A Uniform Approach to the Complexity and Analysis of Succinct Systems

Dissertation zur Erlangung des Grades des Doktors der
Naturwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der
Universität des Saarlandes

Hans-Jörg Peter

Saarbrücken, 2012

Tag des Kolloquiums	2. November 2012
Dekan	Prof. Dr. Mark Groves

Prüfungsausschuss

Vorsitzender	Prof. Dr. Christoph Weidenbach
Berichterstattende	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Jean-François Raskin, Ph.D.
	Oded Maler, Ph.D.
Akademischer Beisitzer	Björn Brandenburg, Ph.D.

To my parents Gertrud and Michael

Abstract

This thesis provides a unifying view on the *succinctness* of systems: the capability of a modeling formalism to describe the behavior of a system of exponential size using a polynomial syntax.

The key theoretical contribution is the introduction of *sequential circuit machines* as a new universal computation model that focuses on succinctness as the central aspect. The thesis demonstrates that many well-known modeling formalisms such as communicating state machines, linear-time temporal logic, or timed automata exhibit an immediate connection to this machine model. Once a (syntactic) connection is established, many complexity bounds for structurally restricted sequential circuit machines can be transferred to a certain formalism in a uniform manner. As a consequence, besides a far-reaching unification of independent lines of research, we are also able to provide matching complexity bounds for various analysis problems, whose complexities were not known so far. For example, we establish matching lower and upper bounds of the small witness problem and several variants of the bounded synthesis problem for timed automata, a particularly important succinct modeling formalism.

Also for timed automata, our complexity-theoretic analysis leads to the identification of tractable fragments of the timed synthesis problem under partial observability. Specifically, we identify timed controller synthesis based on discrete or template-based controllers to be equivalent to model checking. Based on this discovery, we develop a new model checking-based algorithm to efficiently find feasible template instantiations.

From a more practical perspective, this thesis also studies the preservation of succinctness in analysis algorithms using symbolic data structures. While efficient techniques exist for specific forms of succinctness considered in isolation, we present a general approach based on abstraction refinement to combine off-the-shelf symbolic data structures. In particular, for handling the combination of concurrency and quantitative timing behavior in networks of timed automata, we report on the tool Synthia which combines binary decision diagrams with difference bound matrices. In a comparison with the timed model checker Uppaal and the timed game solver Uppaal-Tiga running on standard benchmarks from the timed model checking and synthesis domain, respectively, the experimental results clearly demonstrate the effectiveness of our new approach.

Zusammenfassung

Diese Dissertation liefert eine vereinheitlichende Sicht auf die *Kompaktheit* von Systemen: die Fähigkeit eines Modellierungsformalismus, das Verhalten eines Systems exponentieller Größe mit polynomieller Syntax zu beschreiben.

Der wesentliche theoretische Beitrag ist die Einführung von *sequenziellen Schaltkreis-Maschinen* als neues universelles Berechnungsmodell, das sich auf den zentralen Aspekt der Kompaktheit konzentriert. Die Dissertation demonstriert, dass viele bekannte Modellierungsformalismen, wie z.B. kommunizierende Zustandsmaschinen, linear-Zeit temporale Logik (LTL) oder gezeitete Automaten eine direkte Verbindung zu diesem Maschinenmodell aufzeigen. Sobald eine (syntaktische) Verbindung hergestellt ist, können viele Komplexitätsschranken für strukturell beschränkte sequenzielle Schaltkreis-Maschinen für einen bestimmten Formalismus einheitlich übernommen werden. Neben einer weitreichenden Vereinheitlichung unabhängiger Forschungsrichtungen können auch zahlreiche Komplexitätsschranken für Analyse-Probleme etabliert werden, deren genaue Komplexität bisher noch nicht bekannt war. Zum Beispiel werden passende untere und obere Schranken des small witness Problems und mehrere Varianten des Synthese-Problems von Controllern mit beschränkter Größe für gezeitete Automaten bewiesen.

Die theoretische Analyse deckt Fragmente geringerer Komplexität des partiell informierten Syntheseproblems für gezeitete Automaten auf. Es wird im Besonderen gezeigt, dass das gezeitete Syntheseproblem für diskrete oder Vorlagen-basierte Controller äquivalent zum Model Checking-Problem ist. Basierend auf dieser Einsicht wird ein neuartiger Model Checking-basierter Algorithmus zur effizienten Synthese von gültigen Instantiierungen von Vorlagen entwickelt.

Der praktische Beitrag der Dissertation untersucht die Erhaltung von Kompaktheit in Analyse-Algorithmen durch die Benutzung symbolischer Datenstrukturen. Es wird ein allgemeiner Ansatz zur Kombination von Standard-Datenstrukturen vorgestellt, die jeweils bisher nur in Isolation verwendet werden konnten. Insbesondere wird für die Analyse von Netzwerken von gezeiteten Automaten das Tool Synthia vorgestellt, welches binäre Entscheidungs-Diagramme mit Differenzen-Matrizen verbindet. In einem experimentellen Vergleich mit den Tools Uppaal und Uppaal-Tiga wird klar die Effektivität des neuen Ansatzes belegt.

Acknowledgments

First of all, I want to express my deepest gratitude to my supervisor Bernd Finkbeiner. As a mentor, his ongoing encouragement was the driving force behind my research. As a colleague, his dedication to professionalism and perfection was my guiding maxim. As a friend, his advice was and will be invaluable to me.

I had the privilege to conduct my research in a very supportive and inspiring environment. Foremost, I want to thank the former and current members of the Reactive Systems Group at Saarland University, Jérôme Creci, Rayna Dimitrova, Klaus Dräger, Rüdiger Ehlers, Peter Faymonville, Michael Gerke, Lars Kuhtz, Andrey Kupriyanov, Markus Rabe, Christa Schäfer, Sven Schewe, and Hazem Torfah.

I am also happy that I had the opportunity to work with a number of remarkable people, such as Werner Damm, Alexandre David, Daniel Fass, Holger Hermanns, Robert Mattmüller, Linh Thi Xuan Phan, Andreas Podelski, Jan Rakow, Christoph Scholl, Tobe Toben, and Bernd Westphal, to only mention a few.

I am honored that Jean-François Raskin and Oded Maler joined my thesis committee and reviewed this thesis. I am grateful for their valuable comments.

Furthermore, I want to thank the German Research Foundation (DFG) as part of the AVACS project as well as the International Max Planck Research School for Computer Science (IMPRS-CS) for funding my research.

Many people outside academic computer science influenced this thesis. First of all, I am deeply grateful for the inspiration and love that I found in Barbara. Thank you for being part of my life! I cannot express in words my gratitude to my family, Gertrud, Michael, and Henrike, for their unconditional support and the values they taught me. I also want to thank Renate, Harald, and Christian Breunig for their valuable advice and the great time we had together. Finally, I am happy to have good friends on whom I can always rely on, such as Martin Bauer, Karsten Koller, Christian Krauß, Stefan Mann, Benjamin Ripperger, Stephan Sandvoss, and Oliver Welsch.

Thank you!

Contents

Chapter 1

Introduction

1.1 Succinct Systems

In the last decades, as system designs have become increasingly more complex, the need for computer-aided analysis techniques arose. At the same time, the practically available computing power reached a level, where novel algorithmic techniques enabled the automated reasoning about formally specified systems. An important milestone was the introduction of *model checking* [Clarke and Emerson, 1981, Queille and Sifakis, 1982], a verification technique that automatically checks whether a given system satisfies a specification.

The initial enthusiasm was soon damped by the discovery of the *state explosion problem*: Despite the indisputable algorithmic simplicity of model checking, for even the most technically elaborate approaches (based on, e.g., symbolic techniques [Burch et al., 1992, Clarke et al., 2001a] or abstraction refinement [Clarke et al., 2003]), some instances of some model classes persistently caused an exponential blow-up in the time and space complexity. As a sobering insight, it turned out that this blow-up is inherent for almost all interesting analysis problems for systems whose instances are based on formalisms that allow the *succinct specification* of exponentially large state spaces.

However, the usefulness of automatic analysis techniques such as model checking becomes apparent especially when succinct systems are under investigation, whose manual verification represents a nontrivial task. In the last decades, various modeling formalisms were introduced to succinctly describe the behavior of systems. Three prominent examples for such formalisms are *communicating state machines*, *propositional planning tasks*, and *timed automata*.[1] While each formalism extends finite state machines differently (i.e., by introducing concurrency, propositions, or clocks, respectively), in princi-

[1] Which serve as motivating examples in this introduction, but which are also discussed in more detail later in the thesis.

ple, their models compactly describe systems with an exponential number of states.

After their introduction, for each formalism, independent lines of research emerged aiming at deepen the understanding of the particular succinctness that is specific for the formalism, ranging from establishing worst-case complexity bounds to the development of efficient analysis algorithms and data structures. However, in the vast majority of the works that can be found in the literature, the unifying aspect of succinctness was ignored: From a theoretical point of view, the inherent exponential blow-ups in the worst-case complexities of the main analysis problems were established for *each formalism individually*. Figure 1.1 exemplarily shows the situation for the reachability and alternating reachability problem.

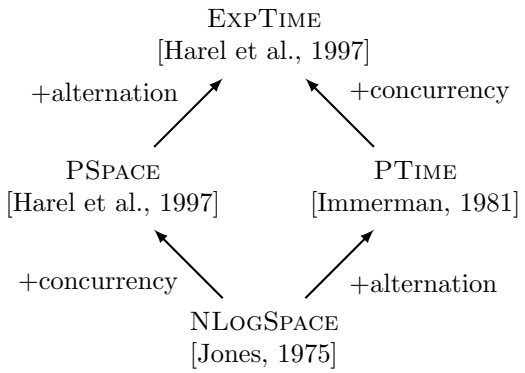
By continuing these lines of research in isolation, we could devise individual results for other problems of interest, each exploiting the particular succinctness of the respective formalism. However, it would be much more desirable to have a deeper understanding of succinctness in general; to have a technique that *captures the succinctness* of a given modeling formalism, and that can then be used to *uniformly provide algorithmic insights* for many analysis problems.

