CONTROL AND SYNTHESIS OF OPEN REACTIVE SYSTEMS

Thesis submitted in
partial fulfillment of the
Degree of Doctor of Philosophy (Ph.D)

by

P. Madhusudan
Theoretical Computer Science Group,
Institute of Mathematical Sciences,
Taramani, Chennai–600 113.

UNIVERSITY OF MADRAS
Chennai 600 005

November 2001

DECLARATION

I declare that the thesis entitled **"Control and Synthesis of Open Reactive Systems"** submitted by me for the Degree of Doctor of Philosophy is the record of work carried out by me during the period from August 1996 to November 2001 under the guidance of Dr. R. Ramanujam and has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other University or other similar institution of higher learning.

November 2001 P. Madhusudan

The Institute of Mathematical Sciences
C.I.T. Campus, Tharamani
Chennai (Madras), Tamilnadu - 600 113

CERTIFICATE

This is to certify that the Ph.D. thesis submitted by P. Madhusudan to the University of Madras, entitled **Control and Synthesis of Open Reactive Systems**, is a record of bonafide research work done during the period 1996–2001 under my supervision. The research work presented in this thesis has not formed the basis for the award to the candidate of any Degree, Diploma, Associateship, Fellowship or other similar titles.

It is further certified that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of problems dealt with.

R. Ramanujam
Thesis Supervisor

November 2001

The Institute of Mathematical Sciences
C.I.T. Campus, Tharamani
Chennai (Madras), Tamilnadu - 600 113

Abstract

The topic of this thesis is the study of the problem of automated synthesis of controllers and systems against formal specifications. Our two central aims are to study these control problems for branching-time specifications and to study them to achieve distributed control.

We start this study by considering the control synthesis problem for simulations and bisimulations. We show that one can solve this in polynomial time. Moreover, whenever a controller exists, we show how to automatically synthesize a polynomial-state controller in polynomial time.

We then study the control synthesis problem for asynchronous simulations and show the surprising result that it is undecidable. In fact, we show that even the associated verification problem is undecidable in this setting. The undecidability results extend even to very simple classes of concurrent systems.

The control and realizability problems for the branching-time temporal logics CTL and $\mathsf{CTL}^\star$ are studied next. It turns out that one can study the problem in two settings — one where the environment is static and universal, the other where it is reactive. The control problem for universal environments reduces to module-checking and hence is, for CTL and $\mathsf{CTL}^\star$, $\mathsf{EXPTIME}$-complete and 2-$\mathsf{EXPTIME}$-complete, respectively [KV96]. We show that the complexities of these problems in reactive environments become exponentially harder — they are 2-$\mathsf{EXPTIME}$-complete for CTL and 3-$\mathsf{EXPTIME}$-complete for $\mathsf{CTL}^\star$.

We also investigate the control-synthesis problem in a distributed setting, where the processes communicate with each other in a synchronous fashion and also interact with their local environments according to an architecture. This model is the one studied in [PR90], and from the results therein it follows that for global specifications, the only decidable architectures are the singly-flanked pipelines. We study the control problem for *local specifications* and show that the class of decidable architectures (mildly) increases. We characterize the exact class of architectures for which the control problem is decidable for local specifications — this is the class of architectures where each connected component is a sub-architecture of a doubly-flanked pipeline.

Acknowledgements

I would like to thank P.S. Thiagarajan for his guidance and encouragement through the last six years. All the work presented herein have been done with him, and in doing so I have learnt a lot from him. He has inculcated in me the importance of rigour for which I will be forever grateful.

I would like to thank R. Ramanujam for his unfailing support, encouragement and advice. Discussions with him in matters academic and otherwise have always had the effect of making me think afresh of things I had prejudged without serious thought.

My father was the main reason why I joined academics and mathematics, in particular. He was a wonderful source of inspiration and he induced a fascination for mathematics in me early in childhood. His encouragement led me to join Matscience, which has changed my life so much. I thank him heartily for his caring early guidance.

The theoretical computer science groups at Matscience and CMI were absolutely wonderful in the way they welcomed me into their community and taught me all that I know in computer science. The subjects of discourse were extremely widespread and the teaching impeccable. I am very grateful to them — Meena Mahajan, V. Arvind, P. S. Thiagarajan, Manindra Agrawal, Kamal Lodaya, Venkatesh Raman, R. Ramanujam, Madhavan Mukund, Anil Seth, K. V. Subrahmanyam, Narayan Kumar, and Srinivasa Raghavan.

Any student who joins these people gets an in-depth view into the various fields of research — I haven't seen this range and depth in any other place. The more I discover groups across the world working in TCS, the more I realize what a superb education these people have given me. I thank them for this.

Appa, amma and Hrishi have been a constant support to me and have been extremely indulgent with the various strange moods I have subjected them to.

Radhika Vathsan has been a close friend through the years — I cherish her friendship and the memories of those deep discussions from stars to life to philosophy we had in my earlier days at Matscience.

Deepak D'Souza has been a friend and a colleague from when we joined for Ph.D. together. It has been fun to pick threads together and work our way through the course-work and later research. His clear mind and his love for rigour have many a time mellowed my wild leaps, while his gentle nature is wonderful to have in a friend.

Meenakshi has also become a close friend in the last couple of years. Working with her (in topics not related to this work) has been pleasant and rewarding.

I spent a memorable nine months at Aachen under Wolfgang Thomas. He was the perfect host and the warmth he extended made me feel completely at home. I learnt a lot from his incredible knowledge during my stay there and I am grateful to him for hosting me and making my stay so rewarding.

The stay at Aachen was extremely pleasant due to other friends there as well — Christof Löding, Dietmar and Andreea Berwanger, and Jens Vöge. I thank Christof especially for the gruelling proofs I made him listen to in my desperate attempts to solve the distributed control problem. He was patient, bore all the vague arguments I threw at him and was very helpful in his remarks and observations. Olivier Carton and Elizabeth Gonçalves were charming and I have pleasant memories of touring the country-side with them, exploring the foreign land we found ourselves in.

I thank Orna Kupferman and Moshe Vardi with whom part of the work here was done in collaboration with.

My thesis writing was many a time, so to speak, “snarked”. The computer science group showed extreme patience and at the same time prodded me to finish it. I would like to thank R. Balasubramanian, Director, IMSc, for allowing me to stay beyond my term and C.S. Seshadhri, Director, CMI, for supporting me for the period of thesis-writing after my term at Matscience.

I thank Thiagarajan and Meenakshi heartily for proof-reading my thesis. Deepak and Hrishi also helped in reading drafts of chapters.

I have learnt a lot from several of my academic colleagues and fellow-students — apart from those mentioned above, these include Barbara Sprick, A. Srinivasan, Samik Sengupta, Swarup Kumar Mohalik, S.V. Nagaraj, N.V. Vinodchandran, Srinivasa Rao, S. P. Suresh and Piyush P. Kurur.

My stay at Matscience has been very pleasant. There were many people who made this stay memorable — people I've spent those long evenings with, from whom I've learnt things or who have made a deep impact on me. I wish to recall them — Tabish Qureshi, Mary Selvadoray, Shubashree Desikan, Sharmila Bagchi, A.R. Krishnan, Nandini Singh née Chatterjee, Katya Balasubramaniam, Suneeta Varadarajan, K.R.S. Balaji, P. Jyoti, Gyan Prakash and Catriona Maclean.

Contents

Chapter 1

Introduction

"Just the place for a Snark!" the Bellman cried,
As he landed his crew with care;
Supporting each man on the top of the tide
By a finger entwined in his hair.

— *The Hunting of the Snark*, Lewis Caroll

The last fifty years have seen a tremendous influx of digital systems into our day-to-day lives — they are used in communications, control of industrial equipment, avionics, managing safety-critical equipment such as reactors, in security systems and even in digital gadgets inside cameras and washing machines.

One of the main concerns of the past three decades in computer science has been the development of correct software. The most popular tool in the software engineering industry has been *testing*. Though very useful, it has been found inadequate because it can give no guarantee of the correctness of code. The failure of many systems due to bugs in code, especially in safety-critical software where such bugs can result in tremendous losses to life and property, has led people to study the problem of formally verifying software. Mathematicians have over the years studied this problem and have come up with various methods such as formal theorem-proving where one gives formal semantics to programs and specifications and proves that a program meets its specification by giving a proof very similar to proofs in mathematics [Hoa69].

The last twenty years have seen a tremendous increase in the complexity of systems built in the industry. Formally proving such programs correct is too daunting a task and consequently, has found few takers in the industry. However, there have been approaches to program verification via theorem-proving with the aid of machines. These theorem provers are built to find proofs, but, in most cases seek human guidance (see [Fit96]). They are not fully automated and hence are not very popular in the industry.

The classical notion of computation has been that of recursive functions or the mechanism of Turing machines that compute functions: they take an input, process it, and output a result. However, there are many programs, such as resource schedulers in operating systems, communication protocols, etc., that do not behave this way. They are usually simple systems that are characterized by an ongoing indefinite interaction with an environment. A resource scheduler continuously takes requests and termination signals, and reacts to such input continuously. These systems are never meant to terminate in some bounded time and hence their behaviours are best viewed as infinite sequences of events. Such systems are termed *reactive systems.*

While specifying properties of classical programs is usually done through the mechanism of first-order logic augmented with program constructs, for a long time it was not clear what a suitable specification mechanism for reactive systems might be. However, the specification mechanisms using *temporal logics*, proposed for such behaviours in [Pnu77, GPSS80, MP81, HP85], have over the years emerged the most popular.

A methodology that has been developed for such finite-state reactive systems is the *automated* verification of programs against their specifications by state-space exploration. These methods are best suited for finite-state systems, or those that can be abstracted so that they are finite-state, and typically include hardware circuits, communication protocols, user-interfaces for machinery, etc. The specification mechanisms usually handled are temporal logic specifications, for which this process can indeed be completely automated and requires no human intervention [LP85, SC85, Pnu85] (see [Eme90] for a survey). Such state-space exploration mechanisms are termed “model-checkers” and constitute an important and useful class of methods for verification. Model-checking, though it had its beginnings in simple state-space exploration methods, can be quite sophisticated — rather than thinking of it as state-space exploration, it is best viewed as any verification problem that is *decidable* and can be solved without human intervention.

Of course, digital systems are not the only systems we live with. Many systems that we deal with are physical machines that have analog components that evolve continuously with time. Scientists and engineers have worked together to harness natural phenomena and used them to make machines work. The study of building such physical systems in order to achieve a desired behaviour is called "control-theory" — this includes most non-digital systems (e.g.: automobile machinery, industrial manufacturing equipments, missile launching mechanisms, flight control, audio and video playback gadgets, etc.)

Since such systems are usually modelled using continuous variables such as displacement, velocity, etc., the main tools to analyse and control them use the mathematics of ordinary and partial differential equations. However, with the increased role of computers and computer-controlled devices, many control-mechanisms have to deal with systems modelled using discrete variables. For example, control of a part of a telephony equipment may only involve variables that record the number of telephone calls active in a region, the set of sectors that are under repair, etc. and not the exact voltage running across various lines. Another feature that these applications of control-theory possess is that they are driven by instantaneous events from the environment — e.g. the press of a keyboard key or a sensor turning on. Systems that possess these two properties above are called "discrete-event systems" and have been a topic of recent study (see [RW89, CL99]). Most instances of control systems that have some interaction with digital computers, such as communication networks, manufacturing facilities, software-controlled hardware devices, etc., come under this class.

In most of these applications, the exact times at which events occur are not very important and there are enough interesting properties one would want to verify based just on the *order* in which the events occur.

Discrete event systems (DES) form a small subclass of systems studied in control-theory. They are dynamic systems (the behaviour of the system depends not only on the current input to it but also on the history of past inputs), time-invariant (the behaviour does not depend on the exact times at which events occur), have a discrete state-space model and are event-driven (they change state only when there is an external instantaneous event) [CL99].

While model-checking is a useful tool to both verify finite-state systems in computer science (like hardware circuits, communication protocols, etc.) and analyze finite-state DES (like a controller for a telephony equipment), a more ambitious question is to ask whether we can *synthesize* such systems. What we would like to do is to fix a prototype of a model that describes exactly how the system and environment interact, and given a specification of the desired behaviour of the system, ask whether one can automatically come up with a system that satisfies the specification. This problem is also known as the *realizability problem*.

This question was first posed by Church in 1963 in the context of synthesizing switching circuits against specifications stated in restricted second-order arithmetic [Chu63]. Büchi and Landweber in [BL69] showed that this realizability problem is decidable, even for a more powerful class of specifications (S1S). The realizability problem can be viewed as a game where the system and the environment play with each other by choosing event-labels (say alternately) and thereby build an infinite sequence of events. The event-sequence is winning for the system in case this sequence is recognized as a desired behaviour according to the specification. If not, the environment wins the play. The realizability problem then boils down to deciding whether the system has a winning strategy for this game, i.e. a strategy such that no matter how the environment plays, the system always wins. If there is indeed a winning strategy, then finding one that uses only a bounded memory of the past gives rise to finite-state programs that realize the specification.

The proof in [BL69] (see also [Tho95]) indeed used game-theoretic techniques, but was very complex and Rabin's later proof of the same theorem was welcome [Rab72]. Rabin's theorem used automata over trees and it is interesting to note that, though we are verifying only the sequential behaviours of systems, the realizability problem leads us to work with trees. However, since emptiness of tree-automata can be seen as a game over finite graphs, the connection is, in retrospect, natural.

The area met a revival in the eighties in the works of [MW80, EC82, MW84] where the automated synthesis of finite-state programs against temporal logic specifications was considered. However, these papers dealt with programs that do not have an environment to interact with — they are *closed systems* where everything that happens to the system can be controlled. In fact, these papers solve the satisfiability problem for temporal logics, show how to come up with a finite-state witness for satisfiability, and how to extract a finite-state program from this. But the important aspect of reactive systems is that there is an interaction with an environment which the system has no control over. Hence the interesting problem is to come up with a program that satisfies the specification no matter how the environment behaves (as in [BL69, Rab72]). To reiterate the fact that such systems have no control over the environment, these systems are called *open reactive systems* and are the main objects of study in this thesis.

The study of synthesis for open reactive systems was taken up later in [ALW89, PR89a]; the emphasis of [PR89a] was on the complexity of synthesizing systems against the linear-time temporal logic LTL using automata-theoretic techniques. Meanwhile, due to the interest in simplifying Rabin's theorem [Rab69], considerable work simplifying the study of infinite games on infinite graphs had been developed [Tho95, Zie98]. An offshoot of this, which is simpler, is the study of infinite games played on finite graphs [McN93], which turns out to be directly connected with synthesis. In fact, this is probably the best setting to understand the synthesis and control results implied in [PR89a].

A problem related to synthesis studied in the area of control-theory is that of *control-synthesis*. The general scenario is that there is a system, called a *plant* in this context, which interacts with an environment. The goal is to design a controller which will interact with the system, observing and controlling it using its own inputs to it (thereby exerting *dynamic feedback control*), in order to make the system behave in a desired way. For example, the control-mechanism for a car might involve the system getting inputs from sensors of the accelerator and brakes and the controller must ensure that the car behaves in the desired manner by issuing commands to the parts in the machinery that control the engine power and brake mechanisms. Such control is in many cases a continuous one, where the controller's inputs may also be continuous.

In discrete-event systems, the control mechanism is also discrete. The automatic generation of a controller to meet a specification for such systems was initiated by Ramadge and Wonham [RW89]. For example, consider a lift mechanism — it has two levels of control. One is the continuous physical controller that controls the lifting of shafts, the power of motors, etc. in order to raise or lower the lift. On the other hand, there is also a high-level controller, which interacts with the user-panel, observes various sensor mechanisms and controls the behaviour of the lift.

This high-level control mechanism works on an abstracted system which is a DES, and simply issues commands such as ordering the closing of a door, movement to a particular floor, etc. The underlying continuous low-level control makes sure that the lift actually performs these tasks.

The high-level control (also called supervisory control) is hence discrete in nature and is amenable to automated synthesis. The control-synthesis problem then is to synthesize a controller for a given plant modeled as a DES. In [RW89], this problem was studied in an automata-theoretic framework and was shown to have reasonable sub-classes where the problem is decidable.

The supervisory control of DES has been fairly well studied in the recent past. The problems studied in this community usually focus on issues like partial observability (where the controller has only a limited power of observing the plant), supremal controllers (controllers that pose the least restriction on the system), decentralized control (where there are many controllers, each having access and control of one part of the system), etc. [KG95, KGM91, KS95, KS97, WW96].

The literature on supervisory control, however, seldom deals with specifications that are given *externally*. The specification is usually stated in terms of the plant itself — for example, as certain states that must be reached or must be avoided when the plant is modelled as a transition system. The control-problem in these cases reduces to searching for certain structures in the state-space, and are usually easy to handle, give rise to minimally restrictive controllers and are solvable in polynomial time. The control-problem where specifications are given independent of the plant are certainly harder to handle and is the main focus in the computer-science literature.

Another difference in approach is that the control-theory community has focussed on various classes of simple problems where one can obtain tractable control-synthesis schemes, while in computer science, people have asked more general questions and proved many undecidability and lower bound complexity-theoretic results. For example, while distributed control has been shown, in a general setting, to be undecidable [PR90], a notion of "co-observability" has been defined which is a sufficient condition under which one can achieve decentralized control [CL99].

The control-synthesis and the realizability problems are technically very similar. While in the former one is given a plant which has to be controlled to meet a specification, the latter is to come up with a program (which can be seen as controlling a plant which allows all possible behaviours) that meets the specification. The theme of this thesis is the synthesis and control of open reactive systems (which could be discrete-event systems) for various models and specifications.

Contributions of the thesis

We study the control and synthesis problems for finite-state reactive systems modeled as discrete transition systems. We do not consider partial-observation — the controllers in all cases will have complete information of the system (or the part of the system) that they control. The two main sub-themes are to study these problems for *branching-time specifications* and to study *distributed control*.

The most common kind of specification is one where the set of desired behaviours is described as the set of desired *sequences* of the plant. An alternative way of looking at the behaviour of a plant is to consider the *tree* representing the set of all sequential behaviours it can exhibit. The behaviour of a program/plant is then a single tree, the branches of which give its sequential (i.e. linear-time) behaviours. However, a specification of this tree can be more expressive than one that describes just the sequential behaviours. A typical example that such a branching-time specification can state, which cannot be stated in the linear-time framework, is one which demands that no matter how the plant evolves, it must always be *possible* to extend this behaviour to one that does a particular action. In fact, in [RW89], the supervisory control problem is framed by using a set of marked states (that represent completion of tasks) and the specification demands that the controlled plant must, after any sequence of moves, be in a state such that a marked state is reachable (this is called the "nonblocking property"). This property is also inherently a branching-time property.

A simple mechanism to study branching-time properties is through the notion of simulations. We start our study by considering the problem of control where the specifications are also given as transition systems, and we want to control the plant such that the specification can simulate the controlled plant. We show that this problem is decidable in time polynomial in the sizes of the finite plants and specifications. Simulations allow us to capture simple safety properties and the results regarding this form the first half of Chapter 2.

An important aspect of the control-synthesis problem that we consider is that the controller can use an unbounded amount of information of past interactions with the environment in order to control moves. The class of controllers amongst which we search is hence an infinite collection, and hence the decidability problem is not trivial.

We also study the control problem for bisimulations in Chapter 2, which is a stronger notion than that of simulations: two systems are bisimilar if they can simulate each other in a tight fashion. We study the problem of control-synthesis where we are given a plant and a specification, both modelled as finite-state transition systems and are asked whether there is a controller for the plant such that the controlled plant is bisimilar to the specification. Again, we show that the control problem is decidable in polynomial time. In both settings we show that when a controller exists, we can synthesize a controller of polynomial size as well.

In Chapter 3, we consider a concurrent variant of the control problem for simulations. We use *asynchronous transition systems*, which are transition systems augmented with concurrency information, and consider a natural notion of simulation between them. Given a plant and a specification, both modelled as finite asynchronous transition systems, the problem is to come up with a controller for the plant such that there is an *asynchronous simulation* from the controlled plant to the specification. Surprisingly, it turns out that the control-synthesis problem in this simple setting is undecidable. In fact, we show that even the corresponding model-checking problem — given two asynchronous transition systems, the problem of verifying if there is an asynchronous simulation from one to the other — is undecidable. We show that these results hold for even very restricted classes of asynchronous transition systems. These results show how complex any notion of distributed control can become.

In Chapter 4, we introduce the various temporal logics we will need in later chapters (namely LTL, CTL and $\mathsf{CTL}^\star$), and also the notions of trees, and nondeterministic and alternating automata working on them. While it is true that the controller-synthesis and realizability problems in a non-distributed setting is perhaps best understood in terms of infinite games on finite graphs [McN93, Tho95, Zie98], automata over trees allow combining strategies by suitable operations on automata that describe them. Alternating automata has been a very important tool in control and synthesis [KV96, KV97a, KV99a, KV00, KV01] and is the main tool in our technical arsenal.

In Chapter 5 we turn to the design of programs and controllers for systems against the branching-time temporal logics CTL and CTL*. In this branching-time setting, it turns out that one can study the problems in two scenarios — one where the environment is static and universal and offers all possible moves at every point (this does not mean that the system is not open) and the other where the environment can offer different subsets of moves at different stages. The environments of the former scenario are called *universal* or *non-reactive environments* while those of the latter are termed *reactive environments*. Since branching-time specifications can specify *possibility* requirements (like "it must always be possible that action 'a' occurs"), a program which satisfies a requirement when interacting with a universal environment need not satisfy it in the context of a reactive environment. In fact, we show that the problem under reactive environments is a harder one to solve.

It turns out that the control and realizability problems of systems in universal environments technically reduces to the *module-checking* problem studied by Kupferman and Vardi in [KV96, KV97a]. From the results on module-checking, it follows that these problems for CTL and CTL* are EXPTIME-complete and 2-EXPTIME-complete respectively. (In [KV00], the authors extend this to the μ-calculus as well.)

Our main result is that the control and realizability problems in the context of reactive environments is solvable for CTL and CTL*, and are 2-EXPTIME-complete and 3-EXPTIME-complete, respectively. We show that when controllers exist, one can synthesize controllers whose sizes match the above time-bounds. The upper bounds are proved using automata-theoretic techniques and interestingly, use the fact that the class of languages accepted by tree automata are closed under complement. Our lower bound results justify this costly step of complementation and perhaps suggest that this is where the study of control-synthesis in terms of games on finite-graphs breaks down.

We then turn to the problems of distributed control synthesis and distributed realizability for linear-time specifications in Chapter 6. The most relevant paper that addresses this problem is [PR90], where the authors study the distributed realizability problem in a setting where processes communicate with each other, work synchronously and have possible local environments that they interact with. Pnueli and Rosner show that the realizability problem is undecidable for almost all architectures and from their results it follows that the only architectures for which the control problem is decidable are the singly-flanked pipelines. Singly-flanked pipeline are architectures where the processes are connected along a line with internal channels, and the first process in the line is the only one that interacts with the environment.

The main departure of our work from that of [PR90] is that we consider local specifications instead of global ones. We show that local specifications do increase the class of architectures for which the control-problem is decidable, but that this is only mild. We characterize the exact class of decidable architectures — the control problem for an architecture is decidable iff each connected component of it is a sub-architecture of a doubly-flanked pipeline (a doubly flanked pipeline is like the singly-flanked pipeline as described above with the only difference being that the processes at *both ends* of the pipeline can interact with the environment).

The research that this thesis is based on was done mainly in cooperation with P.S. Thiagarajan and partly in cooperation with Orna Kupferman and Moshe Vardi. The work on simulations and the undecidability results for asynchronous simulations were first published in CONCUR'98 [MT98a]. The journal version of the above will appear in Theoretical Computer Science [MT01a] and includes the results on bisimulations. Results on the control and realizability problems against reactive environments for branching-time logics appeared in CONCUR'00 [KMTV00a]. The results on distributed control for local specifications were published in ICALP'01 [MT01b]. Technical reports written on these works are [MT98b] and [KMTV00b].

Chapter 2

Control-synthesis for simulations and bisimulations

They sought it with thimbles, they sought it with care;
They pursued it with forks and hope;
They threatened its life with a railway-share;
They charmed it with smiles and soap.

— *The Hunting of the Snark*, Lewis Caroll

2.1 Introduction

In this chapter, we study the problem of synthesizing controllers for discrete event systems by considering the *branching-time* specification mechanisms of *simulations* and *bisimulations*.

First, let us recall the general problem of control synthesis. In informal terms, one is given an open discrete event system called a plant which consists of a system and its environment. One then specifies the desired patterns of interaction between the system and its environment. The problem is to find a controller which will restrict the behaviour of the plant in such a way that the controlled behaviour meets the specification. A characteristic feature of the controller is that it is allowed to restrict only the actions of the system (and not those of the environment). Typical questions that arise are:

- Given finite descriptions of the plant and the specification is it decidable that there exists a controller ?
- In case there is a controller is it always the case that there is a finite controller?
- How big need the controller be — in case it exists — relative to the sizes of the plant and the specification?

A substantial amount of knowledge is available about this problem in the linear time framework, i.e. when the specification describes properties of the *sequences* generated by the plant. In this chapter, the key point of departure is that we study the controller synthesis problem in a branching time setting. We uniformly describe both plants and specifications as certain kinds of labelled transition systems. We then advocate the use of simulations and bisimulations to capture the requirement that the plant-controller combination meets its specification. As a result, behavioural properties that can only be stated in a branching time setting become available as specifications (see [LV95]). Though simulations are a weak way of specifying the required behaviours of a system, it is a good starting point for the study of branching-time specifications and allows us to state simple safety properties.

In this chapter, we show that the problem of checking for the existence of a controller, for specifications formulated using simulations and bisimulations, can be solved in polynomial time. Moreover, if a controller exists, we show that a controller of polynomial size can be synthesized in polynomial time.

In the next section we formulate the model for the plant using transition systems with two layers of labelling on the transitions. This turns out to be a convenient way of capturing the usual two-person game associated with the plant as well as the plant-controller interaction. We use the same class of transition systems to capture specifications. We then define simulations, which are behaviour preserving homomorphisms, in the natural way. A controller is then required to restrict the system's actions so that the restricted behaviour of the plant can be related to the specification via a simulation.

An important lesson derived from existing literature is that a richer class of controllers can be obtained by allowing the controller to make use of memory of the past to achieve its goal. The controllers we consider are also transition systems and work in tandem with the plant, restricting its moves in various ways. The controller can have arbitrarily large state-spaces. Hence it can prescribe different moves at the same state of the plant if the history of moves executed so far are different. Clearly the set of controllers for a plant is an infinite collection even when both the plant and specification are finite objects. Consequently the task of deciding the existence of a controller is not trivial.

In Sections 2.3 and 2.4 we show that the problem of deciding if a pair of finite systems (a plant and a specification) admits a controller is decidable in time which is polynomial in the sizes of the plant and specification. We also show that the size of the controller, whenever one exists, can be bounded from above by a similar polynomial. A point worth noting here is our transition systems are deterministic with respect to an alphabet of events. But the events will have an additional layer of action labels and the simulations are required to preserve only action labels. Consequently the domain of a simulation relative to the action labels will be, in all non-trivial instances, non-deterministic.

In Section 2.5 we extend the techniques of the previous two sections to tackle the case of bisimulations. To our knowledge, bisimulations have never been considered as a specification mechanism in the supervisory control problem, though it has been used as a technique to solve the classical controller synthesis problem [BL97]. Surprisingly, the time complexity and the size of the controller (when one exists) still have polynomial upper bounds. It turns out that a crucial computational step in the decision procedure can be efficiently reduced to a maximal matching problem which is known to be solvable in polynomial time [EK70, CLR92].

The proofs of results regarding simulations, though they are simple, are presented in good detail in order to give a gentle introduction to the problem. As we go on, for example when dealing with bisimulations, we give proofs in fair detail but obvious details have been left out so that the flow of the arguments is not affected.

2.2 The model

It will be convenient to work with deterministic transition systems that have an additional layer of labelling. Through the rest of this chapter we fix a finite set of *actions* (or *labels*) Σ and let a, b range over Σ.

Definition 2.1 A Σ-labelled deterministic transition system is a structure $TS = (Q, E, T, q_{in}, \varphi)$ where:

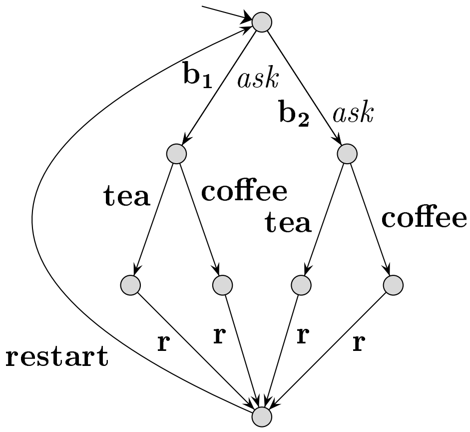
- Q is a (possibly infinite) set of states.
- E is a finite set of events.
- $T \subseteq Q \times E \times Q$ is a deterministic transition relation. In other words, if $(q, e, q') \in T$ and $(q, e, q'') \in T$ then $q' = q''$.
- $q_{in} \in Q$ is the initial state.
- $\varphi : E \longrightarrow \Sigma$ is a labelling function. $\Box$

Let $t = (q, e, q') \in T$. We often write $q \xrightarrow{e} q'$ instead of $(q, e, q') \in T$. Sometimes we write $q \xrightarrow[a]{e} q'$ to indicate further that $\varphi(e) = a$. In all such cases the concerned transition system will be clear from the context.

Henceforth in this chapter, we refer to Σ-labelled deterministic transition systems as just transition systems.

Let $TS = (Q, E, T, q_{in}, \varphi)$ be a transition system. When viewed as the model of a system–environment combination, E will represent the environment actions and Σ the actions of the system. The occurrence of the transition $q \xrightarrow[a]{e} q'$ is to be viewed as the system offering to perform an a-action and the environment choosing the specific (a-labelled) event e as its matching response. There could be more than one a-labelled event enabled at q for the environment to choose from. We note also that it could be the case that $q \xrightarrow[a]{e} q'$ and $q \xrightarrow[b]{e'} q'$. Thus the environment could choose the same response — in terms of the change produced in the global state — to two different actions a and b of the system. This way of describing the system-environment interaction is taken from [AMP95]. In the present setting this will be easier to work with than the usual one in which the system moves and environment moves explicitly alternate [Tho95].

Example 2.1



In the example above, the transition system depicts a vending machine modelled in our framework. In the diagrams, events are written in **bold** while labels of events are written in *italics*. The events are $\{\mathbf{b_1}, \mathbf{b_2}, \mathbf{tea}, \mathbf{coffee}, \mathbf{r}, \mathbf{restart}\}$. The actions are $\Sigma = \{ask, tea, coffee, r, restart\}$. The labelling function maps the events $\mathbf{b_1}$ and $\mathbf{b_2}$ to *ask* and is the identity map on the other events. In its initial state, the machine reads which button (**b1** or **b2**) is pressed — it then decides to serve either tea or coffee. At the initial state, or whenever it reaches that state, the plant can offer to read an input by enabling the action *ask*. However, it has no control over whether **b1** or **b2** will be executed — the environment makes this choice.

We model both plants and specifications as transition systems. The controlled behaviour of a plant will be related to its specification by a simulation.

Definition 2.2 Let $TS_p = (Q_p, E_p, T_p, q_{in}^p, \varphi_p)$ and $TS_s = (Q_s, E_s, T_s, q_{in}^s, \varphi_s)$ be a pair of transition systems. Then a simulation f from TS_p to TS_s — denoted $f : TS_p \longrightarrow TS_s$ — is a map $f : Q_p \cup T_p \longrightarrow Q_s \cup T_s$ with $f(Q_p) \subseteq Q_s$ and $f(T_p) \subseteq T_s$ such that the following conditions are satisfied:

(i) $f(q_{in}^p) = q_{in}^s$.

(ii) Suppose $t = (q_p, e, q_p') \in T_p$ and $f(t) = (q_s, e', q_s')$. Then $f(q_p) = q_s$ and $f(q_p') = q_s'$ and $\varphi_p(e) = \varphi_s(e')$. □

Thus a simulation is just a structure preserving homomorphism. Given two transition systems TS_1 and TS_2 we say that TS_1 and TS_2 are isomorphic in case

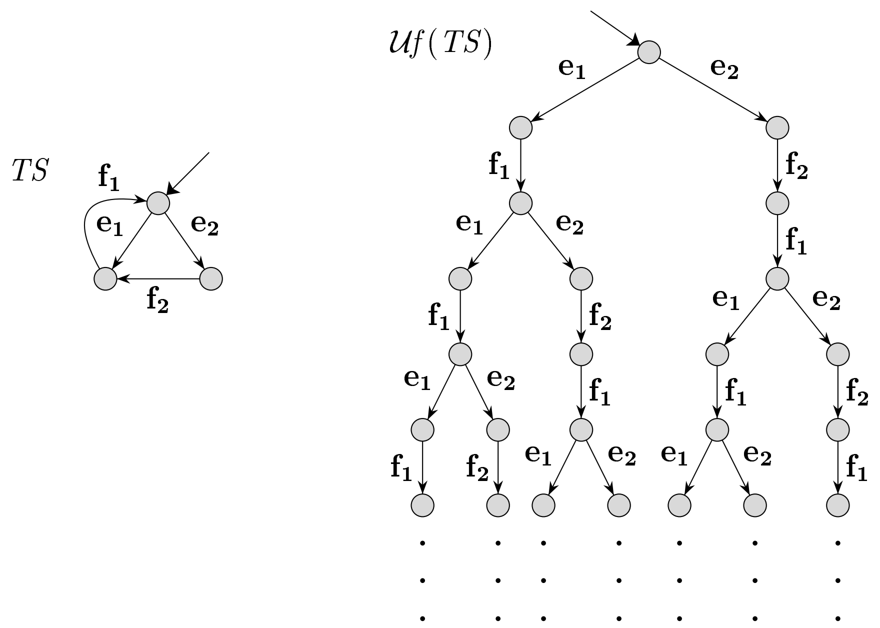


Figure 2.2: A transition system and its unfolding

there is a simulation $f : TS_1 \longrightarrow TS_2$ such that $f : Q_1 \cup T_1 \longrightarrow Q_2 \cup T_2$ is a bijection with Q_i (T_i) being the set of states (transitions) of TS_i for $i = 1, 2$.

The notion of the controlled behaviour of a plant meeting its specification via a simulation will be defined at the level of unfoldings. As we point out later this will permit a larger class of controllers.

Definition 2.3 Let $TS = (Q, E, T, q_{in}, \varphi)$ be a transition system. Then $\mathcal{U}f(TS)$, the *unfolding* of TS, is the structure $\widehat{TS} = (\widehat{Q}, \widehat{E}, \widehat{T}, \widehat{q}_{in}, \widehat{\varphi})$ where $\widehat{Q} \subseteq Q \times E^*$, $\widehat{E} \subseteq E$ and $\widehat{T} \subseteq \widehat{Q} \times \widehat{E} \times \widehat{Q}$ are the least sets satisfying:

(i) $(q_{in}, \varepsilon) \in \widehat{Q}$.

(ii) Suppose $(q, \sigma) \in \widehat{Q}$ and $(q, e, q') \in T$. Then $(q', \sigma e) \in \widehat{Q}$, $e \in \widehat{E}$ and $((q, \sigma), e, (q', \sigma e)) \in \widehat{T}$.

Further, $\widehat{q}_{in} = (q_{in}, \varepsilon)$ and $\widehat{\varphi}$ is φ restricted to $\widehat{E}$. □

It is easy to check that $\widehat{TS}$ is a deterministic Σ-labelled transition system. We could have defined $\widehat{Q}$ in terms of E^* alone but the present formulation will be easier to work with. Figure 2.2 illustrates a transition system and its unfolding.

Finally, the controlled behaviour of a plant will be obtained by taking the (synchronized) product of the plant and a controller.

Definition 2.4 Let $TS_i = (Q_i, E_i, T_i, q^i_{in}, \varphi_i)$, $i = 1, 2$ be a pair of transition systems such that $\forall e \in E_1 \cap E_2$, $\varphi_1(e) = \varphi_2(e)$. Then the product of TS_1 and TS_2 — denoted $TS_1 \| TS_2$ — is the structure $TS = (Q, E, T, q_{in}, \varphi)$ where $Q \subseteq Q_1 \times Q_2$ and $E \subseteq E_1 \cup E_2$, $T \subseteq Q \times E \times Q$ are the least sets that satisfy:

- $(q^1_{in}, q^2_{in}) \in Q$
- Suppose $(q_1, q_2) \in Q$ and $(q_1, e, q'_1) \in T_1$ with $e \notin E_2$. Then $e \in E$, $(q'_1, q_2) \in Q$ and $((q_1, q_2), e, (q'_1, q_2)) \in T$.
- Suppose $(q_1, q_2) \in Q$ and $(q_2, e, q'_2) \in T_2$ with $e \notin E_1$. Then $e \in E$, $(q_1, q'_2) \in Q$ and $((q_1, q_2), e, (q_1, q'_2)) \in T$.
- Suppose $(q_1, q_2) \in Q$, $e \in E_1 \cap E_2$, $(q_1, e, q'_1) \in T_1$ and $(q_2, e, q'_2) \in T_2$. Then $e \in E$, $(q'_1, q'_2) \in Q$ and $((q_1, q_2), e, (q'_1, q'_2)) \in T$.

Further, $q_{in} = (q^1_{in}, q^2_{in})$ and $\varphi : E_1 \cup E_2 \longrightarrow \Sigma$ is given by: $\varphi(e) = \varphi_1(e)$ if $e \in E_1$ and $\varphi(e) = \varphi_2(e)$ if $e \in E_2 \setminus E_1$. $\square$

Again it is easy to check that $TS_1 \| TS_2$ is also a deterministic Σ-labelled transition system. We are now ready to define controllers. As it will turn out, the plant-controller interaction will be a much tighter version of the product operation.

For a transition system $TS = (Q, E, T, q_{in}, \varphi)$ we will say $q \in Q$ is reachable from q_{in} if $q = q_{in}$ or there exists a non-null sequence of states $q_0 q_1 \ldots q_n$ with $q_0 = q_{in}$ and $q_n = q$ and for $0 \leq i < n$, $\exists e \in E : (q_i, e, q_{i+1}) \in T$. Note that for any pair of transition systems TS_1 and TS_2, all states of $TS_1 \| TS_2$ are reachable.

Definition 2.5 Let $TS_x = (Q_x, E_x, T_x, q^x_{in}, \varphi_x)$, $x \in \{p, c, s\}$ be three transition systems. Then TS_c is a controller for the pair (TS_p, TS_s) iff the following conditions are satisfied: Let $TS_p \| TS_c = (Q, E, T, q_{in}, \varphi)$.

(CT1) $E_c = E_p$ and $\varphi_c = \varphi_p$.

(CT2) (*non-restricting*) Suppose (q_p, q_c) is in $TS_p \| TS_c$ and $((q_p, q_c), e, (q'_p, q'_c)) \in T$ and $(q_p, e_1, q''_p) \in T_p$ with $\varphi_p(e) = \varphi_p(e_1)$. Then there exists $q''_c \in Q_c$ such that $((q_p, q_c), e_1, (q''_p, q''_c)) \in T$ (and hence $(q_c, e_1, q''_c) \in T_c$).

(CT3) (*non-blocking*) Suppose (q_p, q_c) is in $TS_p \| TS_c$ and $(q_p, e, q'_p) \in T_p$. Then there exists $e_1 \in E_p$, $q''_p \in Q_p$ and $q''_c \in Q_c$ such that $((q_p, q_c), e_1, (q''_p, q''_c)) \in T$.

(CT4) There is a simulation from $\mathcal{U}f(TS_p \| TS_c)$ to $\mathcal{U}f(TS_s)$. □

The condition (CT1) demands that the plant and the controller be tightly coupled. There are no "autonomous" transitions either for the plant or for the controller. Since the plant is deterministic on events, the controller observes everything that the plant does and can determine the state of the plant by observing the sequence of events it has executed. The condition (CT2) says that TS_c should restrict only the system moves. If at a reachable state it permits one a-move then it should permit *all* a-moves. The condition (CT3) requires that the controller should be non-blocking. Stated differently, the controller should not introduce any new deadlocks in the constrained plant behaviour. This condition also ensures that the problem does not degenerate, as otherwise there is always a controller which restricts *all* system moves and satisfies the specification.

The role of (CT4) should be clear. It says that the specification must be able to simulate the controlled plant. This basically means that we can cater for simple safety properties where the specification describes the tree of *system moves* allowed. We could have defined the simulation direction the other way, i.e. demand that the controlled plant must be able to simulate the specification. This will be a natural way to capture liveness properties. However, in the controller synthesis problem for such specifications, the notion becomes very weak because the controller will have no useful role to play. It is easy to see that if the plant does not satisfy the specification then no pruning of its behaviour (by a controller) will satisfy it.

Note that the controller need could have infinitely many states, even though the plant and specification are finite-state.

In formulating (CT4), we could have used TS_s instead of $\mathcal{U}f(TS_s)$. The choice of the latter is for the sake of uniformity.

