and automata over infinite objects [140] will be very helpful. We have provided net-theoretic examples whenever possible in order to emphasize the broad scope of applicability of the material.

In the next section we introduce linear time temporal logic and sketch the automata-theoretic solutions to the satisfiability problem (does a formula have a model?) and the model checking problem (do all computations of a system constitute models of a given specification formula?). In Section 3 we introduce Mazurkiewicz traces viewed as equivalence classes of sequences. This leads to the precise formualtion of the notion "all-or-none" LTL properties.

Next we introduce a well-understood class of trace languages called product languages. The automata that recognize these languages are called product automata and they incorporate a simple and yet useful method of forming distributed systems. The system consists of a network of sequential agents, each

with its own alphabet of actions. In the interesting instances the alphabets are not pair-wise disjoint. One then imposes a synchronization regime under which the agents are forced to carry out common actions together. After presenting a theory of product languages and automata, we formulate in Section 5 a simple version of a trace-based version of LTL called product LTL. The formulas of this logic have a natural semantics in terms of the computations generated by a network of sequential agents as introduced in the previous section. Using the theory of product automata we then provide solutions to the satisfiability and model checking problems for product LTL.

In Section 6 we introduce the representation of Mazurkiewicz traces as restricted labelled partial orders. We then provide a rapid introduction to the theory of trace languages and automata that we call asynchronous automata for recognizing trace languages. In the subsequent section we introduce the logic TrPTL which is a trace-based logic with much richer possibilities than product LTL. We then provide solutions to the satisfiability and model checking problems for TrPTL using asynchronous automata. This is followed by a brief survey of other trace-based linear time temporal logics available in the literature. Section 8 is devoted to considering various expressiveness issues associated with our temporal logics. We conclude in the final section with remarks about branching time temporal logics based on traces.

8.2 Linear Time Temporal Logic

In our formulation of linear time temporal logics it will be convenient to treat *actions* as first class objects both at the syntactic and semantic levels. As a first step we shall consider a version of LTL (linear time temporal logic) in which the next-state modality is indexed by actions.

Through the rest of the chapter we fix a finite non-empty alphabet of actions Σ. We let a, b range over Σ and refer to members of Σ as actions. Σ^* is the set of finite words and Σ^ω is the set of infinite words generated by Σ with $\omega = \{0, 1, 2, \ldots\}$. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and denote the null word by ε. We let σ, σ' range over Σ^ω and τ, τ', τ'' range over Σ^*. Finally $\preceq$ is the usual prefix ordering defined over Σ^* and for $u \in \Sigma^\infty$, we let $\mathrm{prf}(u)$ be the set of finite prefixes of u.

Next we fix a finite non-empty set of atomic propositions $P = \{p_1, p_2, \ldots\}$ and let p, q range over P. The set of formulas of $\mathrm{LTL}(\Sigma)$ is then given by the syntax:

$$\mathrm{LTL}(\Sigma) ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a \rangle \alpha \mid \alpha \; U \; \beta.$$

Through the rest of this section α, β will range over $\mathrm{LTL}(\Sigma)$.

A model of $\mathrm{LTL}(\Sigma)$ is a pair $M = (\sigma, V)$ where $\sigma \in \Sigma^\omega$ and $V : \mathrm{prf}(\sigma) \to 2^P$ is a valuation function. Let $M = (\sigma, V)$ be a model, $\tau \in \mathrm{prf}(\sigma)$ and α be a formula. Then $M, \tau \models \alpha$ will stand for α being satisfied at τ in M. This notion is defined inductively in the expected manner.

- $M, \tau \models p$ iff $p \in V(\tau)$.

- $M, \tau \models \neg\alpha$ iff $M, \tau \not\models \alpha$.
- $M, \tau \models \alpha \vee \beta$ iff $M, \tau \models \alpha$ or $M, \tau \models \beta$.
- $M, \tau \models \langle a \rangle \alpha$ iff $\tau a \in \text{prf}(\sigma)$ and $M, \tau a \models \alpha$.
- $M, \tau \models \alpha \; U \; \beta$ iff there exists τ' such that $\tau\tau' \in \text{prf}(\sigma)$ and $M, \tau\tau' \models \beta$. Moreover for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $M, \tau\tau'' \models \alpha$.

Along with the usual propositional connectives $\wedge, \Rightarrow$ and $\equiv$ we will also use the propositional constants, $tt \stackrel{\text{def}}{=} p_1 \vee \neg p_1$ and $ff \stackrel{\text{def}}{=} \neg tt$. Some useful derived modalities are:

- $O\alpha \stackrel{\text{def}}{=} \bigvee_{a \in \Sigma} \langle a \rangle \alpha$.
- $\Diamond\alpha \stackrel{\text{def}}{=} ttU\alpha$.
- $\Box\alpha \stackrel{\text{def}}{=} \neg\Diamond\neg\alpha$.

Let $M = (\sigma, V)$ be a model and $\tau \in \text{prf}(\sigma)$. Then it is easy to check the following assertions.

- $M, \tau \models O\alpha$ iff $M, \tau' \models \alpha$ where $\tau' \in \text{prf}(\sigma)$ is such that $|\tau'| = |\tau| + 1$.
- $M, \tau \models \Diamond\alpha$ iff there exists a $\tau' \in \Sigma^*$ with $\tau\tau' \in \text{prf}(\sigma)$ such that $M, \tau\tau' \models \alpha$.
- $M, \tau \models \Box\alpha$ iff for each $\tau' \in \Sigma^*$, $\tau\tau' \in \text{prf}(\sigma)$ implies $M, \tau\tau' \models \alpha$.

Note that $O\alpha$ is the usual next-state operator of LTL.

We say that a formula $\alpha \in \text{LTL}(\Sigma)$ is *satisfiable* iff there exist a model $M = (\sigma, V)$ and $\tau \in \text{prf}(\sigma)$ such that $M, \tau \models \alpha$. This logic does not refer to the past either in the syntax or in the semantics. Hence the formula α is satisfiable iff there exists a model M such that $M, \varepsilon \models \alpha$. This is easy to check. The *satisfiability problem* for LTL is to develop a decision procedure which will determine whether a given formula α is satisfiable. We will later in this section describe such a decision procedure.

We now wish to formulate the model checking problem for $\text{LTL}(\Sigma)$. A *finite-state program* over Σ is a structure $Pr = (S, \longrightarrow, S_{in}, V_{Pr})$ where:

- S is a finite set of states.
- $\longrightarrow \subseteq S \times \Sigma \times S$ is a transition relation.
- $S_{in} \subseteq S$ is a set of initial states of the program.
- $V_{Pr} : S \to 2^P$ assigns a subset of P to each state of the program.

The members of P capture a finite set of basic assertions concerning the program which can usually be "read off" by examining the states of Pr and this is described by V_{Pr}. It will often be the case that the set of initial states is a singleton.

It is easy to arrange matters so that at each reachable state of the program at least one transition can be performed. We will assume that this is indeed the case for all program models we consider in this chapter. Further we will say "program" instead "finite-state program" from now on.

A *computation* of the program Pr is a pair (σ, ρ) where $\sigma \in \Sigma^\omega$ and $\rho : \mathrm{prf}(\sigma) \to S$ is a map which satisfies:

- $\rho(\varepsilon) \in S_{in}$.
- $\rho(\tau) \xrightarrow{a} \rho(\tau a)$ for each $\tau a \in \mathrm{prf}(\sigma)$.

Let (σ, ρ) be a computation of the program Pr. Then this computation canonically induces the model $M_{\sigma,\rho} = (\sigma, V_\rho)$ where V_ρ is given by: $V_\rho(\tau) = V_{Pr}(\rho(\tau))$ for each $\tau \in \mathrm{prf}(\sigma)$.

Let Pr be a program and α be a formula of LTL(Σ). We say that Pr *meets the specification* α — denoted $Pr \models \alpha$ — if for every computation (σ, ρ) of Pr, it is the case that $M, \varepsilon \models \alpha$ where M is the model induced by the computation (σ, ρ). The *model checking problem* is to decide for a given program Pr and a given formula α whether or not $Pr \models \alpha$. We will sketch a solution to the model checking problem later in this section.

Let $\mathcal{N} = (B, E, F, c_{in})$ be a finite elementary net system. In other words, it is an elementary net system in which both B, the set of conditions and E, the set of events are finite sets. We can associate the program $Pr_{\mathcal{N}} = (S, \longrightarrow, S_{in}, V_{Pr})$ with $\mathcal{N}$ as follows:

- $\Sigma = E$ and $P = B$.
- S is the least subset of 2^B and $\longrightarrow$ is the least subset of $S \times \Sigma \times S$ satisfying:
  - $c_{in} \in S$.
  - Suppose $c \in S$ and $e \in E$ such that ${}^\bullet e \subseteq c$ and $e^\bullet \cap c = \emptyset$. Then $c' \in S$ and $(c, e, c') \in \longrightarrow$ where $c' = (c - {}^\bullet e) \cup e^\bullet$.
- $S_{in} = \{c_{in}\}$.
- $V_{Pr}(c) = c$ for every $c \in S$.

Thus the so called case graph is the underlying transition system of the program. The conditions serve as the atomic propositions.

For $c \subseteq B$, let α_c be the formula $\bigwedge_{b \in c} b$. Now consider the specification $\Box\neg\alpha_c$ for some $c \subseteq B$. Then $Pr_{\mathcal{N}} \not\models \Box\neg\alpha_c$ iff c is a reachable state (i.e. $c \in S$) in $\mathcal{N}$. Next suppose e and e' are two events. Then $Pr_{\mathcal{N}} \models \Box\Diamond\langle e\rangle tt \Rightarrow \Box\Diamond\langle e'\rangle tt$ captures the fact that in $\mathcal{N}$, along every computation, if e occurs infinitely often then so does e'. A rich variety of liveness and safety properties can be expressed in LTL(Σ). For a substantial collection of examples the reader should see [83].

It turns out that both the satisfiability and model checking problems for LTL can be solved elegantly using Büchi automata [147]. We start with a brief introduction to these automata. A *Büchi automaton* over Σ is a tuple $\mathcal{B} = (Q, \longrightarrow, Q_{in}, F)$ where:

- Q is a finite non-empty set of states.
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation.
- $Q_{in} \subseteq Q$ is a set of initial states.
- $F \subseteq Q$ is a set of accepting states.

Let $\sigma \in \Sigma^\omega$. Then a *run* of $\mathcal{B}$ over σ is a map $\rho : \mathrm{prf}(\sigma) \longrightarrow Q$ such that:

- $\rho(\varepsilon) \in Q_{in}$.
- $\rho(\tau) \xrightarrow{a} \rho(\tau a)$ for each $\tau a \in \mathrm{prf}(\sigma)$.

The run ρ is *accepting* iff $\inf(\rho) \cap F \neq \emptyset$ where $\inf(\rho) \subseteq Q$ is given by $q \in \inf(\rho)$ iff $\rho(\tau) = q$ for infinitely many $\tau \in \mathrm{prf}(\sigma)$. Finally $\mathcal{L}(\mathcal{B})$, the *language of ω-words accepted by $\mathcal{B}$*, is:

$$\mathcal{L}(\mathcal{B}) = \{\sigma \mid \exists \text{ an accepting run of } \mathcal{B} \text{ over } \sigma\}.$$

The languages recognized by Büchi automata are called the *ω-regular languages*. For an excellent survey of regular languages and automata over infinite objects, the reader is referred to [140].

It is easy to solve the *emptiness problem* for Büchi automata; to determine whether or not the language accepted by a Büchi automaton is empty. This can be done in time linear in the size of the automaton where the size of a Büchi automaton is the number of states of the automaton [140].

We will now show how one can effectively construct for each $\alpha \in \mathrm{LTL}(\Sigma)$, a Büchi automaton $\mathcal{B}_\alpha$ such that the language of ω-words accepted by $\mathcal{B}_\alpha$ is non-empty iff α is satisfiable. This is an action-based version of the elegant solution presented in [147] for LTL.

Through the rest of the section we fix a formula α_0. To construct $\mathcal{B}_{\alpha_0}$ we first define the (Fischer-Ladner) closure of α_0. For convenience we will assume that the derived next-state modality modality O is included in the syntax of $\mathrm{LTL}(\Sigma)$. We take $cl(\alpha_0)$ to be the least set of formulas that satisfies:

- $\alpha_0 \in cl(\alpha_0)$.
- If $\neg\beta \in cl(\alpha_0)$ then $\beta \in cl(\alpha_0)$.
- If $\alpha \vee \beta \in cl(\alpha_0)$ then $\alpha, \beta \in cl(\alpha_0)$.
- If $\langle a \rangle \alpha \in cl(\alpha_0)$ then $\alpha \in cl(\alpha_0)$.
- If $O\alpha \in cl(\alpha_0)$ then $\alpha \in cl(\alpha_0)$.

- If $\alpha\ U\ \beta \in cl(\alpha_0)$ then $\alpha, \beta \in cl(\alpha_0)$. In addition, $O(\alpha\ U\ \beta) \in cl(\alpha_0)$.

Now $CL(\alpha_0)$, the *closure* of α_0, is defined to be:

$$CL(\alpha_0) = cl(\alpha_0) \cup \{\neg\beta \mid \beta \in cl(\alpha_0)\}.$$

In what follows $\neg\neg\beta$ will be identified with β. Moreover, throughout the section, all the formulas that we encounter will be assumed to be members of $CL(\alpha_0)$. For convenience, we shall often write CL instead of $CL(\alpha_0)$.

$A \subseteq CL$ is called an *atom* iff it satisfies :

- $\beta \in A$ iff $\neg\beta \notin A$.
- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- $\alpha\ U\ \beta \in A$ iff $\beta \in A$ or $\alpha, O(\alpha\ U\ \beta) \in A$.
- If $\langle a\rangle\alpha \in A$ and $\langle b\rangle\beta \in A$ then $a = b$.

$AT(\alpha_0)$ is the set of atoms and again we shall often write AT instead of $AT(\alpha_0)$. Finally we set U_{α_0}, the set of *until requirements* of α_0, to be the given by $U_{\alpha_0} = \{\alpha\ U\ \beta \mid \alpha\ U\ \beta \in CL\}$. We will often write U_0 instead of U_{α_0}.

The Büchi automaton $\mathcal{B}_{\alpha_0}$ (from now on denoted as $\mathcal{B}$) is now defined as $\mathcal{B} = (Q, \longrightarrow, Q_{in}, F)$, where the various components of $\mathcal{B}$ are specified as follows.

- $Q = AT \times 2^{U_0}$ is the set of states.
- The transition relation $\longrightarrow \subseteq Q \times \Sigma \times Q$ is given by $(A, x) \xrightarrow{a} (B, y)$ iff the following requirements are met:
  - For every $\langle a\rangle\alpha \in CL$, $\langle a\rangle\alpha \in A$ iff $\alpha \in B$ and for every $O(\alpha) \in CL$, $O(\alpha) \in A$ iff $\alpha \in B$.
  - if $\langle b\rangle\beta \in A$ then $b = a$.
  - if $x \neq \emptyset$ then $y = \{\alpha\ U\ \beta \mid \alpha\ U\ \beta \in x \text{ and } \beta \notin B\}$. If $x = \emptyset$ then $y = \{\alpha\ U\ \beta \mid \alpha\ U\ \beta \in B \text{ and } \beta \notin B\}$.
- $Q_{in} \subseteq Q$ is given by $(A, x) \in Q_{in}$ iff $\alpha_0 \in A$ and $x = \emptyset$.
- $F \subseteq Q$ is given by $(A, x) \in F$ iff $x = \emptyset$.

It is easy to show that $\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff α_0 is satisfiable. It is also easy to check that the size of $\mathcal{B}$ is at most exponential in the size of α_0. As observed earlier the emptiness problem for a Büchi automaton can be solved in time linear in the size of the automaton. Thus we arrive at:

Theorem 8.2.1 *The satisfiability problem for* LTL(Σ) *is decidable in exponential time.*

Turning now to the model checking problem we first recall that the *intersection problem* for Büchi automata can be easily solved. In other words, let $\mathcal{B}_1, \mathcal{B}_2$ be two Büchi automata both operating over Σ. Then one can effectively construct a Büchi automaton $\mathcal{B}$ over the same alphabet such that the language accepted by $\mathcal{B}$ is the intersection of the languages accepted by $\mathcal{B}_1$ and $\mathcal{B}_2$. Moreover, the size of $\mathcal{B}$ can be assumed to be bounded by $2n_1n_2$ where n_1 is the size of $\mathcal{B}_1$ and n_2 is the size of $\mathcal{B}_2$ [140].

Now let $Pr = (S, \longrightarrow, S_{in}, V_{Pr})$ be a program. We associate the Büchi automaton $\mathcal{B}_{Pr} = (S, \rightsquigarrow, S_{in}, S)$ over the alphabet $\Sigma \times 2^P$ with Pr where $\rightsquigarrow$ is given by: $(s, (a, R), s') \in \rightsquigarrow$ iff $(s, a, s') \in \longrightarrow$ and $V_{Pr}(s) = R$.

Let α be a specification. Then we construct the Büchi automaton $\mathcal{B}_{\neg\alpha}$ corresponding to the *negation* of α. Let $\mathcal{B}_{\neg\alpha} = (Q, \Longrightarrow, Q_{in}, F)$. Recall that each state in Q is of the form (A, x) where A is an atom. We now convert this automaton into the automaton $\widehat{\mathcal{B}} = (Q, \Rrightarrow, Q_{in}, F)$ over the alphabet $\Sigma \times 2^P$ by defining $\Rrightarrow$ as: $((A, x), (a, R), (B, y)) \in \Rrightarrow$ iff $((A, x), a, (B, y)) \in \Longrightarrow$ and $A \cap P = R$. Finally, let $\mathcal{B}$ be the Büchi automaton which accepts the intersection of the languages accepted by $\mathcal{B}_{Pr}$ and $\widehat{\mathcal{B}}$. It is straightforward to check that $Pr \models \alpha$ iff the language accepted by $\mathcal{B}$ is *empty*. An easy analysis of the size of $\mathcal{B}$ leads to:

Theorem 8.2.2 *The model checking problem for* LTL(Σ) *is decidable in time* $O(|Pr| \cdot 2^{|\alpha|})$.

In what follows, automata-theoretic constructions and expressiveness issues will play a considerable role. These topics can be treated in a simpler fashion if we eliminate atomic propositions. Most of the material we present can easily accommodate atomic propositions with some notational overhead. Hence from now on, we will not — except for some passing remarks — deal with atomic propositions. To be specific, the syntax of LTL(Σ) will be assumed to be:

$$\text{LTL}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a \rangle \alpha \mid \alpha \, U \, \beta.$$

Notice that a model is now just a member of Σ^ω with the semantics being the obvious one (tt is always true). The set of models of a formula constitute a language of infinite words. More precisely, each α induces the language L_α given by:

$$L_\alpha = \{\sigma \mid \sigma, \varepsilon \models \alpha\}.$$

A program is now just a finite-state transition system $Pr = (S, \longrightarrow, S_{in})$ over Σ. Each such program Pr has the language L_{Pr} associated with it. This is just the language accepted by the Büchi automaton $(S, \longrightarrow, S_{in}, S)$. It is also easy to see that $Pr \models \alpha$ iff $L_{Pr} \subseteq L_\alpha$ iff $L_{Pr} \cap L_{\neg\alpha} = \emptyset$.

8.3 Mazurkiewicz Traces and Trace Consistent Properties

Here we wish to introduce the notion of traces from the standpoint of sequences. This will enable us to define the notion of a trace consistent property. This

notion plays an important role in partial order based reducion methods. As pointed out in the introduction, it also provides the motivation for studying trace based linear time temporal logics.

A *(Mazurkiewicz) trace alphabet* is a pair (Σ, I), where Σ, the alphabet, is a finite set and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric *independence relation*. In most applications, Σ consists of the actions performed by a distributed system while I captures a static notion of causal independence between actions. The idea is that contiguous independent actions occur with no causal order between them. Thus, every sequence of actions from Σ corresponds to an interleaved observation of a partially-ordered stretch of system behaviour. This leads to a natural equivalence relation over execution sequences: two sequences are equated iff they correspond to different interleavings of the same partially-ordered stretch of behaviour.

For the rest of the section we fix a trace alphabet (Σ, I) and assume the terminology developed in the previous section for objects derived from Σ. We define $D = (\Sigma \times \Sigma) - I$ to be the *dependency relation*. Note that D is reflexive and symmetric. A set $p \subseteq \Sigma$ is called a *D-clique* iff $p \times p \subseteq D$. The equivalence relation $\approx_I \subseteq \Sigma^\infty \times \Sigma^\infty$ induced by I is given by:

$$\sigma \approx_I \sigma' \text{ iff } \sigma \restriction p = \sigma' \restriction p \text{ for every } D\text{-clique } p.$$

Here and elsewhere, if A is a finite set, $\rho \in A^\infty$ and $B \subseteq A$ then $\rho \restriction B$ is the sequence obtained by erasing from ρ all occurrences of letters in $A - B$.

Clearly $\approx_I$ is an equivalence relation. Notice that if $\sigma = \tau ab\sigma_1$ and $\sigma' = \tau ba\sigma_1$ with $(a, b) \in I$ then $\sigma \approx_I \sigma'$. Thus σ and σ' are identified if they differ only in the order of appearance of a pair of adjacent independent actions. In fact, for finite words, an alternative way to characterize $\approx_I$ is to say that $\sigma \approx_I \sigma'$ iff σ' can be obtained from σ by a finite sequence of permutations of adjacent independent actions. However the definition of $\approx_I$ in terms of permutations can not be directly transported to infinite words, which is why we work with the definition presented here.

The equivalence classes generated by $\approx_I$ are called *(Mazurkiewicz) traces*. A set of traces is called a *trace language*. The theory of traces is well developed and documented—see [24, 27] for basic material as well as a substantial number of references to related work.

A variety of models of distributed systems naturally have a trace alphabet associated with them [154]. It also turns out that many interesting properties of distributed systems respect the equivalence relation induced by these trace alphabets. This has important consequences for the practical verification of such properties.

The key notion in this context is that of a trace consistent property. To bring this out, we start with a trace alphabet (Σ, I) and recall the remarks concerning the abolition of atomic propositions at the end of Section 8.2. Let $L \subseteq \Sigma^\omega$. We say that L is *trace consistent* in case $\sigma \in L$ and $\sigma \approx_I \sigma'$ implies $\sigma' \in L$; for every $\sigma, \sigma' \in \Sigma^\omega$. In other words, either *all* members of a trace are in L or *none* of them are. We say that the formula α in LTL(Σ) is trace consistent in case L_α is

trace consistent. It is not hard to see that there is a one-to-one correspondence between trace languages and trace consistent languages of strings.

Now suppose Pr is a program over Σ which has a trace alphabet (Σ, I) associated with it in some natural manner. Suppose further that L_{Pr}, the linear time behaviour of Pr, is trace consistent (we will see a number of models of distributed programs that possess these features in the material to follow). Now consider a specification α which happens to be trace consistent. Then, as remarked at the end of Section 8.2, verifying $Pr \models \alpha$ boils down to verifying $L_{Pr} \subseteq L_\alpha$. Instead of checking $L_{Pr} \subseteq L_\alpha$ we can choose to check $L' \subseteq L_\alpha$ where L' is designed to be such that $L' \subseteq L_{Pr}$ and for every $\sigma \in L_{Pr}$, $[\sigma] \cap L' \neq \emptyset$. The key point is, the finite representation of L' can often be substantially smaller than the representation of Pr. This is the insight underlying many of the so called partial-order methods deployed in the model checking world [48, 108, 144].

As pointed out in the introduction this is also the main motivation for considering the trace-based linear time temporal logics that we will encounter later. We shall conclude this section with some examples.

Recall the material on elementary net systems introduced in Section 8.2. Suppose $\mathcal{N} = (B, E, F, c_{in})$ is an elementary net system. Each such system induces the independence relation $I_{\mathcal{N}}$ given by:

$$I_{\mathcal{N}} = \{(e_1, e_2) \mid ({}^\bullet e_1 \cup e_1^\bullet) \cap ({}^\bullet e_2 \cup e_2^\bullet) = \emptyset\}.$$

Let $e \in E$ and consider the formula $\Box\Diamond\langle e\rangle tt$. The property captured by this formula says that (along every computation) the event e occurs infinitely often. It is easy to see that this is a trace consistent property with respect to the trace alphabet $(E, I_{\mathcal{N}})$. Next consider the net system of Figure 8.1.

Consider the formula $\beta = \Box\Diamond(\langle e\rangle tt \wedge \langle e'\rangle tt)$. Suppose $\sigma = (e_1 e_2 e e')^\omega$ and $\sigma' = (e_1 e' e_2 e)^\omega$. Then $\sigma, \varepsilon \models \beta$ and $\sigma \approx_{I_{\mathcal{N}}} \sigma'$ but $\sigma', \varepsilon \not\models \beta$. Thus this property is not trace consistent with respect to the trace alphabet induced by this net system.

8.4 Product Languages and Automata

We will now exhibit a restricted but useful class of distributed behaviours that we call product behaviours. Such behaviours are generated by a network of sequential agents that coordinate their activities by performing common actions together. It will turn out that product behaviours are naturally trace consistent. They also constitute a clean and yet non-trivial subset of the class of trace behaviours considered later.

We first study product Büchi automata. We then formulate in Section 8.5 the product version of LTL(Σ). We will then use product Büchi automata to solve the satisfiability and model checking problems for the product version of LTL(Σ). The technical details — which we suppress here — can be found in [136]. The key notion underlying product behaviours is that of a distributed alphabet. It can be viewed as an "implementation" of a trace alphabet. As a result, distributed alphabets play a fundamental role in the automata-theoretic

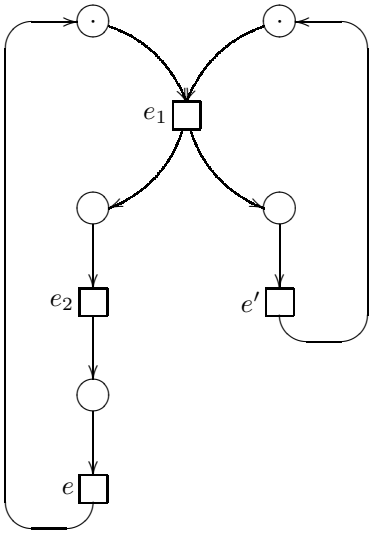


Figure 8.1: Example elementary net system.

aspects of trace languages [43, 157]. This will become more clear when the material in Section 8.6 is encountered.

A *distributed alphabet* is a family $\{\Sigma_p\}_{p\in\mathcal{P}}$ where $\mathcal{P}$ is a finite non-empty set of agents (also referred to as processes in the sequel) and Σ_p is a finite non-empty alphabet for each $p \in \mathcal{P}$. The idea is that whenever an action from Σ_p occurs, the agent p must participate in it. Hence the agents can constrain each other's behaviour, both directly and indirectly.

Trace alphabets and distributed alphabets are closely related to each other. Let $\widetilde{\Sigma} = \{\Sigma_p\}_{p\in\mathcal{P}}$ be a distributed alphabet. Then $\Sigma_{\mathcal{P}}$, the global alphabet associated with $\widetilde{\Sigma}$, is the collection $\bigcup_{p\in\mathcal{P}} \Sigma_p$. The distribution of $\Sigma_{\mathcal{P}}$ over $\mathcal{P}$ can be described using a *location function* $\mathrm{loc}_{\widetilde{\Sigma}} : \Sigma_{\mathcal{P}} \to 2^{\mathcal{P}}$ defined as follows:

$$\mathrm{loc}_{\widetilde{\Sigma}}(a) = \{p \mid a \in \Sigma_p\}.$$

This in turn induces the relation $I_{\widetilde{\Sigma}} \subseteq \Sigma_{\mathcal{P}} \times \Sigma_{\mathcal{P}}$ given by:

$$(a, b) \in I_{\widetilde{\Sigma}} \text{ iff } \mathrm{loc}_{\widetilde{\Sigma}}(a) \cap \mathrm{loc}_{\widetilde{\Sigma}}(b) = \emptyset.$$

Clearly $I_{\widetilde{\Sigma}}$ is irreflexive and symmetric and hence $(\Sigma_{\mathcal{P}}, I_{\widetilde{\Sigma}})$ is a trace alphabet. Thus every distributed alphabet canonically induces a trace alphabet. Two actions are independent according to $\widetilde{\Sigma}$ if they are executed by disjoint sets of processes. Henceforth, we write loc for $\mathrm{loc}_{\widetilde{\Sigma}}$ whenever $\widetilde{\Sigma}$ is clear from the context.

Going in the other direction there are, in general, many different ways to implement a trace alphabet as a distributed alphabet. A standard approach is to create a separate agent for each maximal D-clique generated by (Σ, I). Recall

that a D-clique of (Σ, I) is a non-empty subset $p \subseteq \Sigma$ such that $p \times p \subseteq D$. Let $\mathcal{P}$ be the set of maximal D-cliques of (Σ, I). This set of processes induces the distributed alphabet $\widetilde{\Sigma} = \{\Sigma_p\}_{p \in \mathcal{P}}$ where $\Sigma_p = p$ for every process p. The alphabet $\widetilde{\Sigma}$ implements (Σ, I) in the sense that the canonical trace alphabet induced by it is exactly (Σ, I). In other words, $\Sigma_{\mathcal{P}} = \Sigma$ and $I_{\widetilde{\Sigma}} = I$.

For example, consider the trace alphabet (Σ, I) where $\Sigma = \{a, b, d\}$ and $I = \{(a, b), (b, a)\}$. The canonical D-clique implementation of (Σ, I) yields the distributed alphabet $\widetilde{\Sigma} = \{\{a, d\}, \{d, b\}\}$.

Through the rest of the section we fix a distributed alphabet $\{\Sigma_p\}_{p \in \mathcal{P}}$ and set $\Sigma = \Sigma_{\mathcal{P}}$. It will be convenient to assume that $\mathcal{P} = \{1, 2, \ldots, K\}$. Further, the ith component of a K-tuple $x = (x_1, x_2, \ldots, x_K)$ will be written as $x[i]$. In other words, $x[i] = x_i$.

A *product Büchi automaton* over $\widetilde{\Sigma}$ is a structure $\mathcal{A} = (\{\mathcal{A}_i\}_{i=1}^K, Q_{in})$ where $\mathcal{A}_i = (Q_i, \longrightarrow_i, F_i, F_i^\omega)$ for each i such that :

- Q_i is a finite set of i-local states.
- $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$ is the transition relation of the ith component.
- $F_i \subseteq Q_i$ is a set of finitary accepting states.
- $F_i^\omega \subseteq Q_i$ is a set of infinitary accepting states.
- $Q_{in} \subseteq Q_1 \times Q_2 \times \cdots \times Q_K$ is a set of global initial states.

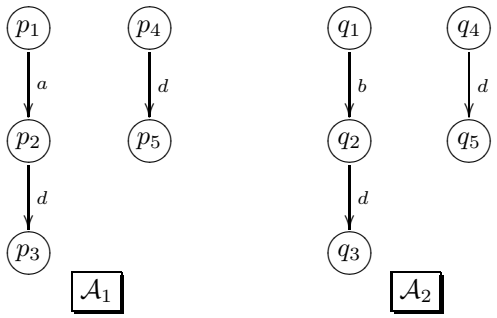
We use two types of accepting states for the components in order to be able to handle both finite and infinite behaviours. Even if one is interested only in global infinite behaviours, finite behaviours at the component level must be treated; a component might quit after engaging in a finite number of actions while a part of the network runs forever. We use global initial states to obtain the required expressive power. In general, the automaton will not be able to branch off into different parts of the state space, starting from a single global initial state. This will be brought out through a simple example after we define the language behaviour of product automata. The same example will also illustrate why using the cartesian product of local initial state sets as global initial states will result in a loss of expressive power.

Let $\mathcal{A} = (\{\mathcal{A}_i\}_{i=1}^K, Q_{in})$ be a product Büchi automaton over $\widetilde{\Sigma}$. From now on we will say just "product automata". Also, we shall often suppress the mention of $\widetilde{\Sigma}$. We will also write $\{\mathcal{A}_i\}$ instead of $\{\mathcal{A}_i\}_{i=1}^K$. Let $\mathcal{A}_i = (Q_i, \longrightarrow_i, F_i, F_i^\omega)$. Then we set $Q_G^{\mathcal{A}} = Q_1 \times Q_2 \times \cdots \times Q_K$. When $\mathcal{A}$ is clear from the context, we will write Q_G instead of $Q_G^{\mathcal{A}}$. The global transition relation of $\mathcal{A}$ is denoted as $\longrightarrow_{\mathcal{A}}$ and it is the subset of $Q_G \times \Sigma \times Q_G$ given by:

$$q \xrightarrow{a}_{\mathcal{A}} q' \text{ iff } \forall\, i \in \mathrm{loc}(a) : q[i] \xrightarrow{a}_i q'[i] \text{ and } \forall\, i \notin \mathrm{loc}(a) : q[i] = q'[i].$$

Let $\sigma \in \Sigma^\infty$. A *run* of $\mathcal{A}$ over σ is a map $\rho : \mathrm{Prf}(\sigma) \longrightarrow Q_G$ which satisfies:

- $\rho(\varepsilon) \in Q_{in}$.



$$Q_{in} = \{(p_1, q_4), (p_4, q_1)\}, \quad F_1 = \{p_3, p_5\}, \quad F_2 = \{q_3, q_5\}, \quad \widehat{F}_1 = \widehat{F}_2 = \emptyset.$$

Figure 8.2: Product automaton accepting $L = \{ad, bd\}$.

- $\forall\ \tau a \in \text{prf}(\sigma).\ \rho(\tau) \xrightarrow{a}_{\mathcal{A}} \rho(\tau a)$.

A simple but useful property of runs is the following. Suppose ρ is a run of the product automaton $\mathcal{A}$ over σ. Further suppose that $\tau, \tau' \in \text{Prf}(\sigma)$ such that $\tau \restriction i = \tau' \restriction i$ for some i. Then $\rho(\tau)[i] = \rho(\tau')[i]$.

Let ρ be a run of the product automaton $\mathcal{A}$ over σ. Then ρ is *accepting* iff for each i, the following condition is satisfied:

- If $\sigma \restriction i$ is finite then $\rho(\tau)[i] \in F_i$ where $\tau \in \text{prf}(\sigma)$ such that $\tau \restriction i = \sigma \restriction i$.
- If $\sigma \restriction i$ is infinite then $\rho(\tau a)[i] \in F_i^\omega$ for infinitely many $\tau a \in \text{prf}(\sigma)$ with $a \in \Sigma_i$.

If $\sigma \restriction i$ is finite then clearly there exists $\tau \in \text{prf}(\sigma)$ such that $\tau \restriction i = \sigma \restriction i$. Now the above property of runs assures us that the notion of an accepting run is well-defined. In case $\sigma \restriction i$ is infinite the acceptance condition can also be phrased as:

- $\rho(\tau)[i] \in F_i^\omega$ for infinitely many $\tau \in \text{prf}(\sigma)$.

This once again follows easily from the definition of a run. We now define $\mathcal{L}(\mathcal{A})$, the *language accepted by the product automaton* $\mathcal{A}$ as,

$$\mathcal{L}(\mathcal{A}) = \{\sigma \mid \exists \text{ an accepting run of } \mathcal{A} \text{ over } \sigma\}.$$

Now consider the alphabet $(\{a, d\}, \{d, b\})$ and the language $L = \{ad, bd\}$. Figure 8.2 shows a product automaton over this alphabet which accepts L. It is easy to verify that *no* product automaton over this alphabet with a *single* global initial state can accept L. It is also easy to verify that no product automaton whose set of initial states is a cartesian product of component initial state sets can accept this language.

A crucial property of product automata is that they accept $\approx$-consistent languages.

Lemma 8.4.1 *Let $\mathcal{A} = (\{\mathcal{A}_i\}, Q_{in})$ be a product automaton over $\widetilde{\Sigma}$. Then $\mathcal{L}(\mathcal{A})$ is trace consistent.*

The class of languages accepted by product automata can now be characterized. To this end we define the K-ary operation $\otimes : 2^{\Sigma_1^\infty} \times 2^{\Sigma_2^\infty} \times \cdots \times 2^{\Sigma_K^\infty} \to 2^{\Sigma^\infty}$ via $\otimes(L_1, \ldots, L_K) = \{\sigma \mid \sigma \restriction i \in L_i \text{ for each } i\}$.

In what follows we will write $L = L_1 \otimes L_2 \cdots \otimes L_K$ to denote the fact $\otimes(L_1, \ldots, L_K) = L$. We say that $L \subseteq \Sigma^\infty$ is a *direct product language* over $\widetilde{\Sigma}$ iff $\exists\ L_i \subseteq \Sigma_i^\infty$ for each i such that $L = L_1 \otimes L_2 \otimes \cdots \otimes L_K$. Here are two useful properties of direct product languages. In stating this result and elsewhere we will say "product language" instead of "product language over $\widetilde{\Sigma}$" etc.

Proposition 8.4.2

(1) Let L be a direct product language and $\sigma \in \Sigma^\infty$. Then $\sigma \in L$ iff for each i there exists $\sigma_i \in L$ such that $\sigma \restriction i = \sigma_i \restriction i$.

(2) Let $L \subseteq \Sigma^\infty$. Then L is a direct product language iff $L = \widehat{L}_1 \otimes \widehat{L}_2 \otimes \cdots \otimes \widehat{L}_K$ where $\widehat{L}_i = \{\sigma \restriction i \mid \sigma \in L\}$ for each i.

As usual, for an alphabet Σ and $L \subseteq \Sigma^\infty$ we say that L is *regular* iff $L \cap \Sigma^*$ is a regular subset of Σ^* and $L \cap \Sigma^\omega$ is an ω-regular subset of Σ^ω as described in Section 8.2. We can now define the class of languages accepted by product automata.

Definition 8.4.3

- $\mathcal{R}_0^\otimes(\widetilde{\Sigma})$ is the subset of 2^{Σ^∞} given by $L \in \mathcal{R}_0^\otimes(\widetilde{\Sigma})$ iff $L = L_1 \otimes L_2 \otimes \cdots \otimes L_K$ with each L_i a regular subset of Σ_i^∞.
- $\mathcal{R}^\otimes(\widetilde{\Sigma})$ is the least subset of 2^{Σ^∞} which contains $\mathcal{R}_0^\otimes$ and is closed under finite unions.