In order to achieve this goal, we have to treat succinctness as the key aspect of modeling formalisms; we need to develop a methodology that can be used to identify the *degree of succinctness* of a particular analysis problem.

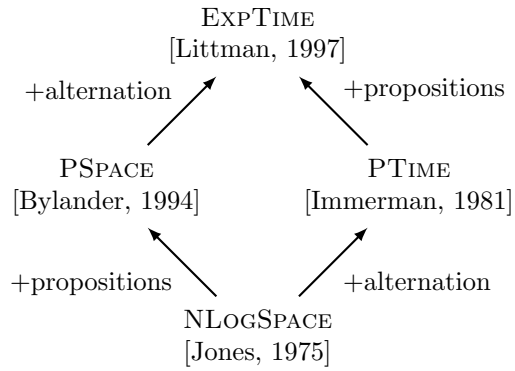
Contribution of this thesis. The main contribution of this thesis is to provide a unifying view on the succinctness of systems. In particular, we have the following key contributions.

- The key theoretical contribution is the introduction of *sequential circuit machines*, a new universal computation model that focuses on succinctness as the central aspect. We will demonstrate that many well-known modeling formalisms exhibit an immediate connection to this new machine model. Once a (syntactic) connection is established, many complexity bounds for structurally restricted sequential circuit machines can be transferred to a specific formalism in a uniform manner.

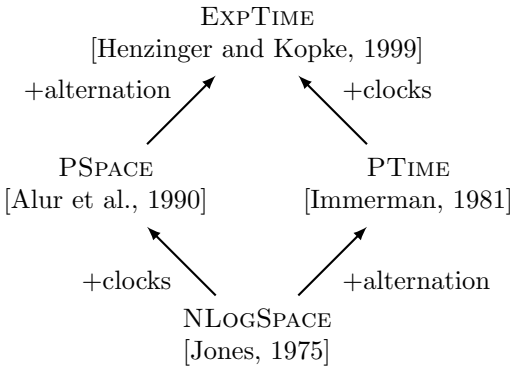
- As a consequence, besides a far-reaching unification of independent lines of research, we are also able to provide matching complexity bounds for various analysis problems, whose complexities were not known so far. For example, we establish matching lower and upper bounds of the small witness problem and several variants of the bounded synthesis problem for timed automata, a particularly important succinct modeling formalism.



(a) Communicating state machines.



(b) Propositional planning tasks.



(c) Timed automata.

Figure 1.1: The complexity of the reachability and alternating reachability problem for three succinct modeling formalisms.

- Also for timed automata, our complexity-theoretic analysis leads to the identification of tractable fragments of the timed synthesis problem under partial observability. Specifically, we identify timed controller

synthesis based on discrete or template-based controllers to be equivalent to model checking. Based on this discovery, we develop a new model checking-based algorithm to efficiently find feasible template instantiations.

- From a more practical perspective, this thesis also studies the preservation of succinctness in analysis algorithms using symbolic data structures. While efficient techniques exist for specific forms of succinctness considered in isolation, we present a general approach based on abstraction refinement to combine off-the-shelf symbolic data structures. In particular, for handling the combination of concurrency and quantitative timing behavior in networks of timed automata, we report on the tool SYNTHIA which combines binary decision diagrams with difference bound matrices. In a comparison with the timed model checker UPPAAL and the timed game solver UPPAAL-TIGA running on standard benchmarks from the timed model checking and synthesis domain, respectively, the experimental results clearly demonstrate the effectiveness of our new approach.

1.2 The Complexity of Explicit Systems

In theoretical computer science, the worst-case complexity of a problem is defined in terms of the least computational power needed to solve its hardest instances. This notion is usually formalized as a restriction imposed on the resources of an underlying universal machine model. The oldest and still most important such machine model is the *Turing machine*, originally introduced by Alan M. Turing in a groundbreaking paper [Turing, 1937] laying down the very foundations of computer science.

A Turing machine is a simple computation device that reads an input, given as a binary string, and either halts eventually or diverges forever. Each machine is supplemented with a (bounded or unbounded) work tape representing a general purpose memory that can sequentially be accessed in a read/write manner. Turing machines are regarded as the canonical universal computation model in the sense that, for a given amount of resources (e.g., maximum running time or memory consumption), there is no other formalism that strictly possesses more computational power – also known as the *Church-Turing Hypothesis*.

The main purpose of Turing machines is to unify the various syntactic flavors of programming languages and, more generally, to capture the essence of *sequential random access machines* satisfying the *von Neumann Property*:

At each instant only a bounded amount of activity can occur.

It is therefore no surprise that in the definition of the model, the behavior of a concrete machine is specified as a collection of instructions defining

which action (i.e., change of state and manipulation of the memory) has to be executed when the machine is in a particular configuration. Hereby, the execution of an instruction may only depend on the current state of the machine and the single memory cell that is currently selected. The effect of an instruction also may only affect the current state and the current memory cell.

1.3 A Uniform Approach to Succinctness

Explicit formalisms such as the Turing machine are perfect for unifying sequential computation models. However, they appear to be inappropriate to capture the succinctness of modeling formalisms: Reductions from the halting problem for resource-bounded Turing machines to a particular analysis problem for a succinct modeling formalism usually require nontrivial constructions exploiting the succinctness of the formalism to cause an exponential blow-up in the space consumption or running time of the Turing machine. For example, in the results depicted in Figure 1.1, each PSPACE or EXPTIME hardness proof is based on a technically involved simulation of a nondeterministic/alternating Turing machine. The question arises, whether we can come up with a universal computation model, which is better suited to capture the succinctness of systems.

As a starting point for our investigation, we recall the line of research on problems defined on succinctly specified graphs that was started by Galperin and Wigderson [1983] (see Section 3.5 on Page 38 for detailed bibliographic remarks). Under the assumption that problem instances are given as Boolean circuits, in these works, a general lifting technique was developed to upgrade hardness proofs for explicitly (i.e., nonsuccinctly) given instances to the exponentially harder succinct case.

Unfortunately, one cannot directly apply that lifting technique to a succinct computation model such as timed automata: Succinct modeling formalisms only indirectly (using, e.g., clocks, propositions, concurrency, etc.) describe graphs of exponential size through their semantics, while the lifting technique assumes that the entire problem instance is already given succinctly. In fact, modeling formalisms often *combine* different forms of succinctness. For example, the discrete behavior of timed automata is described using an explicitly given control structure, while the timed behavior can be succinctly defined using clock variables.

As a quintessence, on the one hand, we have Turing machines capturing the sequential, state-based aspect of computation models, but that fail in reflecting the succinctness in the specification of complex systems. On the other hand, we have combinatorial (i.e., nonpersisting) Boolean circuits as a generalization of succinctness. The key theoretical contribution of this thesis is to introduce *sequential circuit machines* as a new universal computation

model that combines Turing machines with Boolean circuits. Instead of a tape, a sequential circuit machine uses read/write registers to represent its memory. Instead of an explicitly given transition function, the behavior of a sequential circuit machine is defined as a Boolean circuit. In the step-wise execution of a machine, the new values of the registers are computed by the Boolean circuit that reads the old values of the registers. Analogously to Turing machines, the notion of halting is defined in terms of the reachability of a dedicated configuration.

As we will see later in this thesis, many well-known modeling formalisms have an immediate correspondence with sequential circuit machines. On the one hand, this results in an immediate unification of many (so-far unrelated) complexity results that can be found in the literature. On the other hand, once a result is established for sequential circuit machines in general, it can easily be transferred to all formalisms proven to be equivalent. For example, in this manner, we are able to establish the complexities for important analysis problems on timed automata (such as the small witness problem or various variants of the bounded synthesis problem), which were open until now.

1.4 Succinctness and Nondeterminism

Orthogonally to its succinctness, the complexity of an analysis problem also depends on the power of the existential and universal nondeterminism that is needed to solve the problem. That is, the increase in complexity due to the succinctness in the instance description depends on how these two fundamental types of nondeterminism actually make use of it.

As it turns out, by treating the succinctness aspect as a "first class citizen" in the underlying computation model, all major complexity classes[2] between LogSpace and 2ExpTime can be characterized in terms of the power that the existential and universal nondeterminism have at each step of the computation. Note that this is in contrast to the usual characterization based on Turing machines, where one restricts resources such as the maximum running time or memory consumption.

More clearly, in this thesis, we work out that every problem that is complete for a major complexity class can be precisely characterized by the following parameters.

(1) *Degree of nondeterminism*: the number of bits that the two types of nondeterminism can determine in each step;

(2) *Precision of observability*: the number of bits determined by the universal nondeterminism that can be observed by the existential nondeterminism;

[2] which are precisely defined in Section 2.3.2 on Page 19

(3) *Amount of memory*: the number of bits that can be used by the two types of nondeterminism to store information that can be recalled in subsequent steps.

A key strength of sequential circuit machines is their ability to precisely quantify the power of the existential and universal nondeterminism in this respect. In our new setting, it is therefore much simpler to study the impact of restricting the capabilities of a particular nondeterminism on the complexity. For example, we discover the general fact that bounding the memory available to the existential nondeterminism always dominates (complexity-wise) the impact of a bounded observability. In particular, in the context of controller synthesis for succinct systems, by assuming a unary bound on the size of the controller (which, as we will see, corresponds to a bound on the existential memory), every synthesis problem can be reduced to a model checking problem.

