RWTH Aachen

Department of Computer Science
Technical Report

Logics for Mazurkiewicz Traces

Martin Leucker

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2002-10

RWTH Aachen · Department of Computer Science · April 2002

Logics for Mazurkiewicz Traces

Von der Fakultät für Mathematik, Informatik und
Naturwissenschaften der Rheinisch-Westfälischen
Technischen Hochschule Aachen zur Erlangung des
akademischen Grades eines Doktors der
Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Mathematiker
Martin Leucker
aus
Kamp-Lintfort

Berichter: Prof. Dr. K. Indermark
Prof. Dr. W. Thomas

Tag der mündlichen Prüfung: 08.02.2002

Abstract

Linear temporal logic (LTL) has become a well established tool for specifying the dynamic behavior of reactive systems with an interleaving semantics and the automata-theoretic approach has proven to be a very useful mechanism for performing automatic verification in this setting. Especially alternating automata turned out to be a powerful tool in constructing efficient yet simple to understand decision procedures and directly yield *on-the-fly* model checking procedures.

While this technique extends elegantly to richer domains where the underlying computations are modeled as (Mazurkiewicz) traces, it does so only for event- and location-based temporal logics. In this thesis, we exhibit a decision procedure for LTL over Mazurkiewicz traces which generalizes the classical automata-theoretic approach to a linear temporal logic interpreted no longer over sequences but restricted labeled partial orders. Specifically, we construct a (linear) alternating Büchi automaton accepting the set of linearizations of those traces satisfying the formula at hand. The salient point of our technique is to apply a notion of independence-rewriting to formulas of the logic. Furthermore, we show that the class of *linear* and *trace-consistent* alternating Büchi automata corresponds exactly to LTL formulas over Mazurkiewicz traces, lifting a similar result from Löding and Thomas formulated in the framework of LTL over words.

Additionally, a linear temporal logic with a different flavor is introduced as *Foata linear temporal logic* ($\mathrm{LTL_f}$). It is designed for specifying properties of *synchronized* systems that comprise *clocked* hardware circuits or *Petri nets* supplied with a *maximal step semantics*.

Distributed synchronous transition systems (DSTSs) are introduced as formal models of these systems and are equipped with a *Foata configuration graph*-based semantics, which provides a link between these systems and the framework of Mazurkiewicz traces. To simplify the task of defining DSTSs, we introduce a simple calculus in the spirit of CCS.

We give optimal decision procedures for satisfiability of $\mathrm{LTL_f}$ formulas as well as for model checking, both based on alternating Büchi automata. The model checking procedure further employs an optimization which is similar to a technique known as *partial order reduction*.

Acknowledgments

First and foremost I would like to thank my supervisor Prof. Dr. K. Indermark. I am 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 Prof. Dr. W. Thomas for interesting discussions and valuable comments to this dissertation and for volunteering to review this thesis and to become a member of my dissertation committee.

I would also like to thank Prof. M. Nielson for letting me spend one month at BRICS in Aarhus, Denmark. I have benefited greatly from working with Jesper Henriksen with whom I had many instructive discussions about traces and who also has become a friend.

Moreover, I would like to thank Prof. K. Lodaya for inviting me to the Institute of Mathematical Science at Chennai, India. Besides having the opportunity to discuss my ideas with the strong group hosted there, the visit to India has been a truly enjoyable and immensely rewarding experience.

I am very grateful to everybody at Chair of Computer Science II for contributing to an outstanding and pleasant research environment that makes everyday life so enjoyable. I thank my friend Martin Lange for many interesting discussions on games, traces, and life. My special thanks go to my colleagues and friends Benedikt Bollig and Thomas Noll for many enlightening discussions about both academic and non-academic topics. I have been extremely fortunate in having the opportunity to work with them.

Finally, I would like to express my gratitude to Anja for sustaining the enormous alteration of my mood that was often correlated with finding proofs and errors within them.

Martin Leucker
Aachen, Frebruary 2002

Contents

List of Figures

Chapter 1

Introduction

Electronic devices are nowadays ubiquitous in our society. We are in touch with hardware and software systems when using the telephone, the Internet, or the cash dispenser (automated teller machine, ATM). But also typical mechanical devices like the washing machine or the car are meanwhile full of small electronic systems that optimize the washing program or the control of the anti-skid system.

Reliability is one of the crucial issues going along with the development of these electronic devices. Faults can result in enormous loss of money, when, for example, a spacecraft like the Mars Pathfinder is lost because of a communication problem, or a bug in a digital circuit like the Intel® Pentium™ is encountered when already hundred thousands of processors are on the market. But also grave problems can arise when a telephone system for a whole region or an air traffic control system fails.

System validation, i.e., the process of determining the correctness of specifications, designs, and products, is thus an increasingly important activity [Kat99]. Current practice in conventional software engineering is to check a system to a large extent by humans (so-called "peer reviewing") and by dynamic testing. A further technique to achieve reliable systems is the application of appropriate design methodologies [GHJV00]. Even when coping with many rules of good design and even when testing is applied, the designed system may contain errors. One reason is that programmers are tempted to find errors they are already aware of. It is thus crucial to use automatic procedures and tools that attempt to remove man-made mistakes from system realizations. This is the goal of *formal methods*.

Formal methods The term *formal methods* denotes the application of mathematical methods for modeling, specifying, and verifying complex hardware and software systems. Especially distributed systems, which are much more difficult to understand due to their ability to execute actions concurrently, are developed using formal

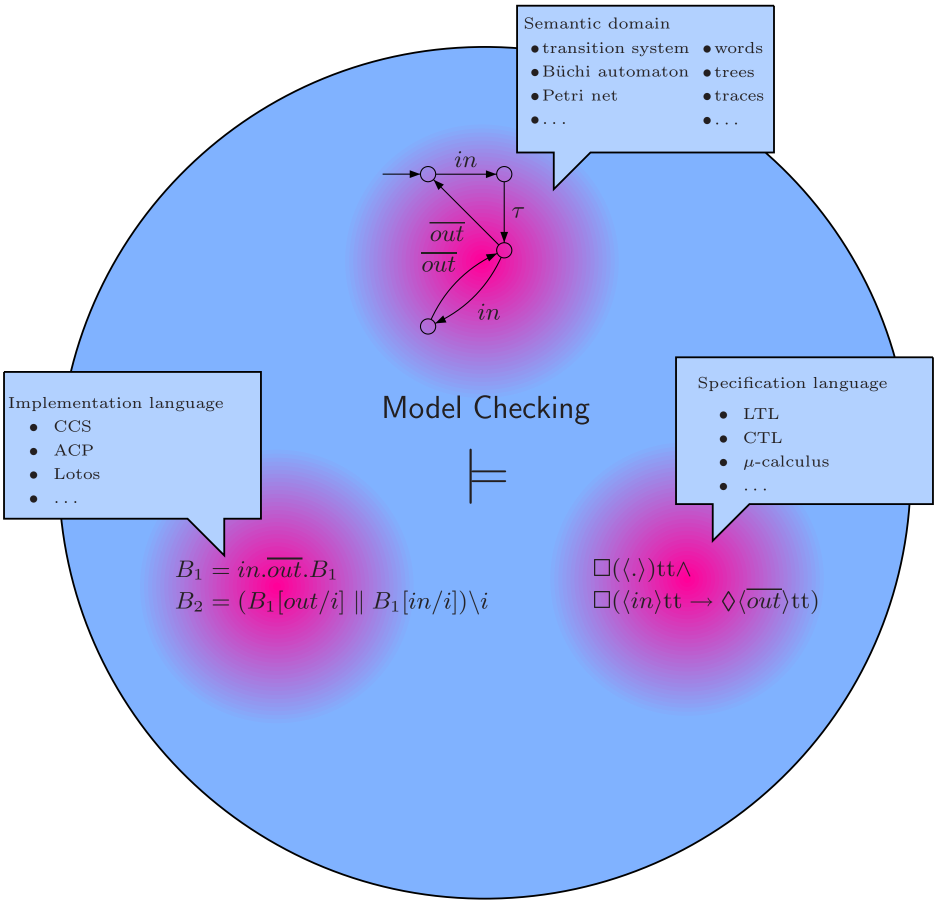


Figure 1.1: The idea of model checking

methods. Confer [BH99] for examples showing the benefits of formal methods in practice.

The process of developing a system usually starts with formulating its *specification*. This step consists of collecting a list of requirements the system-to-be should meet. The requirements are often written down in some logical formalism like monadic second-order logic [HJJ$^+$95, ABP97], linear temporal logic (LTL, [Pnu77]), computation-tree logic (CTL, [CE81]), or the μ-calculus [Koz83]. This part is depicted in right lower corner of the picture shown in Figure 1.1.

A complementary step is to define a formal *model* or *implementation* of the system, which helps to understand the system under development. Furthermore, a common and formal basis for discussing about the system is given. Last but not least, it is the basis of the realization of the system.[1] Usually, a kind of process-algebra formalism

---

[1] We speak of *implementation* although we actually might deal with an abstraction of a real

like CCS [Mil89], ACP [BV94], LOTOS [BB89], or another (semi-)formal design notion like $B(PN)^2$ [BH93], VHDL [Per91], or UML [SP99] is employed. This step is shown in the lower left corner of Figure 1.1.

The *verification* of the implemented system is a further step. Its aim is to guarantee the correctness of the functionality. In practice, verification is often more important for debugging the design instead of showing that the design is correct. This implies that verification usually proceeds in a cycle of finding errors and correcting the implementation until no further errors can be detected.

However, to be able to say that a given implementation expressed in some formal language satisfies its specification expressed in some probably different formal language, there has to be some common semantic domain to compare both sides (cf. Figure 1.1 on the preceding page). Usually, the implementation is transformed into some finite state model. Within the linear temporal framework, a formula expresses a property of a sequence of actions of the underlying system, so that we can take all sequences of executions of the finite state model to decide whether the formulated requirement holds. In the setting of branching time, a formula expresses a property of the computation tree of the underlying system. Here we can "unwind" the transition system to get an answer.

Especially for concurrent systems, the question of the "right" semantic domain is difficult to answer. Several proposals have been given and were analyzed in the past. Transition systems, Petri nets, Mazurkiewicz traces, and event structures head the list. In [SNW96], several models are compared on a formal basis.

The common idea of all models is that each is based on atomic units that are indivisible and constitute the steps from which computations are built. They differ in the level of abstraction from the underlying system. For example, so-called *interleaving models* reduce concurrency to non-determinism. It is postulated that the concurrent execution of two actions a and b is equivalent to the choice of executing a and then b or b and then a, abstracting from *true* concurrency in this way.

Within the true-concurrency approach, the execution of a concurrent system is described by means of partial order retaining the notion of concurrent or independent actions. In other words, the computations of a distributed system will be constituted by interleavings of the occurrences of causally independent actions that 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 behavior.

This observation led to the notion of Mazurkiewicz traces [DM97, DR95]. Traces can be understood as partially commutative words that model the behavior of a concurrent system. While the study of partially-commutative monoids traces back to

---

implementation. For example, we also call a protocol definition in some formal language an implementation rather than only its implementation in some low level programming language.

Cartier and Foata [CF69], it was Mazurkiewicz [Maz77] who applied these structures for modeling concurrent systems. The exciting point on Mazurkiewicz traces is that only a single representative for each equivalence class might be employed for verifying a desired property. This is the insight underlying many of the partial-order based verification methods (e.g. [Pel98, Val91]). As may be guessed, the importance of these methods lies in the fact that the computational resources required for the verification task can often be dramatically reduced.

In general, two approaches for the verification of systems can be distinguished: *model checking* and *theorem proving* [RV01]. Several case studies have shown that especially model checking admits to find errors during the design process (cf. [CW96] for an overview). In this thesis we focus on model checking.

Model checking The crucial virtue of model checking is that it proceeds automatically. The implementation as well as the specification is given to a computer tool which clarifies on its own whether the implementation satisfies its specification. No user interaction is needed for this purpose. Additionally, if an error is recognized, the tool provides a counter example showing under which circumstances the error can be generated. Model checking originates from the independent work of Emerson and Clarke [EC82] and Quielle and Sifakis [QS82]. Several prototypes of model checking tools like the Edinburgh Concurrency Workbench [Mol92], SPIN [GHP97], the symbolic model checker SMV [McM92], and Truth [LN01] have been developed and are used to demonstrate the benefits of this approach.

Model checking is especially suited when the implementation is given by (or can be translated into) a finite state model $\mathcal{M}$ and the specification is given in form of a temporal-logic formula φ; the automata theoretic approach has turned out to be fruitful here. The implementation is transformed into a corresponding automaton $\mathcal{A}_\mathcal{M}$ accepting all behaviors of the system. The specification is negated and an automaton $\mathcal{A}_{\neg\varphi}$ is constructed accepting all models of $\neg\varphi$. Next, the intersection automaton $\mathcal{A} = \mathcal{A}_\mathcal{M} \cap \mathcal{A}_{\neg\varphi}$ of $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_{\neg\varphi}$ is built. The language accepted by $\mathcal{A}$ consists of all counter examples of $\mathcal{M}$ violating φ, so that $\mathcal{A}$ can first be checked for emptiness to answer whether the implementation satisfies its specification and second be consulted to present a counter example. Note that $\mathcal{A}_\varphi$ can be tested for emptiness whether φ is satisfiable. Thus, this approach incorporates a decision procedure for satisfiability of the specification, answering the question whether the specification is contradictory.

The automata-theoretic approach for satisfiability checking was introduced in the pioneering work of Büchi [Büc60] for monadic second-order logic over words. It was transferred to the domain of temporal logics by Vardi and Wolper [VW86]. It has proven to be very useful and efficient for performing the automatic program verification.

In the last years, the general interest shifted towards alternating automata. They provide means for simple, efficient, and easy to understand decision procedures. Satisfiability algorithms for LTL over words [Var96], branching time logics [BVW94, KVW00] over finite transition systems, and Kozen's μ-calculus [Koz83] over (infinite) pushdown and prefix-recognizable graphs [KV00][2] have been defined using this variant. The idea is that the states of the automaton are constructed essentially from the subformula closure of the specification formula, and the automaton operates in a tableau-like fashion. The satisfiability problem is then solved by checking whether the constructed automaton accepts any words.

The automata-theoretic approach forms the conceptual basis of many verification algorithms. Several tools (e.g. SPIN [GHP97]) 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 main idea underlying the so-called *on-the-fly* verification techniques. To support on-the-fly checking, an automaton corresponding to a formula should be defined in a *top-down* manner, hereby allowing a part of the automaton to be analyzed while a different part is yet not constructed. This offers the possibility that the automaton for a given formula as well as a finite state system is only partly constructed viz when an initial segment already clarifies that the specification is fulfilled, or, more often in practice, contains already a counter example.

Temporal logics for traces Linear temporal logic (LTL) as proposed by Pnueli [Pnu77] has become a well established tool for specifying the dynamic behavior of distributed systems. The traditional approach towards automatic program verification is model checking specifications in LTL. A basic feature of LTL has been 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.

Indeed, it is often the case that the computations generated by a distributed system constitute Mazurkiewicz traces. It likewise 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 them for just one member of each equivalence class.

---

[2] Overviews of prefix-recognizable graphs and their monadic-second order theory can be found in [Cau96] and [Leu02]

However, it is a simple matter to specify properties within LTL for words that are not so-called *trace consistent.* From a practical point of view, it is not convenient to allow a prospective user of a verification tool to formulate such requirements. The user should be guided to specify only requirements respecting the inherent structure of an underlying system.

This problem can be tackled by developing linear temporal logics that can be interpreted directly over Mazurkiewicz traces. In these logics, every specification is guaranteed to have the "all-or-none" property and hence can be subjected to the partial-order based reduction methods during the verification process.

A number of linear temporal logics to be interpreted directly over Mazurkiewicz traces (e.g. [APP95, Thi94, TW97]) has been proposed starting with TrPTL [Thi94] that was introduced by Thiagarajan (see [MT96, TH98] for overviews). There are several possible routes towards extending linear temporal logics to traces. TrPTL is based on *locations*, where one reasons explicitly about a distribution of computing agents cooperating through some communication structure given as an alphabet distribution. Another option [APP95] 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. In contrast, in the *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.

The "right" temporal logic for traces should be equal in expressive power to first-order logic for traces (FO). It follows from [EM96] that such a logic would capture exactly those properties of LTL which have the "all-or-none" property and hence are amenable to partial-order verification. However, none of the local logics is known to be expressively equivalent to FO. Recently, it was indeed shown that linear temporal logic (without past tense modalities) is expressively equivalent to FO iff the independence adheres to a certain structure [DG01]. This lead Thiagarajan and Walukiewicz to define the configuration based LTrL [TW97], that they indeed prove equivalent to FO. LTrL was later refined by Diekert and Gastin [DG00] to a straightforward formulation of LTL for traces essentially extending Kamp's Theorem [Kam68] to the setting of traces.

While both the event based and location based logics all have elegant (exponential-time) decision procedures smoothly extending the classical automata-theoretic approach to the setting of traces, no such smooth extension exists for global logics such as LTL. The essence of this anomaly is the complications which arise as a consequence of the fact that the satisfiability problem for LTL has a non-elementary lower bound [Wal98]. However, experience [HJJ$^+$95] has shown that decision procedures can still be useful in practice despite discouraging lower bounds.

Gastin, Meyer, and Petit [GMP98a, GMP98b] do give a direct decision procedure for

LTL based on automata. However, the construction of the automaton corresponding to a given LTL specification φ proceeds by induction on φ, thus in a *bottom-up* manner. Hence, it is not an extension of the classical automata-theoretic approach, and, more importantly, it requires the construction of the full automaton, so optimizations such as on-the-fly checking cannot be applied. A further drawback is its high complexity. While an exponential blow-up is unavoidable for nested *until-formulas*, the procedure has also an exponential blow-up for every negation. Since nested *until*-formulas are rare in specifications but negations are typical for specifying unwanted behavior, this limits the practical applicability of this procedure.

In this thesis, we propose a decision procedure for LTL for traces directly extending the classical approach [Var96]. Our procedure is based on an extended subformula closure and independence rewriting of formulas of LTL. We employ this to construct a tableau-style alternating Büchi automaton accepting the set of linearizations of traces satisfying the specification at hand. In this sense, our procedure fills the missing gap for global trace logics by extending the classical approach to this remaining case. Our procedure corresponds exactly to the version introduced by Vardi [Var96] when restricted to the word case. Thus, we provide a *conservative extension* of the traditional approach, simplifying the theoretical understanding while offering a single yet efficient implementation for words as well as for traces. Furthermore, our automata can be constructed on-the-fly, which speeds up the average time and space used for checking specifications.

We furthermore present a simplified decision procedure for the case that the *until*-operator is substituted by an *eventually*-operator. We show that our decision procedure meets the known lower bound for a restricted kind of alphabets, so that it is optimal in these cases. Last but not least, for the fragment of LTL without *until*-operator and *eventually*-operator, our procedure is shown to be exponential.

While our approach is primarily used in this thesis to show how to cope with LTL for Mazurkiewicz traces, it should be noted that it might be a general technique applicable for temporal logics when partial commutation is present. For example, our method was used in [BL01b] to present a decision procedure for satisfiability of a linear temporal logic for message sequence charts.

Löding and Thomas [LT00] have shown that word languages definable by LTL formulas over words correspond to the languages of *linear* alternating Büchi automata. We prove that our construction yields a linear Büchi automaton as well. Furthermore, we show that our linear Büchi automata accept *trace-consistent* languages. Conversely, we show that the class of trace-consistent languages definable by linear alternating Büchi automata coincides with the class of languages which are definable by LTL formulas over Mazurkiewicz traces for a given dependency relation. In other words, LTL-definable trace languages correspond to languages definable by trace-consistent linear alternating Büchi automata.

Some aspects of the obtained results have been presented in [BL01a].

A linear temporal logic with a different flavor is introduced as *Foata linear temporal logic* (LTL_f). It is designed for specifying properties of *synchronized* systems. Important examples among these systems are hardware circuits that are built up by separate entities working together in parallel but which are synchronized by a global clock. Another model are *Petri nets* supplied with a *maximal step semantics* [Muk92]. The approach was partly introduced in [Leu00].

The idea underlying this model is somehow a reinforcement of the true concurrency approach. If two actions a and b are supposed to occur concurrently, we consider a concurrent system only in the configuration where no action has occurred as well as in the configuration where both actions have occurred. We refrain from considering configurations in which only one of the actions has been noticed since these might be used to differ the interleavings ab and ba.

We exhibit the notion of *distributed synchronous transition systems* (DSTSs) as a model for these hardware designs. DSTSs can be equipped naturally with a *Foata configuration graph*-based semantics which provides a link between these systems and the framework of Mazurkiewicz traces.

We give a decision procedure for satisfiability of LTL_f formulas as well as a model checking procedure, both based on alternating Büchi automata. It turns out that these procedures are as efficient as for LTL (for words) viz they are exponential in the length of the formula and linear in the size of the system and are essentially optimal. The model checking procedure employs an optimization which is similar to *partial order reduction* [Pel98].

To simplify the task of defining DSTSs, we introduce a simple calculus, which we call *synchronous process systems* (SPS) and which is inspired by Milner's CCS [Mil89] and SCCS [Mil83] but is adapted to the special nature of our underlying systems.

Outline of this thesis In the next chapter, we describe a typical scenario of a distributed system. Here we will spot the characteristics of such a system. This will guide us to define the formal notion of Mazurkiewicz traces.

In Chapter 3, we introduce Mazurkiewicz traces. We start with defining alphabets together with an independence relation. Subsequently, we will derive the notion of a trace and configurations of a trace, which serve as models for the temporal logics studied. Furthermore, we provide a link with the theory of words. This allows us to employ the rich theory developed in this domain (especially automata theory) for partial orders.

Chapter 4 recalls the notions of (alternating) Büchi automata. We identify especially *linear* and *trace-consistent* alternating Büchi automata.

To be able to formulate key results for linear temporal logics, we recall first-order logic for words as well as for traces in Chapter 5.

Chapter 7 exhibits one of the main contributions of this dissertation. We present herein the linear temporal logic LTL, which is interpreted over Mazurkiewicz traces, and give a decision procedure for checking satisfiability of LTL formulas. To simplify the overall presentation, we recall a decision procedure for LTL over words in Chapter 6, which gives the outline also in the setting of traces. We first present our procedure for the Hennessy-Milner fragment of LTL in Chapter 7.2.1 and extend this procedure to full LTL in Chapter 7.2.2.

As already mentioned, deciding satisfiability of LTL for traces is non-elementary. LTL^{-} is introduced as a fragment of LTL whose formulas can be checked for satisfiability in exponential space. We adapt our decision procedure for this fragment in Chapter 7.4.

So-called *clocked* hardware systems are studied in Chapter 8, and this chapter is another main ingredient of this thesis. We describe how to model these hardware systems as *distributed transition systems*, that, on their part, constitute traces as their executions. We introduce a simple calculus simplifying the definition of these systems. Furthermore, we define a linear temporal logic, called *Foata LTL*, that is well-suited to formulate specifications of these systems. Optimal procedures for checking satisfiability as well as for model checking are presented. To round off this framework, we end this chapter with a larger example explaining this model.

We sum up our main results in Chapter 9.

Chapter 2

Motivation

Concurrent systems play an important rôle in computer science but also in many different engineering disciplines. However, the last statement induces the fundamental question:

> "What is a concurrent system?"

We do not intend to give a formal definition. Instead we will come up with examples describing our notion of concurrent systems. Hereby, we will derive a set of parameters characterizing concurrent systems. These will guide us to develop the framework of Mazurkiewicz traces, which then captures formally the kind of concurrent systems that we will be able to handle in this thesis.

Let us consider the setting consisting of a bank, an ATM, and a customer (cf. Figure 2.1) and the well-known task of withdrawing money. When a customer tries to withdraw money from his bank account using the ATM, the machine checks whether the requested amount of money is available on the customer's account. If so, the ATM will offer the money to the customer and the withdrawal is acknowledged to the bank. Otherwise, a rejection of the customer's request is displayed to him.

What kind of ingredients can be identified in this example? We have three entities: the bank, the ATM, and the customer. The entities operate autonomously and we call them *processes*. Every process issues some *actions*. For example, the ATM

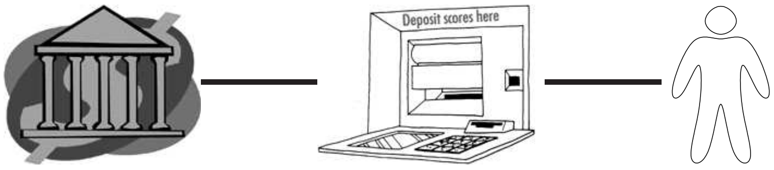


Figure 2.1: A concurrent system

generates print actions which can be read by the customer. The customer, on the other hand, communicates with the ATM. He or she enters, for example, the bank card, which is then collected by the machine. Thus, we can *observe* a *communication action* viz that one representing the collection of the card. A further action that may be observed is the lookup of the customer's bank account. Of course, a lookup action can only be observed if the bank card has been taken by the ATM. Both actions are therefore *dependent*.

Taking a closer look to the bank, the acknowledgement of the withdrawal of the money might result in an *internal* action involving a database located inside the bank. Of course, the database action is not *causally dependent* on the customer's further activities. He or she might take back his or her bank card while the database action takes place. Both actions are *independent*.

The described scenario might be modeled formally by three state transition systems for the bank, the cash dispenser, and the customer, respectively. The transitions of each system are labeled using a finite set of actions representing the internal and communication actions each process is capable of executing. Some of the actions are naturally dependent while others are independent. Observing a single execution of such a system, the adequate structure is a partial order instead of a linear order because there is no reason to distinguish the cases where the database action happens before or after taking back the bank card. It makes a difference though whether the requested money is granted or not. Each transition system may therefore be non-deterministic, if several alternatives are abstracted by non-determinism. Altogether, the overall system may be described by the set of all executions, which is then a set of partial orders.

Often, it is reasonable to abstract from internal actions and to concentrate on communication actions. Analyzing distributed systems, it is especially the communication that involves difficulties. Internal actions of each process are carried out sequentially so that conventional debugging techniques can be employed. Milner's CCS [Mil89] incorporates this view. Note however, that this view is not always adequate, as we will see in Chapter 8.

In the previous example, all communication actions are dependent. The ATM ensures that the communication actions of the bank and the ATM on one hand as well as the ones of the ATM and the customer on the other are dependent.

If we take a more complicated setting, we also obtain independent actions even when restricting to communication actions. Take for example the schematic view of the airplane, as shown in Figure 2.2 on the facing page. The airplane has several fuel tanks. Within each tank, several fuel probes are installed for measuring the fuel level. Each tank is equipped with a separate (distributed) flight data acquisition unit, which checks the fuel probes and communicates with other units. Thus, we have for each tank a system consisting of fuel probes and data acquisition unit, and

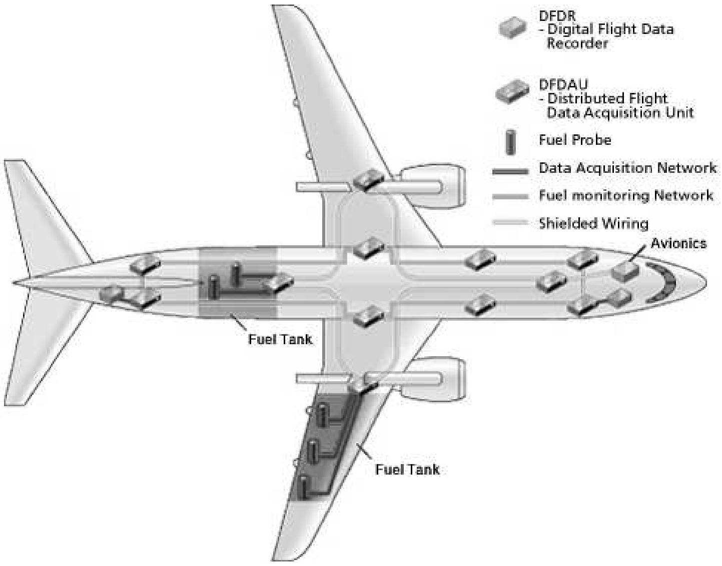


Figure 2.2: An airplane

these systems communicate independently inside.

We have now set out the scene of the systems we want to model and to study. We will meet this goal in the following chapters.

Chapter 3

Mazurkiewicz Traces

In this chapter, we define and study the semantic domain which may be employed for describing concurrent systems. We start with considering *alphabets*, which contain the atomic entities for describing atomic *actions* of our systems. *Mazurkiewicz traces*, for the sake of brevity usually just called *traces*, correspond to sequences of such atomic actions and are employed to model *executions* of our systems. Concurrent systems then may be described by sets of executions, called *languages* in this framework. Partial executions give rise to the notion of *configurations*, and executions may be analyzed with respect to their configurations.

3.1 Alphabets

We directly start with the elementary and simple notion of an alphabet.

Definition 3.1.1
An alphabet *is a non-empty finite set. The elements of* Σ *are called* actions*.*

We usually denote alphabets by Σ or Σ' and actions by $a, b, c, \ldots$ and $a', b', c', \ldots$, respectively. The elements of an alphabet, actions, are the atomic entities which may be executed by our concurrent system. As pointed out in the previous chapter, these actions can be equipped with a natural notion of *dependence* and dually also *independence*.

Definition 3.1.2
A symmetric and irreflexive relation $I \subseteq \Sigma \times \Sigma$ *is called an* independence relation *over* Σ*, and* $D := \Sigma^2 \backslash I$ *is called its* dependence relation*. The pair* (Σ, I) *is called an* independence alphabet*, and, similarly,* (Σ, D) *is called a* dependence alphabet*, respectively.*

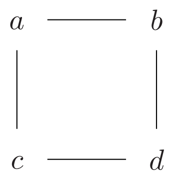


Figure 3.1: A dependence alphabet

Sometimes, we will refer to $\mathcal{I} = (\Sigma, I)$ by the name *concurrent alphabet* ([Maz88]). Let Σ be an alphabet. Then $(\Sigma, \emptyset)$ is an independence alphabet whose actions are all pairwise dependent. We call this alphabet also the *fully-dependent* alphabet. On the other hand, $(\Sigma, (\Sigma \times \Sigma) \setminus \Delta(\Sigma))$ is an independence alphabet in which all actions are independent (except each action on itself).[1] This alphabet is also called the *fully-independent* alphabet. A *transitive* dependence alphabet is a dependence alphabet (Σ, D) such that D is a transitive relation.

An independence alphabet (Σ, I) can be visualized as a graph[2] where the nodes correspond to the actions in Σ and two nodes a and b are connected iff $(a, b) \notin I$ and $a \neq b$. Thus, whenever two distinct actions are dependent they are connected by an edge. For simplicity we omit loops. In the same manner, a dependence alphabet (Σ, D) will be represented graphically by the graphical representation of the independence alphabet (Σ, I) where $I = \Sigma^2 \setminus D$. Let us fix the previous considerations in the following definition.

Definition 3.1.3
Let (Σ, D) be a dependence alphabet. The dependence graph *of (Σ, D), denoted by $\mathcal{G}(\Sigma, D)$, is the graph (V, E) where $V = \Sigma$ and*

$$E = \{(a, b) \in \Sigma \mid (a, b) \in D \text{ and } a \neq b\}$$

The dependence graph $\mathcal{G}(\Sigma, I)$ of an independence alphabet is defined as described before.

Example 3.1.4 Let us consider the independence alphabet (Σ, I) with a set of actions $\Sigma = \{a, b, c, d\}$ and the independence relation $I = \{(a, d), (d, a), (b, c), (c, b)\}$. The dependence graph $\mathcal{G}(\Sigma, I)$ is isomorphic to the one shown in Figure 3.1.

---

[1] $\Delta(R)$ is defined by $\{(q, q) \mid q \in R\}$.

[2] We use notions and notations from graph theory as usual. If necessary, please consult Appendix A.2 on page 153 for details.

Observe that the dependence graph of a fully-independent alphabet has no edges and that the one of a fully-dependent alphabet is a complete graph.[3] Given a transitive dependence alphabet, its dependence graph consists of connected components which are complete.

In our interpretation of dependence/independence alphabets, fully-dependent alphabets will be employed for describing sequential systems while fully-independent alphabets represent concurrent systems which behave autonomously, i.e. without communication or interaction. Transitive alphabets may be employed for describing a set of sequential systems which do not communicate or interact.

Observing a (complex) system from an external viewpoint, one might see that the system is capable of executing actions. Furthermore, one might recognize dependencies of these actions. Thus, considering a single set of actions (together with an independence relation) is adequate for an external view onto a system whose internal structure is not known. However, if the internal structure of a system is given, a somehow different view might be more appropriate. We can understand the system as a set of n *processes* where each process i is capable of executing actions only among Σ_i. These actions are used for communication among the processes or correspond to internal actions of a process (cf. Chapter 2). This leads us to the notion of a *distributed alphabet*, a notion due to Zielonka:

Definition 3.1.5 ([Zie87])
A distributed alphabet $\tilde{\Sigma}$ *(over* n processes*) is an* n*-tuple* $(\Sigma_1, \ldots, \Sigma_n)$ *of (not necessarily disjoint) alphabets. For* $\tilde{\Sigma}$*, we let* $\mathit{Proc}(\tilde{\Sigma})$ *denote the set* $\{1, \ldots, n\}$*.*

When the distributed alphabet is clear from the context, we simplify our notation and write *Proc* instead of $\mathit{Proc}(\tilde{\Sigma})$. In our interpretation, an action a in $\Sigma_i \cap \Sigma_j$ for two different $i, j \in \mathit{Proc}$ may be employed for a communication of the different processes i and j.

To show a natural correspondence of distributed and independence alphabets, we introduce the operators *alphabet* and *pr*. Let us fix the distributed alphabet $\tilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$. Then we let $\mathit{alphabet}_{\tilde{\Sigma}}$ denote the set $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$. Furthermore, let the domain of $\mathit{pr}_{\tilde{\Sigma}}$ range over $\mathit{alphabet}_{\tilde{\Sigma}}$ and let $\mathit{pr}_{\tilde{\Sigma}}(a)$ yield the set $\{i \in \mathit{Proc}(\tilde{\Sigma}) \mid a \in \Sigma_i\}$. We now define an *independence relation* $\mathrm{I}(\tilde{\Sigma})$ by $\mathrm{I}(\tilde{\Sigma}) = \{(a, b) \in \Sigma \times \Sigma \mid \mathit{pr}(a) \cap \mathit{pr}(b) = \emptyset\}$. It is easy to see that $\mathrm{I}(\tilde{\Sigma})$ is well defined, i.e., it is indeed an independence relation. Hence, $(\Sigma, \mathrm{I}(\tilde{\Sigma}))$ is an independence alphabet.

On the other hand, for an independence alphabet (Σ, I), we get a unique (up to the order of the alphabets) distributed alphabet $\tilde{\Sigma}$ such that $(\Sigma, I) = (\Sigma, \mathrm{I}(\tilde{\Sigma}))$ by considering maximal (totally) dependent subsets of Σ, i.e., the subsets $\Sigma_i \subseteq \Sigma$ such that $(a, b) \notin I$ for all $a, b \in \Sigma_i$.

---

[3] A graph (V, E) is *complete* iff $E = (V \times V) \setminus \Delta(V)$.

Example 3.1.6 The independence alphabet of Example 3.1.4 corresponds to the distributed alphabet $(\{a,b\},\{a,c\},\{b,d\},\{c,d\})$ and vice versa.

Note that the maximal dependent subsets of an independence alphabet correspond to the maximal D-cliques in its dependence graph representation. Recall that a set $p \subseteq \Sigma$ is called a *D-clique* iff $p \times p \subseteq D$.

Let us now extend the notion of independence and dependence to pairs of sets of actions. Given an independence alphabet (Σ, I) and two subsets $X, Y \subseteq \Sigma$ of actions, we say that X and Y are *independent* iff $X \times Y \subseteq I$. This is denoted by XIY. This means that any pair of actions from X and Y is independent with respect to (Σ, I). In a similar manner, we say that X and Y are *dependent*, iff there are two actions a and b in X respectively Y such that $(a,b) \in D$ where $D = \Sigma^2 \setminus I$. Hence, two sets are dependent iff two of their members are dependent. This will be denoted by XDY. If one of the sets X or Y is a singleton, we sometimes omit the curly braces to simplify our notation. For example, we will write aIY instead of $\{a\}IY$.

3.2 Traces

The behavior of systems may be described by the actions which are executed. The nature of a sequential system is that it can only execute one action after the other. Hence, an execution of a sequential system may be described by a sequence of actions which constitutes a linear order.

Given a concurrent system with a fixed notion of dependence, we will no longer expect the different actions of the execution to form a linear order but a partial order, as we pointed out in Chapter 2.

Therefore, we define an execution of a concurrent system to be a partial order.

Definition 3.2.1
Let Σ be an alphabet. A Σ-labeled partially ordered set *is a triple $(E, \leq, \lambda)$ such that*

- *$(E, \leq)$ is a partially ordered set (*poset*), i.e., $\leq \subseteq E \times E$ and $\leq$ is reflexive, transitive, and antisymmetric,*
- *λ is a* labeling function *from E to Σ which assigns to every element of E a* label *which is an element of Σ.*

If the alphabet Σ is clear from the context, we may omit it. The labeling function λ can be extended to subsets of E in a straightforward manner viz for $C \subseteq E$, we define $\lambda(C)$ to denote $\{\lambda(e) \mid e \in C\}$.

Example 3.2.2 Let us fix the alphabet $\Sigma = \{a,b,c,d\}$ in this example.

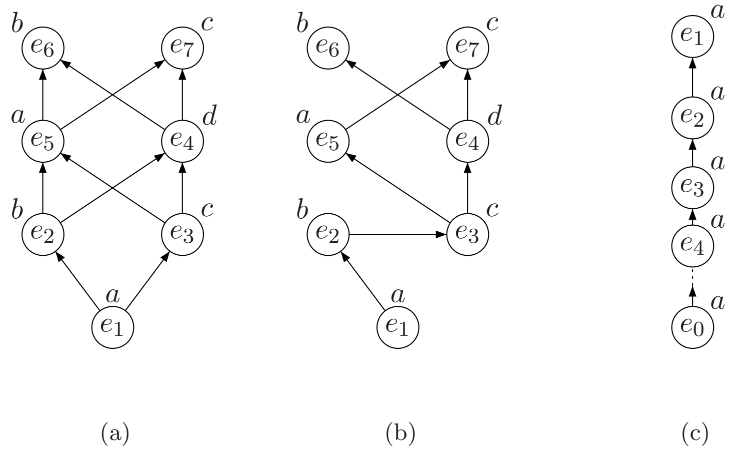


Figure 3.2: Labeled partial orders

1. Let $E = \{e_1, \ldots, e_7\}$, let $\leq$ be the reflexive and transitive closure of $\{(e_i, e_j) \mid i \in \{1, \ldots, 5\}, k = (i+1) \bmod 2,$ and, $j = i + 1 + k$ or $j = i + 2 + k\}$, and let λ be defined by $e_1, e_5 \mapsto a$, $e_2, e_6 \mapsto b$, $e_3, e_7 \mapsto c$ and $e_4 \mapsto d$. Then $(E, \leq, \lambda)$ is a poset. Its Hasse diagram is shown in Figure 3.2(a). Here, the elements of E are written within circles and their labels are written next to them. To denote that e_i is smaller than e_j with respect to $\leq$, we write $e_i \to e_j$.

2. Let us consider a similar example where again $E = \{e_1, \ldots, e_7\}$. Now, let $\leq$ be the reflexive and transitive closure of

   $$\{(e_1, e_2), (e_2, e_3), (e_3, e_4), (e_3, e_5), (e_4, e_6), (e_4, e_7), (e_5, e_7)\}.$$

   Let λ be defined as before by $e_1, e_5 \mapsto a$, $e_2, e_6 \mapsto b$, $e_3, e_7 \mapsto c$ and $e_4 \mapsto d$. Then $(E, \leq, \lambda)$ is the poset depicted in Figure 3.2(b).

3. Let $E = \{e_i \mid i \in \mathbb{N}\}$ and let $\leq \subseteq E \times E$ be defined by $e_i \leq e_j$ iff

   - $i, j \in \mathbb{N} \setminus \{0\}$ and j is less or equal to i with respect to the usual order over the naturals or
   - $i = 0$.

   Let λ send each element simply to a, i.e., for all $i \in \mathbb{N}$, let $\lambda(e_i) = a$. Then $(E, \leq, \lambda)$ is a labeled partially ordered set and its Hasse diagram is indicated in Figure 3.2(c).

For describing executions, arbitrary partial orders are too general. Let us come back to Example 3.2.2. We would like to interpret the elements of the poset as *events* of the system under consideration. A label of an event is interpreted as the action corresponding to the event. In other words, the actions executed by an underlying system are represented by unique events with corresponding action labels. We call an event e with label a also an a*-event*.

As pointed out in Chapter 2, it is reasonable to assume that our concurrent system has an initial state and, as time proceeds, executes actions. Therefore, we require a poset representing a run to have minimal elements, which denote starting points. Let us look at Figure 3.2(c) on the page before. The depicted poset has a unique starting point, the event e_0. However, before event e_4 can occur, an infinite number of events $e_5, e_6, \ldots$ has to occur, i.e., infinitely many actions have to be executed before. Assuming that every action takes a fixed amount of time, the event e_4 describes a part of the behavior of a system after an infinite amount of time. Since we do not want to deal with these situations, it is natural to require that every event of a run is preceded only by a *finite* number of events. So Figure 3.2(c) does not represent an execution.

Furthermore, a poset representing an execution of a system should respect its given fixed dependence relation over the actions. Let us assume that we have the independence relation $I = \{(a,d),(d,a),(b,c),(c,b)\}$ (see Figure 3.1 on page 16). We will not consider the poset shown in Figure 3.2(b) to be a run of our system for two reasons. First, the events e_2 and e_3 are ordered although their corresponding actions (their labels) are independent with respect to I. Second, the events e_5 and e_6 are not ordered although their actions are dependent with respect to I. So Figure 3.2(b) does not represent an execution.

We will limit the kind of partial orders we are considering by the items mentioned before and will gain the notion of *Mazurkiewicz traces*. But let us introduce some definitions before:

Definition 3.2.3
Let $(E, \leq, \lambda)$ *be a poset where* E *is countable.*[4]

- *For* $e \in E$*, we define* $\downarrow e = \{x \in E \mid x \leq e\}$ *and* $\uparrow e = \{x \in E \mid e \leq x\}$*. We call* $\downarrow e$ *the* history *of the event* e *and* $\uparrow e$ *the* future *of the event* e*.*
- *We let* $\lessdot$ *be the* covering relation *given by* $x \lessdot y$ *iff* $x \leq y$*,* $x \neq y$*, and for all* $z \in E$*,* $x \leq z \leq y$ *implies* $x = z$ *or* $z = y$*.*[5]
- *Moreover, we let the* concurrency relation *be defined as* $x \text{ co } y$ *iff* $x \not\leq y$ *and* $y \not\leq x$*.*

---

[4] Throughout this thesis, every set is assumed to be countable, i.e., there is a bijection to the natural numbers (denoted by $\mathbb{N}$) or it is finite. This general assumption is sometimes not made explicit.

[5] In other words, $\lessdot \; = \leq - \leq^2$ where $\leq^2$ denotes the relational product of $\leq$ with itself, i.e., $\leq^2 = \{(x,z) \mid \exists y \; (x,y) \in \leq \text{ and } (y,z) \in \leq\}$.

We are now ready to define the fundamental objects studied in the rest of this work.

Definition 3.2.4
A Mazurkiewicz trace *over the independence alphabet* (Σ, I) *is a* Σ*-labeled poset* $T = (E, \leq, \lambda)$ *satisfying:*

- $\downarrow e$ *is a finite set for each* $e \in E$. (T1)
- *For every* $e, e' \in E$, $e \lessdot e'$ *implies* $\lambda(e) D \lambda(e')$. (T2)
- *For every* $e, e' \in E$, $\lambda(e) D \lambda(e')$ *implies* $e \leq e'$ *or* $e' \leq e$. (T3)

Since in this thesis we only deal with Mazurkiewicz traces, we will simply speak of traces in the following. We call a trace $(E, \leq, \lambda)$ *finite*, if E is a finite set. Otherwise we call it *infinite*.

Example 3.2.5 Let us consider the independence alphabet (Σ, I) with a set of actions $\Sigma = \{a, b, c, d\}$ and the independence relation $I = \{(a, d), (d, a), (b, c), (c, b)\}$. Then, the poset shown in Figure 3.2(a) on page 19 is a trace while the one shown in Figure 3.2(b) violates (T2) and (T3). The poset shown in Figure 3.2(c) does not satisfy (T1).

Please observe that two events e and e' with independent action labels $(\lambda(e) I \lambda(e'))$ still might be ordered. However, (T2) ensures that this can only happen in the case that there is a sequence of events between e and e' with dependent labels, i.e., there are $k \geq 3$ and events $e_1, \ldots, e_k$, and for $i \in \{1, \ldots, k-1\}$, $e_i \lessdot e_{i+1}$, $\lambda(e_i) D \lambda(e_{i+1})$, $e = e_1$ and $e' = e_k$ or $e' = e_1$ and $e = e_k$. For example, the action labels a and d of the events e_1 and e_4, respectively, as shown in Figure 3.2(a), are independent but due to e_2 (or alternatively e_3) the events e_1 and e_4 are ordered.
Let us give further examples:

Example 3.2.6

1. Let (Σ, D) be a fully-dependent alphabet. Then every trace constitutes a linear order with labels from Σ.
2. Let (Σ, I) be a fully-independent alphabet. Then every trace is a disjoint union of linear orders each labeled with a single action.
3. Let (Σ, D) be a transitive dependence alphabet. Then every trace is a disjoint union of linear orders.

As mentioned in Chapter 2, non-determinism is a typical property of concurrent systems. So even if a concurrent behavior of the system is expressed by considering partial orders instead of a set of sequential orders, we usually have to describe a system not only by a single trace but a set of traces, called *language.*

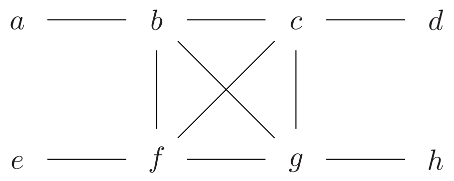


Figure 3.3: A complex dependence alphabet

Definition 3.2.7
We shall let $\mathbb{TR}(\Sigma, I)$ *denote the class of traces over the independence alphabet* (Σ, I)*. A* trace language*, usually denoted by* L*, is a set of traces, i.e.,* $L \subseteq \mathbb{TR}(\Sigma, I)$*.*

We will not distinguish between isomorphic elements in $\mathbb{TR}(\Sigma, I)$. We call two traces $T = (E, \leq, \lambda)$ and $T' = (E', \leq', \lambda')$ *isomorphic* iff there is a bijection $f : E \to E'$ which is label-preserving and order preserving with respect to $\leq$, and whose inverse mapping f^{-1} is order preserving with respect to $\leq'$. In other words, for all $e_1, e_2 \in E$, we have $\lambda(e_1) = \lambda'(f(e_1))$, and $e_1 \leq e_2$ iff $f(e_1) \leq' f(e_2)$. To simplify our notation, we write $T = T'$ not only for equal but also isomorphic T and T'.

A larger example

In the rest of this section, we will consider a larger example showing the potential benefit of traces for describing huge systems space efficiently. To simplify our presentation, we will explain this example on an intuitive basis instead of modeling the situation formally. Since we only intend to give a feeling for the power of traces, one might forgive us.

Let us consider the dependence alphabet given by the graphical representation shown in Figure 3.3. It corresponds to the distributed alphabet

$$\tilde{\Sigma} = (\{a, b\}, \{c, d\}, \{e, f\}, \{g, h\}, \{b, c, f, g\}),$$

which can easily be seen by considering the maximal cliques of the graph shown in Figure 3.3.

Now, suppose we have five agents represented by transition systems over the previous dependence alphabet (Figure 3.4 on the facing page).

We have four agents with pairwise disjoint sets of actions. The last agent is intended to synchronize the behavior of the other four agents.

Let us assume the following execution principle of the five agents: The concurrent system can execute an action a iff all agents participating in a are able to execute

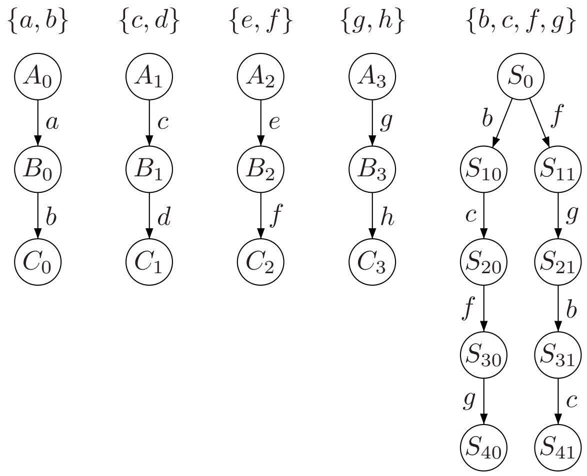


Figure 3.4: Five agents

a. Suppose the concurrent system is initially in the state $(A_0, A_1, A_2, A_3, S_0)$. Then the system may execute the action a and evolve into the state $(B_0, A_1, A_2, A_3, S_0)$. Although it is possible for the third agent to execute the action c, this is not possible for the concurrent system because the last agent has to execute the action b first.

All (sequential) executions of the system are given as maximal paths starting from the root in the graph represented in Figure 3.5 on the next page. Please note that the nodes of the graph are encoded according to the following tabular. Furthermore, we omit the state of the last agent since it is uniquely determined by the combination of s_i and t_i.

$$\begin{array}{lcl@{\qquad}lcl}
A_0,A_1 & \mapsto & s_0 & A_2,A_3 & \mapsto & t_0 \\
B_0,A_1 & \mapsto & s_1 & B_2,A_3 & \mapsto & t_1 \\
C_0,A_1 & \mapsto & s_2 & C_2,A_3 & \mapsto & t_2 \\
C_0,B_1 & \mapsto & s_3 & C_2,B_3 & \mapsto & t_3 \\
C_0,C_1 & \mapsto & s_4 & C_2,C_3 & \mapsto & t_4
\end{array}$$

Another approach to describe a system with the same behavior, i.e., with the same set of sequential executions, can be given by Petri nets. We only deal with Petri nets on an intuitive level and assume that the reader has some basic knowledge about them. See [Rei86] for an introduction to Petri nets.