The class $\mathcal{R}^\otimes(\widetilde{\Sigma})$ defined above will be called the *regular product languages* over $\widetilde{\Sigma}$. As usual, we shall often write $\mathcal{R}_0^\otimes$ instead of $\mathcal{R}_0^\otimes(\widetilde{\Sigma})$ and write $\mathcal{R}^\otimes$ instead of $\mathcal{R}^\otimes(\widetilde{\Sigma})$. An interesting observation concerning $\mathcal{R}^\otimes$ is the following:

Proposition 8.4.4 *$\mathcal{R}^\otimes$ is closed under boolean operations.*

It turns out that $\mathcal{R}^\otimes$ is precisely the class of languages accepted by product automata.

Theorem 8.4.5 ([136]) *Let $L \subseteq \Sigma^\infty$. Then $L \in \mathcal{R}^\otimes$ iff there exists a product automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$.*

We shall be using product automata to settle the decidability and model checking problems for the logic LTL$^\otimes$ to be introduced in the next section. In anticipation of this, we shall put down two more results concerning product automata. While doing so and elsewhere the *size* of the product automaton $\mathcal{A}$ will be understood to be $|Q_G|$.

Theorem 8.4.6 *Let $\mathcal{A}$ be a product automaton. Then the question $\mathcal{L}(\mathcal{A}) \stackrel{?}{=} \emptyset$ can be settled in time $O(2^{2K} \cdot n^2)$ where n is the size of $\mathcal{A}$.*

Theorem 8.4.7 *Let $\mathcal{A}^1$ and $\mathcal{A}^2$ be two product automata. Then one can effectively construct a product automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^1) \cap \mathcal{L}(\mathcal{A}^2)$ and moreover $n = O(2^K \cdot n_1 \cdot n_2)$ where n is the size of $\mathcal{A}$ and n_ℓ is the size of $\mathcal{A}^\ell$ for $\ell = 1, 2$.*

8.5 A Product Version of LTL

We now wish to design a product version of LTL denoted $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$. The set of formulas and their *locations* are given by:

- tt is a formula and $\mathrm{loc}(tt) = \emptyset$.
- Suppose α and β are formulas. Then so are $\neg\alpha$ and $\alpha \vee \beta$. Furthermore, $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$ and $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.
- Suppose $a \in \Sigma_i$ and α is a formula with $\mathrm{loc}(\alpha) \subseteq \{i\}$. Then $\langle a \rangle_i \alpha$ is a formula and $\mathrm{loc}(\langle a \rangle_i \alpha) = \{i\}$.
- Suppose α and β are formulas such that $\mathrm{loc}(\alpha), \mathrm{loc}(\beta) \subseteq \{i\}$. Then $\alpha \mathcal{U}_i \beta$ is a formula. Moreover, $\mathrm{loc}(\alpha \mathcal{U}_i \beta) = \{i\}$.

We note that each formula in $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ is a boolean combination of formulas taken from the set $\bigcup_{i \in \mathrm{loc}} \mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$ where, for each i,

$$\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma}) = \{\alpha \mid \alpha \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma}) \text{ and } \mathrm{loc}(\alpha) \subseteq \{i\}\}.$$

Stated differently, the syntax of $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$ is given inductively by:

- $tt \in \mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$.
- If α and β are in $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$ then $\neg\alpha$ and $\alpha \vee \beta$ are in $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$.
- If α is in $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$ and $a \in \Sigma_i$ then $\langle a \rangle_i \alpha$ is in $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$.
- If α and β are in $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$ then $\alpha \mathcal{U}_i \beta$ is in $\mathrm{LTL}_i^{\otimes}(\widetilde{\Sigma})$.

Once again, we have chosen to avoid dealing with atomic propositions for the sake of convenience. They can be introduced in a local fashion as done in [136]. The decidability result to be presented will go through with minor notational overheads.

As before, we will often suppress the mention of $\widetilde{\Sigma}$. We will also often write τ_i, τ_i' and τ_i'' instead of $\tau \restriction i$, $\tau' \restriction i$ and $\tau'' \restriction i$, respectively with $\tau, \tau', \tau'' \in \Sigma^*$.

A model is a sequence $\sigma \in \Sigma^\infty$ and the semantics of this logic is given, as before, with $\tau \in \mathrm{prf}(\sigma)$.

- $\sigma, \tau \models tt$.

- $\sigma, \tau \models \neg\alpha$ iff $\sigma, \tau \not\models \alpha$.
- $\sigma, \tau \models \alpha \vee \beta$ iff $\sigma, \tau \models \alpha$ or $\sigma, \tau \models \beta$.
- $\sigma, \tau \models \langle a \rangle_i \alpha$ iff there exists $\tau' \in \mathrm{prf}(\sigma)$ such that $\sigma, \tau' \models \alpha$ and $\tau'_i = \tau_i a$. (recall that $\tau'_i = \tau' \restriction i$.)
- $\sigma, \tau \models \alpha \mathcal{U}_i \beta$ iff there exists τ' such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Further, for every $\tau'' \in \mathrm{prf}(\tau')$, if $\varepsilon \preceq \tau''_i \prec \tau'_i$ then $\sigma, \tau\tau'' \models \alpha$.

As before we derive some useful modalities:

- $O_i\alpha \stackrel{\mathrm{def}}{=} \bigvee_{a \in \Sigma_i} \langle a \rangle_i \alpha$.
- $\Diamond_i \alpha \stackrel{\mathrm{def}}{=} tt \mathcal{U}_i \alpha$.
- $\Box_i \alpha \stackrel{\mathrm{def}}{=} \neg \Diamond_i \neg \alpha$.

Let $M = \sigma$ be a model and $\tau \in \mathrm{prf}(\sigma)$. The following assertions can now easily be checked.

- $\sigma, \tau \models O_i\alpha$ iff there exists $\tau' \in prf(\sigma)$ such that $\sigma, \tau' \models \alpha$ and $|\tau'_i| = |\tau_i| + 1$.
- $\sigma, \tau \models \Diamond_i \alpha$ iff there exists τ' with $\tau\tau' \in \mathrm{prf}(\sigma)$ such that $\sigma, \tau\tau' \models \alpha$.
- $\sigma, \tau \models \Box_i \alpha$ iff for each τ', $\tau\tau' \in \mathrm{prf}(\sigma)$ implies $\sigma, \tau\tau' \models \alpha$.

Note that $O_i\alpha$ is the i-local version of the usual next-state operator of LTL.

We will say that a formula $\alpha \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ is *satisfiable* if there exist $\sigma \in \Sigma^{\infty}$ and $\tau \in \mathrm{prf}(\sigma)$ such that $\sigma, \tau \models \alpha$. The *language defined by* α is given by

$$L_\alpha = \{\sigma \in \Sigma^{\infty} \mid \sigma, \varepsilon \models \alpha\}.$$

We will show the satisfiability problem for $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ is solvable in deterministic exponential time. This will be achieved by effectively constructing a product automaton $\mathcal{A}_\alpha$ for each $\alpha \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ such that the language accepted by $\mathcal{A}_\alpha$ is non-empty iff α is satisfiable. Our construction is a generalization of the one for LTL in Section 8.2. The solution to the satisfiability problem will at once lead to a solution to the model checking problem for programs modelled as a product of sequential agents.

Through the rest of the section we fix a formula $\alpha_0 \in \mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$. As before we will for convenience assume that the derived local next-state modality O_i is included in the syntax of $\mathrm{LTL}^{\otimes}$. In order to construct $\mathcal{A}_{\alpha_0}$ we first define the (Fischer-Ladner) closure of α_0. As a first step let $cl(\alpha_0)$ be the least set of formulas satisfying:

- $\alpha_0 \in cl(\alpha_0)$.
- $\neg\alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.

- $\alpha \vee \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$.
- $\langle a \rangle_i \alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.
- $\alpha \mathcal{U}_i \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$. In addition, $O_i(\alpha \mathcal{U}_i \beta) \in cl(\alpha_0)$.

We will now take the *closure* of α_0 to be $CL(\alpha_0) = cl(\alpha_0) \cup \{\neg\alpha \mid \alpha \in cl(\alpha_0)\}$. From now on we shall identify $\neg\neg\alpha$ with α. Set $CL_i(\alpha_0) = CL(\alpha_0) \cap \mathrm{LTL}_i^{\otimes}$ for each i. We will often write CL instead of $CL(\alpha_0)$ and CL_i instead of $CL_i(\alpha_0)$. All formulas considered from now on will be assumed to belong to CL unless otherwise stated.

An *i-type atom* is a subset $A \subseteq CL_i$ which satisfies:

- $tt \in A$.
- $\alpha \in A$ iff $\neg\alpha \notin A$.
- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- $\alpha \mathcal{U}_i \beta \in A$ iff $\beta \in A$ or $\alpha, O_i(\alpha \mathcal{U}_i \beta) \in A$.

The set of i-type atoms is denoted AT_i. We next define, for each $\alpha \in CL(\alpha_0)$ and $(A_1, \ldots, A_K) \in AT_1 \times \cdots \times AT_K$, the predicate Member$(\alpha, (A_1, \ldots, A_K))$. For convenience this predicate will be denoted as $\alpha \in (A_1, \ldots, A_K)$ and is given inductively by:

- Let $\alpha \in CL_i$. Then $\alpha \in (A_1, \ldots, A_K)$ iff $\alpha \in A_i$.
- Let $\alpha = \neg\beta$. Then $\alpha \in (A_1, \ldots, A_K)$ iff $\beta \notin (A_1, \ldots, A_K)$.
- Let $\alpha = \beta \vee \gamma$. Then $\alpha \in (A_1, \ldots, A_K)$ iff $\beta \in (A_1, \ldots, A_K)$ or $\gamma \in (A_1, \ldots, A_K)$.

Finally, we set $U_i = \{\alpha \mathcal{U}_i \beta \mid \alpha \mathcal{U}_i \beta \in CL_i(\alpha_0)\}$ for each i. The product automaton $\mathcal{A}_{\alpha_0}$ associated with α_0 is now defined to be $\mathcal{A}_{\alpha_0} = (\{\mathcal{A}_i\}, Q_{in})$ where, for each i, $\mathcal{A}_i = (Q_i, \longrightarrow_i, F_i, F_i^{\omega})$ is specified as follows:

- $Q_i = AT_i \times \{\mathsf{off}, \mathsf{on}\} \times 2^{U_i}$
- $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$ is given by, $(A, x, u) \xrightarrow{a}_i (B, y, v)$ iff the following conditions are met.
  (1) $x = \mathsf{on}$ and for all $\langle a \rangle_i \alpha \in CL_i(\alpha_0)$, $\langle a \rangle_i \alpha \in A$ iff $\alpha \in B$ and for all $O_i \alpha \in CL_i(\alpha_0)$, $O_i \alpha \in A$ iff $\alpha \in B$. Moreover, if $\langle b \rangle_i \beta \in A$ then $b = a$.
  (2) If $u \neq \emptyset$ then $v = \{\alpha \mathcal{U}_i \beta \mid \alpha \mathcal{U}_i \beta \in u \text{ and } \beta \notin B\}$. If $u = \emptyset$ then $v = \{\alpha \mathcal{U}_i \beta \mid \alpha \mathcal{U}_i \beta \in B \text{ and } \beta \notin B\}$.
- $F_i \subseteq Q_i$ is given by: $(A, x, u) \in F_i$ iff $x = \mathsf{off}$ and for all $\langle a \rangle_i \alpha \in CL_i(\alpha_0)$, $\langle a \rangle_i \alpha \notin A$ and for all $O_i \alpha \in CL_i(\alpha_0)$, $O_i \alpha \notin A$.

- $F_i^\omega \subseteq Q_i$ is given by: $(A, x, u) \in F_i^\omega$ iff $u = \emptyset$.
- $Q_{in} \subseteq Q_1 \times Q_2 \times \ldots \times Q_K$ is given by: $((A_1, x_1, u_1), \ldots, (A_K, x_K, u_K)) \in Q_{in}$ iff $\alpha_0 \in (A_1, \ldots, A_K)$ and $u_i = \emptyset$ for every i.

It is not difficult to now establish the next result by an application of Theorem 8.4.6.

Theorem 8.5.1 *α_0 is satisfiable iff $\mathcal{L}(\mathcal{A}_{\alpha_0}) \neq \emptyset$. Hence the satisfiability problem for* LTL$^\otimes$ *is decidable in exponential time.*

We now turn to the model checking problem for LTL$^\otimes$. A product program (over $\widetilde{\Sigma}$) is a structure $Pr = (\{Pr_i\}_{i=1}^K, Q_{in}^{Pr})$ where, for each i, $Pr_i = (Q_i, \longrightarrow_i)$ with Q_i a finite set and $\longrightarrow_i \subseteq Q_i \times \Sigma_i \times Q_i$. Since we have agreed to drop atomic propositions there is no need for (local) interpretations for the atomic propositions. Let us further assume for convenience that Q_{in}^{Pr} is a singleton with q_{in} as its sole member and with $q_{in}[i] = q_{in}^i$ for each i. With each such program we can associate the product automaton $\mathcal{A}_{Pr} = (\{\mathcal{A}_i\}_{i=1}^K, \{q_{in}\})$ where $\mathcal{A}_i = (Q_i, \longrightarrow_i, Q_i, Q_i)$ for each i.

Now let Pr be a product program and α_0 be a formula of LTL$^\otimes$. As in the case for LTL, we say that Pr *meets the specification* α_0 — again denoted $Pr \models \alpha_0$ — iff $\sigma, \varepsilon \models \alpha_0$ for every $\sigma \in \mathcal{L}(\mathcal{A}_{Pr})$. Once again, using Theorem 8.4.7 it is not difficult to prove the following.

Theorem 8.5.2 *The model checking problem for* LTL$^\otimes$ *is decidable in time* $O(|Pr| \cdot 2^{|\alpha_0|})$.

We wish to observe that each product program can be represented as a Σ-labelled 1-safe net system. To see this let $Pr = (\{Pr_i\}_{i=1}^K, \{q_{in}\})$ be a product program. Let's assume without loss of generality that the family of local states $\{Q_i\}$ is pairwise disjoint. We set $Q = \bigcup_{i \in \mathcal{P}} Q_i$ and define an a-state to be a map $q_a : \mathrm{loc}(a) \to Q$ which satisfies $q_a(i) \in Q_i$ for each i in $\mathrm{loc}(a)$. (A more elaborate development of these notions will appear in the next section). An a-event is a pair of a-states (q_a, q_a') which satisfies $q_a(i) \xrightarrow{a}_i q_a'(i)$ for each i in $\mathrm{loc}(a)$. We let E_a be the set of a-events. We can now define the Σ-labelled 1-safe net system representing Pr to be $\mathcal{N} = (B, E, F, c_{in}, \phi)$ where:

- $B = Q$
- $E = \bigcup_{a \in \Sigma} E_a$
- Let $q_i \in Q_i$ and $e = (q_a, q_a') \in E_a$. Then $(q_i, e) \in F$ iff $i \in \mathrm{loc}(a)$ and $q_a(i) = q_i$. Similarly $(e, q_i) \in F$ iff $i \in \mathrm{loc}(a)$ and $q_a'(i) = q_i$.
- Let $e \in E$. Then $\phi(e) = a$ iff e is an a-event.

On the other hand each 1-safe net system which is covered by a set of S-components can be viewed as a (deterministic) product program; the alphabet of each component is its set of events. If necessary, S-complementation can be

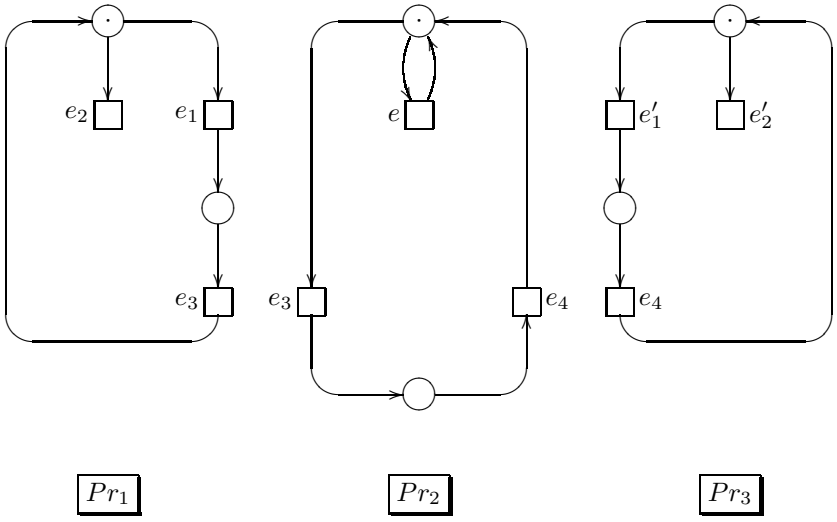


Figure 8.3: 1-safe net with three components.

performed to ensure that the system is covered by a set of S-components. We do not wish to enter into details here. Instead we show on Figure 8.3 an example of a 1-safe net system composed out of three components.

Let Pr denote the associated product program over the distributed alphabet $\{\{e_1, e_2, e_3\}, \{e_3, e_4\}, \{e'_1, e'_2, e_4\}\}$. Then it is easy to check that

$$Pr \models \Box_1 O_1 tt \Rightarrow \Box_3 O_3 tt.$$

This property says that along every computation, if the first component executes infinitely often then so does the third component. The point to note is that the first component and the third component do not have any common events and hence there is no direct communication between them. Nevertheless through the power of the boolean connectives alone the logic can make assertions about the way components that are "far apart" are required to influence each other's behaviour.

8.6 Trace Languages and Automata

Traces have many equivalent representations. Here we shall view them as restricted Σ-labelled partial orders. Abusing terminology we shall call these objects also *traces*. We will then argue that these objects are in a rather precise sense the same as the objects called traces defined in Section 8.3 in terms of equivalence classes of sequences.

Let T be a Σ-labelled poset. In other words, $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $Y \subseteq E$ we define $\downarrow Y = \{x \mid \exists y \in Y : \ x \leq y\}$ and

$\uparrow Y = \{x \mid \exists y \in Y : \ y \leq x\}$. In case $Y = \{y\}$ is a singleton we shall write $\downarrow y$ ($\uparrow y$) instead of $\downarrow\{y\}$ ($\uparrow \{y\}$). We also let $\lessdot$ be the relation: $x \lessdot y$ iff $x < y$ and for all $z \in E$, $x \leq z \leq y$ implies $x = z$ or $z = y$.

A *trace* (over (Σ, I)) is a Σ-labelled poset $T = (E, \leq, \lambda)$ satisfying:

(T1) $\forall e \in E.$ $\quad \downarrow e$ is a finite set
(T2) $\forall e, e' \in E.$ $\ e \lessdot e'$ implies $\lambda(e)\ D\ \lambda(e')$.
(T3) $\forall e, e' \in E.$ $\ \lambda(e)\ D\ \lambda(e')$ implies $e \leq e'$ or $e' \leq e$.

We shall refer to members of E as *events*. The trace $T = (E, \leq, \lambda)$ is said to be *finite* if E is a finite set. Otherwise it is an *infinite* trace. Note that E is always a countable set. T is said to be *non-empty* in case $E \neq \emptyset$. We let $\mathbb{TR}^*(\Sigma, I)$ be the set of finite traces and $\mathbb{TR}^\omega(\Sigma, I)$ be the set of infinite traces over (Σ, I) and set $\mathbb{TR}(\Sigma, I) = \mathbb{TR}^*(\Sigma, I) \cup \mathbb{TR}^\omega(\Sigma, I)$. Often we will write $\mathbb{TR}^*$ instead of $\mathbb{TR}^*(\Sigma, I)$ etc. As before, a subset of traces $L_{Tr} \subseteq \mathbb{TR}$ will be called a *trace language*.

Let $T = (E, \leq, \lambda)$ be a trace. The finite prefixes of T, to be called configurations, will play a crucial role in what follows. A *configuration* of T is a finite subset $c \subseteq E$ such that $c = \downarrow c$. We let $\mathcal{C}_T$ be the set of configurations of T and let c, c', c'' range over $\mathcal{C}_T$. Note that $\emptyset$, the empty set, is a configuration and $\downarrow\ e$ is a configuration for every $e \in E$. Finally, the transition relation $\longrightarrow_T \subseteq \mathcal{C}_T \times \Sigma \times \mathcal{C}_T$ is given by: $c \xrightarrow{a}_T c'$ iff there exists $e \in E$ such that $\lambda(e) = a$ and $e \notin c$ and $c' = c \cup \{e\}$. It is easy to see that if $c \xrightarrow{a}_T c'$ and $c \xrightarrow{a}_T c''$ then $c' = c''$.

Note that we have now introduced two different notions of traces; one in terms of equivalence classes of strings as in Section 8.3 and the other in terms of Σ-labelled partial orders as in this section. We now sketch briefly the constructions that show that $\Sigma^\infty/\approx_I$ and $\mathbb{TR}(\Sigma, I)$ represent the same class of objects. We shall construct representation maps $\mathsf{st} : \Sigma^\infty/\approx_I \to \mathbb{TR}(\Sigma, I)$ and $\mathsf{ts} : \mathbb{TR}(\Sigma, I) \to \Sigma^\infty/\approx_I$ and state some results which show that these maps are "inverses" of each other. We shall not prove these results. The details can be easily obtained using the constructions developed in [154] for relating traces and event structures.

Henceforth, we will not distinguish between isomorphic elements in $\mathbb{TR}(\Sigma, I)$. In other words, whenever we write $T = T'$ for traces $T = (E, \leq, \lambda)$ and $T' = (E', \leq', \lambda')$, we mean that there is a label-preserving isomorphism between T and T'.

Recall that for $\sigma \in \Sigma^\infty$, $[\sigma]$ stands for the $\approx_I$-equivalence class containing σ. We now define $\mathsf{st} : \Sigma^\infty \to \mathbb{TR}(\Sigma, I)$. Let $\sigma \in \Sigma^\infty$. Then $\mathsf{st}(\sigma) = (E, \leq, \lambda)$ where:

- $E = \{\tau a \mid \tau a \in \mathrm{prf}(\sigma)\}$. Recall that $\tau \in \Sigma^*$ and $a \in \Sigma$. Thus $E = \mathrm{prf}(\sigma) - \{\varepsilon\}$, where ε is the null string.
- $\leq \subseteq E \times E$ is the least partial order which satisfies: For all $\tau a, \tau' b \in E$, if $\tau a \preceq \tau' b$ and $(a, b) \in D$ then $\tau a \leq \tau' b$.
- For $\tau a \in E$, $\lambda(\tau a) = a$.

The map st induces a natural map st' from $\Sigma^\infty/\approx_I$ to $\mathbb{TR}(\Sigma, I)$ defined by $\mathsf{st}'([\sigma]) = \mathsf{st}(\sigma)$. One can show that if $\sigma, \sigma' \in \Sigma^\infty$, then $\sigma \approx_I \sigma'$ iff $\mathsf{st}(\sigma) = \mathsf{st}(\sigma')$. This observation guarantees that st' is well-defined. In fact, henceforth we shall write st to denote both st and st'.

Next, let $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$. Then $\sigma \in \Sigma^\infty$ is a *linearization* of T iff there exists a map $\rho : \mathrm{prf}(\sigma) \to \mathcal{C}_T$, such that the following conditions are met:

- $\rho(\varepsilon) = \emptyset$.
- $\forall \tau a \in \mathrm{prf}(\sigma)$ with $\tau \in \Sigma^*$, $\rho(\tau) \stackrel{a}{\longrightarrow}_T \rho(\tau a)$.
- $\forall e \in E\ \exists \tau \in \mathrm{prf}(\sigma).\ e \in \rho(\tau)$.

The function ρ will be called a *run map* of the linearization σ. Note that the run map of a linearization is unique. In what follows, we shall let $lin(T)$ to be the set of linearizations of the trace T.

We can now define the map $\mathsf{ts} : \mathbb{TR}(\Sigma, I) \to \Sigma^\infty/\approx_I$ as: $\mathsf{ts}(T) = lin(T)$. One can now show that for every $\sigma \in \Sigma^\infty$, $\mathsf{ts}(\mathsf{st}(\sigma)) = [\sigma]$ and for every $T \in \mathbb{TR}(\Sigma, I)$, $\mathsf{st}(\mathsf{ts}(T)) = T$. This justifies our claim that $\Sigma^\infty/\approx_I$ and $\mathbb{TR}(\Sigma, I)$ are indeed two equivalent ways of talking about the same class of objects.

We note that every trace consistent subset L of Σ^∞ defines a trace language L_{Tr} given by $L_{Tr} = \{\mathsf{st}(\sigma) \mid \sigma \in L\}$ which has the property $\mathsf{ts}(L_{Tr}) = L$. In this sense every product language defines a trace language. We say that a trace language L_{Tr} is *regular* iff $\mathsf{ts}(L_{Tr})$ is a regular subset of Σ^∞. As we will see later not every (regular) trace language is a (regular) product language. Hence in order to recognize regular trace languages one will have to use strengthened versions of product automata. Such automata called asynchronous automata were formulated by Zielonka for recognizing regular languages of finite traces. These were then generalized for handling infinite traces by Gastin and Petit [43]. We will use a combination of these two types of automata for solving the satisfiability and model checking problems for the trace-based temporal logic called TrPTL to be considered in the next section.

Let $\widetilde{\Sigma}$ be a distributed alphabet with $\mathcal{P}$ as the associated set of agents. In an asynchronous automaton, each process $p \in \mathcal{P}$ is equipped with a finite non-empty set of local p-states, denoted S_p. It will be convenient to develop some notations for talking about "more global" states before defining these automata.

First we set $S = \bigcup_{p \in \mathcal{P}} S_p$ and call S the set of *local states*. We let P, Q range over non-empty subsets of $\mathcal{P}$ and let p, q range over $\mathcal{P}$. A Q-state is a map $s : Q \to S$ such that $s(q) \in S_q$ for every $q \in Q$. We let S_Q denote the set Q-states. We call $S_{\mathcal{P}}$ the set of *global states*.

We use a to abbreviate $\mathrm{loc}(a)$ when talking about states (recall that $\mathrm{loc}(a) = \{\ p \mid a \in \Sigma_p\ \}$). Thus an a-state is just a $\mathrm{loc}(a)$-state and S_a denotes the set of all $\mathrm{loc}(a)$-states.

A *distributed transition system* TS over $\widetilde{\Sigma}$ is a structure

$$(\{S_p\}, \{\longrightarrow_a\}, S_{in}),$$

where

- S_p is a finite non-empty set of p-states for each process p.
- For $a \in \Sigma$, $\longrightarrow_a \subseteq S_a \times S_a$ is a transition relation between a-states.
- $S_{in} \subseteq S_{\mathcal{P}}$ is a set of initial global states.

The idea is that an a-move by TS involves only the local states of the agents which participate in the execution a. This is reflected in the global transition relation $\longrightarrow_{TS} \subseteq S_{\mathcal{P}} \times \Sigma \times S_{\mathcal{P}}$ which is defined as follows: Suppose s and s' are two global states and s_a and s'_a are the two corresponding a-states. In other words, $s_a(i) = s(i)$ and $s'_a(i) = s'(i)$ for each i in $\mathrm{loc}(a)$. Then

$$s \xrightarrow{a}_{TS} s' \text{ iff } (s_a, s'_a) \in \longrightarrow_a \text{ and } s(j) = s'(j) \text{ for every } j \notin \mathrm{loc}(a).$$

From the definition of $\longrightarrow_{TS}$, it is clear that actions which are executed by disjoint sets of agents are processed independently by TS.

An *asynchronous automaton* over $\widetilde{\Sigma}$ is then a distributed transition system equipped with a set of global accepting states. More precisely, it is a structure $\mathcal{A} = (\{S_p\}, \{\longrightarrow_a\}, S_{in}, F)$ where

- $F \subseteq S_{\mathcal{P}}$ is a set of accepting global states.

A *trace run* of $\mathcal{A}$ over the finite trace $T = (E, \leq, \lambda)$ is a map $\rho : \mathcal{C}_T \to S_{\mathcal{P}}$ such that $\rho(\emptyset) \in S_{in}$ and for every $(c, a, c') \in \longrightarrow_T$, $\rho(c) \xrightarrow{a}_{TS} \rho(c')$. We say that ρ is an *accepting run* whenever $\rho(E) \in F$. The *language of finite traces accepted by* $\mathcal{A}$ is given by

$$\mathcal{L}_{Tr}(\mathcal{A}) = \{\ T \in \mathbb{TR}^* \mid \exists \text{ an accepting run of } \mathcal{A} \text{ over } T\ \}.$$

In the present setting Zielonka's fundamental result can now be formulated as

Theorem 8.6.1 ([157]) *$L \subseteq \mathbb{TR}^*(\Sigma, I)$ is regular iff $L = L_{Tr}(\mathcal{A})$ for some asynchronous automaton $\mathcal{A}$ over some $\widetilde{\Sigma}$ where $\widetilde{\Sigma}$ is a distributed alphabet whose induced trace alphabet is (Σ, I). Further, one may assume $\mathcal{A}$ to be deterministic and one may assume $\widetilde{\Sigma}$ to be the distributed alphabet induced by the maximal D-cliques of (Σ, I).*

This result has been generalized to the set of ω-regular trace languages by Gastin and Petit [43] in terms of asynchronous automata with Büchi acceptance conditions. Since we will treat both finite and infinite traces on an equal footing we will present a class of automata capable of accepting both finite and infinite traces. Hence our automata are essentially distributed transition systems augmented with *both* finite and infinite accepting states.

An *asynchronous Büchi automaton* over $\widetilde{\Sigma}$ is a structure

$$\mathcal{A} = (\{S_p\}, \{\longrightarrow_a\}, S_{in}, \{(F_p, F_p^{\omega})\}),$$

where:

- $(\{S_p\}, \{\longrightarrow_a\}, S_{in})$ is a distributed transition system.
- $F_p \subseteq S_p$ is a set of local finitary accepting states of process p.
- $F_p^\omega \subseteq S_p$ is a set of local infinitary accepting states of process p.

For convenience we will from now on denote this class of automata just "asynchronous automata".

To define acceptance we must now compute $\mathrm{Inf}_p(\rho)$, the set of p-states that are encountered infinitely often along ρ. When incorporating both finite and infinite behaviour in this richer domain we have to take care in defining the set of infinitely occuring states of process p. The obvious definition, namely $\mathrm{Inf}_p(\rho) = \{s_p \mid \rho(c)(p) = s_p \text{ for infinitely many } c \in \mathcal{C}_T\}$, will not work. The complication arises because some processes may make only finitely many moves, even though the overall trace consists of an infinite number of events.

For instance, consider the distributed alphabet $\widetilde{\Sigma}_0 = \{\{a\}, \{b\}\}$. In the corresponding distributed transition system, there are two processes p and q which execute a's and b's completely independently. Consider the trace $T = (E, \leq, \lambda)$ where $|E_p| = 1$ and E_q is infinite — i.e., all the infinite words in $\mathsf{ts}(T)$ contain one a and infinitely many b's. Let s_p be the state of p after executing a. Then, there will be infinitely many configurations whose p-state is s_p, even though p only moves a finite number of times.

Continuing with the same example, consider another infinite trace $T' = (E', \leq', \lambda')$ over the same alphabet where both E_p and E_q are infinite. Once again, let s_p be the local state of p after reading one a. Further, let us suppose that after reading the second a, p never returns to the state s_p. It will still be the case that there are infinitely many configurations whose p-state is s_p: consider the configurations $c_0, c_1, c_2, \ldots$ where c_j is the finite configuration after one a and j b's have occurred.

So, we have to define $\mathrm{Inf}_p(\rho)$ so as to detect whether or not process p is making progress. The appropriate formulation is as follows:

- *E_p is finite:* $\mathrm{Inf}_p(\rho) = \{s_p\}$, where $\rho(\downarrow E_p) = s$ and $s_p = s(p)$.
- *E_p is an infinite set:* $\mathrm{Inf}_p(\rho) = \{s_p \mid \text{for infinitely many } e \in E_p, s_e(p) = s_p, \text{ where } \rho(\downarrow e) = s_e\}$.

A *trace run* of an asynchronous automaton over the (possibly infinite) trace $T = (E, \leq, \lambda) \in \mathbb{TR}$ is now defined in the obvious way. A run ρ of $\mathcal{A}$ over the (possibly infinite) trace $T = (E, \leq, \lambda)$ is *accepting* iff for each process p the following conditions are met:

- If E_p is *finite* then $\mathrm{Inf}_p(\rho) \cap F_p \neq \emptyset$.
- If E_p is *infinite* then $\mathrm{Inf}_p(\rho) \cap F_p^\omega \neq \emptyset$.

We then have the following characterization extending Theorem 8.6.1.

Theorem 8.6.2 *A trace language $L \subseteq \mathbb{TR}(\Sigma, I)$ is regular iff $L = L_{Tr}(\mathcal{A})$ for an asynchronous automaton over $\widetilde{\Sigma}$ where $\widetilde{\Sigma}$ is a distributed alphabet whose induced trace alphabet is (Σ, I).*

It should be noted however that deterministic automata no longer suffice for accepting *all* regular languages.

We say that $\mathcal{A}$ is *in standard form* if

- For each p, $F_p \cap F_p^\omega = \emptyset$.
- For each $(s_a, t_a) \in \longrightarrow_a$ and $p \in \mathrm{loc}(a)$ we have that $s_a(p) \notin F_p$.

Thus, $\mathcal{A}$ is in standard form if the p-states in F_p are all "dead" and disjoint from F_p^ω. It is easy to convert every asynchronous automaton into standard form. All our asynchronous automata will be in standard form.

We conclude with a result concerning the emptiness problem for asynchronous automata.

Proposition 8.6.3 ([97]) *Let $\mathcal{A}$ be an asynchronous automaton in standard form. The emptiness problem is decidable in time $O(n^{2|\mathcal{P}|})$, where n is the largest of the local state spaces, S_p.*

We have described here the languages defined by asynchronous automata in terms of traces. We note that these automata can be viewed — and this is the conventional approach — as automata running over Σ-sequences. Using the global transition relations of these automata one can easily define the string languages accepted by these automata. These languages will be naturally trace consistent w.r.t. the trace alphabets induced by the associated distributed alphabets. The resulting trace languages will be precisely the trace languages accepted by these automata according to the definitions we have provided here.

8.7 TrPTL

We present here the linear time temporal logic over traces called TrPTL. This is the first such logic patterned after PTL (i.e. LTL) formulated for traces. For a detailed treatment of this logic the reader is referred to [135, 134].

As before, it will be notationally convenient to deal with distributed alphabets in which the names of the processes are positive integers. Through this section and the next, we fix a distributed alphabet $\widetilde{\Sigma} = \{\Sigma_i\}_{i \in \mathcal{P}}$ with $\mathcal{P} = \{1, 2, \ldots, K\}$ and $K \geq 1$. We let i, j and k range over $\mathcal{P}$. As before, let P, Q range over non-empty subsets of $\mathcal{P}$. The trace alphabet induced by $\widetilde{\Sigma}$ is denoted (Σ, I). We assume the terminology and notations developed in the previous sections. In particular, when dealing with a $\mathcal{P}$-indexed family $\{X_i\}_{i \in \mathcal{P}}$ we will often write just $\{X_i\}$.

The logic TrPTL is parameterized by the class of distributed alphabets. Having fixed $\widetilde{\Sigma}$ we shall often almost always write TrPTL to mean TrPTL($\widetilde{\Sigma}$), the logic associated with $\widetilde{\Sigma}$. In order to better illustrate the main features of

the logic we will first include atomic propositions. They will be dropped once we return to considering the technical aspects of the logic. We fix a finite non-empty set of atomic propositions P with p, q ranging over P. Then $\Phi_{\mathrm{TrPTL}(\widetilde{\Sigma})}$, the set of formulas of $\mathrm{TrPTL}(\widetilde{\Sigma})$, is defined inductively via:

- For $p \in P$ and $i \in \mathcal{P}$, $p(i)$ is a formula (which is to be read "p at i").
- If α and β are formulas, so are $\neg\alpha$ and $\alpha \vee \beta$.
- If α is a formula and $a \in \Sigma_i$ then $\langle a \rangle_i \alpha$ is a formula.
- If α and β are formulas so is $\alpha \mathcal{U}_i \beta$.

Throughout this section, we denote $\Phi_{\mathrm{TrPTL}(\widetilde{\Sigma})}$ as just Φ. In the semantics of the logic, which will be based on infinite traces, the i-view of a configuration will play a crucial role. Let $T \in \mathbb{TR}^\omega$ with $T = (E, \leq, \lambda)$. Recall that $E_i = \{e \mid e \in E \text{ and } \lambda(e) \in \Sigma_i\}$. Let $c \in \mathcal{C}_T$ and $i \in \mathcal{P}$. Then $\downarrow^i(c)$ is the i-view of c and it is defined as:

$$\downarrow^i(c) = \downarrow(c \cap E_i).$$

We note that $\downarrow^i(c)$ is also a configuration. It is the "best" configuration that the agent i is aware of at c. We say that $\downarrow^i(c)$ is an *i-local configuration*. Let $\mathcal{C}_T^i = \{\downarrow^i(c) \mid c \in \mathcal{C}_T\}$ be the set of i-local configurations. For $Q \subseteq \mathcal{P}$ and $c \in \mathcal{C}_T$, we let $\downarrow^Q(c)$ denote the set $\bigcup\{\downarrow^i(c) \mid i \in Q\}$. Once again, $\downarrow^Q(c)$ is a configuration. It represents the collective knowledge of the processes in Q about the configuration c.

The following basic properties of traces follow directly from the definitions.

Proposition 8.7.1 *Let $T = (E, \leq, \lambda)$ be an infinite trace. The following statements hold.*

(1) Let $\leq_i = \leq \cap (E_i \times E_i)$. Then $(E_i, \leq_i)$ is a linear order isomorphic to ω if E_i is infinite and isomorphic to a finite initial segment of ω if E_i is finite.

(2) $(\mathcal{C}_T^i, \subseteq)$ is a linear order. In fact $(\mathcal{C}_T^i - \{\emptyset\}, \subseteq)$ is isomorphic to $(E_i, \leq_i)$.