Based on this insight, for the important class of timed automata, we exemplarily develop the *template-based approach to controller synthesis*, which represents the first efficient synthesis algorithm for timed controllers with partial observability. The key idea is to restrict the model space to the instantiations of a given *template*, represented as a timed automaton with parametric control structure. Due to this restriction, synthesis now boils down to a one-player search problem, which can be efficiently implemented as a symbolic model checking algorithm. Synthesis based on templates is thus significantly cheaper than standard synthesis and produces much simpler controllers.

1.5 Combined Succinctness

A theoretically optimal, yet practically inefficient, analysis algorithm for a certain class of succinct systems can easily be obtained by just applying a nonsuccinct version of the algorithm on the explicit enumeration of the exponentially large state space. However, despite the inherent theoretical blow-up in the analysis of succinct systems, an efficient algorithm would try to *preserve the succinctness* of the given model during the analysis as much as possible. A promising approach into that direction is the use of *symbolic data structures and algorithms* for representing and manipulating sets of (a potentially infinite number of) states.

In practice, certain techniques have been proven effective for specific forms of succinctness. For example, *binary decision diagrams* [Bryant, 1986] and *difference bound matrices* [Dill, 1989] are two symbolic data structures which, to a certain extend, nicely preserve the succinctness introduced by concurrency [Burch et al., 1992] or timing behavior [Alur, 1999], respectively. However, for models exhibiting two different forms of succinctness by, e.g., allowing both concurrency and timing behavior, a technique to *combine*

symbolic data structures is needed. In particular, for timed automata, the development of such a technique has a long line of research (see Section 9.3 on Page 135 for detailed bibliographic remarks) and is still an active field of research and engineering.

This thesis presents a solution to this practical challenge in form of a general approach for combining symbolic data structures yielding promising experimental results. Based on the guiding principle to strictly keep the discrete and the continuous state information apart, the key idea of our approach is to use a data structure specialized for discrete state sets to produce a sequence of *syntactic abstractions* of the original model with increasing precision. For each abstraction, we apply standard analysis techniques based on a data structure specialized for continuous state sets to obtain an under- and an overapproximation of the precise analysis result (e.g., approximations of the set of reachable states). The approximations are used to (1) obtain refinements that increase the precision of the abstractions, and (2) identify irrelevant parts of the model that do not need to be analyzed.

1.6 Relation to Other Works

In the literature, a vast amount of work can be found in which only a particular aspect of succinctness is considered (e.g., the research on concurrent systems, propositional planning, or timed automata). As already indicated in Section 1.1, the majority of these works only consider succinctness in a specific form (e.g., succinctness based on concurrency, clocks, or propositions). A systematic analysis that studies the fundamental properties of succinctness or that treats succinctness as a unifying computational aspect, to the best of the author's knowledge, has not been published yet.

The unifying theory on succinctness that is presented in this thesis is in the same spirit as the line of research on *parallel computations*. In the 1970s and 1980s, numerous formalisms were proposed to model computations that are able to execute steps in parallel. From a complexity-theoretic point of view, for each formalism it was observed that parallel running time corresponds to space consumption in a pure sequential, nonparallel setting. The unification of these results was achieved by extending nondeterministic Turing machines by a universal nondeterminism resulting in the alternating Turing machine model [Chandra et al., 1981] (see Section 3.5 on Page 38 for a detailed related work survey). This thesis can be seen as a further generalization of that seminal result, as parallelism is just a particular form of succinctness. In fact, our new machine model, sequential circuit machines, generalizes alternating Turing machines, as the power of the existential and universal nondeterminism can now be quantified in a much more fine-grained way (in terms of observability and number of controllable bits per step).

As already mentioned in Section 1.3, this thesis is inspired by the line of research on *succinctly specified graphs* that was started by Galperin and Wigderson [1983]. In those works, the underlying structure (i.e., the problem instance) is succinctly described, which leads to a *symmetric* increase of the power of the existential and universal nondeterminism. As an extension of that line of research, this work provides a more detailed analysis on succinctness that also considers an *asymmetric* increase of the power of the two types of nondeterminism.

This thesis is also inspired by *circuit complexity*, a branch of computational complexity theory in which complexity classes are characterized by the expressivity of a certain class of Boolean circuits. The classic theory only focuses on succinctness in *memoryless* computations, as only circuit classes without memory elements are considered. This thesis extends that line of research by introducing sequential, *state-based* computations, which are described by combinatorial circuits.

A different approach to computational complexity is represented by *descriptive complexity theory*. Here, a complexity class is characterized by the expressiveness of a certain logic needed to specify the problems in that class. Descriptive complexity theory and the unifying theory of succinctness developed in this thesis have in common that both view at computational complexity from a different (non Turing machine-based) perspective. They, however, fundamentally differ with respect to that perspective.

From a more practical point of view, there has been a lot of research concerning the development of symbolic algorithms and data structures for the analysis of succinct systems. While efficient techniques exist for specific forms of succinctness (see Section 9.3 on Page 135 for a detailed related work survey), there is relatively little work on the treatment of combined forms of succinctness. An exception are timed automata, where, in the last two decades, several approaches were proposed to deal with both concurrency and timing behavior. A convincing solution, however, has not been achieved so far. This thesis also contributes in that research direction and provides a general combination approach for off-the-shelf symbolic data structures showing promising experimental results.

1.7 Publications

Material from this thesis has been published in the following works.

- Hans-Jörg Peter and Bernd Finkbeiner. **The Complexity of Bounded Synthesis for Timed Control with Partial Observability**. Proceedings of the *10^{th} International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2012)*.

- Bernd Finkbeiner and Hans-Jörg Peter. **Template-based Con-**

troller Synthesis for Timed Systems. Proceedings of the *18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*.

- Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. **Synthia: Verification and Synthesis for Timed Automata**. Proceedings of the *23rd International Conference on Computer Aided Verification (CAV 2011)*.

- Rüdiger Ehlers, Daniel Fass, Michael Gerke, and Hans-Jörg Peter. **Fully Symbolic Timed Model Checking using Constraint Matrix Diagrams**. Proceedings of the *31st IEEE Real-Time Systems Symposium (RTSS 2010)*.

- Rüdiger Ehlers, Michael Gerke, and Hans-Jörg Peter. **Making the Right Cut in Model Checking Data-Intensive Timed Systems**. Proceedings of the *11th International Conference on Formal Engineering Methods (ICFEM 2010)*.

- Rüdiger Ehlers, Robert Mattmüller, and Hans-Jörg Peter. **Combining Symbolic Representations for Solving Timed Games**. Proceedings of the *8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2010)*.

- Hans-Jörg Peter and Robert Mattmüller. **Component-based Abstraction Refinement for Timed Controller Synthesis**. Proceedings of the *30th IEEE Real-Time Systems Symposium (RTSS 2009)*.

1.8 Structure of the Thesis

The central theme of this thesis is the succinctness of systems. The general style of writing is to start abstractly and to finish concretely.

Following this principle, Part I lays down the theoretical foundations and formalizes the notion of succinctness by introducing sequential circuit machines as an alternative to Turing machines. In particular, after recalling the preliminaries in Chapter 2, Chapter 3 defines our new machine model together with its notion of completeness. Chapter 4 investigates equivalences between different structural bounds on sequential circuit machines as well as their relationship to Turing machines. This reveals that all major complexity classes between LogSpace and 2ExpTime can now be characterized in terms of syntactic restrictions on the new machine model. Chapter 5 demonstrates the applicability of our new model by connecting sequential circuit machines to some well-known modeling formalisms from the literature, leading to a unification of so-far independent research directions.

Part II focuses on the timed automaton computation model, which, besides its highly practical relevance, represents an interesting case to study the combination of different forms of succinctness. In particular, after recalling the basic definitions of the timed automaton formalism in Chapter 6, Chapter 7 establishes the connection to sequential circuit machines. Similar to Chapter 5, this results in a drastic simplification of many complexity-theoretic proofs for timed automata. Moreover, thanks to the connection to sequential circuit machines, we are able to establish matching lower and upper complexity bounds for various timed synthesis problems, whose precise complexity was unknown so far. In this context, we identify the subclass of discrete controllers for timed plants as a tractable but still practically relevant subproblem, leading to the template-based synthesis approach, which is presented in Chapter 8. As a concrete enabling technique for template-based synthesis and, more generally, to deal with orthogonal blow-ups induced by different forms of succinctness, Chapter 9 presents a general combination technique for symbolic data structures. Chapter 10 reports on an experimental evaluation based on the tool Synthia.

The thesis concludes with a summary and an outlook in Chapter 11.

Part I

Sequential Circuit Machines

Chapter 2

Preliminaries

2.1 General Notations and Basic Definitions

General notations. When we write $\log(n)$, we always refer to the base 2 logarithm of n. For a set X, we write 2^X to refer to the *power set* of X. To define a (partial or total) function $f : X \to Y$, we use the notation $[\, x \mapsto y \mid P(x) \,]$, where $x \in X$, $y \in Y$, and $P(x)$ is a predicate over X. For an object X, we define its *cardinality* (or, alternatively, its *size* or *length*) $|X|$. For example, $|X|$ refers to the sum of the cardinalities of the components if X is a tuple, or the sum of the cardinalities of the contained elements if X is a set). If X is infinite, by abuse of notation, we write $|X| = \infty$.

A *binary word* (or just a *word*) is a sequence from $\{0,1\}^*$. A *binary language* (or just a *language*) is a set of words.