Let us consider the Petri net shown in Figure 3.6 on the following page. The executions of this Petri net are also represented by Figure 3.5. Analyzing both models, either the system of the five transition systems or the Petri net, we will come to the conclusion that there are basically two runs of each system. In terms of the transition system, it depends on the last agent whether we will find the sequence $bcfg$ or

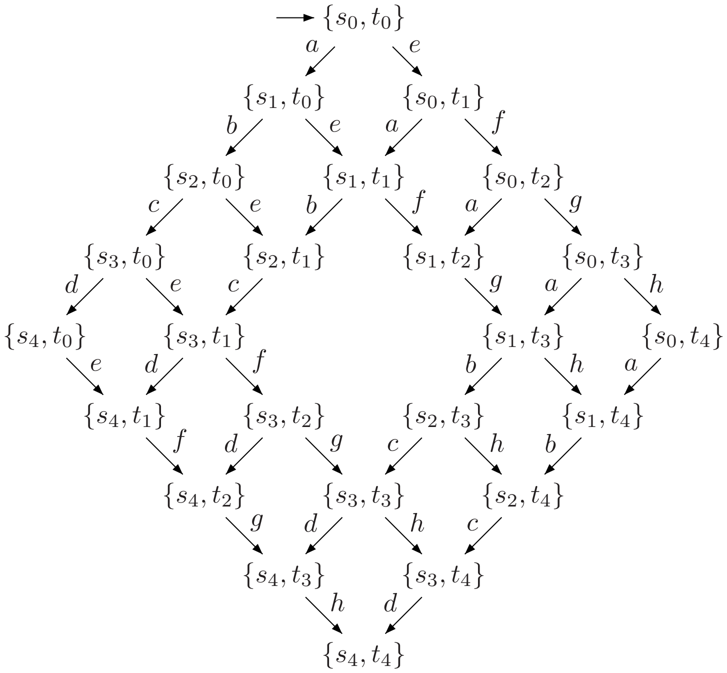


Figure 3.5: The state space of the concurrent system

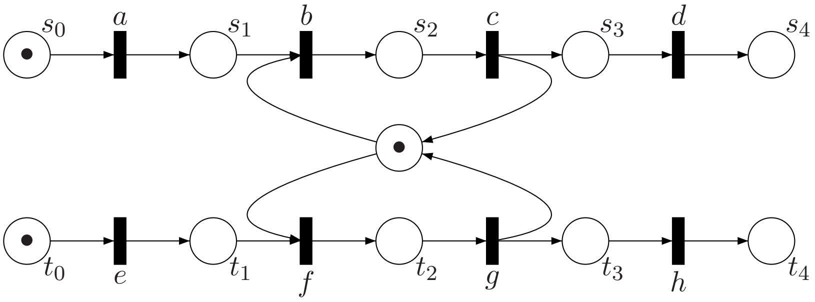


Figure 3.6: A Petri net

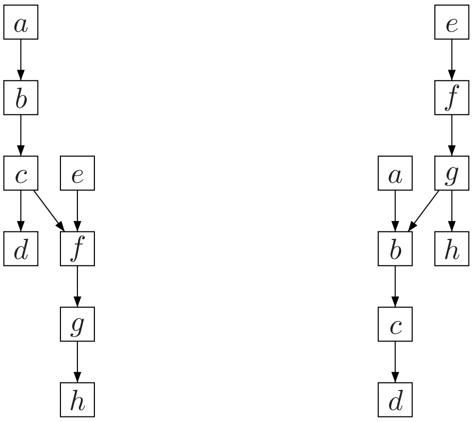


Figure 3.7: Trace representation of state space shown in Figure 3.5

$fgbc$ in the execution. Similarly, looking at the Petri net, one might distinguish two runs. The first one in which transition b fires before f, and the second one in which f fires before b.

Now, let us look at the traces of our system, given the independence alphabet before. We directly see that there are only two different traces describing all possible executions of our system as depicted in Figure 3.7. Please observe that we did not show the events but presented their labels (shown in boxes) directly.

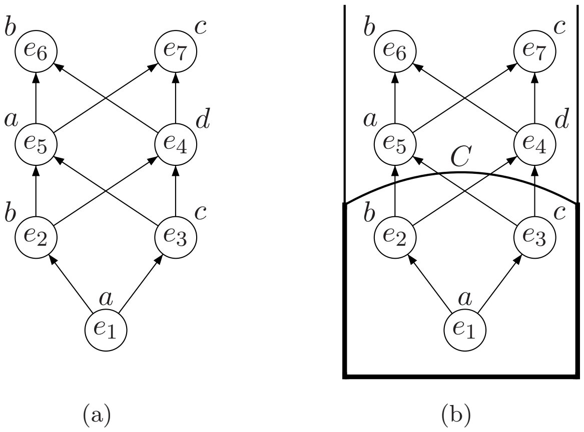
Of course, the trace representation will require less memory when stored on a computer. Apart from the dependence alphabet, less nodes and edges have to be kept in memory. Hence, traces might be a step towards the solution to the well-known *state-space explosion problem.*

3.3 Configurations and Configuration Graphs

A concurrent system is studied in terms of its executions. We want to analyze an execution while it is evolving, step by step. Given a sequence, a so-called prefix of the sequence is a part of the execution after some time elapsed. What is a similar notion in the setting of traces?

Let $T = (E, \leq, \lambda)$ be a trace over (Σ, D). We identified an event $e \in E$ to represent an execution step of our system corresponding to the action $\lambda(e)$. In our interpretation, we require that all events e' before e, i.e., $e' \in \downarrow e \setminus \{e\}$, have already been executed.

So it is natural to model a part of an execution T by a set of events C which contains for every event also its history, i.e., for all $e \in C$, we have $\downarrow e \subseteq C$. We call C a



(a) (b)

Figure 3.8: A trace and one of its configurations

configuration of our execution. Let us be more formal.

Definition 3.3.1
Let $T = (E, \leq, \lambda)$ be a trace over (Σ, D). Let C be a set of events $C \subseteq E$. The history *of C, denoted by $\downarrow C$, is defined by*

$$\downarrow C = \bigcup_{e \in C} \downarrow e.$$

A configuration *of T is a finite subset $C \subseteq E$ such that*

$$\downarrow C = C.$$

We denote by $\mathrm{conf}(T)$ *the set of all configurations of the trace T.*

Configurations will play a crucial rôle in this thesis. These and their natural relations are the objects employed for specifying and analyzing concurrent systems.

Trivially, $\emptyset$ is a configuration for any trace T. We provide a further example:

Example 3.3.2 Let us consider the trace shown in Figure 3.8(a) over the trace alphabet depicted in Figure 3.1 on page 16. The configuration $C = \{e_1, e_2, e_3\}$ is shown in Figure 3.8(b) by the polygon labeled by C. Every configuration in $\mathrm{conf}(T)$ can be represented in a similar way.

As will become apparent in Chapter 7, the formulas of the logics we are going to consider are to be interpreted over configurations of traces.

Prefixes of sequences of actions constitute sequences on their own. To lift this property towards the setting of traces, we define:

Definition 3.3.3
Let $T = (E, \leq, \lambda)$ *be a trace and* $\mathrm{conf}(T)$ *its set of configurations. Every configuration* $C \in \mathrm{conf}(T)$ induces *a finite trace* $T' = (C, \leq|_{C\times C}, \lambda|_C)$. T' *is called* prefix *of the trace* T. *We let* $\mathrm{prf}(T)$ *denote the set of prefixes of a trace* T.

The prefixes of a sequence can be linearly ordered in an obvious way. Intuitively, a prefix is greater than another prefix, iff the first describes the system at a later instance than the second one. Configurations of $\mathrm{conf}(T)$ are defined to be the trace-theoretic analogues of finite prefixes of strings. The set of configurations $\mathrm{conf}(T)$ of a trace $T = (E, \leq, \lambda)$ can be ordered in a natural way by inclusion. Given two configurations $C, C' \in \mathrm{conf}(T)$, we might say that C' is greater than C iff $C' \supset C$, hence, iff C' describes the configuration of the trace in which all events of C and further events occurred. This yields a partial order.

Furthermore, the configurations can be equipped with a transition relation $\longrightarrow_T \subseteq \mathrm{conf}(T) \times \Sigma \times \mathrm{conf}(T)$. Given two configurations $C, C' \in \mathrm{conf}(T)$, we identify C' as an *a-successor* of C, denoted by $C \xrightarrow{a}_T C'$, iff C can be augmented with an a-event yielding C':

Definition 3.3.4
Let $T = (E, \leq, \lambda)$ *be a trace and* $\mathrm{conf}(T)$ *its set of configurations. We define* $\longrightarrow_T \subseteq \mathrm{conf}(T) \times \Sigma \times \mathrm{conf}(T)$ *by* $C \xrightarrow{a}_T C'$ *iff there exists an* $e \in E$ *such that* $\lambda(e) = a$, $e \notin C$, *and* $C' = C \cup \{e\}$. *Furthermore, we define the* configuration graph *of* T, *denoted by* $\mathcal{CG}(T)$, *by* $\mathcal{CG}(T) = (\mathrm{conf}(T), \longrightarrow_T)$.

Let us consider some examples:

Example 3.3.5

1. Let us consider the trace alphabet shown in Figure 3.1 and the trace of Figure 3.8(a). Its configuration graph is presented in Figure 3.9(a). To simplify our presentation, we sometimes omit edge labels and denote the configurations by a list of its labels (see Figure 3.9(b)).

2. Let (Σ, D) be a fully-dependent alphabet. As mentioned before, every trace T over (Σ, D) constitutes a linear order. Thus, the corresponding configuration graph $\mathcal{CG}(T)$ is linearly ordered as well.

Note that the configuration graph (considered as an undirected graph) is connected for every trace over an arbitrary alphabet, even if a trace's underlying graph is not connected.

Although we have not introduced the notion of an execution or observation formally yet, we would like to bring out some insight of configuration graphs. The configuration graph represents all possible executions or observations of the trace T, in

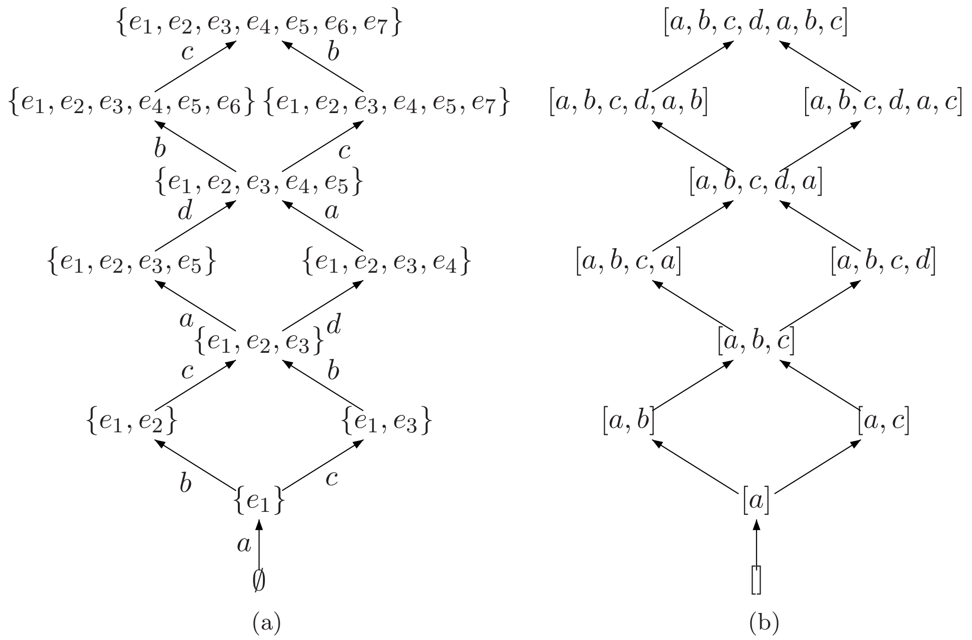


Figure 3.9: The configuration graph of the trace of Figure 3.8 (a)

other words, all executions which we consider to be equivalent. Each of its paths represents a single observation. Every (maximal) path of the configuration graph is isomorphic to a linearization of the trace T, which will be defined in the next section.

3.4 Traces and Words

In its original formulation [Maz77], Mazurkiewicz introduced traces as certain equivalence classes of words, and this correspondence turns out to be essential for our developments here. While considering traces as partial orders reveals their correspondence to partial order executions within the application domain of concurrency, their treatment in terms of (equivalence classes of) words allows the employment of the rich theory of (finite) automata. In this section, we bring out the concept of linearizations providing a link between traces and (certain) equivalence classes of words. We limit our examination to the extent that is needed for our later developments. Confer [DR95] for a thorough investigation on their relationship.

As usual, a *word* w over Σ is a sequence of elements of Σ. It is called *finite* respectively *infinite* if the sequence is finite respectively infinite. The empty word is written as ε. The *length* of a finite word $w = a_0 \dots a_{n-1}$ is n and denoted by $|w|$. The *length* of an infinite word w is ω and also denoted by $|w|$. The length $|\varepsilon|$ of ε is 0. Let Σ^* be the set of finite *words* over Σ and Σ^ω be the set of (countably) infinite words generated by Σ with $\omega = \{0, 1, 2, \dots\}$. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, which is the union of finite and infinite words. We let w and w' range over Σ^ω and u and v with or without primes range over Σ^*. We call $u \in \Sigma^*$ a *prefix* of $z \in \Sigma^\infty$ iff there is a $z' \in \Sigma^\infty$ such that $uz' = z$. Finally, we take $\mathrm{prf}(w)$ to be the set of finite prefixes of w and let $\mathrm{alph}(w)$ denote the set of actions occurring in w.

Let us first bring out a relation of traces and linear orders. The basic idea is to extend the partial order towards a linear order while keeping the well-foundedness property.[6]

Definition 3.4.1
Let $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$. *A* linearization *of* T *is a linear labeled order* $T' = (E, \leq', \lambda)$ *such that* $\leq \subseteq \leq'$ *and, for all* $e \in E$, *we have* $\downarrow e$ *(with respect to* $\leq'$*) is finite. We take* $\mathrm{lin}(T)$ *to denote the* set of linearizations *of the trace* T.

It other words, a linearization of a trace T is a linearly ordered set in which concurrent events (with respect to T) are put into an arbitrary order so that no concurrent events remain but still condition (T1) for traces holds. We can now associate with every linearization a word $w \in \Sigma^\infty$ viz the sequence of labels of the events.

Definition 3.4.2
A word $w = a_0 a_1 \cdots \in \Sigma^\infty$ *is a* linearization *of* $T = (E, \leq, \lambda)$ *iff*

- $|w| = |E|$ *and*
- *there is a linearization* $(E, \leq', \lambda)$ *of* T *such that, for all* $i \in \{0, \dots, |w| - 1\}$, $\lambda(e_i) = a_i$ *where* e_i *is the* i*-th element with respect to* $\leq'$.

Although we call both words and linearly ordered sets *linearizations*, there will be little chance of confusion since we can identify them anyway in most cases. In the remaining cases, the context will give enough information to see whether we talk about words or linearly ordered sets. Consequently, we shall also take $\mathrm{lin}(T)$ to denote the set of words which are linearizations of the trace T.

Example 3.4.3

1. The linear order depicted in Figure 3.10(b) on the next page is a linearization of the trace shown in Figure 3.10(a). Thus, the word *acbdacb* is a linearization of the given trace. *abcdacb* is a different linearization of the given trace.
2. For the fully-dependent alphabet, every trace is a linear order. Thus, every trace over the fully-dependent alphabet has a single linearization.

---

[6] Please wait for Example 3.4.4 on the following page for a motivation illustrating the benefit of requiring well-foundedness.

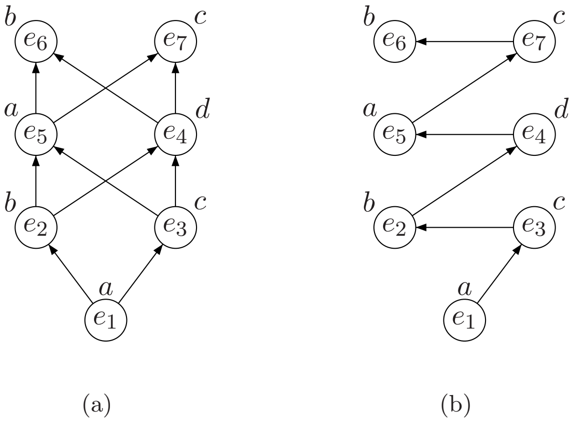


Figure 3.10: A trace and one of its linearizations

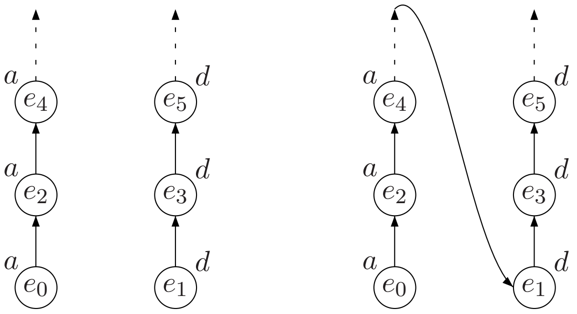
Note that we require a linearization to be well-founded to ensure that we indeed can associate a finite or ω-word with our linearization. Without requiring (T1) to hold, this might not be the case:

Example 3.4.4 Take the dependence alphabet shown in Figure 3.1 on page 16. Consider the trace given by $E = \{e_0, e_1, \dots\}$, $\leq$ given by $e_{2i} \leq e_{2j}$ and $e_{2i+1} \leq e_{2j+1}$, and λ given by $\lambda(e_{2i}) = a$ and $\lambda(e_{2i+1}) = d$, for $i, j \in \mathbb{N}$ and $i \leq j$ (see Figure 3.11(a) on the next page). The trace is partitioned into two linear orders, one always labeled by a, the other one always labeled by d. A linear order $\leq'$ can be obtained by extending $\leq$ by $e_{2i} \leq' e_{2j+1}$ for all $i, j \in \mathbb{N}$, ordering all events with an odd index before the events having an even one (Figure 3.11(b)). However, the associated word $aa \dots dd \dots$ would quit the domain of ω-words. To make our life easier, we refrain from non-well-founded linearizations.

We now wish to identify a trace with its set of linearizations. Therefore, we first show that two different traces have disjoint sets of linearizations.

Proposition 3.4.5 *Let* $T_1, T_2 \in \mathbb{TR}(\Sigma, I)$. *If* $\mathrm{lin}(T_1) \cap \mathrm{lin}(T_2) \neq \emptyset$ *then* $T_1 = T_2$.

Proof
Let us fix $T_1 = (E_1, \leq_1, \lambda_1)$ and $T_2 = (E_2, \leq_2, \lambda_2)$ and assume $w = a_0 a_1 \cdots \in \Sigma^\infty$ is an element of $\mathrm{lin}(T_1)$ as well as of $\mathrm{lin}(T_2)$. Consider linearizations $(E_1, \leq'_1, \lambda_1)$ and $(E_2, \leq'_2, \lambda_2)$ of T_1 and T_2, respectively, which both yield the word w. Then, E_1 and E_2 have the same cardinality and are linearly ordered by $\leq'_1$ and $\leq'_2$, respectively. Thus, there exists a bijection $f : E_1 \to E_2$ sending the i-th element of E_1 to the i-th element of E_2 (with respect to $\leq'_1$ and $\leq'_2$). It is a simple matter to observe that



(a) (b)

Figure 3.11: A trace and one of its extensions into a linear order

f and f^{-1} are order preserving and, furthermore, that f is label preserving. Note that the i-th events of both E_1 and E_2 are labeled by a_i, the i-th action of w. Thus, $T_1 = T_2$. $\square$

So we know that words which are linearizations for traces differ if the corresponding traces are different. Let us now check that every word is also a linearization of a trace. Therefore, we first introduce an operator yielding a trace for a given word.

Definition 3.4.6
We let $\mathrm{tr} : \Sigma^\infty \to \mathbb{TR}(\Sigma, I)$ *be defined by* $w \mapsto (E, \leq, \lambda)$ *where* $E = \mathrm{prf}(w) \setminus \{\varepsilon\}$, $\lambda(va) = a$, *and* $\leq$ *is the smallest relation such that*

- *for all* $ua, vb \in \mathrm{prf}(w)$, $ua \leq vb$ *if* $ua \in \mathrm{prf}(vb)$ *and* aDb, *and*
- $\leq$ *is closed under transitivity.*

We call $\mathrm{tr}(w)$ *the* canonical trace *of* w.

Strictly speaking, we should write $\mathrm{tr}_{(\Sigma,I)}$ instead of tr. Since we usually work with a fixed independence alphabet, this would complicate our notation in an unnecessary way. Thus, we omit the index.

Example 3.4.7 The trace obtained by tr applied to the word *acbdacb* is shown in Figure 3.12 on the following page. Note that tr(*acbdacb*) is isomorphic to the trace shown in Figure 3.10(a) on the preceding page.

Let us check that tr is well-defined.

Remark 3.4.8 For all $w \in \Sigma^\infty$, we have $\mathrm{tr}(w) \in \mathbb{TR}(\Sigma, I)$.

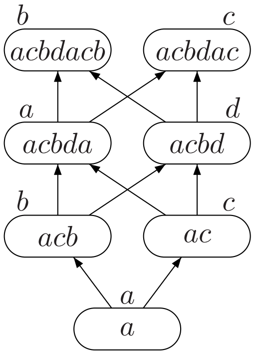


Figure 3.12: $\mathrm{tr}(acbdacb)$

Proof
We first show that $\leq$ is a partial order. Let ua and vb denote prefixes of w. Since $ua \in \mathrm{prf}(ua)$ and aDa, $\leq$ is reflexive. By definition, $\leq$ is transitive. Since the prefix relation on words is antisymmetric, $\leq$ is this as well. Thus, $\leq$ is a partial order. Let us now show the items (T1) – (T3). Since $\mathrm{prf}(ua)$ is finite for every $ua \in \Sigma^*$, every event has a finite history, so (T1) holds. Let us now consider ua and vb with $ua \leq vb$. If there is no v' with $ua \leq v' \leq vb$ and $v' \neq ua$ and $v' \neq vb$, then aDb which shows (T2). Let us now consider ua and vb with aDb. Of course, $ua \in \mathrm{prf}(vb)$ or $vb \in \mathrm{prf}(ua)$, and, by definition, $ua \leq vb$ or $vb \leq ua$, respectively, so that (T3) is satisfied. ☐

We have now an operator yielding a trace, given a word and the concept of linearizations. Let us show that our operator tr takes a linearization and delivers a trace so that the original word is one of its linearizations. Furthermore, let us show a dual property, that is, considering one of a trace's linearizations, tr yields the original trace when applied to that linearization.

Proposition 3.4.9

1. *Let* $w \in \Sigma^\infty$. *Then* $w \in \mathrm{lin}(\mathrm{tr}(w))$.
2. *Let* $T \in \mathbb{TR}(\Sigma, I)$. *Then for every* $w \in \mathrm{lin}(T)$ *we have* $T = \mathrm{tr}(w)$.

Proof
Let us show the first item. Let $w \in \Sigma^\infty$. By definition, $\mathrm{tr}(w) = (E, \leq, \lambda)$ where $E = \mathrm{prf}(w) \setminus \{\varepsilon\}$, $\lambda(va) = a$ and $\leq$ is the smallest relation such that $ua \leq vb$ if $ua \in \mathrm{prf}(vb)$ and aDb, and which is closed under transitivity. Let $\leq'$ be defined by $u \leq' v$ iff $u \in \mathrm{prf}(v)$, for all $u, v \in \mathrm{prf}(w)$. Then obviously $\leq \subseteq \leq'$, and, $\leq'$ is a linear ordering relation. Realizing that the sequence of labels of $(E, \leq', \lambda)$ yields w concludes the proof.

Let us now show the second item. Take a $w \in \Sigma^\infty$ which is in $\mathrm{lin}(T)$. By the first item of this proposition, we conclude that $w \in \mathrm{lin}(\mathrm{tr}(w))$. By Proposition 3.4.5 on page 30, we have $T = \mathrm{tr}(w)$. ☐

The operator tr induces an equivalence relation on Σ^∞ in a natural way:

Definition 3.4.10
Let (Σ, I) *be an independence alphabet. The* trace congruence relation $\equiv_{(\Sigma,I)} \subseteq \Sigma^\infty \times \Sigma^\infty$ *induced by* I *is defined by:* $w \equiv_{(\Sigma,I)} w'$ *iff* $\mathrm{tr}_{(\Sigma,I)}(w) = \mathrm{tr}_{(\Sigma,I)}(w')$*. If* $w \equiv_{(\Sigma,I)} w'$ *we say that* w *and* w' *are* trace equivalent *or* trace congruent.

We abbreviate $\equiv_{(\Sigma,I)}$ by $\equiv$ if the independence alphabet is fixed. It is a simple matter to check that $\equiv$ is an equivalence relation. Furthermore, it is easy to see that $\equiv$ is compatible with respect to (partial) concatenation of words.[7] Thus, $\equiv$ is a congruence relation.

We also obtain easily that $w \equiv w'$ iff there exists a trace T such that $w \in \mathrm{lin}(T)$ and $w' \in \mathrm{lin}(T)$: By definition, $w \equiv w'$ implies that $\mathrm{tr}(w) = \mathrm{tr}(w')$. Hence, $T = \mathrm{tr}(w)$ has the required property (cf. Proposition 3.4.9(1)). If, on the other hand, there is a T such that w and w' are elements of $\mathrm{lin}(T)$, then we know by Proposition 3.4.9(2) that $T = \mathrm{tr}(w) = \mathrm{tr}(w')$. However, there exists a trace T such that $w \in \mathrm{lin}(T)$ and $w' \in \mathrm{lin}(T)$ iff for all T: $w \in \mathrm{lin}(T)$ iff $w' \in \mathrm{lin}(T)$, a result due to Proposition 3.4.5. In other words, the equivalence relation $\equiv$ is identical to the one induced by lin. The latter is defined as expected: Two words are equivalent iff they are linearizations of the same trace.

Let us sum up the results obtained so far: Given a trace, we can obtain a word via the concept of linearizations. We further clarified that a word gives rise to a trace via the operator tr. Both directions can be carried out in a compliant way, i.e., a linearization of a trace yields the trace via tr and the trace for a given word comprises the word among its linearizations. Thus, the domain of words can be partitioned into sets of linearizations of traces, or, obtaining the same partition, by preimages of tr, and there is a natural bijection between linearizations of traces and traces.

We now consider the equivalence classes obtained by tr (or lin) in more detail. In which way do the elements of an equivalence class differ? We mentioned already that independent actions can be ordered in a different way for linearizations. But what does it exactly mean on the level of words?

First, we define an equivalence relation $\sim$ on Σ^* by $u \sim u'$ if there is a finite sequence of words $u_1, \ldots, u_n$ such that $u = u_1$, $u_n = u'$, and for each $i < n$, $u_i = vabv'$,

---

[7] Given two words of Σ^ω, their concatenation quits the domain of Σ^ω. Therefore, we let the concatenation of words in Σ^∞ to be a partial operation which is defined whenever the resulting word is in Σ^∞. We do not provide further details on the algebraic properties of traces since it is not needed for our goals. We refer to [DR95] for further details.

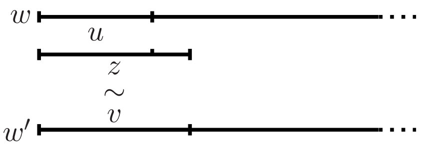


Figure 3.13: Definition of $\sqsubseteq$

$u_{i+1} = vbav'$, for some $(a, b) \in I$ and $v, v' \in \Sigma^*$. In other words, $u \sim u'$ iff u' can be obtained from u by permuting neighbored independent actions finitely often.

Example 3.4.11 With respect to the dependence alphabet represented in Example 3.1.4 on page 16, we see that

$$abcdabc \sim acbdabc \sim abcadbc \sim acbadbc \sim \ldots$$

The first equivalence can be seen by swapping the first b and c. The second and the third word are equivalent with respect to $\sim$ since c and b and the subsequent a and d are permuted. Note that every linearization yields the trace shown in Figure 3.10(a) on page 30.

It is easy to see that for finite traces, we have $u \sim u'$ iff $u \equiv u'$. Furthermore, for finite traces, we easily get that if T is a prefix of T' then every linearization of T is a prefix of a linearization of T'.

Since our main focus lies on infinite traces, we have to extend $\sim$ towards the setting of infinite traces. Taking over $\sim$ directly for infinite traces does not give a characterization as before since a finite number of permutations is in general not sufficient to obtain a second linearization from a given one: Consider again the trace shown in Example 3.4.4 on page 30 respectively Figure 3.11(a) on page 31. Then both $adad\ldots$ and $dada\ldots$ are linearizations. However, infinitely often a and d have to be "swapped" to get the words, respectively. An arbitrary infinite number of permutations, on the other hand, is too much: We might transform $ad\ldots$ to $a\ldots d\ldots$ which, for reasons of simplicity, is what we want to avoid since it is not an ω-word.

We take the following standard approach: Let $\sqsubseteq\, \subseteq \Sigma^\infty \times \Sigma^\infty$ be defined by $w \sqsubseteq w'$ iff for all $u \in \mathrm{prf}(w)$, there is a $v \in \mathrm{prf}(w')$ and a $z \in \Sigma^*$ such that $v \sim z$ and $u \in \mathrm{prf}(z)$. Stated differently, every prefix of a word w is a prefix of a word equivalent to a prefix of w'. Figure 3.13 shows the definition in a more illustrative way.

We learn that two linearizations are trace equivalent iff their prefixes are equivalent in the previous sense:

Proposition 3.4.12 *Given the notation of the previous paragraph, we have*

$$w \equiv w' \text{ iff } w \sqsubseteq w' \text{ and } w' \sqsubseteq w.$$

Proof
Assume $w \equiv w'$. Consider a finite prefix u of w. By definition, the events of $\mathrm{tr}(w)$ are prefixes of w (except the empty word), which furthermore form a configuration C_1 of $\mathrm{tr}(w)$. Since $\mathrm{tr}(w) = \mathrm{tr}(w')$, there is an isomorphism mapping C_1 to a configuration C_1' of $\mathrm{tr}(w')$. As C_1' is a finite set of prefixes of w', it also contains one of maximal length, which we call v. For v, there is the configuration C_2' which consists of all prefixes of v. It is equal to C_1' or a successor configuration of C_1', so that $C_1' \subseteq C_2'$. Altogether, we have that $\mathrm{tr}(u)$ is isomorphic to the trace $T_{C_1'}$ induced by C_1', which is a prefix of the trace $T_{C_2'}$ induced by C_2', which is isomorphic to $\mathrm{tr}(v)$. Hence, u is a linearization of $\mathrm{tr}(u)$ as well as of $T_{C_1'}$, and u is a prefix of a linearization z of $T_{C_2'}$. The second isomorphism yields $z \sim v$. Summing up, we have shown that $w \sqsubseteq w'$. Swapping the rôles of w and w', we get $w' \sqsubseteq w$.

Now suppose $w \sqsubseteq w'$ and $w' \sqsubseteq w$. We have to show that $\mathrm{tr}(w)$ and $\mathrm{tr}(w')$ are isomorphic. It is easy to see that for every finite prefix u of w, the trace $\mathrm{tr}(u)$ is a prefix of $\mathrm{tr}(w)$. By definition of $\sqsubseteq$, we know that there is a $v \in \mathrm{prf}(w')$ and $z \sim v$ with $u \in \mathrm{prf}(z)$. The latter equivalence implies that $\mathrm{tr}(z) = \mathrm{tr}(v)$, and since $u \in \mathrm{prf}(z)$, we have that $\mathrm{tr}(u)$ is a prefix of $\mathrm{tr}(z)$, hence, also of $\mathrm{tr}(v)$ and $\mathrm{tr}(w')$. Similarly, we get that every prefix of w' yields a prefix of $\mathrm{tr}(w')$ which is isomorphic to a prefix of $\mathrm{tr}(w)$. Since every trace is uniquely determined by the union of its prefixes, the desired result follows. ☐

We now know that linearizations of traces differ only by a somehow finite permutation of actions.

Let us recall that our main intention in this thesis is to study temporal logics interpreted over configurations of traces, and this, by studying certain words. We have presented a link between traces and words. A link between words and configurations of traces is still missing for using words instead of traces. We therefore introduce the concept of *run maps*, which is inspired by the previous proof.

Definition 3.4.13
Let $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$ be a trace and $w \in \Sigma^\infty$ one of its linearizations. A function $\varrho : \mathrm{prf}(w) \to \mathrm{conf}(T)$ will be called a run map *of the linearization w to the trace T iff the following conditions are met:*

- $\varrho(\varepsilon) = \emptyset$.
- $\varrho(u) \xrightarrow{a}_T \varrho(ua)$ *for each* $ua \in \mathrm{prf}(w)$.
- *For every $e \in E$, there exists some $u \in \mathrm{prf}(w)$ such that $e \in \varrho(u)$.*

Strictly speaking, we should write $\varrho_{w,T}$ instead of ϱ. For the sake of brevity, we omit the index.

A run map—if it exists—relates prefixes of words to configurations respecting the prefix order and the order on configurations. Every run map for a word w to a trace

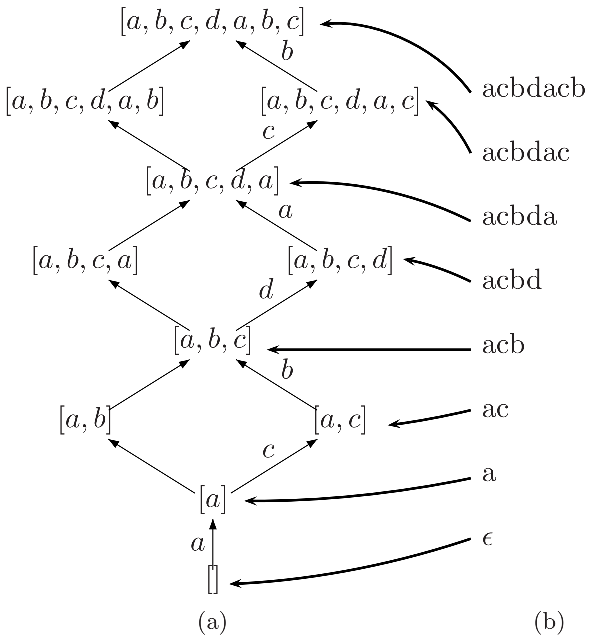


Figure 3.14: A trace and one of its run maps

T induces a path of the configuration graph whose sequence of edge labels yields w. Let us consider an example:

Example 3.4.14 Consider the part of the meanwhile well-known configuration graph shown in Figure 3.14(a). A (part of a) run map for the word $(acbd)^\omega$ is denoted by the arrows.

It is a simple matter to show that the run map of a linearization is unique.

Proposition 3.4.15 *Let* $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$ *and* $w \in \Sigma^\infty$ *be a* linearization *of* T*. Then any two run maps* ϱ *and* ϱ' *are equal.*

Let us now check that a run map exists for every given trace T and linearization w. Even more, for the straightforward generalization of the definition of a run map towards an arbitrary word w and a given trace T, we see that there exists a run map iff w is a linearization of T.

Proposition 3.4.16 *Let* $T = (E, \leq, \lambda) \in \mathbb{TR}(\Sigma, I)$ *and* $w \in \Sigma^\infty$*. Then there is a run map* $\varrho : \mathrm{prf}(w) \to \mathrm{conf}(T)$ *iff* $w \in \mathrm{lin}(T)$*.*

Proof
Assume that w is a linearization of T. Then $\mathrm{tr}(w) = T$ and we can assume that

$E = \mathrm{prf}(w) \setminus \{\varepsilon\}$, $\lambda(va) = a$ and $\leq$ is the smallest transitive relation satisfying for all $ua, vb \in \mathrm{prf}(w)$, $ua \leq vb$ if $ua \in \mathrm{prf}(vb)$ and aDb (cf. Definition 3.4.6 on page 31). Define ϱ by $u \mapsto \{u' \mid u \in \mathrm{prf}(u)\}$. Obviously, ϱ is a function from $\mathrm{prf}(w)$ to $\mathrm{conf}(T)$. Easy to see is that, $\varrho(\varepsilon) = \emptyset$ and $\varrho(u) \xrightarrow{a}_T \varrho(ua)$ for each $ua \in \mathrm{prf}(w)$. Since $\varrho(u)$ contains u, there is for $u \in E$ some $u \in \mathrm{prf}(w)$ such that $u \in \varrho(u)$. Hence, ϱ is a run map.

Now assume that there is a run map ϱ from a word $w \in \Sigma^\infty$ to a trace T. We show that there is an isomorphism between $\mathrm{tr}(w)$ to T and conclude that w is a linearization of T.

Since ϱ is a run map, there is for every $e \in E$ some $u \in \mathrm{prf}(w)$ such that $e \in \varrho(u)$ where E denotes the set of events of T. Mapping every event e to the shortest u yields the required isomorphism θ: Since for every $e \in E$, there exists some $u \in \mathrm{prf}(w)$ such that $e \in \varrho(u)$, we know that θ is total. As $\varrho(u) \xrightarrow{a}_T \varrho(ua)$ for each $ua \in \mathrm{prf}(w)$, we have that for every two different events e and e', the image of θ is different, and that for every u, there is an event e (the one added to the configuration $\varrho(u)$) that yields u under θ. Hence, θ is injective as well as surjective. It is a routine matter to check that θ is compatible with the partial order and labeling function of T. $\square$

Let us indicate a difference between configurations obtained by elements of $\mathrm{tr}(w)$ for a word w and configurations obtained via the run map for a prefix of w, which sometimes causes slight confusion. We provide an example: acb is a prefix of the word $acbdacb$. $\mathrm{tr}(acbdacb)$ is shown in Figure 3.12 on page 32. The configuration obtained by $\varrho(acb)$ contains the events a, ac, and acb while the configuration $\downarrow acb$ only contains the events a and acb.

Let us now come to a last observation for run maps. If we consider two linearizations w and w' of a trace T with run maps ϱ and ϱ', respectively, we see that for every prefix u of w and u' of w' we have $\varrho(u) = \varrho'(u')$ iff $u \equiv u'$ (iff $u \sim u'$).

We have now set out the scene for dealing with traces by means of words. To simplify our forthcoming investigations, we simplify our notation: Instead of $\mathrm{tr}(w)$ we write in the following T_w and instead of $\varrho(u)$ we write C_u.

We end this section with the general convention that, from now on, we only study infinite traces.

3.5 Trace Languages

Regularity is one of the key notions for studying classes of languages. Regular languages can usually be described by some kind of finite automaton giving evidence for a collection of objects to be *realizable* in an intuitive sense. The aim of this section is to recall notions of regularity for word as well as trace languages to the extent needed here (cf. [Tho90a] for a comprehensive overview).

Let us begin with the notion of trace-consistent languages.

Definition 3.5.1
We call a word language $L \subseteq \Sigma^\omega$ trace consistent *iff for all words* $w, w' \in \Sigma^\omega$ *with* $w \equiv w'$, *it holds* $w \in L$ *iff* $w' \in L$.

Note that the set of linearizations of any trace language (also called the *linearization of a trace language*) is a trace-consistent language.

Recall that a subset $L \subset \Sigma^\omega$ is called a *regular language* or *ω-regular language* iff L is a finite union of sets $U.V^\omega$ where $U, V \subseteq \Sigma^*$ are regular sets of finite words. As expected, $U.V$ denotes the set $\{uv \mid u \in U, v \in V\}$ and V^ω is the set $\{v_1 v_2 \ldots \mid v_i \in V \text{ for all } i \in \mathbb{N}\}$.

A simple notion for regularity of trace languages is obtained by considering languages of their linearizations.

Definition 3.5.2
We call a trace language $L \subseteq \mathbb{TR}(\Sigma, I)$ regular, *iff* $\bigcup\{\text{lin}(T) \mid T \in L\}$ *is a regular ω-language.*

We learned in the previous section that every word can be used to derive a trace via the operator tr. Thus, a language L_w of words yields a trace language L_t when applying the operator tr to every word. If L_w is trace-consistent, it is clear that the set of linearizations of L_t is identical to L_w. We conclude:

Remark 3.5.3 Trace-consistent regular languages can be identified with regular trace languages.

A different model characterizing regular trace languages was given by Zielonka [Zie87]. It uses the notion of a distributed alphabet and of *asynchronous automata*. Its benefit is that the language accepted by an asynchronous automaton is a trace-consistent regular language and that for every trace-consistent regular language, there is an asynchronous Büchi automaton accepting this language. This automaton model inspired the model we use for modeling hardware systems studied in Chapter 8.

Chapter 4

Automata for Trace Languages

In this chapter, we introduce the necessary tools for simplifying the construction of decision procedures for the logics studied in this thesis. Following a long tradition originating from Büchi [Büc62], we employ automata as devices recognizing structures satisfying the formula at hand. In our case, we recognize (linearizations of) trace languages which are models for the underlying formulas. For these automata, it is easy to decide whether the accepted language is empty or not yielding the answer of the underlying satisfiability question.

We start by recalling the notion of Büchi automata in Section 4.1. They are the most prominent device with a finite memory employed for studying infinite sequences of actions.

We proceed with defining alternating automata, which have their origin in [BL80, CKS81]. Vardi has given several examples showing the benefit of employing alternating automata for defining decision procedures for satisfiability for temporal logics. [Var96] provides a good overview of the field of alternating Büchi automata together with an application for checking satisfiability of linear temporal logic (LTL) over words. We will apply alternating automata for variants of linear temporal logic interpreted over Mazurkiewicz traces.

We present our notion of alternating automata in Section 4.2. However, we follow the style of Löding and Thomas [LT00]. It is equivalent to Vardi's notion of alternating automata with respect to expressiveness and complexity of checking emptiness but fits nicer into the global picture of automata theory.

In Section 4.2.4, we introduce linear automata, which form a subclass of alternating automata characterizing exactly the languages definable by LTL formulas (over words) [LT00].

We end this chapter with considering trace-consistent alternating Büchi automata, which are well-prepared to be acceptors for (linearizations of) trace languages.

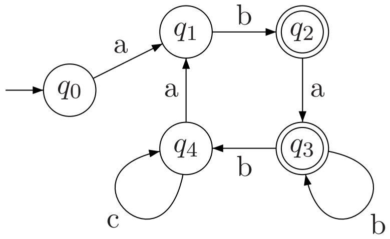


Figure 4.1: A (graphical representation of a) Büchi automaton

4.1 Büchi Automata

Büchi automata were first introduced by Büchi in [Büc62] for obtaining a decision procedure for the monadic second-order theory of structures with one successor. Let us establish the key concepts of this kind of automata to the extent needed in our thesis. For a thorough introduction to Büchi automata we refer to [Tho90a]. We start directly with their definition:

Definition 4.1.1
A (non-deterministic) Büchi automaton *(BA) over an alphabet* Σ *is a tuple* $\mathcal{A} = (Q, \delta, q_0, F)$ *such that*

- Q *is a finite non-empty set of* states*,*
- $q_0 \in Q$ *is the* initial state*,*
- $F \subseteq Q$ *is a set of accepting states, and*
- $\delta : Q \times \Sigma \to 2^Q$ *is the* transition function*.*

Let us fix a Büchi automaton $\mathcal{A} = (Q, \delta, q_0, F)$ for the rest of this section.

A Büchi automaton may be represented as an edge-labeled directed graph. Its nodes are the states and an edge labeled by $a \in \Sigma$ leads from a node (state) $q \in Q$ to a node (state) $q' \in Q$ iff $q' \in \delta(q, a)$. The initial state is marked with an incoming arrow. A final state, on the other hand, is identified by a second circle around the node. Figure 4.1 shows an exemplifying Büchi automaton over the alphabet $\Sigma = \{a, b, c\}$.

The automaton operates on infinite input words. The idea of its behavior is that it chooses (non-deterministically) a possible successor state in $\delta(q, a)$, provided it is in the state q and reads an action a. Of course, the automaton starts in its initial state.

Definition 4.1.2
A run *of $\mathcal{A}$ on a word $w = a_0a_1 \ldots \in \Sigma^\omega$ is a function $\rho : \mathbb{N} \to Q$ such that $\rho(0)$ yields the initial state q_0 of the automaton and $\rho(i+1) \in \delta(\rho(i), a_i)$ for all $i \in \mathbb{N}$.*

Sometimes, we represent a run ρ only by its sequence of images. For example, a run of the automaton shown in Figure 4.1 on the word $a(baba)^\omega$ is given by the sequence $q_0(q_1q_2q_3q_4)^\omega$. A run on $ababc^\omega$ is given by $q_0q_1q_2q_3q_4^\omega$.

For automata over finite words, acceptance is defined according to the last state visited by the run. When the words are infinite, there is no such thing as a "last state". Instead, acceptance is defined according to the set of states that are visited infinitely often. For a run ρ, we define the set of *states visited infinitely often*, denoted by $\mathrm{Inf}(\rho)$, by

$$\mathrm{Inf}(\rho) = \{q \in Q \mid \text{ for infinitely many } k \in \mathbb{N}\text{, we have } \rho(k) = q\}$$

Note that for every run ρ, the set of states visited infinitely often is non-empty. However, a run is accepting iff final states are visited infinitely often.

Definition 4.1.3
A run ρ of $\mathcal{A}$ is accepting *iff* $\mathrm{Inf}(\rho) \cap F \neq \emptyset$.

Let us consider Figure 4.1 on the facing page. The run denoted by $q_0(q_1q_2q_3q_4)^\omega$ on the word $a(baba)^\omega$ is accepting since the final states q_2 and q_3 are visited infinitely often. The run on $ababc^\omega$ denoted by $q_0q_1q_2q_3q_4^\omega$ is not accepting since the only state visited infinitely often is q_4, which is not a final state.

Definition 4.1.4
The language *accepted or recognized by a Büchi automaton $\mathcal{A}$ consists of all words of Σ^ω for which an accepting run of the automaton exists and is denoted by $\mathcal{L}(\mathcal{A})$. A language $L \subseteq \Sigma^\omega$ is* definable *by a Büchi automaton iff there exists a Büchi automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = L$.*

It is well-known that the class of languages definable by Büchi automata is precisely the class of regular languages:

Theorem 4.1.5 ([Büc62])
A language $L \subseteq \Sigma^\omega$ is definable by a Büchi automaton iff L is regular.

A Büchi automaton is *deterministic* iff $|\delta(q, a)| = 1$ for every $q \in Q$ and $a \in \Sigma$. Thus, the transition function can be understood as a mapping to Q instead of a mapping to the power set of Q ($\delta : Q \times \Sigma \to Q$). In other words, the successor state is *determined* by the current state and the action. Note that non-deterministic Büchi automata are strictly more expressive than deterministic Büchi automata. This means, there is a language which is definable by a non-deterministic Büchi automaton but which is not accepted by any deterministic Büchi automaton.

If an automaton $\mathcal{A}$ represents all structures satisfying a formula φ, complementation of an automaton is an interesting operation on automata. Here, the *complement of an automaton* $\mathcal{A}$ is an automaton $\overline{\mathcal{A}}$ such that $\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})}$. If $\mathcal{L}(\mathcal{A})$ consists of all structures satisfying φ, then $\overline{\mathcal{L}(\mathcal{A})}$ comprises all structures satisfying the negation of φ.

Büchi introduced a complementation construction which was based on a complicated combinatorial argument. Indeed, it was one of the key contributions of [Büc62] to show that the class of Büchi automata is closed under complementation, a result by far non-trivial. However, his complementation involves a double exponential blow-up. It was shown by Michel [Mic88] that the complementation of Büchi automata has a lower bound of $2^{\mathrm{O}(n \log n)}$ where n denotes the number of states of the original automaton. In [SVW85], Sistla et al. suggested a construction which yields an automaton with $2^{\mathrm{O}(n^2)}$ states. The first optimal construction was given by Safra in [Saf88]. Klarlund presented a different solution in [Kla91]. Both solutions are difficult to implement though. A further approach for complementation of Büchi automata is via the use of alternating automata (cf. [Tho99] and [KV01]). The resulting algorithms are optimal and easy to implement. We summarize this short overview about complementation in the following theorem:

Theorem 4.1.6 ([Saf88])
Let $\mathcal{A} = (Q, \delta, q_0, F)$ be a Büchi automaton with $|Q| = n$. Then there is an effective algorithm constructing $\overline{\mathcal{A}} = (Q', \delta', q'_0, F')$ where $|Q'| = 2^{\mathrm{O}(n \log n)}$.

Let us now turn to a further important question about Büchi automata. If an automaton represents all structures satisfying a formula, one can check the automaton's language for emptiness to decide whether the formula has a model. Thus, checking emptiness is an important problem for Büchi automata. Let us understand that this question can be answered in linear time, which is a result due to Emerson and Lei [EL85]. However, we should clarify with respect to which measure this is true:

Definition 4.1.7
Let $\mathcal{A} = (Q, \delta, q_0, F)$ be a Büchi automaton.

- *The* unlabeled graph *of $\mathcal{A}$ is the graph (V, E) such that*
  - *$V = Q$ and*
  - *$(q, q') \in E$ iff there is an action a such that $q' \in \delta(q, a)$.*
- *The* size *of $\mathcal{A}$, denoted by $|\mathcal{A}|$, is defined as $|V|$ plus $|E|$ where (V, E) is assumed to be the unlabeled graph representation of $\mathcal{A}$.*

In other words, the size of a Büchi automaton is measured in the number of its states and transitions where transitions differing only in the action label are counted only once.

Remark 4.1.8 For an unlabeled directed graph with n nodes, the number of edges is bounded by n^2. Thus, the size of a Büchi automaton with n states is in $\mathrm{O}(n^2)$. However, for our applications, a Büchi automaton usually has a fixed maximal number c of possible successor states. In this case, the number of edges is bounded by $c \cdot n$ so that the size of the Büchi automaton is in $\mathrm{O}(n)$. We therefore often identify the size of a Büchi automaton with its number of states. The careful reader might forgive us.

Now we could state the previously mentioned result.

Theorem 4.1.9 ([EL85])
The non-emptiness problem for Büchi automata is decidable in linear time.

Proof
Given a Büchi automaton $\mathcal{A}$, consider its unlabeled graph. Since the number of final states is finite, every accepting run visits infinitely often final states iff it visits a single final state infinitely often. As the number of states is finite, the graph of the automaton then has to contain a cycle on which a final state occurs. Furthermore, one of the states on this cycle must be reachable from the initial state of the automaton.

A depth-first-search algorithm can construct a decomposition of the graph into strongly connected components [CLR90] in linear time with respect to the number of nodes plus the number of edges of the graph.

The language of the automaton is non-empty iff from a component that contains the initial state a non-trivial connected component that intersects with the set of final states non-trivially is reachable. Hence, Büchi automata non-emptiness can be reduced to graph reachability.

The latter is known to be an operation which can be carried out in linear time by a standard depth-first-search algorithm [CLR90] (see Theorem A.3.1 on page 154). □

With respect to the space needed for checking emptiness, it is easy to see that only a finite number of states must be stored at the same time, provided our underlying computational model offers non-determinism. The result can even be strengthened in by showing that this problem is complete for NLOGSPACE which was shown by [VW94].

Theorem 4.1.10
The non-emptiness problem for Büchi automata is NLOGSPACE*-complete.*

Proof

As pointed out in the proof of Theorem 4.1.9 on the preceding page, emptiness of a Büchi automaton $\mathcal{A}$ can be reduced to a reachability question for the unlabeled graph of $\mathcal{A}$. Starting from the initial state q_0 of $\mathcal{A}$, an algorithm has to find a state q which is a final state and which is (non-trivially) reachable from q. Thus, a corresponding non-deterministic Turing machine starts with guessing a final state q and writing q_0qq on its working tape (coded binary). Now, it proceeds twice like the Turing machine described in the proof of Theorem A.3.2 on page 154, once to find q starting at q_0, and, if successful, a second time to find q again, but now starting at q. Observe that totally 3 states have to be kept on the tape. Thus, the problem is in NLOGSPACE.

On the other hand, the graph accessibility problem can be reduced to emptiness of a Büchi automaton in a straightforward way. Since the former was proven to be NLOGSPACE-hard [Jon75], we are done. □

4.2 Alternating Büchi Automata

Alternating automata extend non-deterministic automata by universal choices. The transition function denotes no longer a set of possible next states but a (positive) Boolean combination. Alternation in the context of automata was first studied in [BL80] and [CKS81]. In this section, we recall the notion of alternating automata along the lines of [Var96] where alternating Büchi automata are used for model checking LTL over strings. However, we modified the definition of a run to reflect the ideas presented in [LT00] which simplifies several proofs. Furthermore, we mention some of their properties.

4.2.1 The Concept

As we will see, the transition function of our automata will no longer yield a set of successor states but positive Boolean combination thereof. We therefore need the notion of positive Boolean formulas:

Definition 4.2.1

For a finite set X of variables, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas *over X, i.e., the smallest set such that*

- $X \subseteq \mathcal{B}^+(X)$
- $\mathrm{tt}, \mathrm{ff} \in \mathcal{B}^+(X)$
- $\varphi, \psi \in \mathcal{B}^+(X) \Rightarrow \varphi \wedge \psi \in \mathcal{B}^+(X), \varphi \vee \psi \in \mathcal{B}^+(X)$

In the following, we usually assume that every positive Boolean formula is in disjunctive normal form and reduced with respect to idempotency and commutation. Hence, for a set X with $|X|$ elements, the size of $\mathcal{B}^+(X)$ is bounded by $2^{2^{|X|}}$. This can easily be seen by considering the formulas as sets (disjunctions) of sets (conjunctions).