(3) Suppose $\downarrow^i(c) \neq \emptyset$ where $c \in \mathcal{C}_T$. Then there exists $e \in E_i$ such that $\downarrow^i(c) = \downarrow e$. In fact e is the $\leq_i$-maximum event in $(c \cap E_i)$.

(4) Suppose $Q \subseteq Q' \subseteq \mathcal{P}$ and $c \in \mathcal{C}_T$. Then $\downarrow^Q(c) = \downarrow^Q(\downarrow^{Q'}(c))$. In particular, for a single process i, $\downarrow^i(c) = \downarrow^i(\downarrow^i(c))$.

We can now present the semantics of TrPTL. A model is a pair $M = (T, \{V_i\}_{i \in p})$ where $T = (E, \leq, \lambda) \in \mathbb{TR}^\omega$ and $V_i : \mathcal{C}_T^i \to 2^P$ is a valuation function which assigns a set of atomic propositions to i-local configurations for each process i. Let $c \in \mathcal{C}_T$ and $\alpha \in \Phi$. Then $M, c \models \alpha$ denotes that α is satisfied at c in M and it is defined inductively as follows:

- $M, c \models p(i)$ for $p \in P$ iff $p \in V_i(\downarrow^i(c))$.

- $M, c \models \neg\alpha$ iff $M, c \not\models \alpha$.
- $M, c \models \alpha \vee \beta$ iff $M, c \models \alpha$ or $M, c \models \beta$.
- $M, c \models \langle a \rangle_i \alpha$ iff there exists $e \in E_i - c$ such that $\lambda(e) = a$ and $M, \downarrow e \models \alpha$. Moreover, for every $e' \in E_i$, $e' < e$ iff $e' \in c$.
- $M, c \models \alpha \mathcal{U}_i \beta$ iff there exists $c' \in \mathcal{C}_T$ such that $c \subseteq c'$ and $M, \downarrow^i(c') \models \beta$. Moreover, for every $c'' \in \mathcal{C}_T$, if $\downarrow^i(c) \subseteq \downarrow^i(c'') \subset \downarrow^i(c')$ then $M, \downarrow^i(c'') \models \alpha$.

Thus TrPTL is an action based multi-agent version of LTL. Indeed both in terms of its syntax and semantics, LTL(Σ) corresponds to the case where there is only one agent. The semantics of TrPTL when specialized down to this case yields the previous LTL(Σ) semantics.

Returning to TrPTL, the assertion $p(i)$ says that the i-view of c satisfies the atomic proposition p. Observe that we could well have $p(i)$ satisfied at c but not $p(j)$ (with $i \neq j$). It is interesting to note that all atomic assertions (that we know of) concerning distributed behaviours are local in nature. Indeed, it is well-known that global atomic propositions will at once lead to an undecidable logic in the current setting [81, 112].

Suppose $M = (T, \{V_i\})$ is a model and $c \xrightarrow{a}_T c'$ with $j \notin \mathrm{loc}(a)$. Then $M, c \models p(j)$ iff $M, c' \models p(j)$. In this sense the valuation functions are local. There are, of course, a number of equivalent ways of formulating this idea which we will not get into here.

The assertion $\langle a \rangle_i \alpha$ says that the agent i will next participate in an a-event. Moreover, at the resulting i-view, the assertion α will hold. The assertion $\alpha \mathcal{U}_i \beta$ says that there is a future i-view (including the present i-view) at which β will hold and for all the intermediate i-views (if any) starting from the current i-view, the assertion α will hold.

Before considering examples of TrPTL specifications, we will introduce some notation. We let α, β with or without subscripts range over Φ. Abusing notation, we will use loc to denote the map which associates a set of *locations* with each formula.

- $\mathrm{loc}(p(i)) = \mathrm{loc}(\langle a \rangle_i \alpha) = \mathrm{loc}(\alpha \mathcal{U}_i \beta) = \{i\}$.
- $\mathrm{loc}(\neg\alpha) = \mathrm{loc}(\alpha)$.
- $\mathrm{loc}(\alpha \vee \beta) = \mathrm{loc}(\alpha) \cup \mathrm{loc}(\beta)$.

In what follows, $\Phi^i = \{\alpha \mid \mathrm{loc}(\alpha) = \{i\}\}$ is the set of i-type formulas. We note that unlike LTL$^{\otimes}$, a TrPTL formula of the form $\langle a \rangle_i \alpha$ could have $j \in \mathrm{loc}(\alpha)$ with $j \neq i$. A similar remark applies to the indexed until-operators.

A basic observation concerning the semantics of TrPTL can be phrased as follows:

Proposition 8.7.2 *Let $M = (T, \{V_i\})$ be a model, $c \in \mathcal{C}_T$ and α a formula such that $\mathrm{loc}(\alpha) \subseteq Q$. Then $M, c \models \alpha$ iff $M, \downarrow^Q(c) \models \alpha$.*

A corollary to this result is that in case $\alpha \in \Phi^i$ then $M, c \models \alpha$ if and only if $M, \downarrow^i(c) \models \alpha$. As a result, the formulas in Φ^i can be used in exactly the same manner as one would use LTL$^{\otimes}$ to express properties of the agent i. Boolean combinations of such local assertions can be used to capture various interaction patterns between the agents implied by the logical connectives as well as the coordination enforced by the distributed alphabet $\widetilde{\Sigma}$. For writing specifications, apart from the usual derived connectives that we already introduced in Section 8.2 for LTL, the following operators are also available:

- $tt \stackrel{\text{def}}{=} p_1(1) \vee \neg p_1(1)$ denotes the constant "True", where $P = \{p_1, p_2, \ldots\}$. We use $ff = \neg tt$ to denote "False".
- $\Diamond_i \alpha \stackrel{\text{def}}{=} tt \mathcal{U}_i \alpha$ is a local version of the $\Diamond$ modality of LTL.
- $\Box_i \alpha \stackrel{\text{def}}{=} \neg \Diamond_i \neg \alpha$ is a local version of the $\Box$ modality of LTL.
- Let $X \subseteq \Sigma_i$ and $\overline{X} = \Sigma_i - X$. Then $\alpha \mathcal{U}_i^X \beta \stackrel{\text{def}}{=} (\alpha \wedge \bigwedge_{a \in \overline{X}} [a]_i ff) \mathcal{U}_i \beta$. In other words $\alpha \mathcal{U}_i^X \beta$ is fulfilled using (at most) actions taken from X. We set $\Diamond_i^X \alpha \stackrel{\text{def}}{=} tt \mathcal{U}_i^X \alpha$ and $\Box_i^X \alpha \stackrel{\text{def}}{=} \neg \Diamond_i^X \neg \alpha$.
- $\alpha(i) \stackrel{\text{def}}{=} \alpha \mathcal{U}_i \alpha$ (or equivalently $ff \mathcal{U}_i \alpha$). $\alpha(i)$ is to be read as "α at i". If $M = (T, \{V_i\})$ is a model and $c \in \mathcal{C}_T$ then $M, c \models \alpha(i)$ iff $M, \downarrow^i(c) \models \alpha$. It could of course be the case that $\mathrm{loc}(\alpha) \neq \{i\}$.

A simple but important observation is that every formula is a boolean combination of formulas taken from $\bigcup_{i \in \mathcal{P}} \Phi^i$. In TrPTL we can say that a specific global configuration is reachable from the initial configuration. Let $\{\alpha_i\}_{i \in \mathcal{P}}$ be a family with $\alpha_i \in \Phi^i$ for each i. Then we can define a derived connective $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ which has the following semantics at the empty configuration. Let $M = (T, \{V_i\})$ be a model. Then $M, \emptyset \models \Diamond(\alpha_1, \alpha_2, \ldots, \alpha_k)$ iff there exists $c \in \mathcal{C}_T$ such that $M, c \models \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_K$.

To define this derived connective set $\Sigma_1' = \Sigma_1$ and, for $1 < i \leq K$, set $\Sigma_i' = \Sigma_i - \cup\{\Sigma_j \mid 1 \leq j < i\}$. Then $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ is the formula:

$$\Diamond_1^{\Sigma_1'}(\alpha_1 \wedge \Diamond_2^{\Sigma_2'}(\alpha_2 \wedge \Diamond_3^{\Sigma_3'}(\alpha_3 \wedge \cdots \Diamond_K^{\Sigma_K'} \alpha_K)) \cdots).$$

The idea is that the sequence of actions leading up to the required configuration can be reordered so that one first performs all the actions in Σ_1, then all the actions in $\Sigma_2 - \Sigma_1$ etc. Hence, if now is an atomic proposition, the formula $\Diamond(\mathsf{now}(1), \mathsf{now}(2), \ldots, \mathsf{now}(K))$ is satisfied at the empty configuration iff there is a reachable configuration at which all the agents assert now.

Dually, safety properties that hold at the initial configuration can also be expressed. For example, let crt_i be the atomic assertion declaring that the agent i is currently in its critical section. Then it is possible to write a formula φ_{ME} which asserts that at all reachable configurations at most one agent is in its critical section, thereby guaranteeing that the system satisfies the mutual exclusion property. We omit the details of how to specify φ_{ME}.

On the other hand, it seems difficult to express nested global and safety properties in TrPTL. It is also the case that due to the local nature of the modalities, information about the past sneaks into the semantics even though there are no explicit past operators in the logic.

A formula α is said to be *root-satisfiable* iff there exists a model M such that $M, \emptyset \models \alpha$. On the other hand, α is said to be *satisfiable* iff there exists a model $M = (T, \{V_i\})$ and $c \in \mathcal{C}_T$ such that $M, c \models \alpha$. It turns out that these two notions are *not* equivalent. Consider the distributed alphabet $\widetilde{\Sigma}_0 = \{\Sigma_1, \Sigma_2\}$ with $\Sigma_1 = \{a, d\}$ and $\Sigma_2 = \{b, d\}$. Then it is not difficult to verify that the formula $p(2)(1) \wedge \Box_2 \neg p(2)$ is satisfiable but not root-satisfiable. (Recall that $p(2)(1)$ abbreviates $ff\mathcal{U}_1 p(2)$). One can however transform every formula α into a formula α' such that α is satisfiable iff α' is root satisfiable.

This follows from the observation that every α can be expressed as a boolean combination of formulas taken from the set $\bigcup_{i \in \mathcal{P}} \Phi^i$. Hence the given formula α can be assumed to be of the form $\alpha = \bigvee_{j=1}^{m} (\alpha_{j1} \wedge \alpha_{j2} \wedge \cdots \wedge \alpha_{jK})$ where $\alpha_{ji} \in \Phi^i$ for each $j \in \{1, 2, \ldots, m\}$ and each $i \in \mathcal{P}$. Now convert α to the formula α' where $\alpha' = \bigvee_{j=1}^{m} \Diamond(\alpha_{j1}, \alpha_{j2}, \cdots, \alpha_{jK})$. (Recall the derived modality $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ introduced earlier.) From the semantics of $\Diamond(\alpha_1, \alpha_2, \ldots, \alpha_K)$ it follows that α is satisfiable iff α' is root-satisfiable.

Hence, in principle, it suffices to consider only root-satisfiability in developing a decision procedure for TrPTL. There is of course a blow-up involved in converting satisfiable formulas to root-satisfiable formulas. If one wants to avoid this blow-up then the decision procedure for checking root-satisfiability can be suitably modified to yield a direct decision procedure for checking satisfiability as done in [135]. In any case, it is root satisfiability which is of importance from the standpoint of model checking. Hence here we shall only develop a procedure for deciding if a given formula of TrPTL is root-satisfiable.

As a first step we augment the syntax of our logic by one more construct.

- If α is a formula, so is $O_i\alpha$. In the model $M = (T, \{V_i\})$, at the configuration $c \in \mathcal{C}_T$, $M, c \models O_i\alpha$ iff $M, c \models \langle a \rangle_i \alpha$ for some $a \in \Sigma_i$. We also define $\mathrm{loc}(O_i\alpha) = \{i\}$.

Secondly, we will from now on drop the atomic propositions and instead work with the constant tt and its negation ff as done earlier. The semantic definitions are assumed to be suitably modified.

Thus $O_i\alpha \equiv \bigvee_{a \in \Sigma_i} \langle a \rangle_i \alpha$ is a valid formula and O_i is expressible in the former syntax. It will be however more efficient to admit O_i as a first class modality as we did in Section 8.2.

Fix a formula α_0. Our aim is to effectively associate an asynchronous automaton $\mathcal{A}_{\alpha_0}$ with α_0 such that α_0 is root-satisfiable iff $L_{Tr}(\mathcal{A}_{\alpha_0}) \neq \emptyset$. Since the emptiness problem for asynchronous automata is decidable (Proposition 8.6.3), this will yield the desired decision procedure. Let $cl(\alpha_0)$ be the least set of formulas containing α_0 which satisfies:

- $\neg\alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.

- $\alpha \vee \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$.
- $\langle a \rangle_i \alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.
- $O_i \alpha \in cl(\alpha_0)$ implies $\alpha \in cl(\alpha_0)$.
- $\alpha \mathcal{U}_i \beta \in cl(\alpha_0)$ implies $\alpha, \beta \in cl(\alpha_0)$. In addition, $O_i(\alpha \mathcal{U}_i \beta) \in cl(\alpha_0)$.

We then define $CL(\alpha_0)$ to be the set $cl(\alpha_0) \cup \{\neg\beta \mid \beta \in cl(\alpha_0)\}$.

Thus $CL(\alpha_0)$, sometimes called the Fisher-Ladner closure of α_0, is closed under negation with the convention that $\neg\neg\beta$ is identified with β. Moreover, throughout the remainder of the section all formulas that we encounter will be assumed to be members of $CL(\alpha_0)$. From now we shall write CL instead of $CL(\alpha_0)$.

$A \subseteq CL$ is called an *i-type atom* iff it satisfies:

- $tt \in A$.
- $\alpha \in A$ iff $\neg\alpha \notin A$.
- $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- $\alpha \mathcal{U}_i \beta \in A$ iff $\beta \in A$ or ($\alpha \in A$ and $O_i(\alpha \mathcal{U}_i \beta) \in A$).
- If $\langle a \rangle_i \alpha$, $\langle b \rangle_i \beta \in A_i$ then $a = b$.

AT_i denotes the set of i-type atoms. We now need to define the notion of a formula in CL being a member of a collection of atoms. Let $\alpha \in CL$ and $\{A_i\}_{i \in Q}$ be a family of atoms with $\mathrm{loc}(\alpha) \subseteq Q$ and $A_i \in AT_i$ for each $i \in Q$. We'll define the predicate Member(α,$\{A_i\}_{i \in Q}$), which for convenience will be denoted by $\alpha \in \{A_i\}_{i \in Q}$. It is defined inductively as:

- If $\mathrm{loc}(\alpha) = \{j\}$ then $\alpha \in \{A_i\}_{i \in Q}$ iff $\alpha \in A_j$.
- If $\alpha = \neg\beta$ then $\alpha \in \{A_i\}_{i \in Q}$ iff $\beta \notin \{A_i\}_{i \in Q}$.
- If $\alpha = \alpha_1 \vee \alpha_2$ then $\alpha_1 \vee \alpha_2 \in \{A_i\}_{i \in Q}$ iff $\alpha_1 \in \{A_i\}_{i \in Q}$ or $\alpha_2 \in \{A_i\}_{i \in Q}$.

The construction of the asynchronous automaton $\mathcal{A}_{\alpha_0}$ is guided by the construction developed for LTL in Section 8.2. However in the much richer setting of traces it turns out that one must make crucial use of the latest information that the agents have about each other when defining the transitions of $\mathcal{A}_{\alpha_0}$. It has been shown by Mukund and Sohoni [96] that this information can be kept track of by a deterministic asynchronous automaton whose size depends only on $\widetilde{\Sigma}$. (Actually the automaton described in [96] operates over finite traces but it is a trivial task to convert it into an asynchronous automaton having the desired properties). To bring out the relevant properties of this automaton, let $T \in \mathbb{TR}^\omega$ with $T = (E, \leq, \lambda)$. For each subset Q of processes, the function $\mathsf{latest}_{T,Q} : \mathcal{C}_T \times \mathcal{P} \to Q$ is given by $\mathsf{latest}_{T,Q}(c, j) = \ell$ iff ℓ is the least member of Q (under the usual ordering over the integers) with the property $\downarrow^j(\downarrow^q(c)) \subseteq \downarrow^j(\downarrow^\ell(c))$ for every $q \in Q$. In other words, among the agents in Q, ℓ has the best information about j at c, with ties being broken by the usual ordering over integers.

Theorem 8.7.3 ([96]) *There exists an effectively constructible deterministic asynchronous automaton* $\mathcal{A}_\Gamma = (\{\Gamma_i\}, \{\Longrightarrow_a\}, \Gamma_{in}, \{(F_i, F_i^\omega)\})$ *such that:*

(1) $L_{Tr}(\mathcal{A}_\Gamma) = \mathbb{TR}^\omega$.

(2) For each $Q = \{i_1, i_2, \ldots, i_n\}$, *there exists an effectively computable function* $\mathsf{gossip}_Q : \Gamma_{i_1} \times \Gamma_{i_2} \times \cdots \times \Gamma_{i_n} \times \mathcal{P} \to Q$ *such that for every* $T \in \mathbb{TR}^\omega$, *every* $c \in \mathcal{C}_T$ *and every* $j \in \mathcal{P}$, $\mathsf{latest}_{T,Q}(c, j) = \mathsf{gossip}_Q(\gamma(i_1), \ldots, \gamma(i_n), j)$ *where* $\rho_T(c) = \gamma$ *and* ρ_T *is the unique (accepting) run of* $\mathcal{A}_\Gamma$ *over* T.

Henceforth, we refer to $\mathcal{A}_\Gamma$ as the *gossip automaton*. Each process in the gossip automaton has $2^{O(K^2 \log K)}$ local states, where $K = |\mathcal{P}|$. Moreover the function gossip_Q can be computed in time which is polynomial in the size of K.

Each i-state of the automaton $\mathcal{A}_{\alpha_0}$ will consist of an i-type atom together with an appropriate i-state of the gossip automaton. Two additional components will be used to check for liveness requirements. One component will take values from the set $N_i = \{0, 1, 2, \ldots, |U_i|\}$ where $U_i = \{\alpha \mathcal{U}_i \beta \mid \alpha \mathcal{U}_i \beta \in CL\}$. This component will be used to ensure that all "until" requirements are met. The other component will take values from the set $\{\mathsf{on}, \mathsf{off}\}$. This will be used to detect when an agent has quit.

The automaton $\mathcal{A}_{\alpha_0}$ can now be defined as:

$$\mathcal{A}_{\alpha_0} = (\{S_i\}, \{\longrightarrow_a\}, S_{in}, \{(F_i, F_i^\omega)\}),$$

where:

- For each i, $S_i = AT_i \times \Gamma_i \times N_i \times \{\mathsf{on}, \mathsf{off}\}$. Recall that Γ_i is the set of i-states of the gossip automaton and $N_i = \{0, 1, 2, \ldots, |U_i|\}$ with $U_i = \{\alpha \mathcal{U}_i \beta \mid \alpha \mathcal{U}_i \beta \in CL\}$.
- Let $s_a, s'_a \in S_a$ with $s_a(i) = (A_i, \gamma_i, n_i, v_i)$ and $s'_a(i) = (A'_i, \gamma'_i, n'_i, v'_i)$ for each $i \in \mathrm{loc}(a)$. Then $(s_a, s'_a) \in \longrightarrow_a$ iff the following conditions are met.
  - $(\gamma_a, \gamma'_a) \in \Longrightarrow_a$ (recall that $\{\Longrightarrow_a\}$ is the family of transition relations of the gossip automaton) where $\gamma_a, \gamma'_a \in \Gamma_a$ such that $\gamma_a(i) = \gamma_i$ and $\gamma'_a(i) = \gamma'_i$ for each $i \in \mathrm{loc}(a)$.
  - $\forall i, j \in \mathrm{loc}(a)$, $A'_i = A'_j$.
  - $\forall i \in \mathrm{loc}(a)$ $\forall \langle a \rangle_i \alpha \in CL$. $\langle a \rangle_i \alpha \in A_i$ iff $\alpha \in A'_i$.
  - $\forall i \in \mathrm{loc}(a)$ $\forall O_i \alpha \in CL$. $O_i \alpha \in A$ iff $\alpha \in A'_i$.
  - $\forall i \in \mathrm{loc}(a) \forall \langle b \rangle_i \beta \in CL$. If $\langle b \rangle_i \beta \in A_i$ then $b = a$.
  - Suppose $j \notin \mathrm{loc}(a)$ and $\beta \in CL$ with $\mathrm{loc}(\beta) = \{j\}$. Further suppose that $\mathrm{loc}(a) = \{i_1, i_2, \ldots, i_n\}$. Then $\beta \in A'_i$ iff $\beta \in A_\ell$ where $\ell = \mathsf{gossip}_{\mathrm{loc}(a)}(\gamma_{i_1}, \gamma_{i_2}, \ldots, \gamma_{i_n}, j)$.
  - Let $i \in \mathrm{loc}(a)$, $U_i = \{\alpha_1 \mathcal{U}_i \beta_1, \alpha_2 \mathcal{U}_i \beta_2, \ldots, \alpha_{n_i} \mathcal{U}_i \beta_{n_i}\}$. Then u'_i and u_i are related to each other via:

$$u'_i = \begin{cases} (u_i{+}1) \bmod (n_i{+}1), & \text{if } u_i = 0 \text{ or } \beta_{u_i} \in A_i \text{ or } \alpha_{u_i} \mathcal{U}_i \beta_{u_i} \notin A_i \\ u_i, & \text{otherwise} \end{cases}$$

  - For each $i \in \mathrm{loc}(a)$, $v_i = \mathsf{on}$. Moreover, if $v'_i = \mathsf{off}$ then $\langle a \rangle_i \alpha \notin A'_i$ for every $i \in \mathrm{loc}(a)$ and every $\langle a \rangle_i \alpha \in CL$.

- Let $s \in S_{\mathcal{P}}$ with $s(i) = (A_i, \gamma_i, u_i, v_i)$ for every i. Then $s \in S_{in}$ iff $\alpha_0 \in \{A_i\}_{i \in \mathcal{P}}$ and $\gamma \in \Gamma_{in}$ where $\gamma \in \Gamma_{\mathcal{P}}$ satisfies $\gamma(i) = \gamma_i$ for every i. Furthermore, $u_i = 0$ for every i. Finally, for every i, $v_i = \mathsf{off}$ implies that $\langle a \rangle_i \alpha \notin A_i$ for every $\langle a \rangle_i \alpha \in CL$.

- For each i, $F_i^{\omega} \subseteq S_i$ is given by $F_i^{\omega} = \{(A_i, \gamma_i, u_i, v_i) \mid u_i = 0 \text{ and } v_i = \mathsf{on}\}$ and $F_i \subseteq S_i$ is given by $F_i = \{(A_i, \gamma_i, u_i, v_i) \mid v_i = \mathsf{off}\}$.

This construction is an optimized version of the original construction for TrPTL presented in [135, 134]. Note that $\mathcal{A}_{\alpha_0}$ is indeed in standard form. Arguments similar to those presented in [135, 134] lead to the next set of results.

Theorem 8.7.4

(1) α_0 is root-satisfiable iff $L_{Tr}(\mathcal{A}_{\alpha_0}) \neq \emptyset$.

(2) The number of local states of $\mathcal{A}_{\alpha_0}$ is bounded by $2^{O(\max(n, m^2 \log m))}$ where $n = |\alpha_0|$ and m is the number of agents mentioned in α_0. Clearly, $m \leq n$. It follows that the root-satisfiability problem (and in fact the satisfiability problem) for TrPTL *is solvable in time $2^{O(\max(n, m^2 \log m) \cdot m)}$.*

The number of local states of each process in $\mathcal{A}_{\alpha_0}$ is determined by two quantities: the length of α_0 and the size of the gossip automaton $\mathcal{A}_{\Gamma}$. As far as the size of $\mathcal{A}_{\Gamma}$ is concerned, it is easy to verify that we need to consider only those agents in $\mathcal{P}$ that are mentioned in $\mathrm{loc}(\alpha_0)$, rather than all agents in the system.

The *model checking problem* for TrPTL can be phrased as follows. A finite state distributed program Pr over $\widetilde{\Sigma}$ is an asynchronous automaton $\mathcal{A}_{Pr} = (\{S_i^{Pr}\}, \{\Longrightarrow_a^{Pr}\}, S_{in}^{Pr}, \{(S_i^{Pr}, S_i^{Pr})\})$ modeling the state space of Pr.

Viewing a formula α_0 as a specification, we say that Pr *meets the specification* α_0 — denoted $Pr \models \alpha_0$ — if for every $T \in \mathbb{TR}^{\omega}$, if $\mathcal{A}_{Pr}$ has a run over T then $T, \emptyset \models \alpha_0$.

The model checking problem for TrPTL can be solved by "intersecting" the program automaton $\mathcal{A}_{Pr}$ with the formula automaton $\mathcal{A}_{\neg\alpha_0}$ to yield an automaton $\mathcal{A}$ such that $L_{Tr}(\mathcal{A}) = L_{Tr}(\mathcal{A}_{Pr}) \cap L_{Tr}(\mathcal{A}_{\neg\alpha_0})$. As before, $L_{Tr}(\mathcal{A}) = \emptyset$ iff $Pr \models \alpha_0$.

It turns out that this model checking problem has time complexity $O(|\mathcal{A}_{Pr}| \cdot 2^{O(\max(n, m^2 \log m) \cdot m)})$ where $|\mathcal{A}_{Pr}|$ is the size of the global state space of the asynchronous automaton modeling the behaviour of the given program Pr and, as before, $n = |\alpha_0|$ and m is the number of agents mentioned in α_0, where α_0 is the specification formula.

We now take a brief look at some related agent-based linear time temporal logics over traces. The first one is the sublogic of TrPTL denoted which consists of the so called connected formulas of TrPTL. We define $\Phi^{\mathrm{con}}_{\mathrm{TrPTL}}$ (from now on written as Φ^{con}) to be the least subset of Φ satisfying the following conditions:

- $tt \in \Phi^{\mathrm{con}}$ and as before $\mathrm{loc}(tt) = \emptyset$
- If $\alpha, \beta \in \Phi^{\mathrm{con}}$, so are $\neg\alpha$ and $\alpha \vee \beta$.
- If $\alpha \in \Phi^{\mathrm{con}}$ and $a \in \Sigma_i$ such that $\mathrm{loc}(\alpha) \subseteq \mathrm{loc}(a)$ then $\langle a \rangle_i \alpha \in \Phi^{\mathrm{con}}$.
- If $\alpha, \beta \in \Phi^{\mathrm{con}}$ with $\mathrm{loc}(\alpha), \mathrm{loc}(\beta) \subseteq \{i\}$ then $\alpha \mathcal{U}_i \beta \in \Phi^{\mathrm{con}}$. Actually one needs only to demand that $\mathrm{loc}(\alpha)$, $\mathrm{loc}(\beta) \subseteq \bigcap\{\mathrm{loc}(a) \mid a \in \Sigma_i\}$ but this leads to notational complications that we wish to avoid here.
- If $\alpha \in \Phi^{\mathrm{con}}$ and $\mathrm{loc}(\alpha) \subseteq \{i\}$ then $O_i\alpha \in \Phi^{\mathrm{con}}$. (Once again one needs just to demand that $\mathrm{loc}(\alpha) \subseteq \bigcap\{\mathrm{loc}(a) \mid a \in \Sigma_i\}$.)

Connected formulas were first identified by Niebert and used by Huhn [65]. They have also been independently identified by Ramanujam [115]. Thanks to the syntactic restrictions imposed on the next state and until formulas, past information is not allowed to creep in. Indeed one can prove the following:

Proposition 8.7.5 *Let $\alpha \in \Phi^{\mathrm{con}}$. Then α is satisfiable iff α is root-satisfiable.*

Yet another pleasing feature of TrPTL$^{\mathrm{con}}$ is that the gossip automaton can be eliminated in the construction of the automaton $\mathcal{A}_{\alpha_0}$ whenever $\alpha_0 \in \Phi^{\mathrm{con}}$. In fact one can prove the following.

Theorem 8.7.6 *The satisfiability problem for* TrPTL$^{\mathrm{con}}$ *is solvable in time* $2^{O(|\alpha_0|)}$.

Once again, a suitably modified statement can be made about the associated model checking problem. At present we do not know whether or not TrPTL is strictly more expressive than TrPTL$^{\mathrm{con}}$, but it is clear that LTL$^{\otimes}$ is a strict sublogic of TrPTL$^{\mathrm{con}}$. We shall deal with the relative strengths of these logics in the next section. Two of the four logics considered by Ramanujam [115] in a closely related setting turn out to be LTL$^{\otimes}$ and TrPTL$^{\mathrm{con}}$. We conjecture that the other two logics are also expressible within TrPTL.

Katz and Peled introduced the logic ISTL [71] whose semantics has a trace-theoretic flavour. In a subsequent paper by Peled and Pnueli [109] on ISTL, the connection to traces was made more directly. Indeed this is one of the first instances of the explicit use of traces in a temporal logical setting that we know of. However, it has branching time modalities which permit quantification over the so called observations of a trace. ISTL uses global atomic propositions rather than local atomic propositions. Penczek has also studied a number of temporal logics (including a version of ISTL) with branching time modalities and global atomic propositions [112]. His logics are interpreted directly over the space of configurations of a trace resulting in a variety of axiomatizations and undecidability results. We feel that local atomic propositions (as used in TrPTL) are crucial for obtaining tractable partial order based temporal logics. Niebert has considered several μ-calculus versions of TrPTL [101, 102] and has obtained various decidability results using a variant of asynchronous Büchi automata.

The temporal logic of causality (TLC) proposed by Alur, Peled and Penczek is basically a temporal logic over traces [4]. The concurrent structures used in [4] as frames for TLC can be easily represented as traces over an appropriately chosen trace alphabet. The interesting feature of TLC is that its branching time modalities are interpreted over causal paths. In a trace $(E, \leq, \lambda)$, the sequence $e_0 e_1 \cdots \in E^\infty$ is a causal path if $e_0 \lessdot e_1 \lessdot e_2 \cdots$. This logic admits an essentially exponential time decision procedure for checking satisfiablity in terms of a variant of Büchi automata called Street automata.

8.8 Expressiveness Issues

Our aim here is to discuss some expressiveness issues concerning trace-based linear time temporal logics. To set the stage we first quickly review the classical case of sequences.

The *monadic second-order theory of infinite sequences over* Σ is denoted MSO(Σ). Its vocabulary consists of a family of unary predicates $\{R_a\}_{a\in\Sigma}$, one for each $a \in \Sigma$; a binary predicate $\leq$; a binary predicate $\in$; a countable supply of individual variables $Var = \{x, y, z, \ldots\}$; a countable supply of set variables (i.e. monadic predicate variables) $SVar = \{X, Y, Z, \ldots\}$. The formulas of MSO($\Sigma$) are then built up by:

- $R_a(x)$, $x \leq y$ and $x \in X$ are atomic formulas.
- If ϕ and ϕ' are formulas then so are $\neg\phi$, $\phi \vee \phi'$, $(\exists x)\phi$ and $(\exists X)\phi$.

A structure for MSO(Σ) is a ω-sequence $\sigma \in \Sigma^\omega$. Let $\mathcal{I}$ be an interpretation of the variables with $\mathcal{I} : Var \longrightarrow \omega$ and $\mathcal{I} : SVar \longrightarrow 2^\omega$. Then the notion of σ being a model of ϕ under the interpretation $\mathcal{I}$, denoted $\sigma \models_{\mathcal{I}} \phi$, is defined in the expected manner. In particular, $\sigma \models_{\mathcal{I}} R_a(x)$ iff $\sigma(\mathcal{I}(x)) = a$ (note that $\sigma \in \Sigma^\omega$ is viewed as $\sigma : \omega \longrightarrow \Sigma$); $\sigma \models_{\mathcal{I}} x \leq y$ iff $\mathcal{I}(x) \leq \mathcal{I}(y)$ (here $\leq$ is the usual ordering over ω); $\sigma \models_{\mathcal{I}} x \in X$ iff $\mathcal{I}(x) \in \mathcal{I}(X)$.

As usual, a sentence is a formula with no free variables. Each sentence ϕ defines an ω-language, denoted L_ϕ, where:

$$L_\phi = \{\sigma \mid \sigma \models \phi\}.$$

We say that $L \subseteq \Sigma^\omega$ is MSO(Σ)-*definable* iff there exists a sentence $\phi \in$ MSO(Σ) such that $L = L_\phi$. A celebrated result of Büchi [14] shows that the class of languages expressible by sentences in MSO(Σ) coincides with the class of languages recognized by Büchi automata over Σ. This class is the *ω-regular languages* over Σ.

The *first-order theory of infinite sequences over* Σ is denoted FO(Σ) and is obtained from MSO(Σ) by abolishing the monadic second-order quantifications from the logic. The semantics and notions of first-order definability are carried over in the obvious manner.

A fundamental result in the theory of temporal logic is Kamp's Theorem [70] which was later strengthened in [41] to establish that LTL(Σ) is expressively

equivalent to the FO(Σ). The surprise here being that LTL(Σ) admits only a bounded number of operators (one unary and one binary as we have formulated it) whereas infinitely many operators of increasing arities can be defined in FO(Σ). Secondly, as we saw in Section 8.2, the satisfiability problem for LTL(Σ) can be solved in deterministic exponential time. The satisfiability problem for FO(Σ) on the other hand, even when the sentences are interpreted over *finite* words, is known to be non-elementary hard [131]. It is quite easy to see that FO(Σ) — and hence LTL(Σ) — is strictly less expressive than MSO(Σ) in the sense that there is a language which is MSO(Σ)-definable but not FO(Σ)-definable. (Indeed this is the sense in which we shall compare the expressive power of various logics in what follows.) For instance, as pointed out by Wolper in a state-based setting [155], the language $L \subseteq \{a, b\}^\omega$ given by "a is executed at every even position" is not definable in this logic. On the other hand, it is easy to come up with a formula of MSO(Σ) defining L.

The expressive power of LTL can be extended to obtain the expressive power of MSO while still guaranteeing an exponential time decidable satisfiability problem as demonstrated first in [156]. Here we sketch how the regular programs over Σ can be used to achieve this goal [60].

The syntax of regular programs over Σ is given by:

$$\mathrm{Prg}(\Sigma) ::= a \mid \pi_0 + \pi_1 \mid \pi_0; \pi_1 \mid \pi^*.$$

With each program we associate a set of finite words via the map $||\cdot|| : \mathrm{Prg}(\Sigma) \longrightarrow 2^{\Sigma^*}$. This map is defined in the standard fashion:

- $||a|| = \{a\}$.
- $||\pi_0 + \pi_1|| = ||\pi_0|| \cup ||\pi_1||$.
- $||\pi_0; \pi_1|| = \{\tau_0\tau_1 \mid \tau_0 \in ||\pi_0|| \text{ and } \tau_1 \in ||\pi_1||\}$.
- $||\pi^*|| = \bigcup_{i\in\omega} ||\pi^i||$, where
  - $||\pi^0|| = \{\varepsilon\}$ and
  - $||\pi^{i+1}|| = \{\tau_0\tau_1 \mid \tau_0 \in ||\pi|| \text{ and } \tau_1 \in ||\pi^i||\}$ for every $i \in \omega$.

The set of formulas of DLTL(Σ) is given by the following syntax.

$$\mathrm{DLTL}(\Sigma) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha\, U^\pi \beta, \quad \pi \in \mathrm{Prg}(\Sigma)$$

A model is a ω-sequence $\sigma \in \Sigma^\omega$. For $\tau \in \mathrm{prf}(\sigma)$ we define $\sigma, \tau \models \alpha$ just as we did for LTL(Σ) in the case of the first three clauses. As for the last one,

- $\sigma, \tau \models \alpha\, U^\pi \beta$ iff there exists $\tau' \in ||\pi||$ such that $\tau\tau' \in \mathrm{prf}(\sigma)$ and $\sigma, \tau\tau' \models \beta$. Moreover, for every τ'' such that $\varepsilon \preceq \tau'' \prec \tau'$, it is the case that $\sigma, \tau\tau'' \models \alpha$.

Thus DLTL(Σ) adds to LTL(Σ) by strengthening the until-operator. To satisfy $\alpha\, U^\pi \beta$, one must satisfy $\alpha\, U \beta$ along some finite stretch of behaviour which

is required to be in the (linear time) behaviour of the program π. We associate with a formula α of DLTL(Σ) the ω-language L_α in the obvious manner.

A useful derived operator of DLTL is:

- $\langle\pi\rangle\alpha \stackrel{\text{def}}{=} tt\,\mathcal{U}^\pi\alpha$.

By replacing the until-modality of DLTL with the above derived operator we obtain the sublogic DLTL$^-(\Sigma)$, which is essentially Propositional Dynamic Logic [39] equipped with a linear time semantics. It turns out that DLTL(Σ) and DLTL$^-(\Sigma)$ both have the same expressive power as MSO(Σ).

Theorem 8.8.1 *Let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent.*

(1) L is ω-regular (i.e. definable in MSO(Σ)*).*

(2) L is DLTL(Σ)*-definable.*

(3) L is DLTL$^-(\Sigma)$*-definable.*

Both the satisfiablity and model checking problems for DLTL(Σ) are decidable with the same time complexity as for LTL(Σ).

Let (Σ, I) be trace alphabet. Then MSO(Σ, I), the *monadic second-order theory of infinite traces (over Σ, I)*, has the same syntax as MSO(Σ). The structures are elements of $\mathbb{TR}^\omega(\Sigma, I)$. Let $T \in \mathbb{TR}^\omega(\Sigma, I)$ with $T = (E, \leq, \lambda)$ and let $\mathcal{I} : X \to E$ be an interpretation. Then $T \models^{\text{MSO}}_{\mathcal{I}} R_a(x)$ iff $\lambda(\mathcal{I}(x)) = a$ and $T \models^{\text{MSO}}_{\mathcal{I}} x \leq y$ iff $\mathcal{I}(x) \leq \mathcal{I}(y)$. Hence, the essential difference is that the binary predicate symbols is now interpreted as the causal partial order of the trace. The remaining semantic definitions go along the expected lines. Each sentence φ (i.e., a formula with no free occurrences of variables) defines the ω-trace language

$$L_\varphi = \{T \mid T \models^{\text{MSO}} \varphi\}.$$

We say that $L \subseteq \mathbb{TR}^\omega$ is MSO-definable iff there exists a sentence φ in MSO(Σ, I) such that $L = L_\varphi$. It is known that MSO-definable languages are precisely the regular trace languages; i.e. those recognized by asynchronous automata [32].