A *class of functions* $\mathcal{F}$ is an infinite set of functions of the form $f : \mathbb{N} \to \mathbb{N}$. If $f \in \mathcal{F}$ then we require that $O(f) \subseteq \mathcal{F}$. We say that $\mathcal{F}$ is nondecreasing iff all its subsumed functions are nondecreasing. We define the following important classes of functions:

$$\mathcal{L} = O(\log n); \qquad \mathcal{P} = n^{O(1)}; \qquad \mathcal{E} = 2^{n^{O(1)}}$$

Here, we use the notation $O(\log(n))$ to refer to the class of functions of the type $f(n) = c \cdot \log(n)$, for some $c \in \mathbb{N}$. Similarly, we write $n^{O(1)}$ to refer to the set of all polynomials with variable n.

Directed graphs. A finite or infinite *directed graph* is a tuple $G = (V, E)$, where V is a finite or infinite set of *nodes* and $E \subseteq V \times V$ is a finite or infinite *edge relation*. A *path* in G is a sequence of nodes from V of the form $v_1, \ldots, v_n$ such that, for all $1 \leq i < n$, $(v_i, v_{i+1}) \in E$. We say that G is a *directed acyclic graph* (or just call G a *DAG*, for short) iff in every path of G, a particular node occurs at most once. We define the *indegree* or *fan-in* of a node $v \in V$ as $in(v) = |\{v' \mid (v', v) \in E\}|$, and the *outdegree* or *fan-out* of v as $out(v) = |\{v' \mid (v, v') \in E\}|$.

Decision problems. A *decision problem* (or just a *problem*) P is a sentence of the form "Does a property x hold for an object y?", where x and y, being mathematical objects, represent the *input* to P. For a given input i, a decision problem P either yields **yes** or **no**, written as either $\mathsf{P}(i) = \textbf{yes}$ or $\mathsf{P}(i) = \textbf{no}$, respectively. A canonical representation for an input is its binary encoding represented as a binary word. A decision problem is called *finite* iff its input is finite.

Example 2.1.1. *For a given directed graph $G = (V, E)$, where V is a finite set of nodes and E is represented as a set of pairs of nodes, an initial node $v_0 \in V$, and a bad node $b \in V$, the problem* Unreachability *is to decide whether b is not reachable from v_0 via the edges specified in E.*

When we apply the notions introduced above to this example, we have that the decision problem $\mathsf{P} =$ Unreachability *and that the input is the directed graph G, the initial node v_0, and the bad node b. Hence, the size of the input is $|V| + |E| + 2$.*

2.2 Boolean Functions and Combinatorial Circuits

2.2.1 Boolean Functions

For a finite set of *variables* X, we write $\vec{X}$ to refer to the set of total functions $X \to \{\textbf{false}, \textbf{true}\}$ that assign a Boolean value to each element in X. We use 0 for **false** and 1 for **true** interchangeably. We write $\vec{x} = \vec{0}$ as an abbreviation for $\forall x \in X : \vec{x}(x) = 0$. For a set $X' \subseteq X$ and a valuation $\vec{x} \in \vec{X}$, we write $\vec{x}(X')$ to refer to the *projected valuation* $\vec{x}' \in \vec{X}'$ such that $\vec{x}'(x) = \vec{x}(x)$ for all $x \in X'$. Similarly, for $x \in X$ and $b \in \{\textbf{false}, \textbf{true}\}$, we define $\vec{x}[x = b]$ to the valuation $\vec{x}'$ such that

$$\vec{x}'(x') = \begin{cases} b & \text{if } x' = x; \\ \vec{x}(x') & \text{otherwise.} \end{cases}$$

A *Boolean function over* X is a total function f that maps a valuation $\vec{x}$ from $\vec{X}$ to **false** or **true**. Syntactically, Boolean functions are composed of the following *subformulas*:

$$\textbf{false} \mid \textbf{true} \mid x \mid \neg x \mid f_1 \wedge \ldots \wedge f_n \mid f_1 \vee \ldots \vee f_n,$$

where $f_1, \ldots, f_n$ are Boolean functions over X and $x \in X$. **false** and **true** are called *constants*, subformulas of the form x and $\neg x$ are called *literals*, subformulas of the form $f_1 \wedge \ldots \wedge f_n$ called *conjunctions*, and subformulas of the form $f_1 \vee \ldots \vee f_n$ called *disjunctions*. For two Boolean functions f and g, we use $f \Rightarrow g$ as an abbreviation for $\neg f \vee g$, $f \Leftarrow g$ as an abbreviation for $g \Rightarrow f$, and $f \Leftrightarrow g$ as an abbreviation for $f \Rightarrow g \wedge g \Rightarrow f$. Whenever we have a formula of the form $\neg(f_1 \wedge \ldots \wedge f_n)$ or $\neg(f_1 \vee \ldots \vee f_n)$, we use De Morgan's

law to push the negation inwards and obtain $\neg f_1 \vee \ldots \vee \neg f_n$ or $\neg f_1 \wedge \ldots \wedge \neg f_n$, respectively. For some $x \in X$ and $b \in \{\mathbf{false}, \mathbf{true}\}$, we define $f[x = b]$ to be the Boolean function f', for which we have $f'(\vec{x}) = f(\vec{x}[x = b])$. The *size* $|f|$ of a Boolean function f is the number of its subformulas. The *alternation depth* of a Boolean function f is the maximal number of alternating nestings of $\wedge$ and $\vee$ operators. A Boolean formula f is *disjunctive normal form* (DNF) iff f is a disjunction with alternation depth 2. A Boolean formula f is *conjunctive normal form* (CNF) iff f is a conjunction with alternation depth 2.

The *formula DAG* of a Boolean function f is defined as $(G, L) = dag(f)$, where $G = (V, E)$ is a DAG and $L : E \rightharpoonup \{x, \neg x \mid x \in X\}$ is a partial function that labels some edges of G with literals. We define $dag(f)$ inductively as follows:

$$
\begin{aligned}
dag(\mathbf{false}) &:= \big((\{v_0, \bot, \top\}, \{(v_0, \bot)\}), \emptyset\big);\\
dag(\mathbf{true}) &:= \big((\{v_0, \bot, \top\}, \{(v_0, \top)\}), \emptyset\big);\\
dag(x) &:= \big((\{v_0, \bot, \top\}, \{(v_0, \top), (v_0, \bot)\}),\\
&\qquad [\,(v_0, \top) \mapsto x, (v_0, \bot) \mapsto \neg x\,]\big);\\
dag(\neg x) &:= \big((\{v_0, \bot, \top\}, \{(v_0, \top), (v_0, \bot)\}),\\
&\qquad [\,(v_0, \bot) \mapsto x, (v_0, \top) \mapsto \neg x\,]\big);\\
dag(f_1 \wedge \ldots \wedge f_n) &:= dag(f_1) \xrightarrow{\top} \cdots \xrightarrow{\top} dag(f_n);\\
dag(f_1 \vee \ldots \vee f_n) &:= dag(f_1) \mid \cdots \mid dag(f_n)
\end{aligned}
$$

Here, we use the notation $(G, L) \xrightarrow{\top} (G', L')$ to refer to the *conjunctive composition* of (G, L) and (G', L') such that all edges in G leading to G's $\top$ node are redirected to the unique root of G'. We use the notation $(G, L) \mid (G', L')$ to refer to the *disjunctive composition* of (G, L) and (G', L') such that the root nodes of G and G' are merged to a single root node. We define $root(G)$ to be the unique root of G, $true(G) = \bot$, and $false(G) = \top$.

We will use Boolean functions as the most abstract form of a symbolic representation for sets of binary words (e.g., for representing sets of discrete states). A *logarithmic encoding* of a finite nonempty set Y is a set of Boolean variables $\widehat{[\![Y]\!]}$ such that each element $y \in Y$ is characterized by a valuation $[\![y]\!]$ over $\widehat{[\![Y]\!]}$. Note that $|\widehat{[\![Y]\!]}| = \lceil \log |Y| \rceil$. For a Boolean function f over $\widehat{[\![Y]\!]}$ and a variable $x \in \widehat{[\![Y]\!]}$, we write $\exists x : f$ for $f[x = 0] \vee f[x = 1]$. For a set of variables $\{x_1, \ldots, x_n\} = X \subseteq \widehat{[\![Y]\!]}$, we write $\exists X : f$ for $\exists x_1 : \ldots \exists x_n : f$.

2.2.2 Combinatorial Circuits

A *combinatorial circuit* (or just a *circuit*) C is a tuple (V, E, I, O, G), where (V, E) is a directed acyclic graph, I and O are finite sets containing the names of the inputs and outputs of C, respectively, and G labels each node

with a gate type $G : V \to \{AND, OR, NOT\} \cup I \cup O$. Here, we assume that $(I \cup O) \cap \{AND, OR, NOT\} = \emptyset$. We always assume that for every node $v \in V$, the following holds: (1) if $G(v) \in I$, then $in(v) = 0$ and $out(v) > 0$; (2) if $G(v) \in O$, then $in(v) = 1$ and $out(v) = 0$; (3) otherwise (if v is an internal gate), $in(v) > 0$ and $out(v) > 0$ (i.e., we allow an unbounded fan-in for all internal gates). The *size* of C is the defined as $|C| = |V| + |E|$. The *depth* of C is the length of the longest path from an input to an output in C.

It is easy to see that every Boolean function f over a set of variables X can be transformed into a circuit f^c of the same size and (alternation) depth that reads from inputs X and writes to a single output, such that, for every valuation $\vec{x} \in \vec{X}$, $f(\vec{x}) = \mathbf{true}$ iff $f^c(\vec{x})$ computes 1. For the converse, we introduce the following property. We say that a circuit $C = (V, E, I, O, G)$ is *shared sub-circuit free* (or just C is *simple*), iff, for every $v \in V$, if $G(v) \notin I \cup O$ then $out(v) = 1$ (i.e., every gate, except inputs and outputs, is connected to exactly one successor gate). By pushing the NOT gates in front of the inputs (and dualizing the AND and OR gates), one can easily obtain $|O|$ equivalent Boolean functions $C_j : \vec{I} \to \{\mathbf{false}, \mathbf{true}\}$, $1 \leq j \leq |O|$, such that $|C_j| \leq |C|$, $depth(C_j) \leq depth(C)$, and, for a valuation $\vec{i} \in \vec{I}$, $C_j(\vec{i}) = \mathbf{true}$ iff output j of C evaluates to 1 when the inputs are set to $\vec{i}$. Note that, as we assume an unbounded fan-in, many important circuits (such as incrementers, adders, comparators, multiplexers, etc.) can be represented as simple circuits with a constant depth (i.e., a depth that is independent of the number of inputs).