Proposition 2.1 *Let TS_1 and TS_2 be two transition systems. If there is a simulation from TS_1 to $\mathcal{U}f(TS_2)$, then there is a simulation from TS_1 to TS_2.*

Proof Let $TS_i = (Q_i, E_i, T_i, q^i_{in}, \varphi_i)$, where $i = 1, 2$. Let $f : TS_1 \to \mathcal{U}f(TS_2)$ be a simulation. Then define $g : Q_1 \cup T_1 \to Q_2 \cup T_2$ as follows: Let $q_1 \in Q_1$ and $f(q_1) = (q_2, \sigma')$ (where $\sigma' \in E_2^*$). Then $g(q_1) = q_2$. Also, for a transition $t = (q_1, e, q_1')$ in T_1, let $f(t) = ((q_2, \sigma), e', (q_2', \sigma e'))$. Then $g(t) = (q_2, e', q_2')$. It is easy to verify that g is a simulation from TS_1 to TS_2. □

Also,

Proposition 2.2 *Let TS_1 and TS_2 be two transition systems. If there is a simulation from TS_1 to TS_2, then there is a simulation from $\mathcal{U}f(TS_1)$ to $\mathcal{U}f(TS_2)$.*

Proof Let $TS_i = (Q_i, E_i, T_i, q^i_{in}, \varphi_i)$, where $i = 1, 2$. Let $f : TS_1 \to TS_2$ be a simulation. Let $TS'_i = \mathcal{U}f(TS_i) = (Q'_i, E'_i, T'_i, {q^i_{in}}', \varphi'_i)$, where $i = 1, 2$. We now define $g : Q'_1 \cup T'_1 \to Q'_2 \cup T'_2$. Formally, we define, by induction on $|\sigma|$ (where $\sigma \in (E'_1)^*$), the values of $g((q, \sigma))$ and $g((q, \sigma'), e, (q_1, \sigma' e))$ where $\sigma' e = \sigma$. We also inductively maintain the property that if $g((q, \sigma)) = (q', \sigma')$, then $f(q) = q'$. Let $g((q^1_{in}, \varepsilon)) = (q^2_{in}, \varepsilon)$ (note that $f(q^1_{in}) = q^2_{in}$). Now, let $\sigma \in (E'_1)^*$ and $(q, \sigma) \in Q'_1$. Inductively assume that g has been defined on (q, σ) and let $g((q, \sigma)) = (q', \sigma')$. Let $t = ((q, \sigma), e, (q_1, \sigma e)) \in T'_1$ and $f((q, e, q_1)) = (q'', e', q'_1)$. But then $f(q) = q''$. Since inductively, $g((q, \sigma)) = (q', \sigma')$, $f(q) = q'$. So, $q' = q''$ and $f((q, e, q_1)) = (q', e', q'_1)$. Set $g(t) = ((q', \sigma'), e', (q'_1, \sigma' e'))$ and $g((q_1, \sigma e)) = (q'_1, \sigma' e')$. It is now straightforward to verify that g is a simulation from $\mathcal{U}f(TS_1)$ to $\mathcal{U}f(TS_2)$. $\square$

Lemma 2.3 *Let TS_1 and TS_2 be two transition systems. Then there is a simulation from $\mathcal{U}f(TS_1)$ to TS_2 iff there is a simulation from $\mathcal{U}f(TS_1)$ to $\mathcal{U}f(TS_2)$.*

Proof Follows directly from the Proposition 2.1 and Proposition 2.2 above and the fact that $\mathcal{U}f(\mathcal{U}f(TS_1))$ and $\mathcal{U}f(TS_1)$ are isomorphic. $\square$

Lemma 2.3 shows that we could have as well used TS_s instead of $\mathcal{U}f(TS_s)$ in (CT4). However, in (CT4), we cannot instead demand that there is a simulation from $TS_p \| TS_c$ to TS_s, or to $\mathcal{U}f(TS_s)$ — doing so would admit a smaller class of controllers as we illustrate below. But it will be convenient to identify this smaller class of controllers as well:

Definition 2.6 Let $TS_x = (Q_x, E_x, T_x, q^x_{in}, \varphi_x)$, $x \in \{p, c, s\}$ be three transition systems. Then TS_c is a *simple* controller for the pair (TS_p, TS_s) if it satisfies conditions (CT1), (CT2) and (CT3) of Definition 2.5 as well as:

(CT4′) There is a simulation from $TS_p \| TS_c$ to TS_s. $\square$

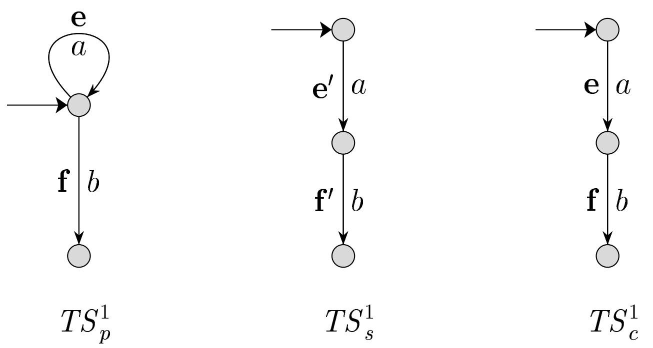
A simple controller, however, is a controller as well:

Proposition 2.4 *Let TS_c be a simple controller for (TS_p, TS_s). Then TS_c is a controller for (TS_p, TS_s).*

Proof Follows directly from Proposition 2.2. □

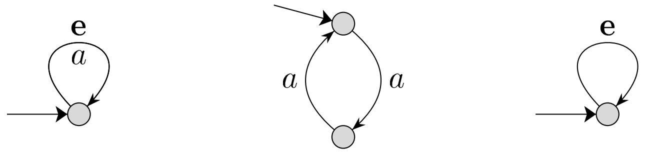
We have defined the goal of the controller to be able to restrict the plant such that there is a simulation function from the unfolding of the plant-controller combination to the unfolding of the specification. We could have instead required that there be *simulation relation* between the plant-controller pair (not its unfolding) and the specification. Though this would have been the more conventional route to take, we have chosen to take the present route because we feel that it is more transparent. Moreover, our notion extends naturally to the concurrent setting considered in the next chapter. In this extended setting the existence of a simulation function between the unfoldings of two transition systems does not imply the existence of a corresponding simulation relation between the two transition systems.
Let us now look at some examples:

Example 2.3



It is easy to see that TS_c^1 is a controller (as well as a simple controller) for the pair (TS_p^1, TS_s^1) — in fact, $TS_p^1 \| TS_c^1$ is isomorphic to TS_s^1. Note that the event names in the specification play no role — we will henceforth not show them in the diagrams of specifications. We will also not denote the labels of events in the controller as it will be the same as the labelling in the plant.

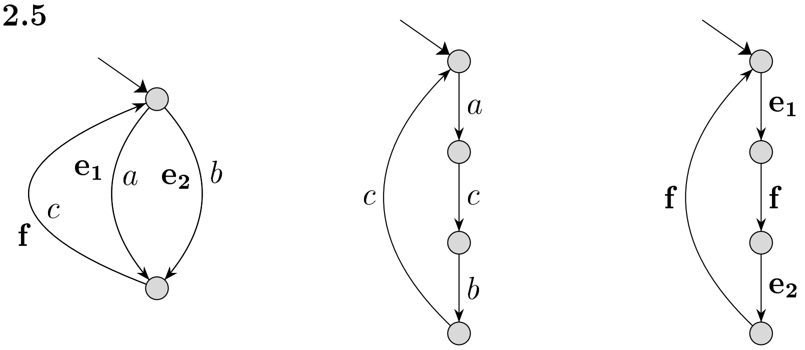
Example 2.4



TS^2_p TS^2_s TS^2_c

Again it is easy to see that TS^2_c is a "trivial" controller for the pair (TS^2_p, TS^2_s) (trivial in the sense that $TS^2_p \| TS^2_c$ is isomorphic to TS^2_p). But note that that it is not a simple controller. In fact, it is easy to see that this plant-specification pair does not admit a simple controller. Thus demanding a simulation map at the level of unfoldings admits a larger class of controllers in general.

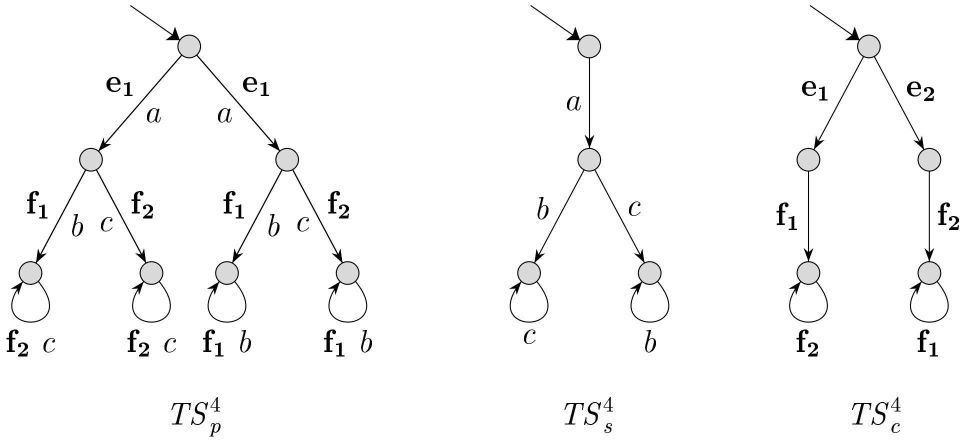
Example 2.5



TS^3_p TS^3_s TS^3_c

In this example, note again that TS^3_c is a controller for (TS^3_p, TS^3_s). However the controller is not a *zero-memory* controller in the sense that it does not simply prescribe a particular move from a state of the plant but prescribes moves depending on the history of interaction (it alternately schedules e_1 and e_2 from the same state of the plant). In other words, the plant-controller product has more states than the plant. It is easy to see that there is no zero-memory controller for (TS^3_p, TS^3_s).

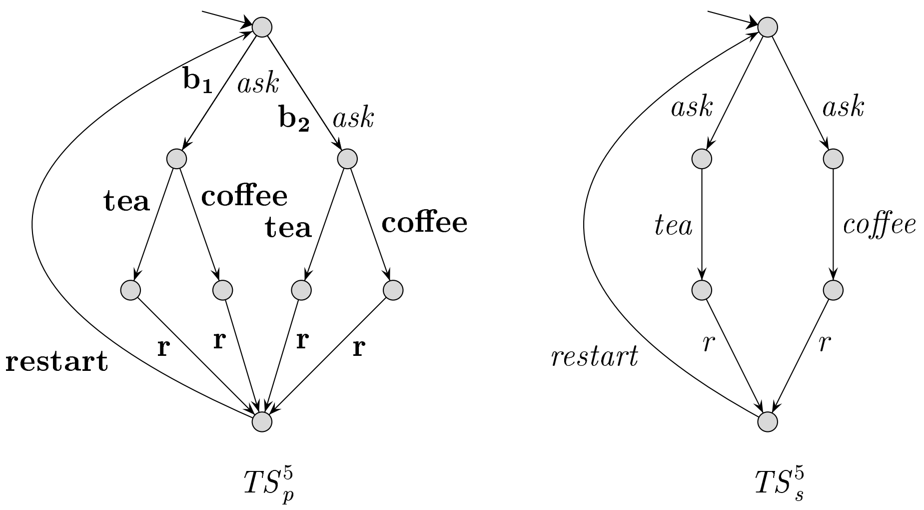
Example 2.6



This example illustrates the fact that the controller needs to observe the events (and not just the labels of the events) the plant executes. The controller in this case schedules the events $\mathbf{f_1}$ and $\mathbf{f_2}$ depending on whether $\mathbf{e_1}$ or $\mathbf{e_2}$ occurred (note that $\mathbf{e_1}$ and $\mathbf{e_2}$ have the same label). It is easy to see that there is no controller for (TS_p^4, TS_s^4) which can schedule events depending only on the sequence of labels of actions that have been executed (i.e. there is no controller which is deterministic on labels).

The following example will illustrate the branching nature of the specification. The plant TS_p^5 is a vending machine which first asks the user to press a button $\mathbf{b_1}$ or $\mathbf{b_2}$. Then it can serve either tea or coffee, and reset (the event $\mathbf{r}$) and restart. The events $\mathbf{b_1}$ and $\mathbf{b_2}$ are thus labelled with *ask* — the other events are pure plant moves and they are labelled using the identity function. In the specification TS_s^5, we show only the labels on the events — in this example, the specification demands that the button which is pressed must determine whether coffee or tea is served. It must not be the case that after a button is pressed, there is a possibility of both coffee and tea being served. However, it cannot demand that the user can get tea if he/she wants tea. We can demand such a specification in the bisimulation setting, which we will discuss in Section 2.5.

Example 2.7



A valid controller has to just disable either coffee or tea after $\mathbf{b_1}$ and $\mathbf{b_2}$. Note that it cannot disable both tea and coffee nor can it leave both enabled. But a controller which serves coffee on both button-presses is also a valid one. It is easy to see that there is no *minimally restrictive controller*, in the sense of one which allows the maximum number of event sequences. This in fact also shows that such specifications cannot be stated in the Ramadge–Wonham framework [RW89], as in their setting minimally restrictive controllers always exist.

We conclude this section by stating one of the main results of this chapter. In doing so and elsewhere we will say that a transition system TS is finite in case Q is finite. In case TS is finite, its size — denoted $|TS|$ — is defined to be $|Q| + |E|$.

Theorem 2.1 *Let* (TS_p, TS_s) *be a pair of finite transition systems and* $m = \max(|TS_p|, |TS_s|)$. *Then the question whether there exists a controller for* (TS_p, TS_s) *can be decided in time polynomial in* m. *Moreover, if a controller exists, we can construct one whose size is bounded by a polynomial in* m. *This construction also takes time bounded by a polynomial in* m. □

2.3 A good subgraph characterization

Our goal here is to characterize controllers in terms of objects called good subgraphs, which is a subgraph that satisfies some closure properties. This will lead to a proof of Theorem 2.1. Given a pair of finite transition systems (TS_p, TS_s) we form an edge-labelled directed graph G_{ps} which is a restricted product of TS_p and TS_s. We then show that (TS_p, TS_s) admits a controller iff G_{ps} contains a good subgraph.

Through the rest of the section we fix a pair of finite transition systems (TS_p, TS_s) with $TS_x = (Q_x, E_x, T_x, q^x_{in}, \varphi_x)$, $x \in \{p, s\}$. Then the edge-labelled directed graph $G_{ps} = (X, \rightarrow)$ is given by:

- $X = Q_p \times Q_s$
- $\rightarrow \subseteq X \times (E_p \times E_s) \times X$ is defined as:

$$(q_p, q_s) \xrightarrow{(e,e')} (q'_p, q'_s) \text{ iff } q_p \xrightarrow{e} q'_p \text{ in } TS_p \text{ and } q_s \xrightarrow{e'} q'_s \text{ in } TS_s \text{ and } \varphi_p(e) = \varphi_s(e').$$

We say that $G = (Y, \Rightarrow)$ is a subgraph of G_{ps} iff $Y \subseteq X$ and $\Rightarrow \subseteq \rightarrow \cap (Y \times (E_p \times E_s) \times Y)$.

Definition 2.7 Let $G = (Y, \Rightarrow)$ be a subgraph of $G_{ps} = (X, \rightarrow)$. Then G is said to be *good* iff it satisfies the following conditions.

(G1) $(q^p_{in}, q^s_{in}) \in Y$.

(G2) Suppose $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ and $q_p \xrightarrow{e_1} q^1_p$ in TS_p with $\varphi_p(e) = \varphi_p(e_1)$. Then $\exists e'_1 \in E_s, q^1_s \in Q_s : (q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q^1_p, q^1_s)$ in G.

(G3) Suppose $(q_p, q_s) \in Y$ and there exists $q_p \xrightarrow{e} \widehat{q_p}$ in TS_p. Then there exists $q'_p \in Q_p$, $q'_s \in Q_s$, $e_1 \in E_p$, $e'_1 \in E_s$ such that $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q'_p, q'_s)$ in G. □

Proposition 2.5 *Let TS_c be a controller for (TS_p, TS_s). Then $\mathcal{U}f(TS_c)$ is a* simple *controller for (TS_p, TS_s).*

Proof We first show a simulation from $TS_p \| \mathcal{U}f(TS_c)$ to $\mathcal{U}f(TS_s)$. Let $f : \mathcal{U}f(TS_p \| TS_c) \rightarrow \mathcal{U}f(TS_s)$ be a simulation. Define $g : (TS_p \parallel \mathcal{U}f(TS_c)) \rightarrow \mathcal{U}f(TS_s)$ as follows: Let $(q_p, (q_c, \sigma))$ be a state in $TS_p \parallel \mathcal{U}f(TS_c)$. It is easy to see that then $((q_p, q_c), \sigma)$ is a state of $\mathcal{U}f(TS_p \| TS_c)$. Define $g((q_p, (q_c, \sigma))) = f((q_p, q_c), \sigma)$. Also, for a transition $t = ((q_p, (q_c, \sigma)), e, (q'_p, (q'_c, \sigma e)))$ in $TS_p \parallel \mathcal{U}f(TS_c)$, the transition $t' = (((q_p, q_c), \sigma), e, ((q'_p, q'_c), \sigma e))$ is in $\mathcal{U}f(TS_p \| TS_c)$. Define $g(t) = f(t')$. It is easy to verify that g is a simulation. By Proposition 2.1 it follows that there is a simulation from $TS_p \| \mathcal{U}f(TS_c)$ to TS_s. □

Lemma 2.6 *Suppose TS_c is a simple controller for (TS_p, TS_s). Then G_{ps} contains a good subgraph.*

Proof Let $TS_c = (Q_c, E_c, T_c, q^c_{in}, \varphi_c)$ and $TS = TS_p \| TS_c = (Q, E, T, q_{in}, \varphi)$. Let f be a simulation from TS to TS_s. We now define the subgraph $(Y, \Rightarrow)$ of G_{ps} induced by f as follows.

- $(q_p, q_s) \in Y$ iff there exists $(q_p, q_c) \in Q$ such that $f((q_p, q_c)) = q_s$ for some $q_c \in Q_c$.
- $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ iff there exists $t = ((q_p, q_c), e, (q'_p, q'_c)) \in T$ such that $f(t) = (q_s, e', q'_s)$ for some $q_c, q'_c \in Q_c$.

We claim that $(Y, \Rightarrow)$ is a good subgraph of G_{ps}. The property (G1) follows from $f((q^p_{in}, q^c_{in})) = q^s_{in}$ which in turn implies $(q^p_{in}, q^s_{in}) \in Y$.

To verify (G2), assume that $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ and that $q_p \overset{e_1}{\longrightarrow} q^1_p$ in TS_p with $\varphi_p(e) = \varphi_p(e_1)$. From the definition of $\Rightarrow$, it follows that there exists $t = ((q_p, q_c), e, (q'_p, q'_c)) \in T$ with $f(t) = (q_s, e', q'_s)$, for some $q_c, q'_c \in Q_c$. Since TS_c is a controller, by property (CT2), it follows that $t_1 = ((q_p, q_c), e_1, (q^1_p, q^1_c))$ is a transition in TS for some q^1_c in Q_c. Let $f(t_1) = (q_s, e'_1, q^1_s)$. Then by the definition of $\Rightarrow$ we are assured that $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q^1_p, q^1_s)$. This establishes (G2).

Let us now prove (G3). Let $(q_p, q_s) \in Y$ and $q_p \overset{e}{\longrightarrow} \widehat{q}_p$. Then $\exists q_c \in Q_c : f((q_p, q_c)) = q_s$. By (CT3), there exists $e_1 \in E_p$ and $q^1_p \in Q_p$ such that $((q_p, q_c), e_1, (q^1_p, q^1_c)) \in T$. Let $f(t) = (q_s, e'_1, q^1_s)$. Then $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q^1_p, q^1_s)$.

□

Lemma 2.7 *Suppose TS_c is a controller for (TS_p, TS_s). Then G_{ps} contains a good subgraph.*

Proof If TS_c is a controller for (TS_p, TS_s), then by Proposition 2.5, $\mathcal{U}f(TS_c)$ is a simple controller for (TS_p, TS_s). From Lemma 2.6 it follows that G_{ps} has a good subgraph. □

As a first step towards proving the converse of Lemma 2.7 we first show that if G_{ps} contains a good subgraph then in fact it contains a good subgraph of a restricted kind.

Lemma 2.8 *Suppose G_{ps} contains a good subgraph. Then it contains a good subgraph $(Y, \Rightarrow)$ which satisfies the following condition.*

Suppose $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ and $(q_p, q_s) \overset{(e,e'')}{\Longrightarrow} (q'_p, q''_s)$. Then $e' = e''$ and hence $q'_s = q''_s$.

Proof Let $(Y_1, \Rightarrow_1)$ be a good subgraph of G_{ps}. Then we set $Y = Y_1$ and fix a linear order $<$ over E_s. Define now $\Rightarrow$ to be the least subset of $\Rightarrow_1$ which satisfies:

> Suppose $((q_p, q_s), (e, e'), (q'_p, q'_s)) \in \Rightarrow_1$ and there does not exist $((q_p, q_s), (e, e''), (q'_p, q''_s)) \in \Rightarrow_1$ with $e'' < e'$. Then $((q_p, q_s)), (e, e'), (q'_p, q'_s)) \in \Rightarrow$.

Hence, at any state, for an event $e \in E_p$, we keep only one representative edge of the kind (e, e') outgoing from the state. It is now easy to check that $(Y, \Rightarrow)$ is a good subgraph of G_{ps} having the desired property. □

We will say that a good subgraph of G_{ps} is s-deterministic ("simulation-deterministic") in case it satisfies the condition specified in the statement of Lemma 2.8.

Let $G = (Y, \Rightarrow)$ be an s-deterministic good subgraph of G_{ps}. We now define the structure $TS_c^G = (Q_c, E_c, T_c, q_{in}^c, \varphi_c)$ induced by G as follows. It will turn out that TS_c^G is a controller for (TS_p, TS_s).

- $Q_c = Y$ and $E_c = E_p$ and $\varphi_c = \varphi_p$.
- $T_c = \{((q_p, q_s), e, (q'_p, q'_s)) \mid \exists e' \in E_s : (q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)\}$
- $q_{in}^c = (q_{in}^p, q_{in}^s)$.

Lemma 2.9 *TS_c^G is a deterministic Σ-labelled transition system.*

Proof Suppose $((q_p, q_s), e, (q'_p, q'_s)) \in T_c$ and $((q_p, q_s), e, (q''_p, q''_s)) \in T_c$. Then there exist $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ and $(q_p, q_s) \overset{(e,e'')}{\Longrightarrow} (q''_p, q''_s)$ in $G = (Y, \Rightarrow)$. Clearly $q'_p = q''_p$ because TS_p is a deterministic Σ-labelled transition system. On the other hand, $e' = e''$ because G is s-deterministic and hence $q'_s = q''_s$ since TS_s is deterministic. □

Proposition 2.10 *Let $(q_p, (q'_p, q_s))$ be a state of $TS_p \| TS_c^G$ with $q_p, q'_p \in Q_p$ and $q_s \in Q_s$. Then $q_p = q'_p$.* □

Hence any state of $TS_p \| TS_c^G$ is of the form $(q_p, (q_p, q_s))$.

Lemma 2.11 *TS_c^G is a controller for (TS_p, TS_s). Hence, if G_{ps} contains a good subgraph then there is a controller for (TS_p, TS_s).*

Proof We will in fact show that TS_c^G is a simple controller (and hence by Proposition 2.4, a controller) for (TS_p, TS_s). Let $TS = TS_p \| TS_c^G$. (CT1) holds by definition of TS_c^G.

Let us now show (CT2). Suppose $(q_p, (q_p, q_s))$ is a state of TS and $(q_p, (q_p, q_s)) \xrightarrow{e} (q'_p, (q'_p, q'_s))$ in TS. Suppose further that $q_p \xrightarrow{e_1} q_p^1$ in TS_p with $\varphi_p(e) = \varphi_p(e_1)$. So $(q_p, q_s) \xrightarrow{e} (q'_p, q'_s)$ is in TS_c^G which means that there is an $e' \in E_s$ such that $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ in G, the good subgraph we started with. Since $q_p \xrightarrow{e_1} q_p^1$ is in TS_p, from property (G2) it follows that $\exists e'_1 \in E_s$ such that $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q_p^1, q_s^1)$ is in G. This implies that $(q_p, q_s) \xrightarrow{e_1} (q_p^1, q_s^1)$ is in TS_c^G and $(q_p, (q_p, q_s)) \xrightarrow{e_1} (q_p^1, (q_p^1, q_s^1))$ is in TS.

To show (CT3), let $(q_p, (q_p, q_s))$ be a state in TS and $q_p \xrightarrow{e} q_p^1$ in TS_p. Then $(q_p, q_s) \in Y$ and by property (G3), we know that there is an edge of the form $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q'_p, q'_s)$ in G. Hence $(q_p, q_s) \xrightarrow{e_1} (q'_p, q'_s)$ is in TS_c^G and $(q_p, (q_p, q_s)) \xrightarrow{e_1} (q'_p, (q'_p, q'_s))$ is in TS.

In order to show (CT4′), let us exhibit a simulation from TS to TS_s. Let $f : TS \to TS_s$ be defined as follows. For a state $(q_p, (q_p, q_s))$, define $f((q_p, (q_p, q_s))) = q_s$. Let $t = ((q_p, (q_p, q_s)), e, (q'_p, (q'_p, q'_s)))$ be a transition in TS. Then there is an $e' \in E_s$ such that $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ in G (since G is s-deterministic, this e' is unique). Define $f(t) = (q_s, e', q'_s)$. It now easily follows that f is a simulation from TS to TS_s.

Hence TS_c^G is a simple controller for (TS_p, TS_s). By Proposition 2.2, it is a controller for (TS_p, TS_s) as well. □

Theorem 2.2 *There is a controller for (TS_p, TS_s) iff G_{ps} has a good subgraph.* □

2.4 The synthesis procedure

We develop here a proof of Theorem 2.1. We know from Theorem 2.2 that deciding whether the pair (TS_p, TS_s) admits a controller boils down to deciding whether or not the graph G_{ps} contains a good subgraph. We establish in two steps that good subgraphs can be efficiently found.

Theorem 2.3 *There is a uniform decision procedure which takes as its input a pair of finite transition systems (TS_p, TS_s) and decides whether or not the edge-labelled directed graph G_{ps} (as defined in the previous section) contains a good subgraph.*

Proof We set $G_0 = G_{ps}$ and construct a sequence of graphs $G_0, G_1, \ldots G_n$ up to a stage where $G_n = G_{n+1}$. For every $i \in \{0, \ldots n\}$, G_{i+1} will be a subgraph of G_i. This pruning procedure will remove edges or vertices which evidence violations of properties (G2) or (G1) of a good subgraph. Then testing G_n for a simple property (whether $(q_{in}^p, q_{in}^s) \in G_n$), we will decide whether or not G_{ps} contains a good subgraph.

Assume that $G_0, \ldots, G_i$, $i \geq 0$ have been constructed. Let $TS_x = (Q_x, E_x, T_x, q_{in}^x, \varphi_x)$, $x \in \{p, s\}$.

Now, G_{i+1} is obtained from G_i by applying one of the following pruning steps to G_i. If neither of these two steps can be applied to G_i then we set $G_{i+1} = G_i$ and stop.

(i) Let $G_i = (X_i, \rightarrow_i)$. Suppose $(q_1, q_2) \in X_i$, (q_1, e_1, q_1'') is in T_p but there is no $(e_1', e_2') \in E_p \times E_s$ such that $(q_1, q_2) \stackrel{(e_1', e_2')}{\longrightarrow} (q_1', q_2')$ in G_i. Then remove (q_1, q_2) from X_i and all edges coming into (q_1, q_2). Let the resulting graph be G_{i+1}.

(ii) Suppose $(q_1, q_2) \stackrel{(e_1, e_2)}{\longrightarrow} (q_1', q_2')$ is an edge of G_i and (q_1, e_1', q_1'') is in T_p such that $\varphi_p(e_1) = \varphi_p(e_1')$. Further, suppose that there is no edge of the form $(q_1, q_2) \stackrel{(e_1', e_2')}{\longrightarrow} (q_1'', q_2'')$ in G_i. Then remove the edge $((q_1, q_2), (e_1, e_2), (q_1', q_2'))$ from G_i and let the resulting graph be G_{i+1}.

Clearly $G_{i+1} = G_i$ (in which case we stop) or G_{i+1} is strictly smaller than G_i. Since G_0 is finite this pruning procedure must stop after a finite number of steps. Let n be the least integer such that $G_n = G_{n+1}$ and let $G_n = (X_n, \rightarrow_n)$.

Claim: G_{ps} contains a good subgraph iff $(q_{in}^p, q_{in}^s) \in X_n$.

To see that the claim holds, suppose G_{ps} contains a good subgraph G. Then, by induction on n, is is easy to prove that G_n must also contain G as its subgraph. Thus $(q^p_{in}, q^s_{in}) \in X_n$.

Next suppose that $(q^p_{in}, q^s_{in}) \in X_n$. From the fact that no pruning rule is applicable on G_n, it follows at once that G_n is a good subgraph of G_{ps}. This establishes the claim and the theorem. $\square$

Corollary 2.12 *Let $TS_p = (Q_p, E_p, T_p, q^p_{in}, \varphi_p)$ and $TS_s = (Q_s, E_s, T_s, q^s_{in}, \varphi_s)$ be a pair of finite transition systems. Let $|Q_p| = n_1$, $|Q_s| = n_2$, $|E_p| = k_1$ and $|E_s| = k_2$. Let $m = max\{n_1, n_2, k_1, k_2\}$. Then in time polynomial in m, one can decide whether or not (TS_p, TS_s) has a controller.*

Proof Due to Theorem 2.2, it suffices to prove that in time polynomial in m one can check whether or not G_{ps} contains a good subgraph. Now consider the decision procedure developed in the proof of Theorem 2.3 for achieving this.

$G_0 = G_{ps}$ has at most $n_1 \cdot n_2$ vertices and $n_1^2 \cdot n_2^2 \cdot k_1 \cdot k_2$ edges. One can compute G_{i+1} from G_i in time which is linear in the size of G_i. Each G_{i+1} is smaller than G_i. Hence the decision procedure will terminate in at most $n_1^2 \cdot n_2^2 \cdot k_1 \cdot k_2$ steps. $\square$

Corollary 2.13 *Let $TS_p = (Q_p, E_p, T_p, q^p_{in}, \varphi_p)$ and $TS_s = (Q_s, E_s, T_s, q^s_{in}, \varphi_s)$ be a pair of finite transition systems. Let m be defined as in the previous corollary.*

(i) *If (TS_p, TS_s) has a controller, then it has a finite controller of size at most $n_1^2 \cdot n_2^2 \cdot k_1 \cdot k_2$.*

(ii) *Such a controller, if it exists, can be computed in time which is polynomial in m.*

Proof Again referring to the proof of Theorem 2.3, let n be the least integer such that $G_n = G_{n+1}$. Assume that $G_n = (X_n, \rightarrow_n)$ and that $(q^p_{in}, q^s_{in}) \in X_n$. We know from the previous corollary that G_n is of size at most $n_1^2 \cdot n_2^2 \cdot k_1 \cdot k_2$ and that G_n can be computed in time which is polynomial in m.

Now suppose $G_n = (X_n, \rightarrow_n)$ has the property $(q^p_{in}, q^s_{in}) \in X_n$. Then following the proof of Lemma 2.11 one can extract a controller TS_c^G for (TS_p, TS_s) (see page 26) in time which is linear in the size of G_n. $\square$

2.5 The bisimulation setting

We show in this section that Theorem 2.1 goes through even if we replace simulations by the stronger notion of bisimulations. Let us first define bisimulations [Mil80].

Definition 2.8 Let $TS_i = (Q_i, E_i, T_i, q_{in}^i, \varphi_i)$, $i = 1, 2$, be a pair of (deterministic Σ-labelled) transition systems. A bisimulation between TS_1 and TS_2 is a relation $R \subseteq Q_1 \times Q_2$ which satisfies:

- $(q_{in}^1, q_{in}^2) \in R$.
- Suppose $(q_1, q_2) \in R$ and $q_1 \xrightarrow[a]{e_1} q_1'$ is in TS_1. Then there exists a transition $q_2 \xrightarrow[a]{e_2} q_2'$ in TS_2 such that $(q_1', q_2') \in R$.
- Suppose $(q_1, q_2) \in R$ and $q_2 \xrightarrow[a]{e_2} q_2'$ is in TS_2. Then there exists a transition $q_1 \xrightarrow[a]{e_1} q_1'$ in TS_1 such that $(q_1', q_2') \in R$. $\square$

We say that TS_1 and TS_2 are bisimilar in case there is a bisimulation between them. Clearly, bisimilarity is an equivalence relation. In fact, we have:

Proposition 2.14 *Let $TS_i = (Q_i, E_i, T_i, q_{in}^i, \varphi_i)$, where $i = 1, 2, 3$ be three transition systems. If R_1 is a bisimulation between TS_1 and TS_2 and R_2 is a bisimulation from TS_2 and TS_3, then $R_1 \cdot R_2$ is a bisimulation between TS_1 and TS_3.* $\square$

In the above proposition, $R_1 \cdot R_2$ stands for the composition of the relations R_1 and R_2, i.e. $(q_1, q_3) \in R_1 \cdot R_2$ iff there is some q_2 such that $(q_1, q_2) \in R_1$ and $(q_2, q_3) \in R_2$.

It is also clear that every transition system is bisimilar to its unfolding. Hence we can work with bisimulations between transition systems rather than between their unfoldings.

Definition 2.9 Let TS_x, $x \in \{p, s, c\}$ be three transition systems. Then TS_c is a *strong* controller for the pair (TS_p, TS_s) iff TS_c satisfies the conditions (CT1), (CT2) of being a controller (Definition 2.5) and $TS_p \| TS_c$ is bisimilar to TS_s. $\square$

Note that we have dropped the non-blocking property (CT3). In the setting of simulations, we were capturing safety properties only and thus required that the controller should not introduce any deadlocks. However, in the bisimulation setting, we can capture liveness properties as well. In particular, the specification could demand that the plant should halt at some specified states.

Consider Example 2.7 (page 22) and view this as a new example where the specification intended as a bisimulation specification. The specification now demands that after a button is pressed, the possibility of serving both tea and coffee does not exist, as in the simulation setting. Further, it demands that there must be a way the user can get tea and a way in which he can get coffee. A controller which serves only coffee on pressing either button will satisfy the specification in the simulation setting but not in the bisimulation setting. A controller in this setting must enable coffee (only) on one input and tea (only) in the other. It is easy to see that a minimally restricting controller does not exist in this setting as well.

The synthesis problem now is the following: Given a pair of finite transition systems (TS_p, TS_s), is there a strong controller for this pair? It will be convenient to solve this problem while assuming that TS_s is reduced with respect to bisimilarity:

Definition 2.10 Let $TS = (Q, E, T, q_{in}, \varphi)$ be a transition system. Then TS is said to be *reduced* (w.r.t bisimilarity) iff the following conditions are satisfied:

(i) $\{R \mid R \subseteq Q \times Q \text{ is a bisimulation}\} = \{id_Q\}$ where $id_Q = \{(q, q) \mid q \in Q\}$

(ii) Suppose $q \xrightarrow[a]{e_1} q'$ and $q \xrightarrow[a]{e_2} q'$. Then $e_1 = e_2$. □

The next observation shows why it is convenient to work with reduced transition systems:

Proposition 2.15 *Let $TS_i = (Q_i, E_i, T_i, q^i_{in}, \varphi_i)$, $i = 1, 2$, be a pair of transition systems such that TS_2 is reduced. If $\approx$ is a bisimulation between TS_1 and TS_2, then it must satisfy the following properties:*

(i) *If $q_1 \approx q_2$ and $q_1 \approx q'_2$, then $q_2 = q'_2$.*

(ii) *If $q_1 \approx q_2$ and $q_1 \xrightarrow[a]{e_1} q'_1$, then there exists a* unique *$e_2 \in E_2 : q_2 \xrightarrow[a]{e_2} q'_2$ and $q'_1 \approx q'_2$.*

Proof Consider $\approx' = (\approx^{-1} \cdot \approx)$, the composition of the inverse relation of $\approx$ with $\approx$. I.e., $(q_2, q'_2) \in \approx'$ iff there is a $q_1 \in Q_1$ such that $(q_1, q_2), (q_1, q_3) \in \approx$. The it is clear that $\approx'$ is a bisimulation from TS_2 to itself. If $q_1 \approx q_2$ and $q_1 \approx q'_2$, then $q_2 \approx' q'_2$ which implies that $q_2 = q'_2$ (since TS_2 is reduced). Now let $q_1 \approx q_2$, $q_1 \xrightarrow[a]{e_1} q'_1$, $q_2 \xrightarrow[a]{e_2} q'_2$, $q_2 \xrightarrow[a]{e'_2} q''_2$ and $q'_1 \approx q'_2$, $q'_1 \approx q''_2$. By (i) we know that $q'_2 = q''_2$. By definition of reduced transition systems, it follows that $e_2 = e'_2$. □

Through the rest of this section, we fix a pair of finite transition systems (TS_p, TS_s) with $TS_x = (Q_x, E_x, T_x, q^x_{in}, \varphi_x)$, $x \in \{p, s\}$. We recall the definition of the edge-labelled directed graph G_{ps} and the associated terminology developed in Section 2.3.

Definition 2.11 Let $G_{ps} = (X, \rightarrow)$ and $G = (Y, \Rightarrow)$ be a subgraph of G_{ps}. Then G is a strong subgraph of G_{ps} iff the following conditions are satisfied:

(BS0) $(q^p_{in}, q^s_{in}) \in Y$

(BS1) Suppose $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q'_p, q'_s)$ is in G and $q_p \xrightarrow{e_2} q''_p$ is in TS_p with $\varphi_p(e_1) = \varphi_p(e_2)$. Then there exists $(q_p, q_s) \overset{(e_2,e'_2)}{\Longrightarrow} (q''_p, q''_s)$ in G (for some $e'_2 \in E_s$, $q''_s \in Q_s$).

(BS2) Suppose $(q_p, q_s) \in Y$ and $q_s \xrightarrow{e'_1} q'_s$ is in TS_s. Then there exists $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q'_p, q'_s)$ in G (for some $e_1 \in E_p$ and $q'_p \in Q_p$).

(BS3) Let $(q_p, q_s) \in Y$ and $E_{q_p,q_s} = \{(e, e') \mid \exists (q'_p, q'_s) : (q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s) \text{ is in } G\}$. Then there exists $\Gamma \subseteq E_{q_p,q_s}$ satisfying:

(i) If $(e_1, e'_1) \in E_{q_p,q_s}$, then there exists $e_2 \in E_p$ such that $(e_2, e'_1) \in \Gamma$.

(ii) If $(e_1, e'_1) \in E_{q_p,q_s}$, then there exists $e'_2 \in E_s$ such that $(e_1, e'_2) \in \Gamma$.

(iii) If $(e_1, e'_1), (e_1, e'_2) \in \Gamma$, then $e'_1 = e'_2$. □

Our aim now is to show that (TS_p, TS_s) admits a strong controller iff G_{ps} contains a strong subgraph.

Lemma 2.16 *If there is a strong controller for the pair of finite transition systems (TS_p, TS_s), where TS_s is reduced, then G_{ps} has a strong subgraph.*

Proof Let $TS_c = (Q_c, E_c, T_c, q^c_{in}, \varphi_c)$ be a strong controller for (TS_p, TS_s). Let $TS_p \| TS_c = TS = (Q, E, T, q_{in}, \varphi)$. Let $\approx \subseteq Q \times Q_s$ be a bisimulation. Now, define $G = (Y, \Rightarrow)$, a subgraph of G_{ps}, as follows:

- $(q_p, q_s) \in Y$ iff $\exists q_c \in Q_c : (q_p, q_c) \approx q_s$

- $(q_p, q_s) \overset{(e_1,e_1')}{\Longrightarrow} (q_p', q_s')$ iff $\exists q_c, q_c' \in Q_c : (q_p, q_c) \approx q_s$, $(q_p, q_c) \xrightarrow{e_1} (q_p', q_c') \in T$, $\varphi_p(e_1) = \varphi_s(e_1')$, $q_s \xrightarrow{e_1'} q_s'$, and $(q_p', q_c') \approx q_s'$.

Claim G is a strong subgraph of G_{ps}.
Since $(q_{in}^p, q_{in}^c) \approx q_{in}^s$, $(q_{in}^p, q_{in}^s) \in Y$.

Now suppose that $(q_p, q_s) \overset{(e_1,e_1')}{\Longrightarrow} (q_p', q_s')$ is in G and $q_p \xrightarrow{e_2} q_p''$ is in TS_p, with $\varphi_p(e_1) = \varphi_p(e_2)$. Then $\exists q_c, q_c' \in Q_c : (q_p, q_c) \approx q_s$, $(q_p, q_c) \xrightarrow{e_1} (q_p', q_c')$ is in TS, $(q_p', q_c') \approx q_s'$ and $q_s \xrightarrow{e_1'} q_s'$ is in TS_s. Since TS_c is a controller, $(q_p, q_c) \xrightarrow{e_2} (q_p'', q_c'')$ is in TS (by (CT2)). Since $\approx$ is a bisimulation, $\exists e_2' : q_s \xrightarrow{e_2'} q_s''$, $\varphi_s(e_2') = \varphi_p(e_2)$ and $(q_p'', q_c'') \approx q_s''$. Hence $(q_p, q_s) \overset{(e_2,e_2')}{\Longrightarrow} (q_p'', q_s'')$ is in G. This establishes (BS1).

Now suppose $(q_p, q_s) \in Y$ and $q_s \xrightarrow{e_1'} q_s'$ is in TS_s. Then $\exists q_c \in Q_c : (q_p, q_c) \approx q_s$. Hence $\exists e_1 : (q_p, q_c) \xrightarrow{e_1} (q_p', q_c')$, $(q_p', q_c') \approx q_s'$ and $\varphi_p(e_1) = \varphi_s(e_1')$. So $(q_p, q_s) \overset{(e_1,e_1')}{\Longrightarrow} (q_p', q_s')$ is in G. This shows (BS2).

To show (BS3), let $(q_p, q_s) \in Y$ and let E_{q_p,q_s} be as defined in the condition. Fix some $q_c \in Q_c$ with $(q_p, q_c) \approx q_s$. Then define $\Gamma = \{(e_1, e_1') \mid \varphi_p(e_1) = \varphi_s(e_1'), (q_p, q_c) \xrightarrow{e_1} (q_p', q_c') \text{ in } TS, q_s \xrightarrow{e_1'} q_s' \text{ in } TS_s \text{ and } (q_p', q_c') \approx q_s'\}$. Clearly $\Gamma \subseteq E_{q_p,q_s}$.

If $(e_1, e_1') \in E_{q_p,q_s}$, then $q_s \xrightarrow{e_1'} q_s'$ is in TS_s (for some $q_s' \in Q_s$). Since $(q_p, q_c) \approx q_s$, $\exists e_2 : (q_p, q_c) \xrightarrow{e_2} (q_p', q_c')$, $\varphi_p(e_2) = \varphi_s(e_1')$, $(q_p', q_c') \approx q_s'$. Hence $(e_2, e_1') \in \Gamma$. This shows BS3(i).

Let $(e_1, e_1') \in E_{q_p,q_s}$. Then $\exists \widetilde{q}_c, \widetilde{q}_c' : (q_p, \widetilde{q}_c) \approx q_s$, $(q_p, \widetilde{q}_c) \xrightarrow{e_1} (q_p', \widetilde{q}_c')$ in TS, $q_s \xrightarrow{e_1'} q_s'$ in TS_s and $(q_p', \widetilde{q}_c') \approx q_s'$. Since $(q_p, q_c) \approx q_s$, $\exists e_2 : (q_p, q_c) \xrightarrow{e_2} (q_p'', q_c'')$ in TS such that $\varphi_p(e_2) = \varphi_s(e_1') = \varphi_p(e_1)$ and $(q_p'', q_c'') \approx q_s'$. Since TS_c is a controller, $q_p \xrightarrow{e_1} q_p'$ and $\varphi_p(e_1) = \varphi_p(e_2)$, $\exists q_c' : (q_p, q_c) \xrightarrow{e_1} (q_p', q_c')$. Then $\exists e_2' : q_s \xrightarrow{e_2'} q_s''$, $\varphi_p(e_1) = \varphi_s(e_2')$, $(q_p', q_c') \approx q_s''$. Then $(e_1, e_2') \in \Gamma$. This proves BS3(ii).

Now let $(e_1, e_1'), (e_1, e_2') \in \Gamma$. Then there are transitions of the form $(q_p, q_c) \xrightarrow{e_1} (q_p', q_c')$ in TS , $q_s \xrightarrow{e_1'} q_s'$, $q_s \xrightarrow{e_2'} q_s''$ in TS_s and $\varphi_p(e_1) = \varphi_s(e_1') = \varphi_s(e_2')$, $(q_p', q_c') \approx q_s'$, $(q_p', q_c') \approx q_s''$. By Proposition 2.15, since TS_s is reduced, we know that $e_1' = e_2'$. □

Lemma 2.17 *Suppose (TS_p, TS_s) is such that TS_s is reduced and G_{ps} has a strong subgraph. Then there exists a strong controller for (TS_p, TS_s).*

Proof Let $G = (Y, \Rightarrow)$ be a strong subgraph of G_{ps}. For each $(q, q') \in Y$, let us fix a $\Gamma_{q,q'} \subseteq E_{q,q'}$ satisfying the condition (BS3). Consider the following transition system: $TS_c = (Q_c, E_c, T_c, q^c_{in}, \varphi_c)$ given by:

- $Q_c = Y$
- $E_c = E_p$; $\varphi_c = \varphi_p$
- $((q_p, q_s), e, (q'_p, q'_s)) \in T_c$ iff $\exists e' \in E_s : (e, e') \in \Gamma_{q_p,q_s}$ and $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ in G.
- $q^c_{in} = (q^p_{in}, q^s_{in})$.