FO(Σ, I), *the first-order theory of traces*, is defined in the obvious way. Clearly it will be strictly weaker than MSO(Σ, I). For more information the reader is referred to [27]. Naturally both these theories can be made to handle finite traces as well.

Through the rest of this section we fix a distributed alphabet $\widetilde{\Sigma}$ and let (Σ, I) be the induced trace alphabet. By MSO($\widetilde{\Sigma}$) we shall mean the theory MSO(Σ, I) and similarly for FO($\widetilde{\Sigma}$), the first-order fragment of MSO($\widetilde{\Sigma}$). In what follows we shall often supress the mention of $\widetilde{\Sigma}$ as well as the induced (Σ, I).

We first consider the logic LTL$^\otimes$. Recall that product languages are trace consistent and hence they induce trace languages via the map st. The resulting trace languages will be called product trace languages. As might be expected, the regular product trace languages are the ones obtained from regular product

languages via the map st. It is easy to show that not every (regular) trace language is a product trace language [136]. It is also easy to see that $\mathrm{LTL}^{\otimes}$-definable trace languages constitute a strict subclass of regular product trace languages. It has been shown that a product version of DLTL denoted $\mathrm{DLTL}^{\otimes}$ captures exactly the class of regular product trace languages [61]. We also claim that it is an easy exercise to formulate a product version of $\mathrm{MSO}(\widetilde{\Sigma})$ and show that it captures exactly the regular product trace languages. Let us denote this product version of $\mathrm{MSO}(\widetilde{\Sigma})$ as $\mathrm{MSO}^{\otimes}(\widetilde{\Sigma})$ and its first-order fragment as $\mathrm{FO}^{\otimes}(\widetilde{\Sigma})$. It is easy to show — using Kamp's theorem — that $\mathrm{LTL}^{\otimes}(\widetilde{\Sigma})$ has exactly the same expressive power as $\mathrm{FO}^{\otimes}(\widetilde{\Sigma})$.

We also know that $\mathrm{LTL}^{\otimes}$ is strictly weaker than TrPTL. First note that each formula (say α of TrPTL) defines a trace language L_{α} via :

$$L_{\alpha} = \{\, T \mid T, \emptyset \models \alpha \}.$$

Hence we can compare the relative expressive powers of $\mathrm{LTL}^{\otimes}$ and TrPTL. It is known that ([97, 136]):

$$\mathrm{LTL}^{\otimes} \subset \mathrm{TrPTL}^{\mathrm{con}} \subseteq \mathrm{TrPTL}.$$

It is still open whether $\mathrm{TrPTL}^{\mathrm{con}}$ is equal to TrPTL in expressive power.

It is not difficult to show that TrPTL is no more expressive than the first-order theory of traces but it is not known whether the converse also holds. It would be nice to have a linear time temporal logic over traces patterned after LTL which has the same expressive power as the first-order theory of traces. The motivation is provided by the next result [32]:

Proposition 8.8.2 *Let $L \subseteq \Sigma^{\omega}$. Then the following statements are equivalent.*

(1) L is trace consistent and $\mathrm{LTL}(\Sigma)$-definable.

(2) $\{\mathsf{st}(\sigma) \mid \sigma \in L\}$ is $\mathrm{FO}(\Sigma, I)$-definable.

Egged on by this result, recently a different kind of trace-based linear time temporal logic called LTrL has been proposed [139]. This logic works directly with a trace alphabet (i.e. it is not based on agents). It is interpreted over the configurations of a trace and its syntax is given by:

$$\mathrm{LTrL}(\Sigma, I) ::= tt \mid \neg\alpha \mid \alpha \vee \beta \mid \langle a \rangle \alpha \mid \alpha \, U \, \beta \mid \langle a^{-1} \rangle tt.$$

Thus the syntax is very close to LTL except for the addition of a very restricted past-operator. In fact, just a *constant* number of past-operators are present in the logic; one for each action.

A model of $\mathrm{LTrL}(\Sigma, I)$ is a trace $T = (E, \leq, \lambda)$. Let $c \in \mathcal{C}_T$ be a configuration of T. Then $T, c \models \alpha$ will stand for α being satisfied at c in T. This notion is defined inductively as follows:

- $T, c \models tt$.

- $T, c \models \neg\alpha$ and $T, c \models \alpha \vee \beta$ are defined in the expected manner.
- $T, c \models \langle a \rangle \alpha$ iff there exists $c' \in \mathcal{C}_T$ with $c \xrightarrow{a}_T c'$ with $T, c' \models \alpha$.
- $T, c \models \alpha \; U \; \beta$ iff there exists $c' \in \mathcal{C}_T$ with $c \subseteq c'$ such that $T, c' \models \beta$. Moreover, for every $c'' \in \mathcal{C}_T$, $c \subseteq c'' \subset c'$ implies $T, c'' \models \alpha$.
- $T, c \models \langle a^{-1} \rangle tt$ iff there exists $c' \in \mathcal{C}_T$ with $c' \xrightarrow{a}_T c$.

The major result concerning LTrL is the following:

Theorem 8.8.3 ([139]) *Let $L \subseteq \mathbb{TR}^\omega(\Sigma, I)$. Then the following statements are equivalent.*

(1) L is $\mathrm{FO}(\Sigma, I)$-definable.

(2) L is $\mathrm{LTrL}(\Sigma, I)$-definable.

Thus — except for the addition of the restricted past-operators — LTrL is a generalization of Kamp's Theorem to the much richer setting of traces. Meyer and Petit have shown that the past-operators can be eliminated without loss of expressive power when the logic is interpreted over *finite* traces [90]. A similar result for infinite traces is not known at present. Unfortunately this logic does not have a matching time complexity in relation to LTL. Recently Walukiewicz has shown that the satisfiability problem for LTrL is non-elementary hard [149]. A related result concerns the logic TLPO formulated by Ebinger [31]. This is also a linear time temporal logic interpreted over traces but with full-fledged past-operators. TLPO is claimed to be expressively complete when interpreted over *finite* traces but nothing is known about the complexity of the satisfiability problem nor about its expressive power in relation to infinite traces.

At present we do not know much about the relationship between TLC and the logics we have mentioned so far, except that it is strictly weaker than the monadic second-order theory of traces.

In an interesting recent development Niebert [102] has formulated a fixed point based linear time temporal logic for traces in the setting of distributed alphabets. This logic is denoted as νTrPTL. It is equal in expressive power to the monadic second-order theory of traces *and* it has decision procedure of essentially exponential time complexity. However, the formulas of this logic are required to satisfy what appears to be awkward syntactic restrictions and it is not clear how one could express global properties of interest in this formalism.

The relative strengths of the various linear time temporal logics over traces mentioned in this section are displayed in Figure 8.4. A dotted (solid) arrow from A to B indicates that B is at least as expressive as (strictly more expressive than) A. Squiggled lines denote that the logics are incomparable to each other.

To conclude this section, a lot is known about linear time temporal logics for traces but at present we still do not have — unlike the case of sequences — pleasing counterparts to the first-order and monadic second-order theories of traces.

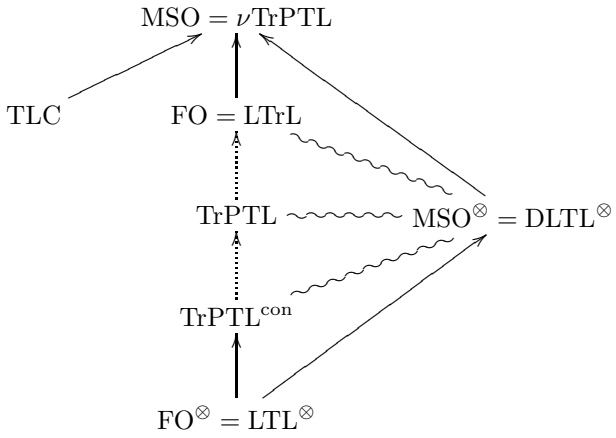


Figure 8.4: Relative expressive power of the logics.

8.9 Conclusion

In this chapter we have attempted an overview of linear time temporal logics interpreted over traces. We have mainly concentrated on the satisfiability and model checking problems as well as expressiveness issues. The problem of axiomatizing these logics seems to be a non-trivial task. Some partial results may be found in [116]. In [109] the authors present proof rules for the logic ISTL with a trace semantics together with a relative expressive completeness result. Reisig has also developed a kit of proof rules for a version of UNITY logic [117, 118]. The models of this logic are the non-sequential processes of a net system and the proof rules are mainly designed to help reason about distributed algorithms modelled using net systems.

At present not much is known about corresponding logics in a branching time setting. Most of the attempts in this direction have lead to logics whose satisfiablity problems are undecidable [18, 81, 112]. It is however the case that the model checking problem often remains tractable [18, 112]. We do not know at present whether the properties expressible in such logics have any type of "all-or-none" flavour and if so whether one can develop some reduction techniques for verifying such properties. Some preliminary attempts in this direction have been made in [45, 153].

Chapter 9

An Expressive Extension of TLC

Contents

A temporal logic of causality (TLC) was introduced by Alur, Penczek, and Peled in [4]. It is basically a linear time temporal logic interpreted over Mazurkiewicz traces which allows quantification over causal chains. Through this device one can directly formulate causality properties of distributed systems. In this chapter we consider an extension of TLC by strengthening the chain quantification operators. We show that our logic TLC* adds to the expressive power of TLC. We do so by defining an Ehrenfeucht-Fraïssé game to capture the expressive power of TLC. We then exhibit a property and by means of this game prove that the chosen property is not definable in TLC. We then show that the same property is definable in TLC*. We prove in fact the stronger result that TLC* is expressively stronger than TLC exactly when the dependency relation associated with the underlying trace alphabet is not transitive. We then show that TLC* defines only regular trace languages by embedding it into the monadic second-order logic. Finally, the relative expressive power of TLC* and similar logics for traces is compared.

9.1 Introduction

One traditional approach to automatic program verification is model checking LTL [113] specifications. In this context, the model checking problem is to decide whether or not all computation sequences of the system at hand satisfy the required properties formulated as an assertion of LTL. Several software packages exploiting the rich theory of LTL are now available to carry out the automated verification task for quite large finite-state systems, e.g. [16, 62].

Usually computations of a distributed system will constitute interleavings of the occurrences of causally independent actions. Often, the computation sequences can be naturally grouped together into equivalence classes of sequences corresponding to different interleavings of the same partially ordered computation stretch. For a large class of interesting properties expressed by linear time temporal logics, it turns out that either all members of an equivalence class satisfy a certain property or none do. For such properties the computional resources needed for the verification task can be substantially reduced by means of the so-called partial-order methods for verification [48, 108, 144].

Such equivalence classes can be canonically represented by restricted labeled partial orders known as Mazurkiewicz traces [27, 86]. These objects — apart from alleviating the state-explosion problems of verification — also allow direct formulations of properties expressing concurrency and causality. A number of linear time temporal logics to be interpreted directly over Mazurkiewicz traces (e.g. [4, 25, 26, 102, 115, 135, 138, 139, 150]) has been proposed in the literature starting with TrPTL [135].

Among these, we consider here a temporal logic of causality (TLC) introduced in [4] to express serializability (of partially ordered computations) in a direct fashion. The operators of TLC are essentially the branching-time operators of CTL [19] interpreted over causal chains of traces. However, the expressive power of this logic has remained an interesting open problem. Indeed, not much is known about the relative expressive powers of the various temporal logics over traces.

What is known is that a linear time temporal LTrL, patterned after LTL, was introduced [139] and proven expressively equivalent to the first-order theory of traces [32]. LTrL has a simple and natural formulation with very restricted past operators, but was shown non-elementary in [149]. Recently, it was shown in a series of papers [25, 26] that the restricted past operators of LTrL can be removed while retaining expressive completeness, essentially extending the celebrated Kamp's Theorem [70] to the setting of traces. In other work, Niebert introduced a fixed point based linear time temporal logic [102]. This logic has an elementary-time decision procedure and is equal in expressive power to the monadic second-order theory of traces. Recently, also Walukiewicz defined a μ-calculus with additional operators with similar expressiveness and decidability, and this logic is interpreted directly over the causal chains of traces in a manner similar to TLC.

However, the expressive powers of most other logics put forth (e.g. [4, 115, 135]) still have an unresolved relationship to each other and, in particular, to

first-order logic. Most notably, it is still a challenging open problem whether or not TrPTL or TLC can express all properties of first-order logic. With virtually no other separation result known, this chapter is a contribution towards understanding the relative expressive power of such logics.

One major weakness of TLC is that it doesn't facilitate direct reasoning about causal relationships between the individual events on the causal chains. In this chapter we remedy this deficiency and extend TLC by strengthening quantification over causal chains. This extended logic, which we call TLC*, will enjoy a similarity to CTL* [19] that TLC has to CTL. The main result of this chapter is that our extension TLC* is expressively stronger than TLC for general trace alphabets whereas they express the same class of properties over trace alphabets where the underlying dependency relation is transitive. We prove this result with the aid of an Ehrenfeucht-Fraïssé game for traces that we develop. To our knowledge this is the first instance of the use of such games to obtain separation results for temporal logics defined over partial orders. We believe that this approach is fruitful and that similar techniques may lead to other separation results within this area.

In the next section we briefly recall Mazurkiewicz traces and a few related notions. In Section 9.3 we introduce TLC and TLC*, the main objects of study in this chapter. We give a very simple and natural example of a property easily captured in TLC* but not in TLC. In Section 9.4 we define an Ehrenfeucht-Fraïssé game and prove its correspondence to TLC. We use this correspondence in Section 9.5 to exhibit a property which we prove is undefinable in TLC. In Section 9.6 we show that the said property can be defined within TLC* and put all the pieces together to arrive at the main result. Following that we show that TLC* can be embedded into monadic second-order logic of traces, which demonstrates that the satisfiability problem of our extension TLC* remains decidable and that the expressive power stays within the realm of the regular trace languages. Finally, we proceed in Section 9.7 with a quick overview of the relative expressive powers of related logics for traces before concluding the chapter by considering a few open problems.

9.2 Preliminaries

A *(Mazurkiewicz) trace alphabet* is a pair (Σ, I), where Σ, the alphabet, is a finite set and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric *independence relation*. Usually, Σ consists of the actions performed by a distributed system while I captures a static notion of causal independence between actions. For the rest of the section we fix a trace alphabet (Σ, I). We define $D = (\Sigma \times \Sigma) - I$ to be the *dependency relation* which is then reflexive and symmetric. For the purpose of interpreting temporal logics of causality we will adopt the viewpoint that traces are restricted labeled partial orders of events and hence have an explicit representation of causality and concurrency.

Let $T = (E, \leq, \lambda)$ be a Σ-labeled poset. In other words, $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $e \in E$ we define $\downarrow e = \{x \in E \mid x \leq e\}$.

We also let $\lessdot$ be the *covering relation* given by $x \lessdot y$ iff $x < y$ and for all $z \in E$, $x \leq z \leq y$ implies $x = z$ or $z = y$. Moreover, we let the *concurrency relation* be defined as $x \; co \; y$ iff $x \not\leq y$ and $y \not\leq x$. A *Mazurkiewicz trace* (over (Σ, I)) is then a Σ-labeled poset $T = (E, \leq, \lambda)$ satisfying:

- $\downarrow e$ is a finite set for each $e \in E$.
- For every $e, e' \in E$, $e \lessdot e'$ implies $\lambda(e) \; D \; \lambda(e')$.
- For every $e, e' \in E$, $\lambda(e) \; D \; \lambda(e')$ implies $e \leq e'$ or $e' \leq e$.

We shall let $\mathbb{TR}(\Sigma, I)$ denote the class of traces over (Σ, I). As usual, a trace language L is a subset of traces, i.e. $L \subseteq \mathbb{TR}(\Sigma, I)$. Throughout the chapter we will not distinguish between isomorphic elements in $\mathbb{TR}(\Sigma, I)$. We will refer to members of E as *events*. We will for convenience always assume the existence of a unique least event $\perp \in E$ corresponding to a system initialization event carrying no label, i.e. $\lambda(\perp)$ is undefined and $\perp < e$ for every $e \in E - \{\perp\}$.

In its original formulation [86], Mazurkiewicz introduced traces as $\approx_I$-equivalence classes of strings induced by the congruence relation generated by: $\tau ab\sigma \approx_I \tau ba\sigma$ whenever aIb. This viewpoint of traces is also the insight underlying to so-called partial order methods for verification. It is, however, not hard to show that the traces introduced above as restricted labeled partial orders can be represented as $\approx_I$-equivalence classes of strings (and vice versa; see e.g. [138]). Hence they denote the same class of objects, so we will sometimes abuse notation and let a string in Σ^* denote its corresponding trace in $\mathbb{TR}(\Sigma, I)$ whenever no confusion arises. This is enforced by using conventional parentheses for string languages and square brackets for trace languages.

In setting the scene for defining the semantics of formulas of TLC* we first introduce some notation for sequences. The length of a finite sequence ρ will be denoted by $|\rho|$. In case ρ is infinite we set $|\rho| = \omega$. Let $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ and $0 \leq k < |\rho|$. We set $\rho_k = (e_k, e_{k+1}, \ldots, e_n, \ldots)$.

Let $T = (E, \leq, \lambda)$ be a trace over (Σ, I). A *future causal chain rooted at* $e \in E$ is a (finite or infinite) sequence $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ with $e = e_0$, $e_i \in E$ such that $e_{i-1} \lessdot e_i$ for every $i \geq 1$. The labelling function $\lambda : E \to \Sigma$ is extended to causal chains in the obvious way by: $\lambda(\rho) = (\lambda(e_0)\lambda(e_1) \cdots \lambda(e_n) \cdots)$. We say that a future causal chain ρ is *maximal* in case ρ is either infinite or it is finite and there exists no $e' \in E$ such that $e_{|\rho|} \lessdot e'$.

A *past causal chain rooted at* $e \in E$ is a (finite) sequence $\rho = (e_n, \ldots, e_1, e_0)$ with $e = e_0$, $e_i \in E$ such that $e_i \lessdot e_{i-1}$ for every $1 \leq i \leq n$. In this case we will use ρ_{-k} to denote $(e_n, \ldots, e_{k+1}, e_k)$ for $0 \leq k < |\rho|$.

9.3 Syntax and Semantics

In this section we will define the syntax and semantics of the temporal logics over traces to be considered in this chapter. We start by introducing TLC* and continue by giving an explicit definition of the sublogic TLC.

TLC* is parameterized by a trace alphabet (Σ, I) and consists of three different syntactic entities; event formulas (Φ_{ev}), future chain formulas (Φ^+_{ch}) and past chain formulas (Φ^-_{ch}) defined by mutual induction as described below:

$$\begin{array}{lcl} \Phi_{ev} & ::= & p_a \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \mathrm{co}(\alpha) \mid E(\varphi) \mid E^-(\psi), \quad a \in \Sigma \\ \Phi^+_{ch} & ::= & \alpha \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2 \\ \Phi^-_{ch} & ::= & \alpha \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid X^-\psi \mid \psi_1 U^- \psi_2 \ , \end{array}$$

where α, φ and ψ with or without subscripts here and throughout the rest of the chapter are formulas of Φ_{ev}, Φ^+_{ch} and Φ^-_{ch}, respectively.

The formulas of TLC$^*(\Sigma, I)$ are the set of *event* formulas Φ_{ev} as defined above[1] and will be interpreted over traces of $\mathbb{TR}(\Sigma, I)$. To make the distinction between event and chain formulas easier we will throughout the remainder of the chapter whenever possible use α, β, γ (respectively, φ, ψ) with or without primes and subscripts to denote event formulas (respectively, chain formulas).

The semantics of formulas of TLC* is divided into two parts; event formulas and chain formulas. Let $T \in \mathbb{TR}(\Sigma, I)$ and $e \in E$. The notion of an event formula α being satified at an event e of T is defined inductively in the following manner:

- $T, e \models p_a$ iff $\lambda(e) = a$.
- $T, e \models \neg\alpha$ iff $T, e \not\models \alpha$.
- $T, e \models \alpha_1 \vee \alpha_2$ iff $T, e \models \alpha_1$ or $T, e \models \alpha_2$.
- $T, e \models co(\alpha)$ iff there exists an $e' \in E$ with $e \; co \; e'$ and $T, e' \models \alpha$.
- $T, e \models E(\varphi)$ iff there exists a future causal chain ρ rooted at e with $T, \rho \models \varphi$.
- $T, e \models E^-(\psi)$ iff there exists a past causal chain ρ rooted at e with $T, \rho \models \psi$.

As usual, $\mathrm{tt} = p_a \vee \neg p_a$ and $\mathrm{ff} = \neg\mathrm{tt}$. Suppose $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ is a future causal chain. The notion of $T, \rho \models \varphi$ for a future chain formula φ is defined inductively below.

- $T, \rho \models \alpha$ iff $T, e_0 \models \alpha$.
- $T, \rho \models \neg\varphi$ iff $T, \rho \not\models \varphi$.
- $T, \rho \models \varphi_1 \vee \varphi_2$ iff $T, \rho \models \varphi_1$ or $T, \rho \models \varphi_2$.
- $T, \rho \models X\varphi$ iff $T, \rho_1 \models \varphi$.
- $T, \rho \models \varphi_1 U \varphi_2$ iff there exists a $0 \leq k < |\rho|$ such that $T, \rho_k \models \varphi_2$. Moreover, $T, \rho_m \models \varphi_1$ for each $0 \leq m < k$.

[1] Another logic was in [4] termed "TLC*", but as that logic denoted TLC interpreted over linearizations it is unrelated to our logic which seems naturally to earn the name "TLC*".

The well-known future chain operators are derived as $F\varphi = \mathrm{tt}U\varphi$ and $G\varphi = \neg F\neg\varphi$.

The notion of $T, \rho \models \psi$ for a past causal chain ρ and past chain formula ψ is defined in the expected manner. In particular;

- $T, \rho \models X^-\psi$ iff $T, \rho_{-1} \models \psi$.
- $T, \rho \models \psi_1 U^- \psi_2$ iff there exists a $0 \leq k < |\rho|$ such that $T, \rho_{-k} \models \psi_2$. Moreover, $T, \rho_{-m} \models \psi_1$ for each $0 \leq m < k$.

(Recall that a past causal chain is of the form $\rho = (e_n, \ldots, e_1, e_0)$ with $e = e_0$, $e_i \in E$ such that $e_i \lessdot e_{i-1}$ for every $1 \leq i \leq n$.)

Suppose $T \in \mathbb{TR}(\Sigma, I)$ and $\alpha \in \mathrm{TLC}^*(\Sigma, I)$. Then T *satisfies* α iff $T, \perp \models \alpha$, which we will usually denote as $T \models \alpha$. The formula α is *satisfiable* if there exists a trace T satisfying α, and the *satisfiability problem* is to decide whether α is satisfiable. The *language defined by* α is $L(\alpha) = \{T \in \mathbb{TR}(\Sigma, I) \mid T \models \alpha\}$. We say that $L \subseteq \mathbb{TR}(\Sigma, I)$ is *definable* in TLC^* if there exists some $\alpha \in \mathrm{TLC}^*(\Sigma, I)$ such that $L(\alpha) = L$. By slight abuse of notation, the class of trace languages over (Σ, I) definable in TLC^* will also be denoted by $\mathrm{TLC}^*(\Sigma, I)$.

The formulas of $\mathrm{TLC}(\Sigma, I)$ — introduced in [4] with a slightly different syntax — is then the set of formulas of $\mathrm{TLC}^*(\Sigma, I)$ where each of the chain operators X, U, G, X^-, U^- is immediately preceded by a chain quantifier E. As TLC will play a prominent role in this chapter we will bring out its definition in more detail. More precisely, the set of formulas of TLC is given as:

$$\begin{aligned} \mathrm{TLC}(\Sigma, I) \quad ::= \quad & p_a \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \mathrm{co}(\alpha), \quad a \in \Sigma \\ & EX(\alpha) \mid EU(\alpha_1, \alpha_2) \mid EG(\alpha) \\ & EX^-(\alpha) \mid EU^-(\alpha_1, \alpha_2) \quad , \end{aligned}$$

The semantics is inherited directly from TLC^* in the obvious manner, so notions of definability etc. are carried over directly.

While the formulas of TLC basically consist of the well-known operators of the branching-time logic CTL [19] augmented with symmetrical past operators and concurrency information, the operators of TLC^* are basically the well-known operators of CTL^*[19] similarly extended with past quantifiers in a restricted fashion as well as concurrency information. The crucial difference is that while CTL and CTL^* are branching-time logics interpreted over Kripke structures, TLC and TLC^* are linear time temporal logics on traces interpreted over the underlying Hasse diagrams of the partial orders.

The salient feature of TLC is that its satisfiability problem is PSPACE-complete and decidable in exponential time [4], i.e. the computational resources needed is similar to those of LTL. It is not hard to show that the satisfiability problem for our extension TLC^* remains decidable. This follows by a translation of formulas of TLC^* into monadic second-order logic of traces, which will be dealt with in Section 9.6 when we investigate the expressive power of TLC^*.

One of the weaknesses of TLC is that it doesn't directly facilitate reasoning about causal relationships of the individual events of the causal chains at hand.

As a consequence, a number of interesting properties are not (either easily or at all) expressible within TLC. Section 9.5 provides a formal proof of this claim, but we will in the following bring out another such property which is very natural.

Suppose that a and b are actions representing the acquiring and releasing, respectively, of some resource. A relevant property of this system is then whether or not there exists some causal chain in the execution of the system — presumably containing other system actions than $\{a, b\}$ — such that the a's and b's alternate strictly until the task is perhaps eventually completed. Via the future chain formula $\varphi_{xy} = p_x \rightarrow X(\neg(p_x \vee p_y)Up_y)$ we can easily express this property in TLC* as $E(G(\varphi_{ab} \wedge \varphi_{ba}))$. The point is here that TLC* allows us to investigate each causal chain in mention by a causal chain formula, which is then confined to this very chain. This is not possible in TLC, as the existential quantifications interpreted at some fixed event of the chain would potentially consider *all* causal chains originating at this event — not just the one presently being investigated.

We conclude this section with two important notions relating to TLC. Firstly, let α be a formula of TLC(Σ, I). The *operator depth* of α is defined inductively as follows:

- $od(p_a) = 0$.
- $od(\neg\alpha) = od(\alpha)$.
- $od(\alpha \vee \beta) = \max(od(\alpha), od(\beta))$.
- $od(EX(\alpha)) = od(EG(\alpha)) = od(EX^-(\alpha)) = od(\mathrm{co}(\alpha)) = 1 + od(\alpha)$.
- $od(EU(\alpha, \beta)) = od(EU^-(\alpha, \beta)) = 1 + \max(od(\alpha), od(\beta))$.

The set of formulas of operator depth k is denoted by $OD(k)$.

Given $T_0, T_1 \in \mathbb{TR}(\Sigma, I)$ and e_i events of T_i we define that $(T_0, e_0) \equiv_n (T_1, e_1)$ if for any formula $\alpha \in \mathrm{TLC}(\Sigma, I)$ with $od(\alpha) \leq n$, $T_0, e_0 \models \alpha$ if and only if $T_1, e_1 \models \alpha$, i.e. both structures agree on all subformulas of operator depth at most $n \geq 0$. It is then not hard to see that $(T_0, e_0) \equiv_0 (T_1, e_1)$ if and only if e_0 and e_1 are identically labeled, i.e. either $\lambda(e_0) = \lambda(e_1)$ or $e_0 = e_1 = \bot$.

9.4 An Ehrenfeucht-Fraïssé Game for TLC

In this section we will present an Ehrenfeucht-Fraïssé game to capture the expressive power of TLC. The game is played directly on the poset representation of (finite or infinite) Mazurkiewicz traces and it is similar in spirit to the Ehrenfeucht-Fraïssé game for LTL introduced by Etessami and Wilke [37]. We extend their approach to the richer setting of traces by highlighting current causal chains in the until-based moves and adding past- and co-moves.

The EF-TLC game is a game played between two persons, Spoiler and Preserver, on a pair of traces (T_0, T_1). The game is played over k rounds starting from an initial game state (e_0, e_1) and after each round the current game state

is a pair of events (e'_0, e'_1) with $e'_i \in E_i$. Each round starts with the game in some specific initial game state (e_0, e_1) and Spoiler chooses one of the moves defined below and the game proceeds accordingly:

EX-**Move:** This move can only be played by Spoiler if there exists an $e'_0 \in E_0$ such that $e_0 \lessdot e'_0$ or there exists an $e'_1 \in E_1$ such that $e_1 \lessdot e'_1$. Spoiler then wins the game in case there either exists no $e'_0 \in E_0$ such that $e_0 \lessdot e'_0$ or no $e'_1 \in E_1$ such that $e_1 \lessdot e'_1$. Otherwise (in which case both e_0 and e_1 has $\lessdot$-successors) the game proceeds as follows:

(i) Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e_i \lessdot e'_i$.

(ii) Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ such that $e_{1-i} \lessdot e'_{1-i}$.

(iii) The new game state is now (e'_0, e'_1).

EU-**Move:**

(i) Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e_i \leq e'_i$ and he highlights a future causal chain $(e_i = f_i^0, f_i^1, \ldots, f_i^n = e'_i)$ with $n \geq 0$.

(ii) Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ with $e_{1-i} \leq e'_{1-i}$ such that if $e_i = e'_i$ then $e_{1-i} = e'_{1-i}$. Furthermore she highlights a future causal chain $(e_{1-i} = f_{1-i}^0, f_{1-i}^1, \ldots, f_{1-i}^m = e'_{1-i})$ with $m \geq 0$.

(iii) Spoiler now chooses one of the following two steps:

- Spoiler sets the game state to (e'_0, e'_1).
- Spoiler chooses an event $f_{1-i} \in \{f_{1-i}^0, f_{1-i}^1 \ldots f_{1-i}^m\}$. Preserver responds with an event $f_i \in \{f_i^0, f_i^1 \ldots f_i^n\}$ and the game continues in the state (f_0, f_1).

EG-**Move:**

(i) Spoiler chooses $i \in \{0, 1\}$, and highlights a maximal future causal chain $(e_i = f_i^0, f_i^1, \ldots, f_i^n, \ldots)$ with $f_i^j \in E_i$ and $n \geq 0$.

(ii) Preserver responds by highlighting a maximal future causal chain $(e_{1-i} = f_{1-i}^0, f_{1-i}^1, \ldots, f_{1-i}^m, \ldots)$ with $f_i^j \in E_{1-i}$ and $m \geq 0$.

(iii) Spoiler chooses an event $f_{1-i} \in \{f_{1-i}^0, f_{1-i}^1 \ldots f_{1-i}^m\}$. Preserver responds with an event $f_i \in \{f_i^0, f_i^1 \ldots f_i^n\}$ and the game continues in the state (f_0, f_1).

co-**Move:** This move can only be played by Spoiler if there exists an $e'_0 \in E_0$ such that $e_0 \mathrel{co} e'_0$ or there exists an $e'_1 \in E_1$ such that $e_1 \mathrel{co} e'_1$. Spoiler then wins the game in case there either exists no $e'_0 \in E_0$ such that $e_0 \mathrel{co} e'_0$ or no $e'_1 \in E_1$ such that $e_1 \mathrel{co} e'_1$. Otherwise (in which case both e_0 and e_1 have concurrent events) the game proceeds as follows:

(i) Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e_i \mathrel{co} e'_i$ in T_i.

(ii) Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ such that e_{1-i} *co* e'_{1-i} in T_{1-i}.

(iii) The new game state is now (e'_0, e'_1).

EX^-**-Move:** This move can only be played by Spoiler if $e_0 \neq \bot$ or $e_1 \neq \bot$. Spoiler then wins the game in case $e_0 = \bot$ or $e_1 = \bot$. Otherwise (in which case neither e_0 nor e_1 is $\bot$) the game proceeds as follows:

(i) Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e'_i \lessdot e_i$.

(ii) Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ such that $e'_{1-i} \lessdot e_{1-i}$.

(iii) The new game state is now (e'_0, e'_1).

EU^-**-Move:**

(i) Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e'_i \leq e_i$ and he highlights a past causal chain $(e'_i = f_i^n, f_i^{n-1}, \ldots, f_i^0 = e_i)$ with $n \geq 0$.

(ii) Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ with $e'_{1-i} \leq e_{1-i}$ such that if $e_i = e'_i$ then $e_{1-i} = e'_{1-i}$. Furthermore she highlights a past causal chain $(e'_{1-i} = f_{1-i}^m, f_{1-i}^{m-1}, \ldots, f_{1-i}^0 = e_{1-i})$ with $m \geq 0$.

(iii) Spoiler now chooses one of the following two steps:

- Spoiler sets the game state to (e'_0, e'_1).
- Spoiler chooses an event $f_{1-i} \in \{f_{1-i}^0, f_{1-i}^1 \ldots f_{1-i}^m\}$. Preserver responds with an event $f_i \in \{f_i^0, f_i^1 \ldots f_i^n\}$ and the game continues in the state (f_0, f_1).

In the 0-round game Spoiler wins if $(T_0, e_0) \not\equiv_0 (T_1, e_1)$ and otherwise Preserver wins. In the $(k+1)$-round game Spoiler wins if $(T_0, e_0) \not\equiv_0 (T_1, e_1)$. If it is the case that $(T_0, e_0) \equiv_0 (T_1, e_1)$, a round is played according to the above moves. This round either results in a win for Spoiler (e.g. by the EX-move) or a new game state (e'_0, e'_1). In the latter case, a k-round game is then played starting from the initial game state (e'_0, e'_1).

We say that *Preserver has a winning strategy* in the k-round game on (T_0, e_0) and (T_1, e_1), denoted $(T_0, e_0) \sim_k (T_1, e_1)$, if she can win the k-round game on the structures T_0 and T_1 starting in the initial game state (e_0, e_1) no matter which moves are performed by Spoiler. If not, we say that *Spoiler has a winning strategy*. We refer to [37] for basic intuitions about the game.

Our interest in the game lies in the following fact.

Proposition 9.4.1 $(T_0, e_0) \sim_k (T_1, e_1)$ *if and only if* $(T_0, e_0) \equiv_k (T_1, e_1)$.

Proof: We prove that $(T_0, e_0) \sim_k (T_1, e_1)$ if and only if $(T_0, e_0) \equiv_k (T_1, e_1)$ by induction on k. The base case where $k = 0$ follows trivially from the definition.

For the inductive step suppose that the claim is true for k. We first prove the direction from left to right. Suppose that $(T_0, e_0) \sim_{k+1} (T_1, e_1)$. Let $\alpha \in$

TLC(Σ, I) with $od(\alpha) = k+1$. We must show that $T_0, e_0 \models \alpha$ if and only if $T_1, e_1 \models \alpha$. It suffices to prove the statement when the top-level connective of α is a chain-operator because by boolean combinations (T_0, e_0) and (T_1, e_1) would then agree on all formulas of operator depth $k+1$. We will only consider the case where the top-level chain-operator is the EU-operator. The other cases proceed in a very similar manner.

Suppose $\alpha = EU(\beta, \beta')$. Assume without loss of generality that $T_0, e_0 \models \alpha$, i.e. there exists a future causal chain $\rho^0 = (f_0^0, f_0^1, \ldots, f_0^n)$ with $e_0 = f_0^0$ and $f_0^n = e_0'$ such that $T_0, f_0^j \models \beta$ for each $0 \leq j < n$ and $T_0, e_0' \models \beta'$. Hence we let Spoiler play the EU-move on T_0 and make him highlight ρ^0 on T_0. Preserver now uses her winning strategy and highlights $\rho^1 = (f_1^0, f_1^1, \ldots, f_1^m)$ with $e_1 = f_1^0$ and $f_1^m = e_1'$. Two subcases now arise.

Assume first that Spoiler sets the new game state to (e_0', e_1'). As e_1' was chosen from Preserver's winning strategy we have that $(T_0, e_0') \sim_k (T_1, e_1')$ which by induction hypothesis implies that $(T_0, e_0') \equiv_k (T_1, e_1')$. Thus $T_1, e_1' \models \beta'$. Now, assume that Spoiler instead picked an event f_1 on ρ^1. By Preserver's winning strategy she could pick an event f_0 on ρ^0 (This is possible due to the requirement that if $e_0 = e_0'$ then $e_1 = e_1'$). Again by the winning strategy we have that $(T_0, f_0) \sim_k (T_1, f_1)$ and by induction hypothesis that $T_1, f_1 \models \beta$. Hence $T_1, f_1 \models EU(\beta, \beta')$, which concludes this direction of the proof.

We prove the direction from right to left by contraposition, so suppose that $(T_0, e_0) \not\sim_{k+1} (T_1, e_1)$. We will then exhibit a formula $\alpha \in$ TLC(Σ, I) with $od(\alpha) = k+1$ such that $T_0, e_0 \models \alpha$ but $T_1, e_1 \not\models \alpha$. Again, we will only prove the case where Spoiler's first move of his winning strategy is either the EU-move. The other cases either follows in analogous or easier manners.

Suppose Spoiler plays the EU-move on T_0 (without loss of generality), i.e. he chooses a future causal chain $\rho^0 = (f_0^0, f_0^1, \ldots, f_0^n)$ with $e_0 = f_0^0$ and $f_0^n = e_0'$. It is not hard to show by induction that there are only a finite number of semantically inequivalent formulas α with $od(\alpha) \leq k$ and $T_0, e \models \alpha$ for any $e \in E_0$. Hence, each formula $\beta_0^j = \bigwedge\{\alpha \in OD(k) \mid T_0, f_0^j \models \alpha\} \wedge \bigwedge\{\neg\alpha \in OD(k) \mid T_0, f_0^j \not\models \alpha\}$ is well-defined and equivalent to a formula of operator depth k for each $0 \leq j < n$, so letting $\beta_{e_0'} = \beta_0^n$ we have that $\alpha = EU(\bigvee_{0 \leq j < n} \beta_0^j, \beta_{e_0'})$ is a TLC-formula with $od(\alpha) = k+1$ and by definition $T_0, e_0 \models \alpha$. We will argue that $T_1, e_1 \not\models \alpha$.