We say that a circuit $C = (V, E, I, O, G)$ *reads from* some elements I' iff $I' \subseteq I$. Analogously, C *writes to* some elements O' iff $O' \subseteq O$.

2.3 Turing Machines and Complexity Classes

We assume that the reader is familiar with the concept of Turing machines and complexity classes [see, e.g., Papadimitriou, 1994]. However, in this section, we define our syntactic variant of the Turing machine model that we will use in the rest of the thesis, and recall the definition of important complexity classes.

2.3.1 Turing Machines

Without loss of generality, we assume that our Turing machines have access to a read-only input tape and a read/write work tape. Also, we assume a binary tape alphabet (for both input and work tape). An *alternating Turing machine* $\mathcal{T}$ is represented by a tuple (Q, q_0, T, δ), where

- Q is a finite set of *states*,
- $q_0 \in Q$ is the initial state,

- $T : Q \to \{\mathsf{D}, \mathsf{E}, \mathsf{A}, \mathsf{Acc}\}$ is a total function that assigns a *type* to each state, and
- $\delta : Q \times \{0,1\}^2 \to S^2$ is the *transition function*.

Here, $S = Q \times \{0,1\}^3$ is the set of *steps* $\mathcal{T}$ can perform. A step $(q', \gamma', d_i, d_w) \in S$ defines the next state q', the symbol γ' that should be written into the current work tape cell, and the directions d_i and d_w in which the head of the input and work tape should move, respectively. For a state $q \in Q$ and an input and work symbol $\gamma_i, \gamma_w \in \{0,1\}$, let $\delta(q, \gamma_i, \gamma_w) = (s_1, s_2)$, q_1 be the next state of s_1, and q_2 be the next state of s_2. Now, q *is accepting* iff either

- $T(q) = \mathsf{Acc}$,
- $T(q) = \mathsf{D}$ and q_1 is accepting,
- $T(q) = \mathsf{E}$ and q_1 or q_2 is accepting, and
- $T(q) = \mathsf{A}$ and both q_1 and q_2 are accepting.

We distinguish between the following *types of Turing machines*: $\mathcal{T}$ is

- *existential nondeterministic* if $T(q) \neq \mathsf{A}$,
- *universal nondeterministic* if $T(q) \neq \mathsf{E}$, and
- *deterministic* if $T(q) \neq \mathsf{A}$ and $T(q) \neq \mathsf{E}$,

for all $q \in Q$.

We say that $\mathcal{T}$ *accepts a word* $w \in \{0,1\}^*$ if, and only if, q_0 is accepting while w is stored in the input tape of $\mathcal{T}$. We say that $\mathcal{T}$ *decides a language* $L \subseteq \{0,1\}^*$ if, and only if, for every word $w \in \{0,1\}^*$, $\mathcal{T}$ accepts w iff $w \in L$. The *running time* is defined as the number of steps $\mathcal{T}$ performs. The *space consumption* is defined as the number of work tape cells $\mathcal{T}$ uses up.

2.3.2 Complexity Classes

In this subsection, we recall the classical definition of complexity classes that can be characterized by Turing machines with *resource bounds*.

Formally, a *complexity class* is a set of binary languages. Let $f(n) \geq \log(n)$ be a nondecreasing function. We define $\textsc{Time}(f)$ as the complexity class that exactly contains those languages that can be decided by a deterministic Turing machine with running time $O(f(n))$, where n is the length of the input. We define $\textsc{Space}(f)$ as the complexity class that exactly contains those languages that can be decided by a deterministic Turing machine with space consumption $O(f(n))$, where n is the length of the input.

Depending on the underlying Turing machine type, we obtain the complexity classes NTime and NSpace for existential nondeterministic (or just nondeterministic) Turing machines, coNTime and coNSpace for universal nondeterministic Turing machines[1], and ATime and ASpace for alternating Turing machines.

In the following, when we write, e.g., $\textsc{Time}(\mathcal{F})$, for a class of functions $\mathcal{F}$, we mean

$$\bigcup \{\textsc{Time}(f) \mid f \text{ is of type } \mathcal{F}\}.$$

For the class of polynomial functions $\mathcal{P}$ and the class of exponential functions $\mathcal{E}$, we define the following important *time-bounded* complexity classes:

$$\begin{aligned}
\textsc{PTime} &= \textsc{Time}(\mathcal{P});\\
\textsc{NPTime} &= \textsc{NTime}(\mathcal{P});\\
\textsc{coNPTime} &= \textsc{coNTime}(\mathcal{P});\\
\textsc{APTime} &= \textsc{ATime}(\mathcal{P});\\
\textsc{ExpTime} &= \textsc{Time}(\mathcal{E});\\
\textsc{NExpTime} &= \textsc{NTime}(\mathcal{E});\\
\textsc{coNExpTime} &= \textsc{coNTime}(\mathcal{E});\\
\textsc{AExpTime} &= \textsc{ATime}(\mathcal{E});\\
\textsc{2ExpTime} &= \textsc{Time}(2^{\mathcal{E}})
\end{aligned}$$

Also, for the class of logarithmic functions $\mathcal{L}$, the class of polynomial functions $\mathcal{P}$, and the class of exponential functions $\mathcal{E}$, we define the following important *space-bounded* complexity classes:

$$\begin{aligned}
\textsc{LogSpace} &= \textsc{Space}(\mathcal{L});\\
\textsc{NLogSpace} &= \textsc{NSpace}(\mathcal{L});\\
\textsc{coNLogSpace} &= \textsc{coNSpace}(\mathcal{L});\\
\textsc{ALogSpace} &= \textsc{ASpace}(\mathcal{L});\\
\textsc{PSpace} &= \textsc{Space}(\mathcal{P});\\
\textsc{NPSpace} &= \textsc{NSpace}(\mathcal{P});\\
\textsc{coNPSpace} &= \textsc{coNSpace}(\mathcal{P});\\
\textsc{APSpace} &= \textsc{ASpace}(\mathcal{P});\\
\textsc{ExpSpace} &= \textsc{Space}(\mathcal{E});\\
\textsc{NExpSpace} &= \textsc{NSpace}(\mathcal{E});\\
\textsc{coNExpSpace} &= \textsc{coNSpace}(\mathcal{E});\\
\textsc{AExpSpace} &= \textsc{ASpace}(\mathcal{E})
\end{aligned}$$

[1] Not to be confused with *universal Turing machines* that simulate other Turing machines.

The scope of this thesis will focus solely on these classes, which we also call *major complexity classes*.

The following relations are known (we refer to Papadimitriou [1994] for a more detailed discussion):

$$\text{LogSpace} \subseteq \text{NLogSpace} = \text{coNLogSpace} \subseteq \text{ALogSpace};$$
$$\text{PTime} \subseteq (\text{co})\text{NPTime} \subseteq \text{APTime};$$
$$\text{PSpace} = \text{NPSpace} = \text{coNPSpace} \subseteq \text{APSpace};$$
$$\text{ExpTime} \subseteq (\text{co})\text{NExpTime} \subseteq \text{AExpTime};$$
$$\text{ExpSpace} = \text{NExpSpace} = \text{coNExpSpace} \subseteq \text{AExpSpace}$$

The equalities PSpace = NPSpace and ExpSpace = NExpSpace were proven by Savitch [1970]. The NSpace = coNSpace equalities are due to Szelepcsényi [1987] and Immerman [1988]. At the time this thesis is being written, to the best of the author's knowledge, none of the above inclusions are proven to be strict.[2] However, due to the *time and space hierarchy theorems* [Hartmanis and Stearns, 1965, Stearns et al., 1965], the following strict inclusions are known:

$$\text{PTime} \subsetneq \text{ExpTime} \subsetneq \text{2ExpTime};$$
$$\text{LogSpace} \subsetneq \text{PSpace} \subsetneq \text{ExpSpace}$$

The following equalities are due to Chandra et al. [1981]:

$$\text{ALogSpace} = \text{PTime};$$
$$\text{APTime} = \text{PSpace};$$
$$\text{APSpace} = \text{ExpTime};$$
$$\text{AExpTime} = \text{ExpSpace};$$
$$\text{AExpSpace} = \text{2ExpTime}$$

2.4 Two-Player Games

We use infinite two-player games to model the reactive interaction between the existential and universal nondeterminism during a computation. We refer to the players as E and A, respectively, and always assume that they have complementary winning objectives. In general, we always assume that games are determined (i.e., there can be no draw in a given game – one player wins iff the other loses). For the purpose of this thesis, it suffices to focus on reachability and safety winning objectives for the players. As a notation, for a player $p \in \{\mathsf{E}, \mathsf{A}\}$, we write $\overline{p}$ to refer to the player in $\{\mathsf{E}, \mathsf{A}\} \setminus \{p\}$.

[2] It is, however, common believe that all inclusions are strict.

2.4.1 Game Arenas

A *game arena* (or just an *arena*) $\mathcal{A}$ is a tuple $(S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$, where

- $S = S_{\mathsf{E}} \uplus S_{\mathsf{A}}$ are the *positions* of the players,
- $s_0 \in S$ is the *initial position*,
- $\Sigma = \Sigma_{\mathsf{E}} \uplus \Sigma_{\mathsf{A}}$ are the *decisions* of the players, and
- $\Delta : ((S_{\mathsf{E}} \times \Sigma_{\mathsf{E}}) \cup (S_{\mathsf{A}} \times \Sigma_{\mathsf{A}})) \rightharpoonup S$ is a partial function defining the *moves* of the players.