Let us now verify that TS_c is a strong controller for the pair (TS_p, TS_s).

If $(q_p, q_s) \overset{e}{\longrightarrow} (q'_p, q'_s)$ and $(q_p, q_s) \overset{e}{\longrightarrow} (q''_p, q''_s)$ in TS_c, then $q'_p = q''_p$ (since TS_p is deterministic) and there exist $e', e'' \in E_s$ such that $(e, e'), (e, e'') \in \Gamma_{q_p,q_s}$, $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ and $(q_p, q_s) \overset{(e,e'')}{\Longrightarrow} (q'_p, q''_s)$. Since TS_s is deterministic, $q'_s = q''_s$. Hence TS_c is deterministic. Also, (CT1) is true by definition of TS_c.

Let $TS = TS_p \| TS_c$. First, it is easy to see that every state of $TS_p \| TS_c$ is of the form $(q_p, (q_p, q_s))$ where $q_p \in Q_p$ and $q_s \in Q_s$. Let us verify (CT2). Let $(q_p, (q_p, q_s)) \overset{e}{\longrightarrow} (q'_p, (q'_p, q'_s))$ in TS. Let $q_p \overset{e_1}{\longrightarrow} q''_p$ in TS_p with $\varphi_p(e) = \varphi_p(e_1)$. Then $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$ is in G (for some $e' \in E_s$). By (BS1), $(q_p, q_s) \overset{(e_1,e'_1)}{\Longrightarrow} (q''_p, q''_s)$ is in G (for some $e'_1 \in E_s$, $q''_s \in Q_s$). Hence $(e_1, e'_1) \in E_{q_p,q_s}$. By (BS3)(ii), $\exists e'_2 \in E_s$: $(e_1, e'_2) \in \Gamma_{q_p,q_s}$. Hence $(q_p, q_s) \overset{(e_1,e'_2)}{\Longrightarrow} (q'_p, q^2_s)$ is in G for some $q^2_s \in Q_s$. By definition of TS_c, $(q_p, q_s) \overset{e_1}{\longrightarrow} (q'_p, q^2_s)$ is in TS_c. Hence $(q_p, (q_p, q_s)) \overset{e_1}{\longrightarrow} (q'_p, (q'_p, q^2_s))$ is in TS.

All that remains is to show that TS is bisimilar to TS_s. Let $\approx$ be defined as follows: for every state $(q_p, (q_p, q_s))$ in TS, let $(q_p, (q_p, q_s)) \approx q_s$. Let us show that $\approx$ is a bisimulation by verifying properties listed in Definition 2.8. Clearly, $(q^p_{in}, (q^p_{in}, q^s_{in})) \approx q^s_{in}$.

Let $(q_p, (q_p, q_s)) \approx q_s$ and $(q_p, (q_p, q_s)) \overset{e}{\longrightarrow} (q'_p, (q'_p, q'_s))$ be in TS. Then by (BS3)(ii), $\exists e' \in E_s : \varphi(e) = \varphi(e')$, $(e, e') \in \Gamma_{q_p,q_s}$ and hence $(q_p, q_s) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$. So, $q_s \overset{e'}{\longrightarrow} q'_s$ is in TS_s and $(q'_p, (q'_p, q'_s)) \approx q'_s$.

Now let $(q_p, (q_p, q_s)) \approx q_s$ and $q_s \overset{e'}{\longrightarrow} q'_s$. By (BS2), $\exists e \in E_p, q'_p \in Q_p$: $(q_p, (q_p, q_s)) \overset{(e,e')}{\Longrightarrow} (q'_p, q'_s)$. Now by (BS3)(i), $\exists e_2 \in E_p : (e_2, e') \in \Gamma$. Let $(q_p, q_s) \overset{(e_2,e')}{\Longrightarrow} (q''_p, q'_s)$. Then $(q_p, (q_p, q_s)) \overset{e_2}{\longrightarrow} (q''_p, (q''_p, q'_s))$ and $(q''_p, (q''_p, q'_s)) \approx q'_s$. $\square$

We now wish to show that the existence of a strong controller can be decided in polynomial time. As a first step we will observe that assuming the specification transition system is reduced involves no loss in generality.

Lemma 2.18 *Let $TS = (Q, E, T, q_{in}, \varphi)$ be a finite transition system. Then in time polynomial in $|TS|$ one can construct a reduced transition system TS' which is bisimilar to TS.*

Proof This observation follows easily from the polynomial time algorithm for checking bisimilarity of two finite transition systems due to [KS83].

To be specific, we set $R_0 = Q \times Q$ and construct a sequence of relations $R_0, R_1, \ldots R_n$ till $R_n = R_{n+1}$ and then stop. Assume inductively that $R_0, R_1, \ldots R_i$ have been constructed. We define R_{i+1} to be the relation obtained by applying one of the following pruning steps to R_i. If neither of the two steps can be applied to R_i, then we set $R_{i+1} = R_i$ and stop.

- Suppose $(q, q') \in R_i$ and $q \xrightarrow{e_1} q_1$ is in T but there is no $q' \xrightarrow{e_1'} q_1'$ in T such that $\varphi(e_1) = \varphi(e_1')$ and $(q_1, q_1') \in R_i$. Then $R_{i+1} = R_i \setminus \{(q, q')\}$.
- Suppose $(q, q') \in R_i$ and $q' \xrightarrow{e_1'} q_1'$ is in T but there is no $q \xrightarrow{e_1} q_1$ in T such that $\varphi(e_1) = \varphi(e_1')$ and $(q_1, q_1') \in R_i$. Then $R_{i+1} = R_i \setminus \{(q, q')\}$.

Since R_0 is a finite set and $R_{i+1} = R_i$ (in which case we stop) or $R_{i+1} \subset R_i$, this procedure will terminate after at most $|Q \times Q|$ steps. Let n be the least integer such that $R_n = R_{n+1}$. It is easy to check that R_n is an equivalence relation. For $q \in Q$, let $[q]$ be the R_n-equivalence class containing q.

It is now easy to see that R_n will be a bisimulation between TS and itself. Also, if $\approx \subset Q \times Q$ is a bisimulation between TS and itself, then $\approx \subset R_n$.
Next we fix a strict linear order $<$ on E. We now define $TS' = (Q', E', T', q'_{in}, \varphi')$ via:

- $Q' = Q/R_n = \{[q] \mid q \in Q\}$
- $E' = E$; $\varphi' = \varphi$
- $([q], e, [q']) \in T'$ iff there exists $(q_1, e, q_1') \in T$ such that $q_1 \in [q]$ and $q_1' \in [q']$ and furthermore, if $(p, e', p') \in T$ with $p \in [q]$ and $p' \in [q']$ and $\varphi(e) = \varphi(e')$, then $e = e'$ or $e < e'$.

- $q'_{in} = [q_{in}]$

It is easy to verify that TS' is reduced and that TS and TS' are bisimilar with $\{(q,[q]) \mid q \in Q\}$ being a bisimulation. It is also easy to verify that $|TS'| \leq |TS|$ and that TS' can be computed in time polynomial in $|TS|$. □

Theorem 2.4 *There is a uniform procedure which takes as input a pair of finite transition systems (TS_p, TS_s) and decides whether or not (TS_p, TS_s) admits a strong controller.*

Proof Due to the previous lemma, it involves no loss of generality to assume that TS_s is reduced. It now follows from Lemmas 2.16 and 2.17 that it suffices to decide whether or not G_{ps} contains a strong subgraph. This can be achieved by constructing a sequence of graphs $G_0, G_1, \ldots G_{n+1}$ such that each G_i is a subgraph of G_{ps} and each G_{i+1} a subgraph of G_i with $G_0 = G_{ps}$ and $G_n = G_{n+1}$. Assume inductively that $G_0, \ldots G_i$ have been constructed. We now obtain G_{i+1} by applying one of the following pruning steps to G_i. If none of the pruning steps can be applied we set $G_{i+1} = G_i$ and stop.
Let $G_i = (X_i, \rightarrow_i)$.

(PR1) Suppose $t = ((q,q'),(e_1,e'_1),(q_1,q'_1)) \in \rightarrow_i$ and there exists $(q,e_2,q_2) \in TS_p$ with $\varphi_p(e_1) = \varphi_p(e_2)$. Further suppose that there exists no edge in $\rightarrow_i$ of the form $((q,q'),(e_2,e'_2),(q_2,q'_2))$. Then remove the edge t from $\rightarrow_i$ and set G_{i+1} to be the resulting graph.

(PR2) Suppose $(q,q') \in X_i$ and $q' \xrightarrow{e'_1} q'_1$ is in T_s but there is no edge of the form $((q,q'),(e_1,e'_1),(q_1,q'_1))$ in $\rightarrow_i$. Then remove (q,q') and all its incoming and outgoing edges from G_i and define G_{i+1} to be the resulting graph.

(PR3) Let $(q,q') \in X_i$.
Let $E^i_{q,q'} = \{(e_1,e'_1) \mid \exists (q_2,q'_2) : ((q,q'),(e_1,e'_1),(q_2,q'_2)) \text{ is in } \rightarrow_i\}$. Suppose every $\Gamma \subseteq E^i_{q,q'}$ fails to satisfy at least one of the conditions BS3 (i), (ii) and (iii). Then remove (q,q') and all its incoming and outgoing edges and define G_{i+1} to be the resulting graph.

Since G_0 is finite, this procedure will terminate after a finite number of steps. Let n be the least integer such that $G_n = G_{n+1}$. Let $G_n = (X_n, \rightarrow_n)$. Now it is easy to show that G_{ps} contains a strong subgraph iff $(q^p_{in}, q^s_{in}) \in X_n$.

First, if $(q^p_{in}, q^s_{in}) \in X_n$, it is clear that G_n is a strong subgraph of G_{ps} as the fact that (PR)(1)–(3) are not applicable means that G_n satisfies (BS)(1)–(3).

To prove the converse, let us assume that G_{ps} has a strong subgraph G. We can inductively prove (by induction on i) that G is a subgraph of G_i. It would then follow that since (q^p_{in}, q^s_{in}) is in G, it would be in G_n also. The induction goes as follows.

Clearly, G is a subgraph of $G_0 = G_{ps}$. Now assume inductively that G is a subgraph of G_i. If pruning step (PR1) or (PR2) is applied, it is easy to see that G will be a subgraph of G_{i+1}. If (PR3) is used, let the pruned node be (q, q'). To prove G is a subset of G_{i+1} it suffices to prove that $(q, q') \notin G$. Assume the contrary. Then since G is a strong subgraph, it satisfies (BS3) for the node (q, q') — let $\Gamma \subseteq E_{q,q'}$ be a set which satisfies (BS3)(i)–(iii) with $E_{q,q'} = \{(e, e') \mid \exists (q_2, q'_2) : ((q, q'), (e, e'), (q_2, q'_2))$ is in $G\} \subseteq E^i_{q,q'}$.

Now it is clear that $\Gamma \subseteq E^i_{q,q'}$. We will show that Γ in fact satisfies (BS3)(i)–(iii) for the node (q, q') in G_i.

Let $(e, e') \in E^i_{q,q'}$. Then there must be a transition $q' \xrightarrow{e'} q'_1$ in T_s. Since G is a strong subgraph, by (BS2), $\exists e_1 : (q, q') \xrightarrow{e_1, e'} (q_1, q'_1)$ is in G. Hence $(e_1, e') \in E_{q,q'}$. By (BS3)(i) for (q, q') in G, $\exists e_2 : (e_2, e') \in \Gamma$. This shows (BS3)(i) for (q, q') in G_i.

Let $(e, e') \in E^i_{q,q'}$. Then, as argued above, $\exists e_1 : (q, q') \xrightarrow{e_1, e'} (q_1, q'_1)$ is in G. Since $\varphi_p(e) = \varphi_p(e_1)$, by (BS1), $\exists e'_2 : (q, q') \xrightarrow{e, e'_2} (q_2, q'_2)$ is in G. So, $(e, e'_2) \in E_{q,q'}$. Since G is a strong subgraph, by BS3(ii), $\exists e'_3 : (e, e'_3) \in \Gamma$, which shows BS3(ii) holds for G_i.

Also, since Γ satisfies BS3(iii) for G, it also satisfies it for G_i.

This shows that the conditions for using (PR3) is not met, which contradicts our assumption. Hence $(q, q') \notin G$ and hence G is a subgraph of G_{i+1}. □

Corollary 2.19 *Let (TS_p, TS_s) be a pair of finite transition systems with $|Q_p| = n_p$, $|Q_s| = n_s$, $|E_p| = k_p$ and $|E_s| = k_s$. Let $m = max\{n_p, n_s, k_p, k_s\}$. Then in time polynomial in m, one can decide whether or not (TS_p, TS_s) admits a strong controller.*

Proof By Lemma 2.18, we can construct in time polynomial in m, a reduced transition system TS'_s such that TS_s and TS'_s are bisimilar. We can now supply (TS_p, TS'_s) as input to the decision procedure presented in the proof of Theorem 2.3.

This procedure will take time only polynomial in m. To show this, the only nontrivial part is to show how rule (PR3) can be implemented in time polynomial in m. We do this by showing a reduction to the *maximal matching problem*.

For an undirected bipartite graph $G = (S_1, S_2, A)$ where S_1 and S_2 are sets of vertices and $A \subseteq S_1 \times S_2$, a *matching* for G is a set $X \subseteq A$ such that for any vertex $s \in S_1 \cup S_2$, there is at most one edge in X that is incident on s. A matching X of G is a maximal matching if it is a matching of maximum cardinality — i.e. for every matching X' of G, $|X'| \leq |X|$. The problem of finding a maximal matching of a given graph is a well-studied problem and can be solved in time polynomial in the size of the given graph [CLR92].

Let $G_i = (X_i, \rightarrow_i)$ and $(q, q') \in X_i$. Let $E_{q,q'}$ be defined as before. Consider the bipartite (undirected) graph (S_1, S_2, A) where

- $S_1 = \{e_1 \mid \exists e_2 : (e_1, e_2) \in E_{q,q'}\}$
- $S_2 = \{e_2 \mid \exists e_1 : (e_1, e_2) \in E_{q,q'}\}$
- $A = E_{q,q'}$

We now claim that there is a Γ satisfying the conditions (BS3)(i)–(iii) iff there is a matching for (S_1, S_2, A) of size $|S_2|$. Let Γ be a subset of $E_{q,q'}$ that satisfies conditions (BS3)(i)–(iii). For each $e_2 \in S_2$, we know by (BS3)(i) that there is an edge of the form (e_1', e_2') in Γ. Pick one such edge for each $e_2 \in E_s$ and call this set X. $|X| = |S_2|$. Let $(e_1, e_2), (e_1', e_2') \in X$. If $e_1 = e_1'$, then by (BS3)(iii) we know that $e_2 = e_2'$. If $e_2 = e_2'$, then from the way X was formed, $e_1 = e_1'$. Hence X is a matching of size $|S_2|$ and is clearly a maximal matching. Conversely, assume $X \subseteq E_{q,q'}$ is a matching of size $|S_2|$. Clearly, for each $e_2 \in S_2$, there must be exactly one edge of the form (e_1, e_2) in X. Let $\Gamma \supseteq X$ be formed by expanding X by adding precisely one edge of the form (e_1, e_2) of $E_{q,q'}$ for each $e_1 \in S_1$ where there is no edge of the kind $(e_1, e_2') \in X$. Clearly, Γ satisfies (BS3)(i)–(iii).

Since the size of the maximal matching of a given graph can be found in polynomial time, we can implement (PR3) in polynomial time. In fact, we can solve the maximal matching problem by reducing it to the max-flow problem and use the Ford-Fulkerson method ([EK70, CLR92]) to get a maximal matching in polynomial time, from which we can get a witness Γ. These witnesses will be useful in constructing the controller. $\square$

Corollary 2.20 *Let (TS_p, TS_s) be a pair of finite transition systems with m defined as above. Then (TS_p, TS_s) admits a strong controller iff it admits a strong controller of size at most a polynomial in m. Moreover, such a controller can be constructed in time polynomial in m.*

Proof Using the decision procedure presented in the proof of Theorem 2.4, once can compute a strong subgraph of G_{ps}, if one exists, in time polynomial in m. We can synthesize a strong controller from the strong subgraph as shown in the proof of Lemma 2.17. Clearly, the size of this controller will be at most polynomial in m. $\square$

2.6 Conclusions

In this chapter we have studied the controller synthesis problem in a branching time setting. We started with a simple notion of branching time specifications, namely simulations, which can capture simple safety properties. We then considered bisimulation specifications, which can express liveness properties as well, and are a natural extension to simulations. In both instances we have established polynomial time decision procedures as well as polynomial time synthesis procedures which produce polynomial sized controllers whenever controllers exist.

A considerable amount of knowledge is available about the control-synthesis problem in the linear time framework. Here the behaviour of the plant will consist of L_P, a suitable collection of (finite or infinite) sequences. One then specifies the desired behaviour by another collection of sequences L_S. The problem then is to come up with a controller such that $L_{PC} \subseteq L_S$ where L_{PC} is the constrained language generated by the plant-controller combination.

As for branching-time specifications, the supervisory control synthesis problem has been studied in a branching time setting using the failure semantics model of processes [Ove94, Ove97]. A pre-order relates the behaviour of the plant-controller to the specification. However their setup is very different. In their setting, the nondeterminism arises due to abstraction and not due to the hiding of the environment's actions. Consequently, their controllers cannot distinguish between the nondeterministic choices made in the plant. In our setting the nondeterminism (on the labels of events) is purely due to the hiding of the environment's responses and the controller can discern between the nondeterministic choices made. A nice feature of [Ove97] is that it deals with partial descriptions via the use of internal events. Extension of our work to handle partial descriptions is yet to be achieved.

There is a neighbouring body of work (see for instance [JL91], [LX90]) which has a similar flavour as the controller synthesis problem and uses techniques similar to those we discuss in this paper. This body of work has to do with equation solving in a process algebraic domain. The simplest problem setting is one where one is given a system A and a specification B both presented as terms in a process algebra, say CCS. The problem is to come up with a CCS term X such that $A|X$ is bisimilar to B. To consider an extreme example, suppose A is the process *nil* which does nothing. Then $X = B$ will be accepted as solution to the equation $A|X = B$. Thus the crucial difference between the work reported here and the work on equation solving in process algebras is that our controllers — unlike the unknown term X in the process algebra setting — can only restrict the behaviour of the plant; it is not allowed to contribute any new behavioural possibilities.

Our results can be extended in a number of ways. To mention just a few, one could consider plants with internal events and also controllers with internal events. In the case of controllers with internal events one will have to deal with refinement maps instead of simulations and one will have to deal with weak bisimulations instead of (strong) bisimulations.

A natural extension of this work is to consider the problem where we can handle specifications written in branching-time logics such as CTL, $\forall$-CTL, $\mathsf{CTL}^\star$, etc. It is hard to pin down a nice logic (say as a sub-logic of CTL) which will capture the notion of simulation/bisimulation we have considered. A related work is [AM95] where the branching time temporal logic CTL is used for specifications. The notion of a controller is however quite weak in that controllers are required to be memoryless.

Though the notions of simulations and bisimulations are weak mechanisms of specification, the attractive feature of these is that model-checking is in polynomial time due the "local" nature of the definitions of such relations. One finds very few specification mechanisms in the literature which yield such polynomial time algorithms. Our work shows that this tractability extends to control-synthesis as well. (The logic CTL also has a polynomial time model-checking algorithm — however as we point out in Chapter 5, the tractability does not extend to control-synthesis for CTL.)

One way to enhance our specification mechanisms is to handle them in a setting which can express *fairness* properties. In formulating liveness specifications, it is important in many cases to say that the *fair* runs (runs where the scheduler does not ignore a process forever, runs where a failure of an event doesn't happen infinitely often, etc.) satisfy a specification. In particular, the notion of *fair simulation* [HKR97, ESW01], introduced in [HKR97] is a notion of simulation which caters to fairness and at the same time allows polynomial-time model checking. It would be interesting to see if the control-synthesis problem for fair simulation, and other notions of simulations catering to fairness requirements, are also tractable.

Another challenging extension is suggested by the environment model considered by Kupferman and Vardi in their work on module checking [KV96, KV97a]. The idea is that in a branching time setting what one should require is: the controller should prune the system moves in such a way that for *every* pruning of its moves by the environment, the resulting computation tree should meet the specification. We note however that in the presence of simulations and bisimulations, this refined modelling of the environment is immaterial. It is however very relevant when we start considering branching time temporal logics, such as CTL, as specification mechanisms. A variety of interesting and computationally hard problems arise in this new setting and will be the subject of Chapter 5.

Yet another extension is to study the control-synthesis problem in a concurrent setting for simulations and bisimulations. This is the topic of study in the next chapter.

Chapter 3

Asynchronous simulations

But the principal failing occurred in the sailing,
And the Bellman, perplexed and distressed,
Said he had hoped, at least, when the wind blew due East,
That the ship would not travel due West!

— *The Hunting of the Snark*, Lewis Caroll

3.1 Introduction

The transition systems we studied in Chapter 2 are a suitable model to describe sequential systems and can be augmented with some concurrency information to model distributed systems. In this chapter we study a well-established variant called *asynchronous transition systems* [Bed88, WN95] and a corresponding notion of simulation (called an asynchronous simulation) between them. We show the surprising result that even checking whether there is such a simulation between the unfoldings of two finite asynchronous transition systems is undecidable. It turns out that, consequently, there is no way to effectively solve the control-synthesis problem in this setting. This is in sharp contrast to the results in Chapter 2 and show how complex the design of any notion of distributed control can become.

There is a natural notion of bisimulation between asynchronous transition systems studied in the literature called the *hereditary history-preserving bisimulation* (see [WN95, JNW96]). A long-standing open question in this area was whether the problem of checking if there is a hereditary history-preserving bisimulation between a pair of finite asynchronous transition systems is decidable.

The result in this chapter hinted that the hereditary history preserving bisimulation problem might be undecidable. Jurdziński and Nielsen [JN00] have recently shown that this problem is indeed undecidable. Their proof makes essential use of the technique we develop here to encode grids into unfoldings of asynchronous transition systems. We conjecture that their result can be extended to show that the controller problem for hereditary history-preserving bisimulation is also undecidable.

Coming back to our work, we model both the system and the specification as asynchronous transition systems and the notion of a simulation from one asynchronous transition system to another will be defined so that it preserves the *independence* of events. Unfoldings of asynchronous transition systems are defined such that states reached after trace-equivalent behaviours are identified with each other. We then show that the problems of model-checking and control-synthesis are undecidable. We also show that our negative result holds even for very restricted classes of asynchronous transition systems.

3.2 The model

We enrich the transition systems defined in Definition 2.1 of Chapter 2 to reflect the notion of independence of events.

Definition 3.1 A Σ-labelled deterministic asynchronous transition system is a structure $TS = (Q, E, T, q_{in}, \varphi, I)$ where $(Q, E, T, q_{in}, \varphi)$ is a transition system (as in Definition 2.1) and $I \subseteq E \times E$ is an irreflexive and symmetric independence relation such that the following conditions are satisfied:

(TR1) Suppose $q \xrightarrow{e_1} q_1$ and $q \xrightarrow{e_2} q_2$ and $e_1 \ I \ e_2$. Then there exists q' such that $q_1 \xrightarrow{e_2} q'$ and $q_2 \xrightarrow{e_1} q'$.

(TR2) Suppose $q \xrightarrow{e_1} q_1 \xrightarrow{e_2} q'$ and $e_1 \ I \ e_2$. Then there exists q_2 such that $q \xrightarrow{e_2} q_2 \xrightarrow{e_1} q'$. □

From now on, we refer to Σ-labelled deterministic asynchronous transition systems as just asynchronous transition systems. Simulations will now be required to preserve the independence of events:

Definition 3.2 Let $TS_1 = (Q_1, E_1, T_1, q^1_{in}, \varphi_1, I_1)$ and $TS_2 = (Q_2, E_2, T_2, q^2_{in}, \varphi_2, I_2)$ be a pair of asynchronous transition systems. Then an asynchronous simulation $f : TS_1 \to TS_2$ is a simulation from $(Q_1, E_1, T_1, q^1_{in}, \varphi_1)$ to $(Q_2, E_2, T_2, q^2_{in}, \varphi_2)$ (as in Definition 2.2, Chapter 2) which in addition satisfies:

- Suppose in TS_1, we have $e_1 \ I_1 \ e_2$, $t_1 = (q, e_1, q_1)$, $t_2 = (q_1, e_2, q')$, $t_3 = (q, e_2, q_2)$ and $t_4 = (q_2, e_1, q')$.

– If $f(t_1) = (p, e'_1, p_1)$ and $f(t_2) = (p_1, e'_2, p')$ then $e'_1 \ I_2 \ e'_2$ and there exists p_2 such that $f(t_3) = (p, e'_2, p_2)$ and $f(t_4) = (p_2, e'_1, p')$.

– If $f(t_1) = (p, e'_1, p_1)$ and $f(t_3) = (p, e'_2, p_2)$ then $e'_1 \ I_2 \ e'_2$ and there exists p' such that $f(t_2) = (p_1, e'_2, p')$ and $f(t_4) = (p_2, e'_1, p')$. □

From now on we will often drop the adjective "asynchronous" in referring to asynchronous simulations. As before controllers will be defined in terms of unfoldings. The new feature is that the independence of events will induce a partial order over the runs of the system. A standard technique taken from Mazurkiewicz trace theory [DR95] will be used to group together different interleavings of the same partially ordered stretch of behaviour.

Definition 3.3 Let $TS = (Q, E, T, q_{in}, \varphi, I)$ be an asynchronous transition system. Then $\sim_{TS}$ is the least equivalence relation (which turns out to be a congruence) contained in $E^* \times E^*$ which satisfies: $\tau e_1 e_2 \tau' \sim_{TS} \tau e_2 e_1 \tau'$ whenever $e_1 \ I \ e_2$ and $\tau, \tau' \in E^*$. We let $[\tau]$ denote the $\sim_{TS}$-equivalence class containing τ. □

Unfoldings are now defined by identifying states that arise by executing sequences of actions that are $\sim_{TS}$ equivalent:

Definition 3.4 Let $TS = (Q, E, T, q_{in}, \varphi, I)$ be an asynchronous transition system. The unfolding of TS is $\mathcal{U}f(TS) = (\widehat{Q}, \widehat{E}, \widehat{T}, \widehat{q}_{in}, \widehat{\varphi}, \widehat{I})$ where $\widehat{Q}$, $\widehat{E}$ and $\widehat{T}$ are the smallest sets that satisfy:

- $(q_{in}, [\varepsilon]) \in \widehat{Q}$.

- If $(q, [\tau]) \in \widehat{Q}$ and $(q, e, q') \in T$ then
  $(q', [\tau e]) \in \widehat{Q}$, $e \in \widehat{E}$ and $((q, [\tau]), e, (q', [\tau e])) \in \widehat{T}$.

The initial state is $\widehat{q}_{in} = (q_{in}, [\varepsilon])$ and $\widehat{\varphi}$ and $\widehat{I}$ are φ and I restricted to $\widehat{E}$ and $\widehat{E} \times \widehat{E}$ respectively. □

Trace theory ensures that $\mathcal{U}f(TS)$ is also an asynchronous transition system. Figure 3.1 shows an asynchronous transition system TS_p and its unfolding $\mathcal{U}f(TS_p)$. The independence relation is the symmetric closure of $\{\mathbf{ask_1}, \mathbf{cs_1}\} \times \{\mathbf{ask_2}, \mathbf{cs_2}\}$. Note that unlike the unfoldings in Chapter 2, the unfolding is not a tree, but is a directed acyclic graph.

The *model-checking problem for asynchronous simulations* is determine, given a pair of transition systems TS_p and TS_s, whether there is an asynchronous simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$.

Let us now consider products of asynchronous transition systems. The new feature is that the concerned independence relations should agree on the common events. Let TS_1 and TS_2 be two asynchronous transition systems with E_i as the set of events and φ_i as the labelling function of TS_i, $i \in \{1, 2\}$. Then $TS_1 \| TS_2$ is defined iff $\forall\ e, e' \in E_1 \cap E_2$, $e\ I_1\ e'$ iff $e\ I_2\ e'$. If this condition is satisfied (and the condition that $\forall e \in E_1 \cap E_2$, $\varphi_q(e) = \varphi_2(e)$ is also met), then $TS_1 \| TS_2$ is defined as done in Definition 2.4, Chapter 2 with the new independence relation defined as $I_1 \cup I_2$. Again, it should be clear that $TS_1 \| TS_2$ is an asynchronous transition system.

Let TS_p, TS_s and TS_c be three asynchronous transition systems. Then TS_c is an asynchronous controller for (TS_p, TS_s) iff TS_c satisfies the usual properties (CT1)–(CT3) of Definition 2.5 for being a controller and if there exists an asynchronous simulation from $\mathcal{U}f(TS_p \| TS_c)$ into $\mathcal{U}f(TS_s)$. The control-synthesis problem is then to check whether, given a pair of finite asynchronous transition systems TS_p and TS_s, there is an asynchronous controller for (TS_p, TS_s).

Let us consider the example given below in Figure 3.1. The plant consists of two agents which do the following: these agents wait for the user to press a button ($\mathbf{ask_i}$) after which they enter a critical section ($\mathbf{cs_i}$). When they finish and exit the critical section, they send a signal ($\mathbf{fin_i}$) which can be observed by the other agent. The two agents are shown in the figure. The combined system is the normal synchronized product of the two systems and is also illustrated. The unfolding of the plant is also shown. The induced independence relation is the symmetric closure of $\{\mathbf{ask_1}, \mathbf{cs_1}\} \times \{\mathbf{ask_2}, \mathbf{cs_2}\}$. Let us fix the labelling function as $\varphi(\mathbf{ask_i}) = ask$, $\varphi(\mathbf{cs_i}) = cs$ and $\varphi(\mathbf{fin_i}) = \mathit{fin}$, $i \in \{1, 2\}$.

The specification TS_s (which is equivalent to its unfolding) is shown in Figure 3.2 with only the labels of events on the transitions — the independence of events should be clear. On the labels, it is identical to the plant, except that it has no moves enabled when both agents are in the critical section. This therefore demands that the plant should not reach a state where both agents are in their critical sections (if it reaches such a state, then the controller will not be able to satisfy the nonblocking condition at this state).

An asynchronous controller is required to respect the independence relation. Hence it cannot enable an agent entering a critical section depending upon an independent event occurring in the other agent. In other words, independent actions have to be controlled independently. In this example, we require that the event $\mathbf{cs_1}$ is controlled independent of the event $\mathbf{cs_2}$ (as they belong to different agents). Hence the controller is forced to sequentialize the agents in a predetermined manner — an example of a valid controller is TS_c shown in the figure which allows the first agent to enter its critical section before the second, regardless of the sequence of buttons pressed.

Before we go on to proving that the model-checking problem in this setting of asynchronous simulations is undecidable, note that the model-checking problem for simulations for sequential transition systems studied in Chapter 2 is decidable. One can easily show that, given two finite-state (sequential) transition systems TS_p and TS_s, there is a simulation from TS_p to TS_s iff the corresponding graph G_{ps} defined in Section 2.3 is itself a good subgraph. Checking whether G_{ps} can clearly be done in polynomial time.

We now wish to show that the problems of model-checking for *finite* asynchronous simulations and the problem of deciding if a pair of finite asynchronous transition systems admits an asynchronous controller — finite or otherwise — are undecidable. The reduction is from the tiling problem [LP81] which is known to be undecidable. In what follows, it will be convenient to talk about the tiling problem as a colouring problem. An instance of the colouring problem is a quadruple $\mathcal{C} = (C, c_{in}, R, U)$ where C is a finite set of colours, $c_{in} \in C$ is a distinguished initial colour and $R : C \longrightarrow 2^C$ and $U : C \longrightarrow 2^C$ are two functions. A solution to $\mathcal{C}$ is a map

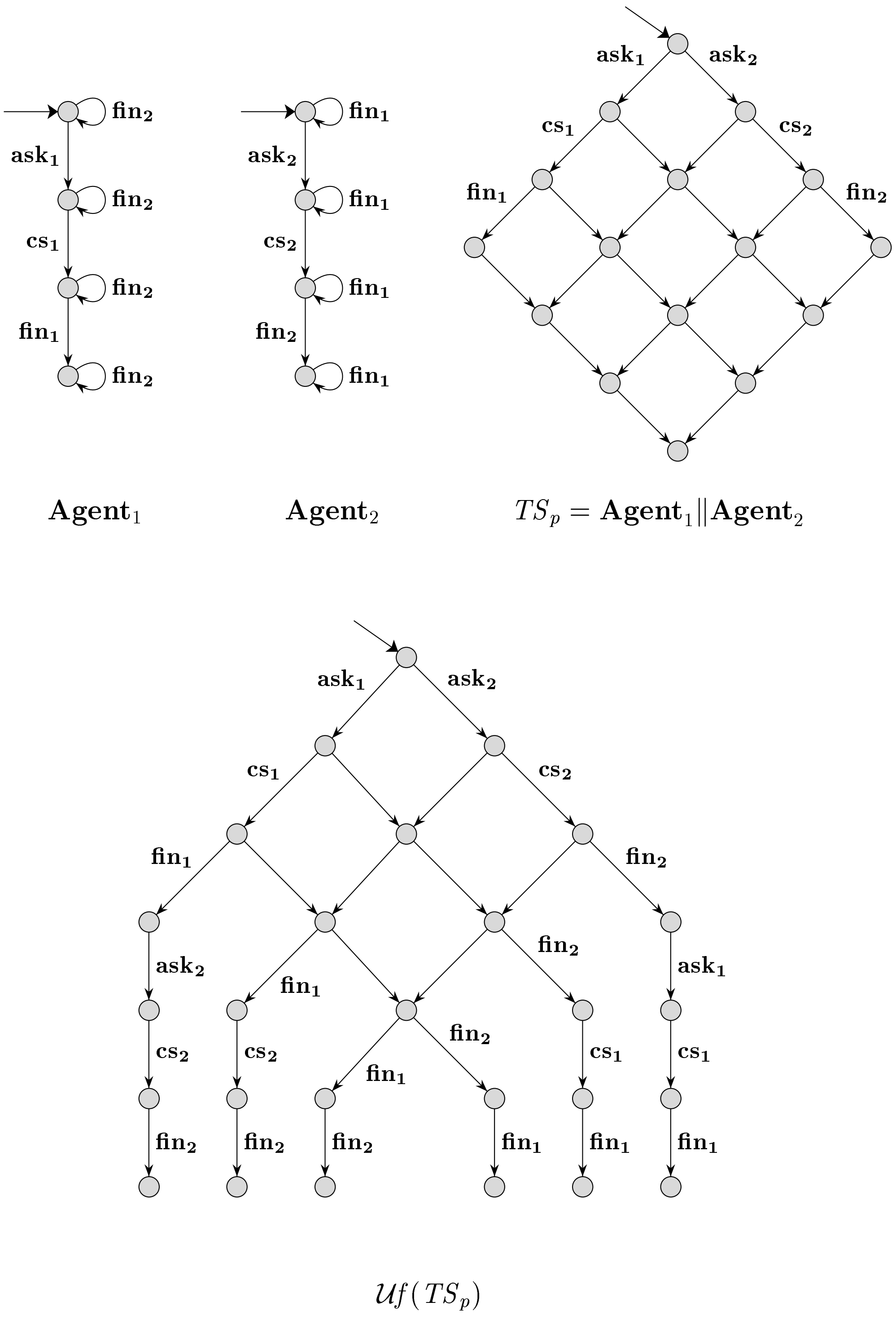
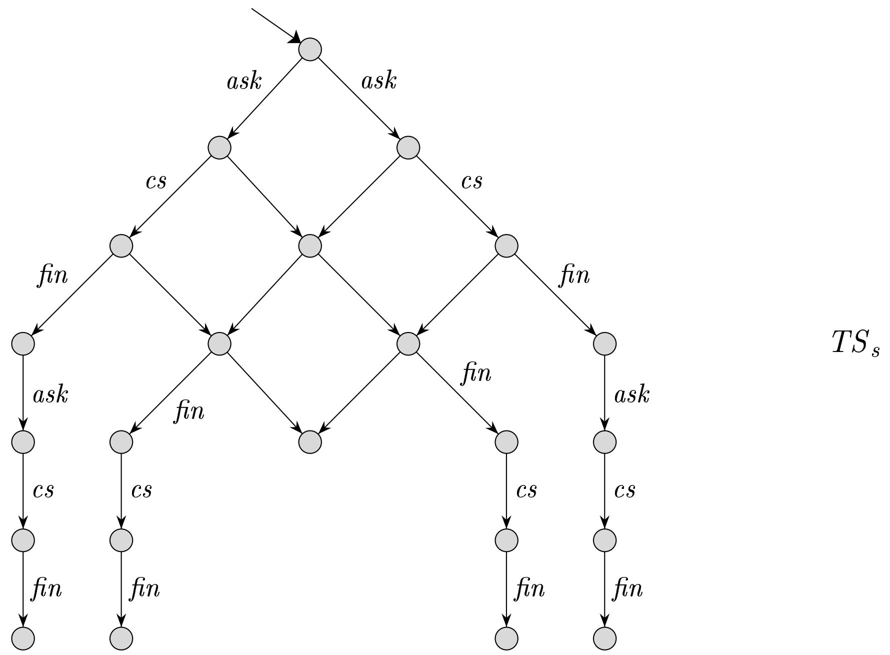


Figure 3.1: A transition system and its unfolding



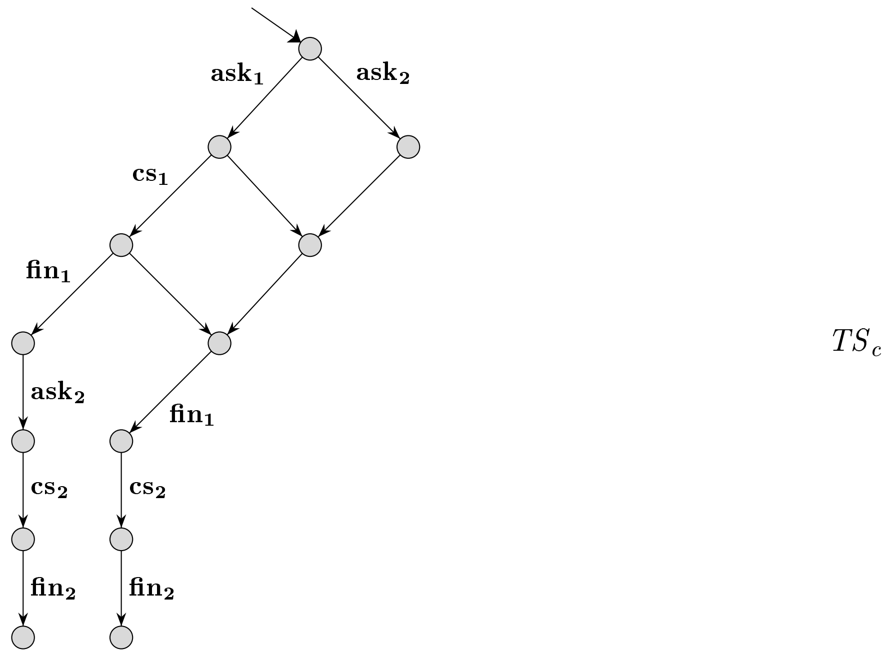


Figure 3.2: Asynchronous specification and controller for TS_p

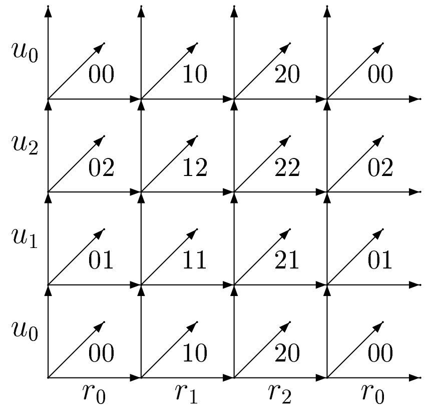


Figure 3.3: The main grid

$col : \mathbb{N}_0 \times \mathbb{N}_0 \longrightarrow C$ ($\mathbb{N}_0$ is the set of natural numbers {0,1,2,...}) which satisfies:

- $col(0,0) = c_{in}$
- $\forall\ (m,n) \in \mathbb{N}_0 \times \mathbb{N}_0.\ \ col(m+1,n) \in R(col(m,n))$ and $col(m,n+1) \in U(col(m,n))$.

For each instance $\mathcal{C}$ of a colouring problem we first construct a pair of *infinite* asynchronous transition systems $(TS_1^{\mathcal{C}}, TS_2^{\mathcal{C}})$ such that $\mathcal{C}$ has a solution iff there exists an asynchronous *simulation* from $TS_1^{\mathcal{C}}$ to $TS_2^{\mathcal{C}}$. We then show how to construct, given $\mathcal{C}$, two *finite* asynchronous transition systems $TS_p^{\mathcal{C}}$ and $TS_s^{\mathcal{C}}$ such that $TS_1^{\mathcal{C}}$ and $TS_2^{\mathcal{C}}$ are isomorphic to $\mathcal{U}f(TS_p^{\mathcal{C}})$ and $\mathcal{U}f(TS_s^{\mathcal{C}})$ respectively. This will show that the model-checking problem is undecidable.

Through the rest of the section fix an instance of the colouring problem $\mathcal{C} = (C, c_{in}, R, U)$ and let c, c' range over C. The associated pair of infinite asynchronous transition systems will be denoted as $TS_1^{\mathcal{C}}$ and $TS_2^{\mathcal{C}}$.

The main part of $TS_1^{\mathcal{C}}$ will look like a two dimensional grid generated by the two sets of events $E_R = \{r_0, r_1, r_2\}$ and $E_U = \{u_0, u_1, u_2\}$ with $E_R \times E_U \subseteq I_1$ where I_1 is the independence relation of $TS_1^{\mathcal{C}}$. This is shown in Figure 3.3. We display only the events concerned and not their labels. We deal with the labels later.

In addition, there will be nine events $\{0,1,2\}^2$. At each grid point at most four such events will be sticking out. For convenience we will often write ij instead of (i,j) for $i,j \in \{0,1,2\}$. At a grid point, the event ij will be enabled if r_i and u_j are enabled at this point. This event will commute with events r_i and u_j enabled at

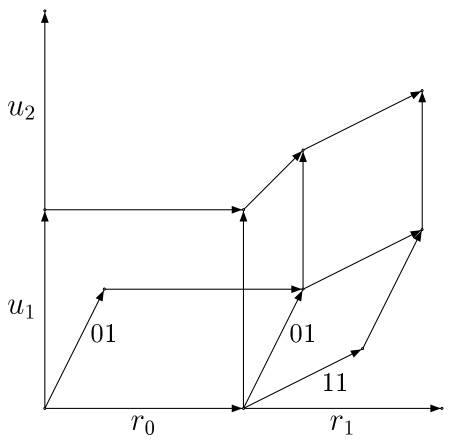


Figure 3.4: A typical neighbourhood of a grid-point

this grid point. It will also commute with the events $i(j+1)$ and $(i+1)j$ enabled at the neighbouring grid points. Here and in what follows addition is taken to be addition modulo 3.