Suppose that $T_1, e_1 \models \alpha$. Then there exists a future causal chain $\rho^1 = (f_1^0, f_1^1, \ldots, f_1^m)$ with $e_1 = f_1^0$ and $f_1^m = e_1'$ such that $T_1, f_1^l \models \bigvee_{0 \leq j < n} \beta_0^j$ for each $0 \leq l < m$ and $T_1, e_1' \models \beta_{e_0'}$.

Assume first that Spoiler chooses to set the new game state to (e_0', e_1') by following his winning strategy. As $T_1, e_1' \models \beta_{e_0'}$ it must be the case that for each $\gamma \in OD(k)$, $T_0, e_0' \models \gamma$ if and only if $T_1, e_1' \models \gamma$. By induction hypothesis $(T_0, e_0') \sim_k (T_1, e_1')$ which contradicts that Spoiler has a winning strategy because Preserver could initially have played ρ^1 as above and continued according to $(T_0, e_0') \sim_k (T_1, e_1')$.

Now assume that Spoiler instead by his winning strategy picks an event f_1 on ρ^1. Then $T_1, f_1 \models \beta_0^j$ for some $0 \leq j < n$ as $T_1, f_1 \models \bigvee_{0 \leq j < n} \beta_0^j$.

Again by induction hypothesis we know that $(T_0, f_0^j) \sim_k (T_1, f_1)$ which again contradicts that Spoiler has a winning strategy because Preserver could respond by picking $f_0^j \in E_0$ and continue from the game state (f_0^j, f_1) according to $(T_0, f_0^j) \sim_k (T_1, f_1)$.

Hence $T_1, e_1 \not\models \alpha$ as required. This concludes the proof of the direction from right to left. □

9.5 An Undefinability Result

We have now set the stage to begin the study of the expressive power of TLC*. It is clear that TLC is a syntactic subset of TLC* whence $\mathrm{TLC}(\Sigma, I) \subseteq \mathrm{TLC}^*(\Sigma, I)$. In this section we will give an example of a natural property, which we by means of the Ehrenfeucht-Fraïssé game characterization for TLC of the previous section will show is *not* expressible in TLC. In the next section we show that this property is indeed definable in TLC*. In conjunction these facts show that TLC* adds to the expressive power of TLC.

Specifically, let (Σ, I) be a trace alphabet with $\{a, b, c\} \subseteq \Sigma$ such that $a \; D \; c$ and $c \; D \; b$ but $a \; I \; b$. Consider $L = [abcabc]^* \subseteq \mathbb{TR}(\Sigma, I)$.

Lemma 9.5.1 *L is not definable in* $\mathrm{TLC}(\Sigma, I)$.

Proof: Let $k \geq 0$ be given and consider $T_0^k = [abc]^{4k}$ and $T_1^k = [abc]^{4k+1}$. It suffices to show that $(T_0^k, \bot) \sim_k (T_1^k, \bot)$. By Proposition 9.4.1 it then follows that $(T_0^k, \bot) \equiv_k (T_1^k, \bot)$. Suppose L would be definable by a TLC-formula α of operator depth n. In particular, then $(T_0^n, \bot) \equiv_n (T_1^n, \bot)$. However, by definition it must be the case that $T_0^n \in L$ and $T_1^n \notin L$, contradicting that T_0^n and T_1^n satisfy the same set of formulas of operator depth at most n. Hence, L cannot be expressed by any formula of TLC assuming that $(T_0^k, \bot) \sim_k (T_1^k, \bot)$ holds for each $k \geq 0$.

The remainder of the proof will be devoted to showing that it is the case that $(T_0^k, \bot) \sim_k (T_1^k, \bot)$. To bring this out we need a few definitions. As depicted in Figure 9.1 the game is played on T_0^k and T_1^k consisting of $4k$ and $4k + 1$ copies of the trace factor $[abc]$, respectively. The section of e_i for $i \in \{0, 1\}$ is then defined to be the number of the enclosing $[abc]$-factor in T_k^i counting from left and starting with 1. We denote this number by $sect(e_i)$. In case $e_i = \bot$ we set $sect(e_i) = 0$. Furthermore, we say that e_0 and e_1 are *position equivalent*, in case either $(e_0, e_1) = (\bot, \bot)$ or $\lambda(e_0) = \lambda(e_1)$. From the definition of T_0 and T_1 it follows that e_0 and e_1 are position equivalent in case e_0 and e_1 denote the same local positions in two (possibly distinct) sections of T_0 and T_1, respectively. The unique event of section $s \geq 1$ labeled with letter $x \in \{a, b, c\}$ in T_i^k will be denoted $e_i^{x,s}$. For example, the fourth b-labeled event of T_0^k is denoted $e_0^{b,4}$.

We will then show that Preserver has a strategy such that after $k' \leq k$ rounds played on (T_0^k, T_1^k) with current game state (e_0, e_1), the following invariant holds:

(i) e_0 and e_1 are position equivalent.

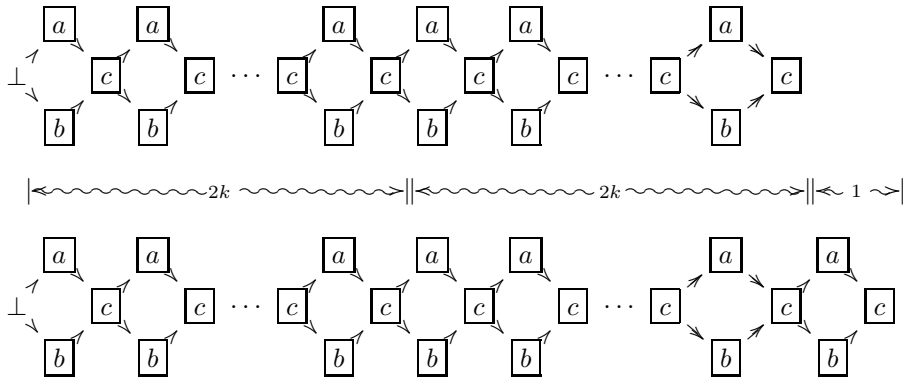


Figure 9.1: T_0^k (top) and T_1^k (bottom) on which the game is played.

(ii) $sect(e_0) = sect(e_1)$ or $sect(e_0) = sect(e_1) - 1$.

(iii) $sect(e_0) = sect(e_1)$ implies $sect(e_0) \leq 2(k + k')$.

(iv) $sect(e_0) = sect(e_1) - 1$ implies $sect(e_0) \geq 2(k - k') + 1$.

We prove that the invariant holds by induction on k'. It is trivial to observe, that in the base case we have that $(e_0, e_1) = (\bot, \bot)$, $sect(e_0) = sect(e_1) = 0$ and $k' = 0$ thus satisfying (i),(ii), (iii) and (iv) above.

For the inductive step, assume that the statement holds for $k' < k$. From (i) it follows that $(T_0, e_0) \equiv_0 (T_1, e_1)$, so a next round is played. We then show that the Preserver can move so as to maintain the invariant for the next game state (e_0', e_1') by case analysis on the next move chosen by Spoiler. We only consider the case for the EU-move. The other moves follow analogously. From (ii) we know that $sect(e_0) = sect(e_1)$ or $sect(e_0) = sect(e_1) - 1$, so two subcases arise.

Case I: $sect(e_0) = sect(e_1)$. Suppose Spoiler chooses to play the EU-move on T_0^k and highlights a future causal chain $\rho^0 = (e_0 = e_0^{x_0,s_0}, e_0^{x_1,s_1}, \ldots, e_0^{x_n,s_n} = e_0')$. By assumption $sect(e_0) \leq 2(k + k')$.

Suppose first that $s_n \leq 2(k+k'+1)$. Then Preserver can just copy the move and respond with $\rho^1 = (e_1 = e_1^{x_0,s_0}, e_1^{x_1,s_1}, \ldots, e_1^{x_n,s_n} = e_1')$. If Spoiler chooses to set the new game state to (e_0', e_1'), $sect(e_0') = sect(e_1') \leq 2(k + k' + 1)$ and the invariant is maintained. If Spoiler instead chooses to pick an event $e_1^{x_i,s_i}$, Preserver would respond by picking $e_0^{x_i,s_i}$ and the invariant is maintained in a similar manner.

Suppose then that $s_n > 2(k + k' + 1)$. Preserver must then "insert" an additional occurrence of a section into ρ^0 at section $2(k + k' + 1)$. To bring this out, let l be the least index such that $s_l = 2(k + k' + 1)$, which exists by assumption. Preserver then responds with $\rho^1 =$

$$(e_1^{x_0,s_0}, \ldots, e_1^{x_l,s_l}, e_1^{x_{l+1},s_{l+1}}, e_1^{x_l,s_l+1}, e_1^{x_{l+1},s_{l+1}+1}, e_1^{x_{l+2},s_{l+2}+1}, \ldots, e_1^{x_n,s_n+1})$$

with $e_1' = e_1^{x_n,s_n+1}$. If Spoiler chooses to set the new game state to (e_0', e_1'), $sect(e_0') = s_n = sect(e_1') - 1$. However, the invariant is maintained as $sect(e_0') \geq 2(k + k' + 1) \geq 2(k - (k' + 1)) + 1$. If Spoiler instead chooses to pick an event on ρ^1, Preserver responds dependent upon its index. If Spoiler picks one of the first $l + 2$ events $e_1^{x_i,s_i}$, Preserver responds with $e_0^{x_i,s_i}$. As $sect(e_0^{x_i,s_i}) = s_i = sect(e_1^{x_i,s_i}) \leq 2(k + k' + 1)$ the invariant is maintained. If Spoiler picks one of the remaining events $e_1^{x_i,s_i+1}$, Preserver responds with $e_0^{x_i,s_i}$ in which case $sect(e_0^{x_i,s_i}) = s_i = sect(e_1^{x_i,s_i+1}) - 1$ and the invariant is maintained as $sect(e_0^{x_i,s_i}) \geq 2(k + k' + 1) > 2(k - (k' + 1)) + 1$.

Suppose spoiler chooses to play the EU-move on T_1^k and highlights a future causal chain $\rho^1 = (e_1 = e_1^{x_0,s_0}, e_1^{x_1,s_1}, \ldots, e_1^{x_n,s_n} = e_1')$. By assumption $sect(e_0) \leq 2(k + k')$. If $s_n \leq 2(k + k' + 1)$ then Preserver can, as above, just copy the move and maintain the invariant, so suppose that $s_n > 2(k + k' + 1)$. Preserver must then "chop" a duplicate occurrence off ρ^1 around the sections $2(k+k')+1, 2(k+k')+2 = 2(k+k'+1), 2(k+k')+3$ which exist by construction. Any causal chain passing through these three sections must pass (at least) two identical ac-labeled or bc-labeled stretches. Now, let l be the least index such that $s_l = 2(k + k') + 1$ and consider the sequence $\sigma = (x_l, x_{l+2}, x_{l+4})$ with $\lambda(\sigma) \in \{a, b\}^3$. Remove from σ the first occurrence x_i where there exists an $j > i$ with x_j in σ and $x_i = x_j$. Let $\sigma' = (x_p, x_q)$ denote the resulting sequence where $p, q \in \{l, l + 2, l + 4\}$. Preserver then plays the chain $\rho^0 =$

$$(e_0^{x_0,s_0}, \ldots, e_0^{x_{l-1},s_{l-1}}, e_0^{x_p,s_l}, e_0^{c,s_{l+1}}, e_0^{x_q,s_{l+2}}, e_0^{c,s_{l+3}}, e_0^{x_{l+5},s_{l+5}-1}, \ldots, e_0^{x_n,s_n-1})$$

with $e_0' = e_0^{x_n,s_n-1}$. If Spoiler chooses to set the new game state to (e_0', e_1') then $sect(e_0') = s_n = sect(e_1') - 1$ so the invariant is maintained because $s_n > 2(k + k' + 1) > 2(k - (k' + 1)) + 1$. If Spoiler chooses to pick an event on ρ^0, Preserver responds according to one of several cases. If Spoiler picks one of the first l events $e_0^{x_i,s_i}$ then Preserver picks $e_1^{x_i,s_i}$ and the invariant is maintained as usual. If Spoiler picks either $e_0^{c,s_{l+1}}$ or $e_0^{c,s_{l+3}}$ then Preserver picks either $e_1^{c,s_{l+1}}$ or $e_1^{c,s_{l+3}}$, respectively. As the sections are both s_{l+1} or both s_{l+3} and $s_{l+1} < s_{l+3} = s_l + 1 = 2(k + k' + 1)$ the invariant follows. If Spoiler picks an event, $e_0^{x_m,s}$ say, in $\{e_0^{x_p,s_l}, e_0^{x_q,s_{l+2}}\}$ *before* the removed occurrence in σ then $m \in \{l, l+2\}$ and Preserver responds by $e_1^{x_m,s}$. Then $sect(e_0^{x_m,s}) = s = sect(e_1^{x_m,s}) \leq s_{l+2} = 2(k + k' + 1)$. Similarly, if $e_0^{x_m,s}$ occurs *after* the removed occurrence then $m \in \{l + 2, l + 4\}$ and Preserver picks $e_1^{x_m,s+1}$. Then $sect(e_0^{x_m,s}) = s = sect(e_1^{x^m,s+1}) - 1 \geq 2(k + k') > 2(k - (k' + 1)) + 1$ and in both cases the invariant is maintained. Finally, if Spoiler picks one of the remaining events $e_0^{x_i,s_i-1}$ with $i \geq l + 5$ then Preserver responds with $e_1^{x_i,s_i}$. As $sect(e_0^{x_i,s_i-1}) = sect(e_1^{x_i,s_i}) - 1 \geq 2(k - (k' + 1)) + 1$ the invariant is also maintained in this case.

Case II: $sect(e_0) = sect(e_1) - 1$. Here the futures of e_0 in T_0^k and e_1 in T_1^k both consist of $4k - sect(e_0)$ factors of $[abc]$ and are identical with respect to future moves. Hence Preserver can just "copy" the move made by Spoiler with the obvious correspondence that $e_0^{x_i,s_i}$ is matched with $e_1^{x_i,s_i+1}$ and vice versa. □

9.6 The Expressiveness of TLC*

In this section we use Lemma 9.5.1 to tie up the knots and pin down the expressive power of TLC*. We first show that TLC* is expressively stronger than TLC for general trace alphabets, where the underlying dependency relation is transitive. We then briefly review monadic second-order for traces and show that TLC* can be embedded into it. It then follows that the satisfiability problem remains decidable for TLC*.

To initiate developments, let (Σ, I) be any trace alphabet with $\{a, b, c\} \subseteq \Sigma$ such that $a\ D\ c$ and $c\ D\ b$ but $a\ I\ b$. Consider $L = [abcabc]^* \subseteq \mathbb{TR}(\Sigma, I)$ introduced in the previous section.

Lemma 9.6.1 *L is definable in* $\mathrm{TLC}^*(\Sigma, I)$.

Proof: Our proof will in fact show that the future fragment of TLC with only *one* future chain quantifier of TLC* suffices to express L. First define

$$\begin{aligned}\alpha_{[abc]^*} \;=\; & AG(\bigwedge_{d\in\Sigma-\{a,b,c\}} \neg p_d) \wedge EX(p_a \wedge EX(p_c)) \wedge EX(p_b \wedge EX(p_c)) \wedge \\ & AG(p_c \wedge EX(\mathrm{tt}) \rightarrow EX(p_a \wedge EX(p_c)) \wedge EX(p_b \wedge EX(p_c)).\end{aligned}$$

It is easy to see that $T \models \alpha_{[abc]^*}$ if and only if $T \in [abc]^*$. We will then use existence of "zig-zagging" future causal chains to restrict to $[abcabc]^* \subsetneq [abc]^*$ below. Define the future chain formula $\varphi_{(acbc)^*}$ as follows:

$$\varphi_{(acbc)^*} = p_a \wedge G(p_a \rightarrow X(p_c \wedge X(p_b \wedge X(p_c \wedge (\neg X\mathrm{tt} \vee Xp_a))))).$$

It's easy to see that $T, e \models E(\varphi_{(acbc)^*})$ if and only if there exists a future causal chain ρ rooted at e such that $\lambda(\rho) \in (acbc)^* \subseteq \Sigma^*$. The statement of the lemma now follows by taking $\alpha_L = \alpha_{[abc]^*} \wedge (\neg EX\mathrm{tt} \vee EX(E(\varphi_{(acbc)^*})))$. □

Putting all the pieces together, we can now state and prove the main result of the chapter.

Theorem 9.6.2 *Let (Σ, I) be any trace alphabet. Then*

(i) $\mathrm{TLC}(\Sigma, I) = \mathrm{TLC}^*(\Sigma, I)$ *if D is transitive.*

(ii) $\mathrm{TLC}(\Sigma, I) \subsetneq \mathrm{TLC}^*(\Sigma, I)$ *if D is not transitive.*

Proof: Obviously $\mathrm{TLC}(\Sigma, I) \subseteq \mathrm{TLC}^*(\Sigma, I)$, so (ii) follows easily from Lemmas 9.5.1 and 9.6.1 as $(a, c), (c, b) \in D$ but $(a, b) \notin D$ witness that D is not transitive. Hence it suffices to prove (i).

Let (Σ, I) be a trace alphabet with D transitive, i.e. the graph (Σ, D) is a disjoint union of cliques $\{C_i\}_{i=1}^n$. Thus any trace $T \in \mathbb{TR}(\Sigma, I)$ consists of disjoint C_i-labeled causal chains connected only initially by $\perp$. We can then define three mutually inductive translations $||\cdot||_{ev}$, $||\cdot||_{ch}^+$, and $||\cdot||_{ch}^-$ converting event formulas, future chain formulas and past chain formulas, respectively, of $\mathrm{TLC}^*(\Sigma, I)$ to formulas of $\mathrm{TLC}(\Sigma, I)$ as follows:

- $||p_a||_{ev} = p_a$.
- $||\neg\alpha||_{ev} = \neg||\alpha||_{ev}$ and $||\alpha_1 \vee \alpha_2||_{ev} = ||\alpha_1||_{ev} \vee ||\alpha_2||_{ev}$.
- $||\mathrm{co}(\alpha)||_{ev} = \mathrm{co}(||\alpha||_{ev})$.
- $||E(\varphi)||_{ev} = ||\varphi||^+_{ch}$ and $||E^-(\psi)||_{ev} = ||\psi||^-_{ch}$.
- $||\alpha||^+_{ch} = ||\alpha||_{ev}$ and $||\alpha||^-_{ch} = ||\alpha||_{ev}$.
- $||\neg\varphi||^+_{ch} = \neg||\varphi||^+_{ch}$ and $||\neg\psi||^-_{ch} = \neg||\psi||^-_{ch}$.
- $||\varphi_1 \vee \varphi_2||^+_{ch} = ||\varphi_1||^+_{ch} \vee ||\varphi_2||^+_{ch}$ and $||\psi_1 \vee \psi_2||^-_{ch} = ||\psi_1||^-_{ch} \vee ||\psi_2||^-_{ch}$.
- $||X\varphi||^+_{ch} = EX(||\varphi||^+_{ch})$.
- $||X^-\psi||^-_{ch} = EX^-(||\psi||^-_{ch})$.
- $||\varphi_1 U \varphi_2||^+_{ch} = EU(||\varphi_1||^+_{ch}, ||\varphi_2||^+_{ch})$.
- $||\psi_1 U^- \psi_2||^-_{ch} = EU^-(||\psi_1||^-_{ch}, ||\psi_2||^-_{ch})$.

A routine induction now shows that $T, e \models \alpha$ if and only if $T, e \models ||\alpha||_{ev}$ for each $\alpha \in \mathrm{TLC}^*(\Sigma, I)$. The required conclusion follows as one readily verifies that $||\alpha||_{ev} \in \mathrm{TLC}(\Sigma, I)$. □

Theorem 9.6.2 states that TLC is expressively equivalent to TLC* exactly when the underlying trace alphabet (Σ, I) gives rise to a transitive dependency relation D. This corresponds to the case where the static notion of dependency of system actions allows a decomposition of the system into a number of autonomous computing processes with no interprocess communication. In this sense, TLC yields a lower bound of the expressiveness of TLC*. In the following we provide an upper bound in terms of monadic second-order logic.

The *monadic second-order theory of traces over* (Σ, I) is denoted $\mathrm{MSO}(\Sigma, I)$. Its vocabulary consists of a family of unary predicates $\{Q_a\}_{a\in\Sigma}$, one for each $a \in \Sigma$; a binary predicate $\leq$; a binary predicate $\in$; a countable supply of individual variables $Var = \{x, y, z, \ldots\}$; a countable supply of set variables (i.e. monadic predicate variables) $SVar = \{X, Y, Z, \ldots\}$. The formulas Ω of $\mathrm{MSO}(\Sigma, I)$ are then given as:

$$\mathrm{MSO}(\Sigma, I) ::= Q_a(x) \mid x \in X \mid x \leq y \mid \neg\Omega \mid \Omega_1 \vee \Omega_2 \mid (\exists x)\Omega \mid (\exists X)\Omega, \quad a \in \Sigma.$$

A structure for $\mathrm{MSO}(\Sigma, I)$ is a trace $T = (E, \leq, \lambda)$ of $\mathbb{TR}(\Sigma, I)$. Let $\mathcal{I}$ be an interpretation of the variables with $\mathcal{I} : Var \to E$ and $\mathcal{I} : SVar \to 2^E$. Then the notion of T being a model of Ω under the interpretation $\mathcal{I}$, denoted $T \models_{\mathcal{I}} \Omega$, is defined in the expected manner. In particular, $T \models_{\mathcal{I}} Q_a(x)$ iff $\lambda(\mathcal{I}(x)) = a$; $T \models_{\mathcal{I}} x \in X$ iff $\mathcal{I}(x) \in \mathcal{I}(X)$; and $T \models_{\mathcal{I}} x \leq y$ iff $\mathcal{I}(x) \leq \mathcal{I}(y)$. For convenience, we use $\leq$ to denote both the predicate symbol in the logic and the corresponding causality relation in the model T. We will freely use the standard abbreviations for derived propositional connectives such as $\wedge, \Rightarrow$,

universal quantifications such as $(\forall x)(\Omega(x))$, together with the derived ordering relations $x \neq y$, $x < y$, $x \lessdot y$, $x \leq y \leq z$, $X \subseteq Y$, etc.

As usual, Ω is a sentence if there are no free occurrences of individual or set variables in Ω. With each sentence Ω we can associate a trace language $L_\Omega = \{T \in \mathbb{TR}(\Sigma, I) \mid T \models \Omega\}$. We say that $L \subseteq \mathbb{TR}(\Sigma, I)$ is $\mathrm{MSO}(\Sigma, I)$-definable if there exists a sentence φ such that $L_\Omega = L$.

The *first-order theory of traces over* (Σ, I) is denoted $\mathrm{FO}(\Sigma, I)$ and is obtained from $\mathrm{MSO}(\Sigma, I)$ by abolishing the monadic second-order quantifications from the logic. The semantics and notions of first-order definability are carried over in the obvious manner.

We now wish to argue that any language definable by a formula of TLC^* is definable by a sentence of MSO.

Lemma 9.6.3 *Let (Σ, I) be any trace alphabet. Then* $\mathrm{TLC}^*(\Sigma, I) \subseteq \mathrm{MSO}(\Sigma, I)$.

Proof: We show by structural induction over formulas of $\mathrm{TLC}^*(\Sigma, I)$ that for each event formula α (future chain formula φ, past chain formula ψ, respectively), there exists a formula Ω_α of $\mathrm{MSO}(\Sigma, I)$ with free variable x (formulas Ω^+_φ and Ω^-_ψ with free set variable X, respectively) such that for each trace $T = (E, \leq, \lambda)$ of $\mathbb{TR}(\Sigma, I)$, event $e \in E$, and causal chain ρ;

(i) $T, e \models \alpha$ if and only if $T \models_{\mathcal{I}} \Omega_\alpha(x)$ with $\mathcal{I}(x) = e$.

(ii) $T, \rho \models \varphi$ if and only if $T \models_{\mathcal{I}} \Omega^+_\varphi(X)$ with $\mathcal{I}(X) = \rho$.

(iii) $T, \rho \models \psi$ if and only if $T \models_{\mathcal{I}} \Omega^-_\psi(X)$ with $\mathcal{I}(X) = \rho$.

This suffices to prove the statement of the lemma, as we can then for a given formula α of TLC^*, define an equivalent formula Ω of MSO with $L_\Omega = L(\alpha)$ via $\Omega = (\exists x)(\neg(\exists y)(y \leq x) \wedge \Omega_\alpha(x))$.

To complete the proof we only present by mutual induction the formulas Ω_α and Ω_φ maintaining the correspondence (i) and (ii) above. The formulas Ω^-_ψ witnessing (iii) follow in a very similar manner. In more detail, for event formulas we inductively define:

- $\Omega_{p_a}(x) = Q_a(x)$.
- $\Omega_{\neg\alpha}(x) = \neg\Omega_\alpha(x)$.
- $\Omega_{\alpha_1 \vee \alpha_2}(x) = \Omega_{\alpha_1}(x) \vee \Omega_{\alpha_2}(x)$.
- $\Omega_{\mathrm{co}(\alpha)}(x) = (\exists y)(\neg(x \leq y) \wedge \neg(y \leq x) \wedge \Omega_\alpha(y))$.
- $\Omega_{E(\varphi)}(x) = (\exists X)(\mathrm{Chain}^+(X, x) \wedge \Omega^+_\varphi(X))$.

We have here used the fact that the MSO formula

$$\begin{aligned}\mathrm{Chain}^+(X, x) = (\forall y, z)(y, z \in X \wedge y \neq z \Rightarrow \\ (y < z \vee z < y) \wedge \\ [\neg(y \lessdot z \vee z \lessdot y) \Rightarrow \\ (\exists w)(w \in X \wedge (y < w < z \vee z < w < y))]) \\ \wedge x \in X \wedge (\forall y)(y \in X \Rightarrow x \leq y)\end{aligned}$$

expresses that the set X denotes a future causal chain rooted at the event denoted by the variable x.

We will use this abbreviation extensively in defining the translation for future chain formulas as follows:

- $\Omega^+_\alpha(X) = (\exists x)(\mathrm{Chain}(X,x)^+ \wedge \Omega_\alpha(x))$.
- $\Omega^+_{\neg\varphi}(X) = \neg\Omega^+_\varphi(X)$.
- $\Omega^+_{\varphi_1\vee\varphi_2}(X) = \Omega^+_{\varphi_1}(X) \vee \Omega^+_{\varphi_2}(X)$.
- $\Omega^+_{X\varphi}(X) = (\exists x, y, Y)(\mathrm{Chain}^+(X,x) \wedge x \lessdot y \wedge \mathrm{Chain}^+(Y,y) \wedge Y \subseteq X \wedge \Omega^+_\varphi(Y))$.
- $\Omega^+_{\varphi_1 U \varphi_2}(X) = (\exists x, y, Y)(\mathrm{Chain}^+(X,x) \wedge x \leq y \wedge \mathrm{Chain}^+(Y,y) \wedge Y \subseteq X \wedge \Omega^+_{\varphi_2}(Y) \wedge (\forall z, Z)(x \leq z < y \wedge \mathrm{Chain}^+(Z,z) \wedge Z \subseteq X \Rightarrow \Omega^+_{\varphi_1}(Z)))$.

It's not hard to verify that the translations satisfy (i) and (ii). □

As $\mathrm{MSO}(\Sigma, I)$ is known to be decidable [14, 32], we obtain the following as an immediate consequence of Lemma 9.6.3.

Corollary 9.6.4 *The satisfiability problem for* TLC^* *is decidable.*

9.7 Conclusion

We conclude by summarizing what is currently known about the relative expressive powers of the various linear time temporal logics over traces. Surprisingly, this area has turned out to be far more challenging than its interleaving counterpart, mainly because the logics are parameterized by trace alphabets which, even with the same set of underlying system actions Σ, might have very different independence structures dictated by the relation $I \subseteq \Sigma \times \Sigma$.

Hence, for a logic A to be *at least as expressive* as B we will demand that any property $L \subseteq \mathbb{TR}(\Sigma, I)$ expressible by a formula of $B(\Sigma, I)$ is also expressible by a formula of $A(\Sigma, I)$, for *every* trace alphabet (Σ, I). A is then *more expressive than* B if A is at least as expressive as B and there exists *some* trace alphabet (Σ, I) and a property $L \subseteq \mathbb{TR}(\Sigma, I)$ such that L is definable in $A(\Sigma, I)$ but *not* in $B(\Sigma, I)$. The notions of expressive equivalence and noninclusion should now be obvious from these remarks.

A quick overview is displayed in Figure 9.2. A dotted (solid) arrow from A to B indicates that B is at least as expressive as (strictly more expressive than) A, while a squiggled line from A to B denotes that A is *not* included in B.

The two most important classes consist of MSO and its first-order fragment, FO. There exist two logics expressively equivalent to MSO; Niebert's νTrPTL [102] and $\mu-$co by Walukiewicz [150]. Both logics are based on fixed point operators, and the latter logic can be viewed as the μ-calculus [75] with

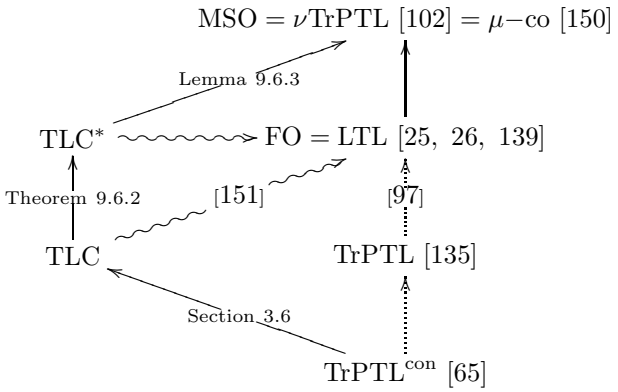


Figure 9.2: Overview of relative expressive powers.

some additional operators (reminiscent of the present co-operator of TLC) interpreted directly over the Hasse diagram of traces. Both logics are decidable in exponential time.

It is a classical fact that FO is strictly weaker than MSO over the domain of sequences, which is reflected in the present setting with $I = \emptyset$. The relationship between first-order logic for strings and first-order logic for traces is very tight, as a given trace language is definable in FO over traces if and only if its set of linearizations is definable in FO over sequences [32]. This result suggests that one should look for an elementary-time (preferably exponential-time) logic equal in expressive power to the first-order theory of traces as such a logic (if it exists!) would capture precisely [32, 70] the properties of LTL amenable to partial-order verification. With this in mind, Thiagarajan proposed the first linear temporal logic for traces, TrPTL [135]. It is known [97] that TrPTL is no more expressive than FO, but it is — perhaps surprisingly — still an open problem whether TrPTL is expressively equivalent to FO.

Thiagarajan and Walukiewicz later defined a linear time temporal logic, LTrL [139], expressively complete with respect to first-order logic for traces. Their logic was later refined [25, 26] to show that LTL interpreted on traces is expressively equivalent to FO, yielding an extension of Kamp's Theorem [70] to the richer setting of traces. Unfortunately, the satisfiability problems for these logics have *nonelementary lower bounds* [149]!

Theorem 9.6.2 and Lemma 9.6.3 have been applied to place TLC* in Figure 9.2. The classical result that FO is weaker than MSO, in conjunction with the fact that TLC$^*(\Sigma, \emptyset)$ is equivalent to FO$(\Sigma, \emptyset)$, yields a strict inclusion of TLC* into MSO. A simple corollary of Lemma 9.6.1 is that TLC* is not included in FO whenever D is transitive, because our example property L is easily seen not be first-order definable. Hence, TLC* does not have a completely resolved relationship to FO.

Neither does TLC, but progress has been made very recently [151], as Walu-

kiewicz has identified an example of a property definable in TLC, but *not* in FO. It is still an open problem, however, whether TLC can express all properties of FO, though it is believed not to be the case. Similarly, the relative expressive power of TLC and the seminal TrPTL is still completely open. For the syntactic subset $\text{TrPTL}^{\text{con}}$ [65] which is not known to be weaker than TrPTL itself, it is not hard to devise a translation into TLC, as mentioned in Section 3.6. The example property of Walukiewicz then witnesses that this inclusion must be strict.

It remains an interesting open problem for future research to give a precise characterization of the expressive power of TLC^*. One possibility to be investigated is in terms of MSO with set quantifications restricted to chains along the lines of [50, 92], which achieves a similar goal for the branching time logic CTL^*. Another possibility would be to consider (a suitably variable-confined subset of) FO with the transitive-closure operator [67].

Finally, the present proof of decidability of the satisfiability problem for TLC^*, is indirect and essentially relies upon the decidability of monadic second-order logic [14], thus yielding a nonelementary upper bound. It seems to be a challenging open problem to pin down the precise complexity of the satisfiability problem for TLC^* and, preferably, to give a direct construction in terms of automata which operates in elementary time.

Chapter 10

Regular Collections of Message Sequence Charts

Contents

Message Sequence Charts (MSCs) are an attractive visual formalism used during the early stages of design in domains such as telecommunication software. A popular mechanism for generating a collection of MSCs is a Hierarchical Message Sequence Chart (HMSC). However, not all HMSCs describe collections of MSCs that can be "realized" as a finite-state device. Our main goal is to pin down this notion of realizability. We propose an independent notion of *regularity* for collections of MSCs and explore its basic properties. In particular, we characterize regular collections of MSCs in terms of finite-state distributed automata called bounded message-passing automata, in which a set of sequential processes communicate with each other asynchronously over bounded FIFO channels. We also provide a logical characterization in terms of a natural monadic second-order logic interpreted over MSCs. It turns out that realizable collections of MSCs as specified by HMSCs constitute a strict subclass of the regular collections of MSCs.

10.1 Introduction

Message Sequence Charts (MSCs) are an appealing visual formalism often used to capture system requirements in the early stages of design. They are par-

ticularly suited for describing scenarios for distributed telecommunication software [68, 121]. They have also been called timing sequence diagrams, message flow diagrams and object interaction diagrams and are used in a number of software engineering methodologies [9, 52, 121]. In its basic form, an MSC depicts the exchange of messages between the processes of a distributed system along a single partially-ordered execution. A collection of MSCs is used to capture the scenarios that a designer might want the system to exhibit (or avoid).

Given the requirements in the form of a collection of MSCs, one can hope to do formal analysis and discover design errors at an early stage. A natural question in this context is to identify when a collection of MSCs is amenable to formal analysis. A related issue is how to represent such collections.

A standard way to generate a collection of MSCs is to use a Hierarchical Message Sequence Chart (HMSC) [84]. An HMSC is a finite directed graph in which each node is labelled, in turn, by an HMSC. The labels on the nodes are not permitted to refer to each other. From an HMSC we can derive an equivalent Message Sequence Graph (MSG) [3] by flattening out the hierarchical labelling to obtain a graph where each node is labelled by a simple MSC. An MSG defines a collection of MSCs obtained by concatenating the MSCs labelling each path from an initial vertex to a terminal vertex. Though HMSCs provide more succinct specifications than MSGs, they are only as expressive as MSGs. Thus, one often restricts one's attention to characterizing structural properties of MSGs rather than of HMSCs [5, 98, 100].

In [5], it is shown that *locally synchronized* MSGs define reasonable collections of MSCs—the collection of MSCs generated by a locally synchronized MSG can be represented as a regular string language. Thus, behaviours captured by locally synchronized MSGs can, in principle, be realized as finite-state automata. In general, the collection of MSCs defined by an *arbitrary* MSG is not realizable in this sense. A characterization of the collections of MSCs definable using locally synchronized MSGs is provided in Chapter 11.

The main goal of this chapter is to pin down this notion of realizability in terms of a notion of *regularity* for collections of MSCs. One consequence of our study is that our definition of regularity provides a general and robust setting for studying collections of MSCs. A second consequence, which follows from the results in Chapter 11, is that locally synchronized MSGs define a strict subclass of regular collections of MSCs. A final consequence is that our notion addresses an important issue raised in [23]; namely, how to convert requirements as specified by MSCs into distributed, state-based specifications.

Another motivation for focussing on regularity is that this notion has turned out to be very fruitful in a variety of contexts including finite (and infinite) strings, trees and restricted partial orders known as Mazurkiewicz traces [27, 140]. In all these settings there is a representation of regular collections in terms of finite-state devices. There is also an accompanying monadic second-order logic that usually induces temporal logics using which one can reason about such collections [140]. One can then develop automated model-checking procedures for verifying properties specified in these temporal logics. In this context, the associated finite-state devices representing the regular collections

often play a very useful role [147].

We show here that our notion of regular MSC languages fits in nicely with a related notion of a finite-state device, as also a monadic second-order logic. We fix a finite set of processes $\mathcal{P}$ and consider $\mathbb{MSC}$, the universe of MSCs defined over the set $\mathcal{P}$. An MSC in $\mathbb{MSC}$ can be viewed as a partial order labelled using a finite alphabet Σ that is canonically fixed by $\mathcal{P}$. We say that $L \subseteq \mathbb{MSC}$ is regular if the set of all linearizations of all members of L constitutes a regular subset of Σ^*. A crucial point is that the universe $\mathbb{MSC}$ is itself *not* regular according to our definition, unlike the classical setting of strings (or trees or Mazurkiewicz traces). This fact has a strong bearing on the automata-theoretic and logical formulations in our work.

It turns out that regular MSC languages can be stratified using the concept of *bounds*. An MSC is said to be B-bounded for a natural number B if at every "prefix" of the MSC and for every pair of processes (p, q) there are at most B messages that p has sent to q that have yet to be received by q. An MSC language is B-bounded if every member of the language is B-bounded. Fortunately, for every regular MSC language L we can effectively compute a (minimal) bound B such that L is B-bounded. This leads to our automaton model called B-bounded message-passing automata. The components of such an automaton correspond to the processes in $\mathcal{P}$. These components communicate with each other over (potentially unbounded) FIFO channels. We say that a message-passing automaton is B-bounded if, during its operation, it is never the case that a channel contains more than B messages. We establish a precise correspondence between B-bounded message-passing automata and B-bounded regular MSC languages. In a similar vein, we formulate a natural monadic second-order logic $\mathrm{MSO}(\mathcal{P}, B)$ interpreted over B-bounded MSCs. We then show that B-bounded regular MSC languages are exactly those that are definable in $\mathrm{MSO}(\mathcal{P}, B)$.