We require both Σ_{E} and Σ_{A} to be nonempty. For a player $p \in \{\mathsf{E}, \mathsf{A}\}$, we say that p *is deterministic in* $\mathcal{A}$ iff p can only play a single move, i.e., $|\Delta_p| = 1$. We say that $\mathcal{A}$ is a *p-arena* iff $\overline{p}$ is deterministic in $\mathcal{A}$. We say that $\mathcal{A}$ is a *one-player arena* iff at least one player is deterministic in $\mathcal{A}$. We say that $\mathcal{A}$ is *deterministic* iff both players deterministic in $\mathcal{A}$. We call $\mathcal{A}$ *infinite* if $S \cup \Sigma_{\mathsf{E}} \cup \Sigma_{\mathsf{A}}$ is infinite, otherwise we call $\mathcal{A}$ *finite*.

A *play* π in $\mathcal{A}$ is a (finite or infinite) sequence of decisions of the form $(d_i)_{i \in \mathbb{N}_{\geq 1}}$, where each $d_i \in \Sigma$. We write $\mathsf{FinPlays}(\mathcal{A})$ to refer to all finite plays, and $\mathsf{Plays}(\mathcal{A})$ to refer to all (finite and infinite) plays in $\mathcal{A}$. We define the *length* of a play π, written as $|\pi|$, to be the number of decisions in the finite sequence, or ∞ if π is infinite. We write $\pi = \bot$, if $|\pi| = 0$. For a finite play π and a decision $d \in \Sigma$, we use the notation $\pi \circ d$ to denote the finite play that is the *concatenation* of π and d. We define the function $\mathsf{Pos} : \mathsf{FinPlays} \to S$ to refer to the position that is reached after a given finite play:

$$\mathsf{Pos}(\pi) = \begin{cases} s_0 & \text{if } \pi = \bot; \\ s & \text{if } \pi' = \pi \circ d \text{ and } s = \Delta(\mathsf{Pos}(\pi'), d). \end{cases}$$

2.4.2 Views and Strategies

For a Player $p \in \{\mathsf{E}, \mathsf{A}\}$ and a game arena $\mathcal{A} = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$, a *view* $\mathcal{V}$ for p on $\mathcal{A}$ is a tuple $(\Sigma^{\mathsf{obs}}, \mathit{vis})$, where

- $\emptyset \neq \Sigma^{\mathsf{obs}} \subseteq \Sigma_{\overline{p}}$ is a subset of decisions of the opponent defining the *observations* of Player p and
- $\mathit{vis} : \Sigma_{\overline{p}} \to (\Sigma^{\mathsf{obs}} \uplus \{\varepsilon\})$ is a function that projects an opponent decision to its corresponding observation or to a dedicated symbol ε representing a *stuttering event*.

A stuttering event is a decision played by $\overline{p}$ that is not perceivable by p. Note that this is in contrast to other decisions played by $\overline{p}$ that are observable but not necessarily distinguishable by p. Since we have turn-based games,

we require Σ^{obs} to contain at least one element to notify Player E when he has to move.

Clearly, if $\Sigma^{\mathsf{obs}} = \Sigma_{\overline{p}}$ and *vis* is the identity function, then we say that p plays a *game of complete information* (alternatively, we also say that p has *full observability*). If Σ^{obs} is a singleton (i.e., Σ^{obs} is finite and $|\Sigma^{\mathsf{obs}}| = 1$), we say that p plays a *blindfold game* (alternatively, we also say that p has *no observability*). Otherwise, we say that p plays a *private game* (alternatively, we also say that p has *partial observability*).

By abuse of notation, for a play $\pi \in \mathsf{FinPlays}(\mathcal{A})$, we define

$$vis(\pi) = \begin{cases} \bot & \text{if } \pi = \bot; \\ vis(\pi') & \text{if } \pi' = \pi \circ d \text{ and } vis(d) = \varepsilon; \\ vis(\pi') \circ vis(d) & \text{if } \pi' = \pi \circ d \text{ and } vis(d) \neq \varepsilon. \end{cases}$$

A function $f : \mathsf{FinPlays}(\mathcal{A}) \to \Sigma_p$ is called a *strategy* for Player p in $\mathcal{A}$ iff for any finite play π we have that if $s = \mathsf{Pos}(\pi) \in S_p$ then $\exists s' : s' = \Delta_p(s, f(\pi))$. We say that f *respects* a view $\mathcal{V} = (\Sigma^{\mathsf{obs}}, vis)$, written as $f \models \mathcal{V}$, iff f does not distinguish between unobservable decisions: for all finite plays $\pi_1, \pi_2 \in \mathsf{FinPlays}(\mathcal{A})$,

$$\text{if } vis(\pi_1) = vis(\pi_2) \text{ then } f(\pi_1) = f(\pi_2).$$

The set of all strategies for a player p in $\mathcal{A}$ is defined as $\mathcal{F}^p_{\mathcal{A}}$.

We use *Mealy machines* as the standard representation for strategies. Formally, a *Mealy machine* M is a tuple $(Q, q_0, \Sigma_{\mathsf{A}}, \Sigma_{\mathsf{E}}, T)$ consisting of

- a set of states Q,
- an initial state $q_0 \in Q$,
- a set of input actions Σ_{A},
- a set of output actions Σ_{E}, and
- a partial function $T : Q \times \Sigma_{\mathsf{A}} \rightharpoonup Q \times \Sigma_{\mathsf{E}}$ defining the transitions of the machine.

Assuming M represents a strategy for E, intuitively, in each round of a play, after A made his decision, T is executed to query the next decision for E and the target state for M. The *size of a strategy* (or, alternatively, the *amount of memory of a strategy*) f is defined as $|f| = |Q|$, where Q is the set of states of the smallest (in terms of states) Mealy machine representation of f.

2.4.3 Model Checking and Synthesis

The semantics of a game arena $\mathcal{A} = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$ is defined in terms of the possible plays that arise in the interplay between the two players. For a strategy $f_e \in \mathcal{F}^{\mathsf{E}}_{\mathcal{A}}$ and a strategy $f_a \in \mathcal{F}^{\mathsf{A}}_{\mathcal{A}}$, we define the *outcome* as the unique play that arises when Player E sticks to f_e and Player A sticks to f_a:

$$\mathsf{Outcome}_{\mathcal{A}}(f_e, f_a) = \big\{\pi \in \mathsf{Plays}(\mathcal{A}) \mid \forall 1 \leq i < |\pi| : \pi[i+1] = \delta(\pi[1..i], f_e, f_a)\big\}$$

where

$$\delta(\pi, f_e, f_a) = \begin{cases} f_e(\pi) & \text{if } \mathsf{Pos}(\pi) \in S_{\mathsf{E}}; \\ f_a(\pi) & \text{if } \mathsf{Pos}(\pi) \in S_{\mathsf{A}}. \end{cases}$$

We define the *set of traces* of $\mathcal{A}$ as the set of all possible plays:

$$\mathsf{Traces}(\mathcal{A}) = \bigcup_{f_e \in \mathcal{F}^{\mathsf{E}}_{\mathcal{A}}, f_a \in \mathcal{F}^{\mathsf{A}}_{\mathcal{A}}} \mathsf{Outcome}_{\mathcal{A}}(f_e, f_a)$$

If $\mathcal{A}$ is a p-arena, for a player $p \in \{\mathsf{E}, \mathsf{A}\}$, by abuse of notation, we also write $\mathsf{Outcome}_{\mathcal{A}}(f)$ for an $f \in \mathcal{F}^{p}_{\mathcal{A}}$. We borrow some notations from CTL model checking [Clarke and Emerson, 1981] and define, for a bound $\beta \in \mathbb{N} \cup \{\infty\}$ on the size of the strategy and a set of positions $X \subseteq S$,

$$\begin{aligned}
\mathcal{A} \models^{\beta} \mathsf{EF}(X) \;:\Longleftrightarrow\; & \\
& \exists f \in \mathcal{F}^{p}_{\mathcal{A}} : (\beta = \infty \vee |f| \leq \beta) \wedge \\
& \quad \exists i \geq 1 : \mathsf{Pos}(\mathsf{Outcome}_{\mathcal{A}}(f)[1..i]) \in X; \\
\mathcal{A} \models^{\beta} \mathsf{EG}(X) \;:\Longleftrightarrow\; & \\
& \exists f \in \mathcal{F}^{p}_{\mathcal{A}} : (\beta = \infty \vee |f| \leq \beta) \wedge \\
& \quad \forall i \geq 1 : \mathsf{Pos}(\mathsf{Outcome}_{\mathcal{A}}(f)[1..i]) \in X; \\
\mathcal{A} \models^{\beta} \mathsf{AF}(X) \;:\Longleftrightarrow\; & \\
& \forall f \in \mathcal{F}^{p}_{\mathcal{A}} : (\beta = \infty \vee |f| \leq \beta) \wedge \\
& \quad \exists i \geq 1 : \mathsf{Pos}(\mathsf{Outcome}_{\mathcal{A}}(f)[1..i]) \in X; \\
\mathcal{A} \models^{\beta} \mathsf{AG}(X) \;:\Longleftrightarrow\; & \\
& \forall f \in \mathcal{F}^{p}_{\mathcal{A}} : (\beta = \infty \vee |f| \leq \beta) \wedge \\
& \quad \forall i \geq 1 : \mathsf{Pos}(\mathsf{Outcome}_{\mathcal{A}}(f)[1..i]) \in X.
\end{aligned}$$

For a one-player arena $\mathcal{A}$, a set of positions $X \subseteq S$, and a property φ of the form EF, EG, AF, or AG, we call problems deciding whether $\mathcal{A} \models^{\infty} \varphi(X)$, *$\varphi$-model checking problems*. For an additionally specified bound $\beta \in \mathbb{N}$, we call problems deciding whether $\mathcal{A} \models^{\beta} \varphi(X)$, *$\varphi$-small witness problems*.