Thus the set of events of $TS_1^{\mathcal{C}}$ is $E_1 = E_r \cup E_u \cup \{ij \mid i,j \in \{0,1,2\}\}$ and its independence relation I_1 is the least symmetric relation of $E_1 \times E_1$ which satisfies :

- $\{r_0, r_1, r_2\} \times \{u_0, u_1, u_2\} \subseteq I_1$
- $ij\ I_1\ r_{i'}$ if $i = i'$
- $ij\ I_1\ u_{j'}$ if $j = j'$
- $ij\ I_1\ i'j'$ if $[(i' = i+1$ and $j = j')$ or $(i = i'$ and $j' = j+1)]$

$TS_1^{\mathcal{C}}$ is such that along any run, an event ij can occur at most once. Thus a typical neighbourhood in $TS_1^{\mathcal{C}}$ will look as in Figure 3.4.

Note that once an event of type ij is performed, one can never get back to the main grid; at most three more events can be performed before reaching a terminal state. These events which stick out of the grid will be used — via a simulation — to check whether the colours assigned to neighbouring grid points are consistent.

The assignment of colours to the grid points will be done in $TS_2^{\mathcal{C}}$. This transition system will look exactly like $TS_1^{\mathcal{C}}$ except that we will use events taken from the set $C \times \{0,1,2\}^2$ instead of $\{0,1,2\}^2$. At a grid point, the event (c, ij) will be enabled if r_i and u_j are enabled at this point. As an exception, at the origin only the event $(c_{in}, 00)$ will be enabled apart from the events r_0 and u_0. In addition the event (c, ij) can wander forward a bit through the independence relation as described below. The crucial point is, the independence relation I_2 of $TS_2^{\mathcal{C}}$ will be used to check for the consistency of the colouring scheme.

The set of events of $TS_2^{\mathcal{C}}$ is $E_2 = E_r \cup E_u \cup \{(c, ij) \mid c \in C,\ i, j \in \{0,1,2\}\}$ We define I_2 to be the least irreflexive and symmetric subset of $E_s \times E_s$ satisfying:

- $\{r_0, r_1, r_2\} \times \{u_0, u_1, u_2\} \subseteq I_2$.
- $r_i\ I_2\ (c, i'j')$ if $i = i'$
- $u_j\ I_2\ (c, i'j')$ if $j = j'$
- $(c, ij)\ I_2\ (c', i'j')$ if $[(i' = i + 1,\ j' = j$ and $c' \in R(c))$ or $(i' = i,\ j' = j + 1$ and $c' \in U(c))]$.

We force $TS_1^{\mathcal{C}}$ and $TS_2^{\mathcal{C}}$ to march together by a suitable choice of labels. Fix $\Sigma = \{r_0, r_1, r_2, u_0, u_1, u_2\} \cup \{0, 1, 2\}^2$. In both the systems the event $x \in E_r \cup E_u$ gets the label x. The events ij in $TS_1^{\mathcal{C}}$ and the events (c, ij) in $TS_2^{\mathcal{C}}$ get the label ij.

Let us define $TS_1^{\mathcal{C}}$ and $TS_2^{\mathcal{C}}$ formally. Let $TS_1^{\mathcal{C}} = (Q_1, E_1, T_1, q_{in}^1, \varphi_1, I_1)$ and $TS_2^{\mathcal{C}} = (Q_2, E_2, T_2, q_{in}^2, \varphi_2, I_2)$ be two Σ-labelled asynchronous transition systems with E_1, φ_1, I_1, E_2, φ_2 and I_2 defined as above. Let $X_r^{(0)} = (r_0 \cdot r_1 \cdot r_2)^*$, $X_r^{(1)} = (r_0 \cdot r_1 \cdot r_2)^* \cdot r_0$ and $X_r^{(2)} = (r_0 \cdot r_1 \cdot r_2)^* \cdot r_0 \cdot r_1$. Similarly, let $X_u^{(0)} = (u_0 \cdot u_1 \cdot u_2)^*$, $X_u^{(1)} = (u_0 \cdot u_1 \cdot u_2)^* \cdot u_0$ and $X_u^{(2)} = (u_0 \cdot u_1 \cdot u_2)^* \cdot u_0 \cdot u_1$. Let

$$Z_1 = \bigcup_{i,j \in \{0,1,2\}} X_r^{(i)} \cdot X_u^{(j)} \cdot ij \cdot r_i \cdot u_j \cdot [(i+1)j \quad + \quad i(j+1)]$$

Z_1 represents representatives of all maximal sequences we want the system to generate. Let $Z_1' = \{y \mid \exists x \in Z_1 : y \sim_1 x\}$, where $\sim_1$ is the trace equivalence relation corresponding to I_1. Set $Z_1'' = \mathrm{Pref}(Z_1')$, the prefixes of words in Z_1'. Z_1'' denotes the set of all sequences we would like $TS_1^{\mathcal{C}}$ to generate. So, set $Q_1 = Z_1'' / \sim_1$ and for each $[x], [x \cdot e] \in Q_1$ where $x \in E_1^*$, $e \in E_1$, let the transition $[x] \xrightarrow{e} [xe]$ belong to T_1. The initial state q_{in}^1 is $[\varepsilon]$.

$TS_2^{\mathcal{C}}$ is defined similarly: For each pair of colours $c, c' \in C$, let

$$R^{(c,c')} = \bigcup_{i,j \in \{0,1,2\}} X_r^{(i)} \cdot X_u^{(j)} \cdot (c, ij) \cdot r_i \cdot u_j \cdot (c', (i+1)j)$$

$$U^{(c,c')} = \bigcup_{i,j\in\{0,1,2\}} X_r^{(i)} \cdot X_u^{(j)} \cdot (c, ij) \cdot r_i \cdot u_j \cdot (c', i(j+1))$$

Now set

$$Z_2 = \bigcup_{i,j\in\{0,1,2\}} [X_r^{(i)} \cdot X_u^{(j)} \cdot (c, ij) \cdot r_i \cdot u_j] \;\cup \bigcup_{c,c'|c'\in R(c)} [R^{(c,c')}] \cup \bigcup_{c,c'|c'\in U(c)} [U^{(c,c')}]$$

Let $Z_2' = \{y \mid \exists x \in Z_2 : y \sim_2 x\}$, where $\sim_2$ is the trace equivalence relation corresponding to I_2. Let $Z_2'' = \text{Pref}(Z_2')$, the prefixes of words in Z_2'. Z_2'' denotes the set of all sequences we would like $TS_2^{\mathcal{C}}$ to generate. Set $Q_2 = Z_2''/\sim_2$ and let us have, for each $[x], [x \cdot e] \in Q_2$ where $x \in E_2^*$, $e \in E_2$, the transition $[x] \xrightarrow{e} [xe]$ in T_2. The initial state q_{in}^2 is $[\varepsilon]$.

Grid points are those states in $TS_1^{\mathcal{C}}$ and $TS_2^{\mathcal{C}}$ reached after executing a sequence in $(E_r \cup E_u)^*$.

Lemma 3.1 *For any colouring problem $\mathcal{C}$, there is a solution for $\mathcal{C}$ iff there is a simulation from $TS_1^{\mathcal{C}}$ to $TS_2^{\mathcal{C}}$.*

Proof

($\Rightarrow$) Let $col : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow C$ be a solution for $\mathcal{C}$. Now there is a simulation which works as follows. Map the grid-points of $TS_1^{\mathcal{C}}$ to the grid-points of $TS_2^{\mathcal{C}}$. This is easily achieved by mapping the states of the form $[x]$ (where $x \in (E_r \cup E_u)^*$ to $[x]$. If at a grid-point, r_i and u_j events are enabled, then map the outgoing edge ij from this grid-point to the (c, ij) event in the corresponding grid-point of $TS_2^{\mathcal{C}}$, where c is the colour assigned by col to that grid-point. That is, at a grid-point $[x]$, if r_i and u_j are enabled, we map the transition $[x] \xrightarrow{ij} [x \cdot ij]$ to $[x] \xrightarrow{(c,ij)} [x \cdot (c, ij)]$. We extend the function to map other occurrences of the same event to appropriate transitions. It is now easy to see that this defines a simulation. The simulation will preserve independence of events since col is a solution for $\mathcal{C}$.

($\Leftarrow$) Let $f : TS_1^{\mathcal{C}} \rightarrow TS_2^{\mathcal{C}}$ be a simulation. First, it is easy to argue that the grid-points of $TS_1^{\mathcal{C}}$ must get mapped to the grid-points of $TS_2^{\mathcal{C}}$. This follows from the fact that f must preserve the label of events that are mapped. Now, we can assign colours to the grid-points as follows: at any grid-point, if r_i and u_j are enabled, then the colour for that grid-point is c where f maps the outgoing edge ij event to (c, ij). It follows easily from the construction and the fact that f preserves the independence of events, that the colouring defined is a solution to $\mathcal{C}$.

$\square$

All that remains now is to exhibit finite state transition systems $TS_p^{\mathcal{C}}$ and $TS_s^{\mathcal{C}}$ such that their unfoldings are isomorphic to $TS_1^{\mathcal{C}}$ and $TS_2^{\mathcal{C}}$ respectively. The systems $TS_p^{\mathcal{C}}$ and $TS_s^{\mathcal{C}}$ are Σ-labelled transition systems, (where $\Sigma = \{r_0, r_1, r_2, u_0, u_1, u_2\} \cup \{0,1,2\}^2$) defined as follows:
$TS_p^{\mathcal{C}} = (Q_p, E_p, T_p, q_{in}^p, \varphi_p, I_p)$ where

- $E_p = E_1$, $\varphi_p = \varphi_1$ and $I_p = I_1$ as defined above We denote by D_p the dependence relation: $D_p = (E_p \times E_p) \setminus I_p$.
- $Q_p = \{r_0, r_1, r_2\} \times \{u_0, u_1, u_2\} \times 2^{\{0,1,2\}} \times 2^{\{0,1,2\}} \times 2^{\{0,1,2\}^2}$

  A state (R, U, L_R, L_U, X) of the plant contains the following information:

  - R encodes which r_i event is enabled and U encodes which u_j event is enabled.
  - L_R encodes the set of all i such that events ij' may be permitted and L_U encodes the set of all j such that events $i'j$ may be permitted. Together they encode exactly which ij events are permitted at a grid-point: an event ij is permitted iff $i \in L_R$ and $j \in L_j$.
  - X encodes the set of all ij events that have occurred so far.
- $q_{in}^p = (r_0, u_0, \{0\}, \{0\}, \emptyset)$
- Let a typical member of Q_p be denoted as a tuple (R, U, L_R, L_U, X).

  T_p is defined as follows:

  - $(R, U, L_R, L_U, X) \xrightarrow{r_i} (R', U', L'_R, L'_U, X')$ if
    $R = r_i$, $(\nexists i'j' \in X : i \neq i')$, $R' = r_{i+1}$, $U' = U$, $L'_U = L_U$, $L'_R = (L_R \setminus \{i-1\}) \cup \{i+1\}$ and $X' = X$
  - $(R, U, L_R, L_U, X) \xrightarrow{u_j} (R', U', L'_R, L'_U, X')$ if
    $U = u_j$, $(\nexists i'j' \in X : j \neq j')$, $U' = u_{j+1}$, $R' = R$, $L'_R = L_R$, $L'_U = (L_U \setminus \{j-1\}) \cup \{j+1\}$ and $X' = X$

– $(R, U, L_R, L_U, X) \xrightarrow{ij} (R', U', L'_R, L'_U, X')$ if
$i \in L_R$, $j \in L_U$, $(\not\exists i'j' \in X : i'j' D_p ij)$, $R' = R$, $U' = U$, $X' = X \cup \{ij\}$,
$L'_R = L_R$ and $L'_U = L_U$

It should be clear now how the definitions of the plant transitions work. Note that $X = \emptyset$ at any grid point.
$TS^{\mathcal{C}}_s$ is defined as $TS^{\mathcal{C}}_s = (Q_s, E_s, T_s, q^s_{in}, \varphi_s, I_s)$ where

- $E_s = E_2$, $\varphi_s = \varphi_2$ and $I_s = I_2$ as defined above We denote by D_s the dependence relation: $D_s = (E_s \times E_s) \setminus I_s$.
- $Q_s = \{r_0, r_1, r_2\} \times \{u_0, u_1, u_2\} \times 2^{\{0,1,2\}} \times 2^{\{0,1,2\}} \times 2^{C \times \{0,1,2\}^2} \times \{init, *\}$
- $q^s_{in} = (r_0, u_0, \{0\}, \{0\}, \emptyset, init)$
- Let a typical member of Q_s be denoted as a tuple (R, U, L_R, L_U, X, S). T_s is defined as follows :
  – $(R, U, L_R, L_U, X, S) \xrightarrow{r_i} (R', U', L'_R, L'_U, X', S')$ if
  $R = r_i$, $(\not\exists i'j' \in X : i \neq i')$, $R' = r_{i+1}$, $U' = U$, $L'_U = L_U$, $L'_R = (L_R \setminus \{i-1\}) \cup \{i+1\}$, $X' = X$ and $S' = *$
  – $(R, U, L_R, L_U, X, S) \xrightarrow{u_j} (R', U', L'_R, L'_U, X', S')$ if
  $U = u_j$, $(\not\exists i'j' \in X : j \neq j')$, $U' = u_{j+1}$, $R' = R$, $L'_R = L_R$, $L'_U = (L_U \setminus \{j-1\}) \cup \{j+1\}$, $X' = X$ and $S' = *$
  – $(R, U, L_R, L_U, X, S) \xrightarrow{(c,ij)} (R', U', L'_R, L'_U, X', S')$ if
  $(S = init \Rightarrow c = c_{in})$, $i \in L_R$, $j \in L_U$, $(\not\exists (c', i'j') \in X : (c', i'j') D(c, ij))$, $R' = R$, $U' = U$, $X' = X \cup \{(c, ij)\}$, $L'_R = L_R$, $L'_U = L_U$ and $S' = S$

The specification is constructed in almost the same way as the plant, except that the L_R and L_U components encode the (c, ij) events enabled and the independence relation of the (c, ij) events are constrained by the given colouring problem. We also keep track in a new component S whether the last grid-point seen was $(0, 0)$ or not. If it is, then we only allow the c_{in} event to occur.

It is tedious but straightforward to see that $\mathcal{U}f(TS^{\mathcal{C}}_p)$ and $\mathcal{U}f(TS^{\mathcal{C}}_s)$ are isomorphic to $TS^{\mathcal{C}}_1$ and $TS^{\mathcal{C}}_2$. Also, from a given $\mathcal{C}$, we can construct $TS^{\mathcal{C}}_p$ and $TS^{\mathcal{C}}_s$ effectively. From Lemma 3.1 we then have

Theorem 3.1 *The problem of uniformly determining the existence of a simulation from the unfolding of a finite asynchronous transition system to the unfolding of another is undecidable.* □

Next we show that the problem of checking for an asynchronous simulation reduces to that of checking for the existence of an asynchronous controller. Given TS_p and TS_s, we construct TS'_p and TS'_s such that there exists a simulation from $\mathcal{U}f(TS_p)$ into $\mathcal{U}f(TS_s)$ iff there is a controller for (TS'_p, TS'_s). It will turn out that $\mathcal{U}f(TS_p)$ and $\mathcal{U}f(TS_s)$ can be embedded into $\mathcal{U}f(TS'_p)$ and $\mathcal{U}f(TS'_s)$. Further, it will turn out that if (TS'_p, TS'_s) has a controller, say TS'_c, then it would have to be the trivial controller which allows *all* system moves. Hence $\mathcal{U}f(TS'_p \| TS'_c)$ will be isomorphic to $\mathcal{U}f(TS'_p)$. Hence, if (TS'_p, TS'_s) has a controller, it would imply that there is a simulation from $\mathcal{U}f(TS'_p)$ to $\mathcal{U}f(TS'_s)$, from which we will show how to extract a simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$. To prove the converse, we will show how any simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$ easily extends to a simulation from $\mathcal{U}f(TS'_p)$ to $\mathcal{U}f(TS'_s)$. This will show that the completely non-restrictive controller is a valid controller for (TS'_p, TS'_s).

Let $TS = (Q, E, T, q_{in}, \varphi)$ be a Σ-labelled asynchronous transition system. We then define its *augmented* version $Aug(TS)$, a Σ'-labelled asynchronous transition system, below, where $\Sigma' = \Sigma \cup \{\star\}$ where $\star$ is a new label not in Σ. First, assume without loss of generality that Σ is disjoint from Q_p, Q_s, E_p and E_s. Then $Aug(TS) = (Q', E', T', q'_{in}, \varphi', I')$ is defined as follows:

- $Q' = Q \cup \{X \mid X \text{ is a non-empty subset of } \Sigma'\} \cup \{q_\star, q_{\star\star}\}$
- $E' = E_p \cup \Sigma' \cup \{\star'\}$
- $\varphi'(e_1) = \varphi_p(e_1)$, if $e_1 \in E_p$; $\varphi'(a) = a$, if $a \in \Sigma'$; $\varphi'(\star') = \star$
- $q'_{in} = q_{in}$.
- $T' = T \cup \{(q_1, a, \{a\}) \mid q_1 \in Q \text{ and } a \in \Sigma'\} \cup$ $\{(X, a, Y) \mid X, Y \text{ are non-empty subsets of } \Sigma \text{ and } a \notin X \text{ and } Y = X \cup \{a\}\} \cup$ $\{(q_1, \star', q_\star) \mid q_1 \in Q\} \cup \{(q_\star, \star', q_{\star\star})\}$
- $I'_p = I_p \cup \{(a, b) \mid a \neq b \text{ and } a, b \in \Sigma'\}$.

We can prove the following:

Lemma 3.2 *For any two asynchronous transition systems TS_p and TS_s, there is a simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$ iff there is a controller for $(Aug(TS_p), Aug(TS_s))$.*

Proof

(⇒) Let $f : \mathcal{U}f(TS_p) \to \mathcal{U}f(TS_s)$ be a simulation. Consider $TS_c = Aug(TS_p)$. Then $\mathcal{U}f(Aug(TS_p))$ and $\mathcal{U}f(Aug(TS_p))\| TS_c)$ are clearly isomorphic. Now it is easy to see that f can be extended to a simulation from $\mathcal{U}f(Aug(TS_p))$ to $\mathcal{U}f(Aug(TS_s))$ by mapping Σ'-events to corresponding Σ'-events and $\star'$-events to corresponding $\star'$-events.

(⇐) Let TS_c be a controller for $(Aug(TS_p), Aug(TS_s))$. Let g be a simulation from $\mathcal{U}f(Aug(TS_p)\| TS_c)$ to $\mathcal{U}f(Aug(TS_s))$. First, we can show that TS_c cannot restrict any system move of $Aug(TS_p)$.

Claim If (q_p, q_c) is reachable in $Aug(TS_p)\| TS_c$ and $q_p \xrightarrow{e} q_p'$, then $\exists q_c'$: $q_c \xrightarrow{e} q_c'$ in TS_c.

The claim can be checked as follows. If q_p is in Q_p, then we know that some event from (q_p, q_c), say e', must be enabled in $Aug(TS_p)\| TS_c$. Now, the corresponding event $\varphi(e')$ is also enabled. We can then argue that if any one Σ'-event is enabled, then all of them must be enabled (using the nonblocking property of the controller and the fact that it preserves independence of events). Using the properties of a controller, it follows that all events from (q_p, q_c) must be enabled in the controlled plant. If q_p is not in Q_p, then also it is easy to check the claim.

End of claim

It therefore follows that $\mathcal{U}f(Aug(TS_p)\| TS_c)$ is isomorphic to $\mathcal{U}f(Aug(TS_p))$. We can also show that g maps the $\mathcal{U}f(TS_p)$ fragment of $\mathcal{U}f(Aug(TS_p))$ to the $\mathcal{U}f(TS_s)$ fragment of $\mathcal{U}f(Aug(TS_s))$, using the fact that two consecutive $\star$-labelled events are enabled only from the states of $\mathcal{U}f(TS_p)$. Hence a restriction of g will give a simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$.

□

This leads to the second main result of this chapter.

Theorem 3.2 *The problem of uniformly determining if a pair of finite asynchronous transition systems admits an asynchronous controller is undecidable.* □

From our constructions above it is easy to deduce that the problem of uniformly determining if a pair of finite transition systems (TS_p, TS_s) admits a *finite* controller is undecidable. This holds since in the reduction from the undecidable simulation problem to the controller problem, our plant-specification pair is such that it admits a controller iff it admits a finite controller (namely the trivial one that is isomorphic to the plant).

Our undecidability result goes through even for the restricted class of transition systems called product transition systems. The main details of the construction of product transition systems whose unfoldings will be the same as we require, are given in the Appendix. Consequently, the undecidability extends to other models — for example, when the plant and specification are presented as labelled 1-safe Petri nets.

Yet another restriction one can consider is the class of asynchronous transition systems where there is an underlying independence over the labels Σ which respects the independence of events, i.e.: $TS = (Q, E, T, q_{in}, \varphi, I, \widehat{I})$ where $TS = (Q, E, T, q_{in}, \varphi, I)$ is an asynchronous transition system and $\widehat{I} \subseteq \Sigma \times \Sigma$ is irreflexive and symmetric and $\forall e, e' \in E,\ e\ I\ e' \Rightarrow \varphi(e)\ \widehat{I}\ \varphi(e')$.

It is easy to see that the class of systems and specifications used in the undecidability result for simulation fall within this class. Hence checking existence of simulation for this class is also undecidable. We can also show that checking for the existence of a controller for this class is undecidable. The reduction is from the simulation problem for this class and the details are given in the Appendix.

Chapter 4

Temporal logics, trees and automata

The Beaver brought paper, portfolio, pens,
And ink in unfailing supplies:
While strange creepy creatures came out of their dens,
And watched them with wondering eyes.

— *The Hunting of the Snark*, Lewis Caroll

In this chapter we review some temporal logics (the branching-time temporal logics CTL and $\mathsf{CTL}^\star$ and the linear-time temporal logic LTL) and also review labeled infinite trees and automata working on them. All these concepts and definitions (except that of alternating automata) will be used in the next chapter where we consider the synthesis and control problems for branching-time logics. Also, tree-automata will be extensively used in Chapter 6 where we consider the distributed control-synthesis problem. Though we won't really work with synthesis/control problems involving LTL, we use LTL in some crucial lower-bound results in the next chapter and it will also be a natural way to view the specification mechanisms in Chapter 6.

4.1 Linear-time temporal logic LTL

We introduce the temporal logics we need in the next two chapters briefly here. We refer the reader to [Eme90] for a more comprehensive and gentle introduction.

Linear-time temporal logic (LTL) is a temporal logic designed to specify properties of infinite sequences. The models for this logic are ω-sequences of letters from Σ, where $\Sigma = 2^{AP}$ for a set of atomic propositions AP.

Let us fix a set of atomic propositions AP. Then the formulas of LTL over AP is defined to be the least set such that:

- For each $p \in AP$, p is a formula
- If φ and ψ are formulas then so are $\neg\varphi$, $\varphi \vee \psi$, $\mathsf{X}\varphi$ and $\varphi\mathsf{U}\psi$

The modalities X and U stand for "Next" and "Until" respectively — i.e. $\mathsf{X}\varphi$ means that the suffix sequence starting from next position satisfies φ while $\varphi\mathsf{U}\psi$ means that the formula ψ holds somewhere down the sequence and till that point φ holds.

Let $\sigma \in (2^AP)^\omega$ be an ω-sequence of subsets of AP. If $\sigma = \sigma_0 \cdot \sigma_1 \cdot \ldots$, then we denote by $\sigma[i]$ the suffix of σ starting from the i^{th} position in the sequence — i.e. $\sigma[i] = \sigma_i \cdot \sigma_{i+1} \ldots$.

Formally, the semantics of when a formula φ is satisfied in a model $\sigma \in (2^AP)^\omega$, denoted $\sigma \models \varphi$, is defined inductively on the structure of φ as follows:

- $\sigma \models p$ iff $\sigma = \sigma_0 \cdot \sigma_1 \ldots$ and $p \in \sigma_0$
- $\sigma \models \varphi \vee \psi$ iff $\sigma \models \varphi$ or $\sigma \models \psi$
- $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$
- $\sigma \models \mathsf{X}\varphi$ iff $\sigma[1] \models \varphi$
- $\sigma \models \varphi\mathsf{U}\psi$ iff there is a $j \in \mathbb{N}_0$ such that $\sigma[j] \models \psi$ and for every $0 \leq i < j$, $\sigma[i] \models \varphi$

We use the following abbreviations as well:

- $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$ ("and").
- $\mathsf{F}\varphi = \mathbf{true}\mathsf{U}\varphi$ ("eventually").
- $\mathsf{G}\varphi = \neg\mathsf{F}\neg\varphi$ ("always").

4.2 Branching-time temporal logics CTL and $\mathsf{CTL}^\star$

In the branching time logic $\mathsf{CTL}^\star$, a path quantifier, E denoting "for some path" (or A denoting "for all paths"), can prefix an assertion composed of an arbitrary combination of the linear time operators X ("next") and U ("until"). CTL and $\mathsf{CTL}^\star$ are interpreted on Kripke structures which we define below.

$\mathsf{CTL}^\star$ has two types of formulas: *state formulas*, whose meaning is related to a specific state, and *path formulas*, whose meaning is related to a specific path. Let AP be a set of atomic propositions. A $\mathsf{CTL}^\star$ state formula is either:

- p, for $p \in AP$.
- $\neg\varphi$ or $\varphi \vee \psi$, where φ and ψ are $\mathsf{CTL}^\star$ state formulas.
- $\mathsf{E}\varphi$ or $\mathsf{A}\varphi$ where φ is a $\mathsf{CTL}^\star$ path formula.

A path formula is either:

- A $\mathsf{CTL}^\star$ state formula.
- $\neg\varphi$ or $\varphi \vee \psi$ or $\mathsf{X}\varphi$, or $\varphi\mathsf{U}\psi$, where φ and ψ are $\mathsf{CTL}^\star$ path formulas.

The logic $\mathsf{CTL}^\star$ consists of the set of state formulas generated by the above rules.

The logic CTL ("computation tree logic") is a restricted subset of $\mathsf{CTL}^\star$. In CTL, the temporal operators X and U must be immediately preceded by a path quantifier. Formally, it is the subset of $\mathsf{CTL}^\star$ obtained by restricting the path formulas in the above definition to be $\mathsf{X}\varphi$ or $\varphi\mathsf{U}\psi$, where φ and ψ are CTL state formulas. In other words, the set of CTL formulas is the smallest set such that:

- For each $p \in P$, p is a CTL formula.
- If φ and φ' are CTL formulas, then so are $\neg\varphi$, $\varphi \vee \varphi'$, $\mathsf{EX}\varphi$, $\mathsf{AX}\varphi$, $\mathsf{E}\varphi\mathsf{U}\varphi'$ and $\mathsf{A}\varphi\mathsf{U}\varphi'$.

The semantics of $\mathsf{CTL}^\star$ (and its sub-logic CTL) is defined with respect to a *(Kripke) structure* $S = \langle AP, W, R, w_0, L\rangle$, where AP is the set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $(w, w') \in R$), w_0 is an initial state, and $L : W \to 2^{AP}$ maps each state to a set of atomic propositions true in this state. We sometimes say $R(w, w')$ to mean that $(w, w') \in R$. If $R(w, w')$ holds, we say that w' is a successor of w . A *path* of S is an infinite sequence $\pi = w^0, w^1, \ldots$ of states such that for every $i \geq 0$, we have $R(w^i, w^{i+1})$. The suffix $w^i, w^{i+1}, \ldots$ of π is denoted by π^i. We use $w \models \varphi$ to indicate that a state formula φ holds at state w, and we use $\pi \models \varphi$ to indicate that a path formula φ holds at path π (assuming a structure S). The relation $\models$ is inductively defined as follows.

- For an atomic proposition $p \in AP$, we have $w \models p$ iff $p \in L(w)$
- $w \models \neg\varphi$ iff $w \not\models \varphi$.
- $w \models \varphi \vee \psi$ iff $w \models \varphi$ or $w \models \psi$.
- $w \models \mathsf{E}\varphi$ iff there exists a path $\pi = w_0, w_1, \ldots$ such that $w_0 = w$ and $\pi \models \varphi$.
- $w \models \mathsf{A}\varphi$ iff for every path $\pi = w_0, w_1, \ldots$ such that $w_0 = w$, $\pi \models \varphi$.
- $\pi \models \varphi$ for a state formula φ iff $\pi = w^0, w^1, \ldots$ and $w^0 \models \varphi$
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$.
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $\pi \models \psi$.
- $\pi \models \mathsf{X}\varphi$ iff $\pi^1 \models \varphi$.
- $\pi \models \varphi \mathsf{U} \psi$ iff there exists $j \geq 0$ such that $\pi^j \models \psi$ and for all $0 \leq i < j$, we have $\pi^i \models \varphi$.

We say that a Kripke structure S satisfies a $\mathsf{CTL}^\star$ (or CTL) formula φ, denoted $S \models \varphi$, if $w_0 \models \varphi$ where w_0 is the initial state of S.

We use abbreviations $\varphi \wedge \psi$, $\mathsf{F}\varphi$, $\mathsf{G}\varphi$ to mean the same as in LTL.

4.3 Trees

A (rooted directed) *tree* is a directed acyclic graph $T = (N, E)$, where N is a set of nodes and $E \subseteq N \times N$ is an edge-relation, that has a designated root r which doesn't have a parent (i.e. there is no $v \in N$ such that $(v, r) \in E$) and every other node of the tree has a unique parent and is reachable from r.

For a finite set Υ, we note that $T_\Upsilon = (\Upsilon^*, E_\Upsilon)$, where $E_\Upsilon = \{(x, x.c) \mid x \in \Upsilon^* \text{ and } c \in \Upsilon\}$, is a tree — we refer to this as the *full* Υ*-tree*. Also, consider the graph $T = (N, E)$ where $N \subseteq \Upsilon^*$ and $E \subseteq E_\Upsilon$ such that, if $v.c \in N$ with $v \in \Upsilon^*$ and $c \in \Upsilon$, then $v \in N$. Then T is a tree (it's a subtree of T_Υ) and we call this an Υ-*tree*. The root of an Υ-tree is the empty word ε.

Let $T = (N, E)$ be a tree. For every $v \in T$, the set $succ_T(v) = \{v' \in N \mid (v, v') \in E\}$ is the set of children (successors) of v. Where T is clear from the context, we drop the subscript T and write $succ(v)$. We usually consider trees that satisfy the condition that $succ(v) \neq \emptyset$ for every v in T.

We also associate a direction $dir(v) \in \Upsilon$ with each node v of an Υ-tree T. A designated element $c_0 \in \Upsilon$ is the direction of ε. For each non-root node $v.c$ with $c \in \Upsilon$, we set $dir(v.c) = c$.

A path π of any tree $T = (N, E)$ is an infinite sequence of nodes of the tree $\pi = v_0 v_1 \ldots$ such that v_0 is the root of T and for each $i \in \mathbb{N}_0$, v_{i+1} is a successor of v_i in T. Finally, given a set Σ, a Σ-labeled tree is a pair (T, V) where T is a tree and $V : T \to \Sigma$ is a labeling function that labels each node of the tree with a symbol of Σ.

Of special interest to us are 2^{AP}-labeled trees, where AP is a finite set of propositions. We call such trees *computation trees* and we sometimes interpret $\mathsf{CTL}^\star$ formulas with respect to them. Formally, a computation tree (T, V), where $T = (N, E)$, satisfies a $\mathsf{CTL}^\star$ formula φ if φ is satisfied in the structure $\langle AP, N, E, r, V\rangle$, where r is the root of T.

Kripke structures can be unfolded into trees just as transition systems were unfolded into trees in Chapter 2. Formally, for a Kripke structure $S = \langle AP, W, R, w_0, L\rangle$, we can associate a 2^{AP}-labelled W-tree (T_S, V_S) where $T_S \subseteq W^*$ is the least set such that:

- $\varepsilon \in T_S$; $dir(\varepsilon) = w_0$
- For every $x \in W^*$, if $x \in T_S$ and $dir(x) = w$ and $(w, w') \in R$, then $x.w' \in T_S$. (Also, by convention, $dir(x.w') = w'$)

It turns out that $\mathsf{CTL}^\star$ formulas cannot distinguish between a Kripke structure and its unwinding:

Proposition 4.1 *Let φ be a $\mathsf{CTL}^\star$ formula and let S be a Kripke structure. Then $S \models \varphi$ iff $(T_S, V_S) \models \varphi$, where (T_S, V_S) is the unwinding of S.*

Proof Follows easily by induction on the structure of φ. □

Hence, we switch freely between Kripke structures and their unfoldings while interpreting $\mathsf{CTL}^\star$ formulas on them.

4.4 Automata on trees

Let us now turn to automata working over Σ-labelled Υ-trees. Let us fix finite sets Σ and Υ. We introduce here Büchi, Rabin and parity tree automata (see [Tho90, Tho97] for more details). Our treatment of automata are a bit non-standard (for the sake of convenience). We define them on Υ-trees while in the literature, automata are usually defined on fixed-arity complete trees. However, one can easily go back and forth between these definitions.

A *non-deterministic tree automaton* over Σ-labeled Υ-trees is $A = \langle \Sigma, Q, \delta, q_0, \mathcal{F} \rangle$ where Σ is the finite alphabet we have fixed. Q is a finite set of states. For each $X \subseteq \Upsilon$, let $\mathcal{G}_X$ denote the set of all functions from X to Q. Also, let $\mathcal{G}$ denote the union of all sets $\mathcal{G}_X$ where $X \subseteq \Upsilon$. Then $\delta : Q \times \Sigma \times 2^\Upsilon \to 2^\mathcal{G}$ is a transition function that maps a state, a letter, and a subset X of Υ to a subset of $\mathcal{G}_X$. In other words, if $q \in Q$, $a \in \Sigma$, $X \subseteq \Upsilon$, then $\delta(q, a, X)$ is a set of functions from X to Q — each such function gives a set of possible propagation of states to the children of the node being read. $q_0 \in Q$ is the initial state, and $\mathcal{F}$ is an acceptance condition (or a winning condition) that depends on the kind of automata we consider:

(Büchi) $\mathcal{F}$ is a subset of Q

(Rabin) $\mathcal{F} = \{(R_1, G_1), \ldots, (R_m, G_m)\}$ is a set of pairs of subsets of Q (i.e. for each i, $R_i, G_i \subseteq Q$).

(parity) $\mathcal{F}$ is a function $\mathcal{F} : Q \to \{0, \ldots, h\}$ for some $h \in \mathbb{N}$ (the set $\{0, \ldots, h\}$ is called the set of *colours*).

Automata with Büchi, Rabin and parity winning conditions are called Büchi, Rabin and parity automata, respectively.

Let $\alpha = x_0 x_1 \ldots \in Q^\omega$ be an infinite sequence of states. Then we denote the states that occur infinitely often in α as:

$$inf(\alpha) = \{q \in Q \mid \text{there are infinitely many } i \in \mathbb{N} \text{ such that } x_i = q\}.$$

The notion of when α *satisfies* a winning condition $\mathcal{F}$ is defined depending on the kind of winning condition:

(Büchi) α is accepting if $\mathit{inf}(\alpha) \cap \mathcal{F} \neq \emptyset$ — i.e. α meets $\mathcal{F}$ infinitely often.

(Rabin) Let $\mathcal{F} = \{(R_1, G_1), \ldots, (R_m, G_m)\}$. Then α is accepting if $\exists i \in \{1, \ldots, m\}$ such that $\mathit{inf}(\alpha) \cap R_i = \emptyset$ and $\mathit{inf}(\alpha) \cap G_i \neq \emptyset$ — i.e. if there is a pair (R_i, G_i) in $\mathcal{F}$ such that α meets G_i infinitely often and meets R_i only finitely often.

(parity) Let $\mathcal{F} : Q \to \{0, \ldots, h\}$ for some $h \in \mathbb{N}$. Then α is accepting if $\min(\mathcal{F}(\mathit{inf}(\alpha)))$ is even — i.e. the smallest colour met infinitely often in α is even.

Let (T, V) be a Σ-labeled Υ-tree (hence $T \subseteq \Upsilon^*$). Let $\mathit{succ\_dir}(x) = \{\mathit{dir}(y) \mid y \in \mathit{succ}_T(x)\}$ denote the directions of the children of x.

A *run* of the automaton A over (T, V) is a Q-labeled tree (T, ρ) such that:

- $\rho(\varepsilon) = q_0$
- Let $x \in T$ and $C = \mathit{succ\_dir}(x)$. Then there is a function $g : C \to Q$ in $\delta(q, V(x), C)$ such that for each $c' \in C$, $\rho(x.c') = g(c')$.

A path π of a run (T, ρ) is said to be accepting if it satisfies the acceptance condition as defined above. The run (T, ρ) itself is said to be accepting if all paths of it are accepting. A Σ-labeled tree (T, V) is accepted by A if there is an accepting run of A over (T, V). The language accepted by A is the set of all Σ-labeled trees accepted by A.

4.5 Alternating tree automata

Let us now define *alternating tree automata* which are a generalization of non-deterministic tree automata. While non-deterministic automata can guess a set of successor states and send one copy of itself along the subtrees rooted at each of its children, an alternating automaton can pass *several* copies of itself to a single child as well.

For a set X, let $\mathcal{B}^+(X)$ denote the set of positive boolean formulas formed using elements in X — i.e $\mathcal{B}^+(X)$ ::- true $\mid$ false $\mid x \mid \phi \vee \phi' \mid \phi \wedge \phi'$ where $x \in X$ and ϕ and ϕ' are in $\mathcal{B}^+(X)$. For a set $Y \subseteq X$, we say that Y satisfies $\theta \in \mathcal{B}^+(X)$ if setting elements in Y to true (and the elements not in Y to false) satisfies θ, with the usual interpretation of $\vee$ and $\wedge$.

An *alternating tree automaton* over Σ-labeled Υ-trees is $A = \langle \Sigma, Q, \delta, q_0, \mathcal{F} \rangle$ where Σ, Q, q_0 and $\mathcal{F}$ are as in a non-deterministic automaton and $\delta : Q \times \Sigma \times 2^{\Upsilon} \to \mathcal{B}^+(Q \times \Upsilon)$ is a transition function which satisfies the following condition: it must map a state, a letter, and a subset C of Υ to a boolean formula involving states along with the directions in C. In other words, if $q \in Q$, $a \in \Sigma$, $C \subseteq \Upsilon$, then $\delta(q, a, C)$ is a boolean formula in $\mathcal{B}^+(Q \times C)$.

Let (T, V) be a Σ-labeled tree. A *run* of the automaton A over (T, V) is a $(T \times Q)$-labeled tree (T_ρ, ρ) where T_ρ is a tree that needn't be isomorphic to T. Intuitively, the label of a node y in T_ρ being (x, q) represents that the run at that node is reading the node x of the tree T and is in state q. Let the root of T_ρ be r. (T_ρ, ρ) is a run if ρ satisfies the following conditions:

- $\rho(r) = (\varepsilon, q_0)$
- Let $y \in T_\rho$ and $\rho(y) = (x, q)$. Let $C = \mathit{succ\_dir}(x)$ and $\delta(q, V(x), C) = \theta$ where θ is a formula in $\mathcal{B}^+(Q \times C)$. Let $Y \subseteq Q \times C$ be the set of all (q', c') such that there is a child y' of y with $\rho(y') = (q', x.c')$. Then we require that Y satisfies the formula θ.

A path π of a run (T_ρ, ρ) is said to be accepting if the sequence of Q-components of the labels of π satisfies the acceptance condition. The run (T_ρ, ρ) is accepting if all paths it it are accepting. A Σ-labeled tree (T, V) is accepted by A if there is an accepting run of A over (T, V) and the language accepted by A is the set of all Σ-labeled Υ-trees accepted by A.

Example 4.1 Let $\Upsilon = \{c, d, e\}$.

Consider the automaton $A = \langle \Sigma, \{q_0, q_a, q_b, q_b'\}, \delta, q_0, \mathcal{F} \rangle$ where $\Sigma = \{a, b\}$ and $\mathcal{F}$ is a Büchi winning condition with $\mathcal{F} = \{q_b\}$.

The transition function is defined as follows: For any $C \subseteq \Upsilon$:

$$\delta(q_0, a, C) = \delta(q_0, b, C) = \bigwedge_{c \in C} ((q_a, c) \wedge (q_b, c) \wedge (q_0, c))$$

$$\delta(q_a, b, C) = \bigwedge_{c \in C} (q_a, c) \quad ; \quad \delta(q_a, a, C) = \mathit{true}$$

$$\delta(q_b, a, C) = \bigwedge_{c \in C} (q_b, c) \quad ; \quad \delta(q_b, b, C) = \bigwedge_{c \in C} (q'_b, c)$$

$$\delta(q'_b, a, C) = \bigwedge_{c \in C} (q'_b, c) \quad ; \quad \delta(q'_b, b, C) = \bigwedge_{c \in C} (q_b, c)$$

The state q_0 propagates itself to all children of the current node and hence walks down all paths. At each node, it also spawns a copy of the automaton in state q_a and another in state q_b. The state q_a persists till it reads an a, at which point it stops. Hence it checks if all the paths of the subtree it is pursuing has an a. The states q_b and q'_b persist in their states when they see an a and switch states when they see a b. Consequently the Büchi condition is met iff b is seen infinitely often along any path pursued by these states. It is easy to see that this automaton accepts an Υ-tree iff any path from any subtree has at least one a and also has infinitely many b's. Note that the copies of the automaton checking the two properties walk down the tree independent of each other. □

Note that for any non-deterministic automaton $A = (\Sigma, Q, \delta, q_0, \mathcal{F})$ we have the alternating automaton $A' = (\Sigma, Q, \delta', q_0, \mathcal{F})$ where for every $q \in Q$, $a \in \Sigma$, $C \subseteq \Upsilon$,

$$\delta'(q, a, C) = \bigvee_{g \in \delta(q,a,C)} \bigwedge_{c \in C} (g(c), c)$$

It is easy to check the language of trees accepted by A' is the same as that of A. The converse also holds:

Theorem 4.1 ([MSS86]) *For every alternating parity automaton A', one can construct an equivalent non-deterministic parity automaton A. Moreover, the number of states in A is at most exponential in the number of states in A'.*

Hence non-deterministic parity automata and alternating parity automata are equally expressive. Rabin tree automata are also as expressive as parity automata. However, it turns out that Büchi non-deterministic tree automata are strictly weaker than Rabin/parity non-deterministic tree automata.

The class of languages accepted by Rabin/parity automata are called *regular tree languages* and it is well-known that this class is closed under union, intersection, projection and complement [Tho97].

Also, Rabin proved in 1969 [Rab69] that the problem of checking whether a non-deterministic tree automaton accepts some tree is decidable. The complexity of this decision procedure has been improved over the years and currently it is known that non-deterministic Rabin automata can be checked for emptiness in time $(nm)^{O(m)}$ where n is the number of states in the automaton and m is the number of Rabin pairs in the winning condition. This problem has also been shown to be NP-complete [EJ88, PR89a]. For parity automata, the complexity of checking emptiness is in NP, and in $\mathsf{co\text{-}NP}$, but is not known to be in P [Eme97]. On the other hand, Büchi tree automata can be checked for emptiness in polynomial time (in fact in time $O(n^2)$ where n is the number of states in the automaton) [VW86b].