In related work, a number of studies are available that are concerned with individual MSCs in terms of their semantics and properties [3, 78]. As pointed out earlier, a nice way to generate a collection of MSCs is to use an MSG. A variety of algorithms have been developed for MSGs in the literature—for instance, pattern matching [79, 98, 100] and detection of process divergence and non-local choice [8]. A systematic account of the various model-checking problems associated with MSGs and their complexities is given in [5].

In this chapter, we confine our attention to *finite* MSCs. The issues investigated here have, at present, no counterparts in the infinite setting. We feel, however, that our results will serve as a launching pad for a similar account concerning infinite MSCs. This should then lead to the design of appropriate temporal logics and automata-theoretic solutions (based on message-passing automata) to model-checking problems for these logics.

The chapter is organized as follows. In the next section we introduce MSCs and regular MSC languages. In Section 10.3 we establish our automata-theoretic characterization and, in Section 10.4, the logical characterization. While doing so, we borrow one basic result and a couple of proof techniques from the theory of Mazurkiewicz traces [27]. However, we need to modify some of these techniques in a non-trivial way (especially in the setting of automata) due to the

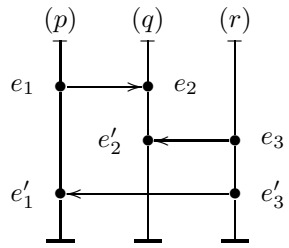


Figure 10.1: An example MSC over $\{p, q, r\}$.

asymmetric flow of information via messages in the MSC setting, as opposed to the symmetric information flow via handshake communication in the trace setting. Due to lack of space, we provide only proof sketches. More details are available in [56].

10.2 Regular MSC Languages

Through the rest of the chapter, we fix a finite set of processes (or agents) $\mathcal{P}$ and let p, q, r range over $\mathcal{P}$. For each $p \in \mathcal{P}$ we define $\Sigma_p \stackrel{\text{def}}{=} \{p!q \mid p \neq q\} \cup \{p?q \mid p \neq q\}$ to be the set of communication actions in which p participates. The action $p!q$ is to be read as *p sends to q* and the action $p?q$ is to be read as *p receives from q*. At our level of abstraction, we shall not be concerned with the actual messages that are sent and received. We will also not deal with the internal actions of the agents. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$ and let a, b range over Σ. We also denote the set of *channels* by $Ch = \{(p, q) \mid p \neq q\}$ and let c, d range over Ch.

A Σ-labelled poset is a structure $M = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $e \in E$ we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \leq e\}$. For $p \in \mathcal{P}$ and $a \in \Sigma$, we set $E_p \stackrel{\text{def}}{=} \{e \mid \lambda(e) \in \Sigma_p\}$ and $E_a \stackrel{\text{def}}{=} \{e \mid \lambda(e) = a\}$, respectively. For each $c \in Ch$, we define the relation $R_c \stackrel{\text{def}}{=} \{(e, e') \mid \lambda(e) = p!q, \lambda(e') = q?p \text{ and } |\downarrow e \cap E_{p!q}| = |\downarrow e' \cap E_{q?p}|\}$. Finally, for each $p \in \mathcal{P}$, we define the relation $R_p \stackrel{\text{def}}{=} (E_p \times E_p) \cap \leq$.

An MSC (over $\mathcal{P}$) is a *finite* Σ-labelled poset $M = (E, \leq, \lambda)$ that satisfies the following conditions:

(i) Each R_p is a linear order.

(ii) If $p \neq q$ then $|E_{p!q}| = |E_{q?p}|$.

(iii) $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$ where $R_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} R_p$ and $R_{Ch} = \bigcup_{c \in Ch} R_c$.

In diagrams, the events of an MSC are presented in *visual order*. The events of each process are arranged in a vertical line and the members of the relation R_{Ch} are displayed as horizontal or downward-sloping directed edges. We illustrate the idea with an example in Figure 10.1. Here $\mathcal{P} = \{p, q, r\}$. For $x \in \mathcal{P}$,

the events in E_x are arranged along the line labelled (x) with smaller (relative to $\leq$) events appearing above the larger events. The R_{Ch}-edges across agents are depicted by horizontal edges—for instance $e_3\ R_{(r,q)}\ e'_2$. The labelling function λ is easy to extract from the diagram—for example, $\lambda(e'_3) = r!p$ and $\lambda(e_2) = q?p$.

We define regular MSC languages in terms of their linearizations. For the MSC $M = (E, \leq, \lambda)$, let $lin(M) \stackrel{\text{def}}{=} \{\lambda(\pi) \mid \pi \text{ is a linearization of } (E, \leq)\}$. By abuse of notation, we have used λ to also denote the natural extension of λ to E^*. The string $p!q\ r!q\ q?p\ q?r\ r!p\ p?r$ is a linearization of the MSC in Figure 10.1.

In the literature [3, 100] one sometimes considers a more generous notion of linearization where two *adjacent* receive actions in a process corresponding to messages from *different* senders are deemed to be causally independent. For instance, $p!q\ r!q\ q?r\ q?p\ r!p\ p?r$ would also be a valid linearization of the MSC in Figure 10.1. All our results go through with suitable modifications even in the presence of this more generous notion of linearization.

Henceforth, we will identify an MSC with its isomorphism class. We let $\mathbb{MSC}(\mathcal{P})$ be the set of MSCs over $\mathcal{P}$. An MSC language $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$ is said to *regular* if $\bigcup\{lin(M) \mid M \in \mathcal{L}\}$ is a regular subset of Σ^*. We note that the entire set $\mathbb{MSC}(\mathcal{P})$ is not regular by this definition.

To directly characterize the subsets of Σ^* that correspond to regular MSC languages, we proceed as follows. Let $Com \stackrel{\text{def}}{=} \{(p!q, q?p) \mid (p, q) \in Ch\}$. For $\tau \in \Sigma^*$ and $a \in \Sigma$, let $|\tau|_a$ denote the number of times a appears in τ. We say that $\sigma \in \Sigma^*$ is *proper* if for every prefix τ of σ and every pair $(a, b) \in Com$, $|\tau|_a \geq |\tau|_b$. We say that σ is *complete* if σ is proper and $|\sigma|_a = |\sigma|_b$ for every $(a, b) \in Com$. Next we define a *context-sensitive* independence relation $I \subseteq \Sigma^* \times (\Sigma \times \Sigma)$ as follows: $(\sigma, a, b) \in I$ if σab is proper, $a \in \Sigma_p$ and $b \in \Sigma_q$ for distinct processes p and q, and if $(a, b) \in Com$ then $|\sigma|_a > |\sigma|_b$. Observe that if $(\sigma, a, b) \in I$ then $(\sigma, b, a) \in I$.

Let $\Sigma^\circ \stackrel{\text{def}}{=} \{\sigma \mid \sigma \in \Sigma^* \text{ and } \sigma \text{ is complete}\}$. We then define $\sim \subseteq \Sigma^\circ \times \Sigma^\circ$ to be the least equivalence relation such that if $\sigma = \sigma_1 ab \sigma_2$, $\sigma' = \sigma_1 ba \sigma_2$ and $(\sigma_1, a, b) \in I$ then $\sigma \sim \sigma'$. It is important to note that $\sim$ is defined over Σ° (and not Σ^*). It is easy to verify that for each $M \in \mathbb{MSC}(\mathcal{P})$, $lin(M)$ is a subset of Σ° and is in fact a $\sim$-equivalence class over Σ°.

We define $L \subseteq \Sigma^*$ to be a *regular string MSC language* if there exists a regular MSC language $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$ such that $L = \bigcup\{lin(M) \mid M \in \mathcal{L}\}$. It is easy to see that $L \subseteq \Sigma^*$ is a regular string MSC language if and only if L is a regular subset of Σ^*, every word in L is complete and L is $\sim$-closed (that is, for each $\sigma \in L$, if $\sigma \in L$ and $\sigma \sim \sigma'$ then $\sigma' \in L$). Clearly regular MSC languages and regular string MSC languages represent each other. Hence, abusing terminology, we will write "regular MSC language" to mean "regular string MSC language". From the context, it should be clear whether we are working with MSCs from $\mathbb{MSC}(\mathcal{P})$ or complete words over Σ^*.

Given a regular subset $L \subseteq \Sigma^*$, we can decide whether L is a regular MSC language. We say that a state s in a finite-state automaton is *live* if there is a path from s to a final state. Let $\mathcal{A} = (S, \Sigma, s_{in}, \delta, F)$ be the minimal DFA

representing L. Then it is not difficult to see that L is a regular MSC language if and only if we can associate with each live state $s \in S$, a channel-capacity function $\mathcal{K}_s : Ch \to \mathbb{N}$ that satisfies the following conditions.

(i) If $s \in \{s_{in}\} \cup F$ then $\mathcal{K}_s(c) = 0$ for every $c \in Ch$.

(ii) If s, s' are live states and $\delta(s, p!q) = s'$ then $\mathcal{K}_{s'}((p,q)) = \mathcal{K}_s((p,q))+1$ and $\mathcal{K}_{s'}(c) = \mathcal{K}_s(c)$ for every $c \neq (p,q)$.

(iii) If s, s' are live states and $\delta(s, q?p) = s'$ then $\mathcal{K}_s((p,q)) > 0$, $\mathcal{K}_{s'}((p,q)) = \mathcal{K}_s((p,q))-1$ and $\mathcal{K}_{s'}(c) = \mathcal{K}_s(c)$ for every $c \neq (p,q)$.

(iv) Suppose $\delta(s,a) = s_1$ and $\delta(s_1, b) = s_2$ with $a \in \Sigma_p$ and $b \in \Sigma_q$, $p \neq q$. If $(a,b) \notin Com$ or $\mathcal{K}_s((p,q)) > 0$, there exists s_1' such that $\delta(s,b) = s_1'$ and $\delta(s_1', a) = s_2$.

These conditions can be checked in time linear in the size of δ. We conclude this section by introducing the notion of B-bounded MSC languages. Let $B \in \mathbb{N}$ be a natural number. We say that a complete word σ is B-bounded if for each prefix τ of σ and for each channel $(p,q) \in Ch$, $|\tau|_{p!q} - |\tau|_{q?p} \leq B$. We say that $L \subseteq \Sigma^\circ$ is B-bounded if every word $\sigma \in L$ is B-bounded. Let L be a regular MSC language and let $\mathcal{A} = (S, \Sigma, s_{in}, \delta, F)$ be its minimal DFA, as described above, with capacity functions $\{\mathcal{K}_s\}_{s \in S}$. Let $B_L \stackrel{\text{def}}{=} \max_{s \in S, c \in Ch} \mathcal{K}_s(c)$. Then it is easy to see that L is B_L-*bounded* and that B_L can be effectively computed from $\mathcal{A}$. Finally, we shall say that the MSC M is B-bounded if every string in $lin(M)$ is B-bounded. A collection of MSCs is B-bounded if every member of the collection is B-bounded.

10.3 An Automata-Theoretic Characterization

Recall that the set of processes $\mathcal{P}$ determines the communication alphabet Σ and that for $p \in \mathcal{P}$, Σ_p denotes the actions in which process p participates.

Definition 10.3.1 A *message-passing automaton* over Σ is a structure $\mathcal{M} = (\{\mathcal{A}_p\}_{p \in \mathcal{P}}, \Delta, s_{in}, F)$ where

- Δ is a finite alphabet of messages.
- Each component $\mathcal{A}_p$ is of the form $(S_p, \longrightarrow_p)$ where
  - S_p is a finite set of p-local states.
  - $\longrightarrow_p \subseteq S_p \times \Sigma_p \times \Delta \times S_p$ is the p-local transition relation.
- $s_{in} \in \prod_{p \in \mathcal{P}} S_p$ is the global initial state.
- $F \subseteq \prod_{p \in \mathcal{P}} S_p$ is the set of global final states.

The local transition relation $\longrightarrow_p$ specifies how process p sends and receives messages. The transition $(s, p!q, m, s')$ specifies that when p is in the state s, it can send the message m to q by executing the action $p!q$ and move to the state s'. The message m is, as a result, appended to the queue in channel (p, q). Similarly, the transition $(s, p?q, m, s')$ signifies that in the state s, the process p can receive the message m from q by executing the action $p?q$ and move to the state s'. The message m is removed from the head of the queue in channel (q, p).

The set of global states of $\mathcal{M}$ is given by $\prod_{p \in \mathcal{P}} S_p$. For a global state s, we let s_p denote the pth component of s. A *configuration* is a pair (s, χ) where s is a global state and $\chi : Ch \to \Delta^*$ is the *channel state* that specifies the queue of messages currently residing in each channel c. The *initial configuration* of $\mathcal{M}$ is $(s_{in}, \chi_\varepsilon)$ where $\chi_\varepsilon(c)$ is the empty string ε for every channel c. The set of *final configurations* of $\mathcal{M}$ is $F \times \{\chi_\varepsilon\}$.

We now define the set of reachable configurations $Conf_{\mathcal{M}}$ and the global transition relation $\Longrightarrow \subseteq Conf_{\mathcal{M}} \times \Sigma \times Conf_{\mathcal{M}}$ inductively as follows:

- $(s_{in}, \chi_\varepsilon) \in Conf_{\mathcal{M}}$.
- Suppose $(s, \chi) \in Conf_{\mathcal{M}}$, (s', χ') is a configuration and $(s_p, p!q, m, s'_p) \in \longrightarrow_p$ such that the following conditions are satisfied:
  - $r \neq p$ implies $s_r = s'_r$ for each $r \in \mathcal{P}$.
  - $\chi'((p, q)) = \chi((p, q)) \cdot m$ and for $c \neq (p, q)$, $\chi'(c) = \chi(c)$.

  Then $(s, \chi) \stackrel{p!q}{\Longrightarrow} (s', \chi')$ and $(s', \chi') \in Conf_{\mathcal{M}}$.
- Suppose $(s, \chi) \in Conf_{\mathcal{M}}$, (s', χ') is a configuration and $(s_p, p?q, m, s'_p) \in \longrightarrow_p$ such that the following conditions are satisfied:
  - $r \neq p$ implies $s_r = s'_r$ for each $r \in \mathcal{P}$.
  - $\chi((q, p)) = m \cdot \chi'((q, p))$ and for $c \neq (q, p)$, $\chi'(c) = \chi(c)$.

  Then $(s, \chi) \stackrel{p?q}{\Longrightarrow} (s', \chi')$ and $(s', \chi') \in Conf_{\mathcal{M}}$.

Let $\sigma \in \Sigma^*$. A run of $\mathcal{M}$ over σ is a map $\rho : \mathrm{prf}(\sigma) \to Conf_{\mathcal{M}}$ (where $\mathrm{prf}(\sigma)$ is the set of prefixes of σ) such that $\rho(\varepsilon) = (s_{in}, \chi_\varepsilon)$ and for each $\tau a \in \mathrm{prf}(\sigma)$, $\rho(\tau) \stackrel{a}{\Longrightarrow} \rho(\tau a)$. The run ρ is *accepting* if $\rho(\sigma)$ is a final configuration. We define $L(\mathcal{M}) \stackrel{\text{def}}{=} \{\sigma \mid \mathcal{M} \text{ has an accepting run over } \sigma\}$. It is easy to see that every member of $L(\mathcal{M})$ is complete and $L(\mathcal{M})$ is $\sim$-closed.

Clearly, $L(\mathcal{M})$ need not be regular. Consider, for instance, a message-passing automaton for the canonical producer-consumer system in which the producer p sends an arbitrary number of messages to the consumer q. Since we can reorder all the $p!q$ actions to be performed before all the $q?p$ actions, the queue in channel (p, q) can grow arbitrarily long. Hence, the reachable configurations of this system are not bounded and the corresponding language is not regular.

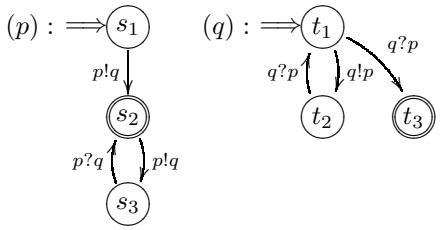


Figure 10.2: A 3-bounded message-passing automaton.

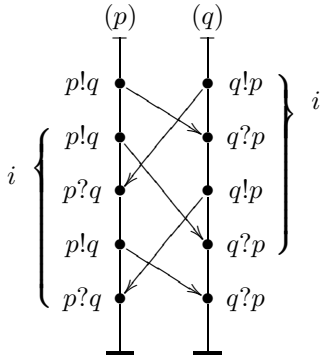


Figure 10.3: The M_i's accepted by the automaton in Figure 10.2.

For $B \in \mathbb{N}$, we say that a configuration (s, χ) of the message-passing automaton $\mathcal{M}$ is *B-bounded* if for every channel $c \in Ch$, it is the case that $|\chi(c)| \leq B$. We say that $\mathcal{M}$ is a B-bounded automaton if every reachable configuration $(s, \chi) \in Conf_{\mathcal{M}}$ is B-bounded. It is not difficult to show that given a message-passing automaton $\mathcal{M}$ and a bound $B \in \mathbb{N}$, one can decide whether or not $\mathcal{M}$ is B-bounded. Figure 10.2 depicts an example of a 3-bounded message-passing automaton with two components, p and q (the message alphabet is a singleton and hence omitted). The automaton accepts the infinite set of MSCs $\mathcal{L} = \{M_i\}_{i=0}^{\omega}$, where M_i is displayed in Figure 10.3 for $i = 2$.

In this example, the message alphabet is a singleton, and is hence omitted. The initial state is (s_1, t_1) and there is only one final state, (s_2, t_3). This automaton accepts an infinite set of MSCs, none of which can be expressed as the concatenation of two or more non-trivial MSCs. As a result, this MSC language cannot be represented using MSGs, as formulated in [5].

Lemma 10.3.2 *Let $\mathcal{M}$ be a B-bounded automaton over Σ. Then $L(\mathcal{M})$ is a B-bounded regular MSC language.*

This result follows from the definitions and it constitutes the easy half of the characterization we wish to obtain. The second half of our characterization says

that every B-bounded regular MSC language can be recognized by a B-bounded message-passing automaton. This is much harder to establish.

Let $\{\Sigma_c\}_{c \in Ch}$ be given by $\Sigma_c \stackrel{\text{def}}{=} \{p!q, q?p\}$ for $c = (p, q)$. We let $\mathcal{X} = \mathcal{P} \cup Ch$. For $a \in \Sigma$, we define the locations of a as $\text{loc}(a) \stackrel{\text{def}}{=} \{x \in \mathcal{X} \mid a \in \Sigma_x\}$. The distributed alphabet $\{\Sigma_x\}_{x \in \mathcal{X}}$ induces the Mazurkiewicz trace alphabet $(\Sigma, I_{\mathcal{X}})$, where $I_{\mathcal{X}} = \{(a, b) \mid a, b \in \Sigma, \text{loc}(a) \cap \text{loc}(b) = \emptyset\}$ is an irreflexive and symmetric independence relation. We can then define $\approx \subseteq \Sigma^* \times \Sigma^*$ to be the least equivalence relation such that if $\sigma = \sigma_1 ab \sigma_2$, $\sigma' = \sigma_1 ba \sigma_2$ and $(a, b) \in I_{\mathcal{X}}$ then $\sigma \approx \sigma'$. We say that $L \subseteq \Sigma^*$ is a *regular (Mazurkiewicz) trace language* over $(\Sigma, I_{\mathcal{X}})$ if L is a regular subset of Σ^* and L is $\approx$-closed— that is, for each $\sigma \in \Sigma^*$, if $\sigma \in L$ and $\sigma \approx \sigma'$ then $\sigma' \in L$.

Let $L \subseteq \Sigma^*$ be a regular MSC language. It is not difficult to verify that L is a regular trace language over $(\Sigma, I_{\mathcal{X}})$—the independence relation $I_{\mathcal{X}}$ corresponds to the static kernel of the context-sensitive independence relation I defined in Section 10.2.

In order to apply Zielonka's Theorem we need to first introduce asynchronous automata. An asynchronous automaton over the distributed alphabet $\{\Sigma_x\}_{x \in \mathcal{X}}$ is a structure $\mathcal{Z} = (\{S_x\}_{x \in \mathcal{X}}, \{\longrightarrow_a\}_{a \in \Sigma}, s_{in}, F)$ where each S_x is a finite set of local states of the component x. Let $S = \prod_{x \in \mathcal{X}} S_x$ denote the set of global states of $\mathcal{Z}$. Then $s_{in} \in S$ is the global initial state and $F \subseteq S$ is the set of global final states. Let $a = p!q$. Then $\longrightarrow_a \subseteq (S_p \times S_{(p,q)}) \times (S_p \times S_{(p,q)})$. The pair $((s_1, s_1'), (s_2, s_2')) \in \longrightarrow_a$ denotes the fact that the p-component in state s_1, and the channel (p, q)-component in state s_1' can together execute $p!q$ and move to the joint state (s_2, s_2'). Similarly, for a receive action $b = q?p$, $\longrightarrow_b \subseteq (S_{(p,q)} \times S_q) \times (S_{(p,q)} \times S_q)$ defines joint moves of the channel (p, q) and process q when q receives messages from p. To define the global transition relation $\longrightarrow \subseteq S \times \Sigma \times S$, we let s_x denote the xth component of the global state s. Suppose $s, s' \in S$, $a = p!q$ and $c = (p, q)$. Then $(s, a, s') \in \longrightarrow$ if $((s_p, s_c), (s_p', s_c')) \in \longrightarrow_a$ and $s_x = s_x'$ for every $x \in \mathcal{X} \setminus \{p, c\}$. Transitions of the form (s, b, s') with $b = q?p$ are defined in a similar fashion. The notions of runs and accepting runs

Zielonka's Theorem [157] asserts that from a regular trace language L, we can construct a deterministic asynchronous automaton $\mathcal{Z}$ such that $L(\mathcal{Z}) = L$. We have already observed that a regular MSC language $L \subseteq \Sigma^*$ is a regular trace language over $(\Sigma, I_{\mathcal{X}})$. It follows that from a regular MSC language $L \subseteq \Sigma^*$ we can effectively construct a deterministic asynchronous automaton $\mathcal{Z}$ over the distributed alphabet $\{\Sigma_x\}_{x \in \mathcal{X}}$ such that $L = L(\mathcal{Z})$.

Fix a B-bounded regular MSC language L and let $\mathcal{Z} = (\{S_x\}_{x \in \mathcal{X}}, \{\longrightarrow_a\}_{a \in \Sigma}, s_{in}, F)$ be a deterministic asynchronous automaton such that $L = L(\mathcal{Z})$. We claim that we can effectively transform $\mathcal{Z}$ into a B-bounded message-passing automaton $\mathcal{M}$ over Σ such that $L(\mathcal{M}) = L$.

This transformation is complicated by the following fact. In $\mathcal{Z}$, for each pair $p, q \in \mathcal{P}$, the actions $p!q$ and $q?p$ are performed by the channel component (p, q) and are hence dependent on each other in all contexts. As a result, the transition relations $\longrightarrow_{p!q}$ and $\longrightarrow_{q?p}$ do not reflect the context-sensitive inde-

pendence of the actions $p!q$ and $q?p$, even though the language accepted by $\mathcal{Z}$ is a regular MSC language and is hence closed with respect to the context-sensitive independence relation I on Σ. This means that for two inputs σ and σ' such that $\sigma \sim \sigma'$, $\mathcal{Z}$ will, in general, admit drastically different runs on σ and σ'. On the other hand, the structure of message-passing automata is such that the moves of any message-passing automaton over Σ can be reordered with respect to the independence relation I. This implies that the simulation of $\mathcal{Z}$ by $\mathcal{M}$ should not depend on the order in which independent occurrences of actions of the form $p!q$ and $q?p$ are linearized in a given input.

To get around this, we simulate the component (p, q) of $\mathcal{Z}$ in $\mathcal{M}$ using the components p and q such that for each input σ, p and q keep track the moves of (p, q) along a *canonical* reordering $\sigma' \sim \sigma$. This simulation is coordinated using the messages sent from p to q.

The key technical input for this simulation comes from [93] where it is shown how each process p in a message-passing system can use a bounded time-stamping protocol to keep track of the latest information about every other process in the system. The protocol does not add extra messages to the system. This protocol also allows each process p to locally keep track of the messages resident in each channel (p, q) for which p has not received an "acknowledgment", directly or indirectly, from q. This list of "unacknowledged" messages yields an upper bound for the number of messages currently resident in each outgoing channel from p. (A more detailed description of the time-stamping protocol is presented in [56, Appendix A].)

The desired automaton $\mathcal{M}$ will be of the form $(\{\mathcal{A}'_p\}_{p\in\mathcal{P}}, \Delta, s'_{in}, F')$, where $\mathcal{A}'_p = (S'_p, \rightsquigarrow_p)$. For each process p, each state in S'_p is of the form $\langle s_p, \bar{s}_p, \tau_p\rangle$ where s_p records a local state of p in $\mathcal{Z}$, $\bar{s}_p$ records a local state s_c in $\mathcal{Z}$ for each incoming channel $c = (q, p)$ at p, and τ is a time-stamp generated by protocol of [93].

The message alphabet is $\Delta = Ev \times \mathcal{T}$ where $Ev = \bigcup_{a\in\Sigma} \longrightarrow_a$ and $\mathcal{T}$ is the set of time-stamps used by the protocol of [93]. (Recall that $\longrightarrow_a$ is the set of a-transitions specified in $\mathcal{Z}$ for each a.) The initial state and the final states are defined in the expected manner using the initial and final states of $\mathcal{Z}$.

The transitions of $\mathcal{M}$ are arranged as follows. The tuple $(\langle s_p, \overline{s}_p, \tau_p\rangle, p!q, (e, \tau), \langle s'_p, \overline{s'}_p, \tau'_p\rangle)$ belongs to the p-local transition relation $\rightsquigarrow_p$ provided the following hold. First, $\tau = \tau'_p$ and τ'_p is the time-stamp generated from τ_p by the protocol of [93]. Let $c = (p, q)$. The e-component of the message is a move $((s_p, s_c), (s'_p, s'_c)) \in \longrightarrow_{p!q}$ for some $s_c, s'_c \in S_c$. Finally, according to τ_p there are at most $B-1$ "unacknowledged" messages in the channel c, indicating that sending this message will not violate the B-boundedness of $\mathcal{M}$.

The tuple $(\langle s_p, \overline{s}_p, \tau_p\rangle, p?q, (e, \tau'), \langle s'_p, \overline{s'}_p, \tau'_p\rangle)$ belongs to $\rightsquigarrow_p$ provided the following hold. From the time-stamp τ' on the incoming message, p collects the latest information from each process $r \in \mathcal{P}$ about new $r!p$ events that have been sent by r but not yet received by p. For each such event, the time-stamp τ' also records the move $((s_r, s_c), (s'_r, s'_c))$ guessed by r when the event occurred. Process p updates the (r, p)-component of $\overline{s}_p$ by applying this move guessed by r. If this guess is not permitted by the current state of (r, p) as recorded in $\overline{s}_p$,

p aborts. If there is more than one such $r!p$ event then p processes each of them in the order in which they were sent. Let the resulting states corresponding to the channel components $\{(r,p) \mid r \in \mathcal{P}\}$ be $\hat{s}_p$. Let $c = (q,p)$. Now, p simulates the unique $p?q$ move $(\hat{s}_c, s_p) \stackrel{p?q}{\Longrightarrow} (\overline{s}'_c, s'_p)$ of $\mathcal{Z}$ (recall that $\mathcal{Z}$ is deterministic). With this, p has updated the components s_p and $\bar{s}_p$ of its state to s'_p and $\overline{s}'_p$. Finally, it uses the time-stamps τ_p and τ' to generate a new time-stamp τ'_p as specified by the protocol of [93].

The automaton $\mathcal{M}$ is nondeterministic because each send action requires guessing a move of $\mathcal{Z}$. It is easy to show that $L(\mathcal{Z}) \subseteq L(\mathcal{M})$. To show the converse, let $\sigma \in L(\mathcal{M})$ and let ρ be an accepting run of $\mathcal{M}$ over σ. From the way $\mathcal{M}$ simulates $\mathcal{Z}$, we can show that there is a canonical reordering $\sigma' \sim \sigma$ such that there is an accepting run ρ' of $\mathcal{M}$ over σ' where ρ' is just a reordered version of ρ. The word σ' has the property that each message is received as soon as possible, subject to causality constraints. For instance, if $\sigma = p!q\ p!q\ q?p\ q?p$ then $\sigma' = p!q\ q?p\ p!q\ q?p$, and if $\sigma = p!q\ p!q\ p!r\ r?p\ r!q\ q?r\ q?p\ q?p$, then $\sigma' = \sigma$. In the second example, the messages via r ensure that p will have sent *both* messages to q before q receives the first one. From ρ' it is easy to extract an accepting run of $\mathcal{Z}$ over σ'. The language accepted by $\mathcal{Z}$ is $\sim$-closed because L is $\sim$-closed. Consequently, σ is also accepted by $\mathcal{Z}$.

Filling in the details of this proof skeleton leads to the following result.

Lemma 10.3.3 *Let $L \subseteq \Sigma^*$ be a B-bounded regular MSC language. Then there exists a B-bounded message-passing automaton $\mathcal{M}$ over Σ such that $L(\mathcal{M}) = L$.*

We say that $\mathcal{M}$ is a bounded message-passing automaton if $\mathcal{M}$ is B-bounded for some $B \in \mathbb{N}$. The main result of this section is an easy consequence of the two previous lemmas.

Theorem 10.3.4 *Let $L \subseteq \Sigma^*$. Then L is a regular MSC language if and only if there exists a bounded message-passing automaton $\mathcal{M}$ over Σ such that $L(\mathcal{M}) = L$.*

10.4 A Logical Characterization

We formulate a monadic second-order logic that characterizes regular B-bounded MSC languages for each fixed $B \in \mathbb{N}$. Thus our logic will be parameterized by a pair $(\mathcal{P}, B)$. For convenience, we fix $B \in \mathbb{N}$ through the rest of the section. As usual, we shall assume a supply of individual variables $x, y, \ldots$, a supply of set variables $X, Y, \ldots$, and a family of unary predicate symbols $\{Q_a\}_{a \in \Sigma}$. The syntax of the logic is then given by:

$$\mathrm{MSO}(\mathcal{P}, B) ::= Q_a(x) \mid x \in X \mid x \leq y \mid \neg\varphi \mid \varphi \vee \varphi \mid (\exists x)\varphi \mid (\exists X)\varphi$$

Thus the syntax does not reflect any information about B or the structural features of an MSC. These aspects will be dealt with in the semantics. Let $\mathbb{MSC}(\mathcal{P}, B)$ be the set of B-bounded MSCs over $\mathcal{P}$. The formulas of our logic

are interpreted over the members of $\mathbb{MSC}(\mathcal{P}, B)$. Let $M = (E, \leq, \lambda)$ be an MSC in $\mathbb{MSC}(\mathcal{P}, B)$ and $\mathcal{I}$ be an interpretation that assigns to each individual variable a member $\mathcal{I}(x)$ in E and to each set variable X a subset $\mathcal{I}(X)$ of E. Then $M \models_{\mathcal{I}} \varphi$ denotes that M satisfies φ under $\mathcal{I}$. This notion is defined in the expected manner. For instance, $M \models_{\mathcal{I}} Q_a(x)$ if $\lambda(\mathcal{I}(x)) = a$, $M \models_{\mathcal{I}} x \leq y$ if $\mathcal{I}(x) \leq \mathcal{I}(y)$ etc. For convenience, we have used $\leq$ to denote both the predicate symbol in the logic and the corresponding causality relation in the model M.

As usual, φ is a sentence if there are no free occurrences of individual or set variables in φ. With each sentence φ we can associate an MSC language $\mathcal{L}_\varphi \stackrel{\text{def}}{=} \{M \in \mathbb{MSC}(\mathcal{P}, B) \mid M \models \varphi\}$. We say that $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P}, B)$ is $\mathrm{MSO}(\mathcal{P}, B)$-definable if there exists a sentence φ such that $\mathcal{L}_\varphi = \mathcal{L}$. We wish to argue that $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P}, B)$ is $\mathrm{MSO}(\mathcal{P}, B)$-definable if and only if it is a B-bounded regular MSC language. It turns out the techniques used for proving a similar result in the theory of traces [32] can be suitably modified to derive our result.

Lemma 10.4.1 *Let φ be a sentence in $\mathrm{MSO}(\mathcal{P}, B)$. Then $\mathcal{L}_\varphi$ is a B-bounded regular MSC language.*

Proof sketch: The fact that $\mathcal{L}_\varphi$ is B-bounded follows from the semantics and hence we just need to establish regularity. Consider $\mathrm{MSO}(\Sigma)$, the monadic second-order theory of finite strings in Σ^*. This logic has the same syntax as $\mathrm{MSO}(\mathcal{P}, B)$ except that the ordering relation is interpreted over the positions of a structure in Σ^*. Let $L = \bigcup\{lin(M) \mid M \in \mathcal{L}_\varphi\}$. We exhibit a sentence $\widehat{\varphi}$ in $\mathrm{MSO}(\Sigma)$ such that $L = \{\sigma \mid \sigma \models \widehat{\varphi}\}$. The main observation is that the bound B ensures that the family of channel-capacity functions $\mathcal{K}$ can be captured by a fixed number of sets, which is used both to assert channel-consistency and to express the partial order of MSCs in terms of the underlying linear order of positions. The required conclusion will then follow from Büchi's Theorem [15].

Let $\{\mathcal{K}_0, \mathcal{K}_1, \ldots, \mathcal{K}_n\}$ be the set $\{\mathcal{K} \in \mathbb{N}^{Ch} \mid \forall c \in Ch.\ \mathcal{K}(c) \leq B\}$. Without loss of generality, assume that $\mathcal{K}_0(c) = 0$ for every $c \in Ch$. For $\mathcal{K} \in \mathbb{N}^{Ch}$ and $c \in Ch$, let $\mathcal{K}^{+c}$ to be the member of $\mathbb{N}^{Ch}$ where $\mathcal{K}^{+c}(c) = \mathcal{K}(c) + 1$ and $\mathcal{K}^{+c}(d) = \mathcal{K}(d)$ for all $d \neq c$. Similarly, for $\mathcal{K} \in \mathbb{N}^{Ch}$ and $c \in Ch$ such that $\mathcal{K}(c) > 0$, $\mathcal{K}^{-c}$ is given by $\mathcal{K}^{-c}(c) = \mathcal{K}(c) - 1$ and $\mathcal{K}^{-c}(d) = \mathcal{K}(d)$ for all $d \neq c$.

The required sentence $\widehat{\varphi}$ will be of the form :

$$(\exists X_{\mathcal{K}_0})(\exists X_{\mathcal{K}_1}) \cdots (\exists X_{\mathcal{K}_n})(COMP \wedge \|\varphi\|),$$

where $COMP$ and $\|\varphi\|$ are defined as follows. We provide these definitions in textual form to enhance readability. They can be easily converted to formulas in $\mathrm{MSO}(\Sigma)$.

First we define $COMP$ to be the conjunction of the following formulas.

(i) Every position x belongs to exactly one of the sets in $\{X_{\mathcal{K}_0}, \ldots, X_{\mathcal{K}_n}\}$.

(ii) If x is the first position then $x \in X_{\mathcal{K}_0}$.

(iii) If x is the last position then $Q_{q?p}(x)$ for some $c = (p, q)$. Moreover x belongs to $X_{\mathcal{K}_m}$ such that $\mathcal{K}_m(c) = 1$ and $\mathcal{K}_m(d) = 0$ for $d \neq c$.

(iv) If y is the successor of x, $Q_{p!q}(x)$, $x \in X_{\mathcal{K}_i}$ and $y \in X_{\mathcal{K}_j}$, then $\mathcal{K}_j = \mathcal{K}_i^{+c}$, where $c = (p, q)$.

(v) If y is the successor of x, $Q_{q?p}(x)$, $x \in X_{\mathcal{K}_i}$ and $y \in X_{\mathcal{K}_j}$, then $\mathcal{K}_i(c) > 0$ and $\mathcal{K}_j = \mathcal{K}_i^{-c}$, where $c = (p, q)$.

The formula $\|\varphi\|$ is given inductively as follows:

- $\|Q_a(x)\| \stackrel{\text{def}}{=} Q_a(x)$.
- $\|x \in X\| \stackrel{\text{def}}{=} x \in X$.
- $\|\neg\varphi'\| \stackrel{\text{def}}{=} \neg\|\varphi'\|$.
- $\|\varphi_1 \vee \varphi_2\| \stackrel{\text{def}}{=} \|\varphi_1\| \vee \|\varphi_2\|$.
- $\|(\exists x)\varphi'\| \stackrel{\text{def}}{=} (\exists x)\|\varphi'\|$.
- $\|(\exists X)\varphi'\| \stackrel{\text{def}}{=} (\exists X)\|\varphi'\|$.
- Finally, $\|x \leq y\| \stackrel{\text{def}}{=} x \sqsubseteq y$ where we shall first define $\sqsubseteq$ in terms of $\sqsubset$ and then define $\sqsubset$. This translation is based on the fact that in an MSC $M = (E, \leq, \lambda)$, $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$.

The formula $x \sqsubseteq y$ asserts that there exist non-empty subsets $\{p_1, p_2, \ldots, p_m\}$ of processes and $\{x_1, y_1, x_2, y_2, \ldots, x_m, y_m\}$ of positions such that $x = x_1$ and $y_m = y$. Further, $x_i \preceq y_i$ and x_i and y_i are both in Σ_{p_i} for $1 \leq i \leq m$. In addition, $y_i \sqsubset x_{i+1}$ for $1 \leq i < m$.