Let us discuss this bound in more detail: Every Boolean formula with n variables induces a Boolean function from $\mathbb{B}^n$ to $\mathbb{B}$ where $\mathbb{B} = \{0, 1\}$. If the formula is positive, the resulting Boolean function will be monotone. We call a Boolean function f *monotone* iff for $(x_1, \ldots, x_n) \leq (x'_1, \ldots, x'_n)$, we have $f(x_1, \ldots, x_n) \leq f(x'_1, \ldots, x'_n)$. Here, we let $(x_1, \ldots, x_n) \leq (x'_1, \ldots, x'_n)$ iff for all $i \in \{1, \ldots, n\}$, $x_i \leq x'_i$ where, as expected, $0 \leq 1$. It is easy to see that the number of all Boolean functions is 2^{2^n}. However, it is well-known that not all Boolean functions are monotone (e.g. the *complement* function which sends 0 to 1 and vice versa). Thus, we should ask for a smaller bound of the number of monotone Boolean functions.

Giving a closed formula for the number of monotone Boolean functions with n variables is known as *Dedekind's problem*. It was first considered by Dedekind in 1897 [Ded69]. Till today, there is no concise closed-form for this number. Its values are known for n up to 8.[1] It was shown in [Kle69] that the n-th Dedekind number is

$$2^{\binom{n}{\lfloor n/2 \rfloor}(1+\eta(n))}$$

where $0 < \eta(n) < c(\log n/n)$ for an appropriate constant c. Thus, the n-th Dedekind number is not in $2^{\mathrm{O}(p(n))}$ for any polynomial function p. Let us end this small excursion by learning that from the point of view of complexity theory, we get no significant improvement of our previously known bound of 2^{2^n}.

Let us continue with introducing further notions needed before we are able to define our kind of automata. We say that a set $Y \subseteq X$ *satisfies* (or is a *model* of) a formula $\varphi \in \mathcal{B}^+(X)$ iff φ evaluates to tt when the variables in Y are mapped to tt and the members of $X \backslash Y$ are mapped to ff. A model is called *minimal* if none of its proper subsets is a model. For example, $\{q_1, q_2\}$ as well as $\{q_1, q_3\}$ are minimal models of the formula $(q_1 \wedge q_2) \vee (q_1 \wedge q_3)$ whereas $\{q_1, q_2, q_3\}$ is a model which is not minimal. Note that the minimal model of tt is the empty set and that ff has no model at all. A variable q is *essential* for a formula $\varphi \in \mathcal{B}^+(X)$ iff there is a minimal model Y of φ containing q. For example, q_1 is essential for $q_1 \vee (q_1 \wedge q_2)$ while q_2 is not. It is easy to see that for every formula φ, there is a formula φ' having the same minimal models and comprising only essential variables. For convenience, we (usually) deal with positive Boolean formulas (in disjunctive normal form) in which every variable is essential.

---

[1] `http://www.mathpages.com/home/kmath030.htm`

Let us consider a Büchi automaton $\mathcal{A} = (Q, \delta, q_0, F)$. For a state q and an action a, let $\{q_1, \ldots, q_k\} = \delta(q, a)$ be the set of possible next states of the automaton when it is in state q reading a. The key idea for alternation is to describe the non-determinism by the formula $q_1 \vee \cdots \vee q_k \in \mathcal{B}^+(Q)$. Hence, we write $\delta(q, a) = q_1 \vee \cdots \vee q_k$. If $k = 0$ we write $\delta(q, a) = \mathrm{ff}$. An alternation is introduced by allowing an arbitrary positive Boolean formula of $\mathcal{B}^+(Q)$. Let us be more precise:

Definition 4.2.2
An alternating Büchi automaton $\mathcal{A} = (Q, \delta, q_0, F)$ *over an alphabet* Σ *is a tuple such that*

- Q *is a finite nonempty set of* states*,*
- $q_0 \in Q$ *is an* initial state*,*
- $F \subseteq Q$ *is a set of accepting states, and*
- $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ *is a* transition function*.*

Remark 4.2.3 Later in our construction, logical formulas will take over the rôle of states. Therefore, we should formally distinguish between a disjunction of formulas and a disjunction of states. However, to simplify our presentation, we identify these disjunctions when the context makes clear which one is meant. In particular, given a formula φ in disjunctive normal form, $\varphi = \bigvee \bigwedge \varphi_{ij}$ where no φ_{ij} is a (top level) disjunction or conjunction, we identify φ with the positive Boolean combination of states φ_{ij}. Measuring the complexity of a construction, it is sometimes necessary to distinguish disjunctions of formulas and disjunctions of states, though. For example, $\varphi \vee \psi$ counts as a single state when understood as formula but as two states when understood as a Boolean combination of states. Then we sometimes write $\mathrm{st}(\varphi)$ to denote $\{\varphi_{ij} \mid \varphi = \bigvee \bigwedge \varphi_{ij}\}$, the Boolean combination of states.

Let us fix the alphabet $\Sigma = \{a, b\}$ and the following alternating Büchi automaton for illustrating further concepts of alternating Büchi automata.

Example 4.2.4 Let $\mathcal{A} = (Q, \delta, q_0, F)$ over the alphabet Σ defined by

- $Q = \{q_0, q_1, q_2, q_3\}$ is the set of states,
- q_0 is the initial state,
- $F = \{q_2\}$ is the set of accepting states, and
- $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$, the transition function, is given by

$$\begin{array}{rclcrcl}
\delta(q_0, a) & = & q_1 \wedge q_2 & \qquad & \delta(q_2, a) & = & q_1 \wedge q_2 \\
\delta(q_0, b) & = & (q_1 \wedge q_2) \vee (q_1 \wedge q_3) & & \delta(q_2, b) & = & q_1 \wedge q_2 \\
\delta(q_1, a) & = & \mathrm{tt} & & \delta(q_3, a) & = & \mathrm{ff} \\
\delta(q_1, b) & = & q_1 & & \delta(q_3, b) & = & \mathrm{ff}
\end{array}$$

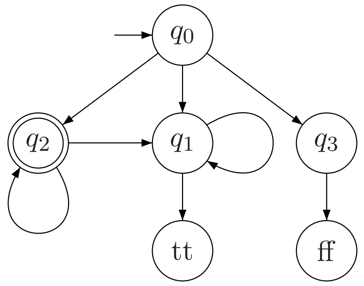


Figure 4.2: A transition graph of an alternating Büchi automaton

Then $\mathcal{A}$ is an alternating Büchi automaton.

We are now going to develop graphical representations for alternating Büchi automata. The key point to visualize is of course the transition function of the automaton. For a transition of a Büchi automaton of the form $\delta(q, a) = \{q_1, \ldots, q_k\}$, we simply introduced k edges labeled by a from q to $q_1, \ldots, q_k$. Let us adopt but also adapt this idea to define the notion of the transition graph of an alternating Büchi automaton.

Definition 4.2.5
Let $\mathcal{A} = (Q, \delta, q_0, F)$ be an alternating Büchi automaton.

- *The* transition graph *of $\mathcal{A}$ is the graph (Q, E, F) such that $(q, q') \in E$ iff there is an action a such that for $\delta(q, a) = \bigvee \bigwedge q_{ij}$, the state q' equals one q_{ij}.*[2]
- *The* size *of $\mathcal{A}$, denoted by $|\mathcal{A}|$, is defined as $|Q|$ plus $|E|$ where (Q, E, F) is assumed to be the transition-graph representation of $\mathcal{A}$.*

Note that Remark 4.1.8 on page 43 for Büchi automata holds similarly for alternating Büchi automata. That is, the size of E for an alternating Büchi automaton is bounded by n^2 if n is the size of Q (the number of states of the automaton). Since we usually have a fixed number of disjunctions and conjunctions, we often forget to consider the size of E when measuring the size of an automaton.

Figure 4.2 shows the transition graph of the automaton given in Example 4.2.4 on the facing page. Final states are highlighted with a second circle around the node.

However, while the previous representation will turn out to be useful to define a certain subclass of alternating automata, it does not suffice to represent an alternating Büchi automaton completely. Remember that for alternating Büchi automata, a transition yields a Boolean combination of states. Representing a transition of the form $\delta(q, a) = \bigvee \bigwedge q_{ij}$ just by edges from q to q_{ij}—as we did before—does not

---

[2] silently considering tt and ff as states here

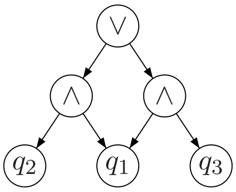


Figure 4.3: A graph representation of a Boolean formula

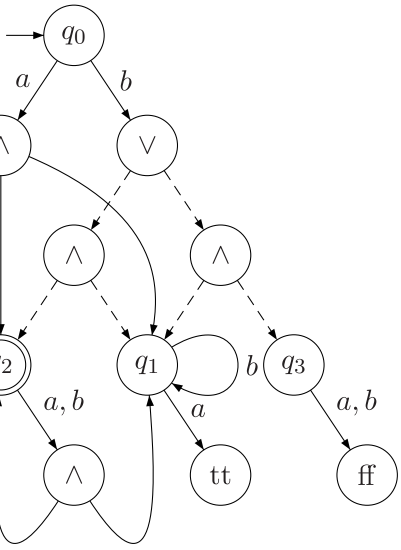


Figure 4.4: A graphical representation of an alternating Büchi automaton

show the Boolean structure. A Boolean formula (over states) can be represented as a graph by its natural tree representation. The latter can be simplified towards a directed acyclic graph by sharing the leafs with the same state. Instead of a formal definition, we just provide an example: Figure 4.3 shows the graph representation of the formula $(q_1 \wedge q_2) \vee (q_1 \wedge q_3)$. Now, we represent a transition of the form $\delta(q, a) = \bigvee \bigwedge q_{ij}$ by adding an edge labeled by a from q to the root of the graph representation of $\bigvee \bigwedge q_{ij}$. The graphical representation of an alternating Büchi automaton is now the union of the graph representations of its transitions where nodes labeled with the same state are identified. We call this graph the *graphical representation* of an alternating Büchi automaton (as opposed to the term “transition graph” which visualizes the transition function of an automaton in an abstracted way). Figure 4.4 shows the graphical representation of the automaton presented in Example 4.2.4 on page 46. Dashed lines are used to identify the graph of Figure 4.3 within the whole graph.

Like a Büchi automaton, an alternating Büchi automaton operates on infinite input words. The idea of its behavior is a little bit more complicated than for the non-deterministic version. Suppose the automaton is in the initial state q_0 and reads an action a. Then, it chooses (non-deterministically) a minimal set of states U satisfying the formula $\delta(q_0, a)$ and sends a copy of itself to every state of U. These copies proceed similarly as before when reading the next input action. Each copy in state q chooses (non-deterministically) a minimal set of states satisfying the formula $\delta(q, b)$ (supposing that b is the next input action). To minimize the coming effort, the copies unite their models to derive the next "level" of states to start with for the next input action.

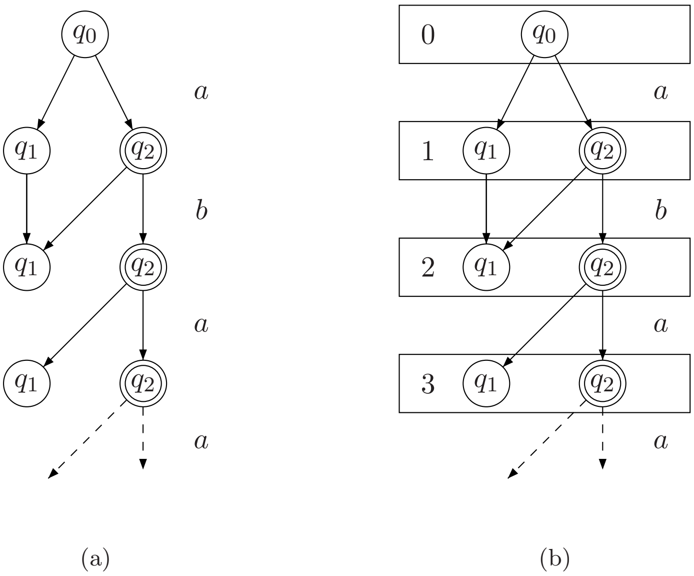
Thus, a *run* on an infinite word is no longer a sequence but a labeled directed acyclic graph. The label $l(v)$ of a node v reflects one of the current states of the automaton, and the edges show transitions of the automaton with respect to the input string. Hence, this graph should have a unique "root" labeled with q_0. Furthermore, it has to be divisible into "levels" $i \in \mathbb{N}$ corresponding to the i-th input letter. Every node except the root must have a "predecessor". For a node v of level i, the labels of nodes of level $i+1$ connected with v should further be a model for the transition in state $l(v)$ reading the i's letter so that indeed a copy of the automaton proceeds in the states required to satisfy the state $l(v)$ with the given input. More precisely:

Definition 4.2.6
A run *on an infinite string* $w = a_0 a_1 \ldots \in \Sigma^\omega$ *is a* Q*-labeled directed acyclic graph* (V, E) *such that there exist labelings* $l : V \to Q$ *and* $h : V \to \mathbb{N}$ *which satisfy the following properties:*

- $h^{-1}(0) = \{v\}$ *with* $l(v) = q_0$.
- $E \subseteq \bigcup_{i \in \mathbb{N}} (h^{-1}(i) \times h^{-1}(i+1))$.
- *For every* $v' \in V$ *with* $h(v') \geq 1$, $\{v \in V \mid (v, v') \in E\} \neq \emptyset$.
- *For every* $v, v' \in V$, $v \neq v'$, $l(v) = l(v')$ *implies* $h(v) \neq h(v')$.
- *For every* $v \in V$, $\{l(v') \mid (v, v') \in E\}$ *is a minimal model of* $\delta(l(v), a_{h(v)})$.

Figure 4.5(a) on the next page shows a (part of a) run of the automaton $\mathcal{A}$ of Example 4.2.4 on page 46 on the word $w = abaa\ldots$. Its levels are emphasized in Figure 4.5(b). Verify that the requirements for a run are indeed satisfied.

Since ff has no model, there is no run in which ff occurs. If the transition function yields tt for a given state q and input action a, the minimal model satisfying tt is the empty set. Thus, the automaton stops processing the corresponding branch (see Figure 4.5(a)). This implies that a run of an automaton on an infinite word can be finite.



(a) (b)

Figure 4.5: An exemplifying run

For Büchi automata, a run is accepting, if infinitely many final states are visited. If a run captures the behavior of several copies of the automaton at the same time, we require that every path of the run visits infinitely many final states of the automaton, unless the path is finite and thereby ending in a state whose transitions yield tt.

Definition 4.2.7
A run (V, E) on $w = a_0a_1 \ldots \in \Sigma^\omega$ is accepting *if every maximal infinite path, with respect to the labeling l, visits at least one final state infinitely often. The* language *$\mathcal{L}(\mathcal{A})$ of an automaton $\mathcal{A}$ is determined by all words for which an accepting run of $\mathcal{A}$ exists.*

Note that every maximal finite path ends in a node $v \in V$ with $\delta(l(v), a_{h(v)}) = \mathrm{tt}$. The run shown in Figure 4.5(a) is accepting since q_2 is a final state.

The transition function δ of an alternating Büchi automaton can be extended to $\check{\delta}$ applicable to Boolean combinations of states in the following way:

$$\begin{array}{lcl}
\check{\delta}(\mathrm{tt}, a) & = & \mathrm{tt} \\
\check{\delta}(\mathrm{ff}, a) & = & \mathrm{ff} \\
\check{\delta}(\varphi \vee \psi, a) & = & \check{\delta}(\varphi, a) \vee \check{\delta}(\varphi, a) \\
\check{\delta}(\varphi \wedge \psi, a) & = & \check{\delta}(\varphi, a) \wedge \check{\delta}(\varphi, a)
\end{array}$$

Sometimes it is convenient to allow an initial positive Boolean formula instead of a single initial state. Thus, $\mathcal{A} = (Q, \delta, \varphi, F)$ for $\varphi \in \mathcal{B}^+(Q)$. The idea is that the automaton can start in a set of states described by the formula. The notion of a run is modified in the expected manner: Using the notation of Definition 4.2.6 on page 49, $l(h^{-1}(0)) = \{l(v) \mid v \in h^{-1}(0)\}$ must be a minimal model of φ and for all $v, v' \in h^{-1}(0)$ with $v \neq v'$, we have $l(v) \neq l(v')$. Using the extended transition function, an alternating Büchi automaton with an initial formula can be translated into one with a single initial state accepting the same language: Let $\mathcal{A}' = (Q', \delta', q_0, F)$ with $Q' = Q \uplus \{q_0\}$,[3] $\delta'(q_0, a) = \check{\delta}(\varphi, a)$, and $\delta'(q, a) = \delta(q, a)$ for $q \in Q$ and $a \in \Sigma$. Then it is easy to see that $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}')$ coincide.

We can then easily extend a transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ to $\hat{\delta} : \mathcal{B}^+(Q) \times \Sigma^* \to \mathcal{B}^+(Q)$ by $\hat{\delta}(q, \varepsilon) = q$ and $\hat{\delta}(q, au) = \hat{\delta}(\check{\delta}(q, a), u)$ for $a \in \Sigma$ and $u \in \Sigma^*$. It is a simple matter to verify that for every automaton $\mathcal{A} = (Q, \delta, q_0, F)$ and every infinite word $uw \in \Sigma^\omega$ with finite prefix $u \in \Sigma^*$, the automaton $\mathcal{A}$ accepts uw iff $\mathcal{A}(\hat{\delta}(q_0, u))$ accepts w, where $\mathcal{A}(\varphi)$ denotes the automaton $\mathcal{A}$ with initial formula φ instead of initial state q_0. Note that for sake of brevity, we usually write δ instead of $\hat{\delta}$.

It is obvious that every Büchi automaton can be turned into an alternating Büchi automaton accepting the same language by representing the set of possible successors for each state and input action as a disjunction, as mentioned before. If the Büchi automaton is deterministic, the disjunction consists of a single state and can also be considered to be a conjunction. An alternating Büchi automaton for which the transition function is restricted to yield only conjunctions is called a *universal Büchi automaton*.

Remark 4.2.8 For every deterministic Büchi automaton $\mathcal{A}$ there is a universal Büchi automaton $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. The size of $\mathcal{A}'$ is the same as the size of $\mathcal{A}$.

Vice versa, given an alternating Büchi automaton $\mathcal{A}$, it is possible to construct a Büchi automaton $\mathcal{A}'$ accepting the same language. However, this construction involves an exponential blow-up. The idea of the construction is that the states of $\mathcal{A}'$ represent different levels of the run tree of $\mathcal{A}$. When the Büchi automaton reads the next input action, it guesses the next level in the run tree of the alternating automaton. However, $\mathcal{A}'$ has to keep track of visited final states. Therefore, a level is split into states which are on a path on which a final state was seen recently and in the other states. Thus, a state of $\mathcal{A}'$ will be a pair of subsets of states of $\mathcal{A}$, the second component of the pair holds the states which hit a final state recently, the first the others. The result is due to [MH84]. However, we slightly adapted the proof to meet our notion of runs.

Theorem 4.2.9 ([MH84])
For every alternating Büchi automaton $\mathcal{A}$, there is a Büchi automaton $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. The size of $\mathcal{A}'$ is exponential in the size of $\mathcal{A}$.

---

[3] $Q' = Q \cup \{q_0\}$ and $q_0 \notin Q$

Proof

Let $\mathcal{A} = (Q, \delta, q_0, F)$ be an alternating Büchi automata. We are going to define $\mathcal{A}' = (Q', \delta', q_0', F')$ accepting the same language. We let $Q' = 2^Q \times 2^Q$. The initial state is $q_0' = (\{q_0\}, \emptyset)$. Thus, its first component contains the initial state of $\mathcal{A}$ and its second component is the empty set since no final state has been visited yet.[4] We let the set of final states F' be $\{\emptyset\} \times 2^Q$. The idea is that successor states of the current states from the first component are shifted into the second if they are final states. Thus, the empty set in the first component identifies that "enough" final states have been seen. Therefore, these states are final in $\mathcal{A}'$. Let us now define the transition function δ'. For a pair (U, V) of sets of states and an action a let δ' yield the pairs (U', V') defined as follows:

- case $U \neq \emptyset$: Intuitively, this case means that there are states in U for which no final state was seen recently. Let $X, Y \subseteq Q$ be minimal sets satisfying the transitions requested by the states of respectively U and V reading the input symbol, i.e. $X \models \bigwedge_{q \in U} \delta(q, a)$ and $Y \models \bigwedge_{q \in V} \delta(q, a)$. We put non-final states of X in the first component and the final states in the second. Furthermore, we add all members of Y to the second component except the ones which are also in the first component. Thus, we set $U' = X - F$ and $V' = (X \cap F) \cup (Y - U')$.
- case $U = \emptyset$: Let $Y \subseteq Q$ such that $Y \models \bigwedge_{q \in V} \delta(q, a)$. Since for all states in U we have seen a final state (the reason why $U = \emptyset$) we put all states into U' except the ones which are final states: $U' = Y - F$ and $V' = Y \cap F$. Thus, we are looking for final states again.

Note that we identify an empty conjunction with tt, so that $(\emptyset, \emptyset) \in \delta'((\emptyset, \emptyset), a)$. Observe that the two components are indeed a partition of each level in the run dag of the alternating automaton. A careful analysis shows that visiting a state having the empty set in its first component infinitely often guarantees visiting final states in the corresponding run of the alternating automaton infinitely often and vice versa.

Let $n = |Q|$. The states of $\mathcal{A}'$ are elements of $2^Q \times 2^Q$. Thus, their number is bounded by $2^n 2^n = 2^{2n}$. However, since the components have an empty intersection, we get a bound of

$$\begin{aligned} & \textstyle\sum_{k=0}^{n} \binom{n}{k} \left(\sum_{l=0}^{k} \binom{k}{l} \right) \\ = & \textstyle\sum_{k=0}^{n} \binom{n}{k} 2^k \\ = & 3^n \\ = & 2^{n \log 3} \end{aligned}$$

□

As a conjunction has a unique (minimal) model, the previous proof shows:

---

[4] If q_0 is also a final state, it is a matter of taste to define $q_0' = (\emptyset, \{q_0\})$ instead.

Corollary 4.2.10 *For every universal alternating Büchi automaton $\mathcal{A}$ there is a deterministic Büchi automaton $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. The size of $\mathcal{A}'$ is exponential in the size of $\mathcal{A}$.*

4.2.2 Emptiness of Alternating Büchi Automata

The transformation of an alternating Büchi automaton $\mathcal{A}$ into a Büchi automaton $\mathcal{A}'$ with an exponential blow-up implies that emptiness of $\mathcal{A}$ can be decided in exponential time and polynomial space with respect to the number of states of $\mathcal{A}$: Just apply the emptiness test for the exponentially larger $\mathcal{A}'$ (cf. Theorem 4.1.10 on page 43).

Corollary 4.2.11 *Emptiness for alternating Büchi automata is decidable in exponential time and polynomial space.*

However, one might think that there is an improved emptiness test by considering the alternating automaton directly. Let us convince ourself that this is not the case (unless P equals PSPACE). The following result can be found in [Var96]. However, we give a different and more direct proof.

Theorem 4.2.12
Emptiness for alternating Büchi automata is PSPACE *complete (with respect to the number of states).*

Proof
It remains to show PSPACE-hardness. Our proof proceeds by a reduction of the PSPACE-complete problem IN-PLACE ACCEPTANCE (cf. Section A.5 on page 155) to the emptiness problem of alternating Büchi automata. IN-PLACE ACCEPTANCE is the problem whether a given deterministic Turing machine $\mathcal{M}$ accepts a given input x using only $|x| + 1$ cells on the working tape. Let the Turing machine be given by $\mathcal{M} = (Q, \Gamma, \Delta, q_0, \delta, F)$ where

- Q is a finite set of states,
- Γ is a finite input alphabet (thus, $x \in \Gamma^*$),
- Δ is a finite working alphabet (with $\Gamma \subset \Delta$),
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states, and
- $\delta : Q \times \Delta \to Q \times \Delta \times \{\mathsf{l}, \mathsf{r}, \mathsf{s}\}$ is the transition function.

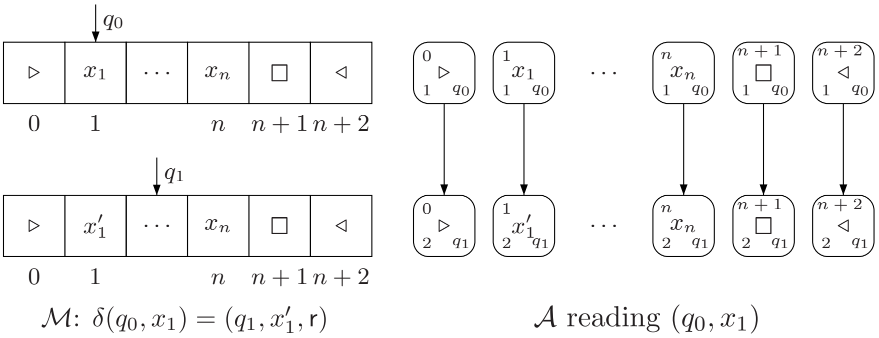


Figure 4.6: Encoding a Turing machine by an alternating Büchi automaton

Suppose the Turing machine is in the state q and the symbol under the head is a. Then the transition function δ yields the next state of the machine, a symbol replacing a, and a direction in which the head of the Turing machine has to move (l means left, r means right, and s means stay).

First we note that we can change our machine so that it never leaves the space already occupied by x plus three further cells. This can be achieved by modifying $\mathcal{M}$ in the way that it first adds two new symbols $\triangleright$ and $\triangleleft$ left and right of $x\square$, respectively, where $x\square$ is x plus the blank symbol of the Turing machine. Furthermore, we add transitions that, whenever $\mathcal{M}$ reads $\triangleright$ or $\triangleleft$, it rejects the input.[5] The benefit of this modification is that the position of the head of the Turing machine is always between 0 and $n+2$ assuming that x is on positions 1 to n. Note that this modification does not change the language consisting of the words accepted in place and that it can be carried out in linear time adding a constant to the size of $\mathcal{M}$. From now on, assume that $\mathcal{M}$ has this restricted form.

The key idea for defining the alternating Büchi automaton $\mathcal{A}$ simulating the computation of the Turing machine is to represent each cell of the Turing machine's working tape together with its current symbol as well as the position of the head and the current state of $\mathcal{M}$ as a separate state of $\mathcal{A}$. Consequently, we set $Q' = \{0, \ldots, n+2\} \times \Delta \times \{0, \ldots, n+2\} \times Q$. A tuple (k, c, l, q) then means that cell k of $\mathcal{M}$ contains the symbol c, the head of $\mathcal{M}$ is at position l, and $\mathcal{M}$ is in state q. Figure 4.6 shows on the upper left half the initial configuration of the (modified) Turing machine where $x = x_1 \ldots x_n$ is the input word. On the upper right side, we can see for each cell a state of the automaton. To optimize the presentation, the tuple (k, c, l, q) is shown in a two-dimensional fashion.

The automaton operates on a sequence of possible transitions of the Turing machine. We let the alphabet Σ consist of input values for δ, i.e. $\Sigma = Q \times \Delta$. The transitions of $\mathcal{A}$ have to mimic the transitions of $\mathcal{M}$. Consider a transition $\delta(q, a) = (q', b, m)$ of

---

[5] That means it stays in a non-final state at the same position.

$\mathcal{M}$ and a state (k, c, l, q''). If $k = l$ then the head of the Turing machine is over the current cell. If further $a = c$ and $q = q''$ the transition is applicable. If q' is a final state $\mathcal{M}$ is done and so should $\mathcal{A}$. Thus, we let $\mathcal{A}$ move from (k, c, l, q'') to tt in this case. If q' is not a final state, we let (k, c, l, q'') evolve to (k, b, l', q') where $l' = l - 1$, $l' = l + 0$, or $l' = l + 1$ if respectively $m = \mathsf{l}$, $m = \mathsf{s}$, or $m = \mathsf{r}$. If a and c or q and q'' differ, the requested transition is not applicable and we let (k, c, l, q'') evolve to ff. A cell currently not involved in the requested transition just updates the head component and the state component in the same manner as before, or is replaced by tt, if q' is a final state. The behavior is visualized in Figure 4.6 on the facing page: On the left hand side, we show the effect of the transition $\delta(q_0, x_1) = (q_1, x_1', \mathsf{r})$ to the working tape. On the right hand side, we show in which way this transition yields a new level in the run tree of the automaton.

We sum up the transition function for $\mathcal{A}$ using the notation as before:

$$\delta'((k, c, l, q''), (q, a)) \quad = \quad \begin{cases} \mathrm{tt} & \text{if } k = l, a = c, q = q'', q' \in F \\ (k, b, l', q') & \text{if } k = l, a = c, q = q'', q' \notin F \\ \mathrm{ff} & \text{if } k = l \text{ but } a \neq c \text{ or } q \neq q'' \\ \mathrm{tt} & \text{if } k \neq l, q = q'', q' \in F \\ (k, c, l', q') & \text{if } k \neq l, q \neq q'', q' \notin F \end{cases}$$

The initial formula of $\mathcal{A}$ corresponds to the initial configuration of our Turing machine:

$$\begin{aligned} q_0' \quad := \quad & (0, \rhd, 1, q_0) \wedge \\ & \quad (1, x_1, 1, q_0) \wedge \ldots \wedge (n, x_n, 1, q_0) \wedge (n+1, \Box, 1, q_0) \wedge \\ & \qquad (n+2, \lhd, 1, q_0) \end{aligned}$$

where $x = x_1 \ldots x_n$. Starting in q_0', the automaton $\mathcal{A}$ mimics a transition of $\mathcal{M}$ in the transitions of the (conjunction of the) states, reading the transitions of $\mathcal{M}$ as input actions. If the Turing machine $\mathcal{M}$ accepts the input x in place, there is a possible run of the constructed automaton ending successfully in tt in every branch. If $\mathcal{M}$ rejects (by diverging) or does not accept in place, every run of $\mathcal{A}$ will be infinite. We set the final states F' of $\mathcal{A}$ to the empty set, so that the corresponding run is not accepting. Thus, $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $\mathcal{M}$ accepts the input x in place. Note that $|Q'| = (n+3)^2|\Delta||Q| = c(n+3)^2$ and that our construction can be carried out in polynomial time. This concludes the proof. □

The previous proof allows us to draw some further conclusions.

Corollary 4.2.13 *Emptiness for universal Büchi automata is decidable in exponential time and is* PSPACE*-complete.*

Proof

Observe that we only constructed conjunctions in the proof of Theorem 4.2.12 on page 53. Thus, together with Corollary 4.2.11, the result follows. □

An *alternating automaton* is the straightforward extension of finite automata towards alternation [Var96], and *universal automata* are alternating automata for which the transition function is restricted to conjunctions.

Corollary 4.2.14 *Emptiness for alternating automata or universal automata is decidable in exponential time and is* PSPACE-*complete.*

Proof
In the proof of Theorem 4.2.12 on page 53, we constructed an automaton accepting infinite sequences of Turing machine transitions for which a prefix yields an accepting configuration of the Turing machine. Restricting to finite words, thus finitely many transitions of the Turing machine, does not change the simulation and the acceptance. $\square$

4.2.3 Weak Alternating Büchi Automata

As pointed out already in the section dealing with Büchi automata (Section 4.1 on page 40), complementation is one of the important operations on automata when defining decision procedures for logics. Similar as in the case of Büchi automata, we recall the basic ideas and results for complementing Büchi automata since they allow us to prove our forthcoming constructions of decision procedures to be correct.

Although complementation of alternating Büchi automata is conceptually simpler and has a lower complexity than for the case of Büchi automata, it is not straightforward. However, the possibility to use positive Boolean combinations of states simplifies this task a little.

Let us consider an input word aw and a transition $\delta(q_0, a) = q_1 \vee q_2$ of an alternating Büchi automaton $\mathcal{A}$ with initial state q_0. Suppose that there is an accepting run on aw. Then the automaton has to guess an accepting run on the given input word by selecting either q_1 or q_2 for the next level in the run dag. From this level, it has to find an accepting run on w in the same manner. Thinking of an automaton $\overline{\mathcal{A}}$ accepting the complement of the first one, it must reject the rest of the word in q_1 as well as in q_2. Thus, given a kind of "dual acceptance condition", it might be a good idea to define the transition function of $\overline{\mathcal{A}}$ (denoted by $\overline{\delta}$) by $\overline{\delta}(q_0, a) = \overline{\delta(q_0, a)} = q_1 \wedge q_2$. Let us fix this general scheme in the following definition:

Definition 4.2.15
The dual *of a formula $\varphi \in \mathcal{B}^+(X)$ denoted by $\overline{\varphi}$ is the formula where* ff *is replaced by* tt, tt *by* ff*, $\vee$ by $\wedge$ and $\wedge$ by $\vee$.*

Since the dual of tt is ff and vice versa, finite paths ending in tt will abolish the path of a run in the automaton having the dual transition function. A dual observation also holds for runs now ending in tt. But dualizing formulas is not enough. A run on w is accepting if all infinite paths visit final states infinitely often. However, non-final states can also be visited infinitely often. Hence, switching to the complement of the final states for the automaton $\overline{\mathcal{A}}$ thus might yield runs which are accepting on the same input for both automata $\mathcal{A}$ and $\overline{\mathcal{A}}$. The situation is easier if our automata are *weak*, a notion due to [MSS86]:

Definition 4.2.16
An alternating Büchi automaton $\mathcal{A} = (Q, \delta, q_0, F)$ *is called* weak *iff there exists a collection of* components $Q_1, \dots, Q_m$ *such that the following holds:*

1. *the collection of the* Q_i *is a partition of* Q, *i.e.,* $Q = \bigcup_{i \in \{1,\dots,m\}} Q_i$ *and for all* $i, j \in \{1, \dots, m\}$ *with* $i \neq j$, *it holds* $Q_i \cap Q_j = \emptyset$.
2. *for every* $i \in \{1, \dots, m\}$, *we have*
   - *(a) either* $Q_i \subseteq F$, *in which case* Q_i *is an* accepting set *or* accepting component, *or*
   - *(b)* $Q_i \cap F = \emptyset$, *in which case* Q_i *is a* rejecting *set or* rejecting component.
3. *There exists a partial order* $\leq$ *on the collection of the components such that for every* $q \in Q_i$ *and* $q' \in Q_j$ *for which* q' *occurs in* $\delta(q, a)$ *for some* $a \in \Sigma$, *we have* $Q_i \leq Q_j$. *Without loss of generality, we may assume that* $Q_i \leq Q_j$ *implies* $i \leq j$.

For the transition graph of an alternating Büchi automaton, weakness means that it can be partitioned into components which consist of either final or non-final states (together with nodes labeled by $\wedge$, $\vee$, ff, or tt), and which can be partially ordered by their edges. A component is lower than another iff there are edges from the first to the second. In other words, transitions from a state in Q_i lead to states in either the same Q_i or a higher one. Figure 4.7 on the following page shows (one) partition of the alternating Büchi automaton of Example 4.2.4 on page 46 into three components. Note that Q_1 and Q_3 are rejecting while Q_2 is accepting. Edges identifying the partial order of the components are printed in boldface. Thus, $Q_1 \leq Q_2 \leq Q_3$. We learn that the automaton is weak. Let us mention that the partition is not unique since tt, for example, could have estimated a single component.

It is now easy to see that every infinite path of a run of a weak alternating Büchi automaton ultimately gets "trapped" within some Q_i. The path then satisfies the acceptance condition if and only if Q_i is an accepting set. Indeed, a path of a run visits infinitely many states in F iff it gets trapped in an accepting set. We conclude that the complementation procedure motivated before works. Taking the complement of the final states of an automaton means that a path of a previously accepting run is it no longer since it visits only finitely many final states. We sum up:

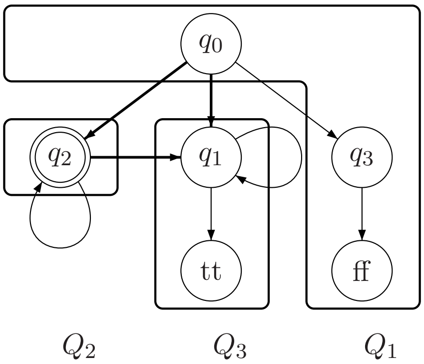


Figure 4.7: The transition graph of a weak alternating Büchi automaton and its partitions

Theorem 4.2.17 ([MS87])
Let $\mathcal{A} = (Q, \delta, q_0, F)$ *be a weak alternating Büchi automaton over* Σ. *Then for* $\overline{\mathcal{A}} = (Q, \overline{\delta}, q_0, Q \setminus F)$, *we have*

$$\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})}$$

where $\overline{\delta}(q, a) = \overline{\delta(q, a)}$ *for all* $q \in Q$ *and* $a \in \Sigma$.

A formal and simple proof of this result can be found in [LT00]. Note that the sizes of the original and the complemented automaton are identical.

The small handicap of the previous theorem seems to be its restriction to weak alternating automata. However, weak alternating Büchi automata and alternating Büchi automata coincide with respect to expressiveness. Even more, there is a simple procedure which transforms a given alternating Büchi automaton into a weak alternating Büchi automaton accepting the same language with a quadratic blow-up.

Theorem 4.2.18 ([KV97, KV01])
Let $\mathcal{A}$ *be an alternating Büchi automaton. There is a weak alternating Büchi automaton* $\mathcal{A}'$ *such that* $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ *and the number of states of* $\mathcal{A}'$ *is quadratic in the number of states of* $\mathcal{A}$.

Combining the two previous theorems we get:

Theorem 4.2.19
Let $\mathcal{A} = (Q, \delta, q_0, F)$ *be an alternating Büchi automaton. Then there is a (weak) alternating Büchi automaton* $\overline{\mathcal{A}}$ *such that*

$$\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})}$$

and the number of states in $\mathcal{A}'$ *is quadratic in the number of states of* $\mathcal{A}$.

4.2.4 Linear Alternating Büchi Automata

As we already mentioned, linear temporal logic formulas can be translated into alternating automata accepting precisely its models. However, the construction yields a restricted kind of alternating Büchi automata, so-called *linear* alternating Büchi automata, a notion due to Löding and Thomas [LT00].

On the other hand, [LT00] show that every linear alternating Büchi automaton can be translated into a linear temporal logic formula having the elements of the language of the automaton as models.

Linearity of alternating automata is defined by a restriction on the cycles of the transition graph of an alternating automaton.

Definition 4.2.20
An alternating Büchi automaton $\mathcal{A}$ is called linear *if the transition graph of $\mathcal{A}$ has only trivial cycles, i.e., for every path $q_1 \dots q_k$ with $k \geq 2$ and $q_1 = q_k$, we have that all q_i are labeled by q_1.*

The automaton shown in Figure 4.4 on page 48 is linear, because (cf. Figure 4.7 on the preceding page)

- there is neither a cycle containing q_0 nor q_3,
- there is no cycle from q_2 back to q_2 via q_1, and
- there is no cycle from q_1 back to q_1 via q_2.

Note however, that there is a cycle from q_2 back to q_2 but also a path from q_2 to q_1.

The linearity of the transition graph has an immediate consequence for the runs of an alternating Büchi automaton. Let us first define linear runs:

Definition 4.2.21
Let $\mathcal{A}$ be a linear alternating Büchi automaton. A run (V, E) of $\mathcal{A}$ on a word w is called linear *iff it holds: For all $n \geq 2$ and pairwise disjoint $v_1, \dots, v_n \in V$ with $(v_i, v_{i+1}) \in E$ (for all $i \in \{1, \dots, n-1\}$), $l(v_1) = l(v_n)$ implies $l(v_1) = l(v_2) = \dots = l(v_n)$.*

Figure 4.8 on the following page shows a linear run for the automaton represented in Figure 4.7 on the preceding page. It is now easy to see that all runs of linear alternating automata are linear:

Lemma 4.2.22 *Let $\mathcal{A}$ be a linear alternating Büchi automaton. Then every of its runs is linear.*

Note that the contrary does not hold:

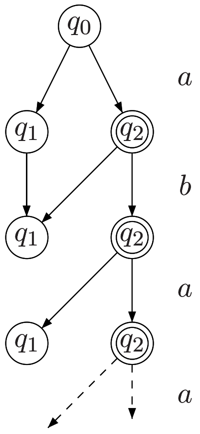


Figure 4.8: A linear run of an alternating Büchi automaton

Example 4.2.23 Consider the alphabet $\Sigma = \{a, b\}$ and the automaton shown in Figure 4.9 on the facing page. Since there is a non-trivial cycle $q_0 q' q_0$, the automaton is not linear. However, since $\delta(q, a) = \delta(q, b) = \mathrm{ff}$, there is no run at all. Trivially, all runs are linear.

Since the transition graph of a linear automaton has only trivial cycles, it is easy to see that it is also weak.

Lemma 4.2.24 *Every linear alternating Büchi automaton is weak.*

Proof
Every single state estimates a component of the automaton. A component is accepting if the state is a final state, otherwise it is rejecting. Since the transition graph has only trivial cycles, the partial order is given by the edge relation of the transition graph, erasing loops. □

Thus, every linear automaton can be complemented by dualizing the transition function and taking the complement of the final states as the new final states (cf. Theorem 4.2.17 on page 58).

4.2.5 Trace-consistent Alternating Büchi Automata

We learned in the previous chapter to identify trace consistent word languages and trace languages. It is therefore natural to separate automata which can be understood as devices accepting trace languages.

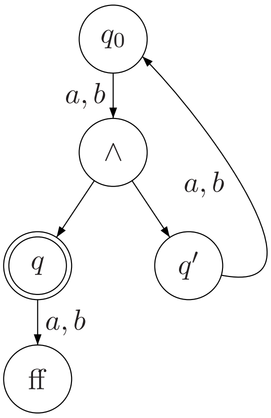


Figure 4.9: A non-linear alternating Büchi automaton

Definition 4.2.25
An alternating Büchi automaton $\mathcal{A}$ *is called* trace consistent *iff* $\mathcal{L}(\mathcal{A})$ *is trace-consistent.*

As alternating Büchi automata accept precisely the regular languages, we conclude that trace-consistent alternating Büchi automata accept precisely the regular trace languages when the latter are considered as word languages.

We will show that *linear* trace-consistent automata correspond to LTL formulas over Mazurkiewicz traces, i.e., that these automata accept precisely the languages definable by formulas of this logic.

Chapter 5

First-Order Logic

We recall the notions of first-order logic interpreted over words and Mazurkiewicz traces and give examples of its usage in the domain of specification. This allows us to formulate expressiveness results for the linear temporal logics to be studied in the next chapters in a precise way.

5.1 FO over Words

Definition 5.1.1 (Syntax of $\mathrm{FO_w}$)
The set of formulas of first-order logic over a countable set of variables $Var = \{x, y, \ldots\}$ *and an alphabet* Σ *ranges over a family of unary predicates* $(R_a)_{a\in\Sigma}$ *and a binary predicate* $\leq$*, is denoted by* $\mathrm{FO_w}(\Sigma)$*, and is given by the following grammar:*

$$\varphi ::= R_a(x) \mid x \leq y \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x\, \varphi \quad (a \in \Sigma)$$

We usually use identifiers like $\varphi, \psi, \eta, \ldots$ or $\varphi', \psi', \eta', \ldots$ for formulas. When it is clear from the context that we interpret over words or if the alphabet is fixed, we abbreviate $\mathrm{FO_w}(\Sigma)$ by $\mathrm{FO}(\Sigma)$, $\mathrm{FO_w}$, or just FO. Let us fix an alphabet Σ and a set of variables Var for the rest of this section.

A *valuation* is a mapping $V : Var \to \mathbb{N}$ yielding for every variable in Var a natural number. For a variable $x \in Var$ and a natural number i, we denote by $V[x/i]$ the valuation that coincides with V for every $y \neq x$ and yields i when applied to x.

Definition 5.1.2 (Semantics of $\mathrm{FO_w}$)
Given a word $w \in \Sigma^\omega$ *of the form* $w = a_1 a_2 \ldots$*, a valuation* $V : Var \to \mathbb{N}$*, and* $\varphi \in \mathrm{FO}(\Sigma)$*, the notion of* $w \models_V \varphi$ *is defined inductively via:*

- $w \models_V R_a(x)$ *iff* $a_{V(x)} = a$

- *$w \models_V x \leq y$ iff $V(x)$ is less or equal $V(y)$ with respect to the natural ordering of natural numbers*
- *$w \models_V \neg\varphi$ iff $w \not\models_V \varphi$*
- *$w \models_V \varphi \vee \psi$ iff $w \models_V \varphi$ or $w \models_V \psi$*
- *$w \models_V \exists x\varphi$ iff there exists an $i \in \mathbb{N}$ such that $w \models_{V[x/i]} \varphi$*

In the following, we assume that φ is a sentence, i.e., φ has no free variables.[1] For a sentence φ, $w \models_V \varphi$ is independent of V. Consequently, we write $w \models \varphi$ instead. The *language* of φ is denoted by $\mathcal{L}(\varphi)$ and is defined by

$$\mathcal{L}(\varphi) = \{w \mid w \models \varphi\}$$

We will say that $L \subseteq \Sigma^\omega$ is FO-*definable* iff there is a $\varphi \in \mathrm{FO}(\Sigma)$ such that $L = \mathcal{L}(\varphi)$.

5.2 FO over Mazurkiewicz Traces

Definition 5.2.1 (Syntax of FO_t)
The set of formulas of first-order logic over a countable set of variables Var $= \{x, y, \ldots\}$ and an independence alphabet (Σ, I) ranges over a family of unary predicates $(R_a)_{a\in\Sigma}$ and a binary predicate $\leq$, is denoted by $\mathrm{FO}_\mathrm{t}(\Sigma, I)$, and is given by the following grammar:

$$\varphi ::= R_a(x) \mid x \leq y \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x\varphi \quad (a \in \Sigma)$$

When it is clear from the context that we interpret over traces or if the alphabet is fixed, we abbreviate $\mathrm{FO}_\mathrm{t}(\Sigma, I)$ by $\mathrm{FO}(\Sigma, I)$, FO_t, or just FO.

Evidently, the syntax of FO_w and FO_t are equally defined. In particular, the syntax of FO_t does not explicitly involve the independence relation I. However, I is reflected in $\leq$, which will be interpreted as the partial order relation associated with a trace that does indeed respect I.

Again, we use identifiers like $\varphi, \psi, \eta, \ldots$ or $\varphi', \psi', \eta', \ldots$ for formulas of $\mathrm{FO}(\Sigma, I)$.

Given a trace $T = (E, \leq, \lambda)$, a *valuation* is a mapping $V : \mathit{Var} \to E$ yielding for every variable in *Var* an event of T. Strictly speaking, we should write V_T, which we avoid for the sake of brevity. For a variable $x \in \mathit{Var}$ and an event e, we denote by $V[x/e]$ the valuation that agrees with V for every $y \neq x$ and yields e when applied to x.

---

[1] The notion of free variables is defined as usual.

Definition 5.2.2 (Semantics of $\mathrm{FO_t}$)
Given a trace $T = (E, \leq, \lambda)$ and an associated valuation $V : \mathit{Var} \to E$, the relation $T \models_V \varphi$ will denote that T is a model of $\varphi \in \mathrm{FO_t}(\Sigma, I)$ under the valuation V, and is defined inductively via:

- *$T \models_V R_a(x)$ iff $\lambda(V(x)) = a$*
- *$T \models_V x \leq y$ iff $V(x)$ is less or equal $V(y)$ with respect to the ordering of the events of T*
- *$T \models_V \neg\varphi$ iff $T \not\models_V \varphi$*
- *$T \models_V \varphi \vee \psi$ iff $T \models_V \varphi$ or $T \models_V \psi$*
- *$T \models_V \exists x\, \varphi$ iff there exists an $e \in E$ such that $T \models_{V[x/e]} \varphi$*

In the following, we assume that φ is a sentence, i.e., φ has no free variables. The *language* of φ is denoted by $\mathcal{L}(\varphi)$ and is defined by

$$\mathcal{L}(\varphi) = \{T \mid T \models \varphi\}$$

We will say that $L \subseteq \mathbb{TR}(\Sigma, I)$ is FO-*definable* iff there is a $\varphi \in \mathrm{FO}(\Sigma, I)$ such that $L = \mathcal{L}(\varphi)$.

We will freely use the standard abbreviations such as $\forall x\, \varphi = \neg\exists x\, \neg\varphi$, $\mathrm{tt} = \forall x\; x \leq x$, $\mathrm{ff} = \neg\mathrm{tt}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, or $\varphi \to \psi = \neg\varphi \vee \psi$. Furthermore, we write $x < y$ for $x \leq y \wedge \neg(y \leq x)$ and $x \not\leq y$ for $\neg(x \leq y)$.

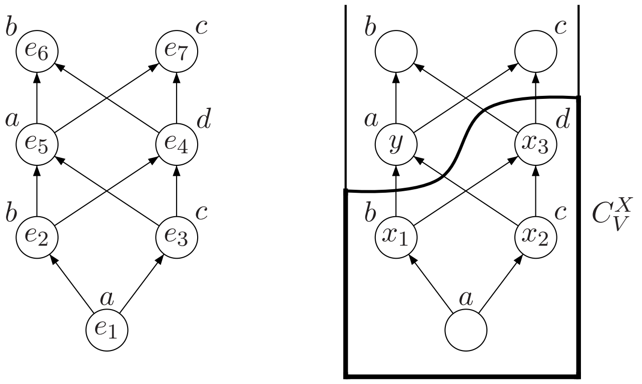
We fix a set of variables $\mathit{Var} = \{x, y, \ldots\}$, an independence alphabet (Σ, I), a trace $T = (E, \leq, \lambda)$, and a valuation $V : \mathit{Var} \to E$ for the rest of this section. We say that x *identifies* $e \in E$ iff $V(x) = e$.

FO is designed to formulate properties of events of a trace. For example, we can say that an event identified by x is labeled by an action independent of some set $Z \subseteq \Sigma$ by

$$\mathrm{Independent}(Z, x) = \neg\left(\bigvee_{aDZ} R_a(x)\right)$$

where D is the dependence relation obtained by complementing I in Σ^2.

As we will see in Chapters 7 and 8, the linear temporal logics to be studied are interpreted in configurations of traces. We have seen that FO can be used to formulate properties of events of a trace but does not provide primitives to reason about properties of configurations. Nevertheless, we can formulate properties of configurations: We can associate a set of events with a configuration. The number of maximal



(a) (b)

Figure 5.1: Identifying configurations and variables

events of each configuration is bounded by $|\Sigma|$, because there are at most $|\Sigma|$ independent events and maximal events are necessarily independent. Thus, a finite set of variables $X \subseteq \mathit{Var}$ can be used to *identify* for a given valuation $V : \mathit{Var} \to E$ the configuration

$$C_V^X = \{e \mid \text{there is an } x \in X \text{ such that } e \leq V(x)\}$$

For example, for the trace shown in Figure 5.1(a) and the valuation mapping $x_i \mapsto e_{i+1}$ for $i \in \{1, 2, 3\}$, the set $X = \{x_1, x_2, x_3\}$ identifies the configuration depicted by C_V^X in Figure 5.1(b). Note that $\{x_1, x_3\}$ identifies the same configuration.

When defining formulas specifying requirements of traces, we can keep track of a finite set X of variables used to identify a configuration of a trace.

Suppose $X = \{x_1, \ldots, x_k\}$ is used to identify the configuration C_V^X. Then, we can express that C_V^X has an a-successor configuration by

$$\exists y\ R_a(y) \wedge \left(\bigwedge_{i=1,\ldots,k} y \not\leq x_i \right) \wedge \forall z \left(z < y \to \left(\bigvee_{i=1,\ldots,k} z \leq x_i \right) \right)$$

In other words, there is an event identified by y, labeled by a, and which is not contained in the current configuration, but can be added to the current configuration, still obtaining a configuration. Note that the set of variables used to represent the successor configuration is $\{x_1, \ldots, x_k, y\}$. The situation is depicted in Figure 5.1(b).

Let us give a collection of examples describing further properties of configurations. We will come back to these formulas in Chapter 7 when explaining how to define properties expressed in linear temporal logic for traces in terms of FO logic.

Suppose we employ finite $X, Y \subseteq \mathit{Var}$ for representing configurations C_V^X and C_V^Y for a given valuation. Then

$$\mathrm{Below}(X, Y) = \bigwedge_{x \in X} \left(\bigvee_{y \in Y} x \leq y \right)$$

holds iff $C_V^X \subseteq C_V^Y$. Furthermore,

$$\mathrm{SBelow}(X, Y) = \mathrm{Below}(X, Y) \wedge \neg \mathrm{Below}(Y, X)$$

holds iff $C_V^X \subsetneqq C_V^Y$.