Moreover, all the above algorithms give a regular witness tree that the automaton accepts, if the language it accepts is nonempty. A tree is *regular* if its set of "complete" subtrees (i.e. the full subtrees rooted at nodes), up to isomorphism, is finite. Such a tree can in fact be presented as a finite-state transition system whose unfolding gives the tree (for a natural notion of unfolding similar to the one defined in Chapter 2). The number of states in this transition system is also bounded by the time-estimate above — i.e. $(nm)^{O(m)}$ for Rabin tree automata, etc. In many of our applications of tree-automata, these witnesses of regular trees will be useful in actually designing controllers.

Chapter 5

Synthesis and control for branching-time logics

They sought it with thimbles, they sought it with care;
They pursued it with forks and hope;
They threatened its life with a railway-share;
They charmed it with smiles and soap.

— *The Hunting of the Snark*, Lewis Caroll

In this chapter we study the problem of control-synthesis for the branching-time temporal logics CTL and CTL*. Apart from the control-synthesis problem, we also study the synthesis problem where we are given a specification in CTL*, say, and are asked to come up with a program that satisfies the specification.

Let us look at the program synthesis problem more closely. Suppose we are given finite sets I and O of input and output signals. A *program* can be viewed as a *strategy* $f : (2^I)^* \to 2^O$ that maps finite sequences of input signal sets into an output signal set. When f interacts with an environment that generates infinite input sequences, what results is an infinite computation over $2^{I \cup O}$. Though f is deterministic, it produces a computation tree. The branches of the tree correspond to external non-determinism caused by the different possible inputs. Many classes of reactive programs like protocols and finite-state hardware devices can be seen to work in such a manner.

One can now specify properties of such an open system by a linear or branching temporal logic formula (over $I \cup O$). Unlike linear temporal logics, in branching temporal logics one can specify possibility requirements such as "every input sequence can be extended so that the output signal v eventually becomes true" ([DTV99]). This is achieved via existential and universal quantification provided in branching temporal logics [Lam80, Eme90].

The *realizability problem* for a branching-temporal logic is to determine, given a branching-time specification φ, whether there exists a program $f : (2^I)^* \rightarrow 2^O$ whose computation tree satisfies φ [ALW89, PR89a]. Realizing φ boils down to synthesizing such a function f. An important aspect of the computation tree associated with f is that it has a fixed branching degree $|2^I|$. It reflects the assumption that at each stage, all possible input signals are provided by the environment. Such environments are referred to as *maximal* or *universal environments*. Intuitively, these are static environments in terms of the branching possibilities they contribute to the associated computation trees. Equivalently, as we have noted already, each program has just one computation tree capturing its behavior in the presence of a maximal environment. In a more general setting, however, we have to consider environments that are, in turn, open systems. We term such environments *reactive*. They might offer different subsets of 2^I as input possibilities at different stages in the computation.

As an illustration, consider $I = \{r_1, r_2, ..., r_n\}$ and $O = \{t_1, t_2, ..., t_n\}$ where I represents n different types of resources and O represents n different types of tasks with the understanding that, at each stage, the system needs to receive r_i from the environment in order to execute t_i. In the case of the maximal environment, the specification "it is always possible to reach a stage where t_i is executed" ($\mathsf{AGEF}(t_i)$ in CTL parlance) is realizable. This is so because at each stage in the computation, the maximal environment presents all possible combinations of the resources. In the case of the reactive environment, the above specification is *not* realizable. This is so because there could be an environment driven by an open system that produces only a finite number of the resource r_i. In the resulting computation tree, each path eventually reaches a node in which the environment stops offering r_i. From then on,

t_i cannot be executed.

So, a reactive environment associates a *set* of computation trees with a program — each tree describes the behaviour of the program when it interacts with some environment. Consequently, in the presence of reactive environments, the realizability problem must seek a program *all* of whose computation trees satisfy the specification.

The control-synthesis problem is closely related to the realizability problem. Here we are given a plant (which is an open system) that suitably models the system and environment interacting with each other. Given a branching-time specification φ, the *control problem* is to come up with a strategy for controlling the moves made by the system so that the resulting computation tree satisfies φ. Here again, assuming a reactive environment requires the controller to function correctly no matter how the environment disables some of its moves; thus correctness should be checked with respect to a whole set of computation trees.

In this chapter, we study the control problems for both CTL^* and CTL specifications against non-reactive and reactive environments. It turns out that the realizability problem can be reduced to the control problem.

The controller-synthesis problem for maximal environments can be transformed (by flipping the role of the system and the environment) into the *module-checking* problems solved in [KV96, KV97a]. Hence, from the results on module checking, it follows that the problem is $\mathsf{EXPTIME}$ and 2-$\mathsf{EXPTIME}$ complete for CTL and $\mathsf{CTL}^\star$ respectively [KV99a, KV99b].

The main result of this chapter is that for reactive environments these problems (realizability and control) are 2-$\mathsf{EXPTIME}$ complete and 3-$\mathsf{EXPTIME}$ complete for CTL and $\mathsf{CTL}^\star$, respectively. In this sense, reactive environments make it more difficult to realize open systems and synthesize controllers for them.

The upper bounds are established using automata-theoretic methods. In case the answer to a realizability/control problem is positive, we also show how to extract a program/controller that meets the specification. The sizes of these and the time to construct them are shown to be of the same order as the time-complexity in solving the problem.

5.1 The problem setting

First, recall the definitions of the branching-time logics CTL and CTL*, Kripke structures as well as the notions of nondeterministic automata on infinite trees introduced in Chapter 4.

In order to study controller synthesis, we model a plant as $P = \langle AP, W_s, W_e, R, w_0, L\rangle$, where AP, R, w_0, and L are as in a Kripke structure with $W = W_s \cup W_e$. Here W_s is a set of *system states* and W_e is a set of *environment states*. Throughout what follows we are concerned only with *finite* plants; AP and W are both finite sets. We also assume that $W_s \cap W_e = \emptyset$. The *size* of the plant is $|P| = |W| + |R|$.

Note that unlike the models in Chapter 2 and Chapter 3, our models no longer have event-labels on transitions and the additional labelling of events to capture the way the system and the environment interact. What we have instead is a partition of the state-space into the set of system states (where it is the turn of the system to move) and environment states (where the environment makes a move). The underlying structure need not be bipartite with respect to these sets, and hence, the system and environment need not strictly alternate. In addition, we have a labelling on the states of the plant that describes the set of atomic propositions that are true in each state.

Recall the notion of Υ-trees, for a finite set Υ. Let $P = \langle AP, W_s, W_e, R, w_o, L\rangle$ be a plant. Then P can be unwound into a W-tree T_P in the obvious manner: it is the smallest set such that the following conditions hold:

- $\varepsilon \in T_P$; $dir(\varepsilon) = w_0$
- If $x \in T_P$ and $w \in W$, then ($x.w \in T_P$ iff $R(dir(x), w)$)

This tree is similar to the unfoldings of transition systems studied before. The tree T_P induces the 2^{AP}-labelled tree (T_P, V_P) where for each $v \in T_P$, we have $V_P(v) = L(dir(v))$.

A controller is now an advice function which, for every system state in this tree, picks a subset of children of the state to indicate which moves it wants to allow at that point. Hence it is just a restriction of the tree at the system nodes.

Let $P = \langle AP, W_s, W_e, R, w_o, L\rangle$ be a plant, (T_P, V_P) be its unwinding and let (T, V) be a subtree of (T_P, V_P). We denote by $T^s = \{v \mid v \in T \text{ and } dir(v) \in W_s\}$

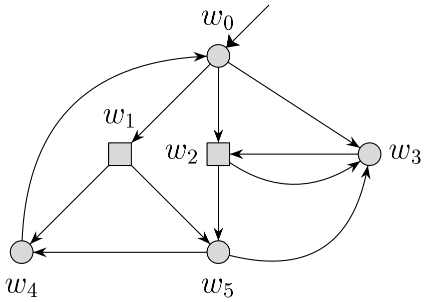
the set of nodes of T that correspond to system states, and by $T^e = T \setminus T^s$ the set of states that correspond to environment states.

A *restriction* (T', V') of (T, V) is a tree $T' \subseteq T$ such that V' is V restricted to T'. A restriction (T', V') of (T, V) is said to be a *system restriction* if for every environment state $v \in T'^e$, all children of v in T belong to T' as well. I.e., there is no pruning of the tree at environment states. Similarly, a restriction (T', V') of (T, V) is said to be an *environment restriction* if for every system state $v \in T'^s$, all children of v in T belong to T'.

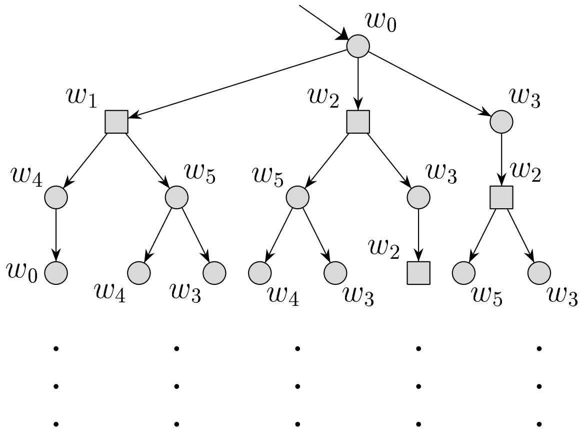
A strategy (or controller) for a plant P is now just a system restriction (T, V) of (T_P, V_P). Note we can have v and v' with $dir(v) = dir(v')$ and still have $succ_T(v) \neq succ_T(v')$. Indeed, the decisions made by the system and the environment depend not only on the current state of the interaction (that is, $dir(v)$), but also on the entire interaction between the system and the environment so far (that is, v). We say that the strategy (T, V) has finite-memory (or that the controller is finite-state) if the number of non-isomorphic subtrees of it is finite. In this case, one can find a finite Kripke structure P' whose unwinding is isomorphic to (T, V). This structure then corresponds to the controlled plant.

For example, Figure 5.1 shows a plant P and the unwinding (T_P, V_P) of it. It also illustrates a system restriction of the plant. The system nodes are denoted by circles and the environment nodes by squares. This tree (T, V) represents the system playing according to its strategy against an environment which plays all moves it can at all its states (which we call an universal environment). The control-synthesis problem for non-reactive environments is to find whether, given a plant P and a specification φ, there is a strategy for P such that the tree corresponding to the strategy playing against the universal environment satisfies the specification.

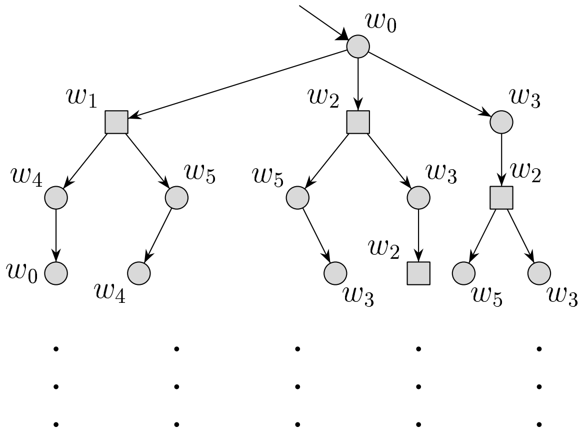
However, in some cases, we may not only want the strategy to win against the universal environment, but also against *all possible environments*. Consider, for example, the scenario where we have a set of plants $\{P_1, \ldots, P_k\}$ that interacts with a (universal) environment E. In this context, in order to make the individual plants satisfy a specification, we may want to solve the problem by synthesizing independently a controller for each plant. Hence, we would want to design a controller C_i, say, for a plant P_i, by considering its environment as all interactions with E as well as the other plants. We would like the controller to make sure that the plant meets its specification no matter how the other processes behave. These processes



A plant P



The unwinding (T_P, V_P) of P



A system restriction of (T_P, V_P)
Corresponds to a universal
g-tree for some
controller g

Figure 5.1: Unwindings and system-restrictions

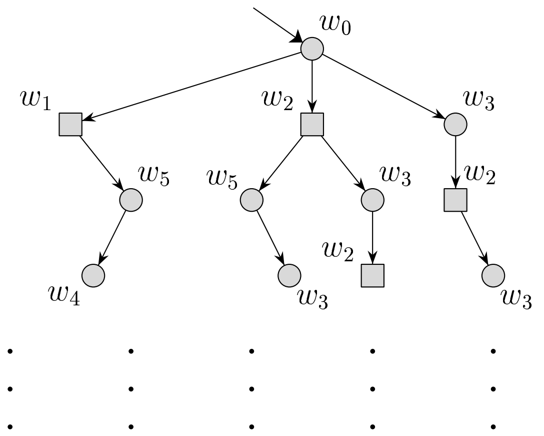


Figure 5.2: g-respecting executions

form part of the environment but being independently designed, may not play like the universal environment. Hence we would like P_i to meet its specification for all possible environments.

The interaction of a reactive environment with a plant can be seen as just a way of pruning the unwinding of a plant at *environment states* — the environment at any environment node of the tree picks a (nonempty) subset of its children which are the possibilities it will offer. Hence the control-synthesis problem for reactive environments is to come up with a strategy such that any pruning of the unwinding of the controlled plant at the environment states satisfies the given specification.

We now make this intuition formal. A *controller* for the system is a function g that assigns to each $v \in T_P^s$, a non-empty subset of $succ_{T_P}(v)$. The universal g-tree is the system restriction of (T_p, V_p) corresponding to g: i.e. it is the tree (T_g, V_g) where T_g is the smallest subset of T_P such that:

- $\varepsilon \in T_g$
- If $v \in T_g$ and $v' \in g(v)$, then $v' \in T_g$

A *g-respecting execution* of P is any environment restriction (T, V) of (T_g, V_g). For example, Figure 5.2 exhibits a g-respecting execution of the controller g depicted in Figure 5.1.

Definition 5.1 [**Control problem for non-reactive environments**]
Given a finite plant P and a specification φ (in CTL or $\mathsf{CTL}^\star$), does there exists a

controller g such that (T_g, V_g) satisfies φ? □

Definition 5.2 [**Control problem for reactive environments**]
Given a finite plant P and a specification φ (in CTL or CTL*), does there exist a controller g such that all g-respecting executions of P satisfy φ?

Controllers for the above problems will be called controllers for (P, φ) against non-reactive and reactive environments, respectively.

The *synthesis problem* (or the *realizability problem*) for programs against reactive and non-reactive environments can be defined along very similar lines: we consider a program f interacting with its environment via two finite sets I and O of input and output signals respectively . We can view f as a strategy $f : (2^I)^* \rightarrow 2^O$ that maps finite sequences of input signal sets into an output signal set. The interaction starts by the program generating the output $f(\varepsilon)$. The environment replies with some $i_1 \subseteq I$. In general, $f(i_1.i_2.\ldots.i_j)$, is the response of f for the input sequence $i_1.i_2.\ldots.i_j$. This (infinite) interaction can be represented by a computation tree. The branches of the tree correspond to external non-determinism caused by different input signal sets chosen by the environment. Thus f can be viewed as the full $2^{I \cup O}$-labeled 2^I-tree (T_f, V_f) with $T_f = (2^I)^*$, $dir(\varepsilon) = \emptyset$ and $V_f(v) = dir(v) \cup f(v)$ for each $v \in T_f$.

Given a CTL or CTL* formula φ, the *realizability problem for non-reactive environments* is to find a a strategy f so that (T_f, V_f) satisfies φ. We say that f *realizes* φ if f is such a strategy. And say that φ is realizable if there is a strategy that realizes it.

On the other hand, the *realizability problem for reactive environments* is to find a strategy f such that no matter how the environment disables (in a non-blocking manner) its possible responses at different stages, the tree of interaction between the system and the environment satisfies φ. Formally, let $f : (2^I)^* \rightarrow 2^O$ be a strategy and let (T, V) be a $2^{I \cup O}$-labeled 2^I-tree with $dir(\varepsilon) = \emptyset$. We say that (T, V) is an *f-respecting execution* iff $V(v) = f(v) \cup dir(v)$ for each $v \in T$. (In other words, (T, V) is a restriction of the computation tree (T_f, V_f)). The realizability problem for reactive environments is to find if there is an f such that every f-respecting execution satisfies φ. We say that f *reactively realizes* φ if f is such a strategy. Also, φ is said to be *reactively realizable* if there is a program f that reactively realizes φ.

It turns out that the control-synthesis problem is harder than that of realizability. Using a universal plant that embodies all the possible assignments to I and O, the realizability problem can be reduced to the control problem. Formally, we have the following:

Lemma 5.1 *Let φ be a* $\mathsf{CTL}^\star$ (CTL) *formula over $AP = I \cup O$. We can effectively construct a finite plant P and a* $\mathsf{CTL}^\star$ *(resp.* CTL*) formula φ' such that $|P| = O((2^{|AP|})^2)$, $|\varphi'| = O(|\varphi| + 2^{|AP|})$, and the following hold:*

- *φ is realizable iff there is controller for (P, φ') against the universal environment*
- *φ is reactively realizable iff there is a controller for (P, φ') against reactive environments.*

Proof We define $P = (AP', W_s, W_e, R, w_0, L)$ as follows:

- $AP' = AP \cup \{p_e\}$ with $p_e \notin AP$. (The role of p_e will become clear soon).
- $W_e = 2^I \times 2^O$
- $W_s = \{w_0\} \cup 2^I$ with $w_0 \notin 2^I \cup W_e$
- $R = R_0 \cup R_1 \cup R_2$ where $R_0 = \{w_0\} \times (\{\emptyset\} \times 2^O)$, $R_1 = W_e \times 2^I$, and $R_2 = \{(X, (X, Y)) \mid X \subseteq I \text{ and } Y \subseteq O\}$.
- $L((X, Y)) = X \cup Y \cup \{p_e\}$ for each $(X, Y) \in W_e$ and $L(w) = \emptyset$ for each $w \in W_s$.

Next, for the formula φ in CTL (or $\mathsf{CTL}^\star$), we construct the CTL (respectively $\mathsf{CTL}^\star$) formula φ' over AP' by setting $\varphi' = \varphi'_1 \wedge \varphi'_2$. The basic idea is that φ'_1 ensures that the truthhood of φ matters only at the states in W_e. The conjunct φ'_1 is the formula $\mathsf{EX}(\|\varphi\|)$ where $\|\varphi\|$ is defined inductively as follows:
For formulas φ in $\mathsf{CTL}^\star$, $\|\varphi\|$ is defined as:

- For state formulas:
  - $\|p\| = p$ for $p \in AP$
  - $\|\neg\varphi\| = \neg\|\varphi\|$
  - $\|\varphi_1 \vee \varphi_2\| = \|\varphi_1\| \vee \|\varphi_2\|$

  - $||\mathsf{E}\varphi|| = \mathsf{E}||\varphi||$
  - $||\mathsf{A}\varphi|| = \mathsf{A}||\varphi||$

- For path formulas
  - If φ is a state-formula, then $||\varphi||$ is already defined.
  - $||\neg\varphi|| = \neg||\varphi||$
  - $||\varphi_1 \vee \varphi_2|| = ||\varphi_1|| \vee ||\varphi_2||$
  - $||\mathsf{X}\varphi|| = \mathsf{X}\ \mathsf{X}\ (||\varphi||)$
  - $||\varphi\ \mathsf{U}\ \varphi'|| = (p_e \Rightarrow ||\varphi||)\ \mathsf{U}\ (p_e \wedge ||\varphi'||))$

For formulas in CTL, it is defined as:

- $||p|| = p$ for $p \in AP$
- $||\neg\varphi|| = \neg||\varphi||$
  $||\varphi_1 \vee \varphi_2|| = ||\varphi_1|| \vee ||\varphi_2||$.
- $||\mathsf{EX}\varphi|| = \mathsf{EX}\ \mathsf{EX}||\varphi||$ and
  $||\mathsf{E}(\varphi\ \mathsf{U}\ \varphi')|| = \mathsf{E}(\ (p_e \Rightarrow ||\varphi||)\ \mathsf{U}\ (p_e \wedge ||\varphi'||)\ )$
- $||\mathsf{AX}\varphi|| = \mathsf{AX}\ \mathsf{AX}||\varphi||$ and
  $||\mathsf{A}(\varphi\ \mathsf{U}\ \varphi')|| = \mathsf{A}(\ (p_e \Rightarrow ||\varphi||)\ \mathsf{U}\ (p_e \wedge ||\varphi'||)\ )$

The conjunct φ_2' ensures that the system chooses only one move at states in W_s (since the 2^O labelling required must be unique). It is given by $\varphi_2' = \mathsf{AG}(\neg p_e \Rightarrow (\bigwedge_{z\in O}(\mathsf{EX}z \Rightarrow \mathsf{AX}z))$ It is easy to check that P and φ' satisfy the required properties. $\square$

5.2 Synthesis and control against the universal environment

It turns out that the control-synthesis problem for universal environments reduces to another problem, the *module checking problem*, that has been already solved [KV96, KV97a].

The *module-checking* problem can be stated in the following manner. Consider a plant modelled as a Kripke structure as above and consider a formula φ in CTL or CTL*. The idea is now to *verify* whether the plant (already) satisfies φ. Again, we can consider either the plant interacting with the universal environment or the plant interacting with a reactive environment.

The model-checking problem is to check whether the universal tree generated by the plant, i.e. the tree (T_P, V_P), satisfies the specification φ. The module-checking problem is to check whether all the trees that are obtained by interactions with all possible environments satisfy φ. In other words, the problem is to check whether all environment restrictions of (T_P, V_P) satisfy φ. If it does, then we say that P module-checks against φ.

Now consider the problem of control-synthesis against the universal environment. The problem is to find whether there is a system restriction of the tree (T_P, V_P) that satisfies φ. But this problem is the same as the module-checking problem, if we interchange the system and environment states, and negate the formula:

Proposition 5.2 *Let $P = \langle AP, W_s, W_e, R, w_o, L\rangle$ be a plant and φ be a formula in* CTL*. *Let $P' = \langle AP, W'_s, W'_e, R, w_0, L\rangle$ be a new plant where $W'_s = W_e$ and $W'_e = W_s$. Then there is a controller for (P, φ) against the universal environment iff P' does* not *module-check against $\neg\varphi$.*

Proof A system restriction of (T_P, V_P) is an environment restriction of $(T_{P'}, V_{P'})$. Hence, there is a system restriction of (T_P, V_P) that satisfies φ iff there is an environment restriction of $(T_{P'}, V_{P'})$ that satisfies φ, i.e. iff it is not the case that all environment restrictions of $(T_{P'}, V_{P'})$ satisfy $\neg\varphi$. $\square$

Kupferman and Vardi have shown in [KV96] that the complexity of the module-checking problems for CTL and CTL* are EXPTIME complete and 2-EXPTIME complete, respectively. Due to this we have:

Theorem 5.1 ([KV99a])

1. *The problems of realizability and control-synthesis for* CTL *against the universal environment are* EXPTIME *complete.*
2. *The problems of realizability and control-synthesis for* CTL* *against the universal environment are* 2-EXPTIME *complete.*

Proof The upper bounds as well as the lower bounds for control-synthesis follow from the proposition above and Lemma 5.1. The lower bound for realizability follows from [KV99a]. □

The procedures used in [KV96] also employ automata-theoretic techniques and it is easy to see that their technique can, when module-checking fails, produce a regular tree (represented as a finite transition system) which is an environment restriction that doesn't satisfy the specification. When one starts with a control-synthesis problem with a plant P and specification φ and reduces it to module-checking as above, this tree is a system-restriction of (T_P, V_P) that satisfies φ. Hence it gives a finite-state controller for P that meets the specification. The sizes of the controllers can also be seen to be bounded by the time-bounds of the module-checking procedure (exponential for CTL formulas and double exponential for $\mathsf{CTL}^\star$ formulas).

5.3 Reactive environments: Upper bounds

In view of the notion of module-checking mentioned above, the control-synthesis problem for reactive environments is the problem of checking if the plant can be controlled so that it *module-checks* against the specification. Before we show that the control-synthesis problems for reactive environments are decidable, let us first recall the following well-known connections relating branching temporal logics and tree automata:

Theorem 5.2

(1) [VW86a] *Given a* CTL *formula* φ *over* AP *and a set* Υ*, we can construct a nondeterministic Büchi tree automaton* $A_{\Upsilon,\varphi}$ *with* $2^{O(|\varphi|)}$ *states that accepts exactly the set of* 2^{AP}*-labeled* Υ*-trees that satisfy* φ*.*

(2) [EJ88, Saf88, Tho97] *Given a* $\mathsf{CTL}^\star$ *formula* φ *over* AP *and a set* Υ*, one can construct a nondeterministic parity tree automaton* $A_{\Upsilon,\varphi}$ *with* $2^{2^{O(|\varphi|)}}$ *states and* $2^{O(|\varphi|)}$ *colours that accepts exactly the set of* 2^{AP}*-labeled* Υ*-trees that satisfy* φ*.*

The first part of the above theorem follows from [VW86a]. For the second part, in [EJ88] it was shown how to construct, given a $\mathsf{CTL}^\star$ formula φ, a nondeterministic Rabin tree automaton $A_{\Upsilon,\varphi}$ with $2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ Rabin pairs that accepts exactly the set of 2^{AP}-labeled Υ-trees that satisfy φ. In this proof, a crucial step is to use the fact that one can convert any nondeterministic automata on infinite *words* to a deterministic automaton on infinite words with the Rabin acceptance condition, shown in [Saf88]. In [Tho97] it is shown that one can in fact build a parity automaton on infinite words, instead of a Rabin automaton. Incorporating this, we can use parity automata instead of Rabin automata in the proof of [EJ88] and build, for a given formula $\varphi \in \mathsf{CTL}^\star$, a nondeterministic parity tree automaton $A_{\Upsilon,\varphi}$ with $2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ colours that accepts exactly the set of 2^{AP}-labeled Υ-trees that satisfy φ.

The decision procedure

Recall that in the control problem for reactive environments we are given a plant $P = \langle AP, W_s, W_e, R, w_0, L\rangle$ and a CTL (or $\mathsf{CTL}^\star$) formula φ over AP, and we have to decide whether there is a strategy g for the system so that all the g-respecting executions of P satisfy φ.

Recall that a strategy g for the system assigns to each $v \in T_P^s$ a nonempty subset of $succ(v)$. We can associate with g a $\{\perp, \top, d\}$-labeled W-tree (T_P, V_g), where for every $v \in T_P$, the following hold:

- If $v \in T_P^s$, then the children of v that are members of $g(v)$ are labeled by $\top$, and the children of v that are not members of $g(v)$ are labeled by $\perp$.
- If $v \in T_P^e$, then all the children of v are labeled by d.

Intuitively, nodes $v.c$ are labeled by $\top$ if g enables the transitions from $dir(v)$ to c (given that the execution so far has traversed v), they are labeled by $\perp$ if g disables the transition from $dir(v)$ to c, and they are labeled by d if $dir(v)$ is an environment state, where the system has no control about the transition from $dir(v)$ to c being enabled. We call the tree (T_P, V_g) the *strategy tree* of g.

Note that not every $\{\perp, \top, d\}$-labeled W-tree (T_P, V) is a strategy tree. Indeed, in order to be a strategy tree, V should label all the successors of nodes corresponding to environment states by d, and it should label all the successors of nodes corresponding to system states by either $\top$ or $\perp$, with at least one successor being labeled by $\top$. Let us fix the convention that the root is always labelled by $\top$. Formally, we have the following.

Theorem 5.3 *Given a plant P with state-space W, there is a nondeterministic Büchi automaton A_{stra} over $\{\perp, \top, d\}$-labeled W-trees, such that A_{stra} has $|W|$ states and accepts exactly the strategy trees of P.*

Proof Let $P = \langle AP, W_s, W_e, R, w_0, L\rangle$ and $W = W_s \cup W_e$. For $w \in W_s$, let $\mathcal{G}_w$ denote the set of all functions $g : succ(w) \to \{\top, \perp\}$ such that there is at least one $w' \in succ(w)$ which is mapped by g to $\top$. $\mathcal{G}_w$ denotes the possible moves the system can make at the state w.

Consider the nondeterministic Büchi tree automaton on Σ-labelled W-trees (where $\Sigma = \{\top, \perp, d\}$): $A_{stra} = (\Sigma, Q, \delta, Q_0, \mathcal{F})$ where $Q = W \times \Sigma$, $Q_0 = \{(w_0, \top)\}$ and δ is defined as:

- For every $(w, m) \in Q$, $m \in \Sigma$ and $X \subseteq W$ where $X \neq succ(w)$, $\delta(q, m, X) = \emptyset$. i.e. A_{stra} accepts only trees that are labellings of the unwinding of the plant — it rejects all other W-trees.

- For every $w \in W$, $m, m' \in \Sigma$ where $m \neq m'$, $\delta((w, m), m', succ(w)) = \emptyset$, i.e. at a state (w, m), A is expecting to see a node labelled m — if it does not see m, it rejects the tree.

- For every $w \in W_s$, $m \in \{\top, \perp, d\}$, $\delta((w, m), m, succ(w)) = \{\ g' \mid \exists g \in \mathcal{G}_w,$ such that for every $w' \in succ(w)$, $g'(w') = (w', g(w'))\ \}$

- For every $w \in W_e$, $m \in \{\top, \perp, d\}$, $\delta((w, m), m, succ(w)) = \{g_d\}$ where $g_d : succ(w) \to Q$ is the function that maps every $w' \in succ(w)$ to (w', d).

The acceptance condition $\mathcal{F}$ is trivial — $\mathcal{F} = Q$. Hence a tree is accepted if there is a run of the automaton on it. It is easy to check that the automaton accepts exactly the set of all strategy trees. □

Our algorithm proceeds as follows. Given a formula φ, we construct a tree automaton A such that A accepts a strategy tree (T_P, V_g) iff there is a g-respecting execution of P that *does not* satisfy φ. Then, there is a controller for (P, φ) against reactive environments iff the automaton A is not universal with respect to strategy trees (that is, the language of A_{stra} is not contained in that of A). Indeed, a strategy tree that is not accepted by A is induced by a strategy g all of whose g-respecting executions satisfy φ.

Theorem 5.4 *Given a plant P with state space W and a branching-time formula φ, we can construct a nondeterministic tree automaton A over $\{\bot, \top, d\}$-labeled W-trees such that the following hold:*

1. *A accepts a strategy tree (T_P, V_g) iff there is a g-respecting execution of P that does not satisfy φ.*
2. *If φ is a* CTL *formula, then A is a Büchi automaton with $|W| \cdot 2^{O(\varphi)}$ states.*
3. *If φ is a* CTL* *formula, then A is a parity automaton with $|W| \cdot 2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ colours.*

Proof Let $P = \langle AP, W_s, W_e, R, w_0, L\rangle$, and let $A_{W,\neg\varphi} = \langle 2^{AP}, Q, \delta, Q_0, \mathcal{F}\rangle$ be the automaton that accepts exactly all 2^{AP}-labeled W-trees that satisfy $\neg\varphi$, as described in Theorem 5.2. Let $W = W_s \cup W_e$. We define $A = \langle \Sigma, Q', \delta', Q'_0, \mathcal{F}'\rangle$ as follows.

- $\Sigma = \{\bot, \top, d\}$
- $Q' = (W \times Q \times \{\top, \bot\}) \cup \{q_{acc}\}$. The state q_{acc} is an accepting sink. Consider a state $(w, q, m) \in W \times Q \times \{\top, \bot\}$. The last component m is the *mode* of the state. When $m = \top$, it means that the transition to the current node is enabled (by either the system or the environment). When $m = \bot$, it means that the transition to the current node is disabled.

  When A is at a state $(w, q, \top)$ as it reads a node v, it means that $dir(v) = w$, and that v has to participate in the g-respecting execution. Hence, A can read $\top$ or d, but not $\bot$. If v is indeed labeled by $\top$ or d, the automaton A guesses a nonempty subset of successors of w. It then moves to states corresponding to the successors of w and q, with an appropriate update of the mode ($\top$ for the successors in the guessed subset and $\bot$ for the rest).

  When A is in a state $(w, q, \bot)$ and it reads a node v, it means that $dir(v) = w$ and that v does not take part in a g-respecting execution. Then, A expects to read $\bot$ or d, in which case it goes to the accepting sink.
- $Q'_0 = \{w_0\} \times Q_0 \times \{\top\}$.
- The transition function δ' is defined as follows:
  For all $w \in W$, $q \in Q$, and $X = succ_P(w)$, we have:.

  - $\delta((w,q,\top),\bot,X) = \delta((w,q,\bot),\top,X) = \emptyset$
  - If $x \in \{\bot, d\}$, then $\delta'((w,q,\bot),x,X) = \{g_{acc}\}$ where g_{acc} is the function that maps each element w' of X to q_{acc}.
  - If $x \in \{\top, d\}$, then $\delta'((w,q,\top),x,X)$ is defined as follows. Let $Y \subseteq X$ be a nonempty subset of $succ(w)$. Then, $\delta'((w,q,\top),x,X)$ contains all the functions g such that there is a function $h \in \delta(q, L(w), Y)$ in $A_{W,\neg\varphi}$ such that:

    * If $w' \in Y$, then g maps w' to $(w', q', \top)$ where $q' = h(w')$
    * If $w' \notin Y$, then g maps w' to $(w', q, \bot)$ (in fact, we can map it to any state $(w', q_1, \bot)$ where $q_1 \in Q$.)

    Intuitively, δ' propagates the requirements imposed by $\delta(q, L(w), Y)$ to the Y-successors of w, for every possible choice of Y.

    Note that δ' is independent of w being a system or an environment state. The type of w is taken into consideration only in the definition of A_{stra}.

  For all $w \in W$, $q \in Q$, $m, m' \in \{\top, \bot\}$ and $X \neq succ_P(w)$, we define $\delta'((w,q,m),m',X) = \emptyset$. This ensures that A works only on unwindings of P. We also define, for every $X' \subseteq W$ and $m \in \{\top,\bot\}$, $\delta'(q_{acc}, m, X') = \{g_{acc}\}$ where g_{acc} is the function that maps each element of X' to q_{acc}.

- The final states are inherited from the formula automaton. Thus, if φ is in CTL, then $\mathcal{F}' = (W \times \mathcal{F} \times \{\top,\bot\}) \cup \{q_{acc}\}$. If φ is in $\mathsf{CTL}^\star$, let $\mathcal{F} : Q \to \{0,\ldots,h\}$. Then, $\mathcal{F}' : Q' \to \{0,\ldots,h\}$ is such that $\mathcal{F}(q_{acc}) = 0$ and for all $w \in W$, $q \in Q$, and $m \in \{\bot,\top\}$, we have $\mathcal{F}'((w,q,m)) = \mathcal{F}(q)$.

□

The above automaton hence accepts a strategy tree iff the strategy corresponding to it is a *losing* strategy for the system. There is hence a winning strategy for the system iff there is a tree accepted by A_{stra} that is not accepted by A. So we are left with the problem of checking whether the language of A_{stra} is contained in the language of A. Since tree automata are closed under complement [Rab69, Tho97], we can complement A, get an automaton $\tilde{A}$, and then check the non-emptiness of the intersection of A_{stra} with $\tilde{A}$. Hence the following theorem.

Theorem 5.5 *Given a plant P and a formula φ in* CTL*, the control problem for φ is in* 2-EXPTIME*. More precisely, it can be solved in time $O(\exp(|P|^2 \cdot 2^{O(|\varphi|)}))$. For φ in* CTL**, the problem is in* 3-EXPTIME*. More precisely, it can be solved in time $O(\exp(|P|^2 \cdot 2^{2^{O(|\varphi|)}}))$.*[1]

Proof For the complexity of this procedure, it is easy to see that if φ is in CTL, the automaton A has a state-space size of $O(|P| \cdot 2^{O(|\varphi|)})$. Though A runs on k-ary trees (where k depends on P), it can be complemented as easily as automata on binary trees — the complemented automaton $\tilde{A}$ (as well as its intersection with A_{stra}) is a parity automaton with $O(\exp(|P| \cdot 2^{O(|\varphi|)}))$ states and $O(|P| \cdot 2^{O(|\varphi|)})$ colours ([Tho97]). Since emptiness of parity tree automata can be done in time polynomial in the state-space and exponential in the number of colours [EJ88, PR89a], we can check emptiness of this intersection in time $O(\exp(|P|^2 \cdot 2^{O(|\varphi|)}))$. For CTL* specifications, the analysis is similar except that the complexity contributed by the formula increases by one exponential. □

By [Rab69], if there is indeed a strategy that is winning for the system, then the automaton that is the product of A_{stra} and the complement of the automaton constructed in Theorem 5.4 accepts it and when we test the automaton for emptiness, we can get a regular tree accepted by the automaton. This then provides a finite-memory winning strategy that can be realized as a finite state controller for the system.

5.4 Reactive environments: Lower bounds

For two 2^{AP}-labeled trees (T, V) and (T', V'), and a set $Q = \{q_1, \dots, q_k\} \subseteq AP$, we say that (T, V) and (T', V') are *Q-different* if they agree on everything except possibly the labels of the propositions in Q. Formally, $T = T'$ and for all $x \in T$, we have $V(x) \setminus Q = V'(x) \setminus Q$. The logic AQCTL* extends CTL* by universal quantification on atomic proposition: if ψ is a CTL* formula and $q_1, \dots, q_k$ are atomic propositions, then $\forall q_1, \dots, q_k \psi$ is an AQCTL* formula. The semantics of $\forall q_1, \dots, q_k \psi$ is given by $S \models \forall q_1, \dots, q_k \psi$ iff for all trees (T, V) such that (T, V) and the unwinding (T_S, V_S) of S are $\{q_1, \dots, q_k\}$-different, $(T, V) \models \psi$. The logics

[1] $exp(x)$ stands for $2^{O(x)}$

$\mathsf{A}\mathsf{Q}\mathsf{LTL}$ and $\mathsf{A}\mathsf{Q}\mathsf{CTL}$ are defined similarly as the extensions of LTL and CTL with universal quantification on atomic propositions.

The following Theorem is taken from [SVW87]. We describe here the full proof, as our lower-bound proofs are based on it.

Theorem 5.6 [SVW87] *The satisfiability problem for* $\mathsf{A}\mathsf{Q}\mathsf{LTL}$ *is* $\mathsf{EXPSPACE}$*-hard.*

Proof We do a reduction from the problem of whether an exponential-space deterministic Turing machine T accepts an input word x. That is, given T and x, we construct an $\mathsf{A}\mathsf{Q}\mathsf{LTL}$ formula $\forall q\varphi$ such that T accepts x iff $\forall q\varphi$ is satisfiable. Below we describe the formula φ informally. The formal description of φ and of the function *next* are detailed later.

Let $T = \langle \Gamma, Q, \rightarrow, q_0, F\rangle$, where Γ is the alphabet, Q is the set of states, $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the transition relation (we use $(q, a) \rightarrow (q', b, \Delta)$ to indicate that when T is in state q and it reads the input a in the current tape cell, it moves to state q', writes b in the current tape cell, and its reading head moves one cell to the left or to the right, according to Δ), q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states. Let $n = a \cdot |x|$, for some constant a, be such that the working tape of T has 2^n cells. We encode a configuration of T by a word $\gamma_1\gamma_2 \dots (q, \gamma_i) \dots \gamma_{2^n}$. That is, all the letters in the configuration are in Γ, except for one letter in $Q \times \Gamma$. The meaning of such a configuration is that the j^{th} cell of T, for $1 \leq j \leq 2^n$, is labeled γ_j, the reading head points on cell i, and T is in state q. For example, the initial configuration of T is $(q_0, x_1)x_2 \cdots x_n\#\# \cdots$, where the input to T is $x = x_0x_1x_2 \dots x_n$ and $\#$ stands for the empty cell. We can now encode a computation of T by a sequence of configurations.

Let $\Sigma = \Gamma \cup (Q \times \Gamma)$. We can encode letters in Σ by a set $AP(T) = \{p_1, \dots, p_m\}$ (with $m = \lceil log|\Sigma|\rceil$) of atomic propositions). We define our formulas over the set $AP = AP(T) \cup \{b, c, d, e, q\}$ of atomic propositions. The task of the last five atoms will be explained shortly. Since T is fixed, so is Σ, and hence so is the size of AP.

Consider an infinite sequence π over 2^{AP}. For an atomic proposition $p \in AP$ and a node u in π, we use $p(u)$ to denote the truth value of p at u. That is, $p(u)$ is 1 if p holds at u and is 0 if p does not hold at u. We divide the sequence π into blocks of length n. Every such block corresponds to a single tape cell of the machine T. Consider a block $u_1, \dots, u_n$ that corresponds to a cell ρ. We use the node u_1 to encode the content of cell ρ. Thus, the bit vector $p_1(u_1), \dots, p_m(u_1)$ encodes the letter (in $\Gamma \cup (Q \times \Gamma)$) that corresponds to cell ρ. We use the atomic proposition b to mark the beginning of the block; that is, b should hold on u_1 and fail on $u_2, \ldots, u_n$ (see **C1** below).

Recall that the letter with which cell ρ is labeled is encoded at the node u_1 of the block $u_1, \ldots, u_n$ that corresponds to ρ. Why then do we need a block of length n to encode a single letter? The reason is that the block also encodes the location of the cell ρ on the tape. Since T is an exponential-space Turing machine, this location is a number between 0 and $2^n - 1$. Encoding the location eliminates the need for exponentially many X operators when we attempt to relate two successive configurations. Encoding is done by the atomic proposition c, called the *counter*. Let $c(u_n), \ldots, c(u_1)$ encode the location of ρ. Note that, for technical convenience, the least significant bit of the counter is in u_1. A sequence of 2^n blocks corresponds to 2^n cells and encodes a configuration of T. The value of the counters along this sequence goes from 0 to $2^n - 1$, and then start again from 0. This is enforced by φ. Since we want the size of φ to be $O(n)$, we need also an atomic proposition d that acts as a "carry" bit (see **C2** and **C3** below).

An atomic proposition e marks the last block of a configuration, that is, e holds in a node u_1 of a block $u_1, \ldots, u_n$ iff c holds on all nodes in the block (see **C4**).

Let $\sigma_1 \ldots \sigma_{2^n}, \sigma'_1 \ldots \sigma'_{2^n}$ be two successive configurations of T. For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 \leq i \leq 2^n$ (taking σ_{2^n+1} to be σ'_1 and σ_0 to be the label of the last letter in the configuration before $\sigma_1 \ldots \sigma_{2^n}$, or some special label when $\sigma_1 \ldots \sigma_{2^n}$ is the initial configuration), we know, by the deterministic transition relation of T, what σ'_i should be. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i. The formal definition of *next* is detailed later.

Consistency with *next* gives us a necessary condition for a word to encode a legal computation. In addition, the computation should start with the initial configuration. Finally, if the computation ends with an accepting configuration (that is, one with (q, γ_i) with $q \in F$), then T accepts x. It is easy to specify in LTL the requirements about the initial and accepting configurations. For a letter $\sigma \in \Sigma$, let $\eta(\sigma)$ be the propositional formula over AP that encodes σ. That is, $\eta(\sigma)$ holds in node u_1 of a block that encodes cell ρ iff the cell ρ is labeled σ. Then, φ contains conjuncts (see **C5** and **C6**) that require the computation to start with the initial configuration and to eventually reach an accepting configuration.

The difficult part in the reduction is in guaranteeing that the sequence of configurations is indeed consistent with *next*. To enforce this, we have to relate σ_{i-1}, σ_i, and σ_{i+1} with σ'_i for any i in any two successive configurations $\sigma_1 \ldots \sigma_{2^n}, \sigma'_1 \ldots \sigma'_{2^n}$. One natural way to do so is by a conjunction of formulas like "whenever we meet a cell with counter $i-1$ and the labeling of the next three cells forms the triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$, then the next time we meet a cell with counter i, this cell is labeled $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$". The problem is that as i can take any value from 1 to 2^n, there are exponentially many such conjuncts. This is where the universal quantification of the AQLTL comes into the picture. It enables us to relate $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with σ'_i, for all i.