The predicate $x \sqsubset y$ is given by: $x \prec y$ and there is a channel $c = (p, q)$ such that $Q_{p!q}(x)$ and $Q_{q?p}(y)$. Further, if $x \in X_{\mathcal{K}_m}$ then there are exactly $\mathcal{K}_m(c)$ occurrences of the symbol $q?p$ between the positions x and y (and not including y). It is now straightforward to show that $\widehat{\varphi}$ has the required property. □

Lemma 10.4.2 *Let $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P}, B)$ be a regular MSC language. Then $\mathcal{L}$ is definable.*

Proof sketch: Let $L = \bigcup\{lin(M) \mid M \in \mathcal{L}\}$. Then L is a regular (string) MSC language over Σ. Hence by Büchi's Theorem [15] there exists a sentence φ in MSO(Σ) such that $L = \{\sigma \mid \sigma \models \varphi\}$. An important property of φ is that one linearization of an MSC satisfies φ iff all linearizations of the MSC satisfy φ. We then define the sentence $\widehat{\varphi} = \|\varphi\|$ in MSO($\mathcal{P}, B$) inductively such that the language of MSCs defined by $\widehat{\varphi}$ is precisely $\mathcal{L}$. The key idea here is to define a canonical linearization of MSCs and show that the underlying linear order is expressible in MSO($\mathcal{P}, B$). As a result, we can look for a formula $\widehat{\varphi}$ which will say "along the canonical linearization of an MSC, the sentence φ is satisfied". We present below the main ideas and constructions involved in arriving at $\widehat{\varphi}$.

Throughout what follows, we fix a strict linear order $\prec \subseteq \Sigma \times \Sigma$. Consider an MSC $M = (E, \leq, \lambda)$. For $e \in E$, let $\uparrow e = \{e' \mid e \leq e'\}$. For events $e, e' \in E$,

we define $e \; co \; e'$ if $e \not\leq e'$ and $e' \not\leq e$. For $X \subseteq E$, let $\lambda(X) \stackrel{\text{def}}{=} \{\lambda(e) \mid e \in X\}$. Next, suppose that $\emptyset \neq \Sigma' \subseteq \Sigma$. Then $\min(\Sigma')$ is the least element of Σ' under $\prec$. Finally, suppose $e, e' \in E$ with $e \; co \; e'$. Then $\Sigma_{ee'} = \lambda(\uparrow e \setminus \uparrow e')$.

Let $M = (E, \leq, \lambda)$ be an MSC. Then the ordering relation $\prec$ induces the ordering relation $\prec_M \subseteq E \times E$ given by $e \prec_M e'$ if $e < e'$ or ($e \; co \; e'$ and $\min(\Sigma_{ee'}) \prec \min(\Sigma_{e'e})$).

Claim: Let $M = (E, \leq, \lambda)$ be an MSC. Then $(E, \prec_M)$ is a strict linear order.

Proof: Same as the proof of [139, Lemma 15], which asserts an identical result in the setting of (infinite) Mazurkiewicz traces. □

We next exhibit a formula in $\text{MSO}(\mathcal{P}, B)$ (for any $B \in \mathbb{N}$) which captures the relation $\prec_M$ for each B-bounded MSC M. First we define the formula $\min(z_1, z_2, a)$ where z_1 and z_2 are individual variables and $a \in \Sigma$ via:

$$\begin{aligned}\min(z_1, z_2, a) \stackrel{\text{def}}{=} (\exists z)\, [\;\; & z_1 \leq z \wedge \neg(z_2 \leq z) \wedge Q_a(z) \wedge \\ & (\forall z') \;\; \big((z_1 \leq z' \wedge \neg(z_2 \leq z')) \Rightarrow Q_a(z') \vee \textstyle\bigvee_{a \prec a'} Q_{a'}(z')\big)]\end{aligned}$$

The formula $\text{Lex}(x, y)$ is now given by:

$$\text{Lex}(x, y) \stackrel{\text{def}}{=} (x < y) \vee \left(\text{co}(x, y) \wedge \bigvee_{a \prec b} \min(x, y, a) \wedge \min(y, x, b) \right)$$

where $\text{co}(x, y)$ is an abbreviation for $\neg(x \leq y) \wedge \neg(y \leq x)$.

Turning now to the proof of the statement of Lemma 10.4.2, let $L = \bigcup\{lin(M) \mid M \in \mathcal{L}\}$. Then L is a regular (string) MSC language over Σ. Hence by Büchi's Theorem [15] there exists a sentence φ in $\text{MSO}(\Sigma)$ such that $L = \{\sigma \mid \sigma \models \varphi\}$. We now define the formula $\widehat{\varphi} = \|\varphi\|$ in $\text{MSO}(\mathcal{P}, B)$ inductively as follows:

$$\|Q_a(x)\| \stackrel{\text{def}}{=} Q_a(x) \text{ and } \|x \preceq y\| \stackrel{\text{def}}{=} (x \leq y \wedge y \leq x) \vee \text{Lex}(x, y).$$

The remaining clauses are the natural ones. It is now straightforward to verify that $\mathcal{L}_{\widehat{\varphi}} = \mathcal{L}$. The key step in the proof is to show the following: Suppose $M \in \mathbb{MSC}(\mathcal{P}, B)$ and σ is the the linearization of M dictated by $\prec_M$. Then M is a model of $\widehat{\varphi}$ iff σ is a model of φ. This follows easily by structural induction on φ. The required conclusion can now be derived by exploiting the fact that L is $\sim$-closed. □

Since $\text{MSO}(\Sigma)$ is decidable, it follows that $\text{MSO}(\mathcal{P}, B)$ is decidable as well. To conclude, we can summarize the main results of this chapter as follows.

Theorem 10.4.3 *Let $L \subseteq \Sigma^*$, where Σ is the communication alphabet associated with a set $\mathcal{P}$ of processes. Then the following are equivalent.*

(i) L is a regular MSC language.

(ii) L *is a* B*-bounded regular MSC language, for some* $B \in \mathbb{N}$.

(iii) There exists a bounded message-passing automaton $\mathcal{M}$ *such that* $L(\mathcal{M}) = L$.

(iv) L *is* $\mathrm{MSO}(\mathcal{P}, B)$*-definable, for some* $B \in \mathbb{N}$.

Chapter 11

On Message Sequence Graphs and Finitely Generated Regular MSC Languages

Contents

Message Sequence Charts (MSCs) are an attractive visual formalism widely used to capture system requirements during the early design stages in domains such as telecommunication software. A standard method to describe multiple communication scenarios is to use Message Sequence Graphs (MSGs). An MSG allows the protocol designer to write a finite specification which combines MSCs using basic operations such as branching choice, composition and iteration. The MSC languages described by MSGs are not necessarily regular in the sense of Chapter 10. We characterize here the class of regular MSC languages that are MSG-definable in terms of a notion called finitely generated MSC languages. We show that a regular MSC language is MSG-definable if and only if it is finitely generated. In fact we show that the subclass of "locally synchronized" MSGs defined in [5] exactly capture the class of finitely generated regular MSC languages.

11.1 Introduction

Message Sequence Charts (MSCs) are an appealing visual formalism often used to capture system requirements in the early design stages. They are particularly suited for describing scenarios for distributed telecommunication software [68, 121]. They also appear in the literature as timing sequence diagrams, message flow diagrams and object interaction diagrams and are used in a number of software engineering methodologies [9, 52, 121]. In its basic form, an MSC depicts a single partially-ordered execution of a distributed system which just describes the exchange of messages between the processes of the system. A collection of MSCs is used to capture the scenarios that a designer might want the system to exhibit (or avoid).

Message Sequence Graphs (MSGs) are a nice mechanism for defining collections of MSCs. An MSG is a finite directed graph with a designated initial vertex and terminal vertex in which each node is labelled by an MSC and the edges represent a natural concatenation operation on MSCs. The collection of MSCs defined by an MSG consists of all those MSCs obtained by tracing a path in the MSG from the initial vertex to the terminal vertex and concatenating the MSCs that are encountered along the path. It is easy to see that this way of defining a collection of MSCs extends smoothly to the case where there are multiple terminal nodes. Throughout what follows we shall assume this extended notion of an MSG (that is, with multiple terminal nodes). For ease of presentation, we shall also not deal with the so called hierarchical MSGs [5].

Intuitively, not every MSG-definable collection of MSCs can be realized as a finite-state device. To formalize this idea we have introduced a notion of a *regular* collection of MSCs and studied its basic properties in Chapter 10. Our notion of regularity is independent of the notion of MSGs.

Our main goal in this chapter is to pin down the regular MSC languages that can be defined using MSGs. We introduce the notion of an MSC language being *finitely generated*. From our results, which we detail below, it follows that a regular MSC language is MSG-definable if and only if it is finitely generated. In fact we establish the following results.

As already mentioned, not every MSG defines a regular MSC language. Alur and Yannakakis have identified a syntactic property called *local synchronicity* and shown that the set of all linearizations of the MSCs defined by a locally synchronized MSG is a regular string language over an appropriate alphabet of events. It then follows easily that, in the present setting, every locally synchronized MSG defines a finitely generated regular MSC language. One of our main results here is that the converse is also true, namely, every finitely generated regular MSC language can be defined by a locally synchronized MSG. Since every MSG (locally synchronized or otherwise) defines only a finitely generated MSC language, it follows that a regular MSC language is finitely generated if and only if it is MSG-definable and, in fact, if and only if it is definable by some locally synchronized MSG.

Since the class of regular MSC languages strictly includes the class of finitely generated regular MSC languages, one could ask when a regular MSC language

is finitely generated. We show that this question is decidable. Finally, one can also ask whether a given MSG defines a regular MSC language (and is hence "equivalent" to a locally synchronized MSG). We show that this decision problem is undecidable.

Turning briefly to related literature, a number of studies are available which are concerned with individual MSCs in terms of their semantics and properties [3, 78]. A variety of algorithms have been developed for MSGs in the literature—for instance, pattern matching [79, 98, 100], detection of process divergence and non-local choice [8], and confluence and race conditions [99]. A systematic account of the various model-checking problems associated with MSGs and their complexities can be found in [5]. Finally, many of our proof techniques make use of results from the theory of Mazurkiewicz traces [27].

In the next section we introduce MSCs and regular MSC languages. We then introduce Message Sequence Graphs in Section 11.3. In Section 11.4 we define finitely generated MSC languages and provide an effective procedure to decide whether a regular MSC language is finitely generated. Our main result that the class of finitely generated regular MSC languages coincides with the class of locally synchronized MSG-definable languages is then established in Section 11.5. Finally, we sketch why the problem of determining if an MSG defines a regular MSC language is undecidable. We give here only the main technical constructions and sketches of proofs. More details are available in [56].

11.2 Regular MSC Languages

We fix a finite set of processes (or agents) $\mathcal{P}$ and let p, q, r range over $\mathcal{P}$. For each $p \in \mathcal{P}$ we define $\Sigma_p \stackrel{\text{def}}{=} \{p!q \mid p \neq q\} \cup \{p?q \mid p \neq q\}$ to be the set of communication actions in which p participates. The action $p!q$ is to be read as *p sends to q* and the action $p?q$ is to be read as *p receives from q*. At our level of abstraction, we shall not be concerned with the actual messages that are sent and received. We will also not deal with the internal actions of the agents. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$ and let a, b range over Σ. We also denote the set of *channels* by $Ch = \{(p, q) \mid p \neq q\}$ and let c, d range over Ch.

A Σ-labelled poset is a structure $M = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $e \in E$ we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \leq e\}$. For $p \in \mathcal{P}$ and $a \in \Sigma$, we set $E_p \stackrel{\text{def}}{=} \{e \mid \lambda(e) \in \Sigma_p\}$ and $E_a \stackrel{\text{def}}{=} \{e \mid \lambda(e) = a\}$, respectively. For each $c \in Ch$, we define the *communication relation* $R_c \stackrel{\text{def}}{=} \{(e, e') \mid \lambda(e) = p!q, \lambda(e') = q?p \text{ and } |\downarrow e \cap E_{p!q}| = |\downarrow e' \cap E_{q?p}|\}$. Finally, for each $p \in \mathcal{P}$, we define the *p-causality relation* $R_p \stackrel{\text{def}}{=} (E_p \times E_p) \cap \leq$.

An MSC (over $\mathcal{P}$) is a *finite* Σ-labelled poset $M = (E, \leq, \lambda)$ which satisfies the following conditions:

(i) Each R_p is a linear order.

(ii) If $p \neq q$ then $|E_{p!q}| = |E_{q?p}|$.

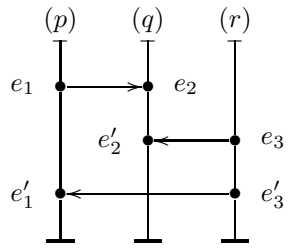


Figure 11.1: An example MSC over $\{p, q, r\}$.

(iii) $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$ where $R_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} R_p$ and $R_{Ch} = \bigcup_{c \in Ch} R_c$.

In diagrams, the events of an MSC are presented in *visual order*. The events of each process are arranged in a vertical line and the members of the relation R_{Ch} are displayed as horizontal or downward-sloping directed edges. We illustrate the idea with an example, shown in Figure 11.1.

Here $\mathcal{P} = \{p, q, r\}$. For $x \in \mathcal{P}$, the events in E_x are arranged along the line labelled (x) with earlier (relative to $\leq$) events appearing above the later events. The R_{Ch}-edges across agents are depicted by horizontal edges—for instance $e_3 \; R_{(r,q)} \; e'_2$. The labelling function λ is easy to extract from the diagram—for example, $\lambda(e'_3) = r!p$ and $\lambda(e_2) = q?p$.

We define regular MSC languages in terms of their linearizations. For the MSC $M = (E, \leq, \lambda)$, let $lin(M) \stackrel{\text{def}}{=} \{\lambda(\pi) \mid \pi \text{ is a linearization of } (E, \leq)\}$. By abuse of notation, we have used λ to also denote the natural extension of λ to E^*. The string $p!q \; r!q \; q?p \; q?r \; r!p \; p?r$ is a linearization of the MSC in Figure 11.1.

We say that $\sigma \in \Sigma^*$ is *proper* if for every prefix τ of σ and every pair (p, q) of processes, $|\tau|_{p!q} \geq |\tau|_{q?p}$. We say that σ is *complete* if σ is proper and $|\sigma|_{p!q} = |\sigma|_{q?p}$ for every pair of processes (p, q). Clearly, any linearization of an MSC is a complete string. Conversely, every complete sequence is the linearization of some MSC.

Henceforth, we identify an MSC with its isomorphism class. We let $\mathbb{MSC}(\mathcal{P})$ be the set of MSCs over $\mathcal{P}$. An MSC language $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$ is said to *regular* if $\bigcup\{lin(M) \mid M \in \mathcal{L}\}$ is a regular subset of Σ^*. We note that the entire set $\mathbb{MSC}(\mathcal{P})$ is not regular by this definition.

We define $L \subseteq \Sigma^*$ to be a *regular string MSC language* if there exists a regular MSC language $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$ such that $L = \bigcup\{lin(M) \mid M \in \mathcal{L}\}$. As shown in [56], regular MSC languages and regular string MSC languages represent each other. Hence, abusing terminology, we will write "regular MSC language" to mean "regular string MSC language". From the context, it should be clear whether we are working with MSCs from $\mathbb{MSC}(\mathcal{P})$ or complete words over Σ^*.

Given a regular subset $L \subseteq \Sigma^*$, we can decide whether L is a regular MSC language. We say that a state s in a finite-state automaton is *live* if there is

a path from s to a final state. Let $\mathcal{A} = (S, \Sigma, s_{in}, \delta, F)$ be the minimal DFA representing L. Then it is not difficult to see that L is a regular MSC language if and only if we can associate with each live state $s \in S$, a (unique) channel-capacity function $\mathcal{K}_s : Ch \to \mathbb{N}$ which satisfies the following conditions.

(i) If $s \in \{s_{in}\} \cup F$ then $\mathcal{K}_s(c) = 0$ for every $c \in Ch$.

(ii) If s, s' are live states and $\delta(s, p!q) = s'$ then $\mathcal{K}_{s'}((p,q)) = \mathcal{K}_s((p,q))+1$ and $\mathcal{K}_{s'}(c) = \mathcal{K}_s(c)$ for every $c \neq (p,q)$.

(iii) If s, s' are live states and $\delta(s, q?p) = s'$ then $\mathcal{K}_s((p,q)) > 0$, $\mathcal{K}_{s'}((p,q)) = \mathcal{K}_s((p,q))-1$ and $\mathcal{K}_{s'}(c) = \mathcal{K}_s(c)$ for every $c \neq (p,q)$.

(iv) Suppose $\delta(s,a) = s_1$ and $\delta(s_1, b) = s_2$ with $a \in \Sigma_p$ and $b \in \Sigma_q$, $p \neq q$. If $(a,b) \notin Com$ or $\mathcal{K}_s((p,q)) > 0$, there exists s_1' such that $\delta(s,b) = s_1'$ and $\delta(s_1', a) = s_2$. (Here and elsewhere $Com = \{(p!q, q?p) \mid p \neq q\}$.)

In the minimal DFA $\mathcal{A}$ representing a regular MSC language, if s is a live state and $a, b \in \Sigma$ then we say that a and b are *independent at* s if $(a,b) \in Com$ implies $\mathcal{K}_s((p,q)) > 0$ where $\mathcal{K}$ is the unique channel-capacity function associated with $\mathcal{A}$ and $a = p!q$ and $b = q?p$.

We conclude this section by introducing the notion of B-bounded MSC languages. Let $B \in \mathbb{N}$ be a natural number. We say that a word σ in Σ^* is B-bounded if for each prefix τ of σ and for each channel $(p,q) \in Ch$, $|\tau|_{p!q} - |\tau|_{q?p} \leq B$. We say that $L \subseteq \Sigma^*$ is B-bounded if every word $\sigma \in L$ is B-bounded. It is not difficult to show:

Proposition 11.2.1 *Let L be a regular MSC language. There is a bound $B \in \mathbb{N}$ such that L is B-bounded.*

11.3 Message Sequence Graphs

An MSG allows a protocol designer to write in a standard way [68], a finite specification which combines MSCs using operations such as branching choice, composition and iteration. Each node is labelled by an MSC and the edges represent the natural operation of MSC concatenation.

To bring out this concatenation operation, we let $M_1 = (E_1, \leq_1, \lambda_1)$ and $M_2 = (E_2, \leq_2, \lambda_2)$ be a pair for MSCs such that E_1 and E_2 are disjoint. For $i \in \{1,2\}$, let R_c^i and $\{R_p^i\}_{p \in \mathcal{P}}$ denote the underlying communication and process causality relations in M_i. The *(asynchronous) concatenation* of M_1 and M_2, denoted $M_1 \circ M_2$, is the MSC $(E, \leq, \lambda)$ where $E = E_1 \cup E_2$, $\lambda(e) = \lambda_i(e)$ if $e \in E_i$, $i \in \{1,2\}$, and $\leq = (R_{\mathcal{P}} \cup R_{Ch})^*$, where $R_p = R_p^1 \cup R_p^2 \cup \{(e_1, e_2) \mid e_1 \in E_1, e_2 \in E_2, \lambda(e_1) \in \Sigma_p, \lambda(e_2) \in \Sigma_p\}$ for $p \in \mathcal{P}$, and $R_c = R_c^1 \cup R_c^2$ for $c \in Com$.

A *Message Sequence Graph (MSG)* is a structure $\mathcal{G} = (Q, \longrightarrow, Q_{in}, F)$, where:

- Q is a finite and nonempty set of states.

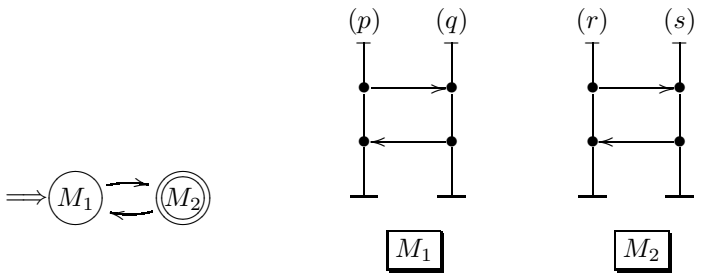


Figure 11.2: An example MSG.

- $\longrightarrow \subseteq Q \times Q$.
- $Q_{in} \subseteq Q$ is a set of initial states.
- $F \subseteq Q$ is a set of final states.
- $\Phi : Q \to \mathbb{MSC}(\mathcal{P})$ is a (state-)labelling function.

A *path* π through an MSG $\mathcal{G}$ is a sequence $q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_n$ such that $(q_{i-1}, q_i) \in \longrightarrow$ for $i \in \{1, 2, \ldots, n\}$. The MSC generated by π is $M(\pi) = M_0 \circ M_1 \circ M_2 \circ \cdots \circ M_n$, where $M_i = \Phi(q_i)$. A path $\pi = q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_n$ is a *run* if $q_0 \in Q_{in}$ and $q_n \in F$. The language of MSCs accepted by $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \{M(\pi) \in \mathbb{MSC}(\mathcal{P}) \mid \pi \text{ is a run through } \mathcal{G}\}$.

An example of an MSG is depicted in Figure 11.2. It's not hard to see that the MSC language $\mathcal{L}$ defined is *not* regular in the sense defined in Section 11.2. To see this, we note that $\mathcal{L}$ projected to $\{p!q, r!s\}$ is not a regular string language.

11.4 Finitely Generated MSC Languages

A key feature of MSG languages is that for each such language there is a fixed finite set $\mathcal{X}$ of MSCs such that each MSC in the language can be expressed as a concatenation of MSCs (with multiple copies) taken from $\mathcal{X}$. We say that they are finitely generated. In the next section we investigate the important connection between MSGs and finitely generated regular MSC languages. More precisely, we characterize the locally synchronized MSG-languages as precisely constituting the class of MSC languages that are both regular and finitely generated.

For the purpose of bringing all this out, let $\mathcal{L}_1, \mathcal{L}_2 \subseteq \mathbb{MSC}$ be two sets of MSCs. As usual, $\mathcal{L}_1 \circ \mathcal{L}_2$ denotes the pointwise concatenation of $\mathcal{L}_1$ and $\mathcal{L}_2$ as brought out in the previous section. For $\mathcal{X} \subseteq \mathbb{MSC}$, we define $\mathcal{X}^0 = \{\varepsilon\}$, where ε denotes the empty MSC, and for $i \geq 0$, $\mathcal{X}^{i+1} = \mathcal{X} \circ \mathcal{X}^i$. The *asynchronous iteration of* $\mathcal{X}$ is then defined by $\mathcal{X}^{\circledast} = \bigcup_{i \geq 0} \mathcal{X}^i$. Now, let $\mathcal{L} \subseteq \mathbb{MSC}$. We say

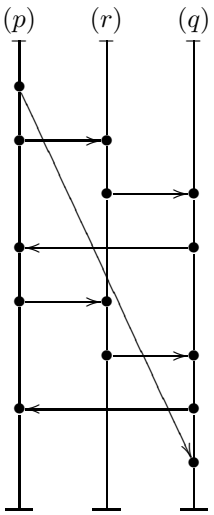


Figure 11.3: An atomic MSC of $\mathcal{L}_{inf}$ which is *not* a finitely generated language.

that $\mathcal{L}$ is *finitely generated* if there is a finite set of MSCs $\mathcal{X} \subseteq \mathbb{MSC}$ such that $\mathcal{L} \subseteq \mathcal{X}^{\circledast}$.

We first observe that not every regular MSC language is finitely generated. Let $\mathcal{P} = \{p, q, r\}$. Consider the regular expression $p!q\ \sigma^*\ q?p$, where σ is the sequence $p!r\ r?p\ r!q\ q?r\ q!p\ p?q$. This expression describes an infinite set of MSCs $\mathcal{L}_{inf} = \{M_i\}_{i=0}^{\omega}$. Figure 11.3 shows the MSC M_2. None of the MSCs in this language can be expressed as the concatenation of two or more non-trivial MSCs. Hence, this language is *not* finitely generated.

Our interest in finitely generated languages stems from the fact that these arise naturally from standard high-level descriptions of MSC languages such as message sequence graphs. However, as we saw earlier, Figure 11.2 provides an example showing that, conversely, not all finitely generated languages are regular.

The first question we address is that of deciding whether a regular MSC language is finitely generated. To do this, we need to introduce atoms. Let $M, M' \in \mathbb{MSC}$ be nonempty MSCs. Then M' *is a component of* M in case there exist $M_1, M_2 \in \mathbb{MSC}$ such that $M = M_1 \circ M' \circ M_2$. We say that M *is an atom* if the only component of M is M itself.

Thus, an atom is a nonempty message sequence chart that cannot be decomposed into non-trivial subcomponents. For an MSC M, we let $Atoms(M)$ denote the set $\{M' \mid M' \text{ is an atom and } M' \text{ is a component of } M\}$. For an MSC language $\mathcal{L} \subseteq \mathbb{MSC}$, $Atoms(\mathcal{L}) = \bigcup\{Atoms(M) \mid M \in \mathcal{L}\}$. It is clear that the question of deciding whether $\mathcal{L}$ is finitely generated is equivalent to that of checking whether $Atoms(\mathcal{L})$ is finite.

Theorem 11.4.1 *Let $\mathcal{L}$ be a regular MSC language. It is decidable whether $\mathcal{L}$ is finitely generated.*

Proof sketch: Let $\mathcal{A} = (S, \Sigma, s_{in}, \delta, F)$ be the minimum DFA for $\mathcal{L}$. From $\mathcal{A}$, we construct a finite family of finite-state automata which together accept the linearizations of the MSCs in $Atoms(\mathcal{L})$. It will then follow that $\mathcal{L}$ is finitely generated if and only if each of these automata accepts a finite language. We sketch the details below.

We know that for each live state $s \in S$, we can assign a capacity function $\mathcal{K}_s : Ch \to \mathbb{N}$ which counts the number of messages present in each channel when the state s is reached. We say that s is a *zero-capacity state* if $\mathcal{K}_s(c) = 0$ for each channel c. The following facts are easy to prove.

Fact 11.4.2 *Let M be an MSC in $Comp(\mathcal{L})$ (in particular, in $Atoms(\mathcal{L})$) and w be a linearization of M. Then, there are zero-capacity live states s, s' in $\mathcal{A}$ such that $s \xrightarrow{w} s'$.*

If M is in $Comp(\mathcal{L})$, then there are MSC's M_1, M_2 such that $M_1 M M_2 \in \mathcal{L}$. Thus, if w_1, w_2 are some linearizations of M_1 and M_2, then $w_1 w w_2$ is accepted by $\mathcal{A}$. Thus, there is an accepting run $s_{in} \xrightarrow{w_1} s \xrightarrow{w} s' \xrightarrow{w_2} t$. As linearizations of MSCs, w_1, w_2 and w are complete words. Further, s_{in} is a zero-capacity state and thus s and s' must be zero-capacity states. This proves Fact 11.4.2.

Fact 11.4.3 *Let M be an MSC in $Comp(\mathcal{L})$. M is an atom if and only if for each linearization w of M and each pair (s, s') of zero-capacity live states in $\mathcal{A}$, if $s \xrightarrow{w} s'$ then no intermediate state visited in this run has zero-capacity.*

Let M an atom and w be a linearization of M. Suppose $w = w_1 w_2$ for nonempty words w_1 and w_2 and $s \xrightarrow{w_1} s_1 \xrightarrow{w_2} s'$, where s_1 is a zero-capacity state. w_1 and w_2 are nonemtpy complete words. Recall that every complete word is the linearization of some MSC. Let M_1 and M_2 be the MSCs corresponding to w_1 and w_2. Then, $M = M_1 \circ M_2 \circ M_3$, where M_3 is the empty MSC, contradicting the assumption that M is an atom. Thus, the run can have no intermediate zero-capacity state.

Suppose, M is not an atom. Then $M = M_1 \circ M_2 \circ M_3$ and at least two of M_1, M_2, M_3 are nonempty. Let w_1, w_2 and w_3 be linearizations of M_1, M_2 and M_3. All three are complete words. Thus, there are states s_1, s_2 such that $s \xrightarrow{w_1} s_1 \xrightarrow{w_2} s_2 \xrightarrow{w_3} s'$. Since at least one of these words in nonempty, one of the states s_1 or s_2 is a zero-capacity intermediate state. This completes the proof of Fact 11.4.3.

Let us say that two complete words w and w' are equivalent, written $w \sim w'$, if they are linearizations of the same MSC. Suppose $s \xrightarrow{w} s'$ and $w \sim w'$. Then it is easy to see that $s \xrightarrow{w'} s'$ as well.

With each pair (s, s') of live zero-capacity states we associate a language $L_{At}(s, s')$. A word w belongs to $L_{At}(s, s')$ if and only if w is complete, $s \xrightarrow{w} s'$ and for each $w' \sim w$ the run $s \xrightarrow{w'} s'$ has no zero-capacity intermediate states.

From Facts 1 and 2 proved above, each of these languages consists of all the linearizations of some subset of $Atoms(\mathcal{L})$ and the linearizations of each element of $Atoms(\mathcal{L})$ is contained in some $L_{At}(s, s')$. Thus, it suffices to check for the finiteness of each of these languages.

Let $L_{s,s'}$ be the language of strings accepted by $\mathcal{A}$ when we set the initial state to be s and the set of final states to be $\{s'\}$. Clearly $L_{At}(s, s') \subseteq L_{s,s'}$. We now show how to construct an automaton for for $L_{At}(s, s')$.

We begin with $\mathcal{A}$ and prune the automaton as follows:

- Remove all incoming edges at s and all outgoing edges at s'.
- If $t \notin \{s, s'\}$ and $\mathcal{K}_t = \overline{0}$, remove t and all its incoming and outgoing edges.
- Recursively remove all states that become unreachable as a result of the preceding operation.

Let $\mathcal{A}_1$ be the resulting automaton. $\mathcal{A}_1$ accepts any complete word w on which the run from s to s' does not visit an intermediate zero-capacity state. Clearly, $L_{At}(s, s') \subseteq L(\mathcal{A}_1)$. However, $L(\mathcal{A}_1)$ may also contain linearizations of non-atomic MSCs that happen to have no complete prefix. For all such words, we know from Fact 11.4.3 that there is at least one equivalent linearization on which the run passes through a zero-capacity state and which would hence be eliminated from $L(\mathcal{A}_1)$. Thus, $L_{At}(s, s')$ is the $\sim$-closed subset of $L(\mathcal{A}_1)$ and we need to prune $\mathcal{A}_1$ further to obtain the automaton for $L_{At}(s, s')$.

Recall that the original DFA $\mathcal{A}$ was structurally closed with respect to the independence relation on communication actions in the following sense. Suppose $\delta(s_1, a) = s_2$ and $\delta(s_2, b) = s_3$ with a, b independent at s_1. Then, there exists s_2' such that $\delta(s_1, b) = s_2'$ and $\delta(s_2', a) = s_3$.

To identify the closed subset of $L(\mathcal{A}_1)$, we look for local violations of this "diamond" property and carefully prune transitions. We first blow up the state space into triples of the form (s_1, s_2, s_3) such that there exist a and a' with $\delta(s_1, a) = s_2$ and $\delta(s_2, a') = s_3$. Let S' denote this set of triples. We obtain a nondeterministic transition relation $\delta' \stackrel{\text{def}}{=} \{((s_1, s_2, s_3), a, (t_1, t_2, t_3)) \mid s_2 = t_1, s_3 = t_2, \delta(s_2, a) = s_3\}$. Set $S_{in} = \{(s_1, s_2, s_3) \in S' \mid s_2 = s_{in}\}$ and $F' = \{(s_1, s_f, s_2) \in S' \mid s_f \in F\}$. Let $\mathcal{A}_2 = (S', \Sigma, \delta', S_{in}, F')$.

Consider any state s_1 in $\mathcal{A}_1$ such that a and b are independent at s_1, $\delta(s_1, a) = s_2$, $\delta(s_2, b) = s_3$ but there is no s_2' such that $\delta(s_1, b) = s_2'$ and $\delta(s_2', a) = s_3$. For each such s_1, we remove all transitions of the form $((t, s_0, s_1), a, (s_0, s_1, t'))$ and $((t, s_2, s_3), b, (s_2, s_3, t'))$ from $\mathcal{A}_2$. We then recursively remove all states which become unreachable after this pruning.

Eventually, we arrive at an automaton $\mathcal{A}_3$ such that $L(\mathcal{A}_3) = L_{At}(s, s')$. Since $\mathcal{A}_3$ is a finite-state automaton, we can easily check whether $L(\mathcal{A}_3)$ is finite. This process is repeated for each pair of live zero-capacity states. $\square$

11.5 Locally Synchronized MSGs and Regular MSC Languages

Alur and Yannakakis [5] introduced the notion of local synchronicity for MSGs. They also showed that the set of all linearizations of the MSCs defined by a locally synchronized MSG is a regular string language. In the present setting this boils down to local synchronicity of an MSG being a sufficient condition for its MSC language to be regular. To state their condition, we first have to define the notion of connectedness.

Let $M = (E, \leq, \lambda)$ be an MSC. We let CG_M, *the communication graph of* M, be the directed graph $(\mathcal{P}, \mapsto)$ defined as follows: $(p, q) \in \mapsto$ if and only if there exists an $e \in E$ with $\lambda(e) = p!q$. M is *connected* if CG_M consists of one non-trivial strongly connected component and isolated vertices. For an MSC language $\mathcal{L} \subseteq \mathbb{MSC}(\mathcal{P})$ we say that $\mathcal{L}$ *is connected* in case every $M \in \mathcal{L}$ is connected.

Let $\mathcal{G} = (Q, \longrightarrow, Q_{in}, F)$ be an MSG. A *loop* in $\mathcal{G}$ is a sequence of edges that starts and ends at the same node. We say that $\mathcal{G}$ is *locally synchronized* if for every loop $\pi = q \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q$, the MSC $M(\pi)$ is connected. An MSC language $\mathcal{L}$ is a *locally synchronized MSG-language* if there exists a locally synchronized MSG $\mathcal{G}$ with $\mathcal{L} = \mathcal{L}(\mathcal{G})$.

It is easy to check whether a given MSG is locally synchronized. Clearly, the MSG of Figure 11.2 is *not* locally synchronized. One of the main results concerning locally synchronized MSGs shown in [5] at once implies:

Lemma 11.5.1 *Every locally synchronized MSG-language is a regular MSC language.*

One way to establish this result is — following [21] — to show that the asynchronous iteration of a connected regular MSC language is regular. The proof in [21] is based on grammars. A more direct, automata-theoretic proof of the same result is described in [56, Appendix B].

Our main interest in this section is to prove the converse of Lemma 11.5.1.

Lemma 11.5.2 *Let $\mathcal{L}$ be a finitely generated regular MSC language. Then $\mathcal{L}$ is a locally synchronized MSG-language.*

Proof sketch: Suppose $\mathcal{L}$ is a regular MSC language accepted by the minimal DFA $\mathcal{A} = (S, \Sigma, s_{in}, \delta, F)$. Let $Atoms(\mathcal{L}) = \{a_1, a_2, \ldots, a_m\}$. For each atom a_i, fix a linearization $u_i \in lin(a_i)$. Define an auxiliary DFA $\mathcal{A}' = (S^{\overline{0}}, Atoms(\mathcal{L}), s_{in}, \widehat{\delta}, \widehat{F})$ as follows:

- $S^{\overline{0}}$ is the set of states of $\mathcal{A}$ which have zero-capacity functions.
- $\widehat{F} = F$.
- $\widehat{\delta}(s, a_i) = s'$ iff $\delta(s, u_i) = s'$ in $\mathcal{A}$. (Note that $u, u' \in lin(a_i)$ implies $\delta(s, u) = \delta(s, u')$, so s' is fixed independent of the choice of $u_i \in lin(a_i)$.)

Thus, $\mathcal{A}'$ accepts the (regular) language of atoms corresponding to $\mathcal{L}(\mathcal{A})$. We can define a natural independence relation I_A on atoms as follows: atoms a_i and a_j are independent if and only if the set of active processes in a_i is disjoint from the set of active processes in a_j. (The process p is *active* in the MSC $(E, \leq, \lambda)$ if E_p is non-empty.)

It follows that $L(\mathcal{A}')$ is a regular Mazurkiewicz trace language over the trace alphabet $(Atoms(\mathcal{L}), I_A)$. As usual, for $w \in Atoms(\mathcal{L})^*$, we let $[w]$ denote the equivalence class of w with respect to I_A.

We now fix a strict linear order $\prec$ on $Atoms(\mathcal{L})$. This induces a (lexicographic) total order on words over $Atoms(\mathcal{L})$. Let $\textsc{Lex} \subseteq Atoms(\mathcal{L})^*$ be given by: $w \in \textsc{Lex}$ iff w is the lexicographically least element in $[w]$.

For a trace language L over $(Atoms(\mathcal{L}), I_A)$, let $lex(L)$ denote the set $L \cap \textsc{Lex}$.

Fact 11.5.3 ([27], Sect. 6.3.1)

(i) If L is a regular trace language over $(Atoms(\mathcal{L}), I_A)$, then $lex(L)$ is a regular language over $Atoms(\mathcal{L})$. Moreover, $L = \{[w] \mid w \in lex(L)\}$.

(ii) If $w_1 w w_2 \in \textsc{Lex}$, then $w \in \textsc{Lex}$.

(iii) If w is not *a connected*[1] *trace, then $ww \notin \textsc{Lex}$.*

From (i) we know that $lex(L(\mathcal{A}'))$ is a regular language over $Atoms(\mathcal{L})$. Let $\mathcal{C} = (S', Atoms(\mathcal{L}), s'_{in}, \delta', F')$ be the DFA over $Atoms(\mathcal{L})$ obtained by eliminating the (unique) dead state, if any, from the minimal DFA for $lex(L(\mathcal{A}'))$. It is easy to see that an MSC M belongs to $\mathcal{L}$ if and only if it can be decomposed into a sequence of atoms accepted by $\mathcal{C}$. Using this fact, we can derive an MSG $\mathcal{G}$ from $\mathcal{C}$ such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}$. We define $\mathcal{G} = (Q, \longrightarrow, Q_{in}, F)$ as follows:

- $Q = S' \times (Atoms(\mathcal{L}) \cup \{\varepsilon\})$.
- $Q_{in} = \{(s'_{in}, \varepsilon)\}$.
- $(s, b) \longrightarrow (s', b')$ iff $\delta'(s, b') = s'$.
- $F' = F \times Atoms(\mathcal{L})$.
- $\Phi(s, b) = b$.