Suppose we want to express that for finite $X, Y \subseteq \mathit{Var}$, the configuration C_V^X is a subconfiguration of C_V^Y and furthermore, all events in $C_V^Y - C_V^X$ are independent of some $Z \subseteq \Sigma$.

A first guess might be

$$\begin{aligned}\mathrm{BelowIndep}(X, Y, Z) = \mathrm{Below}(X, Y) \wedge \\ \forall z \left(\bigvee_{x \in X} \bigvee_{y \in Y} (x < z \wedge x \leq y) \rightarrow \mathrm{Independent}(Z, z) \right)\end{aligned}$$

However, this fails for two reasons. First, there might be an event in C_V^Y which is incomparable with all the events identified by X. This also has to be independent of the actions given by Z. Second, there might be $x, x' \in X$ with $V(x) < V(x')$. Thus, the previous formula would require that $\lambda(V(x'))$ is also independent of Z, which we did not postulate.

It is easy to formulate the desired property by considering when an element in C_V^Y is not in $C_V^Y - C_V^X$. This is whenever there is an event in C_V^Y that is smaller than some event in $V(x)$ for $x \in X$. Thus, we define—thanks to negation—

$$\mathrm{zInYminusX}(X, Y, z) = \left(\bigvee_{y \in Y} z \leq y \right) \wedge \neg \left(\bigvee_{x \in X} z \leq x \right)$$

and

$$\begin{aligned}\mathrm{BelowIndep}(X, Y, Z) = \mathrm{Below}(X, Y) \wedge \\ \forall z \left(\mathrm{zInYminusX}(X, Y, z) \rightarrow \mathrm{Independent}(Z, z) \right)\end{aligned}$$

The formulas reduce as expected when one of the sets is empty.

Chapter 6

LTL over words

In this chapter, we recall the syntax and semantics of linear temporal logic interpreted over words, abbreviated by $\mathrm{LTL_w}$ or LTL if the context makes clear that we interpret the logic over words. In a landmark paper [Pnu77], Pnueli explained that LTL can be used as a tool for specifying and verifying correctness properties of so-called *reactive programs*. LTL gained a lot of interest and is today widely used as a specification language especially for concurrent systems. Confer [MP92, Var01, HHI$^+$01] for overviews.

In this thesis, we consider plain modal logics with labeled modalities in place of propositional temporal logics. Thus, we do not support the usage of propositional variables in our logics but allow labeled next-state modalities instead. We have experienced that this kind of logics fits better to implementations by means of process algebra formalisms like CCS [Mil83, Mil89] or ACP [BK84, Fok00]. It should be mentioned though, that our approach can be extended in a straightforward way to support propositions as well. Basically, it would only complicate the formal presentation rather than give new conceptual insights.

In the second section of this chapter, we rephrase a decision procedure for satisfiability of $\mathrm{LTL_w}$ formulas based on alternating Büchi automata due to Vardi [Var96], which is slightly adapted to support labeled next-state modalities. We will extend this logic and the decision procedure in the next chapter towards the setting of traces.

The subsequent section shows that languages definable by $\mathrm{LTL_w}$ formulas correspond to languages definable by *linear* alternating Büchi automata. This result was independently shown by Rohde [Roh97] and Thomas and Löding [LT00]. In the next chapter, we will learn in which way this result has to be read in the setting of LTL over Mazurkiewicz traces.

We conclude this chapter with a discussion of languages definable by $\mathrm{LTL_w}$ formulas and trace-consistent languages, *model checking* and satisfiability, and the relation of first order logic over words and LTL.

6.1 Syntax and Semantics

Linear temporal logic formulas are designed for describing sequences of actions. They are parameterized by an alphabet Σ and are defined inductively as follows:

Definition 6.1.1 (Syntax of LTL_w)
The set of linear temporal logic formulas over an alphabet Σ is denoted by $\mathrm{LTL}_\mathrm{w}(\Sigma)$ and is given by the following grammar:

$$\varphi ::= \mathrm{tt} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle a\rangle\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \quad (a \in \Sigma)$$

We usually use identifiers like $\varphi, \psi, \eta, \ldots$ or $\varphi', \psi', \eta', \ldots$ for formulas. When it is clear from the context that we interpret over words or if the alphabet is fixed, we abbreviate $\mathrm{LTL}_\mathrm{w}(\Sigma)$ by $\mathrm{LTL}(\Sigma)$, LTL_w, or just LTL.

To simplify the forthcoming definition, let $w^{(i)}$ denote the *suffix* $a_i a_{i+1} \ldots$ of a word $w = a_1 a_2 \ldots \in \Sigma^\omega$. Note that $w^{(i)}$ is in Σ^ω for all $i \in \mathbb{N} \setminus \{0\}$. We are now ready to define the semantics of LTL formulas over words.

Definition 6.1.2 (Semantics of LTL_w)
Given a word $w \in \Sigma^\omega$ of the form $w = a_1 a_2 \ldots$ and $\varphi \in \mathrm{LTL}_\mathrm{w}(\Sigma)$, the notion of $w \models \varphi$ is defined inductively via:

- $w \models \mathrm{tt}$
- $w \models \neg\varphi$ *iff* $w \not\models \varphi$
- $w \models \varphi \vee \psi$ *iff* $w \models \varphi$ *or* $w \models \psi$
- $w \models \langle a\rangle\varphi$ *iff* $a_1 = a$ *and* $w^{(2)} \models \varphi$
- $w \models \varphi \mathcal{U} \psi$ *iff there exists* $j \in \mathbb{N}$ *with* $j \geq 1$ *such that* $w^{(j)} \models \psi$ *and for all* $1 \leq i < j$*, we have* $w^{(i)} \models \varphi$

Iff $w \models \varphi$, we say w *models* φ, w is a *model* for φ, or w *satisfies* φ. Furthermore, we call φ *satisfiable* iff there is a $w \in \Sigma^\omega$ such that $w \models \varphi$. All models of a formula $\varphi \in \mathrm{LTL}_\mathrm{w}(\Sigma)$ constitute a subset of Σ^ω, thus a language. It is denoted by $\mathcal{L}(\varphi)$ and is called the language *defined* by φ.

To give an example, $\langle a\rangle\langle b\rangle\mathrm{tt}$ is satisfied by every word with prefix ab. $\langle a\rangle\mathrm{tt} \vee \langle b\rangle\mathrm{tt}$ is satisfied by every word which starts with either a or b. On the contrary, $\langle a\rangle\mathrm{tt} \wedge \langle b\rangle\mathrm{tt}$ is not satisfiable since there is no word having a as well as b at the first position. To arouse your curiosity, note that the latter formula is satisfiable it the setting of LTL over Mazurkiewicz traces if the right independence alphabet is given (see Section 7.1 on page 84 for details).

We will freely use the standard abbreviations such as $\mathrm{ff} = \neg\mathrm{tt}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, or $\varphi \rightarrow \psi = \neg\varphi \vee \psi$. Furthermore, we usually write $\Diamond\varphi$ for $\mathrm{tt}\,\mathcal{U}\varphi$ and $\Box\varphi$ for $\neg\Diamond\neg\varphi$. Thus, a formula $\Diamond\varphi$ is satisfied by a word w iff there is a suffix $w^{(i)}$ of w such that $w^{(i)} \models \varphi$. Intuitively, $w \models \Diamond\varphi$ iff *eventually* φ holds in w. Dually, *globally* φ is expressed by $\Box\varphi$, i.e., for all $i \in \mathbb{N}$, we have $w^{(i)} \models \varphi$.

To give a flavor of the utilization of LTL formulas within the application domain of specification of concurrent systems, let us recall that a word's sequence of actions can be understood as the behavior of a system. Then we can identify certain types of formulas:

1. *Safety properties* require that “nothing bad” ever happens. If a “bad” property is expressed by φ then $\Box\neg\varphi$ is a safety property.
2. *Liveness properties*, on the other hand, express that something “good” will happen. It can be formulated by $\Diamond\varphi$ if φ identifies the “good” property. Often, one is interested in “good” things to happen infinitely often during a run of a system. For example, a server should always—sooner or later—answer a request. A formula of the form $\Box(\varphi_{request} \rightarrow \Diamond\varphi_{answer})$ describes exactly this requirement if $\varphi_{request}$ and φ_{answer} specify a request and an answer, respectively.

Consult [MP92] for a careful and accurate introduction to the field of specification (and verification) using temporal logic.

6.2 Deciding Satisfiability of $\mathrm{LTL_w}$

In this section, we recall the key issues for defining a decision procedure for checking satisfiability of $\mathrm{LTL_w}$ formulas. It is based on alternating Büchi automata. Although—as will turn out in the next chapter—the following procedure will come up as a special case of the satisfiability procedure for LTL over Mazurkiewicz traces, we provide a separate exposition to simplify our overall presentation.

To be able to estimate the complexity of the forthcoming decision procedure, we have to define the length of a formula, which is going to be the reference value.

Definition 6.2.1
The length $|\varphi|$ *of a formula* $\varphi \in \mathrm{LTL_w}$ *is inductively defined by* $|\mathrm{tt}| = 1$*,* $|\neg\varphi| = |\langle a\rangle\varphi| = 1 + |\varphi|$*, and* $|\varphi \vee \psi| = |\varphi\,\mathcal{U}\psi| = 1 + |\varphi| + |\psi|$*.*

The decision procedure works on the subformulas (and negated subformulas) of a given formula. Subformulas are defined as usual:

Definition 6.2.2
The set $Sub(\varphi)$ of subformulas *for a given formula $\varphi \in \mathrm{LTL_w}$ is inductively defined by*

$$\begin{array}{lcl} Sub(\mathrm{tt}) & = & \{\mathrm{tt}\} \\ Sub(\neg\varphi) & = & \{\neg\varphi\} \cup Sub(\varphi) \\ Sub(\varphi \vee \psi) & = & \{\varphi \vee \psi\} \cup Sub(\varphi) \cup Sub(\psi) \\ Sub(\langle a\rangle\varphi) & = & \{\langle a\rangle\varphi\} \cup Sub(\varphi) \\ Sub(\varphi\mathcal{U}\psi) & = & \{\varphi\mathcal{U}\psi\} \cup Sub(\varphi) \cup Sub(\psi) \end{array}$$

We call ψ a *subformula* of φ iff $\psi \in Sub(\varphi)$. Furthermore, it is called a *strict* subformula iff $\psi \in Sub(\varphi)$ and $\psi \neq \varphi$.

Please observe that the size of $Sub(\varphi)$ is linear in the length of the formula φ for every $\varphi \in \mathrm{LTL}$.

As seen in the section for alternating Büchi automata (Section 4.2 on page 44), dualization is a powerful concept simplifying complementation. Let us define the dual of a positive Boolean combination of LTL formulas, which is employed in the satisfiability construction to cope with negation of formulas.

Definition 6.2.3
The dual *of a positive Boolean combination of* $\mathrm{LTL_w}$ *formulas is given inductively as follows:*

- $\overline{\mathrm{tt}} = \mathrm{ff}$, $\overline{\mathrm{ff}} = \mathrm{tt}$.
- $\overline{\neg\varphi} = \varphi$.
- $\overline{\varphi \vee \psi} = \overline{\varphi} \wedge \overline{\psi}$, $\overline{\varphi \wedge \psi} = \overline{\varphi} \vee \overline{\psi}$.
- $\overline{\langle a\rangle\varphi} = \neg\langle a\rangle\varphi$.
- $\overline{\varphi\mathcal{U}\psi} = \neg(\varphi\mathcal{U}\psi)$.

Consider $\varphi \in \mathrm{LTL}$. For deciding satisfiability of φ, we pursue the following plan. We construct an automaton accepting precisely the models of φ. This can be checked for emptiness giving the desired answer. The states of the automaton turn out to be subformulas of φ and negations thereof. The automaton works by choosing an appropriate subset of the subformulas reading an input action. The idea is understood easily, considering an automaton in state $\langle a\rangle\varphi$. Reading a means that the word indeed begins with a. Thus, it remains to check that the suffix of the word satisfies φ. Hence, the automaton proceeds in state φ after reading a. If, however, the first input symbol is different from a, the formula is not satisfied by the word. Thus, the automaton will proceed in state ff. Note that, while ff is no LTL formula, it is a state of our automaton.

What to do in a state $\varphi \vee \psi$? The automaton has to check whether φ or ψ holds reading an input action. Let φ' and ψ' be the Boolean combination of states in which to proceed for φ or respectively ψ reading the input action. We let the automaton choose φ' or ψ' by defining the successor states of $\varphi \vee \psi$ by $\varphi' \vee \psi'$.

Negation is handled straightforward. For $\neg\varphi$, consider the Boolean combination of successor states of φ reading a and dualize the result. For example, given $\neg(\varphi \vee \psi)$, the automaton proceeds in $\overline{\varphi' \vee \psi'} = \overline{\varphi'} \wedge \overline{\psi'}$ for φ' and ψ' obtained in the way described before. Note that $\wedge$ is a conjunction of states of the automaton and no abbreviation for a negated disjunction here.

It is easy to see that an *until-formula* of the form $\varphi\mathcal{U}\psi$ is logically *equivalent* to $\psi \vee \big(\varphi \wedge \bigvee_{a\in\Sigma}\langle a\rangle(\varphi\mathcal{U}\psi)\big)$, i.e., both formulas have the same class of models. Prosaically, $\varphi\mathcal{U}\psi$ is satisfied iff ψ is satisfied at the current position of the word or the current position satisfies φ and the next position $\varphi\mathcal{U}\psi$. Thus, reading an action a in state $\varphi\mathcal{U}\psi$, it is right to proceed either in the state(s) obtained by reading a in ψ or in states $\varphi\mathcal{U}\psi$ and the one(s) φ yields reading a.

Let us sum up the transition function δ of our alternating Büchi automaton:

Definition 6.2.4
For each formula $\varphi \in \mathrm{LTL_w}(\Sigma)$*, we define a transition function* $\delta_\varphi : \mathit{Sub}(\varphi) \cup \neg\mathit{Sub}(\varphi) \to \mathcal{B}^+(\mathit{Sub}(\varphi) \cup \neg\mathit{Sub}(\varphi))$ *inductively via:*

$$\begin{array}{lcl}
\delta_\varphi(\mathrm{tt}, a) & = & \mathrm{tt} \\
\delta_\varphi(\psi \vee \eta, a) & = & \delta_\varphi(\psi, a) \vee \delta_\varphi(\eta, a) \\
\delta_\varphi(\neg\psi, a) & = & \overline{\delta_\varphi(\psi, a)} \\
\delta_\varphi(\langle b\rangle\psi, a) & = & \begin{cases} \psi & \text{if } a = b \\ \mathrm{ff} & \text{else} \end{cases} \\
\delta_\varphi(\psi\mathcal{U}\eta, a) & = & \delta_\varphi(\eta, a) \vee (\delta_\varphi(\psi, a) \wedge \psi\mathcal{U}\eta)
\end{array}$$

We can now finally bring out the definition of the alternating Büchi automaton $\mathcal{A}_\varphi$ accepting precisely the models of a formula $\varphi \in \mathrm{LTL_w}(\Sigma)$.

Definition 6.2.5
Given a formula $\varphi \in \mathrm{LTL_w}(\Sigma)$*, we let the* alternating Büchi automaton of φ*, denoted by* $\mathcal{A}_\varphi$*, be the tuple* (Q, δ, q_0, F) *where*

- $Q = \mathit{Sub}(\varphi) \cup \neg\mathit{Sub}(\varphi)$ *is the set of states, where* $\neg\mathit{Sub}(\varphi)$ *abbreviates* $\{\neg\psi \mid \psi \in \mathit{Sub}(\varphi)\}$.
- $\delta = \delta_\varphi$ *is the transition function as in Definition 6.2.4.*
- $q_0 = \varphi$ *is the initial state.*
- $F = \{\neg\psi \mid \neg\psi \in \mathit{Sub}(\varphi)\}$ *is the set of accepting states.*

Let us give an intuitive but non-exhaustive argument for defining the set of final states in the way we did: Our whole approach can be understood as a kind of tableau construction. The formula to check is transformed into a Boolean combination of new formulas to check. In this way, a goal is reduced to several subgoals. Of course, the generation of subgoals has to terminate at one point to get an answer. A "positive formula", that is one in which every subformula starting without $\neg$ is preceded by an even number of $\neg$, counting 0 as an even number, must be "proved" indeed: State tt of the automaton has to be reached. Since any positive formula except one containing a subformula of the form $\varphi\,\mathcal{U}\,\psi$ is turned into strict subformulas, every path of a potential run of the automaton reaches either tt or ff. The latter signalizes non-acceptance, of course. For $\varphi\,\mathcal{U}\,\psi$, the automaton might follow an infinite path trying to show $\varphi\,\mathcal{U}\,\psi$. Consequently, we do not accept such a path since ψ is not shown to be satisfied by the input word considered. The situation is dual for a negative, i.e., not positive formula. A path labeled infinitely often with $\neg(\varphi\,\mathcal{U}\,\psi)$, for example, should be accepting since the automaton failed to "prove" ψ or φ. We get:

Theorem 6.2.6
Let φ be a formula of $\mathrm{LTL}_{\mathrm{w}}(\Sigma)$ and let its alternating Büchi automaton be given as $\mathcal{A}_\varphi$. Then

$$\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$$

We do not provide a proof of the previous theorem since we formulate an extended theorem which is accompanied with a proof for the case of traces. Since words are a special case of traces and our LTL version considered for Mazurkiewicz traces coincides for words with $\mathrm{LTL}_{\mathrm{w}}$, as we will see in the next chapter, we get a proof of Theorem 6.2.6 for free.

Let us give an example for the construction of an alternating Büchi automaton, given an $\mathrm{LTL}_{\mathrm{w}}$ formula.

Example 6.2.7 Let $\Sigma = \{a, b\}$. Suppose we want to specify that a sequence of b-actions is followed by an a-action. This can be formulated in $\mathrm{LTL}_{\mathrm{w}}$ by the formula $\varphi_1 = \langle b\rangle\mathrm{tt}\,\mathcal{U}\,\langle a\rangle\mathrm{tt}$. According to Definition 6.2.4 on the preceding page, we therefore get:

$$\begin{aligned}
\delta(\varphi_1, a) &= \delta(\langle a\rangle\mathrm{tt}, a) \vee (\delta(\langle b\rangle\mathrm{tt}, a) \wedge \varphi_1) \\
&= \mathrm{tt} \vee (\mathrm{ff} \wedge \varphi_1) \\
\delta(\varphi_1, b) &= \delta(\langle a\rangle\mathrm{tt}, b) \vee (\delta(\langle b\rangle\mathrm{tt}, b) \wedge \varphi_1) \\
&= \mathrm{ff} \vee (\mathrm{tt} \wedge \varphi_1)
\end{aligned}$$

The result can be simplified to $\delta(\varphi_1, a) = \mathrm{tt}$ and $\delta(\varphi_1, b) = \varphi_1$. Let φ_2 denote that, globally, we can do an a-action or a b-action and then φ_1 is satisfied. More specifically, $\varphi_2 = \Box\,(\langle a\rangle\varphi_1 \vee \langle b\rangle\varphi_1)$. Recall that $\Box\varphi$ is an abbreviation for $\neg\,(\mathrm{tt}\,\mathcal{U}\,\neg\varphi)$. We obtain

$$\begin{array}{rcl}
\delta(\varphi_2, a) & = & \delta(\Box\,(\langle a\rangle\varphi_1 \vee \langle b\rangle\varphi_1)\,, a) \\
& = & \delta(\neg\,(\mathrm{tt}\,\mathcal{U}\neg(\langle a\rangle\varphi_1 \vee \langle b\rangle\varphi_1))\,, a) \\
& = & \overline{\overline{\delta(\langle a\rangle\varphi_1, a) \vee \delta(\langle b\rangle\varphi_1, a)} \vee (\delta(\mathrm{tt}, a) \wedge (\mathrm{tt}\,\mathcal{U}\neg(\langle a\rangle\varphi_1 \vee \langle b\rangle\varphi_1)))} \\
& = & \overline{\overline{\varphi_1 \vee \mathrm{ff}} \vee (\mathrm{tt} \wedge (\mathrm{tt}\,\mathcal{U}\neg(\langle a\rangle\varphi_1 \vee \langle b\rangle\varphi_1)))} \\
& = & (\varphi_1 \vee \mathrm{ff}) \wedge (\mathrm{ff} \vee \varphi_2)
\end{array}$$

Thus, $\delta(\varphi_2, a) = \delta(\varphi_2, b)$ is equivalent to $\varphi_1 \wedge \varphi_2$. $\varphi_3 = \neg\,(\langle a\rangle\mathrm{tt} \vee \langle b\rangle\mathrm{tt})$ is satisfied iff neither a nor b is a possible action to execute. The corresponding transitions $\delta(\varphi_3, a)$ and $\delta(\varphi_3, b)$ are equivalent to ff.

Now, consider the formula $\varphi = \langle a\rangle(\varphi_1 \wedge \varphi_2) \vee \langle b\rangle((\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3))$. A simple calculation shows that altogether we get an automaton accepting the same language as the one of Example 4.2.4 on page 46.

It is obvious that the size of $\mathcal{A}_\varphi$ for a given formula $\varphi \in \mathrm{LTL_w}$ is linear in the size of φ. Thus, by Corollary 4.2.11 on page 53, we conclude:

Corollary 6.2.8 *Let φ be a formula of $\mathrm{LTL_w}(\Sigma)$. Deciding whether φ is satisfiable can be done in exponential time and polynomial space with respect to the size of φ.*

It was shown by Sistla and Clarke [SC82, SC85] that these bounds are optimal:

Theorem 6.2.9
Satisfiability of $\mathrm{LTL_w}$ formulas is PSPACE*-complete.*

The previous result was shown by a reduction of Turing machine. The authors also showed that satisfiability of $\mathrm{LTL_w^-}$ formulas is PSPACE-complete where $\mathrm{LTL_w^-}$ is the logic in which no *until*-operator is allowed but a $\Diamond$-operator.

6.3 Linear Alternating Automata and LTL over Words

The aim of this section is to characterize $\mathrm{LTL_w}$ formulas by means of alternating Büchi automata. We have already seen in the previous section that for every formula, φ there is an alternating Büchi automaton $\mathcal{A}_\varphi$ accepting precisely φ's models. A rather simple observation is that $\mathcal{A}_\varphi$ is *linear*.

Corollary 6.3.1 *Let φ be a formula of $\mathrm{LTL_w}(\Sigma)$. Then there is a linear alternating Büchi automaton $\mathcal{A}$ such that*

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$$

Proof
Consider the transition function in Definition 6.2.4 on page 73 of $\mathcal{A}_\varphi$. Check that $\delta(\psi, a)$ yields a positive Boolean combination of strict subformulas (and their negations) unless ψ is of the form $\psi_1 \,\mathcal{U}\, \psi_2$ or $\neg(\psi_1 \,\mathcal{U}\, \psi_2)$ in which case ψ occurs as well. □

Let us now see that for every linear alternating Büchi automaton there is an LTL_w formula such that the language of the automaton coincides with the language defined by the formula. We therefore can understand linear alternating Büchi automata as a different representation for LTL_w formulas. Due to Kamp's famous result [Kam68], we know that LTL formulas express exactly the first-order properties of words. We therefore can understand linear alternating automata also as a representation for those. We will discuss similar issues for the setting of traces in Chapter 7.3 on page 108.

Note that the results of this section are due to Rohde [Roh97] and Thomas and Löding [LT00]. We follow the latter construction, though with an adaption towards our slightly different version of LTL_w and with a different correctness proof.

Theorem 6.3.2
Let $\mathcal{A} = (Q, \delta, q_0, F)$ be a linear alternating Büchi automaton over the alphabet Σ. Then there is a formula $\varphi_\mathcal{A} \in \mathrm{LTL}_\mathrm{w}(\Sigma)$ such that

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi_\mathcal{A})$$

Proof
Our proof proceeds in the following way: We provide a construction of $\varphi_\mathcal{A}$ and show that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{\varphi_\mathcal{A}})$, i.e., the language of the automaton derived from the LTL formula $\varphi_\mathcal{A}$ coincides with the language of $\mathcal{A}$. Because of Theorem 6.2.6 on page 74, we know that $\mathcal{L}(\varphi_\mathcal{A}) = \mathcal{L}(\mathcal{A}_{\varphi_\mathcal{A}})$. Hence, $\mathcal{L}(\varphi_\mathcal{A}) = \mathcal{L}(\mathcal{A})$.

Before we give the construction, observe that, using distributive laws, every positive Boolean formula $\varphi \in \mathcal{B}^+(Q)$ containing q can easily be transformed into an equivalent one of the form $(q \wedge \varphi_q) \vee \varphi'_q$ as well as $(q \vee \varphi_q) \wedge \varphi'_q$ where φ_q and φ'_q do not contain q. Thus, without loss of generality, $\delta(q, a) = (q \wedge \varphi_{q,a}) \vee \varphi'_{q,a}$, $\delta(q, a) = (q \vee \varphi_{q,a}) \wedge \varphi'_{q,a}$, or $\delta(q, a) = \varphi'_{q,a}$ for all $q \in Q$, $a \in \Sigma$, and appropriate $\varphi_{q,a}, \varphi'_{q,a} \in \mathcal{B}^+(Q \setminus \{q\})$.

Since $\mathcal{A}$ is linear, we further conclude that q is not reachable from states in $\varphi_{q,a}$ and $\varphi'_{q,a}$ (with respect to the transition graph of the automaton).

Let us give the construction of $\varphi_\mathcal{A}$ for a given linear automaton $\mathcal{A}$. We start with a function $ltl_\mathcal{A} : \mathcal{B}^+(Q) \to \mathrm{LTL}_\mathrm{w}$ mapping Boolean combinations of states of $\mathcal{A}$ to LTL_w formulas. It will turn out that $\varphi_\mathcal{A} := ltl_\mathcal{A}(q_0)$ will have the desired properties.

We define $\mathit{ltl}_{\mathcal{A}} : \mathcal{B}^+(Q) \to \mathrm{LTL_w}$ by

$$\begin{array}{lcll}
\mathit{ltl}_{\mathcal{A}}(\mathrm{tt}) & = & \mathrm{tt} & \\
\mathit{ltl}_{\mathcal{A}}(\mathrm{ff}) & = & \neg\mathrm{tt} & \\
\mathit{ltl}_{\mathcal{A}}(\varphi \vee \psi) & = & \mathit{ltl}_{\mathcal{A}}(\varphi) \vee \mathit{ltl}_{\mathcal{A}}(\psi) & \\
\mathit{ltl}_{\mathcal{A}}(\varphi \wedge \psi) & = & \neg(\neg\mathit{ltl}_{\mathcal{A}}(\varphi) \vee \neg\mathit{ltl}_{\mathcal{A}}(\psi)) & \\
\mathit{ltl}_{\mathcal{A}}(q) & = & \bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\delta(q,a)) & \text{if } q \notin \bigvee_{a\in\Sigma}\delta(q,a)
\end{array}$$

where $q \in \varphi$ is a shorthand for $\varphi \in \mathcal{B}^+(Q \setminus \{q\})$. If q occurs on the right side of $\delta(q,a)$ for one $a \in \Sigma$, we distinguish whether q is a final state or not: If $q \notin F$ then

$$\mathit{ltl}_{\mathcal{A}}(q) = \left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi_{q,a})\right) \,\mathcal{U}\, \left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi'_{q,a})\right)$$

for $\delta(q,a) = (q \wedge \varphi_{q,a}) \vee \varphi'_{q,a}$ or $\delta(q,a) = \varphi'_{q,a}$. If $q \in F$ then

$$\mathit{ltl}_{\mathcal{A}}(q) = \neg\left(\left(\bigvee_{a\in\Sigma}\langle a\rangle \overline{\mathit{ltl}_{\mathcal{A}}(\varphi_{q,a})}\right) \,\mathcal{U}\, \left(\bigvee_{a\in\Sigma}\langle a\rangle \overline{\mathit{ltl}_{\mathcal{A}}(\varphi'_{q,a})}\right)\right)$$

for $\delta(q,a) = (q \vee \varphi_{q,a}) \wedge \varphi'_{q,a}$ or $\delta(q,a) = \varphi'_{q,a}$ and the dual $\overline{\varphi}$ of an LTL formula is defined in the expected manner. Note that we can assume that $\mathit{ltl}_{\mathcal{A}}(\varphi_{q,a})$ and $\mathit{ltl}_{\mathcal{A}}(\varphi'_{q,a})$ are given by induction in the two previous definitions.

It is now a routine matter to show by induction that for every q of $\mathcal{A}$ and $a \in \Sigma$, we have $\mathcal{L}(\mathcal{A}(\delta(q,a))) = \mathcal{L}(\mathcal{A}_{\varphi_{\mathcal{A}}}(\delta(\mathit{ltl}_{\mathcal{A}}(q),a)))$. This can easily be seen by obtaining an equivalence of both states considering the latter as a Boolean combination of states. For example, for

$$q = \left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi_{q,a})\right) \,\mathcal{U}\, \left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi'_{q,a})\right)$$

we obtain:

$$\begin{array}{cl}
 & \delta\left(\left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi_{q,a})\right) \mathcal{U} \left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi'_{q,a})\right), a\right) \\
= & \delta\left(\left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi'_{q,a})\right), a\right) \vee \left(\delta\left(\left(\bigvee_{a\in\Sigma}\langle a\rangle \mathit{ltl}_{\mathcal{A}}(\varphi_{q,a})\right), a\right) \wedge q\right) \\
= & \left(\varphi'_{q,a} \vee \bigvee_{b\in\Sigma\setminus\{a\}} \mathrm{ff}\right) \vee \left(\left(\varphi_{q,a} \vee \bigvee_{b\in\Sigma\setminus\{a\}} \mathrm{ff}\right) \wedge q\right) \\
\equiv & \varphi'_{q,a} \vee (\varphi_{q,a} \wedge q)
\end{array}$$

We leave a precise formulation as an exercise for the reader. □

Example 6.3.3 Let us derive a formula $\varphi_{\mathcal{A}}$ defining the language of the automaton $\mathcal{A}$ over the alphabet $\Sigma = \{a, b\}$ given in Example 4.2.4 on page 46. We transform right hand sides of the transition function δ into

$$\begin{array}{lcllcl}
\delta(q_0,a) & = & q_1 \wedge q_2 & \delta(q_2,a) & = & (q_2 \vee \mathrm{ff}) \wedge q_1 \\
\delta(q_0,b) & = & (q_1 \wedge q_2) \vee (q_1 \wedge q_3) & \delta(q_2,b) & = & (q_2 \vee \mathrm{ff}) \wedge q_1 \\
\delta(q_1,a) & = & \mathrm{tt} & \delta(q_3,a) & = & \mathrm{ff} \\
\delta(q_1,b) & = & (q_1 \wedge \mathrm{tt}) \vee \mathrm{ff} & \delta(q_3,b) & = & \mathrm{ff}
\end{array}$$

Using the notation and the construction as in the proof of Theorem 6.3.2 on page 76, we get, for example,

$$\begin{aligned} ltl_{\mathcal{A}}(q_3) &= \bigvee_{a\in\Sigma} \langle a\rangle\, ltl_{\mathcal{A}}(\mathrm{ff}) \\ &= \langle a\rangle(\neg \mathrm{tt}) \vee \langle b\rangle(\neg \mathrm{tt}) \end{aligned}$$

Let φ_3 denote the last formula.

For q_1, we get

$$\begin{aligned} ltl_{\mathcal{A}}(q_1) &= \left(\bigvee_{a\in\Sigma} \langle a\rangle\, ltl_{\mathcal{A}}(\varphi_{q_1,a})\right) \mathcal{U} \left(\bigvee_{a\in\Sigma} \langle a\rangle\, ltl_{\mathcal{A}}(\varphi'_{q_1,a})\right) \\ &= \langle b\rangle \mathrm{tt}\, \mathcal{U}\, (\langle a\rangle \mathrm{tt} \vee \langle b\rangle(\neg \mathrm{tt})) \end{aligned}$$

Note that $ltl_{\mathcal{A}}(q_1)$ is equivalent to $\langle b\rangle \mathrm{tt}\,\mathcal{U}\,\langle a\rangle \mathrm{tt}$ which is abbreviated by φ_1 in the following.

Let us check $ltl_{\mathcal{A}}(q_2)$:

$$\begin{aligned} ltl_{\mathcal{A}}(q_2) &= \neg\left(\left(\bigvee_{a\in\Sigma} \langle a\rangle\overline{ltl_{\mathcal{A}}(\varphi_{q_2,a})}\right) \mathcal{U} \left(\bigvee_{a\in\Sigma} \langle a\rangle\overline{ltl_{\mathcal{A}}(\varphi'_{q_2,a})}\right)\right) \\ &= \neg\left(\left(\bigvee_{a\in\Sigma} \langle a\rangle\overline{ltl_{\mathcal{A}}(\mathrm{ff})}\right) \mathcal{U} \left(\bigvee_{a\in\Sigma} \langle a\rangle\overline{ltl_{\mathcal{A}}(q_1)}\right)\right) \\ &\equiv \neg\left((\langle a\rangle \mathrm{tt} \vee \langle b\rangle \mathrm{tt})\, \mathcal{U} \left(\bigvee_{a\in\Sigma} \langle a\rangle\overline{\varphi_1}\right)\right) \\ &\equiv \neg\,(\mathrm{tt}\,\mathcal{U}\,((\langle a\rangle\neg\varphi_1) \vee (\langle b\rangle\neg\varphi_1))) \\ &\equiv \neg\,(\mathrm{tt}\,\mathcal{U}\neg\,((\langle a\rangle\varphi_1) \vee (\langle b\rangle\varphi_1))) \\ &\equiv \Box\,(\langle a\rangle\varphi_1 \vee \langle b\rangle\varphi_1) \\ &=: \varphi_2 \end{aligned}$$

Finally, $ltl_{\mathcal{A}}(q_0)$ emerges as:

$$\begin{aligned} ltl_{\mathcal{A}}(q_0) &= \bigvee_{a\in\Sigma} \langle a\rangle\, ltl_{\mathcal{A}}(\delta(q_0,a)) \\ &\equiv \langle a\rangle\,(\varphi_1 \wedge \varphi_2) \vee \langle b\rangle\,((\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)) \end{aligned}$$

Obviously, the initial state is—as expected—equivalent to the LTL formula of Example 6.2.7 on page 74 which was the origin of the considered automaton.

A drawback of the previous construction is that it is involved with a blow-up. Starting with an LTL formula, transforming it into a Büchi automaton (of equal size) yields a larger formula when translated back to an LTL formula.

6.4 Trace-consistent $\mathrm{LTL_w}$

As we have seen in Chapter 2 on page 11, Mazurkiewicz traces are an appealing framework for specifying and modeling concurrent systems. On the other hand, $\mathrm{LTL_w}$ is a simple to understand logic for specifying properties of those systems and is widely accepted in industry. However, it's a simple matter to specify properties that are not so-called *trace consistent.* We call a formula $\varphi \in \mathrm{LTL_w}$ *trace consistent* iff the language $\mathcal{L}(\varphi)$ defined by φ is trace consistent.

For example, $\langle a\rangle\langle b\rangle\mathrm{tt}$ defines a language of ω-words with the prefix ab. Thus, the defined language is not necessarily trace consistent. Given a framework in which a and b are independent actions of a concurrent system, one will obtain for every execution sequence beginning with ab also one beginning with ba. Hence, no such concurrent system will satisfy the given requirement. In other words, the formula is not adequate for systems providing the mentioned independence. Note that $\langle a\rangle\langle b\rangle\mathrm{tt} \vee \langle b\rangle\langle a\rangle\mathrm{tt}$ defines a trace-consistent language (if a and b are the only independent actions).

From a practical point of view, it is not convenient to allow a prospective user of a verification tool to formulate such requirements. The user should be guided to specify only requirements respecting the inherent structure of an underlying system.

One solution to this problem is to check whether the formula is compatible with a given independence relation. Stated differently, the problem is to find out whether the formula defines a trace-consistent language. Peled, Wilke and Wolper present in [PWW98] a procedure for checking certain closure properties of ω-regular languages. Among them is also trace consistency for languages defined by LTL formulas. However, they also prove that checking this consistency is PSPACE-complete and hence not efficient.

Apart from complexity issues, the approach has the drawback that it is painful for a user to write a formula, to learn that is not trace consistent and to modify it until it is. We therefore follow a different approach, starting in the next chapter. The logic is directly interpreted over Mazurkiewicz traces (giving the formulas a new meaning, of course). In this way, every formula defines a trace-consistent language. In other words, every requirement for a system will respect its underlying structure automatically.

6.5 Model Checking

The main application for linear temporal logic—we are interested in—is within the domain of formal methods. However, in this area, *model checking* is more important than satisfiability. *Model checking* denotes the problem whether a given language L is a subset of the language defined by a formula φ. The language L is assumed to describe the set of executions of an underlying system while φ is a specification of the system. So $L \subseteq \mathcal{L}(\varphi)$ means that all executions are according to the specification φ.

The model-checking problem has consequently two parameters, first, the language L, usually given by some kind of Büchi automaton, and second, the formula φ. Thus, its complexity can be measured with respect to the size of the (description of the) language L as well as to the size of the formula φ. In the first case, one often speaks about the *program complexity* of model checking while for the latter the term *formula complexity* is used. For practical applications, the program complexity is usually the limiting factor, since the size of the Büchi automaton to study is often much bigger than the formula to check. However, one has to keep an eye on the formula complexity as well, which we do in this section.

$L \subseteq \mathcal{L}(\varphi)$ is equivalent to $L \cap \overline{\mathcal{L}(\varphi)} = \emptyset$ where $\overline{L'}$ denotes the complement of L'. Thanks to negation, we can write this equation as $L \cap \mathcal{L}(\neg\varphi) = \emptyset$.

The automata-theoretic approach makes use of the previous equation. The general idea is to describe the set of executions in terms of a Büchi automaton $\mathcal{A}$ and to define a Büchi automaton $\mathcal{A}_{\neg\varphi}$ accepting precisely the models of $\neg\varphi$ (via an alternating automaton, as proposed here). Then, $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset$ must be tested. The intersection of two languages defined by Büchi automata can easily be reduced two an automata-theoretic construction (cf. [Tho90a]). The resulting Büchi automaton can be tested in linear time for emptiness (cf. Theorem 4.1.9 on page 43).

While this approach truly convinces because of its clear structure, one might ask whether there is a price to pay for this methodology. Clearly, a satisfiability procedure is inhibited within this model-checking procedure. Intuitively, it might be more difficult to search for a model and then to compare whether it is present in a set of executions than directly testing a given execution. Thus, one might think that model checking in easier than satisfiability. While this is true for logics like Kozen's μ-calculus [Koz83], this does not hold in the linear time framework:

Suppose we want to check whether ψ is satisfiable, that is, whether $\mathcal{L}(\psi) \neq \emptyset$. Let a model-checking procedure answering whether $L \subseteq \mathcal{L}(\varphi)$ be given. Take $L = \Sigma^\omega$ and $\varphi = \neg\psi$. Then the model-checking procedure answers whether $\Sigma^\omega \cap \mathcal{L}(\neg\neg\psi) = \emptyset$, and, in this way, whether $\mathcal{L}(\psi) = \emptyset$. Thus, we get an answer whether ψ is satisfiable. We conclude that model checking is at least as complex as satisfiability. We did not explicitly use the syntax of our logic. The only requirement is that it is closed under negation. Of course, we have to take into account the representation of the language handed over to the model checker. As mentioned before, this is usually given in form of a Büchi automaton. Thus, we could restrict model checking to regular languages. However, Σ^ω is regular and can be given by a simple automaton. We conclude that, even when restricted to regular languages, the formula complexity of model checking is at least as high as satisfiability in the domain of words. Note that with the same arguments we show that for every logic that is closed under negation and that is interpreted over words, satisfiability is at most as difficult as model checking. We especially mention the linear temporal logics studied in this thesis that are defined over traces but whose model-checking problem is defined in terms of linearizations.

We sum up:

Theorem 6.5.1
For LTL_w*, the complexity of deciding satisfiability is less than or equal to the formula complexity of model checking.*

The preceding theorem justifies the automata-theoretic approach of model checking, which incorporates a decision procedure for satisfiability.

6.6 FO **versus** LTL

First-order logic turned out to be a fruitful mathematical framework. Introducing a new logical framework requires arguments of its adequacy. This can be achieved by providing several examples for its usefulness but also via a link to a well-known framework. Kamp provided such a link for LTL_w:

Theorem 6.6.1 ([Kam68])
LTL_w *is expressively complete with respect to* FO *over words.*

We call a logic $\mathcal{L}_1$ *expressively complete* with respect to a logic $\mathcal{L}_2$ iff each class definable by a formula of $\mathcal{L}_1$ can also be defined by a formula of $\mathcal{L}_2$ and vice versa. As we have not introduced the notion of a logic formally, let us make this general definition precise for our setting: We say that LTL_w is expressively complete with respect to FO_w iff

- for every $\varphi \in \mathrm{LTL}_\mathrm{w}$, there is a $\psi \in \mathrm{FO}_\mathrm{w}$ such that $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$ and
- for every $\psi \in \mathrm{FO}_\mathrm{w}$, there is a $\varphi \in \mathrm{LTL}_\mathrm{w}$ such that $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$.

Chapter 7

LTL over Mazurkiewicz Traces

In this chapter, we introduce a linear temporal logic which is interpreted over Mazurkiewicz traces and is abbreviated by $\mathrm{LTL_t}$ or LTL if the context makes clear that we interpret the logic over Mazurkiewicz traces. The syntax coincides with that of $\mathrm{LTL_w}$, but the semantics is adapted to respect a given independence relation. In the case of a fully dependent alphabet, in which we can identify traces and words, the semantics of $\mathrm{LTL_w}$ and $\mathrm{LTL_t}$ formulas agree.

$\mathrm{LTL_w}$ has turned out to be well-suited for the specification of hardware and software systems. In Chapter 2 we have seen that Mazurkiewicz traces are an appealing framework for specifying and modeling concurrent systems. Thus, it is natural to look for a linear temporal logic interpreted over Mazurkiewicz traces. It is well-known that $\mathrm{LTL_w}$ is expressively complete with respect to first-order logic over words, a famous result due to Kamp [Kam68]. The "right" logic over traces should therefore be expressively complete with respect to first-order logic over traces.

$\mathrm{LTL_t}$—strictly speaking a syntactically slightly richer version thereof—was introduced by Thiagarajan and Walukiewicz [TW97], and they proved that it is expressively equivalent to first-order logic over Mazurkiewicz traces. Later, Diekert and Gastin showed in [DG00] that even without the restricted past modality employed in [TW97] this logic is expressively complete with respect to first-order logic over traces. Consequently, we consider their LTL version.

In the first section of this chapter, we introduce $\mathrm{LTL_t}$ by means of its syntax and its semantics. We also define a simple fragment of $\mathrm{LTL_t}$ which we call Hennessy-Milner logic since it is in the spirit of the logic defined in [HM85] (see also [Sti01]).

The second section provides one of the main contributions of this thesis: We give a decision procedure for $\mathrm{LTL_t}$ formulas using alternating Büchi automata. To simplify our presentation, we show our method first for the Hennessy-Milner fragment and extend our approach later to support the *until*-operator of $\mathrm{LTL_t}$. We furthermore present a simplified decision procedure for the case that the *until*-operator is substituted by an *eventually*-operator.

7.1 Syntax and Semantics

In this section, we bring out the syntax and semantics of the linear temporal logic interpreted over Mazurkiewicz traces. Its formulas are parameterized by a trace alphabet (Σ, I). Their syntax is defined as for the case of $\mathrm{LTL_w}$:

Definition 7.1.1
The set of linear temporal logic *formulas over an independence alphabet* (Σ, I) *is denoted by* $\mathrm{LTL_t}(\Sigma, I)$ *and is given by the following grammar:*

$$\varphi ::= \mathrm{tt} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle a\rangle\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \ , \quad a \in \Sigma.$$

We usually use identifiers like $\varphi, \psi, \eta, \ldots$ or $\varphi', \psi', \eta', \ldots$ for formulas. When it is clear from the context that we interpret over Mazurkiewicz traces or if the alphabet is fixed, we abbreviate $\mathrm{LTL_t}(\Sigma, I)$ by $\mathrm{LTL}(\Sigma, I)$, $\mathrm{LTL_t}$, or just LTL.

Although $\mathrm{LTL_t}$ formulas are syntactically identical to $\mathrm{LTL_w}$ formulas, they have a different meaning. Formulas of $\mathrm{LTL}(\Sigma, I)$ are interpreted over configurations of traces over (Σ, I). More precisely:

Definition 7.1.2
Given a trace $T \in \mathbb{TR}(\Sigma, I)$*, a configuration* $C \in \mathrm{conf}(T)$*, and a formula* $\varphi \in \mathrm{LTL}(\Sigma, I)$*, the notion of* $T, C \models \varphi$ *is defined inductively via:*

- $T, C \models \mathrm{tt}$.
- $T, C \models \neg\varphi$ *iff* $T, C \not\models \varphi$.
- $T, C \models \varphi \vee \psi$ *iff* $T, C \models \varphi$ *or* $T, C \models \psi$.
- $T, C \models \langle a\rangle\varphi$ *iff there exists a* $C' \in \mathrm{conf}(T)$ *such that* $C \xrightarrow{a}_T C'$ *and* $T, C' \models \varphi$.
- $T, C \models \varphi\mathcal{U}\psi$ *iff there exists a* $C' \in \mathrm{conf}(T)$ *with* $C \subseteq C'$ *such that* $T, C' \models \psi$ *and for all* $C'' \in \mathrm{conf}(T)$ *with* $C \subseteq C'' \subset C'$*, we have* $T, C'' \models \varphi$.

We abbreviate $T, \emptyset \models \varphi$ by $T \models \varphi$. Similarly as in the case of $\mathrm{LTL_w}$, we say T *models* φ, T is a *model* for φ, or T *satisfies* φ iff $T \models \varphi$. Furthermore, we call φ *satisfiable* iff there is a $T \in \mathbb{TR}(\Sigma, I)$ such that $T \models \varphi$. All models of a formula $\varphi \in \mathrm{LTL_t}(\Sigma, I)$ constitute a subset of $\mathbb{TR}(\Sigma, I)$, thus a language. It is denoted by $\mathcal{L}(\varphi)$ and is called the language *defined* by φ. Furthermore, every formula defines an ω-language viz the set $\{w \in \mathrm{lin}(T) \mid T \models \varphi\}$, which is also indicated by $\mathcal{L}(\varphi)$.

Given a (regular) trace-consistent language L and a formula $\varphi \in \mathrm{LTL_t}$, *model checking* is the problem whether $L \subseteq \mathcal{L}(\varphi)$. Analogously to Theorem 6.5.1 on page 81, we obtain the following result:

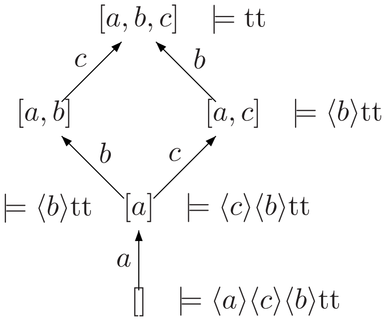


Figure 7.1: The semantics of $\langle \_ \rangle$

Theorem 7.1.3
For $\mathrm{LTL_t}$*, the complexity of deciding satisfiability is less than or equal to the formula complexity of model checking.*

As before, we will freely use the standard abbreviations such as $\mathrm{ff} = \neg\mathrm{tt}$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, and $\varphi \rightarrow \psi = \neg\varphi \vee \psi$.

The meaning of Boolean combinations of formulas is as usual. A new flavor is provided by the temporal operators $\langle \_ \rangle$ and $\mathcal{U}$. Let us illustrate their semantics by giving examples.

A simple $\langle \_ \rangle$-formula[1] of $\mathrm{LTL_t}$ is $\varphi = \langle a\rangle\langle c\rangle\langle b\rangle\mathrm{tt}$. Given a trace T, it is satisfied in the empty configuration iff there is an a-successor configuration satisfying $\langle c\rangle\langle b\rangle\mathrm{tt}$. Let us consider the trace T with the configuration graph shown in Figure 7.1 (cf. Figure 3.1 on page 16, Figure 3.2(a) on page 19, and Figure 3.8 on page 26). The empty configuration indeed has an a-successor configuration C_a (denoted by $[a]$ in the figure), which has to be studied now. C_a has a c-successor configuration C_{ac}, which itself has a b-successor configuration C_{acb}. Thus, φ is satisfied by the trace with the configuration graph shown. The formula φ requested a path labeled by acb in the configuration graph and $\mathcal{CG}(T)$ provides such a graph.

We now might be tempted to think that an $\mathrm{LTL_t}$ formula φ can be understood as an $\mathrm{LTL_w}$ formula interpreted just over the "right" linearization of a trace T. This is not the case! Note that the configuration C_a of T in Figure 7.1 also satisfies the formula $\langle b\rangle\mathrm{tt}$ since it also provides a b-successor configuration. Thus, $T, C_a \models \langle b\rangle\mathrm{tt} \wedge \langle c\rangle\mathrm{tt}$, and, $T \models \langle a\rangle\langle b\rangle\mathrm{tt} \wedge \langle a\rangle\langle c\rangle\mathrm{tt}$. Now, consider the linearizations of T. They start with either ab or ac. But neither satisfies $\varphi' = \langle a\rangle\langle b\rangle\mathrm{tt} \wedge \langle a\rangle\langle c\rangle\mathrm{tt}$ when φ' is considered as an $\mathrm{LTL_w}$ formula. Even more, φ' is not satisfiable at all when considered as an $\mathrm{LTL_w}$ formula.

Intuitively, a Boolean combination of $\langle \_ \rangle$ formulas describes an initial segment of the structure of the configuration graph of a trace T.

---

[1] a formula of the form $\langle a\rangle\psi$ for appropriate a and ψ

The previous example probably raised an unpleasant feeling with respect to the semantics of the $\langle \_ \rangle$-operator since it seems sometimes to be difficult to understand. In many situations, it is very helpful though. Assume that we want to specify that a system has initially to execute the actions a, b, and c to reach a state in which it operates according to a goal expressed by $\varphi_{operate}$. Suppose that a has to be executed before b and c but b and c might be executed in an arbitrary order. In $\mathrm{LTL_w}$ we have to write

$$\langle a\rangle\langle b\rangle\langle c\rangle\varphi_{operate} \vee \langle a\rangle\langle c\rangle\langle b\rangle\varphi_{operate}$$

since we should not care in which order b and c are executed. Specifying only one clause, e.g., $\langle a\rangle\langle b\rangle\langle c\rangle\varphi_{operate}$, we might later fail to verify an implementation of our system since it might provide also the execution b after c.

In the setting of $\mathrm{LTL_t}$, we take an alphabet in which b and c are independent of each other but dependent on a and simply write

$$\langle a\rangle\langle b\rangle\langle c\rangle\varphi_{operate}$$

and do not have to deal with any order of the execution of the events b and c. Note that we could have likewise chosen $\langle a\rangle\langle c\rangle\langle b\rangle\varphi_{operate}$. Both formulas are *equivalent*:

Definition 7.1.4
Two formulas $\varphi, \varphi' \in \mathrm{LTL}(\Sigma, I)$ are called equivalent *iff for every trace $T \in \mathbb{TR}(\Sigma, I)$ and every configuration $C \in \mathrm{conf}(T)$, we have*

$$T, C \models \varphi \textit{ iff } T, C \models \varphi'$$

Let us stress a further fact of $\mathrm{LTL_t}$ formulas. Although the independence relation of a given independence alphabet has no influence on the syntax of $\mathrm{LTL_t}$ formulas, it has on their semantics. $\langle b\rangle\langle c\rangle\mathrm{tt}$ and $\langle c\rangle\langle b\rangle\mathrm{tt}$, for example, are equivalent iff b and c are independent actions. $\langle b\rangle\mathrm{tt} \wedge \langle c\rangle\mathrm{tt}$ is satisfiable iff b and c are independent.