To understand how this is done, consider the atomic proposition q, and assume that the following hold. **(1)** q is true at precisely two points, both are points in which a block starts, **(2)** there is exactly one point between them in which e holds (or possibly that in exactly both of them, and not in between, e holds) — thus, the two points are in successive configurations, and **(3)** the value of the counter at the blocks starting at the two points is the same. Then, it should be true that **(4)** if the labels of the three blocks starting one block before the first q are σ_{i-1}, σ_i, and σ_{i+1}, then the block starting at the second q is labeled by $next(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$.

Using b, c, and e, we can easily express **(1)**–**(4)** with formulas of length polynomial in n (see the formal definition of **(1)**–**(4)** below). Recall that as the set Σ is fixed, scanning all the possible labels of a cell can be done with a formula of a fixed length. Also note that for expressing **(3)**, we need to compare the value of the two counters bit by bit (see definition of **(3)** below).

The formula φ contains the conjunct $(\mathbf{C7}) = ((\mathbf{1}) \wedge (\mathbf{2}) \wedge (\mathbf{3})) \Rightarrow (\mathbf{4})$. Since the condition $(\mathbf{4})$ is checked in $\forall q \varphi$ for all the assignments to q that satisfy $(\mathbf{1}) \wedge (\mathbf{2}) \wedge (\mathbf{3})$, it follows that φ is satisfied only in computations consistent with *next*. Hence, $\forall q \varphi$ is satisfiable iff there is an accepting computation of T on x.

The formulas described above are formally defined as follows:

C1. b should hold on u_1 and fail on $u_2, \ldots, u_n$:

$$b \wedge \mathsf{X}(\neg b \wedge \mathsf{X}(\neg b \wedge \cdots \wedge \mathsf{X}\neg b) \cdots) \wedge \mathsf{G}(b \leftrightarrow \mathsf{X}^n b)$$

C2. The counter starts at 0:

$$\neg c \wedge \mathsf{X}(\neg c \wedge \mathsf{X}(\neg c \ldots \wedge \mathsf{X}\neg c) \cdots)$$

C3. The counter is increased properly. Note that as we always want to increase it by 1 we take b as a carry for the least significant bit.

- $\mathsf{G}(((b \vee d) \wedge \neg c) \rightarrow (\mathsf{X}(\neg d) \wedge \mathsf{X}^n c))$.
- $\mathsf{G}((\neg(b \vee d) \wedge \neg c) \rightarrow (\mathsf{X}(\neg d) \wedge \mathsf{X}^n \neg c))$.
- $\mathsf{G}(((b \vee d) \wedge c) \rightarrow (\mathsf{X}d \wedge \mathsf{X}^n \neg c))$.
- $\mathsf{G}((\neg(b \vee d) \wedge c) \rightarrow ((\mathsf{X}\neg d) \wedge \mathsf{X}^n c))$.

C4. e holds in a node u_1 of a block with counter 1^n:

$$\mathsf{G}(e \leftrightarrow (b \wedge c \wedge \mathsf{X}(c \wedge \mathsf{X}(c \wedge \mathsf{X}(c \cdots))).$$

C5. The computation starts with the initial configuration:

$$\eta(q_0, x_1) \wedge$$

$$\mathsf{X}^n(\eta(x_2) \cdots \wedge \mathsf{X}^n(\eta(x_n) \wedge \mathsf{X}^n(\eta(\#) \wedge [(\eta(\#) \rightarrow \mathsf{X}^n \eta(\#))\mathsf{U}(\eta(\#) \wedge e)])) \cdots)).$$

C6. The computation reaches an accepting configuration:

$$\mathsf{F}(b \wedge \bigvee_{q \in F, \gamma \in \Gamma} \eta(q, \gamma)).$$

C7. The formula $((\mathbf{1}) \wedge (\mathbf{2}) \wedge (\mathbf{3})) \Rightarrow (\mathbf{4}))$ where:

(1) q is true at precisely two points, both are points in which a block starts:

$$(\neg q)\mathsf{U}(b \wedge q \wedge \mathsf{X}((\neg q)\mathsf{U}(b \wedge q \wedge \mathsf{X}\mathsf{G}\neg q)))$$

(2) There is exactly one point between them in which e holds or e holds at both points but not in any point in between them:

$$(\neg q)\mathsf{U}(q \wedge ((\neg e)\mathsf{U}(e \wedge \mathsf{X}((\neg e)\mathsf{U}q)))).$$

(3) The value of the counter at the blocks starting at the two points is the same:

$$\begin{gathered}(\neg q)\mathsf{U}q \wedge [(c \rightarrow \mathsf{XF}(q \wedge c)) \wedge ((\neg c) \rightarrow \mathsf{XF}(q \wedge (\neg c)))\wedge \\ \mathsf{X}((c \rightarrow \mathsf{F}(q \wedge \mathsf{X}c)) \wedge ((\neg c) \rightarrow \mathsf{F}(q \wedge \mathsf{X}(\neg c)))\wedge \\ \mathsf{X}((c \rightarrow \mathsf{F}(q \wedge \mathsf{XX}c)) \wedge ((\neg c) \rightarrow \mathsf{F}(q \wedge \mathsf{XX}(\neg c)))\wedge \\ \vdots \\ \mathsf{X}((c \rightarrow \mathsf{F}(q \wedge \mathsf{X}^{n-1}c)) \wedge ((\neg c) \rightarrow \mathsf{F}(q \wedge \mathsf{X}^{n-1}(\neg c)))) \cdots))].\end{gathered}$$

(4) If the labels of the three blocks starting one block before the first q are σ_{i-1}, σ_i, and σ_{i+1}, then the block starting at the second q is labeled by $next(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$:

$$\bigvee_{\sigma_1,\sigma_2,\sigma_3 \in \Sigma} (\neg q) \, \mathsf{U} \, (\eta(\sigma_1) \wedge \mathsf{X}^n (q \wedge \eta(\sigma_2) \wedge \mathsf{X}^n (\eta(\sigma_3) \wedge \mathsf{F}(q \wedge \eta(next(\sigma_1, \sigma_2, \sigma_3)))))).$$

The function $next$ is defined as follows: $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denotes our expectation for σ'_i. It is defined as: [2].

- $next(\langle \gamma_{i-1}, \gamma_i, \gamma_{i+1} \rangle) = \gamma_i$.
- $next(\langle (q, \gamma_{i-1}), \gamma_i, \gamma_{i+1} \rangle) = \begin{cases} \gamma_i & \text{If } (q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, L). \\ (q', \gamma_i) & \text{If } (q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, R). \end{cases}$
- $next(\langle \gamma_{i-1}, (q, \gamma_i), \gamma_{i+1} \rangle) = \gamma'_i$ where $(q, \gamma_i) \rightarrow (q', \gamma'_i, \Delta)$.
- $next(\langle \gamma_{i-1}, \gamma_i, (q, \gamma_{i+1}) \rangle) = \begin{cases} \gamma_i & \text{If } (q, \gamma_{i+1}) \rightarrow (q', \gamma'_{i+1}, R). \\ (q', \gamma_i) & \text{If } (q, \gamma_{i+1}) \rightarrow (q', \gamma'_{i+1}, L). \end{cases}$

$\square$

We now show that AqCTL is also strong enough to describe an exponential-space Turing machine with a formula of polynomial length. Moreover, since CTL has both universal and existential path quantification, AqCTL can describe an alternating exponential-space Turing machine [CKS81], implying a 2-EXPTIME lower bound for its satisfiability problem.

Theorem 5.7 *The satisfiability problem for* AqCTL *is* 2-EXPTIME*-hard.*

Proof We do a reduction from the problem whether an exponential-space alternating Turing machine T accepts an input word x. That is, given T and x, we construct an AqCTL formula $\forall q \psi$ such that T accepts x iff $\forall q \psi$ is satisfiable.

Let $T = \langle \Gamma, Q_u, Q_e, \mapsto, q_0, F \rangle$, where the sets Q_u and Q_e of states are disjoint, and contain the universal and the existential states, respectively. We denote their union (the set of all states) by Q. Our model of alternation prescribes that $\mapsto \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ has a binary branching degree. When a universal or an

[2] We assume that T's head does not "fall off" from the right or the left boundaries of the tape. Thus, the case where $i = 1$ and $(q, \gamma_i) \rightarrow (q', \gamma'_i, L)$ and the dual case where $i = 2^n$ and $(q, \gamma_i) \rightarrow (q', \gamma'_i, R)$ are not possible.

existential state of T branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(q, a) \mapsto \langle (q_l, b_l, \Delta_l), (q_r, b_r, \Delta_r) \rangle$ to indicate that when T is in state $q \in Q_u \cup Q_e$ reading input symbol a, it branches to the left with (q_l, b_l, Δ_l) and to the right with (q_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the direction of movement of the head; those are determined by Δ_l and Δ_r.)

For a configuration c of T, let $succ_l(c)$ and $succ_r(c)$ be the successors of c when applying to it the left and right choices in $\mapsto$, respectively. Given an input x, a computation tree of T on x is a tree in which each node corresponds to a configuration of T. The root of the tree corresponds to the initial configuration. A node that corresponds to a universal configuration c has two successors, corresponding to $succ_l(c)$ and $succ_r(c)$. A node that corresponds to an existential configuration c has a single successor, corresponding to either $succ_l(c)$ or $succ_r(c)$. The tree is an accepting computation tree if all its branches reach an accepting configuration.

An accepting tree (i.e. a tree labelled with configurations) can be encoded as a tree where each node in the accepting tree is blown up into a path where the labels on the path encode the configuration corresponding to the node. The formula ψ will describe such encodings of accepting trees. As in the linear case, we encode a configuration of T by a sequence $\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{2^n}$, and we use a block of length n to describe each letter $\sigma_i \in \Gamma \cup (Q \times \Gamma)$ in the sequence. The construction of ψ is similar to the construction described for φ in the linear case. For an LTL formula ξ, let ξ_A be the CTL formula obtained from ξ by preceding each temporal operator by the path quantifier A. For example, $(\mathsf{G}(p \rightarrow \mathsf{F}q))_{\mathsf{A}} = \mathsf{AG}(p \rightarrow \mathsf{AF}q)$, and $(\mathsf{GF}p)_{\mathsf{A}} = \mathsf{AGAF}p$. As in the linear case, the atomic propositions c and d are used to count, b is used to mark the beginning of blocks, and e is used to mark the last letter in a configuration. Note that the conjuncts ξ in φ that impose the desired behavior of b, c, d, and e are such that ξ_A impose the desired behavior of b, c, d and e in the branching case. Also, the conjuncts ξ used in φ in order to check that the first configuration is the initial one and that the computation is accepting are such that ξ_A do the same for the branching case. Our formula ψ has all these conjuncts.

The difficult part is to check that the $succ_l$ and $succ_r$ relations are maintained. For that, we add two atomic propositions, e_E and e_U, that refine the proposition e and indicate whether the configuration just ended has been existential or universal. Also, e_E and e_U will continue to hold till the end of the block representing the last cell of the configuration. Formally, ψ contains the conjuncts

- $\mathsf{AG}(e \rightarrow (\mathsf{A}(e_U \,\mathsf{U}\, \mathsf{EX}b) \vee \mathsf{A}(e_E \,\mathsf{U}\, \mathsf{EX}b))) \wedge \mathsf{AG}(\neg(e_U \wedge e_E))$,
- $\mathsf{AG}(\bigwedge_{q \in Q_e, \gamma \in \Gamma}(\eta(q,\gamma) \rightarrow \mathsf{A}(\neg e)\mathsf{U}(e \wedge e_E)))$, and
- $\mathsf{AG}(\bigwedge_{q \in Q_u, \gamma \in \Gamma}(\eta(q,\gamma) \rightarrow \mathsf{A}(\neg e)\mathsf{U}(e \wedge e_U)))$.

In addition, we use an atomic proposition l to indicate whether the nodes belong to a left or a right successor. For clarity, we denote $\neg l$ by r. Formally, ψ contains the conjunct

$$\mathsf{AG}(l \rightarrow (\mathsf{A}l\mathsf{U}e)) \wedge \mathsf{AG}(r \rightarrow (\mathsf{A}r\mathsf{U}e)).$$

Since a universal configuration c has both $succ_l(c)$ and $succ_r(c)$ as successors, and an existential c has only one of them, ψ also contains the conjunct

$$\mathsf{AG}(\,(e_E \wedge \mathsf{EX}b) \rightarrow (\mathsf{EX}l \vee \mathsf{EX}r)\,) \wedge \mathsf{AG}(\,(e_U \wedge \mathsf{EX}b) \rightarrow (\mathsf{EX}l \wedge \mathsf{EX}r)\,)$$

We can now use universal quantification over atomic propositions in order to check consistency with $succ_l$ and $succ_r$. Note that $succ_l(c)$ and $succ_r(c)$ are uniquely defined. Thus, we can define functions, $next_l$ and $next_r$, analogous to function $next$ of the linear case. Given a sequence $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ of letters in c, the function $next_l(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denotes the expectation for the i'th letter in $succ_l(c)$. We denote this letter by σ_i^l, and similarly for $next_r$ and σ_i^r.

In the linear case, we considered assignments to q in which q holds at exactly two points in the computation. Here, we look at assignments where q holds at exactly two points in each branch. The first point is a node where a block of σ_i starts, and the second point is a node where a block of σ_i^l or σ_i^r starts (note that an assignment to q may check consistency with $succ$ along different branches)[3]

Consider the atomic proposition q, and assume that the following hold: **(1)** q is labeled as described above (in particular, in each branch of the tree, q is true at precisely two nodes, both are nodes in which a block starts), **(2)** in every branch with two occurrences of q, there is exactly one node between them in which e holds (thus, the two nodes are in successive configurations), and **(3)** the value of the counter at the blocks starting at the two points is the same. Then, it should be true

[3]It is convenient to think of a satisfying tree for ψ as a tree that has branching degree 1 everywhere except for nodes that are labeled by e_U and have a successor labelled b, where the branching degree is 2. Our reduction, however, makes no assumption about such a structure.

that **(4)** if the labels of the three blocks starting one block before the first q are σ_{i-1}, σ_i, and σ_{i+1}, then the blocks starting at the second q are either left branches in which case they are labeled by $next_l(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$, or they are right branches, in which case they are labeled by $next_r(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$. Hence, ψ contains the conjunct $((\mathbf{1}) \wedge (\mathbf{2}) \wedge (\mathbf{3})) \rightarrow (\mathbf{4})$, where **(1)**–**(4)** are described formally below. Since in $\forall q\psi$, the condition $(\mathbf{4})$ is checked for all the assignments to q that satisfy $(\mathbf{1}) \wedge (\mathbf{2}) \wedge (\mathbf{3})$, it follows that ψ is satisfied only in a computation tree consistent with $succ_l$ and $succ_r$. Hence, $\forall q\psi$ is satisfiable iff there is an accepting computation tree of T on x.

The formulas **(1)**–**(4)** are formally defined as:

(1) q is labeled as described above:

$$A(\neg q)U(b \wedge q \wedge AXA((\neg q)U(b \wedge q \wedge AXAG\neg q))).$$

(2) In every branch with two occurrences of q, there is exactly one node between them in which e holds (thus, the two nodes are in successive configurations):

$$A(\neg q)U(q \wedge A((\neg e)U(e \wedge AXA((\neg e)Uq)))).$$

(3) The value of the counter at the blocks starting at the two points is the same:

$$\begin{gathered}
A(\neg q)Uq \wedge [(c \rightarrow AXAF(q \wedge c)) \wedge ((\neg c) \rightarrow AXAF(q \wedge (\neg c)))\wedge \\
AX((c \rightarrow AF(q \wedge AXc)) \wedge ((\neg c) \rightarrow AF(q \wedge AX(\neg c)))\wedge \\
AX((c \rightarrow AF(q \wedge AXAXc)) \wedge ((\neg c) \rightarrow AF(q \wedge AXAX(\neg c)))\wedge \\
\vdots \\
AX((c \rightarrow AF(q \wedge (AX)^{n-1}c)) \wedge ((\neg c) \rightarrow AF(q \wedge (AX)^{n-1}(\neg c)))) \cdots))].
\end{gathered}$$

(4) If the labels of the three blocks starting one block before the first q are σ_{i-1}, σ_i, and σ_{i+1}, then the blocks starting at the second q are either left branches in which case they are labeled by $next_l(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$, or they are right branches, in which case they are labeled by $next_r(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$:

$$\begin{aligned}
&A(\neg q)U(\textstyle\bigvee_{\sigma_1,\sigma_2,\sigma_3 \in \Sigma}(\eta(\sigma_1) \wedge (AX)^n(q \wedge \eta(\sigma_2) \wedge (AX)^n(\eta(\sigma_3)\wedge \\
&AF[(q \wedge l \wedge \eta(next_l(\sigma_1, \sigma_2, \sigma_3))) \vee (q \wedge r \wedge \eta(next_r(\sigma_1, \sigma_2, \sigma_3)))])))).
\end{aligned}$$

$\square$

The satisfiability problem for $\mathsf{CTL}^\star$ is exponentially harder than the one for CTL. We now show that this computational difference is preserved when we look at the extensions of these logics with universal quantification over atomic propositions. A full exposition of the details for the lower bound below will detract us from the main theme of the thesis. Hence we omit details and present only a gist of the proof.

Theorem 5.8 *The satisfiability problem for* $\mathsf{AQCTL}^\star$ *is* 3-EXPTIME-*hard.*

Proof We do a reduction from the problem whether a doubly-exponential-space alternating Turing machine T accepts an input word x. That is, given T and x, we construct an $\mathsf{AQCTL}^\star$ formula $\forall q\psi$ such that T accepts x iff ψ is satisfiable.

In [VS85], the satisfiability problem of $\mathsf{CTL}^\star$ is proved to be 2-EXPTIME-hard by a reduction from an exponential-space alternating Turing machine. Below we explain how universal quantification can be used to "stretch" the length of the tape that a polynomial $\mathsf{CTL}^\star$ formula can describe by another exponential. As in the proof of Theorem 5.6, the formula in [VS85] maintains an n-bit counter, and each cell of T's tape corresponds to a block of length n.

In order to point on the letters σ_i and σ_i' simultaneously (that is, the letters that the atomic proposition q point on in the proof of Theorem 5.6), [VS85] adds to each node of the tree a branch such that nodes that belong to the original tree are labeled by some atomic proposition p, and nodes that belong to the added branches are not labeled by p. Every path in the tree has a single location where the atom p stops being true. [VS85] uses this location in order to point on σ' and in order to compare the values of the n-bit counter in the current point (where σ is located) and in the point in the computation where p stops being true.

On top of the method in [VS85], we use the universal quantification in order to maintain a 2^n-bit counter and thus count to 2^{2^n}. Typically, each bit of our 2^n-bit counter is kept in a block of length n, which maintains the index of the bit (a number between 0 to $2^n - 1$). For example, when $n = 3$, the counter looks as follows.

000	001	010	011	100	101	110	111	$\leftarrow$ n-bit counter
0	0	0	0	0	0	0	0	$\leftarrow$ 2^n-bit counter
000	001	010	011	100	101	110	111	
0	0	0	0	0	0	0	1	
000	001	010	011	100	101	110	111	
0	0	0	0	0	0	1	0	
000	001	010	011	100	101	110	111	
0	0	0	0	0	0	1	1	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

To check that the 2^n-bit counter proceeds properly, we use a universally quantified proposition q and we check that if q holds at exactly two points (say, last points in a block of the n-bit counter), with the same value to the n-bit counter, and with only one block between them in which the n-bit counter has value 1^n, then the bit of the 2^n-bit counter that is maintained at the block of the second q is updated properly (we also need to relate and update carry bits, but the idea is the same). $\square$

Note that the number of atomic propositions in ψ in the proofs of both Theorems 5.7 and 5.8 is fixed. Note also that if ψ is satisfiable, then it is also satisfied in a tree of a fixed branching degree (a careful analysis can show that for CTL the sufficient branching degree is 2, and for $\mathsf{CTL}^\star$ it is 3).

The logic $\mathsf{EAQCTL}^\star$ extends $\mathsf{AQCTL}^\star$ by adding existential quantification on atomic propositions: if $\forall q_1, \ldots, q_k\psi$ is an $\mathsf{AQCTL}^\star$ formula and $p_1, \ldots, p_m$ are atomic propositions, then $\exists p_1, \ldots, p_m \forall q_1, \ldots, q_k\psi$ is an $\mathsf{EAQCTL}^\star$ formula. The semantics of $\exists p_1, \ldots, p_m \forall q_1, \ldots, q_k\psi$ is given by $S \models \exists p_1, \ldots, p_m \forall q_1, \ldots, q_k\psi$ iff there is a tree (T, V) such that (T_S, V_S) and (T, V) are $\{p_1, \ldots, p_m\}$-different and $(T, V) \models \forall q_1, \ldots, q_k\psi$. The logic EAQCTL is the subset of $\mathsf{EAQCTL}^\star$ corresponding to CTL. For $n \geq 1$, let $[n] = \{1, \ldots, n\}$, and let $S_n = \langle \emptyset, [n], [n] \times [n], 1, L\rangle$ be the structure whose transition relation is the n-state clique (note that since S_n has no atomic propositions, its labeling function L is redundant). For an $\mathsf{AQCTL}^\star$ formula ψ, let $width(\psi)$ be the *sufficient branching degree* for ψ; that is, $width(\psi)$ is such that if there is some tree that satisfies ψ, then there is also a tree with branching degree $width(\psi)$ that satisfies ψ. Recall that the semantics of $\mathsf{EAQCTL}^\star$ formulas is defined with respect to the unwinding (T_S, V_S), for structures S. Hence, as detailed in [Kup97], the satisfiability problem for the AQCTL* formula $\forall q_1, \ldots, q_k \psi$ can be reduced to the model-checking problem of the EAQCTL* formula $\exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$ in $S_{width(\psi)}$. Since the formulas used in the proof of Theorems 5.7 and 5.8 have fixed widths, the following Theorem follows immediately from Theorems 5.7 and 5.8.

Theorem 5.9 *The model-checking problems for* EAQCTL *and* EAQCTL* *are* 2-EXPTIME*-hard and* 3-EXPTIME*-hard in the size of the specification, respectively.* □

Intuitively, the model-checking problem for EAQCTL* asks whether we can find an assignment to the propositions that are existentially quantified so that no matter how we assign values to the propositions that are universally quantified, the formula is satisfied. Recall that in the control problem we ask a similar question, namely whether we can find a strategy for the system so that no matter which strategy the environment uses, the formula is satisfied. In the theorem below we make the relation between existential and universal quantification over atomic propositions and supervisory control formal. The relation is similar to the relation between existential quantification and the module-checking problem, as described in [KV96].

Theorem 5.10 *Given a structure S and an* EAQCTL* *(or* EAQCTL*) formula $\exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$, there is a plant P and a* CTL* *formula (resp.* CTL *formula) ψ' such that $|P| = O((1+k+m) \cdot |S|)$, $|\psi'| = O(|S| + |\psi|)$, and $S \models \exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$ iff there is a controller for (P, ψ') against reactive environments.*

Proof For technical convenience, let us first assume that a plant has a third type of states, W_n, which belong to neither the system nor the environment (that is, all the successors of states in W_n are always taken). Let $E = \{p_1, \ldots, p_m\}$ and $U = \{q_1, \ldots, q_k\}$ be the sets of existentially and universally quantified propositions, and let $S = \langle AP, W, R, w_0, L\rangle$.

We define $P = \langle AP \cup \{yes, dummy\}, W_n, W_s, W_e, R', w_0, L'\rangle$, where

- $W_n = W \cup \{q_{yes}, q_{no}\}$.
- $W_s = W \times E$.
- $W_e = W \times U$.

- $R' = R \cup \{\langle w, \langle w, r\rangle\rangle : w \in W \text{ and } r \in E \cup U\} \cup ((W \times (E \cup U)) \times \{q_{yes}, q_{no}\}) \cup \{\langle q_{yes}, q_{yes}\rangle, \langle q_{no}, q_{no}\rangle\}$.
- For all $w \in W$ and $r \in E \cup U$, we have $L'(w) = L(w) \setminus (E \cup U)$, $L'(\langle w, r\rangle) = \{r, \mathit{dummy}\}$, $L'(q_{yes}) = \{\mathit{yes}\}$, and $L'(q_{no}) = \emptyset$.

That is, the plant P contains the structure S. Each state w in S is duplicated $k + m$ times. Each copy of w is associated with a quantified proposition. States associated with existentially quantified propositions are system states. States associated with universally quantified propositions are environment states. Each duplicated state is labeled by the proposition it corresponds to and by a new proposition *dummy*. In addition, there are two states q_{yes} and q_{no} that all the duplicated states go to. The new atomic proposition *yes* is true in q_{yes}.

We define ψ' in two steps. First, path quantification in ψ' should be restricted to computations of S. That is, to paths that never meet a duplicated state. To do this, we use a function f : $\mathsf{CTL}^\star$ formulas $\to$ $\mathsf{CTL}^\star$ formulas that restricts path quantification to paths that never visit a state labeled with *dummy*. For example, $f(\mathsf{E}q\mathsf{U}(\mathsf{AF}p)) = \mathsf{E}((\mathsf{G}\neg \mathit{dummy}) \wedge (q\mathsf{U}(\mathsf{A}((\mathsf{F}\mathit{dummy}) \vee \mathsf{F}q))))$. The full definition of f and a proof that when ψ is a CTL formula, there is also a CTL formula equivalent to $f(\psi)$, can be found in [KG96, KV96]. We can now define ψ' as $f(\psi)$ with $\mathsf{EX}(r \wedge \mathsf{EX}\mathit{yes})$ replacing each occurrence of a quantified proposition r. So, if r is existentially quantified, the system chooses whether it holds or not (by enabling/disabling the transition from the state $\langle w, r\rangle$ to the state q_{yes}), and dually for universal quantification. Note that we first apply f and only then do the replacement. The length of the formula ψ' is linear in the length of ψ.

Finally, we remove the assumption about a plant having a third type of states by adding to ψ' a conjunct that disables the pruning of transitions from W_n. Formally, this is done by adding W to AP, and having formulas like $\mathsf{AG}(w_1 \to (\mathsf{EX}w_2 \wedge \mathsf{EX}w_3))$ that describe R. This is why the length of ψ' is $O(|\psi| + |P|)$. □

Since the number of atomic propositions in the formulas used in the reductions in Theorems 5.7 and 5.8 is fixed, and since in the case P is fixed the size of ψ' in Theorem 5.10 is $O(|\psi|)$, we can conclude with the following.

Theorem 5.11 *The supervisory control problems for* CTL *and* $\mathsf{CTL}^\star$ *are* 2-EXPTIME-*hard and* 3-EXPTIME-*hard in the size of the specification, respectively.* □

5.5 Conclusions

Our results shed additional light on the discussion regarding the relative merits of linear versus branching temporal logics, cf. [Lam80, Pnu85]. We mainly refer here to the linear temporal logic LTL and the branching temporal logic CTL. One of the beliefs dominating this discussion has been "while specifying is easier in LTL, model checking is easier for CTL". As is argued in [KV96, KV97a, KV99b], the computational advantage of CTL over LTL (cf. [CGP99]), disappears once one considers reactive environments. Our results here show that the same phenomenon occurs in the context of the synthesis problem and the control problem: once one considers reactive environments, the problems for LTL and CTL are equally hard (2-EXPTIME complete).

Note that for LTL, the environment being reactive or universal makes no difference: if a controller meets the specification in a universal environment, then it meets the specification for all possible environments. This is because the set of paths generated when playing against the universal environment subsumes the paths generated when playing against any environment.

In a setting with *incomplete information*, the system (resp., the environment) may not be able to observe all the signals generated by the environment (resp., the system), so a strategy needs to depend only on the observed signals. The effect that incomplete information has on the complexity of the synthesis and control problems can vary dramatically, from having no impact [KV97b] to causing undecidability [PR90]. An interesting question that deserves further study is whether one can handle incomplete information in our setting within the same complexity bounds.

Chapter 6

Distributed Control

They roused him with muffins–they roused him with ice–
They roused him with mustard and cress–
They roused him with jam and judicious advice–
They set him conundrums to guess.

— *The Hunting of the Snark*, Lewis Caroll

6.1 Introduction

The aim of the previous chapters has been the study of the automatic design of controllers for various branching-time specification mechanisms. In all the problems considered so far (except that of Chapter 3), we have assumed that the controller is centralized and can observe every move of the plant. In many settings, however, the control problem arises in a distributed setting, where there are many programs, distributed across a network, that interact with different environments and have some capability to communicate with each other. In these settings, the controllers we look for have to respect the distributed nature of the system. We need to build controllers at the various sites and these controllers may not have complete information about the evolution of the plant at the other sites. In this chapter, we study a version of the distributed control-synthesis problem for a set of programs that behave synchronously and communicate using messages.

Let us say that we have a plant that has k processes, $P_1, P_2, \ldots, P_k$, that interact with their local environments and can communicate with each other in some fixed manner. The control problem in this setting is to come up with controllers $C_1, \ldots, C_k$ for the programs such that C_i is a controller for P_i, for each i. The important point is that a local controller C_i may not have complete information of the inputs fed to the other programs by their local environments, and hence will not know the exact states the other programs are in. However, it is not the case that the processes are completely ignorant of each other's configurations — they can pass information along the fixed communication channels between them. These channels may not be able to convey the entire state-evolution of a program at a site to another site. However, there is some partial information exchange that is possible. The problem is to come up with controllers that will control the plant and the messages sent, so that the programs can exchange enough information to behave in a way so that the specification is met. (We are not fixing many details, like the mode of communication, specification language, etc. — but these remarks are independent of these details.)

There are two ad-hoc ways in which we can (partially) solve the distributed-synthesis problem. One of them was briefly outlined in Chapter 4. Note that the system can be viewed as a set of k players playing a game against an environment. The distributed controller we are looking for is a prescription of how each player can play such that they all win the game. One way to solve this is to assume that the players are in fact playing *against* each other and trying to win the game no matter how the other players play. If we can come up with a set of strategies at each site so that, if a site plays according to its strategy, then it will win no matter how the other players play, then surely this set of strategies will win against the environment. Hence we would have a controller for the plant. However, if we cannot find such a set, it does not mean that there is no distributed controller — it may be that individual players may not be able to win by themselves but only by cooperating in some manner. Hence, this method to solve the control-synthesis problem is not complete.

Another way to solve distributed control without meeting it head-to-head, is to view the system as a global system and build a centralized controller for it. This controller is one which can view all the inputs at all sites and control the behaviour in a global manner so that it meets the specification. Existence and building of such a global controller is often decidable, as shown in [Tho95, PR89a] and the earlier chapters in this thesis. One can then try to see if this (particular) global strategy can be realized as a distributed one — i.e. we can try and decompose this strategy into local strategies at sites. This is decidable sometimes (for example Pnueli and Rosner show this is decidable in their setting [PR90]), and if one succeeds in decomposing it, then we have a distributed controller. However, again, if we don't, we cannot be sure that there is no distributed controller — for there might exist some other global strategy that is decomposable. Hence this method too is sound, but not exhaustive.

A frontal attack on the synthesis problem in a distributed setting was first made by Pnueli and Rosner in [PR90] (see also Rosner's Ph.D thesis [Ros92]). In this paper, they study a model where the programs communicate in a synchronous manner through fixed communication channels, and show the surprising but disheartening result that the realizability problem for almost all architectures is undecidable. The problem can be seen as a multi-player game with incomplete information (as described above) studied by Peterson and Reif [PR79], and the results in [PR90] are extensions of these results. Pnueli and Rosner show that even a two-site architecture where there is no possible communication between the sites, is undecidable. They also identify a small class of architectures (called hierarchical architectures) for which the problem is decidable.

The results in [PR90], though they are meant for realizability, extend to control-synthesis as well. We show that it follows from their results that the only kind of architectures for which the control problem is decidable is the singly-flanked pipeline (called pipelines in [PR90]). A singly-flanked pipeline is a set of sites arranged in a linear order, connected one to the other along the order with internal channels and with a single environment input at the first site of the sequence (see Figure 6.2).

The decidable architectures identified above are particularly disappointing as they don't have external environment inputs at even two sites, and hence are not non-trivial examples of distributed reactive systems. The negative results for other architectures in fact stem from this property of having two external environments and lead to the scenario where there are sites that have incomplete information about each other (in the decidable architectures, for any two sites, one has complete information about the other).

The crucial idea for this chapter comes from our conjecture that the problem is intractable because we are demanding *global properties* of systems that are in fact distributed and have programs that are completely oblivious to each other. The undecidability proofs crucially use this fact that the sites cannot exchange information while the specification can demand global properties of them.

We therefore drop global specifications, and instead consider *local specifications*. Local specifications can describe, for each site, how the site ought to behave — but it cannot demand anything on the global evolution of the plants. Our hope was that this may lead to a larger class of architectures that are decidable whilst reducing the power of specifications but not making the specification too restrictive. Local logics have been studied extensively in the concurrency community [Thi94, Ram96, Zie87].

Indeed, if one considers the control-problem for the two-site architecture which is not connected, then it becomes trivially decidable for local specifications since we can design controllers for the two sites, independent of each other, against their respective specifications.

Though the idea looks promising, it turns out that the class of decidable architectures gets only mildly larger. The main result of this chapter is to classify the exact class of architectures for which the control problem for local specifications is decidable — this turns out to be the class of architectures where each connected component is a sub-architecture of a *doubly-flanked pipeline* (see Figure 6.2). On the positive side, our results show that we can design controllers for doubly-flanked pipelines, which are nontrivial reactive architectures (as they allow at least two sites for environment input). Indeed, our results show that the realizability problem can also be effectively solved in this important setting, where the specifications at the sites can state properties of the internal channels as well.

For convenience we study here only the controller synthesis problem and assume local specifications to be given as Rabin conditions over the states of the local plant. We consider Rabin conditions since they are expressive enough to state any ω-regular specification. Indeed, we could have worked instead with temporal logic specifications, one at each site, which describes the desired behaviours of the local programs. However, since one could cast this as a problem with Rabin conditions (by building a deterministic Rabin automaton over words accepting the desired behaviours [Saf88] and taking its intersection with the local plant), we can solve the problem in that setting as well.

Our undecidability results go through for weaker acceptance conditions right down to reachability. Thus our negative results show that even in the presence of local specifications, the controller synthesis problem is intractable for almost all architectures.

In the next section we introduce the formal setting for our work. In Section 6.3 we briefly review the results on control synthesis for global specifications obtained from the results of [PR90]. Section 6.4 establishes our decidability results for local specifications while Section 6.5 proves the undecidability results.

6.2 Problem setting

An *architecture* is a tuple $\mathcal{A} = (S, X, T, r, w)$ where $S = \{s_1, \ldots, s_k\}$ is a finite nonempty set of *sites*, $X = \{x_1, \ldots, x_l\}$ is a set of external (or environment) input channels and $T = \{t_1, \ldots, t_n\}$ is a set of internal channels. r is a function $r : X \cup T \to S$ which identifies for each channel a process which *reads* the channel; $w : T \to S$ identifies for each internal channel, a process which writes into it.

We assume, without loss in generality, that each process has at most one external input channel and that there is at most one channel from one site to another.

We represent architectures graphically as directed graphs whose nodes are the sites and every channel $z \in X \cup T$ is represented by an edge — if $z \in T$, then it is an edge from $w(z)$ to $r(z)$ and if $z \in X$, then it is a sourceless edge to $r(z)$. We only consider *acyclic* architectures — i.e. those architectures whose graph representation does not have a directed cycle. We will assume further that every site has at least one input (external or internal) channel.

For example, consider the architecture in Figure 6.1. It represents the architecture $\mathcal{A} = (\{s_1, s_2, s_3, s_4\}, \{x_1, x_4\}, \{t_1, t_2, t_3, t_4\}, r, w)$ where $r(x_1) = s_1$, $r(x_4) = s_4$, $r(t_1) = s_2$, $w(t_1) = s_1$, $r(t_2) = s_3$, $w(t_2) = s_1$, $r(t_3) = s_4$, $w(t_3) = s_2$, $r(t_4) = s_4$ and $w(t_4) = s_3$. The external channels are represented by dotted lines.

As done in [PR90], we could have architectures with *external output channels* as well. However, since we state our specifications in terms of how the sequences evolve and not what is output along channels, and since specifications involving the values output on channels can be converted to state-based specifications, we omit these external output channels.

In our framework, each site runs a program which reads its external and internal

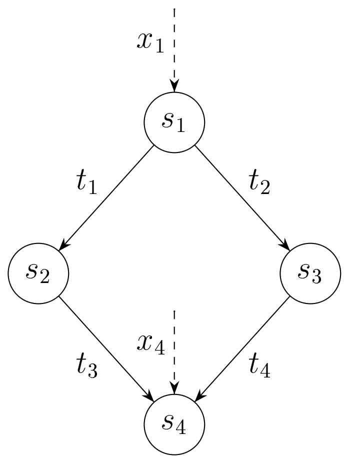


Figure 6.1: An architecture

channel inputs and reacts by sending outputs along the internal channels to other processes and changing its state. The moves are synchronous — i.e. the programs read a set of external inputs and make one collective move while respecting the partial order imposed by the architecture.

For example, in the architecture in Figure 6.1, in a synchronous step, s_1 will read the environment's input on x_1, change its state and write onto t_1 and t_2; s_2 and s_3 will, independently, read the inputs on t_1 and t_2 respectively; s_2 and s_3 change state and write onto t_3 and t_4, respectively; finally s_4 will read both inputs on t_3 and t_4 and the external input on x_4, and change its internal state.

For a site $s \in S$, let $in(s) = r^{-1}(s)$, the set of channels which s reads from and let $out(s) = w^{-1}(s)$, the set of channels s writes into.

Given an architecture $\mathcal{A}$, a *domain definition* for $\mathcal{A}$ is a function D which associates with each $z \in X \cup T$, a finite set of values that can be sent along the channel z. We denote $D(z)$ sometimes as D_z. For a set of channels Z, a *valuation function for* Z is a function h whose domain is Z and which maps each $z \in Z$ to an element of D_z. Let $\mathcal{H}_Z$ denote the set of all valuation functions for Z.

Definition 6.1 A *reactive synchronous plant* (a plant in short) is a tuple $(\mathcal{A}, D, \widehat{P})$ where $\mathcal{A}$ is an architecture (say having k sites $\{s_1, \ldots, s_k\}$), D is a domain definition for $\mathcal{A}$, $\widehat{P}$ is a set of *programs*, one at each site — i.e. $\widehat{P}$ is a tuple $\langle P_1, \ldots P_k \rangle$. Each P_i is a nondeterministic transition system $(Q_i, q_i^{in}, \delta_i)$ where Q_i is a set of states, $q_i^{in} \in Q_i$ is the initial state, δ_i is a nondeterministic transition function $\delta_i : Q_i \times \mathcal{H}_{in(s_i)} \rightarrow \mathcal{P}(Q_i \times \mathcal{H}_{out(s_i)})$.[1] □

The transition function of each program defines the different ways in which a site can react to a set of inputs on its input channels. Each such reaction gives a set of values which can be written on the output channels together with a (possible) state change. We say a plant is *finite* if Q_i is finite for each P_i.

For example, consider an architecture with a site s_i with $in(s_i) = \{x, t_1\}$ and $out(s_i) = \{t_2, t_3\}$, where x is an external input channel and t_1, t_2 and t_3 are internal channels. Consider a reactive synchronous plant where the program at s_i is $P_i = (Q_i, q_i^{in}, \delta_i)$. Now let $\delta(q_i, h)$ have the element (q_i', g) where $q_i, q_i' \in Q_i$, h is a function that takes x to an element in D_x and t_1 to an element in D_{t_1}, and g is a function that takes t_2 and t_3 to elements in D_{t_2} and D_{t_3} respectively. Then this means that the plant, when at state q_i, and reading the inputs $h(x)$ on channel x and $h(t_1)$ on channel t_1, can write $g(t_2)$ onto channel t_2 and $g(t_3)$ onto channel t_3 and change its state to q_i'.

Let $(\mathcal{A}, D, \widehat{P})$ be a plant. For a program $P = (Q, q^{in}, \delta)$ at a site s in $\mathcal{A}$, a *local strategy for* P is a function $f : Q \times \mathcal{H}^+_{in(s)} \rightarrow Q \times \mathcal{H}_{out(s)}$ such that $\forall q \in Q$, $\pi \in \mathcal{H}^+_{in(s)}$, if $\pi = \pi' \cdot h$ then $f(q, \pi) \in \delta(q, h)$. Thus the local strategy f is an advice function for P which looks at the history of values (π') on the local input channels and the current values on the local input channels (h), and prescribes a move which the local program P should take.

A *distributed control-strategy* is a set of local strategies, one for each site: i.e. a tuple $f = \langle f_1, \ldots f_k \rangle$ where each f_i is a local strategy for P_i. We sometimes refer to distributed control strategies as simply a distributed controller. A distributed controller is said to be finite-state, if it can be realized as a finite-state transition system. Let $f = \langle f_1, \ldots f_k \rangle$ and let the state-space of P_i be Q_i for each i. Formally, f is finite-state if there is a (complete) transition system TS_i, for each f_i, on the alphabet $\mathcal{H}_{in(s_i)}$ and a function g from the state-space of TS_i to the set of functions from Q_i to $Q_i \times \mathcal{H}_{out(s_i)}$ such that for every $x \in (\mathcal{H}_{in(s_i)})^+$ and $q \in Q_i$, if the transition system reaches state u on x, then $g(u)(q) = f(q, x)$.

We will call a plant along with a strategy $((\mathcal{A}, D, \widehat{P}), f)$ a *controlled system*. Let us fix for now a controlled system $((\mathcal{A}, D, \widehat{P}), f)$.

[1] $\mathcal{P}(R)$ denotes the power-set of R

We need some notations for talking about sequences. For a sequence α, let $\alpha[i]$ denote the i^{th} element in α and $\alpha[i,j]$ denote the finite subsequence of α from the i^{th} to the j^{th} element, both inclusive, for $0 \leq i \leq j$, $i, j \in \mathbb{N}_0$. If α is a sequence of functions on a domain Z, let $\alpha \downarrow Z'$, where $Z' \subseteq Z$ denote the sequence of functions obtained by restricting each function in α to Z': i.e. $\alpha \downarrow Z' = \beta$ where domain of each $\beta[i]$ is Z' and $\beta[i](z) = \alpha[i](z)$ for each $z \in Z'$.

Consider an environment input sequence (on the channels in X) $\alpha \in (\mathcal{H}_X)^\omega$. Since P, when controlled by the strategy f, is deterministic, there is a unique way in which the plant and controller respond to the external input — i.e. there is a unique sequence of states each program takes and a unique set of channel values sent along each channel. We can define these sequences as follows. Let $\beta \in (\mathcal{H}_{X \cup T})^\omega$ and $\gamma \in (Q_1 \times \dots Q_k)^\omega$ such that:

1. $\gamma[0] = \langle q_1^{in}, \dots, q_k^{in} \rangle$
2. $\beta \downarrow X = \alpha$
3. $\forall t \in T$, $j \in \mathbb{N}_0$, if $w(t) = s_i$ and $\gamma[j] = \langle q_1, \dots q_k \rangle$ with $f_i(q_i, \beta[0,j] \downarrow in(s_i)) = (q_i', h)$, then $\beta[j](t) = h(t)$
4. $\forall j \in \mathbb{N}_0$, if $\gamma[j] = \langle q_1, \dots q_k \rangle$ and $\gamma[j+1] = \langle q_1', \dots q_k' \rangle$, then it must be the case that $\forall i \in \{1, \dots, k\}$: $f_i(q_i, \beta[0,j] \downarrow in(s_i)) = (q_i', h)$ for some $h \in \mathcal{H}_{out(s_i)}$

The definitions above formalize how the programs behave when they get an external input. (1) says that the global state-sequence starts with the initial states. The next condition requires that the values which the external channels take are defined by the external input α. (3) demands that the internal channels take values according to the move defined by the local strategy and (4) ensures that the states also evolve according to the moves given by the local strategies.