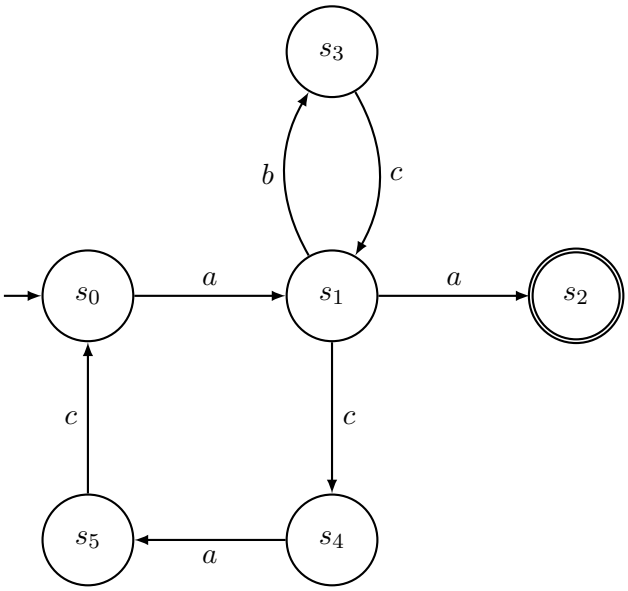
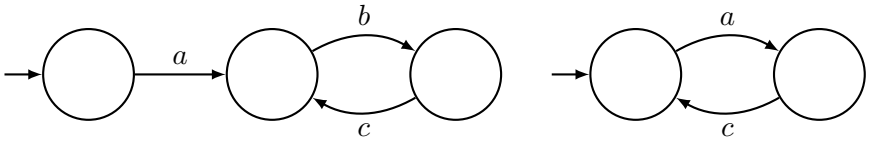


Figure 2.1: Example one-player game arena with initial position s_0 and bad position s_2.

Example 2.4.1. *Figure 2.1 shows an example one-player game arena with initial position s_0. Let s_2 be a dedicated bad position. The decisions are a, b, and c.*

A witness of size 3 for $\mathsf{EG}(\overline{\{s_2\}})$ is $a(bc)^\omega$ and is shown in Figure 2.2(a). However, the smallest witness is $(ac)^\omega$ and is shown in Figure 2.2(b).



(a) Witness of size 3. (b) Witness of size 2.

Figure 2.2: Witnesses for $\mathsf{EG}(\overline{\{s_2\}})$ for the game arena shown in Figure 2.1.

Assuming a two-player setting, for a view $\mathcal{V}$ for Player E, a bound $\beta \in \mathbb{N} \cup \{\infty\}$ on the size of the strategy for Player E, and a set of positions

$X \subseteq S$, we define

$$\begin{aligned}
&\mathcal{A} \models^{\beta}_{\mathcal{V}} \mathsf{Enforce}(X) \;:\Longleftrightarrow\\
&\qquad \exists f_e \in \mathcal{F}^{\mathsf{E}}_{\mathcal{A}} : f_e \models \mathcal{V} \wedge (\beta = \infty \vee |f_e| \leq \beta) \wedge\\
&\qquad\qquad \forall f_a \in \mathcal{F}^{\mathsf{A}}_{\mathcal{A}}\, \exists i \geq 1 : \mathsf{Pos}(\mathsf{Outcome}_{\mathcal{A}}(f_e, f_a)[1..i]) \in X;\\
&\mathcal{A} \models^{\beta}_{\mathcal{V}} \mathsf{Avoid}(X) \;:\Longleftrightarrow\\
&\qquad \exists f_e \in \mathcal{F}^{\mathsf{E}}_{\mathcal{A}} : f_e \models \mathcal{V} \wedge (\beta = \infty \vee |f_e| \leq \beta) \wedge\\
&\qquad\qquad \forall f_a \in \mathcal{F}^{\mathsf{A}}_{\mathcal{A}}\, \forall i \geq 1 : \mathsf{Pos}(\mathsf{Outcome}_{\mathcal{A}}(f_e, f_a)[1..i]) \notin X.
\end{aligned}$$

For a two-player arena $\mathcal{A}$, a view $\mathcal{V}$ for Player E, and a set of positions $X \subseteq S$, we call problems deciding whether $\mathcal{A} \models^{\infty}_{\mathcal{V}} \mathsf{Enforce}(X)$, *reachability synthesis problems*, and problems deciding whether $\mathcal{A} \models^{\infty}_{\mathcal{V}} \mathsf{Avoid}(X)$, *safety synthesis problems*. For an additionally specified bound $\beta \in \mathbb{N}$, we call problems deciding whether $\mathcal{A} \models^{\beta}_{\mathcal{V}} \mathsf{Enforce}(X)$ or $\mathcal{A} \models^{\beta}_{\mathcal{V}} \mathsf{Avoid}(X)$, *bounded safety synthesis problems* or *bounded reachability synthesis problems*, respectively.

Finally, a *safety game* $\mathcal{G}$ is a tuple $(\mathcal{A}, B, \mathcal{V}, \beta)$ consisting of

- a game arena $\mathcal{A} = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$,
- a set of bad or losing positions $B \subseteq S$,
- a view $\mathcal{V} = (\Sigma^{\mathsf{obs}}, \mathit{vis})$ for Player E on $\mathcal{A}$, and
- a bound $\beta \in \mathbb{N} \cup \{\infty\}$ on the size of the feasible strategies for Player E.

We say that $\mathcal{G}$ is finite iff $\mathcal{A}$ is finite. We say that Player E *wins* $\mathcal{G}$ iff $\mathcal{A} \models^{\beta}_{\mathcal{V}} \mathsf{Avoid}(B)$.

Example 2.4.2. *Figure 2.3 shows an example safety game with initial position s_0 and bad position s_2. The positions of the Players E and A are drawn as circles and squares, respectively. The decisions of E are e_1, e_2 and e_3, and the decisions of A are a_1 and a_2.*

Player A has a winning strategy (that enforces a play to s_2) from positions s_4 and s_5. However, assuming full observability, Player E can win the game by playing one of the strategies shown in Figures 2.4(a) and 2.4(b). Assuming a blindfold game, where Player E can only observe when it is her turn, represented by the decision τ, she can still win by playing the strategy shown in Figure 2.4(c).

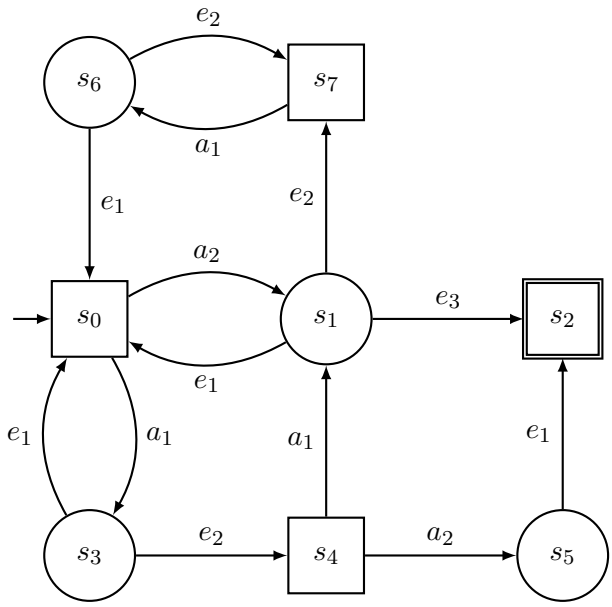
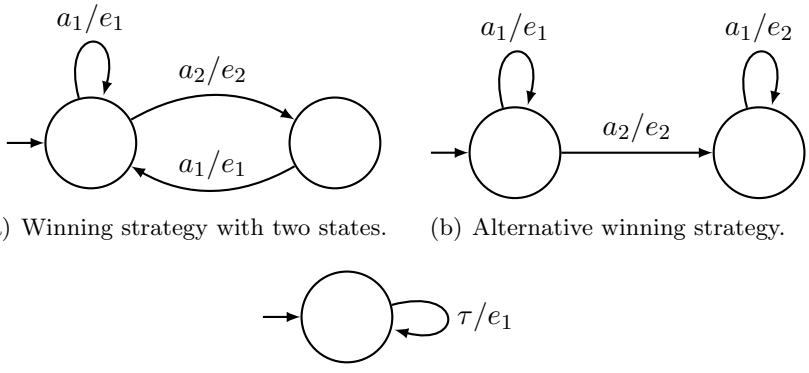


Figure 2.3: Example safety game with initial position s_0 and bad position s_2. The positions of the Players E and A are drawn as circles and squares, respectively.



(a) Winning strategy with two states. (b) Alternative winning strategy.

(c) Blindfold winning strategy.

Figure 2.4: Winning strategies for E represented as Mealy machines for the game shown in Figure 2.3.

Chapter 3

Succinctness Signatures and Sequential Circuit Machines

In this chapter, we develop our new universal computation model for succinct systems.

We start in Section 3.1 with a fundamental consideration how decision problems can be characterized according to their succinctness. Then, in Section 3.2, we formalize these considerations by introducing *succinctness signatures*, which serve as a uniform notion for comparing the succinctness of problems. Section 3.3 represents the main part of this chapter, where we introduce *sequential circuit machines* as the canonical computation model for succinctness signatures. In that section, we also define the *divergence problem*, the decision problem whether a dedicated state of a given sequential circuit machine is unreachable, as the canonical complete decision problem for a class of succinctness signatures. Section 3.4 gives sufficient conditions when a given safety game can be represented as a divergence problem. The chapter finishes with an overview on the related work in Section 3.5.