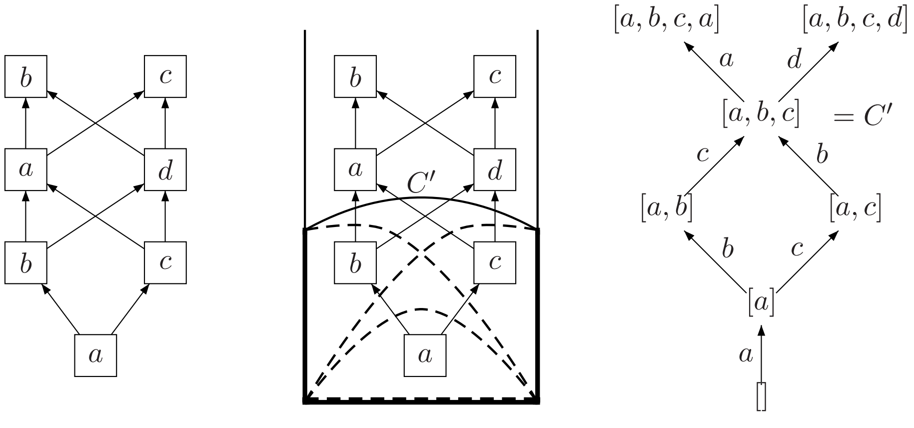
In case of the fully-dependent alphabet, the empty configuration of a trace can be understood as the empty prefix of a word. Furthermore, every configuration has a unique successor configuration. Thus, the $\langle \_ \rangle$-operator of $\mathrm{LTL_t}$ coincides with the $\langle \_ \rangle$-operator of $\mathrm{LTL_w}$.

Now let us try to understand *until*-formulas[2] in more detail. We recall that $T, C \models \varphi \mathcal{U} \psi$ iff there exists a $C' \in \mathrm{conf}(T)$ with $C \subseteq C'$ such that $T, C' \models \psi$ and all $C'' \in \mathrm{conf}(T)$ with $C \subseteq C'' \subset C'$ satisfy φ.

Suppose we consider the trace shown in Figure 7.2(a) on the facing page and the formula $\psi = \langle a\rangle\mathrm{tt} \wedge \langle d\rangle\mathrm{tt}$. It is obvious that configuration C' depicted in Figure 7.2(b) satisfies ψ and that C' is the only one doing so. Thus, for an arbitrary formula φ,

---

[2] formulas of the form $\varphi \mathcal{U} \psi$ for appropriate φ and ψ



(a) (b) (c)

Figure 7.2: The meaning of *until*

the formula $\varphi \mathcal{U} \psi$ holds for the given trace iff every dashed configuration satisfies φ. The situation is also depicted in the (fragmentary) configuration graph shown in Figure 7.2(c). For φ, every configuration on every path from the empty configuration to configuration C' has to be checked. Intuitively, $\varphi \mathcal{U} \psi$ holds if there is a future configuration C' that satisfies ψ and meanwhile φ holds, regardless how C' is reached. Note that the configurations to analyze are in general only partially but not linearly ordered.

In case of the fully-dependent alphabet, the configurations of a trace are linearly ordered and can be identified with prefixes of its linearization. Thus, the *until*-operator of $\mathrm{LTL_t}$ coincides with the *until*-operator of $\mathrm{LTL_w}$. We conclude that $\mathrm{LTL_t}$ and $\mathrm{LTL_w}$ can be identified when the underlying dependence relation is fully-dependent. More specifically, the set of linearizations of all traces satisfying a formula $\varphi \in \mathrm{LTL_t}(\Sigma, I)$ is identical to the set of words satisfying φ when it is considered as a formula of $\mathrm{LTL_w}(\Sigma)$ when I is empty.

From the practical point of view, $\mathrm{LTL_t}$ allows to specify arbitrary liveness and safety properties, similar as in the case of $\mathrm{LTL_w}$. $\neg\,(\mathrm{tt}\,\mathcal{U} \neg\psi)$, for example, is satisfied by a trace T iff in all configurations, the (bad) property ψ does not hold.

Another logic interpreted over partial order executions is *interleaving set temporal logic*, ISTL for short, which was introduced by Katz and Peled in [KP91]. Its syntax and semantics is also defined in the spirit of $\mathrm{LTL_w}$ and adapted towards the setting of partial order executions. One of its main differences to $\mathrm{LTL_t}$ is that the *until*-operator requires a future configuration C' to satisfy ψ and on one path from the current configuration to C' φ has to hold, as opposed to $\mathrm{LTL_t}$, where on every path to C' φ must be satisfied. Figure 7.2(c) illustrates that—as expected—the two definitions are not equivalent. Consider the formula $\eta = (\langle a\rangle \mathrm{tt}\,\mathcal{U}\,\langle c\rangle \mathrm{tt})\;\mathcal{U}\,\langle d\rangle \mathrm{tt}$. $\langle d\rangle \mathrm{tt}$

is obviously satisfied in configuration C'. $\langle c\rangle\mathrm{tt}$ holds in configuration C_a. Thus, along the path $\emptyset\xrightarrow{a}_T C_a$ the formula $\langle a\rangle\mathrm{tt}\,\mathcal{U}\,\langle c\rangle\mathrm{tt}$ holds. Thus, using the semantics employed in ISTL for the *until*-operator, η holds. However, since C_{ac} has no c-successor configuration and not all configurations that are subsets of C_{abcdac} provide an a-successor configuration, η does not hold when understood as an $\mathrm{LTL_t}$ formula. Note that satisfiability of ISTL formulas is undecidable because of the *until*-operator, as shown in [AMP98].

For $\mathrm{LTL_w}$, we saw in Chapter 6 that $\varphi\mathcal{U}\psi$ is equivalent to $\psi \vee \big(\varphi \wedge \bigvee_{a\in\Sigma}\langle a\rangle\varphi\mathcal{U}\psi\big)$: $\varphi\mathcal{U}\psi$ is satisfied iff ψ is satisfied at the current position of the word or the current position satisfies φ and the next position satisfies $\varphi\mathcal{U}\psi$. The next position is identified by using the $\langle\_\rangle$-operator. This equivalence allowed a—conceptually as well as with respect to the complexity—simple treatment of the *until*-operator when defining a decision procedure for $\mathrm{LTL_w}$ formulas. Does such a simple equivalence also hold in the setting of traces? We learned already in the previous paragraph that the equivalence does not hold: Let $\varphi_c = \langle a\rangle\mathrm{tt}\,\mathcal{U}\,\langle c\rangle\mathrm{tt}$ then $\varphi_c\mathcal{U}\langle d\rangle\mathrm{tt}$ is not satisfied in the empty configuration but $\varphi_c \wedge \langle a\rangle\,(\varphi_c \wedge \langle b\rangle\,(\varphi_c\mathcal{U}\langle d\rangle\mathrm{tt}))$ is. The reason is of course that the c-successor configuration of C_a is not studied. A next guess might result in $\psi \vee \big(\varphi \wedge \bigwedge_{a\in\Sigma}\langle a\rangle\varphi\mathcal{U}\psi\big)$ as an equivalent formula for $\varphi\mathcal{U}\psi$ since now every successor configuration is required to satisfy $\varphi\mathcal{U}\psi$. Of course, the idea is not correct because $\langle a\rangle$ requires an a-successor configuration to exist which is often not the case.

Let us study a final plan for the moment. We define a weak next state operator $\mathrm{O}\varphi$ as an abbreviation for $\bigwedge_{a\in\Sigma}(\langle a\rangle\mathrm{tt} \rightarrow \langle a\rangle\varphi)$ with the meaning that every (existing) successor configuration has to satisfy φ. We are now tempted to think that $\varphi\mathcal{U}\psi$ is equivalent to $\psi \vee (\varphi \wedge \mathrm{O}\,(\varphi\mathcal{U}\psi))$. However, again our intuition is deceiving: Let $\psi = \langle b\rangle\mathrm{tt} \wedge \neg\langle c\rangle\mathrm{tt}$, expressing that a configuration has one b-successor but no c-successor configuration. Obviously, it is satisfied by the trace shown in Figure 7.2 on the preceding page in configuration C_{ac} but not in configuration C_a. Thus, $\mathrm{tt}\,\mathcal{U}\psi$ is satisfied in C_a. However, $\psi \vee (\mathrm{tt} \wedge \mathrm{O}\,(\mathrm{tt}\,\mathcal{U}\psi))$ is not satisfied in C_a: $C_a \not\models \psi$ and in its b-successor configuration C_{ab} there is no b-successor configuration reachable which is requested by ψ.

The previous discussion shows that the *until*-operator is somehow awkward to handle. That is why we first identify the so-called *Hennessy-Milner* fragment of $\mathrm{LTL_t}$, which is $\mathrm{LTL_t}$ without *until*-operator. This will simplify the presentation of the forthcoming decision procedure for satisfiability of $\mathrm{LTL_t}$ formulas as well as the estimation of its complexity. We will see in Section 7.2 on the next page how to subdue the *until*-operator.

Definition 7.1.5 (Hennessy-Milner logic)
Hennessy-Milner logic over an independence alphabet (Σ, I) *is the fragment of* $\mathrm{LTL_t}$ *with formulas given by the following grammar:*

$$\varphi ::= \mathrm{tt} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle a\rangle\varphi\ , \quad a \in \Sigma.$$

The semantics is according to Definition 7.1.2 on page 84. We denote the Hennessy-Milner fragment of $\mathrm{LTL_t}$ *by* $\mathrm{LTL_{HM}}$.

It is evident that Hennessy-Milner logic can only speak over a restricted prefix of a trace. Concepts describing properties of its infinite "behavior" are missing. Thus, it is a quite simple logic but the first choice to study *independence* in the context of temporal logics.

Let us end this section by mentioning the main appealing property of $\mathrm{LTL_t}$:

Theorem 7.1.6 ([DG00])
$\mathrm{LTL_t}$ *is expressively complete with respect to* FO *over traces.*

We say that $\mathrm{LTL_t}$ is expressively complete with respect to $\mathrm{FO_t}$, iff the following two propositions are fulfilled.

Proposition 7.1.7 *For every* $\varphi \in \mathrm{LTL_t}$, *there is a* $\psi \in \mathrm{FO_t}$ *such that* $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$.

Proposition 7.1.8 *For every* $\psi \in \mathrm{FO_t}$, *there is a* $\varphi \in \mathrm{LTL_t}$ *such that* $\mathcal{L}(\varphi) = \mathcal{L}(\psi)$.

Since satisfiability of FO over traces was shown to be decidable by Ebinger and Muscholl in [EM93] we immediately get:

Corollary 7.1.9 *Satisfiability of* $\mathrm{LTL_t}$ *formulas is decidable.*

7.2 Deciding Satisfiability of $\mathrm{LTL_t}$

In this section, we present a decision procedure for satisfiability of $\mathrm{LTL_t}$ formulas. Before one starts to consider a decision procedure for a certain problem, it is worthwhile to estimate its complexity. We already learned that satisfiability of $\mathrm{LTL_t}$ is decidable. However, it was shown by Walukiewicz in [Wal98] that its complexity is *non-elementary*, i.e., the complexity with respect to a measure n cannot be bounded by a function that is an iterated composition of an exponential function for a fixed number c, thus by a function of the form

$$\left. 2^{2^{\cdot^{\cdot^{2^n}}}} \right\} c$$

He shows, using the fact that a model can consist of two independent linear orders, that one can describe in $\mathrm{LTL_t}$ very large counters and compare their values. This in turn allows to code "long" computations of Turing machines.

On the first sight, this might limit $\mathrm{LTL_t}$'s usage in verification tools. However, as we will point out in detail, an exponential blow-up only occurs for a nesting of *until*-operators. Since nested *until*-operators are difficult to understand, they are rarely used in practice.

A decision procedure for LTL was given by Gastin, Meyer, and Petit in [GMP98a] (see also [GMP98b]), which is based on automata. Their construction was inspired by the seminal work of Büchi [Büc62] who provided an automaton construction for monadic second-order logic interpreted over words. This construction proceeds *bottom-up*, subject to the inductive structure of the formula. For atomic formulas, a Büchi automaton is constructed and combinations of formulas are reflected by combinations of the previously defined automata. For example, for $\neg\varphi$, an automaton for φ is complemented—involving an exponential blow-up.

Thus, their approach is not based on alternating automata, and more importantly, it requires the construction of the full automaton every time, so optimizations such as on-the-fly checking cannot be applied. A further drawback is its high complexity. While an exponential blow-up is unavoidable for nested *until*-formulas, their procedure produces also an exponential blow-up for every *negation.* Since deeply-nested *until*-formulas are rare in specifications but negations are typical for specifying unwanted behavior, this limits the practical applicability of this procedure.

Our procedure generalizes the approach shown in the previous chapter by constructing an alternating Büchi automaton $\mathcal{A}_\varphi$ accepting the models for a given formula φ. As motivated before let us start with considering satisfiability of $\mathrm{LTL}_{\mathrm{HM}}$, before we cover full $\mathrm{LTL}_{\mathrm{t}}$.

7.2.1 Deciding Hennessy-Milner logic

Our goal is to construct an alternating Büchi automaton $\mathcal{A}_\varphi$ accepting the set of linearizations of traces satisfying a given formula φ. The states of this automaton are derived from an extended subformula closure, which we first define. Following this, we define a notion of independence rewriting of such formulas, and this will eventually become the transition relation of $\mathcal{A}_\varphi$. Finally, we pin down the details of our construction and give a proof of correctness.

In essence, the automaton $\mathcal{A}_\varphi$ accepts a word $w \in \Sigma^\omega$ whenever the corresponding trace T_w satisfies φ. To appreciate the developments to come, we commence with a small example.

Consider the formula $\varphi = \langle a\rangle\langle b\rangle\psi$. Suppose that w_1 is of the form abw' for some $w' \in \Sigma^\omega$. It is then not hard to see that $T_{w_1}, \emptyset \models \varphi$ if and only if $T_{w_1}, C_a \models \langle b\rangle\psi$. Consequently, a device that studies φ can proceed with $\langle b\rangle\psi$ if it is fed with an action a. Note that $\langle b\rangle\psi$ is obtained from $\langle a\rangle\langle b\rangle\psi$ by removing $\langle a\rangle$: $\_\langle b\rangle\psi$.

Let us now take $w_2 = baw'$. Thus, the only difference between w_1 and w_2 is that b and a are swapped. Suppose an algorithm is initialized with b and φ as before. When φ is considered as $\mathrm{LTL}_{\mathrm{w}}$ formula, we suggest that an algorithm stops the analysis of φ since the first given b does not match the a requested by $\varphi = \langle a\rangle \ldots$. There is no chance to satisfy φ.

However, in the domain traces, T_{w_2} might still satisfy φ even though the first action of w_2 is b and not a: If a and b are independent then the empty configuration has an a-successor configuration as well as a b-successor configuration. Thus, if we want a decision procedure to accept all linearizations of a formula φ, we cannot reject w_2 for such an alphabet (yet).

As this point we might ask whether our goal to find a decision procedure accepting precisely all linearizations of all models of a given formula is a tall order. Of course, we could think of an automaton construction accepting only a certain linearization of a trace satisfying the formula at hand. The emptiness procedure for the automaton only answers *yes* or *no* so that the overall procedure does not loose its correctness when instead of all linearizations of a model for the formula only a special characteristic one is accepted.[3] One might even be very pleased when remembering that we employ non-deterministic automata: Perhaps the automaton can guess the "right" linearization to accept? However, the careful reader remembers the discussion in the previous section, which already exposed that there are formulas that are not satisfiable when understood as LTL_w formulas but very well satisfiable when interpreted over traces. In other words, there are LTL_HM formulas that are satisfiable, but no linearization will show this fact when the methods for LTL_w are used. Thus, there is no chance to take over the construction for LTL_w directly.

Let us come back to $\varphi = \langle a\rangle\langle b\rangle\psi$. Suppose a and b are indeed independent. Then we see that $T_{w_2}, \emptyset \models \varphi$ exactly when $T_{w_2}, C_b \models \langle a\rangle\psi$. In this sense, the "proof obligation" at the empty configuration, $\langle a\rangle\langle b\rangle\psi$, has been transformed by b to the proof obligation "$\langle a\rangle\psi$" at C_b; the a-action still has to be witnessed, but the present b has been matched. Note that $\langle a\rangle\psi$ can be obtained from $\langle a\rangle\langle b\rangle\psi$ by erasing $\langle b\rangle$: $\langle a\rangle\_\psi$.

In effect, our automaton proceeds in this way by "independence rewriting" the proof obligations by the actions read. The state space thus consists of all subformulas together with formulas obtained by transformations as described above and Boolean combinations thereof. We will call this set the *extended closure* of φ.

Definition 7.2.1
Let η be a formula of LTL_HM*. We define the* extended closure *of η denoted by $ecl(\eta)$ to be the least set that satisfies the following:*

- *η itself is contained in its closure: $\eta \in ecl(\eta)$.*
- *For $\varphi \vee \psi \in ecl(\eta)$, it also contains the closure of φ and of ψ as well as the disjunction of any φ' and ψ' of the respective closures: $ecl(\varphi) \subseteq ecl(\eta)$, $ecl(\psi) \subseteq ecl(\eta)$, and $\varphi' \vee \psi' \in ecl(\eta)$ for all $\varphi' \in ecl(\varphi)$, $\psi' \in ecl(\psi)$.*

---

[3] We indeed follow this idea for the decision procedure in Chapter 8 on page 119.

- *For $\langle a\rangle\varphi \in ecl(\eta)$, it also contains the extended closure of φ as well as $\langle a\rangle\varphi'$ for every $\varphi' \in ecl(\varphi)$.*
- *For $\varphi \in ecl(\eta)$, it also contains $\neg\varphi \in ecl(\eta)$. We identify $\neg\neg\varphi$ with φ. Hence, $ecl(\eta)$ is closed under negation.*
- *The closure is closed under positive Boolean combinations, i.e., $\mathcal{B}^+(ecl(\eta)) \subseteq ecl(\eta)$.*

At first sight, Definition 7.2.1 on the preceding page might seem not to be well-defined since the domain of $\mathrm{LTL}_{\mathrm{HM}}$ formulas could be left since the closure under positive combination yields subformulas that are conjunctions of subformulas. Note that exactly in these cases, a conjunction has to be understood as an abbreviation.

We assume that all positive Boolean formulas are in disjunctive normal form[4] and, moreover, that they are reduced with respect to idempotency and commutation. With these assumptions we can prove the following essential result.

Proposition 7.2.2 *$ecl(\eta)$ is a finite set for each formula η of* $\mathrm{LTL}_{\mathrm{HM}}$.

Proof
The proof proceeds by a standard induction. The claim is obvious for atomic formulas. For $\eta = \langle a\rangle\varphi$, $ecl(\eta)$ consists of positive Boolean combinations of formulas in $ecl(\varphi)$ and formulas of the form $\langle a\rangle\varphi'$ for $\varphi' \in ecl(\varphi)$. Thus, $|ecl(\eta)| \leq \mathrm{O}(2^{2^{|ecl(\varphi)|}})$. The results for negation and disjunction follow with similar arguments. $\square$

For extended formulas, we will make use of the important notion of its *dual*, which is obtained as usual by applying de Morgan's laws to push negations inwards as far as possible.

Definition 7.2.3
The dual *of a formula is given inductively as follows:*

- $\overline{\mathrm{tt}} = \mathrm{ff}$, $\overline{\mathrm{ff}} = \mathrm{tt}$.
- $\overline{\neg\varphi} = \varphi$.
- $\overline{\varphi \vee \psi} = \overline{\varphi} \wedge \overline{\psi}$, $\overline{\varphi \wedge \psi} = \overline{\varphi} \vee \overline{\psi}$.
- $\overline{\langle a\rangle\varphi} = \neg\langle a\rangle\varphi$.

Again, we have to understand conjunctions or ff as abbreviations in a certain context. Our further study will clarify when.

---

[4] In general, transforming a formula into disjunctive normal form might involve an exponential blow-up. For the moment, we are not interested in exact bounds so that we abstract from this. In a forthcoming detailed analysis we will learn that we can withdraw this assumption.

We are now set to introduce the operator $||\_||_\_$, which will constitute the transition relation of the alternating automaton. Essentially, $||\varphi||_a$ is to be thought of as the independence rewriting of φ by the action a.

It follows from the intuition conveyed earlier that it should be the case that $||\langle a\rangle\psi||_a$ is ψ. Then, for the case where aIb, $||\langle b\rangle\psi||_a = \langle b\rangle\psi'$ where $\psi' = ||\psi||_a$. Of course, whenever aDb and the actions are not identical, then $||\langle b\rangle\varphi||_a$ must be ff because b cannot be the next action of a trace satisfying $\langle a\rangle\langle b\rangle\varphi$. Definition 7.2.4 formally captures this intuition:

Definition 7.2.4
For each formula $\eta \in \mathrm{LTL_{HM}}$ *and each action a, the* rewrite operator $||\eta||_a$ *yields a formula of* $\mathcal{B}^+(ecl(\eta))$ *and is defined inductively via:*

$$\begin{array}{lcl} ||\mathrm{tt}||_a & = & \mathrm{tt} \\ ||\varphi \vee \psi||_a & = & ||\varphi||_a \vee ||\psi||_a \\ ||\neg\varphi||_a & = & \overline{||\varphi||_a} \\ ||\langle b\rangle\varphi||_a & = & \begin{cases} \varphi & \textit{if } a = b \\ \langle b\rangle||\varphi||_a & \textit{if } aIb \\ \mathrm{ff} & \textit{if } aDb, a \neq b \end{cases} \end{array}$$

Note that since $ecl(\eta)$ is closed under positive Boolean combination, we have $ecl(\eta) = \mathcal{B}^+(ecl(\eta))$.

It is not hard to verify that $||\_||_\_$ is well-defined. As will turn out soon, the rewrite operator will be the transition function of the automaton accepting the models of the formulas at hand. To show that our construction is indeed correct, we proceed in two steps. First, we show that the automaton's transition function is correct: We show that given a formula φ and an action a, the formula is satisfied by a trace T in the empty configuration iff $||\varphi||_a$ is satisfied in configuration C_a of T. In terms of automata this means that reading a it is OK to move from state φ to state $||\varphi||_a$.

The second part of our correctness proof consists of defining the remaining components of an automaton, especially initial and final states, and to show that this guides the automaton to accept precisely the models of the underlying formula.

Let us now formulate and prove the first part:

Proposition 7.2.5 *Let* η *be any formula of* $\mathrm{LTL_{HM}}(\Sigma, I)$. *Then for every* $w \in \Sigma^\omega$ *with* $w \equiv vaw'$,

$$T_w, C_v \models \eta \textit{ if and only if } T_w, C_{va} \models ||\eta||_a$$

Proof
The proof proceeds by induction on the formula η. We only show the most important cases since the other cases follow in a similar manner.

The case when $\eta = \mathrm{tt}$ is trivial.

Suppose $\eta = \varphi \vee \psi$. Then $T_w, C_v \models \varphi \vee \psi$ means by definition that $T_w, C_v \models \varphi$ or $T_w, C_v \models \psi$. By induction, this is equivalent to $T_w, C_{va} \models ||\varphi||_a$ or $T_w, C_{va} \models ||\psi||_a$, which is equivalent to $T_w, C_{va} \models ||\varphi \vee \psi||_a$ by definition of the rewrite operator.

Suppose $\eta = \neg\varphi$. By definition, $T_w, C_v \models \neg\varphi$ iff not $T_w, C_v \models \varphi$. Induction yields that not $T_w, C_{va} \models ||\varphi||_a$, which means $T_w, C_{va} \models \neg||\varphi||_a$. The dual of a formula is obviously logically equivalent to the negation of the formula so that the previous statement is equivalent to $T_w, C_{va} \models \overline{||\varphi||_a}$.

Suppose $\eta = \langle b\rangle\varphi$. By definition, $T_w, C_v \models \langle b\rangle\varphi$ iff there is a configuration C' such that $C_v \xrightarrow{b}_T C' = C_{vb}$ and $T_w, C' \models \varphi$. We consider three different cases:

- $b = a$: Then $C' = C_{va}$. Hence $T_w, C_{va} \models \varphi$.
- $b \neq a, bDa$: Then $C_v \xrightarrow{b}_T C'$ and $C_v \xrightarrow{a}_T C_{va}$. However, then w is not a linearization of the trace, which is a contradiction.
- bIa: $T_w, C_{vb} \models \varphi$ is by induction equivalent to $T_w, C_{vba} \models ||\varphi||_a$. Since aIb, this means $T_w, C_{vab} \models ||\varphi||_a$ which is equivalent to $T_w, C_{va} \models \langle b\rangle||\varphi||_a$.

Putting together all the cases, we get that $T_w, C_v \models \langle b\rangle\varphi$ if and only if $T_w, C_{va} \models ||\langle b\rangle\varphi||_a$.

This concludes the proof. □

According to the previous proposition, it is possible to consider the word action by action and to modify the formula according to the rewrite operator.

We can now finally bring out the definition of the alternating Büchi automaton $\mathcal{A}_\varphi$ corresponding to a formula $\varphi \in \mathrm{LTL}_{\mathrm{HM}}(\Sigma, I)$ as follows:

Definition 7.2.6
Given a formula $\varphi \in \mathrm{LTL}_{\mathrm{HM}}(\Sigma, I)$, the alternating Büchi automaton $\mathcal{A}_\varphi$ is the tuple (Q, δ, q_0, F) where

- $Q = ecl(\varphi)$ *is the set of states.*
- $\delta(q, a) = ||q||_a$ *is the transition function.*
- $q_0 = \varphi$ *is the initial state.*
- $F = \{\neg\psi \mid \neg\psi \in ecl(\varphi)\}$ *is the set of accepting states.*

Note that we defined the set of final states to be all negated formulas. The intuitive idea is that failing to prove a proposition infinitely often suffices to assume that its negation is true. In the work of Vardi [Var96], one could likewise take all negated formulas as final states as we did in Chapter 6 in Definition 6.2.5 on page 73. Since in the case of LTL over words, only *until*-formulas can occur infinitely often, the set of final states is restricted to negated *until*-formulas in [Var96].

The correctness of the construction is summarized in the following theorem:

Theorem 7.2.7
Let φ be a formula of $\mathrm{LTL_{HM}}(\Sigma, I)$ and let its alternating Büchi automaton over the alphabet Σ be given as $\mathcal{A}_\varphi = (Q, \delta, q_0, F)$. Then

$$w \in \mathcal{L}(\mathcal{A}_\varphi) \textit{ if and only if } T_w, \emptyset \models \varphi$$

for every $w \in \Sigma^\omega$.

Proof
For $w \in \Sigma^\omega$, we have to show that $\mathcal{A}_\varphi$ has an accepting run on w iff $T_w \models \varphi$. Note that every run has (at most) three kinds of paths

- finite paths ending in tt,
- infinite paths on which from some point on every node is labeled by a $\langle \_ \rangle$-formula, or
- infinite paths on which from some point on every node is labeled by a negated $\langle \_ \rangle$-formula.

For $\psi \in ecl(\varphi)$ and $w = aw' \in \Sigma^\omega$, let $\hat{\delta}(\psi, w)$ be the extension of δ defined by $\hat{\delta}(\psi, aw') = \hat{\delta}(\check{\delta}(\psi, a), w')$ (cf. Chapter 4.2.1, page 50). By Proposition 7.2.5, $\hat{\delta}(\varphi, w) = \hat{\delta}(\check{\delta}(\varphi, a), w') = \hat{\delta}(||\varphi||_a, w')$ and $T_w, C_\epsilon \models \varphi$ iff $T_w, C_a \models ||\varphi||_a$. Now, consider an accepting run of $\mathcal{A}_\varphi$. Its finite paths end in tt, thus all proof obligations are proved. Conversely, a run should be accepted only if the finite paths end in tt, i.e., that all proof obligations are proved indeed. Now, let us consider the infinite paths of a run. These can only occur by reading actions independent of the one given within a (negated) $\langle \_ \rangle$-formula. Thus, the requested successor configuration is not witnessed, which can be accepted iff the underlying $\langle \_ \rangle$-formula is preceded by a negation. This is captured by the acceptance condition for infinite paths given by the final states of the automaton. □

Let us explain our automaton construction for an $\mathrm{LTL_{HM}}$ formula in a more illustrative way by providing an example.

Example 7.2.8 We take $\varphi = \langle a \rangle \langle b \rangle \langle c \rangle \mathrm{tt}$ to be checked for satisfiability over our standard alphabet with actions a, b, c, d where a and d as well as b and c are independent (cf. Example 3.1.4 on page 16). It is easy to see that every trace with a minimal event labeled by a which has two direct successor events labeled by b and respectively c satisfies φ. In other words, the automaton $\mathcal{A}_\varphi$ to be defined has to

$\|\|\_\|\|_{\_}$	a	b	c	d
φ	$\langle b\rangle\langle c\rangle\mathrm{tt}$	ff	ff	$\langle a\rangle\neg\mathrm{tt}$
$\langle b\rangle\langle c\rangle\mathrm{tt}$	ff	$\langle c\rangle\mathrm{tt}$	$\langle b\rangle\mathrm{tt}$	ff
$\langle b\rangle\mathrm{tt}$	ff	tt	$\langle b\rangle\mathrm{tt}$	ff
$\langle c\rangle\mathrm{tt}$	ff	$\langle c\rangle\mathrm{tt}$	tt	ff
$\langle a\rangle\neg\mathrm{tt}$	$\neg\mathrm{tt}$	ff	ff	$\langle a\rangle\neg\mathrm{tt}$
$\neg\mathrm{tt}$	ff	ff	ff	ff

Table 7.1: The transition function of $\mathcal{A}_\varphi$

accept all words with a prefix of the form ab^+c or ac^+b. We should start with the state space of $\mathcal{A}_\varphi$, which is $ecl(\varphi)$. However, as we will see in the next paragraph, it is a good idea to restrict the presentation to the states reachable from the initial sate φ. Let us therefore consider the successor states of φ: Reading a "removes" $\langle a\rangle$, thus $||\varphi||_a = \langle b\rangle\langle c\rangle\mathrm{tt}$. $||\varphi||_b = ||\varphi||_c = \mathrm{ff}$ because a and b as well as a and c are dependent. Finally, let us consider $||\varphi||_d$. As a and d are independent, it evolves to $\langle a\rangle||\langle b\rangle\langle c\rangle||_d$. $||\langle b\rangle\langle c\rangle||_d$ yields ff because d is dependent on b. Note that $\langle a\rangle\mathrm{ff}$ is no formula so that ff has to be understood as an abbreviation for $\neg\mathrm{tt}$. Hence, $||\varphi||_d = \langle a\rangle\neg\mathrm{tt}$. Note that although $\langle a\rangle\neg\mathrm{tt}$ is logically equivalent to ff, our construction does not directly yield ff.

Reading b or c in state $\langle b\rangle\langle c\rangle\mathrm{tt}$ yields $\langle c\rangle\mathrm{tt}$ or respectively $\langle b\rangle\mathrm{tt}$. a and d will guide the automaton to state ff.

A little more involved are the successor states of $\langle a\rangle\neg\mathrm{tt}$. Reading a yields $\neg\mathrm{tt}$. Since b and c are dependent on a, the automaton will move to state ff reading these actions. If the next input symbol is d, we obtain $||\langle a\rangle\neg\mathrm{tt}||_d = \langle a\rangle||\neg\mathrm{tt}||_d = \langle a\rangle\overline{||\mathrm{tt}||_d} = \langle a\rangle\overline{\mathrm{tt}} = \langle a\rangle\mathrm{ff}$, which has to be understood as an abbreviation for $\langle a\rangle\neg\mathrm{tt}$.

Let us summarize the definition of the transition function in Table 7.1.

The set of final states consists only of $\neg\mathrm{tt}$ because it is the only state/formula beginning with a negation symbol. However, since there are no cycles containing this state, every accepting run has to reach state tt. The automaton can be visualized as in Figure 7.3 on the facing page where edges to the state ff are left out to enhance the presentation. It is easy to see that along every path reaching tt, we indeed read a word of the desired form.

Before we study the complexity of our procedure, let us consider it for the case that the underlying alphabet is fully-dependent so that traces and words can be identified. We have seen that LTL_t and LTL_w coincide over this alphabet. What happens for the decision procedure? In case of the fully dependent alphabet, no two

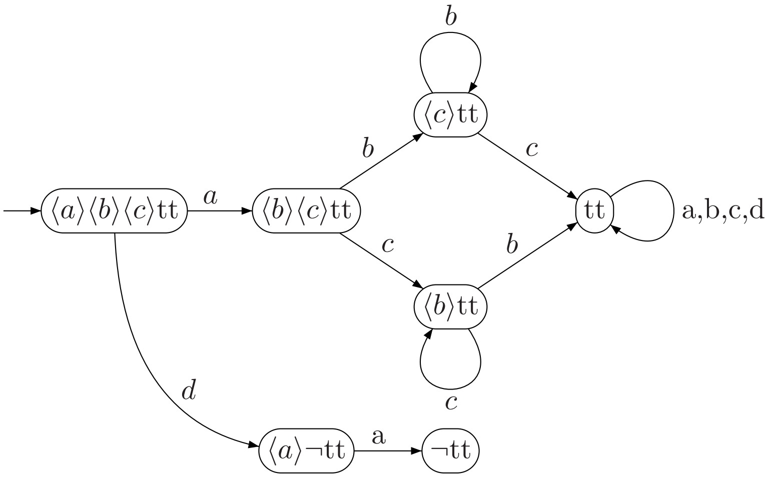


Figure 7.3: The graphical representation of $\mathcal{A}_\varphi$

independent actions exist. Hence,

$$||\langle b\rangle\varphi||_a \quad = \quad \begin{cases} \varphi & \text{if } a = b \\ \langle b\rangle||\varphi||_a & \text{if } aIb \\ \text{ff} & \text{if } aDb, a \neq b \end{cases}$$

of Definition 7.2.4 on page 93 can be simplified to

$$||\langle b\rangle\varphi||_a \quad = \quad \begin{cases} \varphi & \text{if } a = b \\ \text{ff} & \text{else} \end{cases}$$

and we get exactly the construction shown in Section 6.2 on page 71 for LTL$_t$ (when restricted to formulas without *until*-operators).

Complexity Proposition 7.2.2 guarantees that the state space of the constructed automaton is finite. When we take a look at its proof, we get a double exponential upper bound for the state space with respect to the length of the underlying formula. This gives immediately a triple exponential upper bound for the overall decision procedure for satisfiability.

However, to appreciate the presentation, we have defined the state space of our automaton in a straightforward manner and presented a simple argument to show that it is finite to guarantee that we indeed get a decision procedure. When we analyze our construction in more detail, we obtain a smaller upper bound. Let us find out, which states are really needed in our construction. In other words, let us consider the states that are reachable from the initial state.

Given a formula $\varphi \in \mathrm{LTL}_{\mathrm{HM}}(\Sigma, I)$, a state ψ of $\mathcal{A}_\varphi$, and a set $Y \subseteq \Sigma$, let $\mathit{reach}_Y(\psi)$ denote the set of states reachable from ψ in $\mathcal{A}_\varphi$ by words whose actions are independent of Y. More precisely:

$$\mathit{reach}_Y(\psi) = \{\psi' \mid \exists w \in \Sigma^*,\ wIY :\ \psi' \in \mathit{st}(\hat{\delta}(\psi, w))\}$$

where $\hat{\delta}$ is the extension of δ defined in the obvious manner (cf. Chapter 4.2.1, page 50), $\mathit{st}(\varphi)$ yields the disjuncts of φ (cf. Remark 4.2.3 on page 46), and wIY is a shorthand for $\mathrm{alph}(w)IY$. Let us give an inductive characterization of the set of states reachable from a given one:

Proposition 7.2.9 *Given $\varphi \in \mathrm{LTL}_{\mathrm{HM}}(\Sigma, I)$, we get upper bounds for the number of states reachable from a state of $\mathcal{A}_\varphi$ by words independent of Y inductively as follows:*

- $|\mathit{reach}_Y(\mathrm{tt})| = 1$
- $|\mathit{reach}_Y(\mathrm{ff})| = 1$
- $|\mathit{reach}_Y(\neg\psi)| = |\mathit{reach}_Y(\psi)|$
- $|\mathit{reach}_Y(\psi_1 \vee \psi_2)| \leq |\mathit{reach}_Y(\psi_1)| + |\mathit{reach}_Y(\psi_2)|$
- $|\mathit{reach}_Y(\psi_1 \wedge \psi_2)| \leq |\mathit{reach}_Y(\psi_1)| + |\mathit{reach}_Y(\psi_2)|$
- $|\mathit{reach}_Y(\langle a\rangle\psi)| \leq \begin{cases} |\mathit{reach}_Y(\psi)| + |\mathit{reach}_{Y\cup\{a\}}(\psi)| + 1 & \text{if } aIY \\ |\mathit{reach}_{Y\cup\{a\}}(\psi)| + 1 & \text{if } aDY \end{cases}$

Proof

The obvious cases are if the state formula is tt or ff.

Since negation is shifted inwards by the dual operator $\bar{\ }$, the states reachable from $\neg\psi$ are the same states that are reachable from ψ, except that every state is preceded by $\neg$. Thus, the cardinality is the same.

Given $\langle a\rangle\psi$, assume a to be independent of Y. Reading an action dependent on but different from a (and independent of Y) yields the state ff and in our formula the 1. Reading a yields the state ψ, thus, the states reachable from ψ are obviously reachable from $\langle a\rangle\psi$, which results in $|\mathit{reach}_Y(\psi)|$. The last possibility is reading an action b independent of a and Y. This yields formulas of the form $\langle a\rangle\psi'$ where ψ' is obtained by rewriting ψ by actions independent of Y and a. Since $\langle a\rangle\psi'$ distributes over disjunctions and conjunctions, we get the same number of states as obtained by considering the states reachable from ψ by words independent of $Y \cup \{a\}$ ($|\mathit{reach}_{Y\cup\{a\}}(\psi)|$). If a is dependent on some action in Y, the case yielding $|\mathit{reach}_Y(\psi)|$ will not occur since $\langle a\rangle$ cannot be "removed". □

With the help of the operator $\mathit{reach}_Y(\psi)$ and the previous proposition we are able to give a coarser bound for the states reachable from a given state:

Proposition 7.2.10 *For every state* ψ *of an automaton* $\mathcal{A}_\varphi$ *for a formula* $\varphi \in \mathrm{LTL}_{\mathrm{HM}}(\Sigma, I)$*, we obtain* $|\mathit{reach}_Y(\psi)| \leq |\psi|^{|\Sigma - Y|}$.

Proof
First, recall the binomial formula $(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$. The proof follows by induction of which we pick out the most difficult two cases:

Consider $|\mathit{reach}_Y(\psi_1 \vee \psi_2)|$. By Proposition 7.2.9 on the facing page, this number is smaller than $|\mathit{reach}_Y(\psi_1)| + |\mathit{reach}_Y(\psi_2)|$. Using the induction hypothesis, we know that the sum is smaller than $|\psi_1|^{|\Sigma-Y|} + |\psi_2|^{|\Sigma-Y|}$. Let $n = |\Sigma - Y|$ so that we abbreviate the studied term by $|\psi_1|^n + |\psi_2|^n$. The previous value can be augmented by adding $\sum_{i=1}^{n-1} \binom{n}{i} |\psi_1|^{n-i} |\psi_2|^i$. The binomial formula now yields that the cardinatility is bounded by $(|\psi_1| + |\psi_2|)^{|\Sigma-Y|}$ which is of course smaller than $(|\psi_1| + |\psi_2| + 1)^{|\Sigma-Y|}$.

Let us study $|\mathit{reach}_Y(\langle a \rangle \psi)|$. Assume that a is independent of Y. From Proposition 7.2.9 on the preceding page we learn, that the value is smaller than $|\mathit{reach}_Y(\psi)| + |\mathit{reach}_{Y \cup \{a\}}(\psi)| + 1$. Induction yields the bound $|\psi|^{|\Sigma-Y|} + |\psi|^{|\Sigma-Y|-1} + 1$ which reads with $n = |\Sigma - Y|$ as $|\psi|^n + |\psi|^{n-1} + 1$. Note that $|\psi|^n$ can be written as $|\psi|^n \cdot 1^0$ and $|\psi|^{n-1}$ is less or equal $\binom{n}{1} |\psi|^{n-1}$ so that we obtain the desired upper bound $(|\psi| + 1)^{|\Sigma-Y|}$ by applying the binomial formula. $\square$

Let us come back to a formula φ and its automaton $\mathcal{A}_\varphi$. The previous proposition limits the number of states reachable from the initial state φ by $|\varphi|^{|\Sigma-Y|}$ for an arbitrary set Y. This value is maximized when Y equals the empty set. Thus, the number of states visited and hence to be constructed is bounded by $|\varphi|^{|\Sigma|}$ which is a polynomial in the size of the underlying formula with degree bounded by the size of the alphabet.

Due to the exponential blow-up, the construction of an equivalent Büchi automaton for $\mathcal{A}_\varphi$ causes, we conclude

Corollary 7.2.11 *Checking satisfiability of a formula from the Hennessy-Milner fragment* $\mathrm{LTL}_{\mathrm{HM}}(\Sigma, I)$ *can be done in exponential time and polynomial space with respect to the size of* φ.

7.2.2 Supporting Until-formulas

In this section, we will see how to extend our decision procedure towards full LTL$_\mathrm{t}$. Therefore, we augment all definitions, propositions and proofs with appropriate notions for the remaining *until*-operator.

For bringing out the decision procedure itself, it will be convenient to assume that the syntax of LTL is augmented with an indexed *until*-operator $\Phi \mathcal{U}^Z \psi$ where $\Phi = \{\varphi_1^{Y_1}, \ldots, \varphi_n^{Y_n}\}$ is a finite set of annotated formulas, where Z and $Y_1, \ldots, Y_n$ are subsets of actions. Formally, it will have the following semantics:

- $T, C \models \Phi\mathcal{U}^Z\psi$ iff there exists a $C' \in \text{conf}(T)$ with $C \subseteq C'$ such that $T, C' \models \psi$ and $\lambda(C' - C)IZ$. Moreover, for each $1 \leq i \leq n$ and every C'' with $C \subseteq C'' \subset C'$ and $\lambda(C'' - C)IY_i$, it holds $C'' \models \varphi_i$.

Hence, a trace satisfies the formula $\Phi\mathcal{U}^Z\psi$ in the configuration C iff there is a future configuration C' satisfying ψ and all the actions from C to C' are independent of the actions in Z. Furthermore, the configurations between C and C' that can be reached from C by performing actions independent of Y_i, all satisfy φ_i.

Note that $\varphi\mathcal{U}\psi$ can be identified with $\{\varphi^{\emptyset}\}\mathcal{U}^{\emptyset}\psi$ and we will not always make this distinction explicit.

Let us present some examples. Recall that $\langle d\rangle\text{tt}$ is satisfied by the trace shown in Figure 7.2 on page 87 exactly in configuration C'. Thus, $\{\text{tt}^{\emptyset}\}\mathcal{U}^Z\langle d\rangle\text{tt}$ is satisfied in the empty configuration for $Z = \emptyset$. However, it is not for $Z = \{d\}$ because configuration C' can only be obtained by adding a b- and a c-event to the empty configuration but both actions are dependent on d. We turn our interest towards the formula $\varphi = (\langle b\rangle\text{tt} \wedge \langle c\rangle\text{tt})\mathcal{U}\langle d\rangle\text{tt}$. We easily see that it is not satisfied in configuration C_a of the trace shown in Figure 7.2: C' satisfies $\langle d\rangle\text{tt}$, C_a models $\langle b\rangle\text{tt} \wedge \langle c\rangle\text{tt}$ but C_{ab} has no b-successor configuration and C_{ac} has no c-successor configuration. However, $\{(\langle b\rangle\text{tt})^{\{b\}}, (\langle c\rangle\text{tt})^{\{c\}}\}\mathcal{U}^{\emptyset}\langle d\rangle\text{tt}$ is satisfied in configuration C_a. C' still satisfies $\langle d\rangle\text{tt}$ and because the *until*-operator is indexed with the empty set, there is no restriction on the labels of the events added to gain C'. The configurations between C_a and C' are C_a, C_{ab}, and C_{ac}. C_a still satisfies $\langle b\rangle\text{tt}$ and $\langle c\rangle\text{tt}$. As C_{ab} is obtained from C_a by adding a b-event, which is of course dependent on b, there is no reason for $\langle b\rangle\text{tt}$ to hold in C_{ab}. Similarly, although C_{ac} does not satisfy $\langle c\rangle\text{tt}$, the overall formula holds, because C_{ac} evolves from C_a by adding c.

The examples show that with the help of the extended *until*-operator a finer control on the configurations to be considered can be realized.

Let us convince ourself, that $\Phi\mathcal{U}^Z\psi$ is derivable within LTL itself and has consequently no influence on the expressiveness of LTL. Instead of defining for $\Phi\mathcal{U}^Z\psi$ an equivalent LTL formula directly, we provide a translation of LTL_t with extended *until*-operator to FO_t. As the latter captures exactly the LTL-definable languages (cf. Theorem 7.1.6 on page 89), we know that LTL with and without extended *until*-operator coincide with respect to expressiveness. Let $\text{LTL}^Z(\Sigma, I)$ denote the set of LTL formulas with extended *until*-operator.

Proposition 7.2.12 *Let* $\varphi \in \text{LTL}^Z(\Sigma, I)$. *Then there exists* $\eta \in \text{FO}_\text{t}(\Sigma, I)$ *such that*

$$\mathcal{L}(\varphi) = \mathcal{L}(\eta)$$

Proof

As pointed out in Section 5.2 on page 64, we use finite sets of variables to represent configurations. For a countable set of variables *Var*, for every finite set $X \subset \mathit{Var}$, and every formula $\varphi \in \mathrm{LTL}^Z(\Sigma, I)$, we will construct a formula η_φ^X of $\mathrm{FO}(\Sigma, I)$ with free variables in the set X. This formula will have the property that for every valuation $V : \mathit{Var} \to E$

$$T \models_V \eta_\varphi^X \text{ iff } T, C_V^X \models \varphi$$

where C_V^X denotes the configuration identified by X (cf. Section 5.2 on page 64). In particular, taking $X = \emptyset$ will obtain the desired result.

The construction proceeds by structural induction on φ. If $\varphi = \mathrm{tt}$ then for every X we put $\eta_\varphi^X = \mathrm{tt}$. The cases for disjunction and negation are straightforward.

Suppose $\varphi = \langle a \rangle \eta$. Let $X = \{x_1, \dots, x_k\}$ (which may be empty). As pointed out in Section 5.2 on page 64, the formula η_φ^X defined by

$$\exists y\ R_a(y) \wedge \varphi_\eta^{X \cup \{y\}} \wedge \left(\bigwedge_{i=1,\dots,k} y \not\leq x_i \right) \wedge \forall z \left(z < y \to \left(\bigvee_{i=1,\dots,k} z \leq x_i \right) \right)$$

has the desired property.

Suppose $\varphi = \Phi \mathcal{U}^Z \psi$ where $\Phi = \{\varphi_1^{Y_1}, \dots, \varphi_n^{Y_n}\}$. For $X \neq \emptyset$, we let η_φ^X be given by

$$\exists Y'\ \mathrm{BelowIndep}(X, Y', Z) \wedge \eta_\psi^{Y'} \wedge \bigwedge_{\varphi_i^Y \in \Phi} \forall Y''\ (\mathrm{BelowIndep}(X, Y'', Y) \wedge \mathrm{SBelow}(Y'', Y') \to \eta_{\varphi_i}^{Y''})$$

In the above, the quantifier $\exists Y'$ is a shorthand for $\exists y_1' \dots \exists y_{|\Sigma|}'$. Similar holds for $\forall Y''$. Note that we make use of the abbreviations defined in Section 5.2 on page 64.

For $X = \emptyset$, we have to take into account that ψ holds in the empty configuration which cannot be identified with a non-empty set of variables (like $y_1', \dots, y_{|\Sigma|}'$). Thus, we define $\eta_\varphi^\emptyset$ by

$$\eta_\psi^\emptyset \vee \exists Y'\ \mathrm{BelowIndep}(\emptyset, Y', Z) \wedge \eta_\psi^{Y'} \wedge \bigwedge_{\varphi_i^Y \in \Phi} \forall Y''\ (\mathrm{BelowIndep}(\emptyset, Y'', Y) \wedge \mathrm{SBelow}(Y'', Y') \to \eta_{\varphi_i}^{Y''})$$

The case for $\varphi \mathcal{U} \psi$ is just a special case of the previous one.

By structural induction, the claim follows. □

Note that the previous proposition proves Proposition 7.1.7 on page 89, which provides one direction of the expressive-completeness result. The other direction is employed for showing that for every formula of $\mathrm{LTL}^Z(\Sigma, I)$ there is one of $\mathrm{LTL}(\Sigma, I)$ defining the same language.

Following the scheme worked out in the previous section, we now have to extend the definition of the subformula closure towards *until*-formulas:

Definition 7.2.13 (extends Definition 7.2.1)
Let η be a formula of LTL*. We take $ecl(\eta)$ to be the least set that satisfies the items of Definition 7.2.1 on page 91 and furthermore:*

- *For $\varphi\mathcal{U}\psi \in ecl(\eta)$, the closure contains $ecl(\varphi)$ as well as $ecl(\psi)$. Furthermore, for all $Z \subseteq \Sigma$, all $\psi' \in ecl(\psi)$, and all $\Phi \subseteq \{\varphi'^{Y} | \varphi' \in ecl(\varphi), Y \subseteq \Sigma\}$, the closure contains $\Phi U^Z \psi'$.*
- *For $\{\varphi_1^{Y_1}, \dots, \varphi_n^{Y_n}\}\mathcal{U}^Z\psi \in ecl(\eta)$, the closure contains $ecl(\varphi_i)$ as well as $ecl(\psi)$, for $i \in \{1, \dots, n\}$. Furthermore, for all $Z \subseteq Z' \subseteq \Sigma$, all $\psi' \in ecl(\psi)$, and all $\Phi \subseteq \{\varphi'^{Y'} | \exists i\ \varphi' \in ecl(\varphi_i), Y_i \subseteq Y' \subseteq \Sigma\}$, the closure contains $\Phi U^{Z'} \psi'$.*

Again, the definition reflects the idea that "everything" which might be obtained by rewriting is contained in the closure while assuring the closure to be finite:

Proposition 7.2.14 *$ecl(\eta)$ is a finite set for each formula η of* LTL_t*.*

Proof
For $\eta = \{\varphi_1^{Y_1}, \dots, \varphi_n^{Y_n}\}\mathcal{U}^Z\psi$ it is easy to see that $|ecl(\eta)|$ is bounded by

$$2^{2^{\left(2^{2^{\sum_{i=1}^{n}(|ecl(\varphi_i)|\cdot 2^{|\Sigma|})}}\cdot|ecl(\psi)|\cdot 2^{|\Sigma|}\right)}}$$

The three factors are upper bounds for the derivatives of $\{\dots\}$, $\mathcal{U}^Z$, and ψ, respectively, and the powers bound their positive Boolean combination. The remaining cases are treated as in the proof of Proposition 7.2.2 on page 92. ☐

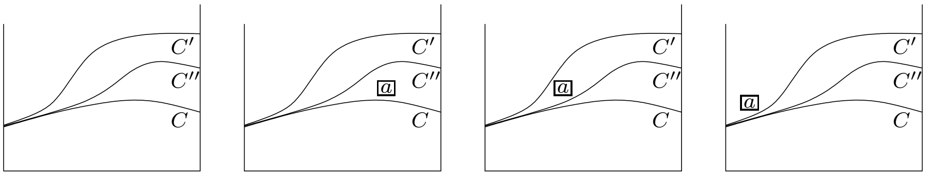
We call formulas of the form $\Phi\mathcal{U}^Z\psi$ with Φ being a finite set of extended formulas, ψ a single extended formula and $Z \subseteq \Sigma$ again *until*-formulas.

We likewise will make use of the important notion of its *dual*, which is obtained as usual by applying de Morgan's laws to push negations inwards as far as possible.

Definition 7.2.15 (extends Definition 7.2.3)
The dual *of an (extended) formula is given inductively by the rules stated in Definition 7.2.3 on page 92 as well as:*

- $\overline{\varphi\mathcal{U}\psi} = \neg(\varphi\mathcal{U}\psi)$.
- $\overline{\Phi\mathcal{U}^Z\psi} = \neg(\Phi\mathcal{U}^Z\psi)$.