It is easy to see that, since the architecture is acyclic, there are unique sequences β and γ which satisfy the above conditions. We call γ the *state-behaviour* of the system for the input α.

Global specifications

A global specification describes the set of sequences the plant is allowed to generate globally. Hence it is just a subset $\mathcal{G}_{spec}$ of $(Q_1 \times \dots Q_k)^\omega$.

A controlled plant, when given an external input stream α on the external channels, produces a state-behaviour γ as described above. The controlled plant meets the specification $\mathcal{G}_{spec}$ if for every possible input stream α, the state-behaviour γ produced is in $\mathcal{G}_{spec}$.

The set $\mathcal{G}_{spec}$ can be specified in various ways. For example, it could be specified using LTL formulas where the atomic propositions are the local states of the plant with the understanding that a state-behaviour γ is interpreted as an infinite sequence γ' over subsets of states where if $\gamma[i] = \langle q_1, \ldots, q_k\rangle$, then $\gamma'[i] = \{q_1, \ldots, q_k\}$. A more powerful mechanism would be to specify $\mathcal{G}_{spec}$ as an ω-automaton over infinite words over the alphabet 2^Q where Q is the union of the sets of local states [Tho90].

Local specifications

Local specifications are defined on the *local* state-behaviours of the programs of the plant. Since we wish to capture local linear-time properties, we define local Rabin winning conditions and the specification then demands that the local runs of the controlled system meet these conditions.

A *local Rabin winning condition* $\mathcal{R}_i$ for a site s_i is a set $\{(R_1, G_1), \ldots (R_m, G_m)\}$ where R_j, G_j are subsets of Q_i, the state-space of the program at s_i. A *Rabin winning condition* $\mathcal{W}$ for a plant is a tuple $\langle \mathcal{R}_1, \ldots \mathcal{R}_k\rangle$ where each $\mathcal{R}_i$ is a local Rabin winning condition for s_i.

Let $\gamma \in (Q_1 \times \ldots Q_k)^\omega$ be a sequence of global states of the system. Let $\gamma \downarrow i$ denote the sequence in Q_i^ω obtained by projecting γ to the component involving Q_i. A sequence of global states γ is said to *satisfy* a Rabin condition $\mathcal{W}$ if for each site s_i, $\gamma \downarrow i$ satisfies the local winning condition $\mathcal{R}_i$, i.e. if for each site s_i, there is a pair (R, G) in $\mathcal{R}_i$ such that $\mathit{inf}(\gamma \downarrow i) \cap R = \emptyset$ and $\mathit{inf}(\gamma \downarrow i) \cap G \neq \emptyset$.

Finally, a controlled system is said to satisfy a Rabin winning condition $\mathcal{W}$ if for every sequence of external inputs $\alpha \in (\mathcal{H}_X)^\omega$, the state-behaviour γ defined by α satisfies $\mathcal{W}$.

Note that, like in global specifications, we could have defined a set of local specifications where the local specification of a site s is description of set of "local state-sequences" the site can go through. However, any such ω-regular set of local sequences at a site can be *determinized* to get a deterministic Rabin automaton on infinite words ([Saf88, Tho97]). One can take a product of this deterministic automaton with the concerned local plant to reframe the specification in terms of a

Rabin winning condition.

Control synthesis problem

Given a local or global specification, a distributed control-strategy f for a plant $(\mathcal{A}, D, \widehat{P})$ is said to be *winning* if the controlled system $((\mathcal{A}, D, \widehat{P}), f)$ satisfies the winning condition. Note that the strategies are always local. We henceforth just say "strategy" to mean a distributed control-strategy.
We can now state the control-synthesis problem for an architecture $\mathcal{A}$:

Definition 6.2 [**Control problem for $\mathcal{A}$ against global specifications**]
Given a finite reactive plant $(\mathcal{A}, D, \widehat{P})$, and a global specification $\mathcal{G}_{spec}$, does there exist a winning strategy for the plant? □

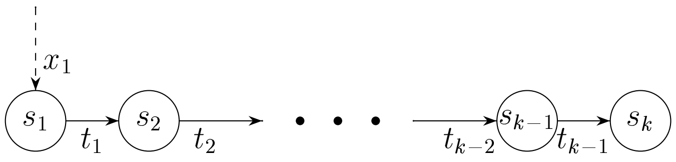
Definition 6.3 [**Control problem for $\mathcal{A}$ against local specifications**]
Given a finite reactive plant $(\mathcal{A}, D, \widehat{P})$, and a Rabin winning condition $\mathcal{W}$, does there exist a winning strategy for the plant? □

Our main aim is to classify those architectures for which, given a domain definition and a plant, the control problem is decidable.

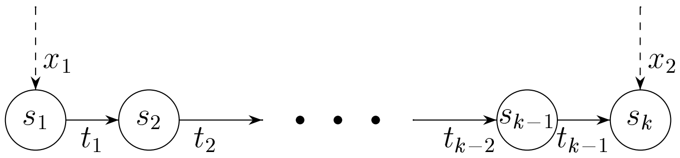
In this connection, two important classes of architectures are the *singly-flanked pipelines* and the *doubly-flanked pipelines*. Singly-flanked pipelines are pipelines that have external inputs only at the left end while doubly-flanked pipelines have external inputs at both ends (see Figure 6.2):

Definition 6.4 An architecture $\mathcal{A}$ is said to be a *pipeline* if the sites in $\mathcal{A}$ are $s_1, \ldots s_k$ (for some $k \in \mathbb{N}$) and there are exactly $k-1$ internal channels $t_1, \ldots t_{k-1}$, with $w(t_i) = s_i$ and $r(t_i) = s_{i+1}$ for $i \in \{1, \ldots, k-1\}$. A *singly-flanked pipeline* is a pipeline that has a single external channel x_1 with $r(x_1) = s_1$. A *doubly-flanked pipeline* is a pipeline that has exactly two external channels x_1 and x_2 with $r(x_1) = s_1$ and $r(x_2) = s_k$. □

We will also need the notion of a sub-architecture — an architecture $\mathcal{A}' = (S', X', T', r', w')$ is a sub-architecture of an architecture $\mathcal{A} = (S, X, T, r, w)$ if the graph of $\mathcal{A}'$ is isomorphic to a subgraph of the graph of $\mathcal{A}$: i.e. there is a 1-1 function $g : S' \cup X' \cup T' \to S \cup X \cup T$ such that $g(S') \subseteq S$, $g(X') \subseteq X$, $g(T') \subseteq T$,



A generic singly-flanked pipeline



A generic doubly-flanked pipeline

Figure 6.2: Flanked pipelines

for each $x' \in X'$, $g(r'(x')) = r(g(x'))$, and for each $t' \in T'$, $g(r'(t')) = r(g(t'))$ and $g(w'(t')) = w(g(t'))$. Note that an architecture is a sub-architecture of itself.

The control problem for global specifications has already been virtually settled by Pnueli and Rosner [PR90]. It follows from their results that there is only one kind of architecture for which the control problem is decidable — the singly-flanked pipeline architectures.

The main result of this chapter is to give identify the precise class of architectures for which the control problem for local specifications is decidable. This class is the class of all architectures all of whose connected components are sub-architectures of a doubly-flanked pipeline.

6.3 Control synthesis against global specifications

Pnueli and Rosner show in [PR90] that a variant of the control problem, namely the realizability problem, is decidable for the singly-flanked pipeline. The realizability problem is awkward to formalize in our setting — it is better stated in a setting where sites can have external output channels as well. Then, the realizability problem is one where we are given an architecture and a specification describing sequences of values output by each site, and are asked whether there is a program at each site that realizes the specification. A program at a site now not only outputs values along internal channels, but also along the external output channels. Also, the program is not constrained in any way — it can output any set of values at any point, unlike the strategies in the control problem.

The proof in [PR90] can however be easily modified to handle the control problem and we have:

Theorem 6.1 ([PR90]) *The control problem for the class of singly-flanked pipeline architectures against global specifications is decidable.* □

In [PR90], it is also shown that the two-site architecture with no internal channels, i.e. the architecture $\mathcal{A}_\perp = (\{s_1, s_2\}, \{x_1, x_2\}, \emptyset, r, w)$ with $r(x_1) = s_1$ and $r(x_2) = s_2$, is one for which the realizability problem (and hence the control problem) is undecidable. In fact they show this for LTL and even weaker "reachability" specifications. Using this, we can in fact show that the control problem for an architecture that is not a singly-flanked pipeline is undecidable:

Theorem 6.2 *Let $\mathcal{A}$ be an architecture that is not a singly-flanked pipeline. Then the control problem for $\mathcal{A}$ against global specifications is undecidable.*

Proof We prove this theorem by reductions from the control problem for $\mathcal{A}_\perp$. First, consider the case where $\mathcal{A}$ has two sites, s and s', both of which have external input channels. Then, given an instance of the control problem I for $\mathcal{A}_\perp$, we can produce an instance I' of the control problem for $\mathcal{A}$ by setting the programs of s and s' to be the two programs assigned in I to the two sites in $\mathcal{A}_\perp$. Also, we can make all the other sites "dummy" by making them output on any input, a fixed letter on each local output channel. In this way, we virtually cut off any way of s and s' communicating with each other. Also, we set the global specification to be any sequence where the behaviours of the programs at sites s and s' satisfy the global specification assigned in I. It is now easy to see that there is a strategy for the instance I' of the control problem for $\mathcal{A}$ iff there is a strategy for the instance I for $\mathcal{A}_\perp$.

Now assume $\mathcal{A}$ is an architecture that doesn't have two such sites and neither is it a singly-flanked pipeline. Then, it is easy to see that $\mathcal{A}$ must have sites $s_0, \ldots, s_i$ (where $i \geq 0$) and two sites s and s' such that s_0 has an external input, there is an internal channel from s_j to s_{j+1} for each $j \in \{0, \ldots i-1\}$ and there is an internal channel from s_i to s and one from s_i to s'. (Note that we do not claim that there are no other sites or channels in $\mathcal{A}$.) Now we reduce an instance I for $\mathcal{A}_\perp$ to $\mathcal{A}$ by setting the programs in I to s and s'. Also, the combined inputs for the programs will be input into s_1, which will propagate it through the internal channels to s_i. The program at s_i will separate these inputs and feed them to s and s'. All other sites will be "dummy" as in the previous reduction. Note that no program at any site except those at s and s' are really controllable (i.e. the programs at other sites are deterministic). The global specification for the new instance I' contains all sequences where the programs at s and s' satisfy the global specification of I. It is again easy to see that there is a strategy for I' in $\mathcal{A}$ iff there is a strategy for I in $\mathcal{A}_\perp$. □

Observe that the above proof does not go through for the problem of realizability of distributed programs, for we are crucially using the programs at "dummy" sites to force the channels to carry only a fixed value. In [PR90], the authors define architectures so that they also take into account the number of values a channel can take (i.e. the domain function) and identify a larger class of architectures for which realizability is decidable. It is still an open problem to identify the exact class of architectures for which the realizability problem is decidable.

6.4 Local specifications: Decidable architectures

In this section, we show that for architectures where each connected component is a sub-architecture of a doubly-flanked pipeline, the control problem against local specifications is decidable. Firstly, since we have local winning conditions, it is easy to observe the following.

Proposition 6.1 *The control problem for an architecture $\mathcal{A}$ is decidable iff the control problem is decidable for each of its connected components.*

Proof Clearly, the control problem for an architecture $\mathcal{A}$ can be reduced to a set of control problems, one for each component, by splitting the specification for the components. This can be done as the specification is local. Also, the control problem for a component $\mathcal{A}'$ of $\mathcal{A}$ can reduced to a control problem for $\mathcal{A}$ by inheriting the programs and specifications for sites in $\mathcal{A}'$ and by giving "dummy" programs at sites in $\mathcal{A}$ that are not in $\mathcal{A}'$, and also making the local specification on these sites always win. $\square$

Hence it suffices to prove that the problem is decidable for architectures which are sub-architectures of doubly-flanked pipelines. We use tree automata to establish our results. Recall from Chapter 4 the definition of trees and nondeterministic and alternating automata working on them.

Consider a plant $(\mathcal{A}, D, \widehat{P})$ and a distributed control-strategy f for it. Let s be a site in $\mathcal{A}$ with an output channel t.

Let $\mathcal{L} \subseteq (D_t)^\omega$ be the language of infinite strings output on t by the controlled system $((\mathcal{A}, D, \widehat{P}), f)$ (by considering all possible inputs on the external input channels of the plant). We call such a language of infinite words, *a communication language for the channel* t. (Note that $\mathcal{L} \neq \emptyset$.) Let $\mathit{Pref}(\mathcal{L}) = \{x \mid \exists \beta \in \mathcal{L},\ x \textit{ is a prefix of } \beta\}$. Then it is not difficult to see that $\mathcal{L} = \mathit{lim}(\mathit{Pref}(\mathcal{L}))$ where $\mathit{lim}(L) = \{\alpha \in \Sigma^\omega \mid \textit{ for every prefix } x \textit{ of } \alpha,\ x \in L\}$. Though this is true for any architecture, we show it only for doubly-flanked pipelines as this will suffice for our purpose.

Proposition 6.2 *Let $(\mathcal{A}, D, \widehat{P})$ be a reactive plant and let $\widehat{f}$ be a distributed controller for it with $\mathcal{A} = (S, X, T, r, w)$ a doubly-flanked pipeline. Let $t \in T$ be an internal channel. Then, for sequences of external inputs $\alpha \in (\mathcal{H}_X)^\omega$, let the sequence of values output on t be $\mathcal{L} \subseteq D(t)^\omega$. Then $\mathcal{L} = \mathit{lim}(\mathit{Pref}(\mathcal{L}))$.*

Proof Let us fix a doubly-flanked pipeline, say a canonical one with k sites as illustrated in Figure 6.2. Let us first do the proof for the first internal channel. Let s_1 be the first site of the pipeline, with external input channel x_1 and output channel t_1. Let P_1 be the program at s_1 and let f_1 be a local strategy at s_1. Let us show that if the language of strings written onto t_1 is $\mathcal{L}$, then $\mathcal{L} = \mathit{lim}(\mathit{Pref}(\mathcal{L}))$. Clearly, $\mathcal{L} \subseteq \mathit{lim}(\mathit{Pref}(\mathcal{L}))$.

Let $T_{x_1} = (D(x_1)^*, E)$ be the full $D(x_1)$-tree. Now label this tree as follows: label $w \in D(x_1)^*$ by the last value output by P_1 and f_1 on t_1, when working on input w on x_1. Then, clearly one can label the whole tree (except the root) with labels from $D(t)$ and the labels on the path to a node w gives the string output on t while reading w.

Let $\alpha \in lim(Pref(\mathcal{L}))$. Then for every prefix y of α, there is a $\beta \in \mathcal{L}$ such that y is a prefix of β. Now prune the $D(x_1)$-tree as follows: at a level i, retain only the nodes which are labelled $\alpha[i]$ and then take the connected part of the tree from the root. That is, take the smallest subtree T of T_{x_1} such that

- $\varepsilon \in T$
- If $z \in T$ and $z.d \in T_{x_1}$ (where $z \in D(x_1)^*$, $d \in D(x_1)$) and if the label of the nodes in the path from the root to $z.d$ is a prefix of α, then $z.d \in T$

For any prefix y of α, since there is a $\beta \in \mathcal{L}$ such that y is a prefix of β, along the path in the $D(x_1)$-type tree which generates β, there will be a node such that the path to it is labelled by y. Hence this node and all the nodes in the path from the root to it will be in the pruned tree T. Hence the tree T is infinite. Since $D(x_1)$ is finite, by König's lemma, there must be an infinite path in this tree. Surely this is labelled by α. Hence $\alpha \in \mathcal{L}$.

We can, by induction over the j, show that the property holds for the j'th internal channel t_j in the pipeline. In the induction step, we start with the assumption that the sequences fed into the input channel t_{j-1} to a site s_j satisfies the property (when $j = 1$, t_{j-1} is interpreted as x_1). We consider then the subtree of the $D(t_{j-1})$-tree, that represents the set of input sequences (since the language of input sequences satisfies the required property, we can find a subtree such that the infinite paths in the subtree is exactly the set of input sequences). Now consider the output channel t_j from s_j. We proceed to label the tree with the values output by the controlled program at s_j on the corresponding sequence of inputs, as above. Then, by a similar argument, we can establish that the language of infinite strings output on t_j also satisfies the required property. □

So $L = Pref(\mathcal{L}) \subseteq D_t^*$, the set of finite sequences sent along t represents the set of infinite sequences sent along the channel as well. Let $\mathcal{L}$ be a communication language of the channel t and $L = Pref(\mathcal{L})$. Then $\mathcal{L}$ can be represented (uniquely) by a $\{\top, \bot\}$-labelled D_t-tree $T = (D_t^*, \tau)$, where $\tau(x) = \top$ if $x \in L$ and $\tau(x) = \bot$, otherwise. In such a tree if a node is labelled $\top$ then it will have at least one child also with label $\top$ and if a node has label $\bot$ then all its children (and hence the entire subtree below it) will be labelled $\bot$. Also, the root, ϵ is labelled $\top$. Clearly each such $\{\top, \bot\}$-labelled D_t-tree uniquely represents a communication language of the channel t. In what follows, we shall refer to such trees as *t-type* trees and work with automata running over t-type trees. The t-type trees will also be referred to as *communication trees of* t. If T is a t-type tree then we let $Lang(T)$ denote the language of infinite strings it represents.

Let us fix a doubly-flanked pipeline which has k sites, as shown in Figure 6.2. We refer to s_1 as the *left-site*, s_k as the *right-site* and each of the s_i's, $1 < i < k$ as *middle-sites*.

We need one more notion before proceeding with the main constructions.

Definition 6.5

- Let s be the left-site of a doubly-flanked pipeline with input channel x and output channel t. Suppose P is the program at s. Then, a language of infinite words $\mathcal{L} \subseteq D(t)^\omega$ is said to be an *s-successful* language if there is a local *winning* strategy f at s such that the sequences of outputs produced on t, when P and f work on all possible input sequences on x, is $\mathcal{L}$.

- Let s be the right-site of a doubly-flanked pipeline with input channels t and x. Let P be the program at s. We say that a language $\mathcal{L}$ of infinite strings over D_t is *s-successful*, if there is a strategy f at s which can work on the input sequences in $\mathcal{L}$ on channel t and arbitrary inputs on channel x, and win locally.

- Let s be a middle-site of a doubly-flanked pipeline with input channel t and output channel t'. Let s host a program P. We say that $\mathcal{L}' \subseteq D(t')^\omega$ is *successfully generable by s on $\mathcal{L} \subseteq D(t)^\omega$* if there is a strategy at s that wins locally when reading inputs from $\mathcal{L}$ on t and the sequences of outputs produced on t' is (precisely) $\mathcal{L}'$. □

When we do the construction of automata below for various languages, we would like the automata to accept only communication trees. Instead of making sure each time that the tree that is being read is indeed a communication tree, we first show that the set of communication trees can be recognized by a tree automaton. In later constructions, we always assume that we take the intersection with the automaton that accepts the set of communication trees.

Proposition 6.3 *Let t be an internal channel. Then there is an automaton which accepts the communication trees of t.*

Proof The automaton is an alternating Büchi automaton $A = (Q, q_\top, \delta, \mathcal{F})$ where $Q = \{q_\top, q, q_\perp\}$, $\mathcal{F} = Q$ and δ is defined as:

- $\delta(q_\top, \top) = (\bigvee_{d \in D(t)}(q_\top, d)) \wedge (\bigwedge_{d \in D(t)}(q, d))$
- $\delta(q_\top, \perp) = \mathsf{false}$
- $\delta(q, \top) = \delta(q_\top, \top)$
- $\delta(q, \perp) = \bigwedge_{d \in D(t)}(q_\perp, d)$
- $\delta(q_\perp, \top) = \mathsf{false}$
- $\delta(q_\perp, \perp) = \delta(q, \perp)$

The states $q_\top$ and q make sure that the root is labelled $\top$ and that every $\top$-labelled node has a $\top$-labelled child. The state $q_\perp$ keeps track whether the subtree below a $\perp$-labelled node is fully labelled $\perp$. □

Lemma 6.4 *Let s be the left-site of a doubly-flanked pipeline with input channel x, output channel t and program P. Then there is an alternating tree automaton (on t-type trees) which accepts a t-type tree T iff $Lang(T)$ has an s-successful sublanguage.*

Proof The automaton we construct, while running over a t-type tree T, guesses a local strategy f for the program at s, makes sure that f produces no string which is not in $Lang(T)$ and also checks that f is locally winning.

The automaton has, in its state-space, a component encoding which state of the program it is currently simulating. Then reading a node y of T, it does the following:

- Guess a set of moves from the current state for each possible input d in $D(x)$.
- The automaton then propagates, for each $d \in D(x)$, a copy along the direction $d' \in D(t)$ where d' is the output of the program on d according to the guessed move. The corresponding successor state of the program is also propagated and the automaton will check in these copies whether the labels of the nodes it reads are $\top$. This will ensure that the outputs are allowed by T.

Let the local Rabin winning condition be $\mathcal{R}$. The acceptance condition ensures that paths on a run encode state-sequences which satisfy $\mathcal{R}$ — this ensures that the guessed local strategy is a winning one.

Formally, the automaton is defined as follows. Let the program be $P = (Q, q_{in}, \delta)$ where the transition system of the program is a function $\delta : Q \times \mathcal{H}_{\{x\}} \to \mathcal{P}(Q \times \mathcal{H}_{\{t\}})$. Since there is only one input and one output channel, we can view δ as a function $\delta : Q \times D(x) \to \mathcal{P}(Q \times D(t))$ with the obvious interpretation.

The alternating Rabin automaton A is defined as $A = (Q, q_{in}, \delta', \mathcal{R})$ where δ' is given by:

- Let $q \in Q$. Let a *partition at q* be a function $g : D(x) \to Q \times D(t)$ such that for every $d \in D(x)$, $g(d) \in \delta(q, d)$. Let G_q be the set of all partitions at q. Then
$$\delta'(q, \top) = \bigvee_{g \in G_q} \bigwedge_{d \in D(x)} g(d)$$
- $\delta'(q, \bot) = \mathsf{false}$, for every $q \in Q$

The first transition says that when the program is in state q and reading $\top$, it guesses a set of moves on each $d \in X$. The automaton propagates a copy for each input $d \in D(x)$ along the direction corresponding to the program's output on d. The second transition says that these propagated states should read a $\top$, verifying that the outputs guessed are allowed by the tree.

Note that a node in a run-tree of the above automaton corresponds to a unique input history sequence on the channel x. This is why guessing the strategy of P independently at the nodes of the run is justified. $\square$

Note that the automaton in the above construction accepts a tree provided the language represented by the tree merely *contains* an s-successful language. It seems hard to strengthen this containment to equality. However, the present version will suffice.

Lemma 6.5 *Let s be the right-site of a pipeline with $in(s) = \{x, t\}$ and let the program at s be P. Then there is an alternating tree automaton on t-type trees which accepts a tree T iff the language that T represents is s-successful.*

Proof The automaton will guess a local strategy for P at s on input sequences $\alpha \in Lang(T)$ along t and arbitrary input sequences $\beta \in D(x)^\omega$ on x and make sure that f is winning for all local runs on these sequences.

The automaton will keep track in its state-space the current state of P it is simulating. Reading a node y of the input tree, it will do the following:

- Guess $Y \subseteq D(t)$ corresponding to the set of successors of y labelled $\top$. The automaton will (in its next move) check if Y is indeed the set of $\top$-successors.

- The strategy has to handle all inputs in Y on the channel t along with an arbitrary input in $D(x)$ on channel x. The automaton guesses such a strategy at this point by guessing moves from the current state of P on each $h \in \mathcal{H}_{\{x,t\}}$ with $h(t) \in Y$. It then propagates along each direction d in Y, one copy of the automaton for each $d' \in D(x)$ corresponding to the chosen move when channel t carries d and channel x carries d'. It propagates the corresponding state of P as well.

Let $\mathcal{R}$ be the local winning condition. The acceptance condition for the automaton makes sure that all paths on a run encode a state-sequence in P which satisfies $\mathcal{R}$.

Formally, let $P = (Q, q_{in}, \delta)$, where the transition function of the program is a function $\delta : Q \times \mathcal{H}_{\{x,t\}} \to \mathcal{P}(Q)$. Since there is only one internal input and one external input channel, we will use a function $\delta : Q \times D(t) \times D(x) \to \mathcal{P}(Q)$ with the obvious interpretation. Let $\mathcal{R}$ be the local winning condition for P.

The alternating Rabin automaton is defined as follows: $A = (Q \cup \{q_\perp\}, q_{in}, \delta', \mathcal{R})$ where δ' is defined as follows:

- For a set $Y \subseteq D(t)$, let a Y *guess at* q be a function $g : Y \times D(x) \to Q$ such that for every $d \in Y$, $d' \in D(X)$, $g(d,d') \in \delta(q,d,d')$. Let $G_{Y,q}$ be the set of all Y guesses at q. Then

$$\delta'(q,\top) = \bigvee_{\emptyset \neq Y \subseteq D(t)} \{ (\bigvee_{g \in G_{Y,q}} \bigwedge_{d \in Y} \bigwedge_{d' \in D(x)} (g(d,d'),d) \ ) \wedge \bigwedge_{d \notin Y} (q_\perp, d) \}$$

- $\delta'(q,\perp) = \mathsf{false}$, for every $q \in Q$

- $\delta'(q_\perp,\top) = \mathsf{false}$

- $\delta'(q_\perp,\perp) = \mathsf{true}$

The first transition says that when the plant is in state q and reading $\top$, it guesses the set Y of the $\top$-successors of this node, a way to choose transitions from this state on every possible input in Y on t and any input on x. The automaton propagates a copy, for each input on $d \in Y$ and $d' \in D(x)$, along the direction d with the corresponding successor state of the program according to the guessed move. It also propagates the state $q_\perp$ along the directions not in Y.

The other transitions check whether the guess of Y in the previous step was correct. Again, since each node in a run-tree corresponds to a unique input history on t and x, the guessing of the strategy at these points independently is justified. $\square$

Theorem 6.3 *The control problem for the two-site doubly-flanked pipeline is decidable.*

Proof Let the sites and channels of the pipeline be labelled as in Figure 6.2. Using Lemma 6.4, construct an automaton A_1 which accepts a t_1-type tree T iff s_1 can successfully generate a sublanguage of $Lang(T)$. Using Lemma 6.5, construct A_2 which accepts t_1-type trees which represent languages which s_2 can win on. The claim now is that a distributed winning strategy exists iff $L(A_1) \cap L(A_2)$ is nonempty.

Assume $T \in L(A_1) \cap L(A_2)$ and let $\mathcal{L}$ be the language it represents. Then there is a strategy f_2 at s_2 which will win on $\mathcal{L}$. Also, there is a local winning strategy f_1 at S_1 which will generate a sublanguage $\mathcal{L}'$ of $\mathcal{L}$. However, since the local winning conditions are linear-time specifications, f_2 will win on $\mathcal{L}'$ as well. Hence $\langle f_1, f_2 \rangle$ is a distributed winning strategy. Furthermore, one can construct, from the runs of A_1 and A_2 on a regular tree in $L(A_1) \cap L(A_2)$, a strategy which can be realized as finite-state transition systems.

It is easy to see that if $\langle f_1, f_2 \rangle$ is any winning distributed strategy, then the tree corresponding to the language f_1 generates is accepted by A_1 as well as A_2. $\square$

Lemma 6.6 *Let s be a middle-site of a doubly-flanked pipeline with $in(s) = \{t\}$ and $out(s) = \{t'\}$, and let the program at s be P. Let A be a nondeterministic automaton accepting t-type trees. Then there is an automaton on t'-type trees that accepts a tree T' iff there is a t-type tree T accepted by A and a language $\mathcal{L}_0 \subseteq Lang(T')$ such that $\mathcal{L}_0$ is successfully generable by s on $Lang(T)$.*

Proof Let T' be an input to the automaton and $\mathcal{L}'$ be the language it represents. The automaton, while reading T', will guess a t-type tree T, guess a run of A on T, guess a strategy f for P on the input strings represented in T and make sure that the run on T is accepting, make sure that the strategy outputs strings which are included in $\mathcal{L}'$ and make sure that the strategy locally wins!

A node in the run on T' will correspond to a node y' in T' as well as a node x of the tree T being guessed — here x is the sequence in $D(t)^*$ on which the guessed strategy has output y'. Note that each sequence in $D(t)^*$ can lead to at most one sequence in $D(t')^*$ being output and hence guessing of the tree T at nodes of the run is justified.[2]

The state-space of the automaton will code both the current state of P as well as a state of the automaton A which represents the state-label of the corresponding node in T, in the guessed run on T. The automaton at a node in the run corresponding to the node y' in T' and x in T will do the following:

- Guess the set $Y' \subseteq D(t')$ which corresponds to the children of y' in T' labelled $\top$.
- Guess the labels of the children of x in T. This is the point where T is being guessed. Let $X \subseteq D(t)$ be the children of x labelled $\top$.
- The automaton now guesses a move of P from the current state on each $d \in X$ and makes sure that the output on t is in Y'. It then propagates along each direction $d' \in Y'$ in T', many copies of itself — each corresponding to a $d \in X$ on which the guessed move outputs d'. The appropriate successor state of P is propagated. The automaton also guesses a transition of A from the node x and propagates these automaton states as well.

The acceptance condition makes sure that along any path in the run, the state-sequence of P along the run meets the local winning condition of s and the state-sequence of the automaton meets the winning condition of A.

Formally, let $P = (P, p_{in}, \delta)$ (by abuse of notation, we refer to state-space of P also as P) where $\delta : P \times D(t) \to \mathcal{P}(P \times D(t'))$ (note the change in notation). Let $\mathcal{R}$ be the local winning condition at s.

Let $A = (Q, q_{in}, \delta_A, \mathcal{F})$ be the nondeterministic Rabin automaton where $\delta_A : Q \times \{\top, \bot\} \to \mathcal{P}(\mathcal{M})$, where $\mathcal{M}$ is the set of all functions $m : D(t) \to Q$.

[2] If the site also has an external input, this will not be the case.

Fix an arbitrary element e_0 in $D(t')$. Formally, the alternating Rabin automaton on t'-type trees is defined as follows:
$A' = ((P \times Q) \cup Q, (p_{in}, q_{in}), \delta', \mathcal{F}')$ where δ' is defined as follows:

- Let $p \in P$, $q \in Q$. For a set $Y \subseteq D(t')$ and $X \subseteq D(t)$, a *partition of X into Y at p* is a function $g : X \to P \times Y$ such that $\forall d \in X$, if $g(d) = (p', d')$ then $(p', d') \in \delta(p, d)$ and $d' \in Y$. For such a function g, let $g(d)[1]$ and $g(d)[2]$ denote the P and $D(t')$ components of $g(d)$, respectively. Let $\Pi^p_{X,Y}$ denote the set of partitions of X into Y at p. Now,

$$\delta'((p,q),\top) = \bigvee_{\emptyset \neq Y \subseteq D(t')} \; \bigvee_{\emptyset \neq X \subseteq D(t)} \; \bigvee_{g \in \Pi^p_{X,Y}} \; \bigvee_{m \in \delta_A(q,\top)} ((\mathbf{1}) \wedge (\mathbf{2}))$$

  where

$$(\mathbf{1}) = \bigwedge_{d \in X} (\ (g(d)[1], m(d)),\ g(d)[2]\ )$$

  and

$$(\mathbf{2}) = \bigwedge_{d \notin X} (m(d), e_0)$$

- $\delta'((p,q),\bot) = \mathsf{false}$, for all $p \in P$, $q \in Q$.

- $\delta'(q,\bot) = \delta'(q,\top) = \bigvee_{m \in \delta_A(q,\bot)} \bigwedge_{d \in D(t)} (m(d), e_0)$, for all $q \in Q$.

The acceptance condition is defined so that a path in the run-tree is accepted iff one of the following happen:

- The states along the path never leave the set $(P \times Q)$ and the first component meets the Rabin condition $\mathcal{R}$ and the second component meets the Rabin condition $\mathcal{F}$.

- The states along the path eventually land in the set Q (and hence stay there) and this infinite suffix meets the Rabin condition $\mathcal{F}$.

In the first kind of transition above, the formulas induced by $(\mathbf{1})$ are similar to the one in Lemma 6.4 except that now it is specialized to work over a guessed subset X of $D(t)$ rather than the whole of $D(t)$. It also does the additional job of guessing the automaton A's move at this point on the letter $\top$. The automaton propagates along the children corresponding to moves on X, the program state as well as the automaton state according to the guessed move.

For successor states of the automaton A on directions not in X, the automaton propagates these states along the arbitrary direction e_0. This is done by the formulas enforced by (**2**). The idea is that, since in guessing T, we know that these children are all going to read the full subtree labelled $\perp$, we just have to make sure that A accepts this subtree from each of these states. These copies of the automaton will not read the tree from this point, but simply guess some move on $\perp$ and propagate these states (as formalized in the last transition above).

Let us now turn to formalizing the acceptance condition. Let $\mathcal{R} = \{(R_1, G_1), \ldots, (R_s, G_s)\}$ and $\mathcal{F} = \{(R'_1, G'_1), \ldots, (R'_t, G'_t)\}$. We augment the state-space of the above automaton (in fact only the states that are in $(P \times Q)$) with another component, which contains the set of functions $r : \{1, \ldots s\} \times \{1, \ldots, t\} \rightarrow \{0, 1\} \times \{0, 1, 2\}$. Thus we have states of the form (p, q, r). The transition function remains the same except that the r component is updated to r' as follows: for every $i \in \{1, \ldots, s\}$, $j \in \{1, \ldots, t\}$,

- The first component of $r'(i, j)$ is 1 iff the current P-state is in R_i or the current Q-state is in R'_j.
- The second component of $r'(i, j)$ is defined as:
  - If second-component of $r(i, j)$ is 0, then $r'(i, j) = 1$ if the current P-state is in G_i, else $r'(i, j) = 0$.
  - If second-component of $r(i, j)$ is 1, then $r'(i, j) = 2$ if the current Q-state is in G'_j, else $r'(i, j) = 1$.
  - If second-component of $r(i, j)$ is 2, then $r'(i, j) = 0$.

Intuitively, the first component of $r(i, j)$ turns 1 whenever the current state hits R_i or R'_j. The second-component of $r(i, j)$ evolves in such a way that it takes the value 2 infinitely often iff the run meets both G_i and G'_j infinitely often.

Now, the acceptance condition $\mathcal{F}'$ has, for every $i \in \{1, \ldots, s\}$, $j \in \{1, \ldots, t\}$, the pair $(P \times Q \times W, P \times Q \times W')$ where W is the set of all functions r where the first component of $r(i, j)$ is 1 and W' is the set of all functions r where the second component of $r(i, j)$ is 2. A run satisfies such a pair iff it meets both G_i and G'_j infinitely often and meets R_i and R'_j only finitely often.

$\mathcal{F}'$ also contains, for every $(R, G) \in \mathcal{F}$, the pair $(R \cup (P \times Q), G)$. These pairs accept those runs that eventually settle in the state-space Q and meet the acceptance condition of $\mathcal{F}$. □

Theorem 6.4 *The control problem for doubly-flanked pipelines is decidable.*

Proof Let the pipeline have k sites, as in Figure 6.2. Starting with the left-site, use Lemma 6.4 to construct an alternating automaton A_1 that accepts a t_1-type tree T_1 iff s_1 can successfully generate a sublanguage of $Lang(T_1)$. Convert A_1 into a nondeterministic automaton $\widehat{A}_1$. Invoking Lemma 6.6 for s_2, with $\widehat{A}_1$ as the automaton accepting t_1-type trees, we can construct an automaton A_2 which accepts a t_2-type tree T_2 iff there is a tree T_1 which $\widehat{A}_1$ accepts and there is a local strategy which wins on $Lang(T_1)$ and generates a sublanguage of $Lang(T_2)$. Arguing in a manner similar to the one in Theorem 6.3, we can show that A_2 accepts a tree T_2 iff there is a strategy f_1 at s_1 and a strategy f_2 at s_2 which work on all possible inputs on x_1, win locally, and generate a sublanguage of $Lang(T_2)$.

Invoking Lemma 6.6 repeatedly, we can walk down the pipeline till we have an automaton A_{k-1} which accepts a t_{k-1}-type tree T_{k-1} iff there are strategies at sites $s_1, \ldots, s_{k-1}$ which win locally and produce a sublanguage of $Lang(T_{k-1})$ on t_{k-1}. Now using Lemma 6.5, construct an automaton A_k which accepts t_{k-1}-type trees which s_k can win on.

We can now show that $\mathcal{L}(A_{k-1}) \cap \mathcal{L}(A_k) \neq \emptyset$ iff there is a distributed winning strategy for the plant. Further, if there is a winning strategy, we can using the runs on regular trees, walk back along the pipeline and synthesize winning strategies which are represented by finite-state transition systems. This will then correspond to a finite-state distributed controller. □

We note that sub-architectures of doubly-flanked pipelines are either doubly-flanked pipelines or singly-flanked pipelines. Lemma 6.5 can be easily modified to handle such a site. Hence we have:

Theorem 6.5 *Let $\mathcal{A}$ be any architecture such that all connected components of $\mathcal{A}$ are sub-architectures of doubly-flanked pipelines. Then the control problem for $\mathcal{A}$ is decidable.*

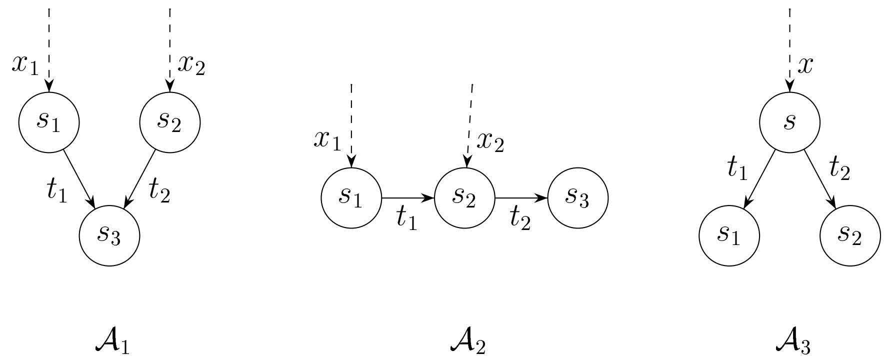


Figure 6.3: Basic undecidable architectures

6.5 Local specifications: Undecidable architectures

We show now that any architecture that is not a sub-architecture of a clean pipeline is undecidable. We show first the undecidability of three basic architectures shown in Figure 6.3.

The reductions will be from the halting problem for deterministic Turing machines starting with a blank tape. Our proofs are extensions of the undecidability proof developed in [PR90].

Let us first assume a standard notion of Turing machines, as say in [HU79]. A Turing machine working over a tape-alphabet Γ is a tuple $M = (Q, q_{in}, \rightarrow, q_h)$ where Q is a finite set of states, q_{in} and q_h belong to Q and are the initial are halting states, respectively, and $\rightarrow$: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function. $\rightarrow (q, a) = (q', b, d)$ is interpreted as meaning that if M is in state q and reads the symbol a at the current head position, then it rewrites the cell with the symbol b, the tape-head moves one cell to the right/left (depending on whether d is R or L) and M changes its state to q'.

A configuration of the deterministic Turing machine M is a sequence $C \in \Gamma^* \cdot Q \cdot \Gamma^+$ where Γ is the set of tape symbols and Q is the set of states. If $C = x.q.y$, with $q \in Q$, then the machine is in state q and has $x.y$ written on the tape with the head position on the cell after x. The initial configuration, $C_{in} = q_{in} \cdot \flat$ where q_{in} is the initial state and $\flat$ is the special tape symbol called blank. The transition relation $\vdash$ on configurations is defined in the obvious way. We say that the machine halts on the blank-tape if $C_{in} \vdash^* C_h$ where $C_h = q_h \cdot y$ with q_h being the designated halt state and $y \in \Gamma^+$. We assume that the tape-head of M never falls off the left-end of the tape.

A crucial mechanism used in the proofs will be the encoding of a program at a site so that it generates sequences of configurations when controlled by a strategy. Let us explain this first before going into the proofs. Let s be a site with an input channel z (z could be internal or external) and an output channel t. In order to have s generate configurations on the channel t, we equip the domain of values for channel t to include a suitable vocabulary to describe configurations. In particular, channel t will be able to take values in Γ and Q and also the special symbols \$ and $*$. The input channel z can take at least the two values S (which means "Start outputting a new configuration") and N (which means "output the Next symbol of the current configuration"). The site s will host a program P which will behave as follows. On receiving S, the program will output \$, and on being prompted repeatedly with a sequence of N's, will output a configuration (i.e. a word in $\Gamma^* \cdot Q \cdot \Gamma^+$). At the end of the configuration, it will output a \$ again and wait for the next S input. If at this point it gets N, then it simply outputs the symbol $*$. The first configuration output by the program is always C_{in}.

The program P will, of course, be finite state and will not encode the exact configurations generated. It will just be a transition system that allows, when fed an input S when it is not generating a configuration, to generate any word in $\Gamma^* \cdot Q \cdot \Gamma^+$. However, the program does enforce the condition that no matter how many N's are fed in the beginning, the first configuration output is the initial configuration. Note that the program cannot by itself force the configuration output to even be finite. What is important is that a controller strategy working on the program must be able to generate any sequence of configurations, when prompted by the channel z, as described above.

If something abnormal occurs (for example, if while in the middle of outputting a configuration, the program receives S as input), we can assume that it goes to a special state q_{win} where it is stuck and where, on any input, it outputs a special symbol *win* on channel t.

Note that a strategy working on the program could generate different sequences of configurations, depending on how long the delay was in between configurations. Thus the behaviour of the controlled program is best viewed as a tree of configurations.

Let us now prove the first undecidable architecture:

Lemma 6.7 *The control problem for the architecture $\mathcal{A}_1$ is undecidable.*

Proof Let us first note that there is a simple proof of this using the fact that the realizability/control problem for the two-site disconnected architecture $\mathcal{A}_\perp$ is undecidable for *global* specifications, a result proved in [PR90]. We can reduce an instance I of the control problem for $\mathcal{A}_\perp$ to an instance I' of the control problem on $\mathcal{A}_1$ by setting the programs at s_1 and s_2 to be the two programs assigned to the sites in I. We can now engineer the programs at s_1 and s_2 so that they send the current states they are in, along the internal channel, to s_3 at every move. The site s_3 now reads the global behaviour of the programs and we can suitably state a winning condition on s_3 so that the program at s_3 wins iff the global behaviour of s_1 and s_2 meets the global specification mentioned in I.