Clearly $\mathcal{G}$ is an MSG and the MSC language that it defines is $\mathcal{L}$. We need to show that $\mathcal{G}$ is locally synchronized. To this end, let $\pi = (s, b) \longrightarrow (s_1, b_1) \longrightarrow \cdots \longrightarrow (s_n, b_n) \longrightarrow (s, b)$ be a loop in $\mathcal{G}$. We need to establish that the MSC $M(\pi) = b_1 \circ \cdots \circ b_n \circ b$ defined by this loop is connected. Let $w = b_1 b_2 \ldots b_n b$.

Consider the corresponding loop $s \xrightarrow{b_1} s_1 \xrightarrow{b_2} \cdots \xrightarrow{b_n} s_n \xrightarrow{b} s$ in $\mathcal{C}$. Since every state in $\mathcal{C}$ is live, there must be words w_1, w_2 over $Atoms(\mathcal{L})$ such that $w_1 w^k w_2 \in lex(L(\mathcal{A}'))$ for every $k \geq 0$.

[1] Recall from Section 3.2.5 that a trace is said to be *connected* if, when viewed as a labelled partial order, its Hasse diagram consists of a single connected component. See [27] for a more formal definition.

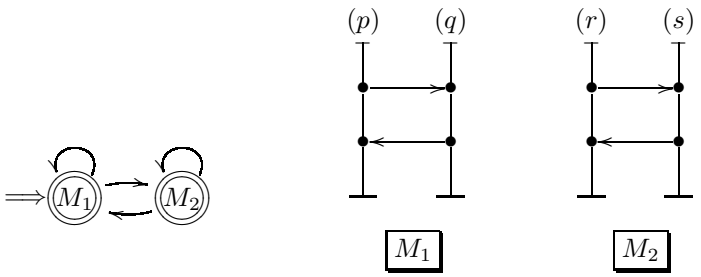


Figure 11.4: An non-locally synchronized MSG whose language is regular.

From (ii) of Fact 11.5.3, $w^k \in \textsc{Lex}$. This means, by (iii) of Fact 11.5.3, that w describes a connected trace over $(Atoms(\mathcal{L}), I_A)$. From this, it is not difficult to see that the underlying undirected graph of the communication graph $CG_{M(\pi)} = (\mathcal{P}, \mapsto)$ consists of a single connected component $C \subseteq \mathcal{P}$ and isolated processes. We have to argue that the component C is, in fact, *strongly* connected. We show that if C is not strongly connected, then the regular MSC language $\mathcal{L}$ is not B-bounded for any $B \in \mathbb{N}$, thus contradicting Proposition 11.2.1.

Suppose that the underlying graph of C is connected but C not strongly connected. Then, there exist two processes $p, q \in C$ such that $p \mapsto q$, but there is no path from q back to p in $CG_{M(\pi)}$. For $k \geq 0$, let $M(\pi)^k = (E, \leq, \lambda)$ be the MSC corresponding to the k-fold iteration $\underbrace{M(\pi) \circ M(\pi) \circ \cdots \circ M(\pi)}_{k \text{ times}}$. Since $p \mapsto q$ in $CG_{M(\pi)}$, it follows that there are events labelled $p!q$ and $q?p$ in $M(\pi)$. Moreover, since there is no path from q back to p in $CG_{M(\pi)}$, we can conclude that in $M(\pi)^k$, for each event e with $\lambda(e) = p!q$, there is no event labelled $q?p$ in $\downarrow e$. This means that $M(\pi)^k$ admits a linearization v'_k with a prefix τ'_k which includes all the events labelled $p!q$ and excludes all the events labelled $q?p$, so that $|\tau|_{p!q} - |\tau|_{q?p} \geq k$.

By Proposition 11.2.1, since $\mathcal{L}$ is a regular MSC language, there is be a bound $B \in \mathbb{N}$ such that every word in $\mathcal{L}$ is B-bounded—that is, for each $v \in \mathcal{L}$, for each prefix τ of v and for each channel $(p, q) \in Ch$, $|\tau|_{p!q} - |\tau|_{q?p} \leq B$. Recall that $w_1 w^k w_2 \in lex(L(\mathcal{A}'))$ for every $k \geq 0$. Fix linearizations v_1 and v_2 of the atom sequences w_1 and w_2, respectively. Then, for every $k \geq 0$, $u_k = v_1 v'_k v_2 \in \mathcal{L}$ where v'_k is the linearization of $M(\pi)^k$ defined earlier. Setting k to be $B{+}1$, we find that u_k admits a prefix $\tau_k = v_1 \tau'_k$ such that $|\tau_k|_{p!q} - |\tau_k|_{q?p} \geq B{+}1$, which contradicts the B-boundedness of $\mathcal{L}$.

Hence, it must be the case that C is a strongly connected component, which establishes that the MSG $\mathcal{G}$ we have constructed is locally synchronized. □

Putting together Lemmas 11.5.1 and 11.5.2, we obtain the following characterization of MSG-definable regular MSC languages.

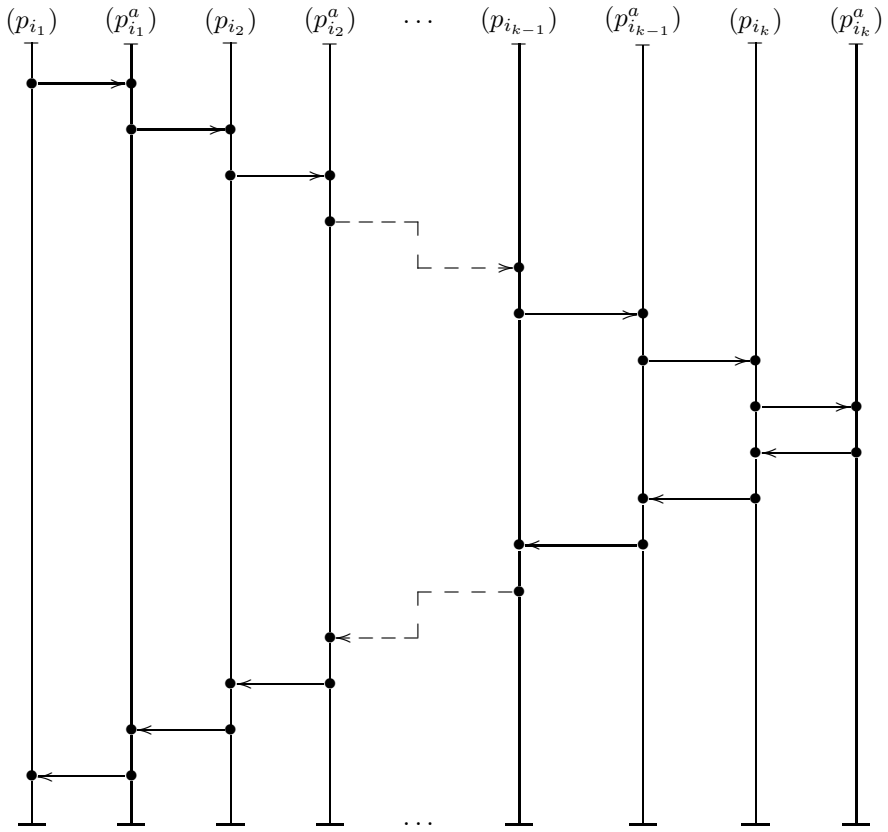


Figure 11.5: The MSC M_a encoding the letter $a \in A$.

Theorem 11.5.4 *Let $\mathcal{L}$ be a regular MSC language. Then the following statements are equivalent:*

(i) $\mathcal{L}$ is finitely generated.

(ii) $\mathcal{L}$ is a locally synchronized MSG-language.

(iii) $\mathcal{L}$ is MSG-definable.

It is easy to see that local synchronicity is not a necessary condition for regularity. Consider the MSG in Figure 11.4, which is not locally synchronized. It accepts the regular MSC language $M_1 \circ (M_1 + M_2)^{\circledast}$.

Thus, it would be useful to provide a characterization of the class of MSGs representing regular MSC languages. Unfortunately, the following result shows that there is no (recursive) characterization of this class.

Theorem 11.5.5 *The problem of deciding whether a given MSG represents a regular MSC language is undecidable.*

Proof sketch: It is known that the problem of determining whether the trace-closure of a regular language $L \subseteq A^*$ with respect to a trace alphabet (A, I) is also regular is undecidable [123]. We reduce this problem to the problem of checking whether the MSC language defined by an MSG is regular.

Let $\widetilde{A} = (A_1, \ldots, A_n)$ be a distributed alphabet implementing the trace alphabet (A, I) [27]. We will fix a set of processes $\mathcal{P}$ and the associated communication alphabet Σ and encode each letter a by an MSC M_a over $\mathcal{P}$.

For each i, we create $1 + |A_i|$ processes which we will denote by $p_i, p_i^{a_1}, p_i^{a_2}, \ldots, p_i^{a_k}$, where $A_i = \{a_1, a_2, \ldots, a_k\}$. Suppose now that the letter a appears in the components $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ of the distributed alphabet $\widetilde{A}$ with $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. The MSC M_a representing a is then given in Figure 11.5. It's easy to see that the communication graph CG_{M_a} is strongly connected. Moreover, if $(a, b) \in I$, then the sets of active processes of M_a and M_b are disjoint. The encoding ensures that we can construct a finite-state automaton to parse any word over Σ and determine whether it arises as the linearization of an MSC of the form $M_{a_1} \circ M_{a_2} \circ \cdots \circ M_{a_k}$. If so, the parser can uniquely reconstruct the corresponding word $a_1 a_2 \ldots a_k$ over A.

Let $\mathcal{A}$ be the minimal DFA corresponding to a regular language L over A. We construct an MSG $\mathcal{G}$ from $\mathcal{A}$ as described in the proof of Lemma 11.5.2. Given the properties of our encoding, we can then establish that the MSC language $L(\mathcal{G})$ is regular if and only if the trace-closure of L is regular, thus completing the reduction. □

Bibliography

[1] Aho, A., Hopcroft, J., Ullman, J.: *Design and Analysis of Computer Algorithms.* Addison-Wesley (1974)

[2] Alpern, B., Schneider, F. B.: Recognizing safety and liveness. *Distributed Computing* **2** (1987) 117–126

[3] Alur, R., Holzmann, G. J., Peled, D.: An analyzer for message sequence charts. *Software Concepts and Tools* **17**(2) (1996) 70–77

[4] Alur, R., Peled, D., Penczek, W.: Model checking of causality properties. *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, IEEE Computer Society Press (1995) 90–100

[5] Alur, R., Yannakakis, M.: Model checking of message sequence charts. *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, Lecture Notes in Computer Science **1664**, Springer-Verlag (1999) 114–129

[6] Balarin, F., Sangiovanni-Vincentelli, A. L.: An iterative approach to language containment. *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, Lecture Notes in Computer Science **697**, Springer-Verlag (1993) 29–40

[7] Basin, D., Klarlund, N.: Automata-based symbolic reasoning in hardware verification. *Formal Methods in System Design* **13** (1998) 255–288

[8] Ben-Abdallah, H., Leue, S.: Syntactic detection of process divergence and non-local choice in message sequence charts. *Proceedings of the 3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, Lecture Notes in Computer Science **1217**, Springer-Verlag (1997) 259–274

[9] Booch, G., Jacobson, I., Rumbaugh, J.: *Unified Modeling Language User Guide.* Addison-Wesley (1997)

[10] Bracho, F., Droste, M., Kuske, D.: Representation of computations in concurrent automata by dependence orders. *Theoretical Computer Science* **174**(1-2) (1997) 67–96

[11] Bryant, R. E.: Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing surveys* **24**(3) (1992) 293–318

[12] Bryant, R. E.: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* **C-35**(8) (1986) 677–691

[13] Burkart, O., Esparza, J.: More infinite results. *Bulletin of the EATCS* **62** (1997) 138–159

[14] Büchi, J. R.: On a decision method in restricted second order arithmetic. *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, Stanford University Press (1960) 1–11

[15] Büchi, J. R.: Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.* **6** (1960) 66–92

[16] Cadence Design Systems: Affirma FormalCheck model checker. Further information can be obtained at `http://www.cadence.com/technology/funct/products/formalcheck.html`

[17] Cai, J., Paige, R.: Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science* **145**(1-2) (1995) 189–228

[18] Cheng, A.: Petri nets, traces, and local model checking. *Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology (AMAST'95)*, Lecture Notes in Computer Science **936**, Springer-Verlag (1995) 322–337

[19] Clarke, E. M., Emerson, E. A., Sistla, A. P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* **8**(2) (1986) 244–263

[20] Clarke, E. M., Grumberg, O., Peled, D.: *Model checking*. MIT Press (1999)

[21] Clerbout, M., Latteux, M.: Semi-commutations. *Information and Computation* **73**(1) (1987) 59–74

[22] Corsini, M.-M., Rauzy, A.: Symbolic model checking and constraint logic programming: a cross-fertilisation. *Proceedings of the 5th European Symposium on Programming (ESOP'94)*, Lecture Notes in Computer Science **788**, Springer-Verlag (1994) 180–194

[23] Damm, W., Harel, D.: LCSs: Breathing life into message sequence charts. *Proceedings of the 3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, Kluwer Academic Publishers (1999) 293–312

[24] Diekert, V.: *Combinatorics on Traces*. Lecture Notes in Computer Science **454**, Springer-Verlag (1990)

[25] Diekert, V., Gastin, P.: An expressively complete temporal logic without past tense operators for Mazurkiewicz traces. *Proceedings of the 13th Annual Conference of the European Association for Computer Science Logic (CSL'99)*, Lecture Notes in Computer Science **1683**, Springer-Verlag (1999) 188–203

[26] Diekert, V., Gastin, P.: LTL is expressively complete for Mazurkiewicz traces. *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP'00)*, Lecture Notes in Computer Science **1853**, Springer-Verlag (2000) 211–222

[27] Diekert, V., Rozenberg, G. (Eds.): *The Book of Traces*. World Scientific (1995)

[28] Droste, M.: Recognizable languages in concurrency monoids. *Theoretical Computer Science* **150**(1) (1995) 77–109

[29] Droste, M., Gastin, P.: Asynchronous cellular automata for pomsets without auto-concurrency. *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science **1119**, Springer-Verlag (1996) 627–638

[30] D'Souza, D., Madhusudan, P.: On-the-fly verification for product-LTL. *Proceedings of the 7th Indian National Seminar of Theoretical Computer Science (NSTCS'97)* (1997)

[31] Ebinger, W.: *Charakterisierung von Sprachklassen Unendlicher Spuren durch Logiken*. Ph.D. Dissertation, Institut für Informatik, Universität Stuttgart (1994)

[32] Ebinger, W., Muscholl, A.: Logical definability on infinite traces. *Theoretical Computer Science* **154**(1) (1996) 67–84

[33] Ehrenfeucht, A.: An application of games to the completeness problem for formalized theories. *Fund. Math.* **49** (1961) 129–141

[34] Elgot, C. C.: Decision problems of finite automata design and related arithmetics. *Transactions of American Mathematical Society* **98** (1961) 21–52

[35] Emerson, A. E.: Temporal and modal logic. In van Leeuwen, J. (Ed.): *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier Science Publishers (1990) 996–1072

[36] Etessami, K., Vardi, M. Y., Wilke, Th.: First-order logic with two variables and unary temporal logic. *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, IEEE Computer Society Press (1997) 228–235

[37] Etessami, K., Wilke, Th.: An until hierarchy for temporal logic. *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, IEEE Computer Society Press (1996) 108–117

[38] Fagin, R., Halpern, J. Y., Moses, Y., Vardi, M. Y.: *Reasoning about Knowledge*. MIT Press (1995)

[39] Fischer, M. J., Ladner, R. E.: Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences* **18**(2) (1979) 194–211

[40] Fraïssé, R.: Sur quelques classifications des relations, basés sur des isomorphismes restreints. *Publ. Sci. de l'Univ. Alger*, Sér. **A 1** (1954) 35–182

[41] Gabbay, A., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. *Proceedings of the 7th Annual Symposium on Principles of Programming Languages (POPL'80)*, ACM Press (1980) 163–173

[42] Gastin, P., Meyer, R., Petit, A.: A (non-elementary) modular decision procedure for LTrL. *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, Lecture Notes in Computer Science **1450**, Springer-Verlag (1998) 356–365

[43] Gastin, P., Petit, A.: Asynchronous cellular automata for infinite traces. *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*. Lecture Notes in Computer Science **623**, Springer-Verlag (1992) 583–594

[44] Gastin, P., Petit, A., Zielonka, W.: An extension of Kleene's and Ochmanski's theorems to infinite traces. *Theoretical Computer Science* **125** (1994) 167–204

[45] Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial-order approach to branching time model checking. *Proceedings of the 3rd Israeli Symposium on Theory of Computing and Systems (ISTCS'95)*, IEEE Computer Society Press (1995) 130–139

[46] Gerth, R., Peled, D., Vardi, M. Y., Wolper, P.: Simple on-the-fly automatic verification of linear time temporal logic. *Proceedings of the 15th IFIP WG 6.1 International Workshop on Protocol Specification, Testing, and Verification (PSTV'95)*, North-Holland (1995)

[47] van Glabbeek, R. J.: The linear time–branching time spectrum. *Proceedings of Theories of Concurrency: Unification and Extension (CONCUR'90)*, Lecture Notes in Computer Science **458**, Springer-Verlag (1990) 278–297

[48] Godefroid, P.: *Partial-order Methods for the Verification of Concurrent Systems*. Lecture Notes in Computer Science **1032**, Springer-Verlag (1996)

[49] Gupta, A., Fisher, A. L.: Parametric circuit representation using inductive boolean functions. *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, Lecture Notes in Computer Science **697**, Springer-Verlag (1993) 15–28

[50] Hafer, Th., Thomas, W.: Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. *Proceedings of the 14th International Colloquium on Automata, Languages and Programming (ICALP'87)*, Lecture Notes in Computer Science **267**, Springer-Verlag (1987) 269–279

[51] Harel, D.: Dynamic logic. In Gabbay, D., Guenthner, F. (Eds.): *Handbook of Philosophical Logic – Volume II*, Reidel Dordrecht (1984) 497–604

[52] Harel, D., Gery, E.: Executable object modeling with statecharts. *IEEE Computer*, July 1997 (1997) 31–42

[53] Harel, D., Kozen, D., Parikh, R.: Process logic: expressiveness, decidability, completeness. *Journal of Computer and System Sciences* **25** (1982) 144–170

[54] Henriksen, J. G.: An expressive extension of TLC. *Proceedings of the 5th Asian Computing Science Conference (ASIAN'99)*, Lecture Notes in Computer Science **1742**, Springer-Verlag (1999) 126–138

[55] Henriksen, J. G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: MONA: monadic second-order logic in practice. *Proceedings of the 1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95), Selected Papers*, Lecture Notes in Computer Science **1019**, Springer-Verlag (1996) 89–110

[56] Henriksen, J. G, Mukund, M., Narayan Kumar, K., Thiagarajan, P. S.: Towards a theory of regular MSC languages. Technical report RS-99-52, BRICS, Department of Computer Science, University of Aarhus (1999)

[57] Henriksen, J. G, Mukund, M., Narayan Kumar, K., Thiagarajan, P. S.: Regular collections of message sequence charts. *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'00)*, Lecture Notes in Computer Science **1893**, Springer-Verlag (2000) 405–414

[58] Henriksen, J. G, Mukund, M., Narayan Kumar, K., Thiagarajan, P. S.: On message sequence graphs and finitely generated regular MSC languages. *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP'00)*, Lecture Notes in Computer Science **1853**, Springer-Verlag (2000) 675–686

[59] Henriksen, J. G., Thiagarajan, P. S.: Dynamic linear time temporal logic. Technical report RS-97-8, BRICS, Department of Computer Science, University of Aarhus (1997)

[60] Henriksen, J. G., Thiagarajan, P. S.: Dynamic linear time temporal logic. *Annals of Pure and Applied Logic* **96**(1-3) (1999) 187–207

[61] Henriksen, J. G., Thiagarajan, P. S.: A product version of dynamic linear time temporal logic. *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*, Lecture Notes in Computer Science **1243**, Springer-Verlag (1997) 45–58

[62] Holzmann, G. J.: An overview of the SPIN model checker. In *On-the-fly Model Checking Tutorial, BRICS Autumn School on Verification*, Note NS-96-6, BRICS, Department of Computer Science, University of Aarhus (1996)

[63] Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In Kohavi, Z., Paz, A. (Eds.): *Theory of Machines and Computations*, Academic Press (1971) 189–196

[64] Hromkovič, J., Seibert, S., Wilke, Th.: Translating regular expressions into small ε-free nondeterministic automata. *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97)*, Lecture Notes in Computer Science **1200**, Springer-Verlag (1997) 55–66

[65] Huhn, M.: On semantic and logical refinement of actions. Technical Report, Institut für Informatik, Universität Hildesheim, Germany (1996)

[66] Immermann, N., Kozen, D.: Definability with a bounded number of bound variables. *Information and Computation* **83**(2) (1989) 121–139

[67] Immermann, N., Vardi, M. Y.: Model checking and transitive-closure logic. *Proceedings of 9th International Conference on Computer Aided Verification (CAV'97)*, Lecture Notes in Computer Science **1254**, Springer-Verlag (1997) 291–302

[68] ITU-TS Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU-TS, Geneva (1997)

[69] Jensen, J., Jørgensen, M., Klarlund, N.: Monadic second-order logic for parameterized verification. Technical report RS-94-10, BRICS, Department of Computer Science, University of Aarhus (1994)

[70] Kamp, H. R.: *Tense Logic and the Theory of Linear Order*. Ph.D. thesis, University of California (1968)

[71] Katz, S., Peled, D.: Interleaving set temporal logic. *Theoretical Computer Science* **73**(3) (1992) 21–43

[72] Klarlund, N., Møller, A.: MONA version 1.2 – user manual. Note NS-98-3, BRICS, Department of Computer Science, University of Aarhus (1998)

[73] Klarlund, N., Møller, A.: The MONA project homepage. `http://www.brics.dk/mona/`.

[74] Kleene, S. C.: Representation of events in nerve nets and finite automata. In Shannon, C. E., McCarthy, J. (Eds.): *Automata Studies*. Princeton University Press (1956) 3–41

[75] Kozen, D.: *Automata and Computability*. Undergraduate Texts in Computer Science, Springer-Verlag (1997)

[76] Kozen, D., Parikh, R.: An elementary proof of the completeness of PDL. *Theoretical Computer Science* **14** (1981) 113–118

[77] Kurshan, B., McMillan, K.: A structural induction theorem for processes. *Proceedings of the 8th Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press (1989) 239–247

[78] Ladkin, P. B., Leue, S.: Interpreting message flow graphs. *Formal Aspects of Computing* **7**(5) (1995) 473–509

[79] Levin, V., Peled, D.: Verification of message sequence charts via template matching. *Proceedings of the 7th International Conference on Theory and Practice of Software Development (TAPSOFT'97)*, Lecture Notes in Computer Science **1214**, Springer-Verlag (1997) 652–666

[80] Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In Parikh, R. (Ed.): *Logics of Programs*, Lecture Notes in Computer Science **193**, Springer-Verlag (1985) 196–218

[81] Lodaya, K., Parikh, R., Ramanujam, R., Thiagarajan, P. S.: A logical study of distributed transition systems. *Information and Computation* **119**(1) (1995) 91–118

[82] Manna, Z., Pnueli, A.: The anchored version of the temporal framework. In de Bakker, J. W. (Ed.): *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science **345**, Springer-Verlag (1989) 210–284

[83] Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems (Specification)*. Springer-Verlag (1992)

[84] Mauw, S., Reniers, M. A.: High-level message sequence charts. *Proceedings of the Eighth SDL Forum (SDL'97): Time for Testing — SDL, MSC and Trends*, Elsevier Science Publishers (1997) 291–306

[85] Mayr, R.: *Decidability and Complexity of Model Checking Problems for Infinite-state Systems*. Ph.D. thesis, Institut für Informatik, Technische Universität München (1998)

[86] Mazurkiewicz, A.: Concurrent program schemes and their interpretations. Technical report DAIMI PB-78, Department of Computer Science, University of Aarhus, Denmark (1977)

[87] McNaughton, R.: Testing and generating infinite sequences by a finite automaton. *Information and Control* **9** (1966) 521–530

[88] McNaughton, R., Papert, S.: *Counter-free Automata*. MIT Press (1971)

[89] Meyer, R., Petit, A.: Decomposition of TrPTL formulas. In *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS'97)*, Lecture Notes in Computer Science **1295**, Springer-Verlag (1997) 418–427

[90] Meyer, R., Petit, A.: Expressive completeness of LTrL on finite traces: an algebraic proof. *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS'98)*, Lecture Notes in Computer Science **1373**, Springer-Verlag (1998) 533–543

[91] Milner, R.: *Communication and Concurrency*. Prentice-Hall (1989)

[92] Moller, F., Rabinovich, A.: On the expressive power of CTL*. *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*, IEEE Computer Society Press (1977) 360–369

[93] Mukund, M., Narayan Kumar, K., Sohoni, M.: Keeping track of the latest gossip in message-passing systems. *Proceedings of Structures in Concurrency Theory (STRICT)*, Workshops in Computing Series, Springer-Verlag (1995) 249–263

[94] Mukund, M., Narayan Kumar, K., Sohoni, M.: Synthesizing distributed finite-state systems from MSCs. *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, Lecture Notes in Computer Science **1877**, Springer-Verlag (2000) 521–535

[95] Mukund, M., Sohoni, M.: Gossiping, asynchronous automata and Zielonka's theorem. Technical report TCS-94-2, Chennai Mathematical Institute (1994)

[96] Mukund, M., Sohoni, M.: Keeping track of the latest gossip in a distributed system. *Distributed Computing* **10**(3) (1997) 117–127

[97] Mukund, M., Thiagarajan, P. S.: Linear time temporal logics over Mazurkiewicz traces. *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science (MFCS'96)*, Lecture Notes in Computer Science **1113**, Springer-Verlag (1996) 62–92

[98] Muscholl, A.: Matching specifications for message sequence charts. *Proceedings of the 2nd International Conference on Foundations of Software Science and Computation Structures (FOSSACS'99)*, Lecture Notes in Computer Science **1578**, Springer-Verlag (1999) 273–287

[99] Muscholl, A., Peled, D.: Message sequence graphs and decision problems on Mazurkiewicz traces. *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, Lecture Notes in Computer Science **1672**, Springer-Verlag (1999) 81–91

[100] Muscholl, A., Peled, D., Su, Z.: Deciding properties for message sequence charts. *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures (FOSSACS'98)*, Lecture Notes in Computer Science **1378**, Springer-Verlag (1998) 226–242

[101] Niebert, P.: A ν-calculus with local views for systems of sequential agents. *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, Lecture Notes in Computer Science **969**, Springer-Verlag (1995) 563–573

[102] Niebert, P.: *A Temporal Logic for the Specification and Validation of Distributed Behaviour*. Ph.D. thesis, University of Hildesheim (1997)

[103] Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains, part I. *Theoretical Computer Science* **13**(1) (1981) 85–108

[104] Nishimura, H.: Descriptively complete process logic. *Acta Informatica* **14**(4) (1980) 359–369

[105] Ochmański, E.: Regular behaviour of concurrent systems. *Bulletin of the EATCS* **27** (1985) 56–67

[106] Paige, R., Tarjan, R.: Three efficient algorithms based on partition refinement. *SIAM Journal of Computing* **16**(6) (1987) 973–989

[107] Park, D. M. R.: *Concurrency and Automata on Infinite Sequences*. Lecture Notes in Computer Science **104**, Springer-Verlag (1980)

[108] Peled, D.: Partial order reduction: model checking using representatives. *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science (MFCS'96)*, Lecture Notes in Computer Science **1113**, Springer-Verlag (1996) 93–112

[109] Peled, D., Pnueli, A.: Proving partial order properties. *Theoretical Computer Science* **126**(2) (1994) 143–182

[110] Peled, D., Wilke, Th.: Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters* **63** (1997) 243–246

[111] Peled, D., Wilke, Th., Wolper, P.: An algorithmic approach for checking closure properties of temporal logic properties and omega-regular languages. *Theoretical Computer Science* **195**(2) (1998) 183–204

[112] Penczek, W.: Temporal logics for trace systems: on automated verification. *International Journal of the Foundations of Computer Science* **4**(1) (1993) 31–68

[113] Pnueli, A.: The temporal logic of programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, IEEE Computer Society Press (1977) 46–57

[114] Pratt, V. R.: Process logic. *Proceedings of the 6th Symposium on Principles of Programming Languages (POPL'79)*, ACM Press (1979) 93–100

[115] Ramanujam, R.: Locally linear time temporal logic. *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, IEEE Computer Society Press (1996) 118–127

[116] Ramanujam, R.: Rules for trace consistent reasoning. *Proceedings of the 3rd Asian Computing Science Conference (ASIAN'97)*, Lecture Notes in Computer Science **1345**, Springer-Verlag (1997) 57–71

[117] Reisig, W.: Temporal logic and causality in concurrent systems. *Proceedings of CONCURRENCY'88*, Lecture Notes in Computer Science **335**, Springer-Verlag (1988) 121–139

[118] Reisig, W.: Petri net models for distributed algorithms. In van Leeuwen, J. (Ed.): *Computer Science Today — Recent Trends and Developments*, Lecture Notes in Computer Science **1000**, Springer-Verlag (1995) 441–454

[119] Revuz, D.: Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, **92**(1) (1992) 181–189

[120] Rho, J.-K., Somenzi, F.: Automatic generation of network invariants for the verification of iterative sequential systems. *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, Lecture Notes in Computer Science **697**, Springer-Verlag (1993) 123–137

[121] Rudolph, E., Graubmann, P., Grabowski, J.: Tutorial on message sequence charts. In *Computer Networks and ISDN Systems — SDL and MSC* **28** (1996).

[122] Safra, S.: On the complexity of ω-automata. *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS'88)*, IEEE Computer Society Press (1988) 319–327

[123] Sakarovitch, J.: The "last" decision problem for rational trace languages. *Proceedings of the 1st Latin American Symposium on Theoretical Informatics (LATIN'92)*, Lecture Notes in Computer Science **583**, Springer-Verlag (1992) 460–473

[124] Segerberg, K.: A completeness theorem in the modal logic of programs. *Notices AMS* **24**(6) (1977) A–522

[125] Sieling, D., Wegener, I.: Reduction of OBDDs in linear time. *Information Processing Letters* **48** (1993) (139–144)

[126] Sistla, A. P.: *Theoretical Issues in the Design and Verification of Distributed Systems*. Ph.D. Thesis, Harvard University (1983)

[127] Sistla, A. P., Clarke, E.: The complexity of propositional linear temporal logics. *Journal of the ACM* **32**(3) (1985) 733–749

[128] Sistla, A. P., Vardi, M. Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science* **49** (1987) 217–237

[129] Steinmann, M.: Übersetzung von logischen ausdrücken in baumautomaten: entwicklung eines verfahrens und seine implementierung. Unpublished (1993)

[130] Stirling, C.: Modal and temporal logics. In Abramsky, S., Gabbay, D. M., Maibaum, T. S. E. (Eds.): *Handbook of Logic in Computer Science, Vol. 2*, Oxford University Press (1992) 477–563

[131] Stockmeyer, L. J.: *The Complexity of Decision Problems in Automata Theory and Logic*. Ph.D. thesis, MIT, Cambridge, Massachusetts (1974)

[132] Thérien, D., Wilke, Th.: Temporal logic and semidirect products: an effective characterization of the until hierarchy. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, IEEE Computer Society Press (1996) 256–263

[133] Thérien, D., Wilke, Th.: Over words, two variables are as powerful as one quantifier alternation: $\mathrm{FO}^2 = \Sigma_2 \cap \Pi_2$. *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98)*, ACM Press (1998) 234–240

[134] Thiagarajan, P. S.: TrPTL: a trace based extension of linear time temporal logic. Technical report TCS-93-6, Chennai Mathematical Institute (1993)

[135] Thiagarajan, P. S.: A trace based extension of linear time temporal logic. *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS'94)*, IEEE Computer Society Press (1994) 438–447

[136] Thiagarajan, P. S.: PTL over product state spaces. Technical report TCS-95-4, Chennai Mathematical Institute (1995)

[137] Thiagarajan, P. S.: A trace consistent subset of PTL. *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, Lecture Notes in Computer Science **962**, Springer-Verlag (1995) 438–452

[138] Thiagarajan, P. S., Henriksen, J. G.: Distributed versions of linear time temporal logic: A trace perspective. In Reisig, W., Rozenberg, G. (Eds.): *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science **1491**, Springer-Verlag (1998) 643–681

[139] Thiagarajan, P. S., Walukiewicz, I.: An expressively complete linear time temporal logic for Mazurkiewicz traces. *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, IEEE Computer Society Press (1997) 183–194

[140] Thomas, W.: Automata on infinite objects. In van Leeuwen, J. (Ed.): *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier Science Publishers (1990) 133–191

[141] Thomas, W.: On logical definability of trace languages. *Proceedings of the ASMICS Workshop on Free Partially Commutative Monoids*, Technical report TUM-I9002, Institut für Informatik, Technische Universität München (1990) 172–182

[142] Thomas, W.: On the Ehrenfeucht-Fraïssé game in theoretical computer science. *Proceedings of the 5th International Conference on Theory and Practice of Software Development (TAPSOFT'93)*, Lecture Notes in Computer Science **668**, Springer-Verlag (1993) 559–568

[143] Thomas, W.: Languages, automata, and logic. In Rozenberg, G., Salomaa, A. (Eds.): *Handbook of Formal Language Theory, Volume III*, Springer-Verlag (1997) 389–455

[144] Valmari, A.: A stubborn attack on state explosion. *Formal Methods in Systems Design* **1** (1992) 285–313

[145] Vardi, M. Y.: An automata-theoretic approach to linear time temporal logic. In Moller, F., Birtwistle, G. (Eds.): *Logics for Concurrency — Structure vs. Automata*, Lecture Notes in Computer Science **1043**, Springer-Verlag (1996) 238–266

[146] Vardi, M. Y.: Linear time vs. branching time: a complexity perspective. *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, IEEE Computer Society Press (1998) 394–405

[147] Vardi, M. Y., Wolper, P.: An automata-theoretic approach to automatic program verification. *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science (LICS'86)*, IEEE Computer Society Press (1986) 332–345

[148] Vardi, M. Y., Wolper, P.: Reasoning about infinite computations. *Information and Computation* **115**(1) (1994) 1–35

[149] Walukiewicz, I.: Difficult configurations — on the complexity of LTrL (extended abstract). *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*. Lecture Notes in Computer Science **1443**, Springer-Verlag (1998) 140–151

[150] Walukiewicz, I.: Local logics for traces. Technical report RS-00-02, BRICS, Department of Computer Science, University of Aarhus (2000)

[151] Walukiewicz, I.: Personal communication.

[152] Wilke, Th.: Classifying discrete temporal properties. *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS 99)*, Lecture Notes in Computer Science **1563**, Springer-Verlag (1999) 32–46

[153] Willems, B., Wolper, P.: Partial-order methods for model checking: from linear time to branching time. *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, IEEE Computer Society Press (1996) 294–303

[154] Winskel, G., Nielsen, M.: Models for concurrency. In Abramsky, S., Gabbay, D. M., Maibaum, T. S. E. (Eds.): *Handbook of Logic in Computer Science, Vol. 4*, Oxford University Press (1995) 1–148

[155] Wolper, P.: Temporal logic can be more expressive. *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS'81)*, IEEE Computer Society Press (1981) 340–348

[156] Wolper, P., Vardi, M. Y., Sistla, A. P.: Reasoning about infinite computation paths. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Computer Society Press (1983) 185–194

[157] Zielonka, W.: Notes on finite asynchronous automata. *R.A.I.R.O. Informatique Théorique et Applications* **21** (1987) 99–135

Recent BRICS Dissertation Series Publications

DS-00-6 **Jesper G. Henriksen.** ***Logics and Automata for Verification: Expressiveness and Decidability Issues.*** **May 2000. PhD thesis. xiv+229 pp.**

DS-00-5 **Rune B. Lyngsø.** ***Computational Biology.*** **March 2000. PhD thesis. xii+173 pp.**

DS-00-4 **Christian N. S. Pedersen.** ***Algorithms in Computational Biology.*** **March 2000. PhD thesis. xii+210 pp.**

DS-00-3 **Theis Rauhe.** ***Complexity of Data Structures (Unrevised).*** **March 2000. PhD thesis. xii+115 pp.**

DS-00-2 **Anders B. Sandholm.** ***Programming Languages: Design, Analysis, and Semantics.*** **February 2000. PhD thesis. xiv+233 pp.**

DS-00-1 **Thomas Troels Hildebrandt.** ***Categorical Models for Concurrency: Independence, Fairness and Dataflow.*** **February 2000. PhD thesis. x+141 pp.**

DS-99-1 **Gian Luca Cattani.** ***Presheaf Models for Concurrency (Unrevised).*** **April 1999. PhD thesis. xiv+255 pp.**

DS-98-3 **Kim Sunesen.** ***Reasoning about Reactive Systems.*** **December 1998. PhD thesis. xvi+204 pp.**

DS-98-2 **Søren B. Lassen.** ***Relational Reasoning about Functions and Nondeterminism.*** **December 1998. PhD thesis. x+126 pp.**

DS-98-1 **Ole I. Hougaard.** ***The CLP(OIH) Language.*** **February 1998. PhD thesis. xii+187 pp.**

DS-97-3 **Thore Husfeldt.** ***Dynamic Computation.*** **December 1997. PhD thesis. 90 pp.**

DS-97-2 **Peter Ørbæk.** ***Trust and Dependence Analysis.*** **July 1997. PhD thesis. x+175 pp.**

DS-97-1 **Gerth Stølting Brodal.** ***Worst Case Efficient Data Structures.*** **January 1997. PhD thesis. x+121 pp.**

DS-96-4 **Torben Braüner.** ***An Axiomatic Approach to Adequacy.*** **November 1996. Ph.D. thesis. 168 pp.**

DS-96-3 **Lars Arge.** ***Efficient External-Memory Data Structures and Applications.*** **August 1996. Ph.D. thesis. xii+169 pp.**