We now only need to specify the case of $||\Phi\mathcal{U}^Z\psi||_a$. This turns out to be inherently more complex, and before providing the precise definition, we carefully analyze the semantics of the indexed until modality in Figure 7.4. For this purpose, consider some given trace T and suppose $C, C' \in \mathrm{conf}(T)$ such that $C \subseteq C'$. Furthermore, let C'' be a configuration between C and C' (Figure 7.4(i)). Of course, the idea in



(i) (a) (b) (c)

Figure 7.4: Configuration and actions.

mind is that we consider an *until*-formula $\varphi\mathcal{U}\psi$ in the current configuration C and assume that ψ holds in configuration C'. Then φ has to be considered in C''.

Let us now consider the next action of a linearization. In other words, suppose we can augment C by an a-labeled event e to obtain a successor configuration C''' of C, i.e., $C\xrightarrow{a}_T C'''$. Then $C''' \subseteq C'' \subseteq C'$, or $C''' \not\subseteq C''$ but $C''' \subseteq C'$, or $C''' \not\subseteq C'$. This situation can be stated also in the following way (cf. Figure 7.4 (a) – (c)): The action a is neither in the future of C'' nor of C' (Case (a)), in the future of C'' (Case (b)) or in the future of C'' as well as of C' (Case (c)).

In Case (b), it is obvious that $\lambda(C'' - C)Ia$, and for Case (c), we have $\lambda(C' - C)Ia$ (as well as $\lambda(C'' - C)Ia$).

Consider a formula $\varphi\mathcal{U}\psi$ which is to be checked in the configuration C. In Case (c), we have to employ a for verifying ψ as well as φ. Note that for Case (c) we get two subcases depending upon whether $C' = C$ or $C' \supset C$. While φ is not relevant in the first case, φ is required to hold in the configurations between C and C'. Note that these configurations are reached by actions independent of a.

For Case (a), we have to use a for verifying φ in configuration C but not for C''. In Case (b), we have to prove φ considering a in the configuration C'', which might be equal to C as well as different from C. Note that in the latter case, every event of $C'' - C$ is independent of a.

Consequently, we define the rewrite operator for a formula $\Phi\mathcal{U}^Z\psi$ as follows.

Definition 7.2.16 (extends Definition 7.2.4)
Let

$$\Psi_1 = ||\psi||_a \quad \Psi_2 = \{||\varphi||_a^{Y\cup\{a\}} \mid \varphi^Y \in \Phi\}\,\mathcal{U}^{Z\cup\{a\}}||\psi||_a.$$

Moreover, we set $\Psi' = \Psi_1 \vee \Psi_2$. *Let*

$$\begin{array}{rcll} ||\Phi||_a & = & \{\ ||\varphi||_a^{Y\cup\{a\}} & \mid \varphi^Y \in \Phi\} \\ & \cup & \{\ \varphi^Y & \mid \varphi^Y \in \Phi, aIY\} \end{array}$$

and

$$\Phi_1 = \bigwedge_{\varphi^Y \in \Phi} ||\varphi||_a \quad \Phi_2 = ||\Phi||_a\mathcal{U}^Z\psi$$

and $\Phi' = \Phi_1 \wedge \Phi_2$.

Then we define

$$||\Phi\mathcal{U}^Z\psi||_a = \begin{cases} \Psi' & \textit{if } aDZ \\ \Psi' \vee \Phi' & \textit{if } aIZ \end{cases}$$

Note that Ψ' captures Case (c) in which an action a is used for verifying ψ under the assumption that $C' = C$ (Ψ_1) or not (Ψ_2). Φ' covers the idea that a is not in the future of C' but is employed for verifying the obligations in Φ.

We now state that our rewrite operator is locally correct:

Proposition 7.2.17 *Let* η *be any formula of* $\mathrm{LTL_t}(\Sigma, I)$. *Then for every* $w \in \Sigma^\omega$ *with* $w \equiv vaw'$,

$$T_w, C_v \models \eta \textit{ if and only if } T_w, C_{va} \models ||\eta||_a$$

Proof
As in Proposition 7.2.5 on page 93, the proof proceeds by induction on the structure of the formula. The only remaining case is $\eta = \Phi\mathcal{U}^Z\psi$. Let $\Phi = \{\varphi_1^{Y_1}, \ldots, \varphi_N^{Y_N}\}$. Recall that $T_w, C_v \models \{\varphi_1^{Y_1}, \ldots, \varphi_N^{Y_N}\}\mathcal{U}^Z\psi$ if and only if

$$\begin{array}{l} \exists x \in \Sigma^*, y \in \Sigma^\omega, aw' \equiv xy, xIZ, \text{ such that } T_w, C_{vx} \models \psi, \text{ and} \\ \forall i \in \{1, \ldots, N\}, \forall x_1, x_2 \in \Sigma^* \text{ satisfying } x_1x_2 \equiv x, x_1IY_i, x_2 \neq \varepsilon, \text{ it holds} \\ T_w, C_{vx_1} \models \varphi_i. \end{array}$$

We consider here only the case where $aI(Y_i \cup Z)$. The other cases follow similarly. Let us first discuss the "*if*"-part: We consider the following cases for x:

- $x = \varepsilon$: Then $T_w, C_{vx} \models \psi$ means $T_w, C_v \models \psi$ which implies by induction $T_w, C_{va} \models ||\psi||_a$. This shows ($\Psi_1$).
- $x \neq \varepsilon, a \notin \mathrm{alph}(x)$:
  We consider the cases for ψ and φ_i simultaneously:

$$\begin{array}{ll|ll} \Rightarrow & aIx & \Rightarrow & aIx_1 \\ & T_w, C_{vx} \models \psi & & T_w, C_{vx_1} \models \varphi_i \\ \overset{I.H.}{\Rightarrow} & T_w, C_{vxa} \models ||\psi||_a & \overset{I.H.}{\Rightarrow} & T_w, C_{vx_1a} \models ||\varphi_i||_a \\ \Rightarrow & T_w, C_{vax} \models ||\psi||_a & \Rightarrow & T_w, C_{vax_1} \models ||\varphi_i||_a \\ \Rightarrow & \exists x \in \Sigma^*, xI(Z \cup \{a\}), & \Rightarrow & \forall x_1, x_2 \in \Sigma^*, x_1x_2 \equiv x, x_1I(Y_i \cup \{a\}), \\ & T_w, C_{vax} \models ||\psi||_a. & & x_2 \neq \varepsilon, T_w, C_{vax_1} \models ||\varphi_i||_a. \end{array}$$

  Hence, $T_w, C_{va} \models \{||\varphi||_a^{Y \cup \{a\}} \mid \varphi^Y \in \Phi\}\mathcal{U}^{Z \cup \{a\}}||\psi||_a$ which shows (Ψ_2).

- $x \neq \varepsilon, a \in \text{alph}(x)$:
  We easily see that $x \equiv ax'$ and $a, x'IZ$ and $T_w, C_{vax'} \models \psi$. We will show (Φ_1) and (Φ_2). Let us consider x_1. If $x_1 = \varepsilon$ then $T_w, C_v \models \varphi_i$ implies by induction $T_w, C_{va} \models ||\varphi_i||_a$. If $x_1 \neq \varepsilon$ and $a \notin \text{alph}(x_1)$, we see that aIx_1 and x_1IY_i. Hence, $x_1I(Y_i \cup \{a\})$. Now, $T_w, C_{vx_1} \models \varphi_i$ yields by induction $T_w, C_{vx_1a} \models ||\varphi_i||_a$ proving $T_w, C_{vax_1} \models ||\varphi_i||_a$ since aIx_1. For the case $x_1 \neq \varepsilon$ but $a \in \text{alph}(x_1)$ we see that $x_1 \equiv ax_1'$ and $T_w, C_{vax_1'} \models \varphi_i$. Summing up the cases for x_1 we get $T_w, C_{va} \models ||\Phi||_a \mathcal{U}^Z \psi$, which shows (Φ_2), and $T_w, C_{va} \models ||\varphi||_a$ for all $\varphi^Y \in \Phi$, which shows (Φ_1).

Altogether, we showed that Ψ' or Φ' hold in the until case proving the "*if*"-part. Now, let us consider the "*only-if*" part: Suppose $T_w, C_{va} \models ||\Phi \mathcal{U}^Z \psi||_a$, i.e.

$$T_w, C_{va} \models \Psi_1 \vee \Psi_2 \vee \Phi'.$$

We discuss the disjunction by drawing the conclusions of each formula

- $T_w, C_{va} \models ||\psi||_a$:
  This implies by induction that $T_w, C_v \models \psi$. Hence, $T_w, C_v \models \Phi \mathcal{U}^Z \psi$.

- $T_w, C_{va} \models \{||\varphi||_a^{Y \cup \{a\}} \mid \varphi^Y \in \Phi\} \mathcal{U}^{Z \cup \{a\}} ||\psi||_a$:
  Then there exist x, y, $xI(Z \cup \{a\})$, $w \equiv vaxy$ such that $T_w, C_{vax} \models ||\psi||_a$. Since xIa also $T_w, C_{vxa} \models ||\psi||_a$ which yields by induction $T_w, C_{vx} \models \psi$. We further know that for every proper prefix (modulo $\equiv$) x_1 of x with $x_1I(Y \cup \{a\})$ we have $T_w, C_{vax_1} \models ||\varphi||_a$. Then $T_w, C_{vx_1a} \models ||\varphi||_a$ and, by induction, $T_w, C_{vx_1} \models \varphi$. Hence, $T_w, C_v \models \Phi \mathcal{U}^Z \psi$.

- $T_w, C_{va} \models \bigwedge_{\varphi^Y \in \Phi} ||\varphi||_a \wedge ||\Phi||_a \mathcal{U}^Z \psi$:
  We first obtain by induction that $T_w, C_v \models \varphi$ for every $\varphi^Y \in \Phi$, considering the first conjunct. Let us analyze

  $$T_w, C_{va} \models (\{||\varphi||_a^{Y \cup \{a\}} \mid \varphi^Y \in \Phi\} \cup \{\varphi^Y \mid \varphi^Y \in \Phi, aIY\}) \mathcal{U}^Z \psi$$

  It implies that there is an x', independent of Z, such that $T_w, C_{vax'} \models \psi$. Since we are in the case of aIZ, we conclude that there is an x, xIZ (viz $x \equiv ax'$) such that $T_w, C_{vx} \models \psi$. Now, consider $x_1x_2 \equiv x$, $x_2 \neq \varepsilon$. For every $x_1I(Y \cup \{a\})$, x_1 a prefix of x', we know $T_w, C_{vax_1} \models ||\varphi||_a$ and, by induction, $T_w, C_{vx_1} \models \varphi$. For $x' = \varepsilon$ we already know $T_w, C_v \models \varphi$. For x_1IY and x_1Da we obtain $x_1 \equiv ax_1'$, $x_1'IY$, since C_{va} is a valid configuration. By $T_w, C_{vax_1'} \models \varphi$ we deduce $T_w, C_{vx_1} \models \varphi$. Altogether, this shows $T_w, C_v \models \Phi \mathcal{U}^Z \psi$.

This concludes the proof. □

We can now finally bring out the definition of the alternating Büchi automaton $\mathcal{A}_\varphi$ corresponding to a formula $\varphi \in \text{LTL}_\text{t}(\Sigma, I)$ similar as in Definition 7.2.6 on page 94:

Definition 7.2.18
Given a formula $\varphi \in \mathrm{LTL_t}(\Sigma, I)$*, the alternating Büchi automaton* $\mathcal{A}_\varphi$ *is the tuple* (Q, δ, q_0, F) *where*

- $Q = ecl(\varphi)$ *is the set of states.*
- $\delta(q, a) = ||q||_a$ *is the transition function.*
- $q_0 = \varphi$ *is the initial state.*
- $F = \{\neg\psi \mid \neg\psi \in ecl(\varphi)\}$ *is the set of accepting states.*

To show the overall correctness of our construction we must extend Theorem 7.2.7 on page 95 and its proof:

Theorem 7.2.19
Let φ *be a formula of* $\mathrm{LTL_t}(\Sigma, I)$ *and let its alternating Büchi automaton over the alphabet* Σ *be given as* $\mathcal{A}_\varphi$*. Then*

$$w \in \mathcal{L}(\mathcal{A}_\varphi) \text{ if and only if } T_w, \emptyset \models \varphi$$

for every $w \in \Sigma^\omega$*.*

Proof
For $w \in \Sigma^\omega$, we have to show that $\mathcal{A}_\varphi$ has an accepting run on w iff $T_w \models \varphi$. Note that every run has (at most) three kinds of paths

- finite paths ending in tt
- infinite paths on which from some point on every node is labeled by an *until*-formula or $\langle\_\rangle$-formula, or
- infinite paths on which from some point on every node is labeled by a negated *until*-formula or a negated $\langle\_\rangle$-formula.

Note that additionally to the cases considered in Theorem 7.2.7 on page 95, we now obtain infinite paths labeled with (negated) *until*-formulas. Thus, suppose we have an infinite path along which a (negated) *until*-formula is seen infinitely often. This can be accepted iff the underlying *until*-formula is preceded by a negation. This is captured by the acceptance condition for infinite paths given by the final states of the automaton. □

Let us again examine our procedure for the case that the underlying alphabet is fully-dependent so that traces and words can be identified. When we start in state $\{\varphi^\emptyset\}\mathcal{U}^\emptyset\psi$, reading an action a, it is independent of $Z = \emptyset$ so that the resulting state is obtained by considering the second case in Definition 7.2.16 on page 103. We get, using the notation in the mentioned definition:

$$\begin{aligned}
\Psi_1 &= ||\psi||_a \\
\Psi_2 &= \{||\varphi||_a^{\{a\}}\}\mathcal{U}^{\{a\}}||\psi||_a \\
\Psi' &= \Psi_1 \vee \Psi_2 \\
\Phi_1 &= ||\varphi||_a \\
\Phi_2 &= \{||\varphi||_a^{\{a\}}, \varphi^{\emptyset}\}\mathcal{U}^{\emptyset}\psi \\
\Phi' &= \Phi_1 \wedge \Phi_2
\end{aligned}$$

A simple analysis shows that states Ψ_2 and $||\varphi||_a^{\{a\}}$ in Φ_2 can be left out without changing the accepted language of the automaton. Thus, the correctness of our construction yields a correctness proof for the word case which was formulated as Theorem 6.2.6 on page 74. Moreover, we will point out in detail in Section 7.4 on page 110 that it is easy to adapt the rewrite operator to get a conservative extension:

Definition 7.2.20 (substitutes Definition 7.2.16)
Let

$$\Psi_1 = ||\psi||_a \quad \Psi_2 = \{||\varphi||_a^{Y\cup\{a\}} \mid \varphi^Y \in \Phi\}\mathcal{U}^{Z\cup\{a\}}||\psi||_a.$$

Let

$$\begin{aligned}
||\Phi||_a \;=\; &\{\; ||\varphi||_a^{Y\cup\{a\}} \mid \varphi^Y \in \Phi, \exists b \in \Sigma\; bI(Y \cup \{a\}\}\\
\cup\; &\{\; \varphi^Y \mid \varphi^Y \in \Phi, aIY\}
\end{aligned}$$

and

$$\Phi_1 = \bigwedge_{\varphi^Y \in \Phi} ||\varphi||_a \quad \Phi_2 = ||\Phi||_a \mathcal{U}^Z \psi$$

and $\Phi' = \Phi_1 \wedge \Phi_2$.
Let $Z' = Z \cup \{a\}$. *Then we define*

$$||\Phi\mathcal{U}^Z\psi||_a = \begin{cases} \Psi_1 \vee \Psi_2 & \text{if } aDZ, \exists b \in \Sigma\; bIZ' \\ \Psi_1 & \text{if } aDZ, \nexists b \in \Sigma\; bIZ' \\ \Psi_1 \vee \Psi_2 \vee \Phi' & \text{if } aIZ, \exists b \in \Sigma\; bIZ' \\ \Psi_1 \qquad\;\; \vee \Phi' & \text{if } aIZ, \nexists b \in \Sigma\; bIZ' \end{cases}$$

Complexity We do not intend to give a detailed analysis of the complexity of our decision procedure for whole LTL$_\mathrm{t}$. As mentioned before, [Wal98] has shown that the *until*-operator of LTL$_\mathrm{t}$ makes the decision problem for satisfiability of LTL$_\mathrm{t}$ formulas non-elementary. Thus, we are pleased having eliminated the exponential blow-up for negation present in the decision procedure presented in [GMP98b] and leaving an exponential blow-up only for the *until*-operator.

7.3 Linearity of our Construction

Now, we characterize $\mathrm{LTL}_{\mathrm{t}}(\Sigma, I)$ as equivalent to that subclass of alternating Büchi automata that we called trace-consistent linear alternating Büchi automata, and we start observing the linearity of the above construction.

Proposition 7.3.1 *Given* $\varphi \in \mathrm{LTL}(\Sigma, I)$*,* $\mathcal{A}_\varphi$ *is linear.*

Proof
We have to show that the transition graph of $\mathcal{A}_\varphi$ has only trivial cycles, i.e., for every path $q_1 \dots q_k$ with $k \geq 2$ and $q_1 = q_k$, we have that all q_i are labeled by q_1. Therefore we define a well-founded strict ordering relation[5] $\prec$ on the states of our automaton and show that $||\psi||_a$ yields a Boolean combination of strictly smaller states or ψ.

For a formula $\eta \in \mathrm{LTL}(\Sigma, I)$, $\prec \subseteq ecl(\eta) \times ecl(\eta)$ is inductively defined by

- $\varphi \prec \langle a\rangle\varphi$,
- $\langle a\rangle\varphi \prec \langle a\rangle\psi$ if $\varphi \prec \psi$,
- $\overline{\varphi} \prec \neg\psi$ if $\varphi \prec \psi$,
- $\psi_1^{Y_1} \prec \psi_2^{Y_2}$ if $\psi_1 \preceq \psi_2$ and $Y_1 \supseteq Y_2$ and one of the orderings is strict, i.e., $\psi_1 \prec \psi_2$ or $Y_1 \supsetneq Y_2$,
- $\bigvee\bigwedge \varphi_{ij} \prec \bigvee\bigwedge \psi_{ij}$ if $\{\varphi_{ij}\} \ll \{\psi_{ij}\}$ where $\ll$ is the (strict) (multi-)set ordering induced by $\prec$, i.e., $M_1 \ll M_2$ iff there exist a set X and an element $m \in M_2$ with $m' \prec m$ for all $m' \in X$ such that $M_1 = (M_2 - \{m\}) \cup X$. In other words, a set M_1 is smaller than M_2 if an element of M_2 is replaced by a set of smaller elements resulting in M_1 (cf. [BN98]).
- $\psi' \prec \Phi\mathcal{U}^Z\psi$ if $\psi' \prec \psi$,
- $\bigvee\bigwedge \varphi_{ij} \prec \Phi\mathcal{U}^Z\psi$ if $\{\varphi_{ij}^{\Sigma}\} \ll \Phi$,
- $\Phi_1\mathcal{U}^{Z_1}\psi_1 \prec \Phi_2\mathcal{U}^{Z_2}\psi_2$ if $\Phi_1 \underline{\ll} \Phi_2$ and $Z_1 \supseteq Z_2$ and $\psi_1 \preceq \psi_2$ and one of the orderings is strict, i.e., $\Phi_1 \ll \Phi_2$ or $Z_1 \supsetneq Z_2$ or $\psi_1 \prec \psi_2$, where $\underline{\ll}$ is the reflexive closure of $\ll$,

and contains its transitive closure. Here, $\preceq$ is the reflexive closure of $\prec$.

We easily verify that, given formulas $\varphi, \psi \in ecl(\eta)$, an action $a \in \Sigma$, and a minimal model Ψ of $||\psi||_a$ with $\varphi \in \Psi$, it holds $\varphi \preceq \psi$ and furthermore that for arbitrary

---

[5] i.e. a transitive and acyclic relation

$\varphi, \psi \in ecl(\eta)$, $\varphi \prec^{+} \psi$ implies $\varphi \neq \psi$. We conclude the linearity of our construction. $\square$

Let us bring out some important consequences of the last proposition:

Given an LTL formula φ over Mazurkiewicz traces, it is simple to construct a trace-consistent LTL formula ψ over words defining the same set of ω-words:

1. Construct $\mathcal{A}_\varphi$ according to Definition 7.2.18 on page 106.
2. For $\mathcal{A}_\varphi$, construct a formula $\psi \in \mathrm{LTL}_\mathrm{w}$ according to the proof of Theorem 6.3.2 on page 76.

As $\mathcal{A}_\varphi$ is a (trace-consistent) linear alternating Büchi automaton, Theorem 6.3.2 can be applied indeed to derive an LTL_w formula ψ accepting the same language.

To derive a second consequence, let us recall that LTL_w-definable trace-consistent languages coincide with first-order definable languages over traces:

Proposition 7.3.2 ([EM96]) *Let $\mathcal{L} \subseteq \Sigma^\omega$ and $I \subseteq \Sigma \times \Sigma$ be an independence relation. Then the following statements are equivalent:*

1) $\mathcal{L}$ is trace-consistent with respect to I and $\mathrm{LTL}(\Sigma)$-definable.

2) $\{T_w \mid w \in \mathcal{L}\}$ is $FO(\Sigma, I)$-definable.

So we can conclude that the languages definable by LTL_t formulas over Mazurkiewicz traces are FO-definable over Mazurkiewicz traces: For every LTL_t formula φ, we can obtain an LTL_w formula ψ defining the same language which is therefore an LTL_w definable trace-consistent ω-language. By Proposition 7.3.2, this is also FO-definable (over traces). Thus, we get a proof for one direction of the expressive-completeness result for LTL_t (Proposition 7.1.7 on page 89).

Let us further mention a practical consequence of our construction. Partial order reduction techniques work for LTL over Mazurkiewicz traces as usual: Given an LTL formula φ over Mazurkiewicz traces, consider its automaton $\mathcal{A}_\varphi$. It is a (trace-consistent) linear automaton over words so that we are in a well-known setting. Several powerful partial-order reduction techniques have been developed, which will have the same success here [Val91, Pel98] without any modifications. Hence, specifying with LTL over Mazurkiewicz traces promises—despite the bad worst-case runtime of its decision procedure—efficient verification tasks in practice.

Let us close this section with a result explaining that trace-consistent linear alternating Büchi automata correspond to LTL_t formulas, lifting a similar result (cf. Theorem 6.3.2 on page 76) to the setting of traces.

Theorem 7.3.3
Let $\mathcal{A}$ be a trace-consistent linear alternating Büchi automaton. There is a formula $\varphi \in \mathrm{LTL}(\Sigma, I)$ such that

$$\mathcal{L}(\mathcal{A}) = \mathrm{lin}(\mathcal{L}(\varphi))$$

Recall that $\mathrm{lin}(\mathcal{L}(\varphi))$ is a shorthand for $\{\mathrm{lin}(T) \mid T \in \mathcal{L}(\varphi)\}$.

Proof
According to Proposition 6.3.2 on page 76, given a trace-consistent linear alternating Büchi automaton $\mathcal{A}$, there is a formula $\psi_{\mathcal{A}} \in \mathrm{LTL}_{\mathrm{w}}(\Sigma)$ satisfying $\mathcal{L}(\psi_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$ where $\mathcal{L}(\psi_{\mathcal{A}})$ is likewise trace-consistent. Employing Proposition 7.3.2 on the page before, we obtain an FO formula defining the same language. By Theorem 7.1.6 on page 89, the existence of a formula $\varphi \in \mathrm{LTL}_{\mathrm{t}}(\Sigma, I)$ with $T_w, \emptyset \models \varphi$ if and only if $w \in \mathcal{L}(\mathcal{A})$ for every $w \in \Sigma^\omega$ immediately follows. □

A little sloppily, we can say that linear alternating Büchi automata coincide with LTL over words. When transferring LTL to traces, one gets or has to add trace consistency to this result.

7.4 $\mathrm{LTL}_{\mathrm{t}}^-$

As mentioned before, the complexity of deciding $\mathrm{LTL}_{\mathrm{t}}$ is non-elementary. It is the nesting of *until*-operators that gives such a high complexity. It is therefore natural to examine a restricted fragment of $\mathrm{LTL}_{\mathrm{t}}$. $\mathrm{LTL}_{\mathrm{t}}^-$ is the logic in which no *until*-operator is allowed but a $\Diamond$-operator. In other words, it is $\mathrm{LTL}_{\mathrm{HM}}$ enriched with a $\Diamond$-operator.

Walukiewicz pointed out in [Wal98] that deciding $\mathrm{LTL}_{\mathrm{t}}^-$ is Expspace-hard. Thus, the best we can hope is to get a decision procedure using exponential space with respect to the length of the formula. Employing the automata theoretic approach, we meet this goal by providing a decision procedure with double exponentially many states.

In [Wal98] a sketch of a decision procedure for $\mathrm{LTL}_{\mathrm{t}}^-$ using time that grows double exponential in the length of the underlying formula was given. The procedure adopted a construction presented in [AMP98]. However, some “difficulties” arose with this construction and it is no longer present in the extended version of the paper [Wal].

We apply our method to define a decision procedure for $\mathrm{LTL}_{\mathrm{t}}^-$. For the cases of the fully-dependent and the fully-independent alphabet, we show that the number of reachable states of our alternating Büchi automaton is bounded by a single exponential number so that the resulting decision procedure is indeed optimal. For an arbitrary dependence alphabet, however, it is not clear to us whether we obtain a single exponential number of reachable states.

Similarly as in Subsection 7.2.2, it will be convenient to assume that the syntax of $\mathrm{LTL}_{\mathrm{t}}^{-}$ is augmented with an extended $\Diamond$-operator $\Diamond^Z\varphi$. Formally, it has the following semantics:

- $T, C \models \Diamond^Z\varphi$ iff there exists a $C' \in \mathrm{conf}(T)$ with $C \subseteq C'$ such that $T, C' \models \varphi$ and $\lambda(C' - C)IZ$.

Hence, a trace satisfies the formula $\Diamond^Z\varphi$ iff there is a future configuration C' satisfying φ and all the actions from C to C' are independent of the actions in Z.

Note that $\Diamond\varphi$ can be identified with $\Diamond^{\emptyset}\varphi$ and we will not always make this distinction explicit. Furthermore, we can understand $\Diamond^Z\varphi$ as $\{\mathrm{tt}^{\emptyset}\}\mathcal{U}^Z\varphi$.

Following the scheme worked out in Subsection 7.2.1 on page 90, we now have to extend the definition of the subformula closure towards $\Diamond$-formulas:

Definition 7.4.1 (extends Definition 7.2.1)
Let η be a formula of $\mathrm{LTL}_{\mathrm{t}}^{-}$. We take ecl($\eta$) to be the least set that satisfies the items of Definition 7.2.1 on page 91 and furthermore:

- *For $\Diamond^Z\varphi \in ecl(\eta)$ the closure contains $ecl(\varphi)$, and, furthermore, for all $Z' \subseteq \Sigma$, all $\varphi' \in ecl(\varphi)$ the closure contains $\Diamond^{Z'}\varphi'$.*

Again, the definition reflects the idea that "everything" that might be obtained by rewriting is contained in the closure while assuring the closure to be finite, which is a direct consequence of Proposition 7.2.14 on page 102:

Proposition 7.4.2 *$ecl(\eta)$ is a finite set for each formula η of* $\mathrm{LTL}_{\mathrm{t}}^{-}$.

Formulas of the form $\Diamond\varphi$ or $\Diamond^Z\varphi$ are both called $\Diamond$-formulas.

The notion of a dual of formula carries over as expected:

Definition 7.4.3 (extends Definition 7.2.3)
The dual *of an (extended) formula is given inductively by the rules stated in Definition 7.2.3 on page 92 as well as:*

- $\overline{\varphi} = \neg(\varphi)$.
- $\overline{\Diamond^Z\varphi} = \neg(\Diamond^Z\varphi)$.

We now only need to specify the case of $||\Diamond^Z\varphi||_a$. Our approach is similar as in Section 7.2.2 on page 99. However, we have to be a little bit more careful to obtain a conservative extension of a decision procedure for $\mathrm{LTL}_{\mathrm{t}}^{-}$ over words which is defined in the expected manner.

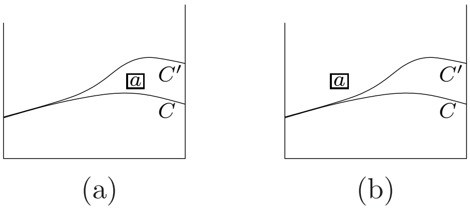


Figure 7.5: Configuration and actions for a $\Diamond$-formula

Again, consider some given trace T with $C, C' \in \mathrm{conf}(T)$ such that $C \subseteq C'$ and a formula $\Diamond^Z\varphi$. Suppose that C is the current configuration and that C' is a configuration in which φ holds (cf. Figure 7.5).

Let us now consider the next action a of a linearization. This might be within configuration C' or not, as depicted in Figure 7.5, Case (a) and Case (b), respectively. We can distinguish the following cases: If $C' = C$ then φ has to hold in the current configuration. If C' is a superset of C then there are events in $C' - C$. If one of these is labeled a, then we are in Case (a) and $\Diamond^Z\varphi$ holds in the configuration obtained by adding the a-event to C. If none of them is labeled a, then a has to be used for verifying φ still allowing the situation that there are actions which "turn" C into C'. Note that the latter is only possible if there are indeed actions left which are independent of Z and the current action a.

Consequently, we define the rewrite operator for a $\Diamond$-formula as follows:

Definition 7.4.4 (extends Definition 7.2.4)
Let $\Diamond^Z\varphi$ and $a \in \Sigma$, and $Z' = Z \cup \{a\}$. We define

$$||\Diamond^Z\varphi||_a = \begin{cases} ||\varphi||_a \;\vee\; \Diamond^Z\varphi \;\vee\; \Diamond^{Z'}||\varphi||_a & \text{if } aIZ\ ,\ \exists b\, bIZ' \\ ||\varphi||_a \;\vee\; \Diamond^Z\varphi & \text{if } aIZ\ ,\ \nexists b\, bIZ' \\ ||\varphi||_a \;\vee\; \Diamond^{Z'}||\varphi||_a & \text{if } aDZ,\ \exists b\, bIZ' \\ ||\varphi||_a & \text{if } aDZ,\ \nexists b\, bIZ' \end{cases}$$

We now state that our rewrite operator is locally correct:

Proposition 7.4.5 *Let η be any formula of $\mathrm{LTL}_t^-(\Sigma, I)$. Then for every $w \in \Sigma^\omega$ with $w \equiv vaw'$,*

$$T_w, C_v \models \eta \text{ if and only if } T_w, C_{va} \models ||\eta||_a$$

Proof
As in Proposition 7.2.5 on page 93, the proof proceeds by induction on the structure of the formula. The only remaining case is $\eta = \Diamond^Z\varphi$.

We can easily conclude the direction "$\Leftarrow$": Suppose aIZ. If $T, C_{va} \models ||\Diamond^{Z}\varphi||_a$ then $||\varphi||_a \vee \Diamond^{Z}\varphi \vee \Diamond^{Z'}||\varphi||_a$ holds in this configuration (uniting the first two cases in Definition 7.4.4 on the preceding page). This formula can be written as

$$||\varphi||_a \vee (||\mathrm{tt}||_a \wedge \{\mathrm{tt}^{\{a\}}\}\mathcal{U}^{Z}\varphi) \vee \{\mathrm{tt}^{\{a\}}\}\,\mathcal{U}^{Z'}||\varphi||_a$$

and thus as

$$||\{\mathrm{tt}\}\mathcal{U}^{Z}\varphi||_a$$

which consequently holds in C_{va}. By Proposition 7.2.17 on page 104, we get $T, C_v \models \{\mathrm{tt}\}\mathcal{U}^{Z}\varphi$ which can be read as

$$T, C_v \models \Diamond^{Z}\varphi$$

Similarly, we show this result for aDZ.

Although the direction "$\Rightarrow$" cannot directly be reduced to the proof of Proposition 7.2.17 on page 104, we can at least use the results obtained there: $||\Diamond^{Z}\varphi||_a$ can be read as $||\{\mathrm{tt}^{\emptyset}\}\mathcal{U}^{Z}\varphi||_a$ which, supposing the case aIZ, evolves to

$$||\varphi||_a \vee (||\mathrm{tt}||_a \wedge \{||\mathrm{tt}||_a^{\{a\}}, \mathrm{tt}^{\emptyset}\}\mathcal{U}^{Z}\varphi) \vee \{\mathrm{tt}^{\{a\}}\}\,\mathcal{U}^{Z'}||\varphi||_a$$

This formula can be reduced to the equivalent formula

$$||\varphi||_a \vee \Diamond^{Z}\varphi \vee \Diamond^{Z'}||\varphi||_a$$

By definition, $\Diamond^{Z'}\psi$ holds in a configuration C iff $C \models \psi$ or there is a future configuration $C' \supsetneqq C$ with $C' \models \psi$ and $\lambda(C' - C)IZ'$. The latter is only possible if there is an action independent of Z'. Thus, we can split the disjunction into two cases:

$$\begin{array}{ll} ||\varphi||_a \vee \Diamond^{Z}\varphi \vee \Diamond^{Z'}||\varphi||_a & \text{if } \exists b\, bIZ' \\ ||\varphi||_a \vee \Diamond^{Z}\varphi & \text{if } \nexists b\, bIZ' \end{array}$$

Using the same arguments, the case aDZ can be split up into two cases.

Altogether, we get

$$T_w, C_v \models \Diamond^{Z}\varphi \text{ if and only if } T_w, C_{va} \models ||\Diamond^{Z}\varphi||_a$$

□

Let us consider the rewrite operator for the fully-dependent alphabet. For $\Diamond^{\emptyset}\varphi$, we are in the case that regardless which action a is read, it is independent of $\emptyset$. However, there is no action b which is independent of a. Thus, $||\Diamond^{\emptyset}\varphi||_a = ||\varphi||_a \vee \Diamond^{\emptyset}\varphi$. The automaton to be defined will exactly behave as the decision procedure for the word case (cf. Chapter 6) assumed to be reduced to the straightforward definition of $\mathrm{LTL}_{\mathrm{w}}^{-}$ for words.

Retrospectively, it is clear, how to adapt the decision procedure for full LTL_t to get a conservative extension of the decision procedure for LTL_w, which is pointed out in Definition 7.2.20 on page 107.

In the previous constructions for LTL_HM and LTL_t, we identified the transition function of the automaton and the rewrite operator. Our goal is now to prove an exponential bound on the number of states of our automaton to be defined. To be able to achieve this result, we shall be more precise. We therefore define:

Definition 7.4.6
Given a formula $\eta \in \mathrm{LTL}_\mathrm{t}^-(\Sigma, I)$*, the alternating Büchi automaton* $\mathcal{A}_\eta$ *is the tuple* (Q, δ, q_0, F) *where*

- $Q = ecl(\eta)$ *is the set of states.*
- δ *is the transition function defined by*

$$
\begin{array}{lcl}
\delta(\mathrm{tt}, a) & = & \mathrm{tt} \\
\delta(\varphi \vee \psi, a) & = & \delta(\varphi, a) \vee \delta(\psi, a) \\
\delta(\neg\varphi, a) & = & \overline{\delta(\varphi, a)} \\
\delta(\langle b \rangle \varphi, a) & = & \begin{cases} \varphi & \mathit{if}\ a = b \\ \langle b \rangle ||\varphi||_a & \mathit{if}\ aIb \\ \mathrm{ff} & \mathit{if}\ aDb, a \neq b \end{cases} \\
\delta(\Diamond^Z \varphi, a) & = & \begin{cases} \delta(\varphi, a) \ \vee \ \Diamond^Z \varphi \ \vee \ \Diamond^{Z'} ||\varphi||_a & \mathit{if}\ aIZ\ ,\ \exists b\, bIZ' \\ \delta(\varphi, a) \ \vee \ \Diamond^Z \varphi & \mathit{if}\ aIZ\ ,\ \nexists b\, bIZ' \\ \delta(\varphi, a) \ \vee \ \Diamond^{Z'} ||\varphi||_a & \mathit{if}\ aDZ,\ \exists b\, bIZ' \\ \delta(\varphi, a) & \mathit{if}\ aDZ,\ \nexists b\, bIZ' \end{cases}
\end{array}
$$

  for $Z' = Z \cup \{a\}$.

- $q_0 = \eta$ *is the initial state.*
- $F = \{\neg\psi \mid \neg\psi \in ecl(\eta)\}$ *is the set of accepting states.*

Of course, there seems to be only a subtle difference between the rewrite operator and the transition function. However, for two formulas φ and ψ, their disjunction $\varphi \vee \psi$ counts as two states for our automaton but only as a single formula or *string* when obtained by the rewrite operator.

The key result we are looking for is an exponential bound for the number of reachable states for an automaton $\mathcal{A}_\varphi$ with respect to the length of φ.

Our procedure is built-up using two different ingredients, (string) rewriting and the transition function of the final automaton. One question is, how many “different objects” we can obtain for a given formula using the rewrite operator. Here, we consider a formula and consequently also the result obtained by applying the rewrite operator as a *string* and identify two strings iff they differ only with respect to commutativity and associativity. We call the strings obtained by (subsequently) rewriting a formula φ also *derivative* of φ, and we let φ to be among its derivatives. Let $\mathit{rew} : \mathrm{LTL}_\mathrm{t}^- \to 2^{\mathrm{LTL}_\mathrm{t}^-}$ be defined by

$$\mathit{rew}(\varphi) = \{\varphi' \mid \exists v \in \Sigma^* \; ||\varphi||_v = \varphi'\}$$

Thus, we are looking for a bound on *rew*.

The final question is of course: How many states does our automaton have with respect to the length of the formula φ? Given a formula $\varphi \in \mathrm{LTL}_\mathrm{t}^-$ and a state ψ of $\mathcal{A}_\varphi$, let $\mathit{reach}(\psi)$ denote the set of states reachable from ψ in $\mathcal{A}_\varphi$:

$$\mathit{reach}(\psi) = \{\psi' \mid \exists w \in \Sigma^* : \; \psi' \in \mathit{st}(\hat{\delta}(\psi, w))\}$$

To refrain from trivial considerations and to simplify the notations to come, we assume that $|\Sigma| \geq 2$. Suppose the number of $\mathit{rew}(\varphi)$ is bounded by $k^{p(|\varphi|)}$ for some constant $k \geq 2$ and some polynomial function p in the length of φ. Then the number of reachable states (from the initial state φ) is bounded by $2^{kp(|\varphi|)}$ when we require $k \geq |\Sigma| + 1$.

Proposition 7.4.7 *For $\varphi \in \mathrm{LTL}_\mathrm{t}^-(\Sigma, I)$ and $\mathit{rew}(\varphi) \leq 2^{kp(|\varphi|)}$ for $k \geq |\Sigma| + 1$ and a polynomial function p, we get*

$$\mathit{reach}(\varphi) \leq 2^{kp(|\varphi|)}$$

Proof

The proof proceeds by induction on the formula $\eta \in \mathrm{LTL}_\mathrm{t}^-$. For $\eta = \mathrm{tt}$, the claim is immediate. For $\eta = \varphi \vee \psi$, we get by induction $2^{kp(|\varphi|)}$ and $2^{kp(|\psi|)}$ many states reachable from φ and respectively ψ. Thus, we get less than $2^{kp(|\varphi|)+kp(|\psi|)}$ many states, which is less than $2^{kp(|\varphi|+|\psi|)}$. Negation does not change the number of states.

The interesting case is $\eta = \Diamond^Z \varphi$. For a single action a, $\Diamond^Z \varphi$ transforms at most into a disjunction of the states $\delta(\varphi, a)$, $\Diamond^Z \varphi$, and $\Diamond^{Z \cup \{a\}} ||\varphi||_a$. The situation is depicted in Figure 7.6 on the next page where the successor states of $\Diamond^\emptyset \varphi$ for a and b are shown. Thus, it suffices to sum up the number of states reachable from φ (colored light grey in Figure 7.6) and all states of the form $\Diamond Z' \varphi'$ for φ' a rewriting of φ and $Z' \subseteq \Sigma$ (painted grey in the figure). The latter are rewritings of φ which are decorated with some $\Diamond^{Z'}$ for some $Z' \subseteq \Sigma$.

Thus, we get $2^{kp(|\varphi|)} + 2^{|\Sigma|} 2^{kp(|\varphi|)}$. The first addend is obtained by induction while the second addend is due to the bound given for *rew*. The sum is less or equal to $2^{kp(|\varphi|)} + 2^{k-1} 2^{kp(|\varphi|)}$, which is $2^{kp(|\varphi|)}(1 + 2^{k-1})$. This is bounded by $2^{kp(|\varphi|)}(2^{k-1} + 2^{k-1})$, so that we get a bound of $2^{k(p(|\varphi|)+1)}$, which is less or equal to $2^{k(p(|\varphi|+1))}$.

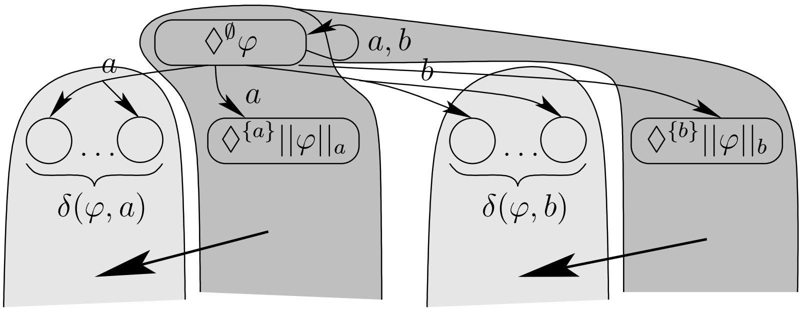


Figure 7.6: Reachable states for a $\Diamond$-formula

It seems that we have forgotten to count the states which are "obtained" within a state of the form $\Diamond^{Z'}\varphi'$. If the automaton chooses $\Diamond^{Z'}\varphi'$ as one successor state, then it might further proceed in state φ' (or one derivative hereof). However, for every $\Diamond^{Z'}\varphi'$ we already counted the states reachable from φ'. This is indicated by the thick arrows from the grey areas to the light grey areas in shown in Figure 7.6. Thus, we are done. ☐

The difficult part is to count the different number of derivatives for a given formula. It is easy to see that for formulas of the Hennessy-Milner fragment of LTL, *rew* is bounded by an exponential number. The interesting case is when we consider a $\Diamond$-formula. For technical reasons, we modify the definition of the rewrite operator slightly, without loosing any of the complexity results obtained so far:

$$\delta(\Diamond^Z\varphi, a) \quad = \quad \begin{cases} \delta(\varphi, a) \vee \Diamond^{Z'}||\varphi||_a \vee \varphi \vee \Diamond^Z\varphi & \text{if } aIZ \\ \delta(\varphi, a) \vee \Diamond^{Z'}||\varphi||_a & \text{if } aDZ \end{cases}$$

Thus, $\Diamond^Z\varphi$ rewrites for a single action a, which is assumed to independent of Z, to

$$||\varphi||_a \vee \Diamond^{Z\cup\{a\}}||\varphi||_a \vee \varphi \vee \Diamond^Z\varphi$$

which turns into

$$\begin{array}{l} ||\varphi||_{ab}\vee \\ ||\varphi||_{ab} \vee \Diamond^{Z\cup\{a,b\}}||\varphi||_{ab} \vee ||\varphi||_a \vee \Diamond^{Z\cup\{a\}}||\varphi||_a\vee \\ ||\varphi||_b\vee \\ ||\varphi||_b \vee \Diamond^{Z\cup\{b\}}||\varphi||_b \vee \varphi \vee \Diamond^Z\varphi \end{array}$$

assuming that b is independent of Z and a and that $||\_||\_$ is extended in the obvious way to cope with words instead of actions. Modulo associativity and commutativity this reads as

$$\bigvee_{u\in\{\varepsilon,a,b,ab\}} \Big(||\varphi||_u \vee \Diamond^{Z\cup\mathrm{alph}(u)}||\varphi||_u \Big) \qquad (*)$$

Obviously, $\Diamond^{Z}\varphi$ can be rewritten to some disjunction of derivatives of φ, some of which are decorated with $\Diamond^{Z'}$. This yields a bound exponential in the number of derivatives of φ. However, since the latter is supposed to be exponential as well, we would get a non-elementary upper bound, which is too much to be contented with.

Let us restrict to a fully-independent alphabet for the rest of this section. Consider formula $(*)$. Suppose $\varphi = \langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt}$. Then $(*)$ can be written as

$$\begin{array}{rl}
& \displaystyle\bigvee_{u\in\{\varepsilon,a,b,ab\}} \Big(||\langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt}||_u \vee \Diamond^{Z\cup\mathrm{alph}(u)}||\langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt}||_u \Big) \\
= & \_\_\langle a\rangle\langle b\rangle\mathrm{tt} \vee \Diamond^{Z\cup\{a,b\}}\_\_\langle a\rangle\langle b\rangle\mathrm{tt} \\
\vee & \_\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt} \vee \Diamond^{Z\cup\{a\}}\_\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt} \\
\vee & \langle a\rangle\_\langle a\rangle\langle b\rangle\mathrm{tt} \vee \Diamond^{Z\cup\{b\}}\langle a\rangle\_\langle a\rangle\langle b\rangle\mathrm{tt} \\
\vee & \langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt} \vee \Diamond^{Z}\langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt}
\end{array}$$

The formula $\Diamond^{Z}\langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt}$ rewrites for aa to

$$\begin{array}{rl}
& \_\langle b\rangle\_\langle b\rangle\mathrm{tt} \vee \Diamond^{Z\cup\{a\}}\_\langle b\rangle\_\langle b\rangle\mathrm{tt} \\
\vee & \_\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt} \vee \Diamond^{Z\cup\{a\}}\_\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt} \\
\vee & \langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt} \vee \Diamond^{Z}\langle a\rangle\langle b\rangle\langle a\rangle\langle b\rangle\mathrm{tt}
\end{array}$$

We can recognize a kind of monotonicity within this shape. It is immediate to write down each of the previously mentioned formulas given their second disjunct, which are $\Diamond^{Z\cup\{a,b\}}||\varphi||_{ab}$ and respectively $\Diamond^{Z\cup\{a\}}||\varphi||_{aa}$. It is a simple observation that this remark holds for arbitrary formulas of $\mathrm{LTL}_{\mathrm{HM}}$ and, using induction, also for arbitrary formulas of $\mathrm{LTL}_{\mathrm{t}}^{-}$ (over the fully-independent alphabet). Again by induction, we can bound the number of derivatives for φ by an exponential number, so that we obtain an exponential bound for the number of derivatives for $\Diamond^{Z}\varphi$:

Theorem 7.4.8

Let (Σ, I) be a fully-independent alphabet and $\varphi \in \mathrm{LTL}_{\mathrm{t}}^{-}$. Then

$$rew(\varphi) \leq 2^{kp(|\varphi|)}$$

for $k = |\Sigma| + 1$.

Let us set the formal framework that allows a formal proof of the previous theorem. We introduce constants e and f representing the empty word and f within formulas, respectively. Counting (certain) formulas built-up with these constants is much easier and we will get an exponential bound for the number of those. We show that every formula of $\mathrm{LTL}_\mathrm{t}^-$ can be represented by at least one such extended formula, so that we get the same bound for the number of formulas.

Consider the following rewrite operator $|\lceil \_ \rceil|_\_$:

$$
\begin{array}{lcl}
|\lceil \mathrm{tt} \rceil|_a & = & \mathrm{tt} \\
|\lceil \varphi \vee \psi \rceil|_a & = & |\lceil \varphi \rceil|_a \vee |\lceil \psi \rceil|_a \\
|\lceil \neg\varphi \rceil|_a & = & \overline{|\lceil \varphi \rceil|_a} \\
|\lceil \langle b \rangle \varphi \rceil|_a & = & \begin{cases} \langle \mathsf{e} \rangle |\lceil \varphi \rceil|_a & \text{if } b = \mathsf{e} \\ \langle \mathsf{f} \rangle \varphi & \text{if } b = \mathsf{f} \\ \langle \mathsf{e} \rangle \varphi & \text{if } a = b \\ \langle b \rangle |\lceil \varphi \rceil|_a & \text{if } aIb \\ \langle \mathsf{f} \rangle \varphi & \text{if } aDb, a \neq b \end{cases} \\
|\lceil \Diamond^Z \varphi \rceil|_a & = & |\lceil \varphi \rceil|_a \vee \Diamond^{Z'} |\lceil \varphi \rceil|_a \vee \varphi \vee \Diamond^Z \varphi
\end{array}
$$

for $Z' = Z \cup \{a\}$. The idea of the previous rewrite operator is that $\langle \_ \rangle$-operators with a formula are not deleted or transformed to ff but only turned into e or f, respectively. We can understand the objects created by this rewrite operator as $\mathrm{LTL}_\mathrm{t}^-$ formulas over the alphabet $\Sigma' = \Sigma \cup \{\mathsf{e}, \mathsf{f}\}$, which are rewritten by actions of Σ. For convenience, we do not place any independence/dependence restrictions on e and f.

Note that every $\mathrm{LTL}_\mathrm{t}^-$ formula over Σ is also one over Σ'. Furthermore, every formula over Σ' can be transformed into one over the alphabet Σ by mapping e to the empty word and f to ff, additionally "removing the rest of the formula". Of course, one formula over Σ is *represented* by many over Σ'. Importantly, given a formula φ and a word v, $||\varphi||_v$ is represented by $|\lceil \varphi \rceil|_v$.

Let us now find a bound for the rewrite operator $|\lceil \_ \rceil|_\_$, which is consequently also a bound for the rewrite operator *rew*. The benefit of the operator $|\lceil \_ \rceil|_\_$ is that it does not modify the length of the original formula except when it is of the form $\Diamond^Z \varphi$. A part of a formula of the form $\langle a \rangle$ stays $\langle a \rangle$ or is turned into $\langle \mathsf{e} \rangle$ or $\langle \mathsf{f} \rangle$. Thus, for $\langle \_ \rangle$-formulas, we get a bound of $3^{|\varphi|}$. What happens with $\Diamond^Z \varphi$? For the case of the fully-independent alphabet, this is easy to see:

Consider the rewrite operator $|\lceil \_ \rceil|'_\_$ that is defined as $|\lceil \_ \rceil|_\_$, except for the case

$$|\lceil \Diamond^Z \varphi \rceil|'_a = \Diamond^{Z \cup \{a\}} |\lceil \varphi \rceil|'_a$$

For the fully-independent alphabet, there will never occur the symbol f, so that $|\lceil \_ \rceil|_\_$ and $|\lceil \_ \rceil|'_\_$ yield sets of the same cardinality when applied to a formula and to a set of finite words. For $|\lceil \_ \rceil|'_\_$, it is immediate that the number of reachable states is bounded by a single exponential number. Thus, the same result holds for $|\lceil \_ \rceil|_\_$ and consequently for $||\_||_\_$, which shows Theorem 7.4.8 on the preceding page.

Chapter 8

LTL over Foata Configuration Graphs

In this chapter, we introduce a different kind of linear temporal logic which can be used for specifying properties of *synchronized* systems. Important examples among these systems are hardware circuits which are build up by separate entities working together in parallel but which are synchronized by a global clock.

We exhibit the notion of a *distributed synchronous transition system* (DSTS) as a model for these hardware designs. DSTSs can be equipped naturally with a *Foata configuration graph*-based semantics, which provides a link between these systems and the framework of Mazurkiewicz traces.

We define *Foata linear temporal logic* ($\mathrm{LTL_f}$) which is a temporal logic with a flavor of linear temporal logic adapted for specifying properties of the behavior of DSTSs. More specifically, $\mathrm{LTL_f}$ formulas are interpreted over Foata configuration graphs of traces.

We give a decision procedure for satisfiability of $\mathrm{LTL_f}$ formulas as well as a model checking procedure, both based on alternating Büchi automata. It turns out that these procedures are as efficient as for $\mathrm{LTL_w}$ (for words) viz they are exponential in the length of the formula and linear in the size of the system and are essentially optimal. The model checking procedure employs an optimization which is similar to a technique known as *partial order reduction* [Pel98]. However, instead of defining the interleaving product of the sequential processes and then trying to omit states with no influence to the result of the model checking procedure, we are able to define smaller systems directly due to our underlying model. This relieves us of the difficulties involved with computing so-called ample sets [Val91].