We however go through a longer proof here by incorporating the proof in [PR90] to our setting since understanding this will be a stepping-stone in proving the other undecidability results.

Given a Turing machine M we construct an instance of the control problem on $\mathcal{A}_1$ as follows. The sites s_1 and s_2 will host programs so that a controller working on them can generate configuration sequences of M, as described earlier. The channels x_1 and x_2 hence can carry values S and N and the sites s_1 and s_2 will produce configuration sequences on t_1 and t_2 respectively.

The site s_3 will process the configurations sent by s_2 and s_3 as follows. Suppose, starting from the same time instant, s_1 starts sending C and s_2 starts sending C'. If s_1 and s_2 are both outputting the i^{th} configuration for some i, then s_3 checks whether $C = C'$. If not, it goes to a state from where it cannot win. If s_2 leads s_1 by exactly one configuration, then s_3 checks whether $C \vdash C'$. If not, it again goes to a losing state. The behaviour for the case when s_1 leads s_2 by one configuration is analogous. If it so happens that at a point s_1 or s_2 leads the other by more than one configuration, then s_3 starts to ignore its inputs and goes to a state from which it locally wins. Note that the program at s_3 does not need to count the exact number of configurations it has seen, but just maintain whether the configurations are proceeding together, or if not, whether one of them leads the other by precisely one configuration (and if so, which). The crucial fact is that this can be achieved by s_3 with a bounded amount of memory.

If s_3 receives *win* from either of the other two sites, it goes to a state where it always wins. Finally, s_3 will check whether any of the sites output the halting configuration — if they do, it enters a winning state.

The site s_1 and s_2 are controllable while s_3 is not (i.e. the program at s_3 is deterministic). The local winning condition will be trivial for s_1 and s_2 while in s_3 it will demand that it wins if it enters the winning states mentioned above (e.g. when the configurations move more than one configuration apart or when the site s_3 reads a halting configuration from s_1 or s_2).

Let us now look closely at how a pair of strategies at s_1 and s_2 can hope to meet the local specification of s_3. We claim that for them to win, they must both output the proper configuration sequence which the Turing machine goes through starting from the initial configuration. In other words, we claim that if a pair of strategies doesn't do this, it will definitely not meet the specification (we are not saying yet anything about what happens when they do output the correct configuration sequences).

For assume that a pair of strategies at s_1 and s_2 do not output the proper configuration sequence. Then, the environment can suitably schedule the outputs of the configurations to make the controller lose. For example, let the first site to go off the correct configuration sequence be s_1. That is, there is a sequence of S's and N's of length i such that after the i'th input, s_1 outputs the first symbol that is a wrong configuration while for *all* possible sequences of S and N of length i, the configuration sequences output by s_2 all conform to M.

Let s_1 on that particular sequence generate a configuration sequence $C_1, C_2, \ldots C_n, C'$ where $C_1, \ldots, C_n$ is the correct run of the machine and $C_n \not\vdash C'$. The environment can now let s_1 run one configuration ahead of s_2. Then, at some point, s_3 will read C_n from s_2 and C' from s_1 simultaneously. The check $C_n \vdash C'$ will fail, and s_3 will go to a state from where the controller cannot win.

Now, if the strategies do indeed play the proper sequences of M, then in order to win, s_3 must see the halting configuration (in the scenario where the the environment forces the configurations to proceed together, say) It follows then that there is a distributed controller for this plant iff the Turing machine halts. $\square$

Note that in the reduction above, if there was a controller for the plant that meets the specification, then M halts and one can in fact build a finite-state controller for the programs at s_1 and s_2. Hence it follows that M halts iff there is a finite-state controller for the plant that meets the specification; this shows that the control problem is undecidable for $\mathcal{A}_1$ even if we are seeking only finite-state controllers.

Let us now move on to the architecture $\mathcal{A}_2$. Here, the difference from the previous setting is that we no longer have a site that can globally observe the plant while maintaining that the other two sites can communicate "secretly" to it. (If we choose s_3 to be the global observer, then though s_1 and s_2 can pass on information to s_3, s_2 will always be able to read the message from s_1 to s_3, and change its behaviour accordingly).

In order to get around this, we will use s_1 and s_3 as the independent agents (analogous to s_1 and s_2 in $\mathcal{A}_1$) which will generate configuration sequences, while s_2 will be the one that checks these sequences. However, note that s_3 cannot communicate to s_2.

Let us now introduce a mechanism whereby a site can "accept" sequences of configurations rather than generate them. Let s be a site with no output channel and a single internal input channel t. As usual, channel t will carry values S and N in order to prompt s to generate sequences. However, when s starts a configuration, it will generate it one unit time in advance and keep the generated symbol of $\Gamma \cup Q$ in its state-space. It can then proceed from this state only if the input it receives on t is the same as the symbol it has committed to. For example, at a particular point, on prompting, let us say that s commits that the *next* symbol it will generate is a. Then the program at s moves to a state of the form (q, a). At the next instant, s can move from (q, a) only if it receives a as input on t. It then proceeds to commit the next symbol. If the expected signal is not read, then s will go to a state where it loses. This mechanism can be viewed as a way so that a strategy can fix a tree of configurations which a site "generates" — and the program will win only if the sequences generated on t conform to these sequences.

Lemma 6.8 *The control problem for the architecture $\mathcal{A}_2$ is undecidable.*

Proof Site s_1 will output configurations on t_1 when prompted by the environment on the channel x_1. Site s_3 will, when prompted by s_2 on t_2, "accept" configurations.

Site s_2 can go into two modes, A and B, the decision being taken according to the first environment input on x_2. In mode A, the program at s_2 simply passes the configurations which it receives on t_1 to t_2. In Mode B, the program first outputs the initial configuration to s_3 and after that, each time it receives a configuration C on t, it propagates online C' to s_3 where $C \vdash C'$. (Note that a finite transition system can indeed generate C' from C online, with a constant delay, say three time units).

Recall that if s_3 receives a symbol it has not committed to, it goes to a reject state. Mode A ensures that the two sites output/accept the same configuration sequences while Mode B ensures that if the i^{th} configuration output by s_1 is C and the $(i+1)^{th}$ configuration accepted by s_2 is C', then $C \vdash C'$. So the only way the controller can hope to win is by s_1 and s_3 accepting the configuration sequence of M. By introducing a winning condition on s_2 which makes sure that s_2 locally wins only if it outputs the halting configuration, one can show that the plant has a distributed controller iff M halts on the blank tape. $\square$

Lemma 6.9 *The control problem for the architecture $\mathcal{A}_3$ is undecidable.*

Proof As done by s_3 of $\mathcal{A}_2$ in the previous lemma, s_1 and s_2 will now accept configurations of M. Site s can be in two modes, A and B, the mode chosen by the first input on x. In Mode A, the program at s_1 passes the initial configuration C_{in} to s_1 and makes s_2 wait. Then, while getting as input an *arbitrary* configuration C from the environment on x, it passes C to s_2 and simultaneously passes C' to s_1 where $C \vdash C'$. Mode B is analogous with the roles of s_1 and s_2 interchanged.

Sites s_1 and s_2 accept configuration sequences and when they get an input which is not what they have committed to accept, they go to a stuck state *stk*. If they are not in this state, we say they are *unstuck*.

Now, assume that at least one of the sites s_1 and s_2 doesn't accept the correct configuration — say s_2 is the one which accepts the smallest wrong sequence. Then the environment can force s_1 to be unstuck and get s_2 stuck by playing in mode B and sending the proper sequence of configurations of M to s_1 and s_2. If s_1 was the one that accepted the smallest wrong sequence, then the environment can get s_1 stuck while keeping s_2 unstuck, by playing in mode A. Note, however, that no matter how the controller plays, the environment *can* force both s_1 and s_2 to get stuck by feeding a completely unrelated configuration to s. Also, if programs at s_1 and s_2 accept the correct run of M, then there is no way for the environment *to get the site scheduled earlier to be unstuck and the site scheduled later to get stuck*, i.e.

it is impossible to go into mode A and keep s_1 unstuck and get s_2 stuck or go into mode B and get s_1 stuck and keep s_2 unstuck.

Hence, to force s_1 and s_2 to accept the correct configuration sequence of M, we would like the environment to win iff it can get the site scheduled first to be unstuck and get the other stuck. This kind of condition, however, is not realizable as local winning conditions on s_1 and s_2. The trick now is to have another mode C for s where the controller is forced to emulate the combined (product) behaviour of s_1 and s_2.

In mode C, the program will enter a zone where the state-space is the product state-spaces of the programs at s_1 and s_2. This zone of the program will in fact be controllable and we would like to enforce the condition that a winning strategy for s must control it in a manner such that it exactly mimics the (combined) behaviour of s_1 and s_2. Just after entering mode C, the program will, depending on the next input on x, decide whether it enters the submode A or B. Hence it goes to a mode C_A or C_B. In mode C_A, it virtually passes the initial configuration to its first component (that corresponding to s_1) and makes the second component wait. The strategy for s at this point has to decide how the component states will evolve. (Note that this strategy has access to a lot of information — the mode of the state, the exact product state, etc. Its choices need not be "independently" made for the two components). When the components evolve, they commit to certain symbols and s is forced, by the structure of the program, to send these values along the channels t_1 and t_2. The crucial point is that the sites s_1 and s_2 are oblivious to these modes and have to behave the same way on all modes.

At any point in this interaction, the environment can choose to send a special symbol *check* to s. When s in mode C receives this, it immediately sends the value of the current local states of the programs of s_1 and s_2 along the channels t_1 and t_2. The programs of s_1 and s_2 are augmented in such a way that on receiving a state of the program, they go to a state that is winning if it is the same state they are in; otherwise they go to a losing state.

Now, if a strategy at s doesn't mimic the exact behaviour of the strategies at s_1 and s_2, then it is easy to see that the environment can play in mode C and make the strategy lose at s_1 or s_2.

The winning condition can now be stated on the state-space of s in the zone corresponding to mode C, by allowing a behaviour to be winning for the environment only if it reaches a state where the program is in mode C_A, the component of s_1 is unstuck the component for s_2 is stuck, or, the program is in mode C_B, the component of s_2 is unstuck the component for s_1 is stuck. One can make make sure that one of the sites, say s_1, wins when it accepts the halting configuration. One can show now that a distributed controller exists iff M halts on the blank tape. $\square$

Though we have proved the undecidability result only for three architectures, they show the undecidability of all other architectures as well. Using Lemma 6.7 we can show that any architecture $\mathcal{A}'$ which has a site s with two internal channels is undecidable. Let s' be such a site with s''_1 and s''_2 such that there are internal channels from s''_1 and s''_2 to s'. The idea is to pick a minimal site s'_1 above s''_1 and a minimal site s'_2 above s''_2. One can reduce the control problem for $\mathcal{A}_1$ to such an architecture by setting the programs at sites s_3, s_1 and s_2 in $\mathcal{A}_1$ to be the programs at s', s'_1 and s'_2 in $\mathcal{A}'$, respectively. We can make the rest of the sites "dummy" by making them just pass their input to their output and always win locally.

If there is only one minimal site $\tilde{s}$ above s''_1 and it is the only minimal site above s''_2 as well, then it must be the case that there is a path from $\tilde{s}$ to a site $\tilde{s}'$ such that $\tilde{s}'$ has two internal output channels to sites s'_1 and s'_2 that are above s''_1 and s''_2, respectively. We can now set the programs of s_1 and s_2 in $\mathcal{A}_1$ to be the programs at s''_1 and s''_2, set the program at s_3 in $\mathcal{A}_1$ to be the program at site s', make $\tilde{s}$ take the inputs for both sites s''_1 and s''_2 and propagate it all the way to $\tilde{s}'$; the program at $\tilde{s}'$ will simply separate the inputs and feed them to s'_1 and s'_2 and the programs at s'_1 and s'_2 will propagate these inputs to s''_1 and s''_2 respectively.

Similarly, using Lemma 6.9 we can show that any architecture which has a site with two internal output channels is undecidable.

What we are left with are pipelines. Since we require each process to have an input channel, the left-site of the pipeline must have an external input channel. Let us have a pipeline with sites $\{s'_1, \ldots, s'_k\}$, with s'_1 having an external input channel and with internal channels from s'_i to s'_{i+1} for each $0 < i < k$. If $k = 2$ or if s'_k is the only other site which has an external input, then it is a sub-architecture of a doubly-flanked pipeline (and hence the control problem is decidable). So let $k > 2$ and let s'_i have an external input channel, where $1 < i < k$. Then, by making the sites s'_j "dummy", where $1 < j < i$ or $i < j < k$, we can reduce the control problem for $\mathcal{A}_2$ to the control problem for this pipeline, by coding the program at s_1 into s'_1, the program at s_2 into s'_i and the program at s_3 into s'_k. Hence we have:

Theorem 6.6 *If $\mathcal{A}$ is an architecture which has a connected component which is not a sub-architecture of a doubly-flanked pipeline, then the control problem for $\mathcal{A}$ is undecidable.* □

All the architectures shown to be undecidable above can be shown to be undecidable even if we are looking for only finite-state controllers — this is so because in all the reductions above, if there was a controller, then there was always a finite-state one. The results above can be also suitably changed to show that even for weaker winning conditions such as Büchi, co-Büchi, or even safety conditions, the architectures remain undecidable.

The proof of Lemma 6.9 shows a method to convert certain global specifications to local ones. We can use this technique to prove a lower bound on the complexity of the control problem on the decidable architectures shown in Section 6.4.

Note the our decision procedure for the doubly-flanked pipeline works in time that is non-elementary in the number of sites in the pipeline. This is because the operation of converting alternating automata to nondeterministic automata when we walk down the pipeline (see Theorem 6.4) blows up the state-space of the automaton by one exponential each time. In the setting of global specifications too, the complexity of decidability of singly-flanked pipelines is non-elementary in the number of sites, as shown in [PR90]. Pnueli and Rosner show, using results in [PR79], that this non-elementary complexity is unavoidable as well.

Let us consider singly-flanked pipelines. Using the fact that for solving realizability for these pipelines against global specifications needs time non-elementary in the number of sites, we can show that this lower bound extends to control-synthesis for local specifications on these architectures as well. First, it is easy to see that the realizability problem for a pipeline reduces to that of control on the same architecture and hence one needs time non-elementary in the number of sites for the control problem against global specifications.

Now, we can reduce a control problem on these pipelines against global specifications to a control problem on these pipelines against local specifications. Consider a control problem on a k-site ($\{s_1, \ldots, s_k\}$) singly-flanked pipeline with a global specification. We reduce this to a control problem on the $(k+1)$-site ($\{s'_1, \ldots, s'_{k+1}\}$) singly-flanked pipeline with local specifications. The program at s'_i is inherited from the program at site s_{i-1}, for each $i \in \{2, \ldots, k+1\}$. Site s_1 will behave in two modes A and C; the mode it goes into will be decided by the first environment input. In mode A, the input it gets is meant for s'_2 (the external input to s_1 in the problem we started with) and hence it passes it on to the site s'_2.

In mode C, it behaves like the program at s in the proof of Lemma 6.9: it enters a zone where it emulates the product behaviour of the programs at sites $s_1, \ldots, s_k$. At any point, if it gets a special symbol *check* from the environment, it sends its current state along the pipeline. The programs at sites $s'_2, \ldots, s'_k$ are modified so that when they receive such a global state, they go to a state where the system loses if the state doesn't match the state they are in. If it does match, they move to a state where the system wins and pass the state to the next site in the pipeline.

Using arguments similar to that of Lemma 6.9, one can show that for any winning strategy, the local strategy at s'_1 must emulate the strategies at $s'_2, \ldots, s'_{k+1}$ when s'_1 is in mode C. One can now encode the global specification for $s_1, \ldots, s_k$ as a local specification on s'_1 and show that the reduction preserves the property of existence of a controller.

Though this reduction is exponential in the number of sites, it shows that the complexity of the control problem for singly-flanked architectures is non-elementary in the number of sites. The same lower bound follows for doubly-flanked architectures as one can engineer the program at the last site to ignore its external input. Finally, it is easy to see that for any decidable architecture, the control problem must take time non-elementary in the maximum number of sites in a connected component of the architecture.

6.6 Conclusions

In this chapter, we have studied the problem of control-synthesis in a distributed but synchronous setting for local specifications and characterized the exact class of architectures for which the problem is decidable.

We could extend our study of distributed controllers for architectures that are not acyclic — for example, rings. In a recent paper [KV01], the control problem for the singly-flanked pipeline with other extra internal channels thrown in, has been studied for global specifications, and shown to be decidable. However, this is not surprising as one can show (as observed in [KV01] as well) that the extra internal channels cannot change the answer to the problem — there is a controller on such an architecture iff there is one for the singly-flanked pipeline corresponding to it. This is because the sites have strictly decreasing information of the global state of the system and a site knows completely the configurations of the programs at all sites down the pipeline. Hence adding internal channels that go backward on the pipeline can carry no useful information. (Adding extra internal edges that go forward will make them undecidable, even for local specifications, as we have shown.)

One can also study the control problem for local specifications for architectures that have cycles. However, one can show, using our results, that even here most architectures are undecidable. An important architecture for which the problem is still open is the two-site ring (i.e. the architecture where there are two sites, both having external input channels and having a channel both ways between them). For rings with more than two sites (where at least two sites have external input channels), one can show that the problem is undecidable.

Another direction to explore would be to try and extend the local specifications to more powerful specifications that respect the connectivity in the architecture but for which the control-problem remains decidable (for the class of decidable architectures for local specifications). One such mechanism would be to have a set of tuples of local winning conditions — a behaviour of the plant satisfies this specification if there is *some* tuple in the set for which the local winning conditions are met. There are logics like Product-LTL [Thi94], for example,, that offer a localized temporal logic that can be encoded in this manner.

A natural extension of our work would be to consider local *branching-time* winning conditions instead of linear-time winning conditions. Again, in [KV01], Kupferman and Vardi extend the decidability results of [PR90] for singly-flanked architectures to global branching-time specifications. As for local specifications, the results for doubly-flanked pipelines don't seem to extend to the branching-time setting. The crucial difficulty lies in Lemma 6.4 where it seems to be hard to accept exact languages rather than sublanguages. Note that in Theorem 6.3, we crucially use the fact that the specification is linear-time. We conjecture that the control problem for branching-time logics (say for $\mathsf{CTL}^\star$ adapted in this setting) would be undecidable for doubly-flanked pipelines.

Chapter 7

Conclusions

In the midst of the word he was trying to say,
In the midst of his laughter and glee,
He had softly and suddenly vanished away—
For the Snark was a *Boojum, you see.*

— *The Hunting of the Snark*, Lewis Caroll

In summary, we have shown the following results in this thesis:

- The control synthesis problem was studied for simulations and bisimulations and it was shown that one can solve this in polynomial time. Moreover, whenever a controller exists, one can synthesize a controller of polynomial size within the same time-bounds.

- The control synthesis and model-checking problems for asynchronous simulations is undecidable. The undecidability extends even to very simple classes of concurrent systems.

- The control and realizability problems for the branching-time temporal logics were studied. For universal environments, these problems reduce to module-checking and hence are, for CTL and CTL*, EXPTIME-complete and 2-EXPTIME-complete, respectively [KV96]. The complexity of these problems in reactive environments become exponentially harder — they are 2-EXPTIME-complete for CTL and 3-EXPTIME-complete for CTL*.

- We also investigated the control-synthesis problem in a distributed setting, where the processes communicate with each other in a synchronous fashion and also interact with their local environments according to an architecture. From the results of [PR90], it follows that for global specifications, the only decidable architecture is the singly-flanked pipeline. We studied the problem for local specifications and showed that the class of decidable architectures (mildly) increases. The control problem for an architecture for local specifications is decidable iff each connected component of it is a sub-architecture of a doubly-flanked pipeline.

Future directions

Apart from the various open problems and directions mentioned in the concluding sections of the earlier chapters, a theme that universally begs attention is that of *partial observation*. In many settings, the controller cannot observe all the moves of the plant and has to give its advice based on only the partial information it has about it [KS95, KS97]. A related technically similar work is that concerning realizability under incomplete information studied in various papers by Kupferman and Vardi [KV97b, KV99a]. The control-synthesis problem for simulations, bisimulations, branching-time logics and distributed systems, in the setting of partial observation, needs to be explored.

Another aspect of control usually studied is that of *performance evaluation* of controllers. In settings involving simple state-based internal specifications, one can define the notion of a minimally restrictive controller and synthesize it [CL99]. However, in many cases involving liveness conditions, as in temporal logics, such a naive way of defining minimally restrictive controllers is useless as they almost never exist. However, one does require a fair notion of when a controller is "better" than another such that it is possible to synthesize the "best" controller.

The results on simulations and bisimulations of Chapter 2 are conspicuous in that they are not solved using automata-theoretic methods. Of course, one could use tree-automata here as well — but, a naive way of using them, would not give us polynomial-time decidability. The most natural way to code the problem into an automaton would involve the automaton guessing the moves enabled and how they will be simulated and this itself would cause an exponential blow-up. Of course, one just needs to define a class of automata which are suited for handling simulations and bisimulations and use the efficient mechanisms employed in Chapter 2 to get optimal results. We feel that this requires further study and a definition of tree automata which, *by their very structure*, accept only bisimilar sets of trees and followed by a formal study of them, will be rewarding.

The results in Chapter 6 also suggest many extensions. The study of distributed controllers for subclasses of branching-time specifications would be interesting. A more interesting question is whether one can restrict the control problem in a different way (say by reducing the power of the controller's memory) in order to get decidability across a more generous class of architectures.

A serious drawback of our distributed-control model is that the communication is synchronous. In most practical situations (say in protocols, etc.), the processes cannot hope to communicate in this manner but only by message-passing along channels, where messages can take arbitrary time to reach their destination. Our setting is only distributed in nature and it is important to bring concurrency into play as well, and try to synthesize controllers in such a setting. A possible place to start is the work by Pnueli and Rosner on the synthesis of asynchronous reactive modules [PR89b].

The close link between synthesis and games extends to the distributed setting, where we can consider designing a distributed controller as finding strategies in multi-party cooperative games studied in [PR79, APR91]. The authors in these works show that "hierarchical games", where the information flow proceeds in one direction, is decidable. Our results show that there can exist settings where there is no such hierarchy, where two players can have incomplete information about each other, and yet the problem of finding winning strategies is decidable. It would be interesting to study in more generality the games that correspond to the control problem in our setting, and find the real reason why such games turn out to be decidable. Proving our results in the more general framework of multi-player games will give a better understanding of the issues involved and may find applications in other areas as well.

Yet another general extension would be to study control-synthesis for systems where the behaviours encode not only the order in which events occur, but also at the exact *times* at which they occur as well. There are fairly robust mechanisms in computer science such as timed-automata [AD94] using which one can model and analyze timed-systems. The problem of control synthesis for such systems against linear-time specifications has been undertaken in recent works [AMP95, AMPS98, HW91]. Extension of these results to the the setting where specifications are external or where the setting is distributed, would be rewarding in terms of theoretical understanding as well as practical use.

Reaching out to handle more continuous forms of behaviour can be stretched even further — for example, in the control of hybrid systems, say using the model of hybrid automata [Hen96]. However, these models, though interesting and extremely useful, seem hard at present to analyze, let alone achieve automated control.

On the practical side, considerable effort is needed in terms of building tools and heuristics to do control-synthesis. There has been very little implementation of programs for automated synthesis of controllers and this situation needs to improve. The high-complexity of control-synthesis should not deter trying big examples, as it is not clear how the complexity will play out in practice. The complexity is also usually high only in terms of the size of the specification, and not the plant itself, and hence might be practically feasible. We hope that the work presented in this thesis will one day find uses in practical applications in industry.

Appendix

Undecidability of simulation for products of systems

Here we will show how to realize the plants and specifications given in Chapter 3 as a restricted class of asynchronous transition systems — those which can be described as synchronized products of ordinary transition systems.

A Σ-*labeled deterministic synchronized product system* is a structure $(\{P_i\}_{i=1}^n, \varphi)$ which consists a set of deterministic transition systems (processes) $P_i = (Q_i, E_i, T_i, q_{in}^i)$. The P_i's are supposed to represent sequential processes which work concurrently and independently while synchronizing on common events. φ is a labeling function $\varphi : \bigcup E_i \to \Sigma$. The asynchronous transition system which captures the behaviours of such a system is defined as the following "global" system $TS = (Q, E, T, q_{in}, \varphi, I)$ where:

- $Q = Q_1 \times Q_2 \times \ldots Q_n$
- $E = \bigcup E_i$
- $q_{in} = (q_{in}^1, \ldots, q_{in}^n)$
- $(q_1, \ldots, q_n) \xrightarrow{e} (q'_1, \ldots q'_n)$ iff
  $\forall i : e \in E_i \Rightarrow (q_i \xrightarrow{e} q'_i)$ is in P_i and
  $\forall i : e \notin E_i \Rightarrow q_i = q'_i$
- $e_1 \ I \ e_2$ iff $\{i \mid e_1 \in E_i\} \cap \{j \mid e_2 \in E_j\} = \emptyset$

It is easy to see that the system defined above is indeed an asynchronous transition system.

The construction of the plant TS_p.

TS_p can be realised as a product of the following processes:

- A process $R = (\{R_0, R_1, R_2\}, \{r_0, r_1, r_2\}, T_R, R_0)$ where T_r has the transitions $R_0 \xrightarrow{r_0} R_1 \xrightarrow{r_1} R_2 \xrightarrow{r_2} R_0$

- A process $U = (\{U_0, U_1, U_2\}, \{u_0, u_1, u_2\}, U_R, U_0)$ where U_r has the transitions $U_0 \xrightarrow{u_0} U_1 \xrightarrow{u_1} U_2 \xrightarrow{u_2} U_0$

- For every $i, j \in \{0, 1, 2\}$ we have a process $R_{ij} = (\{q_1, q_2, q_3\}, \{ij, r_{i+1}, r_{i-1}\}, T, q_{in})$ where T has the transitions:
  $q_1 \xrightarrow{r_{i+1}} q_2 \xrightarrow{r_{i-1}} q_1 \quad q_1 \xrightarrow{r_{i-1}} q_1 \quad q_2 \xrightarrow{r_{i+1}} q_2 \quad q_1 \xrightarrow{ij} q_3$

  $q_{in} = q_1$ if $i = 0$ and $q_{in} = q_2$ if $i \neq 0$

- For every $i, j \in \{0, 1, 2\}$ we have a process $U_{ij} = (\{q_1, q_2, q_3\}, \{ij, u_{j+1}, u_{j-1}\}, T, q_{in})$ where T has the transitions:
  $q_1 \xrightarrow{u_{j+1}} q_2 \xrightarrow{u_{j-1}} q_1 \quad q_1 \xrightarrow{u_{j-1}} q_1 \quad q_2 \xrightarrow{u_{j+1}} q_2 \quad q_1 \xrightarrow{ij} q_3$

  $q_{in} = q_1$ if $j = 0$ and $q_{in} = q_2$ if $j \neq 0$

- For every $i, j, i', j' \in \{0, 1, 2\}$ such that ij and $i'j'$ are distinct events and it is not the case that $ij \ I \ i'j'$ (as defined in the construction), we have a process $(\{q_1, q_2, q_3\}, \{ij, i'j'\}, T, q_1)$ where T has the transitions: $q_1 \xrightarrow{ij} q_2$ and $q_1 \xrightarrow{i'j'} q_3$

The construction of the specification TS_s.

TS_s can be realised as a product of the following processes:

- The same processes R and U as in the definition of TS_p

- For every (c, ij)-event in TS_s, we have a process $R_{(c,ij)} = (\{q_1, q_2, q_3\}, \{(c, ij), r_{i+1}, r_{i-1}\}, T, q_{in})$ where T has the transitions:
  $q_1 \xrightarrow{r_{i+1}} q_2 \xrightarrow{r_{i-1}} q_1 \quad q_1 \xrightarrow{r_{i-1}} q_1 \quad q_2 \xrightarrow{r_{i+1}} q_2 \quad q_1 \xrightarrow{(c,ij)} q_3$

  $q_{in} = q_1$ if $(i = 0$ and $c = c_{in})$
  $q_{in} = q_2$ if $(i \neq 0$ or $c \neq c_{in})$

- For every (c, ij)-event in TS_s, we have a process
  $U_{(c,ij)} = (\{q_1, q_2, q_3\}, \{(c, ij), u_{j+1}, u_{j-1}\}, T, q_{in})$ where T has the transitions:
  $q_1 \xrightarrow{u_{j+1}} q_2 \xrightarrow{u_{j-1}} q_1 \qquad q_1 \xrightarrow{u_{j-1}} q_1 \qquad q_2 \xrightarrow{u_{j+1}} q_2 \qquad q_1 \xrightarrow{(c,ij)} q_3$

  $q_{in} = q_1$ if ($j = 0$ and $c = c_{in}$)
  $q_{in} = q_2$ if ($j \neq 0$ or $c \neq c_{in}$)

- For every pair of distinct events (c, ij) and $(c', i'j')$ in TS_s such that it is not the case that $(c, ij)\ I\ (c', i'j')$ (as defined in the construction), we have a process $(\{q_1, q_2, q_3\}, \{(c, ij), (c', i'j')\}, T, q_1)$ where T has the transitions:
  $q_1 \xrightarrow{(c,ij)} q_2$ and $q_1 \xrightarrow{(c',i'j')} q_3$

It is tedious but routine to verify that the product systems given above do generate the asynchronous transition systems we need.

Undecidability of controller synthesis for a restricted class

Here we consider asynchronous transition systems of the form $TS = (Q, E, T, q_{in}, \varphi, I, \widehat{I})$ where $TS = (Q, E, T, q_{in}, \varphi)$ is an asynchronous transition system and $\widehat{I} \subseteq \Sigma \times \Sigma$ is an irreflexive symmetric independence relation over Σ which satisfies the following property: $\forall e_1, e_2 \in E,\ e_1\ I\ e_2 \Rightarrow \varphi(e_1)\widehat{I}\varphi(e_2)$.

Note that any transition system $TS = (Q, E, T, q_{in}, \varphi, I)$ can be expressed as such a restricted transition system $TS' = (Q, E, T, q_{in}, \varphi, I, \widehat{I})$ by setting $\widehat{I} = \{(a, a') \mid a, a' \in \Sigma, a \neq a'\}$, provided that every two events e and e' which have the same label are dependent.

In the proof of undecidability of checking for simulations in Chapter 3, observe that any two events of the same label are indeed dependent. Hence simulation checking for the restricted class is undecidable as well.

Now we show that the controller synthesis problem is also undecidable for this class by reducing the simulation-checking problem to this problem.

Let $TS_p = (Q_p, E_p, T_p, q^p_{in}, \varphi_p, I_p, \widehat{I_p})$ and $TS_s = (Q_s, E_s, T_s, q^s_{in}, \varphi_s, I_s, \widehat{I_s})$ be two such systems. We will construct TS'_p and TS'_s such that there is a simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$ iff there is a controller for (TS'_p, TS'_s).

We first expand our alphabet. TS'_p and TS'_s will be Σ'-labelled transition systems where $\Sigma' = \Sigma_\star \cup \Sigma_1$, where $\Sigma_\star = \Sigma \cup \{\star\}$ and $\Sigma_1 = \{a' \mid a \in \Sigma_\star\}$. Thus, for every action a in $\Sigma_\star$ we have introduced a new action a'.

Assume without loss of generality that Σ' as well as $2^{\Sigma'}$ are disjoint from Q_p, Q_s, E_p and E_s. Then define $TS'_p = (Q'_p, E'_p, T'_p, q^{p'}_{in}, \varphi'_p, I'_p, \widehat{I'_p})$ as follows:

- $Q'_p = Q_p \cup \{q_a, q_{a'}, q_{a,a'} \mid a \in \Sigma\} \cup \{X \mid X$ is a nonempty subset of $\Sigma_1\} \cup \{q_\star, q_{\star\star}\}$
- $E'_p = E_p \cup \Sigma' \cup \{\widetilde{a}' \mid a \in \Sigma_\star\} \cup \{\star''\}$
- $\varphi'_p(e) = \varphi_p(e)$ if $e \in E_p$; $\varphi'_p(e) = a$ if $e = a \in \Sigma'$; $\varphi'_p(e) = a'$ if $e = \widetilde{a}'$; $\varphi'_p(\star'') = \star$.
- $q^{p'}_{in} = q^{p}_{in}$
- $$\begin{aligned} T'_p = \; & T_p \cup \\ & \{(q_1, a, q_a), (q_1, \widetilde{a}', q_{a'}), (q_a, \widetilde{a}', q_{a,a'}), (q_{a'}, a, q_{a,a'}) \mid q_1 \in Q_p, a \in \Sigma_\star\} \cup \\ & \{(q_1, a', \{a'\}) \mid q_1 \in Q_p \text{ and } a \in \Sigma_\star\} \cup \\ & \{(X, a', Y) \mid X, Y \text{ are non-empty subsets of } \Sigma_1 \text{ and } a' \notin X \text{ and} \\ & Y = X \cup \{a'\}\} \cup \\ & \{(q_1, \star'', q_\star) \mid q_1 \in Q_p\} \;\cup\; \{(q_\star, \star'', q_{\star\star})\} \end{aligned}$$
- $$\begin{aligned} I'_p = I_p \cup \; & \{(a, \widetilde{a}') \mid a \in \Sigma_\star\} \cup \\ & \{(a', b') \mid a \neq b \text{ and } a, b \in \Sigma_\star\}. \end{aligned}$$
- $\widehat{I'_p} = \widehat{I_p} \cup \{(a, a') \mid a \in \Sigma_\star\} \cup \{(a', b') \mid a \neq b \text{ and } a, b \in \Sigma_\star\}$

TS'_s is defined in a similar way. Note that the construction preserves the property required to stay within this class.

Again, using the basic properties of asynchronous controllers, we can prove that any controller for (TS'_p, TS'_s) must be the trivial one which allows all system moves at all times. We can use arguments similar to those in the proof of Theorem 3.2 to show that there is a simulation from $\mathcal{U}f(TS_p)$ to $\mathcal{U}f(TS_s)$ iff there is a controller for (TS'_p, TS'_s).

Publications

[MT98a] P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In *Davide Sangiorgi and Robert de Simone, editors, CONCUR'98, Concurrency Theory, 9th International Conference, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 18–33, Nice, France, September 1998. Springer-Verlag.

[KMTV00a] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc., CONCUR '00, Concurrency Theory, 11th Int. Conf.*, volume 1877 of *LNCS*, Penn. State Univ, USA, September 2000. Springer-Verlag.

[MT01a] P. Madhusudan and P. S. Thiagarajan. Branching time controllers for discrete event systems. *To appear in CONCUR'98 Special Issue, Theoretical Computer Science*, 2001.

[MT01b] P. Madhusudan and P. S. Thiagarajan. Distributed control and synthesis for local specifications. In *Proc., ICALP '01, 28th International Colloquium on Automata, Lang. and Programming*, volume 2076 of *LNCS*, Crete, Greece, July 2001.

Bibliography

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, July 1989.

[AM95] M. Antoniotti and B. Mishra. The supervisor synthesis problem for unrestricted CTL is NP-complete. Technical Report Technical Report TR1995-707, New York University, NY, USA, November 1995.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999, pages 1–20. Springer-Verlag, 1995.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[APR91] D. Salman Azhar, Gary L. Peterson, and John H. Reif. On multi-player non-cooperative games of incomplete information: Part 1 - decision algorithms. Technical Report TR CS1991 -37, Computer Science Department, Duke University, Durham, NC 27706, October 1991.

[Bed88] M. A. Bednarczyk. *Categories of Asynchronous transition systems.* PhD thesis, University of Sussex, 1988. Technical Report No. 1/88.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.

[BL97] G. Barrett and S. Lafortune. Using bisimulation to solve discrete event control problems. In *Proceedings of the American Control Conference*, pages 2337–2341, June 1997.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Chu63] A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pages 23–35. institut Mittag-Leffler, 1963.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.

[CL99] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[DTV99] M. Daniele, P. Traverso, and M.Y. Vardi. Strong cyclic planning revisited. In S. Biundo and M. Fox, editors, *5th European Conference on Planning*, pages 34–46, 1999.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 328–337, White Plains, October 1988.

[EK70] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In R. K. Guy, H. Hanani, N. Sauer, and J. Schonheim, editors, *Proceedings of the Calgary International Conference on Combinatorial Structures and their Applications*, pages 93–96. Gordon and Breach, New York, London, Paris, 1970.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pages 997–1072, 1990.

[Eme97] E. A. Emerson. Model checking and the mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, chapter 6. American Mathematical Society, 1997.

[ESW01] K. Etessami, R. Schuller, and T. Wilke. Fair simulation relations, parity games, and state space reduction for Büchi automata. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707, Crete, Greece, July 2001. Springer.

[Fit96] M. Fitting. *First Order Logic and Automated Theorem Proving.* Springer Verlag, New York, 2nd edition, 1996.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 163–173, January 1980.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[HKR97] T.A. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *Proc. 8th Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 273–287, Warsaw, July 1997. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[HW91] G. Hoffmann and H. Wong-Toi. The control of dense real-time discrete event systems. In *Conference on Decision and Control*, pages 1527–1528, Brighton, England, December 1991.

[JL91] B. Jonsson and K. G. Larsen. On the complexity of equation solving in process algebra. In *TAPSOFT*, volume 493 of *Lecture Notes in Computer Science*, pages 381–396. Springer-Verlag, 1991.

[JN00] M. Jurdziński and M. Nielsen. Hereditary history preserving bisimilarity is undecidable. In *Proc., 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2000)*, volume 1770 of *Lecture Notes in Computer Science*, Lille, France, 2000. Springer.

[JNW96] André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation from open maps. *Information and Computation*, 127(2):164–185, June 1996. A preliminary version appeared in *Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 418–427, Montreal, Canada, June 1993. IEEE Computer Society Press.

[KG95] R. Kumar and V.K. Garg. *Modeling and control of logical discrete event systems*. Kluwer Academic Publishers, 1995.

[KG96] O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996.

[KGM91] R. Kumar, V. Garg, and S. I. Marcus. On controllability and normality of discrete event dynamical systems. *System and Control Letters*, 17:157–168, 1991.

[KMTV00a] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc., CONCUR '00, Concurrency Theory, 11th Int. Conf.*, volume 1877 of *LNCS*, Penn. State Univ, USA, September 2000. Springer-Verlag.

[KMTV00b] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M. Vardi. Open systems in reactive environments: Control and synthesis. Technical Report Technical Report TCS-00-03, Chennai Mathematical Institute, Chennai, India, 2000. Available at `http://www.cmi.ac.in`.

[KS83] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes and three problems of equivalence. In *Proc. Second ACM Symposium on Principles of Distributed Computing*, pages 228–240, Montreal, Quebec, August 1983.

[KS95] R. Kumar and M.A. Shayman. Supervisory control of nondeterministic systems under partial observation and decentralization. *SIAM Journal of Control and Optimization*, 1995.

[KS97] Ratnesh Kumar and Mark A. Shayman. Centralized and decentralized supervisory control of nondeterministic systems under partial observation. *SIAM Journal on Control and Optimization*, 35(2):363–383, March 1997.

[Kup97] O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. *Journal of Logic and Computation*, 7:1–14, 1997.

[KV96] O. Kupferman and M.Y. Vardi. Module checking. In *Computer Aided Verification, Proc. 8th Int. Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 1996.

[KV97a] O. Kupferman and M.Y. Vardi. Module checking revisited. In *Computer Aided Verification, Proc. 9th Int. Conference*, volume 1254 of

Lecture Notes in Computer Science, pages 36–47. Springer-Verlag, 1997.

[KV97b] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997.

[KV99a] O. Kupferman and M.Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245 – 263, June 1999.

[KV99b] O. Kupferman and M.Y. Vardi. Robust satisfaction. In *Proc. 10th Conference on Concurrency Theory*, Lecture Notes in Computer Science. Springer-Verlag, August 1999.

[KV00] O. Kupferman and M. Vardi. μ-calculus synthesis. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[KV01] O. Kupferman and M. Vardi. Synthesizing distributed systems. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 16–19, Boston, Massachusetts, USA, June 2001. IEEE Computer Society.

[Lam80] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, New Jersey, U.S.A., 1981.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[LX90] K.G. Larsen and L. XinXin. Equation solving using modal transition systems. In *Proc. 5th Symposium on Logic in Computer Science*, pages 108–117, Philadelphia, June 1990.

[McN93] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65:149–184, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. International Lecture Series in Computer Science, Academic Press, London, 1981.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *Proc. 13th Int. Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1986.

[MT98a] P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In *Davide Sangiorgi and Robert de Simone, editors, CONCUR'98, Concurrency Theory, 9th International Conference, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 18–33, Nice, France, September 1998. Springer-Verlag.

[MT98b] P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. Technical Report TCS-98-02, Chennai Mathematical Institute, 1998. Available at `http://www.cmi.ac.in`.

[MT01a] P. Madhusudan and P. S. Thiagarajan. Branching time controllers for discrete event systems. *To appear in CONCUR'98 Special Issue, Theoretical Computer Science*, 2001.

[MT01b] P. Madhusudan and P. S. Thiagarajan. Distributed control and synthesis for local specifications. In *Proc., ICALP '01, 28th International Colloquium on Automata, Lang. and Programming*, volume 2076 of *LNCS*, Crete, Greece, July 2001.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[Ove94] A. Overkamp. Supervisory control for nondeterministic systems. In G. Cohen and J.-P. Quadrat, editors, *11th International Conference on Analysis and Optimization of Systems - Discrete Event Systems*, volume 199 of *Lecture Notes in Control and Information Sciences*, pages 56–65, London, 1994. Springer-Verlag.

[Ove97] A. Overkamp. Supervisory control using failure semantics and partial specifications. *IEEE Trans. on Automatic Control*, 42(4):498–510, 1997.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[Pnu85] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Proc. Advanced School on Current Trends in Concurrency*, pages 510–584, Berlin, 1985. Volume 224, LNCS, Springer-Verlag.

[PR79] G.L. Peterson and J.H. Reif. Multiple-person alternation. In *Proc. 20th IEEE Symposium on Foundation of Computer Science*, pages 348–363, 1979.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372, pages 652–671. Lecture Notes in Computer Science, Springer-Verlag, July 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symposium on Foundation of Computer Science*, pages 746–757, 1990.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Rab72] M.O. Rabin. Automata on infinite objects and Church's problem. *Amer. Mathematical Society*, 1972.

[Ram96] R. Ramanujam. Locally linear time temporal logic. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 118–127, New Brunswick, New Jersey, July 1996. IEEE Computer Society.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989.

[Saf88] S. Safra. On the complexity of ω-automata. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, White Plains, October 1988.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Thi94] P. S. Thiagarajan. A trace based extension of propositional linear time temporal logic. In *9th Annual IEEE Symposium on Logic in Computer Science*, pages 438–447, Paris, France, July 1994. IEEE Computer Society.

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of Theoretical Computer Science*, pages 165–191, 1990.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In E.W. Mayr and C. Puech, editors, *Proc. 12th Symp. on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1995.

[Tho97] W. Thomas. Languages, automata, and logic. *Handbook of Formal Language Theory*, III:389–455, 1997.

[VS85] M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, April 1986.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3. Oxford University Press, 1995.

[WW96] K. C. Wong and W. M. Wonham. Modular control and coordination of discrete-event systems. *Discrete Event Dynamic Systems*, 6(3):241–273, July 1996.

[Zie87] W. Zielonka. Notes on finite asynchronous automata. *RAIRO Informatique Théorique et Applications/Theoretical Informatics and Applications*, 21:99–135, 1987.

[Zie98] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.