To simplify the task of defining DSTSs, we introduce a simple calculus, which we call *synchronous process systems* (SPS) and which is inspired by Milner's CCS [Mil89] but is adapted towards the special nature of our underlying systems.

In Section 8.1, we present some examples for the kind of systems we want to support. We proceed with introducing the key structure providing a link for the study of the underlying systems in terms of Mazurkiewicz traces, which is a *Foata configuration graph.* Furthermore, we describe their relation to words in *Foata normal form.* We carry on by defining distributed transition systems (Section 8.3) and a calculus for synchronous process systems allowing them to be presented in a compact way (Section 8.4). In Sections 8.5 – 8.7, we define LTL_f and develop a decision procedure for satisfiability as well as for model checking. We conclude this chapter with a larger example in that we design a two-bit counter.

8.1 Motivation

Many digital circuits, especially embedded controllers, can be modeled as transition systems with respect to their logical behavior. The controller is in one of finitely many states and executes one of its instructions which we call *actions*, as usual. The action modifies the current state transforming it into a new one. Usually, the executions are synchronized by a global clock or oscillator. Every time *tick*, an action takes place. Actions lasting for more than one tick can be modeled as a sequence of single-tick actions. Several circuits or controllers for different tasks are combined on a switching board. The global clock synchronizes the execution.

Let us consider Figure 8.1 on the next page which shows a sample layout of a simple so-called *embedded system.* Embedded systems are (the heart of) electrical devices such as mass storage systems, ISDN cards, video adapters, laser printers, etc. The shown setup is taken from "The PowerPC$^{\text{TM}}$ 601 User's Manual" [Mot93], which explains how to use the *central processing unit* (*CPU*) called *PowerPC*$^{\text{TM}}$ and developed by IBM$^{®}$ and Motorola$^{®}$ to construct these kinds of systems.

Abstracting from the details to obtain main ingredients of this example, we can think of Circuit 1 to Circuit 3 to be PowerPCs$^{\text{TM}}$ and let Circuit 4 be a memory controller. These controllers are connected to a *bus.* The general idea is that, whenever a CPU wants to obtain the content of a memory location, it puts a corresponding request on the bus. Every circuit connected to the bus can see this request. However, only the memory controller is responsible for this request so that it will be the only one answering the request by putting the demanded memory content on the bus.

If several circuits write to the bus at the same time, the information gets corrupted, of course. Thus, an *arbiter* is used to coordinate the access to the bus. Every circuit has to ask the arbiter, which grants the access to the bus. This can be realized as follows: Every circuit is connected by two wires to the arbiter, one is employed for requesting the bus (r_i), the second one for granting it (g_i). The circuits and the arbiter communicate via common actions. A request of Circuit 1 recognized by the arbiter can be modeled by the common action r_1. If the arbiter is not in the state

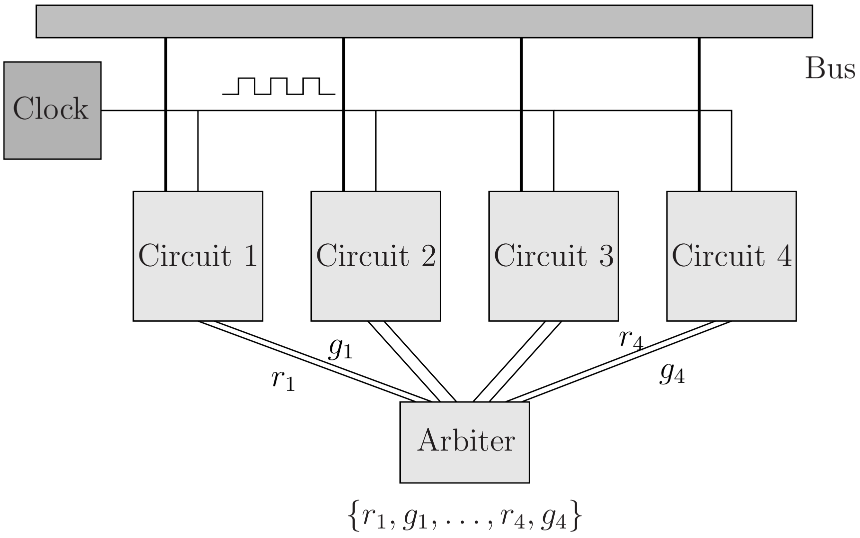


Figure 8.1: Synchronized digital circuits

for receiving the request, Circuit 1 suspends.

The overall setting is synchronized or *clocked* by a global clock. In every tick, the other circuits may execute actions *independently.*

Note that this setting is only adequate for small electronic devices. Larger systems, like main boards of todays personal computers are build up by several embedded systems. While each of them is synchronized by a clock, they may run on different speeds or they have their own clock. Then, for example, the CPU works with a higher clock rate than the memory controller. For these systems, an asynchronous communication scheme as the one underlying CCS and the view on traces taken in the previous chapters is more realistic and hence should be preferred.

Verifying embedded systems, however, the synchronous approach is closer to the realization of the system and should therefore be used. In effect, for model checking linear time specifications, one might even fail to prove a property of a system when its clock is ignored and the components of the system are assumed to run asynchronously, although the underlying system fulfills the requirement.

Let us sum up the main ingredients of our system: We have several devices which are connected by wires. The execution of the devices is clocked and might change an internal state of the system. As the devices are capable of executing actions independently, we can observe a set of actions for every time tick. The devices can communicate via wires to rule out which device is allowed to use a common resource. The communication can be understood as an agreement on the same (communication) action of the two devices communicating (circuit/arbiter).

We employ *distributed transition systems* (DTS), which are a well-known model for distributed systems (cf. [Zie87, TH98]) for describing such a given setup. However, DTSs are usually considered with an asynchronous model of execution. We introduce *distributed synchronous transition systems* (DSTS), which are distributed transition systems with a global clock synchronizing the execution of actions. They can be understood as a model for the parallel composition of hardware circuits as described above.

Distributed synchronous transition systems can also be interpreted as a model for *Petri nets* with a *maximal step semantics*. We only want to provide the idea on an intuitive level and refer for basic notions on Petri nets to [Rei86] and to [Muk92] for an analysis of Petri nets with respect to (not necessarily maximal) step semantics. Figure 8.2 recalls the Petri net (place-transition-net) we already have seen in Figure 3.6 on page 24. The maximal step semantics is obtained by the rule that all transitions which are capable of firing simultaneously, fire simultaneously. In other words, in every step, we can observe a set of transitions which have fired, and, this set is required to be maximal.

Let us come back to the presented Petri net. Transitions a and e are ready to fire since each of the places s_1 and t_1 contain a token. As they do not rely on the same place, they can fire simultaneously. Furthermore, they are the only transitions which can fire. Thus, the first maximal step is $\{a, e\}$ so that each of s_2 and t_2 are filled with a token. Now, b as well as f can fire. But, in contrast to the previous situation, this cannot happen simultaneously because both transitions require a token to be present in r. Thus, we get two different possible steps, either $\{b\}$ or $\{f\}$. Hence, a possible execution sequence with respect to maximal step semantics for the given Petri net is, for example, the sequence

$$\{a, e\}\{b\}\{c\}\{d, f\}\{g\}\{h\}$$

where each set consists of the actions occurring concurrently. When we speak of a step in the following, we always assume it to be maximal.

In the literature, the simultaneous execution of two independent actions a and b is usually modeled by interleaving a and b, i.e., first a and then b as well as b and then a [Mil89]. In this way, concurrency is reduced to sequences and non-deterministic choice. Although we introduced Mazurkiewicz traces as a model to avoid this interleaving, we adopted this view when defining our decision procedures for $\mathrm{LTL}_{\mathrm{t}}$ in the previous chapters while passing from traces to linearizations of traces. The view taken here is somehow *dual*. If two actions a and b can occur concurrently, then we require them to occur concurrently and abstract from interleaving.

Synchronous systems have been studied by several authors. Milner defined a variant of his (asynchronous) *Calculus of Communicating Systems* (*CCS*, [Mil80, Mil89]) for synchronous systems (*SCCS*, [Mil83], see also [Bru97]). Lustre [CPHP87] is a programming language for synchronous systems. Usually, these contributions concentrate on the design of the underlying systems. The problem of verification is

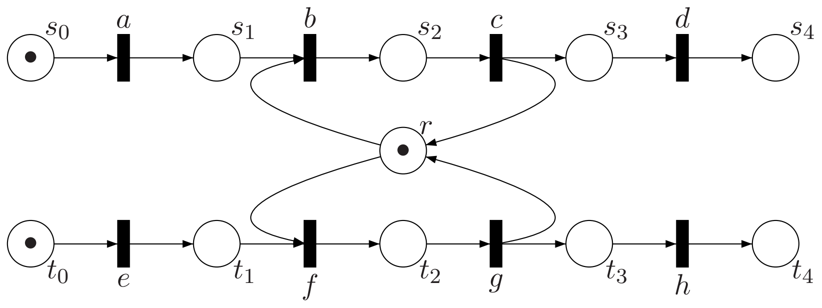


Figure 8.2: A Petri net

tackled by the notion of *bisimilarity* [Mil89] or by *theorem proving* [BCPVD99]. Simple model-checking-based verification techniques are lacking.

We present a simple model for synchronous hardware systems together with an implementation driven definition of a satisfiability and a model checking algorithm. Confer [Kro99] for an introduction to formal hardware verification.

8.2 Foata Configurations

In this section, we derive the basic structure underlying distributed synchronous transition systems as well as Foata linear temporal logic: *Foata configurations*. If not stated otherwise, we argue with respect to a fixed dependence alphabet (Σ, D) in this chapter.

In the previous section, we suggested to consider independent actions which can be executed concurrently as a single *step* of a system. Since within a Mazurkiewicz trace independent actions are not ordered, it is easy to identify them. Consider the trace shown in Figure 8.3(a) on the next page (over the meanwhile well-known dependence alphabet shown in Figure 3.1 on page 16). The first step in every linearization will consist of the event e_1 labeled by a. It is the only minimal event which must have occurred in all non-empty configurations. In Figure 8.3(b) on the next page, we identify the first step by drawing a line above the event e_1. If the event e_1 is removed from the trace, the minimal events are e_2 and e_3 labeled by b respectively c. These events form the next step. We proceed in the same manner to identify all steps of the given trace.

Observe that the action label of every event e' of a subsequent step is dependent on some action label of an event of the current step because otherwise e' would already be minimal with respect to the current step. It is clear that the steps of a trace are a linearly ordered and that the union of a step together with all smaller steps yields a configuration.

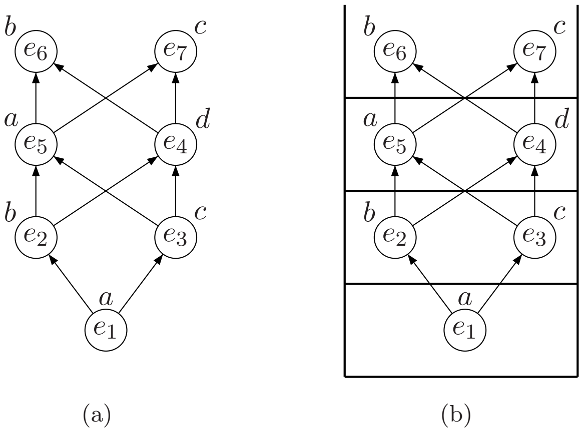


Figure 8.3: A trace and one of its partition into steps

We have now set out the scene to make our ideas more precise:

Definition 8.2.1
Given a trace $T = (E, \leq, \lambda)$*, let* $\min(E') = \{e \in E' \mid e \textit{ is minimal with respect to } \leq\}$ *denote the set of minimal elements of a partially ordered set. Let* FC *be the smallest set such that*

- $\emptyset \in FC$ *and*
- *for every* $C \in FC$ *also* $\downarrow\min(E \setminus C) \in FC$*.*

The Foata configuration graph *of a trace* T *is the subgraph* $(FC, \subseteq)$ *of* $\mathcal{CG}(T)$ *and is denoted by* $\mathcal{FCG}(T)$*. The set of* Foata configurations *of the trace are the elements of* FC *and are denoted by* $\mathrm{fconf}(T)$*.*

The Foata configurations are linearly ordered. More precisely, $\mathcal{FCG}(T) = (FC, \subseteq)$ is a linearly ordered set. Let $\subset\!\!\cdot$ be the covering relation of $\subseteq$, i.e., $\subset\!\!\cdot = \subseteq - (\subseteq \circ \subseteq)$. For two Foata configurations C and C' with $C \subset\!\!\cdot\, C'$, we call the events of $C' - C$ a *step* of T.

Figure 8.4(a) on the facing page shows the configuration graph of the trace shown in Figure 8.3(a) while Figure 8.4(b) shows its Foata configuration graph, both representing the configuration in the slightly sloppy notation introduced in Chapter 3. Comparing Figure 8.4(b) on the facing page with Figure 8.3(b), we can see the correspondence of Foata configurations and steps.

As we will see in Section 8.5 on page 134, the formulas of the Foata temporal logic are interpreted with respect to Foata configurations of traces.

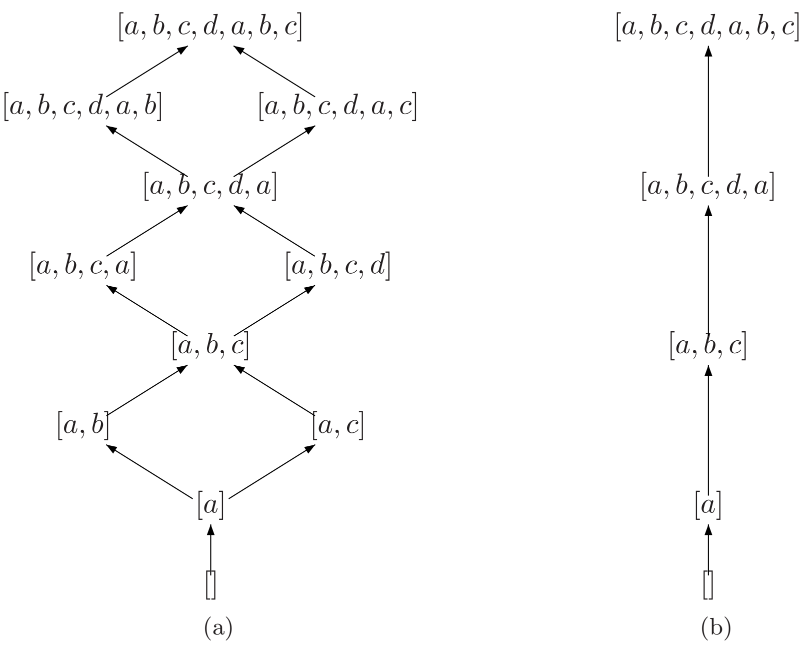


Figure 8.4: The configuration graph of the trace of Figure 8.3 (a)

We learned in Section 3.4 on page 28 that a linearization of a trace $(E, \leq, \lambda)$ is a linearization of the partial order, i.e., it is a labeled linear order $(E, \leq', \lambda)$ such that $\leq \subseteq \leq'$. To find a link with automata theory for words, we again consider the concept of linearizations and identify so-called *Foata linearizations* which are linearizations that conform with the order of steps:

Definition 8.2.2
A Foata linearization *of a trace* $(E, \leq, \lambda)$ *is a linearization* $(E, \leq', \lambda)$ *which can be written as a product of disjoint finite traces*

$$(E, \leq', \lambda) = \prod_{i=1,\ldots,\infty} (E_i, \leq'_i, \lambda_i)$$

such that for every $i \geq 1$,

- E_i *is a set of pairwise independent actions, i.e., for every* $e \in E_i$, *we have* $\lambda_i(e) I \lambda_i(E_i \setminus \{e\})$ *and*
- *for every* $e \in E_{i+1}$, *there is an* $e' \in E_i$ *such that* $\lambda(e) D \lambda(e')$.

Note that the product of finite traces is defined canonically by uniting the set of events of both traces, uniting the labeling functions (considered as a graph), and uniting the ordering relations of the events plus ordering dependent events:[1]

$$(E, \leq, \lambda)(E', \leq', \lambda') = (E \cup E', \leq'', \lambda \cup \lambda')$$

where

$$\leq'' = \leq \cup \leq' \cup \{(e, e') \in E \times E' \mid (\lambda(e), \lambda'(e')) \in D\}$$

The first item of Definition 8.2.2 on the preceding page guarantees that only independent events are present in each step and the second item checks that every event of each step cannot occur in a previous step.

In the same manner as in Section 3.4, a Foata linearization corresponds to a word in Foata normal form which is again called a *Foata linearization.* It is defined similarly having the idea of steps in mind.

Definition 8.2.3 ([DM96])
A word $w \in \Sigma^\omega$ is in Foata normal form iff

1. $w = u_1 u_2 \dots$ *for* $u_i \in \Sigma^*$,
2. *for each* $i \geq 1$, *the word* u_i *is a product of pairwise independent actions, and*
3. *for each* $i \geq 1$ *and for each letter* a *of* u_{i+1}, *there exists a letter* b *in* u_i *which is dependent on* a.

The words u_i are again called *steps.* It is easy to see that for $w = u_1 u_2 \dots$ in Foata normal form and for every i, the suffix $u_i u_{i+1} \dots$ is in Foata normal form. The steps correspond to the top actions of every configuration in the Foata configuration graph of a trace. For example, a Foata linearization of the trace shown in Figure 8.3(a) is $(a)(bc)(ad)(bc)$ where the steps are accentuated by parentheses.

Note that the Foata normal form as defined here is unique up to permutation of independent actions within a step. Given a linear oder $\prec$ for the actions of Σ and requiring each step to be minimal with respect to the lexicographic order derived from $\prec$, we obtain a unique Foata normal form for every trace.

Let $\mathrm{I}(\Sigma)$ denote the set of sets of pairwise independent actions of Σ. As every step consists only of pairwise independent actions, it can be considered as an element of $\mathrm{I}(\Sigma)$. Thus, an ω-word in Foata normal form can also be identified as an element of $\mathrm{I}(\Sigma)^\omega$. Since Foata configuration graphs and words in Foata normal form are the basic objects to be studied in this chapter, we could debate whether we are just in the case of words over the alphabet $\mathrm{I}(\Sigma)$ and could take over notions of linear temporal logic and its decision procedures directly.

---

[1] Note that we silently assume that $E \cap E' = \emptyset$ within this definition.

However, not every word of I(Σ) is in Foata normal form. Suppose, for example, that the actions b and c are independent. $(b)(bc)$ can be considered as a word of I(Σ) but its Foata normal form (as a word over Σ) would be $(bc)(b)$. Thus, words in Foata normal form can only be considered as *special* words over the alphabet I(Σ). This implies that the existing decision procedures cannot be taken over directly. These have to be modified to reflect the special structure of words in Foata normal form.

We follow a different, more direct approach. We stay with the alphabet Σ and provide decision procedures for satisfiability and model checking incorporating words in Foata normal form over Σ.

8.3 Distributed Synchronous Transition Systems

We now introduce a formal model which is useful for describing our kind of underlying concurrent systems in form of transition systems: *distributed synchronous transition systems*. It is based on Zielonka's asynchronous automata (without final states, [Zie87]), or the notion of distributed transition systems (described for example in [TH98]). Our presentation is inspired by [PP95].

While the definition of the components of a distributed (synchronous) transition system is as usual, the definition of its *execution*[2] is modified to reflect the idea of a global synchronizing clock. Strictly speaking, we only define *distributed transition systems* (DTS) as well as their *synchronous* and *asynchronous* executions. However, to identify the respective context, we speak of either *synchronous* or *asynchronous* DTSs. We fix a distributed alphabet $\tilde{\Sigma} = (\Sigma_1, \dots, \Sigma_n)$ with $\mathit{Proc}(\tilde{\Sigma}) = \{1, \dots, n\}$ for this chapter (cf. Chapter 3.1 on page 15).

Definition 8.3.1
A distributed transition system (DTS) *over a distributed alphabet* $\tilde{\Sigma}$ *is a tuple* $\mathcal{A} = (Q_1, \dots, Q_n, \longrightarrow, \mathcal{I})$ *with the following:*

- *Each* Q_i *is a finite nonempty set of* local states *of the* i*-th component.*
- *Let* $\bar{Q} = \prod_{i \in \mathit{Proc}} Q_i$ *be the set of* global states *and* $\mathcal{I} \subseteq \bar{Q}$ *be the set of* initial states*.*
- *Let* $\mathit{States} = \prod_{i \in \mathit{Proc}} (Q_i \cup \{-\})$*. The dummy* $-$ *is used as a placeholder in components which have no significance for the transition:* $\longrightarrow \subseteq \mathit{States} \times \Sigma \times \mathit{States}$ *is a* transition relation *satisfying the following condition:*

$$\text{if } (\bar{q}, a, \bar{q}') \in \longrightarrow \ \text{ then } \bar{q}[i] = \bar{q}'[i] = - \text{ for } i \in \mathit{Proc} \backslash pr(a)$$

---

[2] In the framework of transition systems, we prefer the notion of *executions* rather than the one of *runs*, because we associate with *runs* a device gaining input. Technically, it is possible to identify transition systems with certain automata and to speak of runs instead of executions.

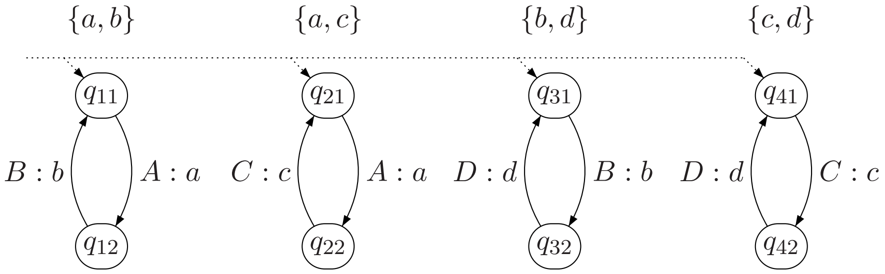


Figure 8.5: A distributed transition system

where $\bar{q}[i] \in Q_i \cup \{-\}$ *denotes the* i*-th component of* $\bar{q} \in$ *States.*

Of course, we often write $\bar{q} \xrightarrow{a} \bar{q}'$ instead of $(\bar{q}, a, \bar{q}') \in\longrightarrow$. The dummy $-$ in the definition of the transition relation $\longrightarrow$ is used for denoting components of a global state which are not affected by the transition. Given a global state $\bar{q} = (q_1, \dots, q_n) \in \bar{Q}$ and $M \subseteq \{1, \dots, n\}$, we denote by $\bar{q}|_M$ the element $(q'_1, \dots, q'_n) \in$ *States* such that $q_i = q'_i$ for $i \in M$ and $q'_i = -$ else.

As an example, consider the distributed transition system shown in Figure 8.5. The underlying distributed alphabet is $\tilde{\Sigma} = (\{a,b\}, \{a,c\}, \{b,d\}, \{c,d\})$. The transition system consists of four components $Q_1, \dots, Q_4$ and each component Q_i comprises two states q_{i1} and q_{i2}. Action a participates exactly in components Q_1 and Q_2 so that every transition in which a is involved can only modify the components 1 and 2. A transition can be visualized by an edge showing the movement in each component labeled with an action plus a sequence of *names* (taken from some domain of names) to identify which moves must be taken together. An action labeled edge is called a *local transition* in the following so that, using this notation, we can say that a transition can be represented by a set of local transitions plus a sequence of names grouping together local transitions. For example, the transition $(q_{11}, q_{21}, -, -) \xrightarrow{a} (q_{12}, q_{22}, -, -)$ can be represented by two a-labeled arrows (q_{11}, q_{12}) and respectively (q_{21}, q_{22}) together with the name A which is written before the action labels, as shown in the figure. Note that for representing the two transitions $(q_{11}, q_{21}, -, -) \xrightarrow{a} (q_{12}, q_{21}, -, -)$ and $(q_{11}, q_{21}, -, -) \xrightarrow{a} (q_{11}, q_{22}, -, -)$ (which are not present in the automaton of our example), it would be necessary to use two different names A_1 and A_2 preceding the labels of the edges of each component. Often, the transitions are of the form that a single name can be used for every action, as in our example. In these cases, the name grouping together local transitions can be omitted which is done in further examples. Note that in our example, we could indeed leave out A, B, C and D. Initial states can be represented using an edge linking together the local states, as shown in Figure 8.5 by the dotted line.

Let us now define the "execution" of a DTS. Usually, the idea is that the system starts in one initial state and proceeds with a sequence of transitions yielding a

$$\begin{tikzcd}
(q_{11}, q_{21}, q_{31}, q_{41}) \arrow[d, "\{a\}"] \\
(q_{12}, q_{22}, q_{31}, q_{41}) \arrow[d, "\{b, c\}"] \\
(q_{11}, q_{21}, q_{32}, q_{42}) \arrow[d, "\{a, d\}"] \\
(q_{12}, q_{22}, q_{31}, q_{41})
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
(q_{11}, q_{21}, q_{31}, q_{41}) \arrow[d, "a"] \\
(q_{12}, q_{22}, q_{31}, q_{41}) \arrow[d, "b"] \\
(q_{11}, q_{22}, q_{32}, q_{41}) \arrow[d, "c"] \\
(q_{11}, q_{21}, q_{32}, q_{42}) \arrow[d, "d"] \\
(q_{11}, q_{21}, q_{31}, q_{41}) \arrow[d, "a"] \\
(q_{12}, q_{22}, q_{31}, q_{41})
\end{tikzcd}$$

(b)

Figure 8.6: Synchronous vs. asynchronous execution

sequence of action labels. To capture our intuition that the underlying system should commence "as many transitions" in parallel as possible, we introduce the notion of a *synchronous execution*:

Definition 8.3.2
A synchronous execution ρ *of a* DTS *is an infinite sequence* $\bar{q}_1 A_1 \bar{q}_2 \ldots$ *of global states and sets of pairwise independent actions which satisfies the following conditions:*

- $\bar{q}_1 \in \mathcal{I}$*, i.e.,* $\bar{q}_1$ *is an initial state.*
- *For* $j \geq 1$ *and all* $a \in A_j$, $(\bar{q}_j|_{pr(a)}, a, \bar{q}_{j+1}|_{pr(a)}) \in \longrightarrow$ *and* $\bar{q}_j|_P = \bar{q}_{j+1}|_P$ *for* $P = Proc \backslash \bigcup_{a \in A_j} pr(a)$*. Hence, a transition is the "parallel" execution of concurrent actions according to the transition rules.*
- *Further,* A_j *must be maximal in the following sense: For every* $j \geq 1$ *and for all* $A' \in \mathrm{I}(\Sigma)$ *with* $A' \supseteq A_j$ *such that for all* $a \in A'$, $(\bar{q}_j|_{pr(a)}, a, \bar{q}_{j+1}|_{pr(a)}) \in \longrightarrow$ *we have* $A' = A_j$*. This ensures that all components being able to do a transition participate in the execution step.*

Abusing notation, we call a DTS also a *distributed synchronous transition system* if we consider its synchronous executions. Figure 8.6(a) shows a part of one of the executions of the DTS shown in Figure 8.5 on the preceding page. The crucial point in the execution is that the actions b and c occur synchronously.

For a distributed transition system, we also define the notion of an asynchronous execution which is obtained by interleaving transitions.

Definition 8.3.3
An asynchronous execution ρ *of a* DTS *is an infinite sequence* $\bar{q}_1 a_1 \bar{q}_2 \ldots$ *of global states* $\bar{q}_j$ *and actions* a_j *which satisfies the following conditions:*

- $\bar{q}_1 \in \mathcal{I}$*, i.e.,* $\bar{q}_1$ *is an initial state.*
- *For every* $j \geq 1$*, we have* $(\bar{q}_j|_{pr(a_j)}, a_j, \bar{q}_{j+1}|_{pr(a_j)}) \in \longrightarrow$ *and* $\bar{q}_j|_{Proc \setminus pr(a_j)} = \bar{q}_{j+1}|_{Proc \setminus pr(a_j)}$.

In the same manner as before, we call a distributed transition system also a *distributed asynchronous transition system* when considering its asynchronous executions. For the distributed transition system presented in Figure 8.5 on page 128, a part of an asynchronous execution is shown in Figure 8.6(b) on the page before.

A synchronous execution can be related to a Mazurkiewicz trace and and a Foata configuration graph as expected: Consider an execution sequence $\bar{q}_1 A_1 \bar{q}_2 \ldots$. Every a in every A_i can be identified with a unique event e_{ia} labeled by a labeling function λ with a. We let $\leq$ be the least transitive relation satisfying $e_{ia} \leq e_{jb}$ iff $i \leq j$ and aDb where D is the dependence relation given by $\tilde{\Sigma}$ (cf. Chapter 3.1 on page 15). It is easy to see that $(\{e_{ia} | i \geq 1, a \in A_i\}, \leq, \lambda)$ is a Mazurkiewicz trace with steps $A_1, A_2, \ldots$. The Foata configuration graph of this trace is now of the form

$$C_{A_0} \longrightarrow C_{A_1} \longrightarrow C_{A_2} \longrightarrow \ldots$$

where $C_{A_0} = \emptyset$ and C_{A_i} consists of the events e_{ia} for $a \in A_i$ and of the ones in $C_{A_{i-1}}$ if $i \geq 1$. Thus, we can understand executions of DSTSs as Foata configuration graphs. This allows us to analyze DSTSs by considering the Foata configuration graphs corresponding to executions.

Observe that an asynchronous execution $\bar{q}_1 a_1 \bar{q}_2 a_2 \bar{q}_3 \ldots$ can be identified with the word $a_1 a_2 \ldots$ which can be understood as a Mazurkiewicz trace (cf. Section 3.4 on page 28).

We have seen in the previous section that Foata configuration graphs can be identified with words in Foata normal form. Thus, every execution can be considered to be an infinite word which is furthermore in Foata normal form. Thus, we have a link between executions and words in Foata normal form.

Every synchronous execution $\bar{q}_1 A_1 \bar{q}_2 A_2 \bar{q}_3 \ldots$ can be translated into an asynchronous execution by interleaving each step of actions

$$\bar{q}_1\, a_{1,1}\, \bar{q}_{1,1}\, \cdots\, \bar{q}_{1,k_1-1}\, a_{1,k_1}\, \bar{q}_2\, a_{2,1}\, \bar{q}_{2,1}\, \cdots\, \bar{q}_{2,k_2-1}\, a_{2,k_2}\, \bar{q}_3\, \cdots$$

for $a_{i,j} \in A_i$, $k_i = |A_i|$ and suitable $\bar{q}_{i,l}$. We call the latter the *interleaved synchronous execution* of the synchronous one.

However, not every asynchronous execution can be obtained by interleaving a synchronous one. Note that, obviously, the sequence $a_{1,1} \ldots a_{1,k_1} a_{2,1} \ldots a_{2,k_2} \ldots$ is a

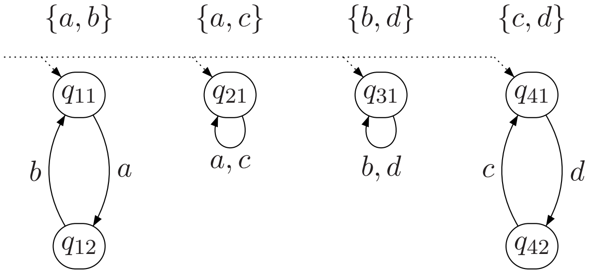


Figure 8.7: Asynchronous executions and synchronous executions differ

word in Foata normal form. Thus, it remains to show that there is an asynchronous execution with a sequence of actions which is not Foata normal form:

Theorem 8.3.4
There is a distributed transition system for which the class of interleaved synchronous executions is strictly contained in the class of asynchronous executions.

Proof
Consider the system depicted in Figure 8.7. The steps of every synchronous execution form the sequence $(ad)(bc)(ad)\ldots$. Thus, for every asynchronous execution which is an interleaving synchronous one, there is an action d between two actions a. However, one possible sequence of actions obtained by an asynchronous execution is $abab\ldots$. □

Though the previous theorem is easy to obtain, it has an important impact for model checking. Suppose we indeed model a synchronized system by means of a DTS. To be able to use standard model checking approaches, one might be tempted to consider all of its asynchronous executions instead of all of its synchronous ones. Furthermore, one might be in favor of using $\mathrm{LTL_w}$ to specify requirements of our sequences of actions. But then it is likely to get false evidence. We might find a sequence of actions which is a counter example for the given $\mathrm{LTL_w}$ formula which is not obtained by an interleaving synchronous execution.

8.4 A Calculus for DSTS

In this section, we introduce the process calculus *synchronous process system* (SPS) which may be employed to define a distributed synchronous transition system. Within the area of verification, a distributed system is preferably given in terms of such a calculus instead of directly presenting a transition system. A lot of different kinds of so-called process algebras have been developed. Besides CCS [Mil89], there are CSP [Hoa85] and ACP [BV94, Fok00] to name the most popular ones. See [BPS01] for an overview.

Our notion of *synchronous process systems* is inspired by Milner's CCS [Mil89]. CCS is, however, designed to support the communication of two communication partners. It is difficult to realize a so-called broadcast in which every entity of a system has to be consulted. SPS does not distinguish between sender and receiver but communication is modeled by executing common actions which can also be requested by more than two communication partners. Thus, broadcasts can easily be defined.

The presented approach is quite simple and is mainly intended to show that our synchronous approach can be enriched with a process algebra formalism in a straightforward manner.

An SPS specification consists of the synchronous product of independently defined processes. Each process is given as a set of recursive equations that are built up using prefixing with actions and non-deterministic choice operators. Furthermore, we add the empty process nil.

Definition 8.4.1
Let $\Gamma = \{\mathrm{nil}^{(0)}, +^{(2)}, .^{(2)}\}$ *be a ranked alphabet,* Σ *a finite set of nullary actions and* $\mathcal{P}$ *a set of (process) variables. The set of* sequential process terms $SPT(\Sigma, \mathcal{P})$ *is inductively defined by*

- $P, \mathrm{nil} \in SPT(\Sigma, P)$*, if* $P \in \mathcal{P}$*,*
- $t_1, t_2 \in SPT(\Sigma, P), a \in \Sigma \Rightarrow a.t_1, t_1 + t_2 \in SPT(\Sigma, \mathcal{P})$.

We let $Sub(t)$ denote the set of subterms of t which is defined in the usual way. For the alphabet $\Sigma = \{a, b, c, d\}$ (which underlies also further examples in this chapter), $a.b.\mathrm{nil}$, $a.b.P$, and $a.\mathrm{nil} + b.\mathrm{nil}$ are examples of process terms.

Process terms can be used in process definitions as right hand sides of equations. Several process definitions are combined to yield the overall process system:

Definition 8.4.2
A process definition *over* $(\Sigma, \mathcal{P})$ *is a tuple* $\mathcal{D} = (P^0, (P = t_P)_{P \in \mathcal{P}})$ *where* P^0 *is an initial process variable from* $\mathcal{P}$ *and* $(P = t_P)_{P \in \mathcal{P}}$ *is a family of equations where* t_P *is a sequential process term over* $(\Sigma, \mathcal{P})$*.*

A synchronous process system *over a distributed alphabet* $\tilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ *and a finite tuple of sets of process variables* $\mathcal{P} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ *is a family of process definitions*

$$\mathcal{S} = (\mathcal{D}_1, \ldots, \mathcal{D}_n)$$

where each $\mathcal{D}_i$ $(i \in \{1, \ldots, n\})$ *is a process definition over* $(\Sigma_i, \mathcal{P}_i)$*.*

When the process definitions $\mathcal{D}_i = (P_i^0, \ldots)$ are given by the context, we often write $\mathcal{S}$ as

$$\mathcal{S} = P_1^0 \parallel \cdots \parallel P_n^0$$

For example,

$$\begin{array}{lcl} P_1 & = & a.b.P_1 \\ P_2 & = & a.c.P_2 \\ P_3 & = & b.d.P_3 \\ P_4 & = & c.d.P_4 \end{array}$$

are process definitions (the processes $P_1, \ldots, P_4$ are defined), which can be turned into a synchronous process specification by adding the equation

$$P = P_1 \parallel P_2 \parallel P_3 \parallel P_4$$

The semantics of a synchronous process system is defined in two steps. First, we define the semantics of a process definition, i.e., the semantics of a single family of equations, using inference rules in the way as done to define an SOS semantics [Plo81]. Second, we explain how to combine the local systems to derive a distributed synchronous transition system.

Definition 8.4.3
The semantics of a process definition $(P^0, (P = t_P)_{P \in \mathcal{P}})$ over $(\Sigma, \mathcal{P})$ is a (finite) transition system $(S, \rightarrow)$ where $S = \bigcup_{P \in \mathcal{P}} (\{P\} \cup Sub(t_P) - \{t_P\})$ and $\rightarrow : S \times \Sigma \times S$ is a labeled transition relation defined by the following inference rules:

$$\frac{}{a.t_1 \overset{a}{\rightarrow} t_1} \qquad \frac{t_1 \overset{a}{\rightarrow} t_1'}{t_1 + t_2 \overset{a}{\rightarrow} t_1'}$$

$$\frac{t \overset{a}{\rightarrow} t'}{P \overset{a}{\rightarrow} t'} \ (P{=}t) \qquad \frac{t_2 \overset{a}{\rightarrow} t_2'}{t_1 + t_2 \overset{a}{\rightarrow} t_2'}$$

The semantics of the definition $P_1 = a.b.P_1$ is given by the transition system with states $\{P_1, b.P_1\}$ and transitions $P_1 \xrightarrow{a} b.P_1$ and $b.P_1 \xrightarrow{b} P_1$.

The single transition systems are now combined to derive a synchronous process system. We simply take a typical product approach to obtain an overall semantics for an SPS:

Definition 8.4.4
The semantics of a process system $\mathcal{S} = (\mathcal{D}_1, \ldots, \mathcal{D}_n)$ *is defined to be the following distributed synchronous transition system: For* $\mathcal{D}_i$*, let* $(S_i, \rightarrow_i)$ *be its semantics. Let*

$$\longrightarrow = \left\{ ((q_1, \ldots, q_n), a, (q_1', \ldots, q_n')) \;\middle|\; \begin{array}{l} \forall i \in pr(a)\ (q_i, a, q_i') \in \rightarrow_i \textit{ and} \\ \forall i \in Proc \backslash pr(a)\ q_i = q_i' = - \end{array} \right\}$$

The distributed transition system for $\mathcal{S}$ *is* $(S_1, \ldots, S_n, \longrightarrow, \{(P_1^0, \ldots, P_n^0)\})$

It is easy to see that the synchronous process specification shown in the example yields the distributed transition system shown in Figure 8.5 on page 128.

A drawback of our calculus is that not every DSTS can be defined in terms of an SPS. This can easily be seen by considering the *language* of an DSTS, which is the set of sequences of actions of every possible execution whereby every set of actions is linearized in every possible way. It is an easy exercise to see that no SPS can define a DSTS such that we obtain the trace language $\Sigma^*(bc + cb)\Sigma^\omega$ over the alphabet $\Sigma = \{a, b, c, d\}$ where the only independent actions are a and d, and b and c (cf. Example 3.1.4 on page 16). Intuitively, we will either get a DSTS whose language contains $bacc^\omega$ or one whose language does not contain $bc\ldots$. On the contrary, it is simple to define this language when DSTSs are considered.

Lemma 8.4.5 *The class of languages definable by* SPS *is strictly contained in the class of languages definable by* DSTSs.

For a further study, we mention the concepts of *product languages* and *trace languages* and refer to [Thi95], in which these concepts are investigated in detail.

8.5 Foata Linear Temporal Logic ($\mathrm{LTL_f}$)

We are ready to introduce *Foata linear temporal logic* ($\mathrm{LTL_f}$), which is patterned after $\mathrm{LTL_w}$ and may be used to specify the behavior of a distributed synchronous transition system.

Definition 8.5.1 (Syntax of $\mathrm{LTL_f}$)
Let (Σ, I) *be an independence alphabet and* $\mathrm{I}(\Sigma)$ *the set of pairwise independent subsets of* Σ. $\mathrm{LTL_f}(\Sigma, I)$ *is the set of formulas given by the following grammar:*

$$\varphi ::= \mathrm{tt} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \langle A \rangle \varphi \mid \mathrm{O}\varphi \mid \varphi U \psi$$

where $A \in \mathrm{I}(\Sigma)$.

Of course, we make use of our standard abbreviation scheme for $\mathrm{LTL_f}(\Sigma, I)$. $\mathrm{LTL_f}$ comprises the usual Boolean connectives and an *until*-operator. An important difference in the syntax of $\mathrm{LTL_f}$ and $\mathrm{LTL_w}$ is that (independent) sets of actions may be used in the $\langle \_ \rangle$-operator in $\mathrm{LTL_f}$ while in $\mathrm{LTL_w}$ only a single action is allowed. The set of actions is used to define atomic steps of a DSTS which becomes visible considering the semantics of $\mathrm{LTL_f}$ formulas:

Definition 8.5.2 (Semantics of $\mathbf{LTL_f}$)
Let T be a trace over (Σ, I). The satisfaction relation of a formula $\varphi \in \mathrm{LTL_f}(\Sigma, I)$ with respect to a Foata configuration C of T is inductively defined by

- $T, C \models \mathrm{tt}$,
- $T, C \models \neg\varphi \Leftrightarrow T, C \not\models \varphi$,
- $T, C \models \varphi \vee \psi$ *iff* $T, C \models \varphi$ *or* $T, C \models \psi$,
- $T, C \models \langle A \rangle \varphi$ *iff there exists an* $A' \in \mathrm{I}(\Sigma)$, $A' \supseteq A$ *and* $C' \in \mathrm{fconf}(T)$ *such that* $C \overset{A'}{\to} C'$ *and* $T, C' \models \varphi$, *where* $C \overset{A'}{\to} C'$ *iff* $C, C' \in \mathrm{fconf}(T)$, $A' \in \mathrm{I}(\Sigma)$, *and* $\lambda(C' \setminus C) = A'$,
- $T, C \models \mathrm{O}\varphi$ *iff there exists an* $A \in \mathrm{I}(\Sigma)$ *and* $C' \in \mathrm{fconf}(T)$, *such that* $C \overset{A}{\to} C'$ *and* $T, C' \models \varphi$,
- $T, C \models \varphi U \psi$ *iff there exists* $C' \in \mathrm{fconf}(T)$, $C' \supseteq C$ *such that* $T, C' \models \psi$ *and for all* $C'' \in \mathrm{fconf}(T)$, $C \subseteq C'' \subset C'$ *implies* $T, C'' \models \varphi$.

$T, \emptyset$ is often abbreviated by T, a habit which was already practiced when considering $\mathrm{LTL_t}$. Similarly, we say T *models* φ, T is a *model* for φ, or T *satisfies* φ iff $T \models \varphi$. Furthermore, we call φ *satisfiable* iff there is a $T \in \mathbb{TR}(\Sigma, I)$ such that $T \models \varphi$. All models of a formula $\varphi \in \mathrm{LTL_f}(\Sigma, I)$ constitute a subset of $\mathbb{TR}(\Sigma, I)$, thus a language. It is denoted by $\mathcal{L}(\varphi)$ and is called the *language* defined by φ. Furthermore, every formula defines an ω-language viz the set $\{w \in \mathrm{lin}(T) \mid T \models \varphi \text{ and } w \text{ in Foata normal form}\}$, which is also indicated by $\mathcal{L}(\varphi)$.

Let a distributed synchronous transition system and a formula $\varphi \in \mathrm{LTL_f}$ be given. Then let L be the set of all words (in Foata normal form) corresponding to executions of the transition system. *Model checking* is the problem whether $L \subseteq \mathcal{L}(\varphi)$.

The semantics of Boolean connectives is as usual. For formulas of the kind $\langle A \rangle \varphi$, we require a superset A' of A to exist for transforming the system from configuration C to C'. This simplifies the task of specification since the user only has to specify the actions he or she wants to see while leaving the atomic actions of the components not involved by actions in A unspecified. For example, $\langle \{a\} \rangle \langle \{b\} \rangle \mathrm{tt}$ is satisfied by the configuration graph shown in Figure 8.4(b) on page 125 (which can be identified with the execution shown in Figure 8.6(a) on page 129) because Foata configuration $\{b, c\}$ follows $\{a\}$.

If we change our semantics in the way that exactly the actions specified must be employed to move from configuration $C \rightarrow C'$, we can transform every formula of our logic into this logic by taking any combination of the remaining actions. However, in general this causes an *exponential blow-up* of our formula augmenting the overall effort of deciding satisfiability and model checking. For the same reason, we also added a O-operator, which could have been simulated by a disjunction of $\langle \_ \rangle$-operators where the disjunction ranges over all elements of $\mathrm{I}(\Sigma)$.

Of course, additional operators requiring A' to be a subset of A or requiring A' to be equal to A are desirable as well. It is an easy exercise to enrich our logic and algorithms to support these additional operators without increasing the complexity of the latter. To simplify the presentation, we only picked out a single interpretation of a $\langle \_ \rangle$-operator.

The *until*-operator is defined similarly as in the case of LTL_t. However, only Foata configurations are considered, which are linearly ordered. As we will see in the next section, this simplifies the algorithm as well as its complexity of deciding satisfiability dramatically.

Let us note that LTL_f can be called a conservative extension of LTL_w:

Remark 8.5.3 For a fully-dependent alphabet Σ, $\mathrm{I}(\Sigma)$ is a set of singletons, each containing a single action. Thus, every step of a trace consists of a singleton. Hence, the LTL_f and LTL_w can be identified.

As usual, we introduce abbreviations of the following kind to simplify the task of specifying requirements:

- $\varphi \wedge \psi$ for $\neg(\neg\varphi \vee \neg\psi)$
- $\Diamond\varphi$ for $\mathrm{tt} U \varphi$
- $\Box\varphi$ for $\neg\Diamond\neg\varphi$

Hence, it is possible to express global liveness and safety properties in a manner as known from LTL.

Our logic can be understood as LTL over the alphabet $\mathrm{I}(\Sigma)$ with the exception of the different interpretation of the next-state operator. Then one might think of employing the standard LTL algorithms for deciding LTL_f. However, since not every word in $\mathrm{I}(\Sigma)^\omega$ is in Foata normal form, the standard algorithms for deciding and model checking algorithm for LTL have to be modified to consider only models in Foata normal form. Furthermore, the algorithms have to be modified to respect our special form of the $\langle \_ \rangle$-operator.

With respect to model checking, the models to analyze are given by distributed transition systems over the alphabet Σ. To employ a logic over $\mathrm{I}(\Sigma)^\omega$, the transition system has to be transformed into a single *bisimilar* [Mil89] one over I(Σ). For practical reasons, this has to be carried out *on-the-fly*. It is not clear how to achieve this goal.

Altogether, we are convinced that understanding LTL$_\mathrm{f}$ as a logic over I$(\Sigma)^\omega$ might be theoretically interesting but is the second choice for practical algorithms. We therefore directly formulate decision procedures for satisfiability and model checking over Σ, since this yields more efficient practical implementations. However, the decision procedure for satisfiability owes some ideas from the previously mentioned interpretation.

8.6 Satisfiability of LTL$_\mathrm{f}$

We now present a decision procedure for LTL$_\mathrm{f}$ formulas by means of alternating Büchi automata. The decision procedure is divided into two steps. First, given a formula $\varphi \in \mathrm{LTL}_\mathrm{f}(\Sigma, I)$, we define an automaton $\mathcal{A}_\varphi$ that, for all *Foata linearizations* w, accepts w if and only if the trace T_w induced by w satisfies φ. In a second step, we will define a Büchi automaton $\mathcal{A}_\mathcal{F}$ accepting a word in Σ^ω iff it is in *Foata normal form.* Hence, the language of the automaton accepting the intersection of the languages $\mathcal{A}_\varphi$ and $\mathcal{A}_\mathcal{F}$ is non-empty if and only if φ is satisfiable.

Let us fix a formula $\varphi \in \mathrm{LTL}_\mathrm{f}$ for this section. Again, the idea for defining $\mathcal{A}_\varphi$ is to take $Sub(\varphi)$ (plus their negations) as (one part of) the state space of $\mathcal{A}_\varphi$ where $Sub(\varphi)$ denotes the set of subformulas of φ, which is defined as usual. Boolean combinations of formulas and *until*-formulas can be treated as usual, the latter because *until*-formulas can be "unwinded" subject to the equivalence $\varphi U \psi \equiv \psi \vee (\varphi \wedge \mathrm{O}(\varphi U \psi))$. The only non-straightforward case is a $\langle \_ \rangle$-formula.

Given a linearization of a trace in Foata normal form, the different steps are characterized by actions dependent on the preceding step. The idea for the transition function of the automaton with respect to $\langle \_ \rangle$-formulas is to collect the independent actions of the current step in one component of the automaton's state. When an action dependent on the current step is read, the underlying formula (which is contained in a second component of the current state) is compared with the actions of the step. Thus, the state space of the automaton is defined as $\mathrm{I}(\Sigma) \times (Sub(\varphi) \cup \neg Sub(\varphi))$.

Let us give a precise definition of $\mathcal{A}_\varphi$:

Definition 8.6.1
Let $\varphi \in \mathrm{LTL}_\mathrm{f}(\Sigma, I)$*. Then* $\mathcal{A}_\varphi = (Q, \delta, q_0, F)$ *is defined by* $Q = \mathrm{I}(\Sigma) \times (Sub(\varphi) \cup$

$\neg Sub(\varphi)$), $q_0 = (\emptyset, \varphi)$ and $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ by

$$\begin{array}{rcl}
((S, \mathrm{tt}), a) & \mapsto & \mathrm{tt} \\
((S, \psi \vee \eta), a) & \mapsto & \delta((S, \psi), a) \vee \delta((S, \eta, a) \\
((S, \neg\psi), a) & \mapsto & \overline{\delta((S, \psi), a)} \\
((S, \langle A \rangle \psi), a) & \mapsto & \begin{cases} \delta((\emptyset, \psi), a) & \text{if } aDS, A \subseteq S \\ \mathrm{ff} & \text{if } aDS, A \not\subseteq S \\ (S \cup \{a\}, \langle A \rangle \psi) & \text{if } aIS \end{cases} \\
((S, \mathrm{O}\psi), a) & \mapsto & \begin{cases} \delta((\emptyset, \psi), a) & \text{if } aDS \\ (S \cup \{a\}, \mathrm{O}\psi) & \text{if } aIS \end{cases} \\
((\emptyset, \psi U \eta), a) & \mapsto & \delta((\emptyset, \eta \vee (\psi \wedge \mathrm{O}(\psi U \eta))), a)
\end{array}$$

The set of final states is, as usual, given by the states with negative formulas, $F = \mathrm{I}(\Sigma) \times \{\neg\varphi \mid \varphi \in Sub(\varphi)\}$.

Note that the dual of an $\mathrm{LTL_f}$ formula is defined is a straightforward way (cf. Definition 6.2.3 on page 72).

As mentioned before, *until*-formulas are directly unwinded according to $\varphi U \psi \equiv \psi \vee (\varphi \wedge \mathrm{O}(\varphi U \psi))$. Hence, the transition function δ just treats the situation for an empty step.

Let us explain the acceptance conditions of the automaton. Every finite branch of a run of $\mathcal{A}_\varphi$ ending in tt gives a "proof" for our formula. Infinite branches only occur by infinitely often unwinding *until*-formulas. Hence, they must be accepted iff the *until*-formula is negated. A formal proof of our construction is straightforward and only technically involved, which encourages us to declare it an exercise. We instead commence with a small example:

Example 8.6.2 Suppose we want to know for the formula $\varphi = \langle\{a\}\rangle\langle\{b\}\rangle\mathrm{tt}$ whether it is satisfiable or not. We underlay our well-known independence alphabet of Figure 3.1.4 on page 16. We already know that φ is satisfiable. To simplify our presentation, we do not present $\mathcal{A}_\varphi$ completely[3] but only accepting and rejecting runs. Suppose the first input action is a. Starting in state $(\emptyset, \varphi)$, the automaton then proceeds in state $(\{a\}, \varphi)$. Suppose the next input action is d. As d is independent of a, it is added to the current step and the automaton moves to state $(\{a, d\}, \varphi)$. Now, let us read the action c, which is dependent on a. As the a action requested by φ is contained in $\{a, d\}$, the automaton proceeds in state $(\{c\}, \langle\{b\}\rangle\mathrm{tt})$. Note that, without having seen a before reading c, the run of the automaton would have been withdrawn. Reading b and one further action dependent on b will guide the automaton to tt and the corresponding run is accepting.

---

[3] $\mathcal{A}_\varphi$ has 21 reachable states and far more transitions

As the example shows, it is important that we deal with infinite words. This guarantees that we indeed consider every step because every step consists of only finitely many actions.

Note that for the case of the fully-dependent alphabet, we obtain a slightly different decision procedure as for LTL_w (cf. Chapter 6 on page 69), although maintaining the same complexity bounds. Intuitively, $\mathcal{A}_\varphi$ "carries" the action read and modifies the underlying formula when reading the subsequent action while the construction in the case of LTL_w directly processes the input action read.

Now, we define an automaton $\mathcal{A}_\mathcal{F}$ accepting Foata linearizations of traces, which will be used to guarantee that the previous automaton indeed "is fed" with words in Foata normal form. It can be understood as a kind of filter, rejecting ω-words which cannot be a Foata linearization of a trace.

According to the definition of the Foata normal form (see Section 8.2), a word is in Foata normal form if it can be written as a product of steps. A step is a word of pairwise independent letters. Furthermore, for every step (excluding the first one) there is a dependent action in the previous one. $\mathcal{A}_\mathcal{F}$ reads a word step by step. A part of a step is stored in S, which is one part of $\mathcal{A}_\mathcal{F}$'s current state. An action independent on actions of the current step must belong to the current step. Hence it is added to S. As soon as an action is read which is dependent on one of the actions of the current step, it must be part of the next step which is initialized by this action. Furthermore, to reflect the second requirement for steps, we store in G the actions dependent on S or the current read action. These are the *good* actions which we allow to be read from now on, because these are either dependent of the previous step (and independent of the current) or dependent on the current.

Let $D(S)$ denote the set of actions dependent on some action in S, i.e., $D(S) = \{b \in \Sigma \mid bDS\}$, and let us make our considerations precise.

Definition 8.6.3
$\mathcal{A}_\mathcal{F} = (Q, \delta, q_0, F)$ is defined by

- $Q = 2^\Sigma \times \mathrm{I}(\Sigma)$,
- $q_0 = (\Sigma, \emptyset)$,
- $F = Q$, *and*
- $\delta : Q \times \Sigma \to 2^Q$ *by*

$$((G,S),a) \mapsto \begin{cases} \emptyset & \textit{if } a \notin G \\ \{(G',S')\} & \textit{if } a \in G \end{cases}$$

where G' and S' are defined in the following way:

- *if* aIS *then* $S' = S \cup \{a\}$, $G' = G \cup D(S')$ *and*
- *if* aDS *then* $S' = \{a\}$, $G' = D(S \cup S')$.

If an input action is independent on the current step, new actions might be "good" viz the ones which are dependent on the action currently read. This is expressed by $G' = G \cup D(S')$ in the previous definition. Because of $((G, S), a) \mapsto \emptyset$ if $a \notin G$, it is ensured that all read actions added to the step S are indeed dependent on one action of the previous step. Again, correctness is a matter of studiousness.

To obtain a decision procedure for checking satisfiability of a formula φ, it remains to construct an automaton accepting the intersection of the languages of $\mathcal{A}_\varphi$ and $\mathcal{A}_\mathcal{F}$ which then can be examined for (non-)emptiness. A simple way to reach this goal is translating $\mathcal{A}_\varphi$ into a Büchi automaton $\mathcal{A}'_\varphi$ and to use standard constructions on Büchi automata [Tho90a].

Since in $\mathcal{A}_\mathcal{F}$ every state is also a final state, its intersection with a Büchi automaton has a simple form: Let $\mathcal{A} = (Q, \delta, q_0, F)$ be a Büchi automaton. Define $\mathcal{A}' = (Q \times 2^\Sigma \times \mathrm{I}(\Sigma), \delta', (q_0, \Sigma, \emptyset), F \times 2^\Sigma \times \mathrm{I}(\Sigma))$ by

$$(q', G', S') \in \delta'((q, G, S), a)$$

iff $q' \in \delta(q, a)$ and $a \in G$ and if aIS then

$$\begin{array}{rcl} S' & = & S \cup \{a\} \\ G' & = & G \cup D(S') \end{array}$$

and if aDS then

$$\begin{array}{rcl} S' & = & \{a\} \\ G' & = & D(S \cup S') \end{array}$$

Complexity It is easy to see that for $\varphi \in \mathrm{LTL_f}$, the size of $\mathcal{A}_\varphi$ is linear in the size of φ. Hence, the size of the resulting Büchi automaton is exponential in the size of φ. $\mathcal{A}_\mathcal{F}$ is independent of φ, so is its size. Hence, deciding whether there is a model for φ is exponential in its length and can be carried out in polynomial space. This is optimal since for the fully-dependent alphabet, we are in the situation of LTL.

Expressiveness A formula $\varphi \in \mathrm{LTL_f}$ defines a trace language $\mathcal{L}(\varphi) = \{T \mid T \models \varphi\}$. Which kind of languages are definable by $\mathrm{LTL_f}$ formulas? Foata configurations are defined in an inductive manner. Thus, it is easy to see that these languages are definable by monadic second order logic for Mazurkiewicz traces, which is defined as expected [EM93]. However, it is neither clear whether first-order formulas are expressive enough nor whether any language defined by a first-order formula can also be defined by an $\mathrm{LTL_f}$ formula. We assume that both is not the case.

8.7 Model Checking for DSTS and $\mathrm{LTL_f}$

In this section, we present a model checking algorithm for $\mathrm{LTL_f}$ with respect to the executions of a distributed synchronous transition system. Let a DSTS $\mathcal{A}$ and a formula φ be given. We construct a Büchi automaton $\mathcal{B}_{\mathcal{A}}$ accepting for every synchronous execution of $\mathcal{A}$ a *single* asynchronous one. In contrast to accepting every asynchronous execution, this reduces the number of possible transitions and, more importantly, the number of reachable states.

For the negation of φ, we construct an alternating Büchi automaton $\mathcal{A}_{\neg\varphi}$ and transform it into a Büchi automaton $\mathcal{B}_{\neg\varphi}$ as described in the previous section. Testing the intersection of $\mathcal{B}_{\mathcal{A}}$ and $\mathcal{B}_{\neg\varphi}$ for emptiness answers whether there is an execution of $\mathcal{A}$ violating φ.

The underlying idea is rather simple. Every word in Foata normal form is accepted by $\mathcal{B}_{\neg\varphi}$, iff the corresponding Foata configuration graph satisfies $\neg\varphi$. A straightforward approach would be to translate $\mathcal{A}$ into a Büchi automaton $\mathcal{B}_{\mathcal{A}}$ accepting all Foata normal forms such that the corresponding Foata configuration graph corresponds to an execution of $\mathcal{A}$. However, several words in Foata normal form yield the same Foata configuration graph. Since $\mathcal{B}_{\neg\varphi}$ accepts *all* Foata linearizations of a Foata configuration graph satisfying $\neg\varphi$, we may design $\mathcal{B}_{\mathcal{A}}$ in the way that it only accepts a single characteristic one.

Definition 8.7.1
Let $\mathcal{A} = (Q_1, \ldots, Q_n, \longrightarrow, \mathcal{I})$ *be a* DSTS*. Then let* $\mathcal{B}_{\mathcal{A}} = (Q, \delta, \mathcal{I} \times \{\emptyset\}, F)$ *be the Büchi automaton defined by* $Q = Q_1 \times \cdots \times Q_n \times \mathrm{I}(\Sigma)$ *and* $F = Q$*, i.e., every state is also a final state. Fix a linear order* $\prec$ *on the alphabet* Σ.[4] *We call an action* a enabled *in* $\bar{q} \in \bar{Q}$ *iff there is a* $\bar{q}' \in \bar{Q}$ *such that* $(\bar{q}|_{pr(a)}, a, \bar{q}'|_{pr(a)}) \in \longrightarrow$*. Let* $(\bar{q}', S') \in \delta((\bar{q}, S), a)$ *iff*

1. $(\bar{q}|_{pr(a)}, a, \bar{q}'|_{pr(a)}) \in \longrightarrow$*, i.e., it is a valid transition according to the underlying* DTS*, and*

2. *if* aDS *then*

    (a) $\{b \in \Sigma \mid bIS \text{ and } b \text{ enabled in } \bar{q}\} = \emptyset$*, i.e., there is no action independent of the current step left for execution, and*

    (b) *a is strictly smaller than each element of the set* $\{b \in \Sigma \mid bIa \text{ and } b \text{ enabled in } \bar{q}\}$ *with respect to* $\prec$ *and* $S' = \{a\}$

    and

---

[4] Note that it suffices to define $\prec$ of pairs of independent actions only. Hence, $\Sigma_1, \ldots, \Sigma_n$ induces an appropriate $\prec$.

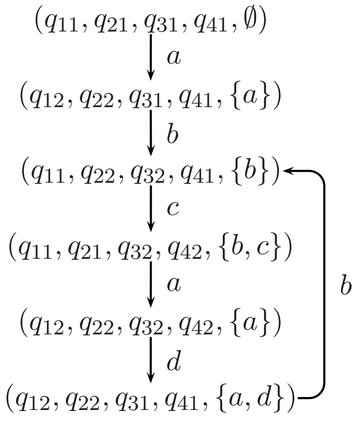


Figure 8.8: A Büchi automaton for the DSTS shown in Figure 8.5

3. *if* aIS *then* a *is strictly smaller than each element of* $\{b \in \Sigma \mid bIa, bIS,$ *and* b *enabled in* $\bar{q}\}$ *with respect to* $\prec$ *and* $S' = S \cup \{a\}$*.*

Item 2.(a) guarantees that the next step is considered only if the current one is "full". Items 2.(b) and 3 handle the selection of "equivalent" transitions. The rule is: Just let the smallest action with respect to $\prec$ make a transition. The new current step S' is treated as in the definition of $\mathcal{A}_{\mathcal{F}}$ (see Section 8.6). Note that we do not have to concentrate on *good* actions since our selection strategy ensures to fill a step before considering the next one. This shows that $\mathcal{B}_{\mathcal{A}}$ accepts for every synchronous execution of $\mathcal{A}$ exactly one linearization. For example, $\mathcal{B}_{\mathcal{A}}$ for $\mathcal{A}$ as in Figure 8.5 is shown in Figure 8.8

Complexity Model Checking is exponential in the size of the formula and linear in the size of $\mathcal{B}_{\mathcal{A}}$. The size of $\mathcal{B}_{\mathcal{A}}$ is exponential in the number of the components of $\mathcal{A}$. The experiences gained by partial order reduction [Pel98] allow the conclusion that the number of reachable states is in the average case much smaller. Note that, using the same arguments as for Theorem 6.5.1 on page 81 and using the fact that it is easy to define a distributed transition system whose executions form Σ^{ω}, we get

Theorem 8.7.2
The complexity of deciding satisfiability is less than or equal to the formula complexity of model checking for $\mathrm{LTL}_{\mathrm{f}}$*.*

8.8 A Two Bit Counter

Let us commence with a larger, more detailed example, again taken from the domain of hardware systems. We use notions of circuit design on an intuitive basis because

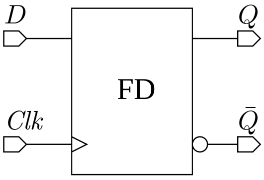


Figure 8.9: A D-flip-flop

we only want to give a flavor of our methods. Please consult [TS01] or [Cha99] for further details on designing digital systems.

Suppose we want to develop a *two bit counter*. This is a digital circuit in which we can spot two bit values that take all of the values $\{00, 01, 11, 10\}$ in an infinite loop. In other words, we want to identify 4 states determined by the value of two bits and the circuit loops in these four states.

Furthermore, the device should be "*hazard-free*". In our setting, this means that only one bit is changed at a time. Thus, from state 01 we proceed in state 11 instead of 10. The reason is that changing a single bit each time makes a design more "robust" in practice. Suppose we are in state 01 and proceed in state 10, which can be obtained by "flipping" both bits. Only in theory, it is possible to flip two bits *exactly* at the same point in time, but in practice, there will be a (quite small) delay between both changes. Hence, we would be in state 11 or 00 for a moment. As these cases are undesirable, we will go for the safer design requiring our circuit to behave like:

$$00, 01, 11, 10, 00, \ldots$$

It is simple to realize our design using a *D-flip-flop*. A D-flip-flop can be visualized as in Figure 8.9. It has four connectors, *Clk*, D, Q, and $\bar{Q}$. *Clk* is the input of the (global) clock signal. We take a setting in which the D-flip-flop proceeds with every rising edge of the clock. D is the *data input*, Q is the output, and $\bar{Q}$ is the inverted output of Q. Input- and Output-lines can take the values logical 0 or 1. Reading a value on the input D, the flip-flop stores this input in an internal state S with each rising edge. The previous internal state is available at Q and its negation at $\bar{Q}$. Thus, it takes two rising edges to pass an input value on D to the output Q. The behavior of the flip-flop can be described by the function shown in Table 8.1 on the next page in which the input D and the current state S are transformed to output values Q and $\bar{Q}$ as well as to a new state S.[5]

We are now ready to combine two D-flip-flops to obtain a two bit counter, as shown in Figure 8.10 on the following page. The negated output of the first flip-flop is

---

[5] The behavior of circuits can also be described by means of finite automata with output like Mealy- and Moore-automata [Kro99]. However, to limit the formal notions to be introduced, we rely on a slightly more sloppy description of circuits.

D	S	$\mapsto$	Q	$\bar{Q}$	S
0	0	$\mapsto$	0	1	0
0	1	$\mapsto$	1	0	0
1	0	$\mapsto$	0	1	1
1	1	$\mapsto$	1	0	1

Table 8.1: Function table of a D-flip-flop

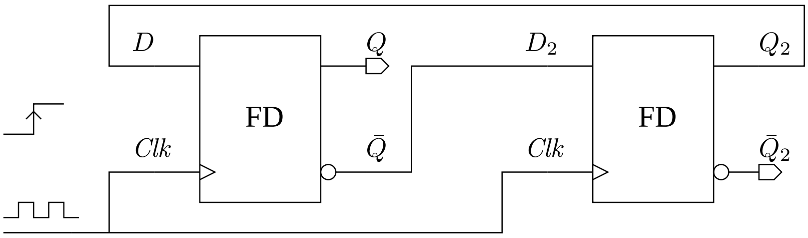


Figure 8.10: A two bit counter

connected with the input of the second one, and the input of the first is fed with the output of the second. It is easy to see that we get the bit values shown in Table 8.2 where in the first column the rising edges of the clock are counted ($\#re$) and the subsequent columns represent the values in this moment. The initial situation is consequently shown in the first row beginning with 0.

The values of the bits for the eights rising edge are identical to the initial case so that the first eight cases are characteristic for the whole system. The value of the two bit counter can be obtained by considering Q and Q_2 which is highlighted in the last two columns in Table 8.2. Thus, we indeed get the desired sequence $00, 01, 11, 10, 00, \ldots$.

Our goal is to describe the system formally using our notions of synchronous process systems and distributed synchronous transition systems.

$\#re$	D	S	Q	D_2	S_2	Q_2	Q	Q_2
0	0	0	0	1	0	0	0	0
1	0	0	0	1	1	0	0	0
2	1	0	0	1	1	1	0	1
3	1	1	0	1	1	1	0	1
4	1	1	1	0	1	1	1	1
5	1	1	1	0	0	1	1	1
6	0	1	1	0	0	0	1	0
7	0	0	1	0	0	0	1	0
8	0	0	0	1	0	0	0	0

Table 8.2: The two bit counter

We first start with modeling a D-flip-flop.

The idea is to implement the dynamic behavior of the system incorporating the logical values on each connector and to describe each connector as a separate process.

Thus, we deal with actions of the form $D0$ and $D1$ denoting that connector D is logical 0 or logical 1, respectively. As pointed out before, a D-flip-flop has an internal state which is affected by the value of D and influences the values of Q and $\bar{Q}$. Thus, we commence with two further actions $S0$ and $S1$. However, the values at the connectors D and Q are not directly dependent. Altogether, we set the distributed alphabet as

$$\tilde{\Sigma} = (\{D0, D1, S0, S1\}, \{Q0, Q1, S0, S1\}, \{\bar{Q}0, \bar{Q}1, S0, S1\})$$

Let us define the behavior of a connector. A connector can be considered to be a process. For each connector we employ two variables P and P', for example, P_D and P'_D for connector D. As mentioned before, connector D affects the internal state. Let us assume that initially the value at connector D is 0. Then, the internal state is set 0 as well. Whenever the value at connector D is changed, the internal state is modified accordingly. Thus we define

$$P_D = D0.S0.P'_D \qquad\qquad P'_D = D0.S0.P'_D + D1.S1.P'_D$$

Similarly, we define P_Q and $P_{\bar{Q}}$ as

$$P_Q = Q0.P'_Q \qquad\qquad P'_Q = S0.Q0.P'_Q + S1.Q1.P'_Q$$

$$P_{\bar{Q}} = \bar{Q}1.P'_{\bar{Q}} \qquad\qquad P'_{\bar{Q}} = S0.\bar{Q}1.P'_{\bar{Q}} + S1.\bar{Q}0.P'_{\bar{Q}}$$

The D-flip-flop FD can now be defined as

$$FD = P_D \parallel P_Q \parallel P_{\bar{Q}}$$

The resulting DSTS can be visualized as in Figure 8.11 on the next page.

One synchronous execution of this system can be given as the word in Foata normal form

$$(D0\,Q0\,\bar{Q}1)(S0)(D1\,Q0\,\bar{Q}1)(S1)(D1\,Q1\,\bar{Q}0)\ldots$$

which represents the situation that the input on connector D is initially 0 and then changed to 1.

We now turn towards the definition of the two bit counter. Similarly as combing the D-flip-flops, we combine their equations. In principal, we now deal with a second copy of the previous actions D_20, D_21, Q_20, Q_21, $\bar{Q}_20$, and $\bar{Q}_21$, which represent the logical values at the connectors of the second flip-flop. As the output Q_2 of

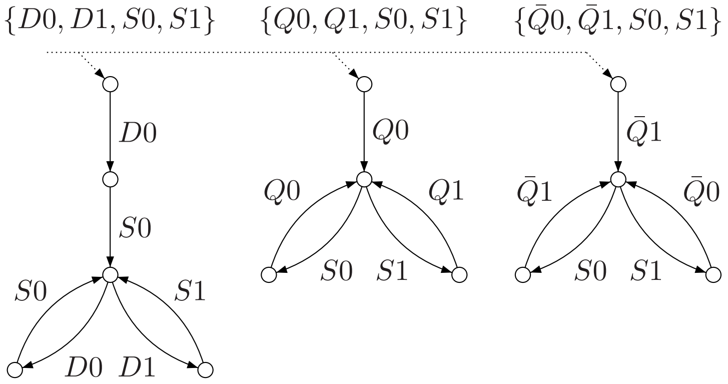


Figure 8.11: A DSTS for the D-flip-flop

the second flip-flop is connected with input D of the first, we have to identify the connectors in our definition and therefore also the actions $D0$ and $Q_2 0$. This counts also for the other logical value on this wire and the values on the wire connecting $\bar{Q}$ and D_2.

As our goal by introducing SPS was only to give a flavor of a process-algebra-like calculus suitable for defining DSTSs, we refrained from introducing operations like *renaming* [Mil89] etc. However, we have slightly more work to do now. We give the list of the equations defining the elements of the two bit counter:

$$\begin{aligned}
P_D &= D0.S0.P'_D \\
P'_D &= D0.S0.P'_D + D1.S1.P'_D \\
P_Q &= Q0.P'_Q \\
P'_Q &= S0.Q0.P'_Q + S1.Q1.P'_Q \\
P_{\bar{Q}} &= \bar{Q}1.P'_{\bar{Q}} \\
P'_{\bar{Q}} &= S0.\bar{Q}1.P'_{\bar{Q}} + S1.\bar{Q}0.P'_{\bar{Q}} \\
P_{D_2} &= \bar{Q}1.S_2 1.P'_{D_2} \\
P'_{D_2} &= \bar{Q}0.S_2 0.P'_{D_2} + \bar{Q}1.S_2 1.P'_{D_2} \\
P_{Q_2} &= D0.P'_{Q_2} \\
P'_{Q_2} &= S_2 0.D0.P'_{Q_2} + S_2 1.D1.P'_{Q_2} \\
P_{\bar{Q}_2} &= \bar{Q}_2 1.P'_{\bar{Q}_2} \\
P'_{\bar{Q}_2} &= S_2 0.\bar{Q}_2 1.P'_{\bar{Q}_2} + S_2 1.\bar{Q}_2 0.P'_{\bar{Q}_2}
\end{aligned}$$

Thus, the two bit counter is given by

$$TBC = P_D \parallel P_Q \parallel P_{\bar{Q}} \parallel P_{D_2} \parallel P_{Q_2} \parallel P_{\bar{Q}_2}$$

The overall system is depicted in Figure 8.12.

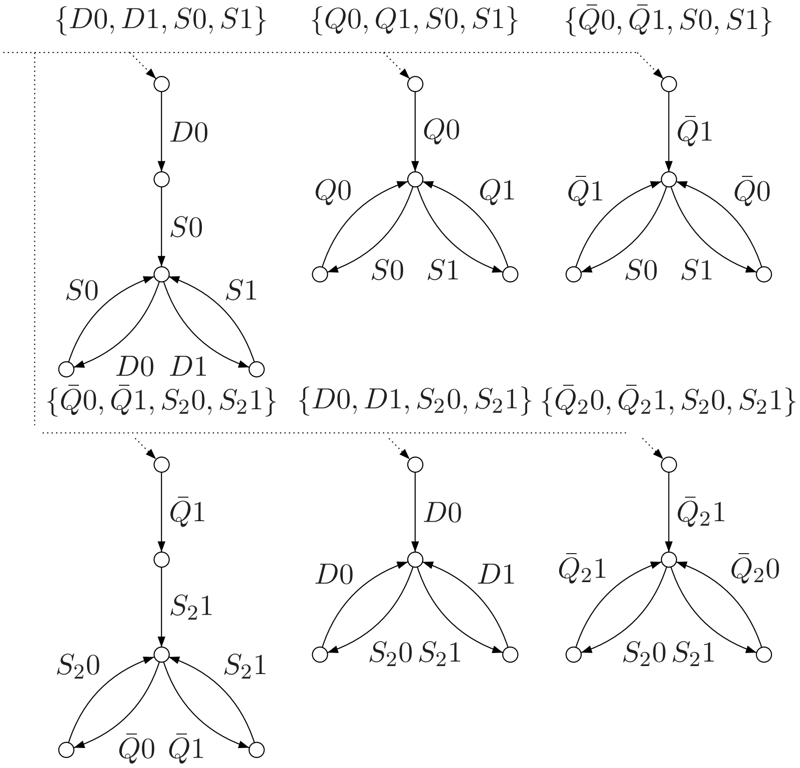


Figure 8.12: A DSTS for the two bit counter

It is now a simple matter to observe the following execution, which is again abstracted to a word in Foata normal form:

$$(D0\, Q0\, \bar{Q}1\, \bar{Q}_2 1)(S0\, S_2 1)(D1\, Q0\, \bar{Q}1\, \bar{Q}_2 0)(S1\, S_2 1)\ldots$$

Comparing this sequence with Table 8.2 on page 144, we see the correspondence of the execution of the defined two bit counter DSTS and the behavior of the two bit counter.

The system may now be analyzed with respect to specifications in terms of LTL$_\mathrm{f}$ formulas. For example, we can verify whether state 00 is always followed by 01 by checking the formula

$$\Box\,(\langle\{D0, Q0\}\rangle \mathrm{tt} \rightarrow \langle\{D0, Q0\}\rangle\langle\{\}\rangle\langle\{D1, Q0\}\rangle \mathrm{tt})$$

Chapter 9

Conclusion

In this thesis, we have introduced a new method for defining decision procedures for satisfiability of logical formulas in the domain of partial commutation. It is based on alternating automata and uses a notion of independence rewriting to formulas of the logic.

Specifically, we presented a decision procedure for LTL over Mazurkiewicz traces that generalizes the classical automata-theoretic approach to a linear temporal logic interpreted no longer over sequences but restricted labeled partial orders. We constructed a linear alternating Büchi automaton accepting the set of linearizations of traces satisfying the formula at hand. The salient point of our technique is to apply a notion of independence-rewriting to formulas of the logic.

Furthermore, we showed that the class of *linear* and *trace-consistent* alternating Büchi automata corresponds exactly to languages definable by LTL formulas over Mazurkiewicz traces, lifting a similar result from Löding and Thomas formulated in the framework of LTL over words.

For language theory, we could therefore paint the picture visualized in Figure 9.1 on the following page. Within the domain of all languages, we can identify the regular languages and herein the star-free languages. Orthogonally, we can distinguish trace-consistent and non-trace-consistent languages. We can sum-up the following characterizations:[1]

Theorem
Let Σ be an alphabet. Furthermore, let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent:

1. L is regular.

---

[1] Note that this list is far from complete. Further characterizations can be found in [Tho90a]. Since we only want to give an overview, we refer to this article also for notions we have not introduced.

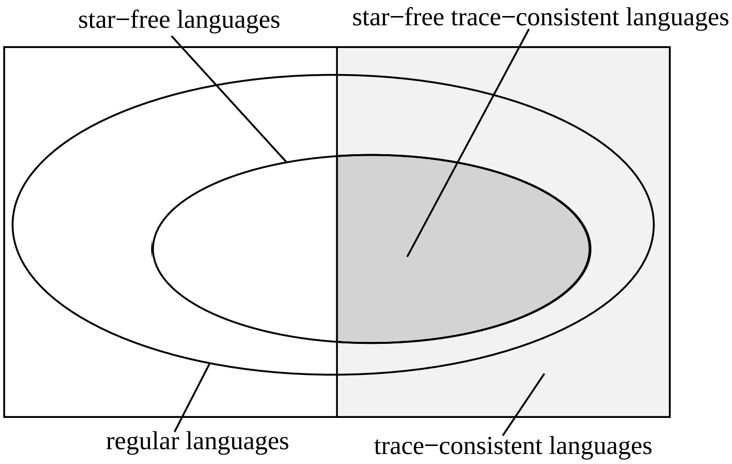


Figure 9.1: Regular languages

2. L is recognizable by a Büchi automaton.

3. L is definable in monadic second-order logic (over words).

4. L is recognizable by an alternating Büchi automaton.

Equivalences of (1) – (3) trace back to Büchi [Büc62]. Equivalence of (2) and (4) was shown in [MH84] (cf. Theorem 4.2.9 on page 51).

For star-free languages, we have the following characterizations:

Theorem
Let Σ be an alphabet. Furthermore, let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent:

1. L is a star-free language.[2]

2. L is definable in first-order logic (over words).

3. L is definable in $\mathrm{LTL}_{\mathrm{w}}$.

4. L is recognizable by a linear alternating Büchi automaton.

Thomas [Tho79] has shown equivalence of (1) and (2). Equivalence of (2) and (3) is Kamp's famous theorem [Kam68]. Löding and Thomas [LT00] have shown the last equivalence (cf. Theorem 6.3.2 on page 76).

Let us now turn towards the domain of traces, more precisely, to the domain of trace-consistent languages:

---

[2] We call L star free if it is a finite union of sets $U.V^\omega$ where $U, V \subseteq \Sigma^*$ are star-free and $V.V \subseteq V$.

Theorem
Let Σ be an alphabet. Furthermore, let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent:

1. L is a trace-consistent regular language.
2. L is recognizable by a trace-consistent Büchi automaton.
3. L is recognizable by an asynchronous automaton.
4. L is a linearization of a language definable in monadic second-order logic (over traces).[3]

Equivalence of (1) and (2) is immediate. The equivalence of (1) and (3) was introduced by Zielonka [Zie87]. Ebinger and Muscholl [EM93] have shown the equivalence of (1) and (4), extending a similar result for finite traces due to Thomas [Tho90b].

The star-free trace-consistent setting looks as follows:

Theorem
Let Σ be an alphabet. Furthermore, let $L \subseteq \Sigma^\omega$. Then the following statements are equivalent:

1. L is a star-free trace-consistent language.
2. L is a linearization of a language definable in first-order logic (over traces).
3. L is a linearization of a language definable in $\mathrm{LTL_t}$.
4. L is recognizable by a linear trace-consistent alternating Büchi automaton.

Equivalence of (1) and (2) traces back to [EM93]. Diekert and Gastin have shown equivalence of (2) and (3). We have shown the equivalence of (3) and (4) in Theorem 7.3.3 on page 110.

While the studied temporal logic $\mathrm{LTL_t}$ is a trace theoretic analogue of $\mathrm{LTL_w}$ because it captures the first-order definable languages, it does differ from $\mathrm{LTL_w}$ in one important point. Satisfiability of first-order logic is non-elementary for words as well as traces. $\mathrm{LTL_w}$ formulas can be tested for satisfiability in EXPTIME. For $\mathrm{LTL_t}$, however, the complexity for satisfiability remains non-elementary.

It is therefore natural to look for an expressively complete linear temporal logic for traces with an elementary time satisfiability procedure. We have not touched this problem here. But we feel that our characterization in terms of linear trace-consistent alternating Büchi automata might give a new direction to tackle this question.

---

[3] Recall that a linearization of a trace language is defined as the set of the linearizations of members.

In a second part of this thesis, we introduced a linear temporal logic designed for specifying properties of *synchronized* systems that comprise *clocked* hardware circuits or *Petri nets* supplied with a *maximal step semantics.* We called this logic Foata linear temporal logic ($\mathrm{LTL}_{\mathrm{f}}$). The idea underlying this model is dual to the well-known approach of *interleaving.* Concurrent actions are not put into some order but are considered as single steps of the distributed system.

Distributed synchronous transition systems have been defined as a formal model of these systems and were equipped with a *Foata configuration graph*-based semantics, which provides a link between these systems and the framework of Mazurkiewicz traces. Foata configuration graphs can be understood as a kind of subgraph of the configuration graph of a trace.

We have given optimal decision procedures for satisfiability of $\mathrm{LTL}_{\mathrm{f}}$ formulas as well as for model checking, both based on alternating Büchi automata. The model checking procedure further employs an optimization which is similar to a technique known as *partial order reduction.* The complexity of both procedures is single exponential in the length of the formula. Thus, leaving out configurations which are obtained by a kind of interleaving in the formal study of trace configuration graphs pays back in terms of complexity. It would be interesting to identify the expressiveness of $\mathrm{LTL}_{\mathrm{f}}$.

Appendix A

Basic Notations, Notions, and Problems

A.1 Basic Notations

Within this whole thesis, every set is assumed to be countable. This general assumption is sometimes not made explicit. We call a set *countable* if there is a bijection to the set of natural numbers, which is denoted by $\mathbb{N}$, or if it is finite.

Let X be a (countable) set. The set of all subsets of X is called the *power set* of X and is denoted by 2^X. The *disjoint union* of two sets X and Y is the set $\{(1, x) \mid x \in X\} \cup \{(2, y) \mid y \in Y\}$ and is denoted by $X \uplus Y$. If $X \cap Y = \emptyset$, we identify $X \uplus Y$ with $X \cup Y$. For a fixed set Z, we denote the *complement* of $X \subseteq Z$ by $\overline{X}$, which is, of course, the set $\{z \in Z \mid z \notin X\}$.

As usual, we write the *binomial coefficients* $\frac{n!}{k!(n-k)!}$ as $\binom{n}{k}$, where n and k are assumed to be natural numbers with $n \geq k$. For a positive real number x, *floor* of x is denoted by $\lfloor x \rfloor$ and is the largest natural number less or equal x.

A.2 Graph-theoretic Notions and Notations

A *graph* $\mathcal{G}$ is a pair $\mathcal{G} = (Q, \leftrightarrow)$ where Q is an arbitrary set and $\leftrightarrow$ is a subset of $Q \times Q$ that satisfies $(q, q') \in\leftrightarrow$ iff $(q', q) \in\leftrightarrow$, for all $q, q' \in Q$. Instead of $\leftrightarrow$, we sometimes use $-$, especially when a graph is depicted. A *directed graph* $\mathcal{G}$ is a pair $\mathcal{G} = (Q, \rightarrow)$ where Q is an arbitrary set and $\rightarrow\subseteq Q \times Q$. Instead of $(q, q') \in\leftrightarrow$ or $(q, q') \in\rightarrow$, we usually write $q \leftrightarrow q'$ or $q \rightarrow q'$, respectively. We use $\rightleftarrows$ whenever we mean $\leftrightarrow$ or $\rightarrow$.

For a (directed) graph $(Q, \rightleftarrows)$, the elements of Q are called *nodes* of $\mathcal{G}$ and the elements of E are called *edges* of $\mathcal{G}$.

For $q, q' \in Q$, a *path* from q to q' is a sequence of nodes $q_0, \ldots, q_n \in Q$, $n \geq 1$, such that $q = q_0 \rightleftarrows \ldots \rightleftarrows q_n = q'$. A *cycle* in $\mathcal{G}$ is a path $q_0, \ldots, q_n$ such that $q_0 = q_n$. We say that a node $q \in Q$ *is contained* or *reached in* a cycle iff there is a cycle $q_0, \ldots, q_n$ in $\mathcal{G}$ and $q = q_i$ for an $i \in \{0, \ldots, n\}$. A *loop* is a cycle containing two nodes. For a directed graph $(Q, \rightarrow)$, we call $q' \in Q$ a *successor* of $q \in Q$, iff $q \rightarrow q'$.

A *component* of a graph $\mathcal{G}$ is a subgraph $\mathcal{G}'$ of $\mathcal{G}$ *induced by* a set of nodes Q', that is $\mathcal{G}' = (Q', \rightleftarrows')$ where $Q' \subseteq Q$, and $\rightleftarrows' = \rightleftarrows \cap (Q' \times Q')$. Sometimes, we call also Q' a component. A *(strongly) connected component* is a component $\mathcal{G}' = (Q', \rightleftarrows')$ of $\mathcal{G}$ such that for all $q, q' \in Q'$ there is a path from q to q' in $\mathcal{G}'$. A (connected) component and a cycle are called *non-trivial* if they contain a least two nodes.

A graph $(Q, \leftrightarrow)$ is called *complete* iff $\leftrightarrow = Q \times Q$. Similarly, the notion of a *complete component* is defined. A complete component of a graph is also called *clique*.

When we consider node-labeled graphs, especially graphs, whose nodes are labeled by formulas, we say that a cycle contains a formula φ, iff the cycle contains a node that is labeled by φ.

Often, we deal with so-called *pointed* graphs, that is, a graph $\mathcal{G} = (Q, \rightarrow)$ together with an *initial node* $q_0 \in Q$. A node is *reachable* in $\mathcal{G}$ iff there is a path from q_0 to q. For the sake of brevity, we often do not explicitly mention that the considered graph is pointed.

A.3 Graph-theoretic Problems

Let $\mathcal{G} = (Q, \rightarrow)$ be a directed graph. For $x, y \in Q$, REACHABILITY is the problem whether there is a path in $\mathcal{G}$ from x to y. Strictly speaking, REACHABILITY is a language that consists of all words that are an encoding of a graph together with two of its nodes x and y for which a path from x to y in $\mathcal{G}$ exists.

Theorem A.3.1
REACHABILITY *can be decided in linear time with respect to the size of a graph.*

Theorem A.3.2
REACHABILITY *can be decided in logarithmic space with respect to the number of nodes of a graph on a non-deterministic Turing machine.*

Proof
Suppose you want to check whether y is reachable from x via a path in the graph. Then y is reachable from x iff

- $x = y$ or
- there is a successor x' of x and y is reachable from x'.

Starting with x, the Turing machine first checks whether $x = y$ and if not, it guesses an appropriate successor x'. Since x is no longer important, the space needed for storing x can be reused. Now, the Turing machine proceeds with trying to find a path from x' to y.

Please observe that iff there is a path from x to y, then there is one with length at most $|Q|$. Thus, the machine can count the length of the path k and can stop in a state denoting non-acceptance, if n is obtained. To keep k, the current x', and y on the tape, a space of $3 \cdot \log n$ is used at most. Note that nodes are coded binary on the tape of the Turing machine, thus, a number k takes $\log k$ bits. $\square$

A.4 Notions for Relations

Let R be a binary relation over a set Q, that is $R \subseteq Q \times Q$. The *diagonal* of R is denoted by $\Delta(R)$ and is defined by

$$\{(q, q) \mid q \in R\}$$

For two relations R and R' over Q, we define their *product* by

$$\{(q, q') \mid q, q' \in Q, \exists q'' \in Q \text{ such that } (q, q'') \in R \text{ and } (q'', q') \in R'\}$$

and denote it by $R \circ R'$.

The *covering relation* of R or *cover* of R is the subrelation of R where two elements q and q' are only related, iff there is no third q'' with $(q, q'') \in R$ as well as $(q'', q') \in R$. In other words, the covering relation is the least relation such that its transitive closure is a superset of R. Stated differently, it is the relation obtained from R by removing all pairs which can be obtained by transitivity. Thus, the cover of R is defined by $R - (R \circ R)$. For a partial order $\leq$, we denote the covering relation of $\leq$ usually by $\lessdot$.

For a graph $\mathcal{G} = (Q, \rightleftarrows)$, the *Hasse diagram* of $\mathcal{G}$ is given by the graph $(Q, \rightleftarrows')$ where $\rightleftarrows'$ is the cover of $\rightleftarrows$.

A.5 Turing Machine Problems

In this section, we present a Pspace-complete problem. Recall that a problem is in Pspace, if it can be solved on a deterministic Turing machine using space bounded by a polynomial with respect to the size of its input. A problem is Pspace-complete, if it is in Pspace and every problem in Pspace cab be reduced to this problem.

A basic Pspace-complete problem is In-Place Acceptance: Given a deterministic Turing machine $\mathcal{M}$ and input x, does $\mathcal{M}$ accept x without ever leaving the $|x|+1$ first symbols of its strings?

Theorem A.5.1 ([Pap94], Chapter 19)
In-Place Acceptance *is* Pspace-*complete.*

Proof
It is easy to see that this problem is in Pspace: In linear space, we can simulate $\mathcal{M}$ on x, keeping a counter for the number of steps. We reject if the machine rejects. Furthermore, we reject if the machine tries to use more than the admitted space by adding a blank symbol $\Box$. Finally, we reject the input if the machine operates for more than $k = |Q||x||\Sigma|^{|x|}$ steps where Q denotes the set of states and Σ the working alphabet of the Turing machine. Note that k is the number of different configurations of a Turing machine using at most $|x|$ cells of the working tape. Thus, after k steps, the deterministic Turing machine would enter an infinite loop. Hence, rejecting the input in this case is the appropriate thing to do. Using a logarithmic encoding for this number, our constructed machine only uses space polynomially in the size of the input $|x|$.

We consider completeness: Suppose a language L is in Pspace, i.e., it is accepted by a Turing machine $\mathcal{M}$ in space n^k for some constant $k \in \mathbb{N}$. Obviously, $\mathcal{M}$ accepts x if and only if it accepts the string $x\Box^{n^k}$, the string consisting of x and n^k blanks. Hence, $x \in L$ iff $(\mathcal{M}, x\Box^{n^k})$ is decided positively by the previous procedure. Since $x\Box^{n^k}$ can be computed in polynomial time, we have reduced an arbitrary problem in Pspace to In-Place Acceptance. $\Box$

Bibliography

[ABP97] A. Ayari, D. Basin, and A. Podelski. LISA: A specification language based on WS2S. In M. Nielsen and W. Thomas, editors, *11th International Conference of the European Association for Computer Science Logic (CSL'97)*, volume 1414 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 1997.

[AMP98] R. Alur, K. McMillan, and D. Peled. Deciding global partial-order properties. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 41–52, 1998.

[APP95] R. Alur, D. Peled, and W. Penczek. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 90–100, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[BCPVD99] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In C. B. Weinstock and J. Rushby, editors, *Dependable Computing for Critical Applications—7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, pages 89–107, San Jose, CA, January 1999. IEEE Computer Society Press.

[BH93] E. Best and R. P. Hopkins. B(PN)^2 — A basic Petri net programming notation. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93 Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 379–390. Springer, June 1993.

[BH99] J. P. Bowen and M. G. Hinchey, editors. *Industrial-Strength Formal Methods in Practice.* Formal Approaches to Computing and Information Technology. Springer, 1999.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[BL80] J. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.

[BL01a] B. Bollig and M. Leucker. Deciding LTL over Mazurkiewicz traces. In C. Bettini and A. Montanari, editors, *Proceedings of the Symposium on Temporal Representation and Reasoning (TIME'01)*, pages 189–197. IEEE Computer Society Press, June 2001.

[BL01b] B. Bollig and M. Leucker. Modelling, specifying, and verifying message passing systems. In C. Bettini and A. Montanari, editors, *Proceedings of the Symposium on Temporal Representation and Reasoning (TIME'01)*, pages 240–248. IEEE Computer Society Press, June 2001.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, 1998.

[BPS01] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.

[Bru97] G. Bruns. *Distributed Systems Analysis.* Prentice Hall, 1997.

[Büc60] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.

[Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, pages 1–12, Stanford, CA, USA, 1962. Stanford University Press.

[BV94] J. C. M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 149–268. Oxford University Press, 1994.

[BVW94] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata–theoretic approach to branching–time model checking. In D. Dill, editor, *Proceedings of the 6th International Conference on Computer–Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 1994.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In F. M. auf der Heide and B. Monien, editors, *Proceedings of the 23th International Colloquium on Automata, Languages and Programming (ICALP'96)*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205, Berlin-Heidelberg-New York, 1996. Springer.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer.

[CF69] P. Cartier and D. Foata. *Problèmes combinatoires de commutation et réarrangements*. Number 85 in Lecture Notes in Mathematics. Springer, 1969.

[Cha99] K. C. Chang. *Digital Systems Design with "VHDL" and Synthesis*. IEEE Computer Society Press, 1999.

[CKS81] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1990.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[Ded69] R. Dedekind. Über Zerlegungen von Zahlen durch ihre grössten gemeinsamen Teiler. In R. Fricke, E. Noether, and Ö. Ore, editors, *Richard Dedekind. Gesammelte mathematische Werke*, volume II, pages 103–148. Chelsea Publishing Corporation, 1969. appeared 1897.

[DG00] V. Diekert and P. Gastin. LTL is expressively complete for Mazurkiewicz traces. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, volume

1853 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2000.

[DG01] V. Diekert and P. Gastin. Local temporal logic is expressively complete for cograph dependence alphabets. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'01)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 55–69. Springer, 2001.

[DM96] V. Diekert and Y. Métivier. Partial commutation and traces. Technical Report TR-1996-02, Universität Stuttgart, Fakultät Informatik, Germany, March 1996.

[DM97] V. Diekert and Y. Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, *Handbook on Formal Languages*, volume III. Springer, Berlin-Heidelberg-New York, 1997.

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, Singapore, 1995.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982.

[EL85] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press. Extended abstract.

[EM93] W. Ebinger and A. Muscholl. Logical definability on infinite traces. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (ICALP'93)*, volume 700 of *Lecture Notes in Computer Science*, pages 335–346, Lund, Sweden, 1993. Springer.

[EM96] W. Ebinger and A. Muscholl. Logical definability on infinite traces. *Theoretical Computer Science*, 154(1):67–84, January 1996.

[Fok00] W. Fokkink. *Introduction to Process Algebra.* Springer, 2000.

[GHJV00] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 2000.

[GHP97] J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors. *The Spin Verification System*, volume 32 of *DIMACS series*. American Mathematical Society, 1997.

[GMP98a] P. Gastin, R. Meyer, and A. Petit. A (non-elementary) modular decision procedure for LTrL. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, 1998.

[GMP98b] P. Gastin, R. Meyer, and A. Petit. A (non-elementary) modular decision procedure for LTrL. Technical report, LSV, ENS de Cachan, 1998. extended version of MFCS'98.

[HHI$^+$01] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7(2):213–236, jun 2001.

[HJJ$^+$95] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

[HM85] M. Hennessy and R. Milner. Algebraic laws for indeterminism and concurrency. *Journal of the ACM*, 32:137–162, 1985.

[Hoa85] C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.

[Jon75] N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, August 1975.

[Kam68] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968.

[Kat99] J.-P. Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32-1 of *Arbeitsberichte der Informatik*. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999.

[Kla91] N. Klarlund. Progress measures for complementation of omega-automata with applications to temporal logic. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, FoCS'91*, pages 358–367. IEEE Computer Society Press, 1991.

[Kle69] D. Kleitman. On Dedekind's problem: the number of monotone Boolean functions. In *5–th Proceedings of the American Mathematics Society*, volume 21, pages 677–682, 1969.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[KP91] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75(3):21–43, 1991. Preliminary versions appeared in 6th Annual ACM Symposium on Distributed Computing 1987, and in LNCS 398, Temporal Logic in Specification, 1988.

[Kro99] T. Kropf. *Introduction to Formal Hardware Verification.* Springer, 1999.

[KV97] O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. In *Proceedings of the Fifth Israel Symposium on Theory of Computing and Systems, ISTCS'97*, pages 147–158, Los Alamitos, California, 1997. IEEE Computer Society Press.

[KV00] O. Kupferman and M. Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[KV01] O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(3):408–429, 2001.

[KVW00] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Leu00] M. Leucker. On model checking synchronised hardware circuits. In J. He and M. Sato, editors, *Proceedings of the 6th Asian Computing Conference (ASIAN'00)*, volume 1961 of *Lecture Notes in Computer Science*, pages 182–198, Penang, Malaysia, November 2000. Springer.

[Leu02] M. Leucker. Prefix-recognizable graphs and monadic second order logic. In W. Thomas, T. Wilke, and E. Grädel, editors, *Automata, Logics and Infinite Games*. Springer, 2002. to be published.

[LN01] M. Leucker and T. Noll. Truth/SLC - A parallel verification platform for concurrent systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 255–259. Springer, July 2001.

[LT00] C. Löding and W. Thomas. Alternating automata and logics over infinite words. In *Proceedings of the IFIP International Conference on Theoretical Computer Science, IFIP TCS2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 521–535. Springer, August 2000.

[Maz77] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[Maz88] A. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer, June 1988.

[McM92] K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words. CNET, Paris, 1988.

[Mil80] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mol92] F. Moller. *The Edinburgh Concurrency Workbench (Version 6.1)*. Department of Computer Science, University of Edinburgh, October 1992.

[Mot93] Motorola, editor. *The PowerPC (TM) 601 User's Manual*. Motorola, 1993.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[MS87] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[MSS86] D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of the tree, and its complexity. In *Proceedings of the 13th International Colloquium on Automata, Languages and Programming, ICALP'86*, volume 226 of *Lecture Notes in Computer Science*, pages 275–283. Springer, 1986.

[MT96] M. Mukund and P. S. Thiagarajan. Linear time temporal logics over Mazurkiewicz traces. *Lecture Notes in Computer Science*, 1113:62–92, 1996.

[Muk92] M. Mukund. Petri nets and step transition systems. *IJFCS: International Journal of Foundations of Computer Science*, 3, 1992.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994.

[Pel98] D. Peled. Ten years of partial order reduction. In *Proceedings of 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28, Vancouver, BC, Canada, 1998. Springer.

[Per91] D. Perry. *VHDL.* McGraw-Hill, New York, 1991.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, September 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE Computer Society Press.

[PP95] D. Peled and W. Penczek. Using asynchronous Büchi automata for efficient model-checking of concurrent systems. In *Protocol Specification Testing and Verification*, pages 90–100, Warsaw, Poland, 1995. Chapman & Hall.

[PWW96] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of ω-regular languages. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 596–610, Pisa,Italy, 1996. Springer.

[PWW98] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of ω-regular languages. *Theoretical Computer Science*, 195(2):183–203, 1998. A preliminary version appeared in [PWW96].

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, New York, 1982. Springer.

[Rei86] W. Reisig. *Petrinetze.* Springer, 2 edition, 1986.

[Roh97] S. Rohde. *Alternating automata and the temporal logic of ordinals.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[RV01] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier, 2001.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, FoCS'88*, pages 319–327, Los Alamitos, California, October 1988. IEEE Computer Society Press.

[SC82] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 159–168, San Francisco, California, 5–7 May 1982.

[SC85] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[SNW96] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1–2):297–348, 15 December 1996.

[SP99] P. Stevens and R. Pooley. *Using UML: software engineering with objects and components.* Object Technology Series. Addison-Wesley Longman, 1999. Updated edition for UML1.3: first published 1998 (as Pooley and Stevens).

[Sti01] C. Stirling. *Modal and Temporal Properties of Processes.* Texts in Computer Science. Springer, 2001.

[SVW85] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic (extended abstract). In W. Brauer, editor, *Automata, Languages and Programming,* *12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 465–474, Nafplion, Greece, 15–19 July 1985. Springer.

[TH98] P. S. Thiagarajan and J. G. Henriksen. Distributed versions of linear time temporal logic: A trace perspective. *Lecture Notes in Computer Science*, 1492:643–681, 1998.

[Thi94] P. S. Thiagarajan. A trace based extension of linear time temporal logic. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 438–447, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[Thi95] P. S. Thiagarajan. A trace consistent subset of PTL. In I. Lee and S. A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *Lecture Notes in Computer Science*, pages 438–452, Philadelphia, Pennsylvania, 21–24 August 1995. Springer.

[Tho79] W. Thomas. Star-free regular sets of ω-sequences. *Information and Control*, 42(2):148–156, August 1979.

[Tho90a] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[Tho90b] W. Thomas. On logical definability of trace languages. In V. Diekert, editor, *Proceedings of a workshop of the ESPRIT Basic Research Action No 3166: Algebraic and Syntactic Methods in Computer Science (ASMICS)*, TUM-I9002, pages 172–182, Kochel am See, Bavaria, 1990. Technical University of Munich.

[Tho99] W. Thomas. Complementation of Büchi automata revisited. In J. Karhumäki et al., editors, *Jewels are forever – Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 109–122. Springer, 1999.

[TS01] U. Tietze and C. Schenk. *Halbleiter-Schaltungstechnik*. Springer, 11 edition, 2001.

[TW97] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 183–194, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.

[Val91] A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Berlin, Germany, June 1991. Springer.

[Var96] M. Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, New York, NY, USA, 1996.

[Var01] M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, April 2001.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS'86)*, pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.

[Wal] I. Walukiewicz. Difficult configurations – on the complexity of LTrL. *Formal Methods in System Design.* to appear.

[Wal98] I. Walukiewicz. Difficult configurations - on the complexity of LTrL. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 140–151, 1998.

[Zie87] W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987.

Symbol Index

Index

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult `http://aib.informatik.rwth-aachen.de/` or send your request to: `biblio@informatik.rwth-aachen.de`

99-01 *	Jahresbericht 1998
99-02 *	F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
99-03 *	R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager
99-04	M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
99-07	Th. Wilke: CTL+ is exponentially more succinct than CTL
99-08	O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 *	Jahresbericht 1999
2000-02	Jens Vöge / Marcin Jurdziński: A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-04	Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 *	Mareike Schoop: Cooperative Document Management
2000-06 *	Mareike Schoop, Christoph Quix (Ed.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 *	Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08	Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation free μ-calculus
2001-05	Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem

2001-08	Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark and Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

Lebenslauf

Name	Martin Leucker
Geburtsdatum	16.05.1971
Geburtsort	Kamp-Lintfort

Bildungsgang

1977 – 1981	Besuch der Grundschule am Niersenberg (Kamp-Lintfort)
1981 – 1990	Besuch des Gymnasiums Kamp-Lintfort Abschluß: allgemeine Hochschulreife
Okt. 1990	Beginn des Studiums der Mathematik (mit Nebenfach Informatik) an der RWTH Aachen
Okt. 1991	Beginn des Studiums der Informatik (mit Nebenfach Mathematik) an der RWTH Aachen
Aug. 1992	Vordiplom in den Fächern Mathematik und Informatik
26.04.1996	Beendigung des Studiums der Mathematik mit Abschluß Diplom in Mathematik mit Nebenfach Informatik
seit Mai 1996	Beschäftigung als wissenschaftlicher Angestellter am Lehrstuhl für Informatik II (Prof. Dr. K. Indermark) an der RWTH Aachen