3.1 A Succinct View on Complexity

The purpose of the classical Turing machine model is to capture the essence of sequential computations that only access a constant (i.e., independent in the size of the input) number of state bits at each step. In succinct computations, on the other hand, the number of state bits being accessed at each step varies with the size of the input.

Hence, when going from Turing machines to a computation model that captures the essence of succinct computations, the question arises, whether the resource restrictions that were used for Turing machines to characterize complexity classes are still meaningful in the succinct case. For example, the running times (i.e., the number of steps) in the succinct world can be much smaller than the ones in the explicit world, as succinct computations can

modify many bits at once, which is in contrast to explicit computations. We thus arrive at the question, which restrictions are meaningful in the succinct world.

In this section, we give an answer to that question. It turns out that the expressivity of succinct computations highly depends on the *power that the existential and universal nondeterminism have at each step of the computation.*

We make this claim more precise by switching to a game-based interpretation: In a turn-based safety game between Player A and Player E, we identify the following *capabilities of a particular player* $p \in \{E, A\}$:

(1) *Degree of nondeterminism*: number of bits p can determine in each round of the game;

(2) *Precision of observability*: number of bits determined by $\overline{p}$ in each round that are observable by p;

(3) *Amount of memory*: number of bits p can use in each round to store information that can be recalled in later rounds.

By *capabilities of a game*, we refer to the capabilities of the two players. We can now state our (yet informal) central claim:

> *By seeing the instances of a decision problem* P *as games, we can classify* P *according to the capabilities of its games.*

One might ask the question, whether our selection of these three kinds of capabilities is reasonable. As we will demonstrate later in Section 4.2, this is indeed the case: We will prove that any decision problem that is complete for a major complexity class can be characterized by a certain class of capability-restricted games.

Moreover, as we will show in Chapters 5 and 7, it turns out that this classification also perfectly captures well-known modeling and specification formalisms such as concurrent state machines, linear-time temporal logic, one-safe Petri nets, or timed automata.

3.2 Succinctness Signatures

This section introduces basic notions that serve as basis for characterizing decision problems and complexity classes in terms of restrictions imposed on the capabilities of the existential and the universal nondeterminism.

The *instance signature* σ of some finite decision problem P and an input x for P, written as $\sigma = \mathit{sig}(\mathsf{P}, x)$, is represented by a tuple (N, M), where N characterizes the amount of (existential and universal) nondeterminism and M characterizes the amount of memory available to the two types of nondeterminism that are needed to solve P on x. More precisely, N is a tuple $(n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}})$, where

- $n_{\mathsf{E}} \in \mathbb{N}$ specifies the number of bits controlled by the existential nondeterminism in one step,
- $n_{\mathsf{A}} \in \mathbb{N}$ specifies the number of bits controlled by the universal nondeterminism in one step, and
- $n_{\mathsf{A}}^{\mathsf{E}} \in \mathbb{N} \cup \{\infty\}$ specifies the number of bits determined by the universal nondeterminism in one step that are visible to the existential nondeterminism.

The memory characterization M is a tuple $(m_{\mathsf{E}}, m_{\mathsf{A}})$, where

- $m_{\mathsf{E}} \in \mathbb{N} \cup \{\infty\}$ specifies the number of bits that can be used by the existential nondeterminism to store information that can be used later, and
- $m_{\mathsf{A}} \in \mathbb{N}$ specifies the number of bits that can be used by the universal nondeterminism to store information.

If $n_{\mathsf{A}}^{\mathsf{E}} = \infty$ then the existential nondeterminism can observe all decisions of the universal nondeterminism. If $m_{\mathsf{E}} = \infty$ then the existential nondeterminism has an unbounded amount of memory available. For an instance signature $\sigma = ((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (m_{\mathsf{E}}, m_{\mathsf{A}}))$, as a shorthand, we define

$$\log(\sigma) = \big((\lceil \log n_{\mathsf{E}} \rceil, \lceil \log n_{\mathsf{A}}^{\mathsf{E}} \rceil, \lceil \log n_{\mathsf{A}} \rceil), (\lceil \log m_{\mathsf{E}} \rceil, \lceil \log m_{\mathsf{A}} \rceil)\big) \text{ and}$$
$$O(\sigma) = \big((O(n_{\mathsf{E}}), O(n_{\mathsf{A}}^{\mathsf{E}}), O(n_{\mathsf{A}})), (O(m_{\mathsf{E}}), O(m_{\mathsf{A}}))\big),$$

where, by abuse of notation, we define $\lceil \log \infty \rceil = O(\infty) = \infty$.

A *succinctness signature* $\widehat{\sigma}$ is a tuple $((a, b, c), (d, e))$ comprising five nondecreasing functions $a, c, e : \mathbb{N} \to \mathbb{N}$ and $b, d \in (\mathbb{N} \to \mathbb{N}) \cup \{\infty\}$. The *restriction* of a decision problem P to $\widehat{\sigma}$, written as $\mathsf{P}_{\widehat{\sigma}}$, is defined as the decision problem P' that accepts only inputs x for which $sig(\mathsf{P}, x) = \widehat{\sigma}(n)$, where $n = |x|$ and $\widehat{\sigma}(n)$ is defined as $\big((a(n), b(n), c(n)), (d(n), e(n))\big)$.

A *signature class* is a set of succinctness signatures. For the definition of signature classes, we introduce the *$\mathcal{C}$-notation*, which is defined as follows. Let a, b, c, d, e represent each a class of functions, a constant, or ∞. Then, we write $\mathcal{C}((a, b, c), (d, e))$ to refer to the smallest set that contains all succinctness signatures that can be obtained by instantiating the individual functions. For example, when we write $\mathcal{C}((\mathcal{L}, \mathcal{P}, 42), (\infty, \mathcal{P}))$, we actually mean the signature class

$$\Big\{\big((O(\log n),\ n^{O(1)},\ 42),\ \ (\infty,\ n^{O(1)})\big)\Big\}.$$

For convenience, we define $\mathcal{L}_\infty = \mathcal{L} \cup \{\infty\}$ and $\mathcal{P}_\infty = \mathcal{P} \cup \{\infty\}$.

A signature class $\mathcal{C}$ induces a complexity class: we then implicitly refer to the set of binary encodings of those inputs x, for which there exists a decision

Notion	Characterizes	Example
Instance signature	Problem & input	"Is the node v unreachable from the initial node v_0 in the directed graph G?"
Succinctness signature	Problem	UNREACHABILITY for explicitly represented directed graphs
Signature class	Complexity class	$\mathcal{C}((\mathcal{L},0,0),(\infty,\mathcal{L}))$ NLOGSPACE

Table 3.1: Overview on the notions of signatures and what they characterize.

problem P and a succinctness signature $\widehat{\sigma} \in \mathcal{C}$ such that $sig(\mathsf{P}, x) = \widehat{\sigma}(|x|)$ and $\mathsf{P}(x) = \textbf{yes}$. In the rest of this thesis, whenever we relate a signature class to a (Turing machine-based) complexity class, we assume this implicit notion. Table 3.1 summarizes the notions of signatures introduced in this section.

The following example gives an intuition how succinctness signatures and decision problems correlate. In the remainder of this chapter, we will make the connection precise.

Example 3.2.1. *Recall the problem* UNREACHABILITY *from Example 2.1.1.*

For a given directed graph $G = (V, E)$, where E is represented as an adjacency matrix such that $|E| = |V|^2$, an initial node $v_0 \in V$, and a bad node $b \in V$, UNREACHABILITY *can be solved by iteratively letting the universal nondeterminism guess a sequence of adjacent nodes until b is reached starting in v_0, or to report that no such sequence exists [Jones, 1975].*

*Since we have no alternation and G is given explicitly (i.e., each node in V can be represented using a logarithmic number of bits), a candidate for an appropriate definition of $sig($*UNREACHABILITY*$, x)$, for a given input $x = ((V, E), v_0, b)$, is*

$$((0, 0, \lceil \log |E| \rceil), (0, \lceil \log |V| \rceil)) = ((0, 0, 2\lceil \log |V| \rceil), (0, \lceil \log |V| \rceil)).$$

We can hence conjecture that an appropriate succinctness signature for UNREACHABILITY *is $O((0, 0, \log n), (0, \log n))$. Furthermore, we can also conjecture that* UNREACHABILITY *is* complete *for $\mathcal{C}((0, 0, \mathcal{L}), (0, \mathcal{L}))$ (i.e., all problems in $\mathcal{C}((0, 0, \mathcal{L}), (0, \mathcal{L}))$ can be reduced to* UNREACHABILITY *and vice versa).*

In the next section, we will introduce the canonical computation model for succinctness signatures and formally define its notion of completeness.

3.3 Sequential Circuit Machines

In this section, we present *sequential circuit machines* as the canonical computation model for succinctness signatures. Just as the reachability (i.e., halting) problem for resource-bounded Turing machines characterizes complexity classes, we will use the co-reachability (i.e., divergence) problem for sequential circuit machines to characterize signature classes.

3.3.1 Syntax

A *sequential circuit machine* $\mathcal{S}$ is a tuple (σ, C_{A}) represented in unary, comprising an instance signature $\sigma = ((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (m_{\mathsf{E}}, m_{\mathsf{A}}))$ and a combinatorial circuit C_{A} such that

(1) C_{A} reads from the circuit elements

- $N_{\mathsf{A}} = \{\mathsf{NA}_i \mid 1 \leq i \leq n_{\mathsf{A}}\}$,
- $N_{\mathsf{E}} = \{\mathsf{NE}_i \mid 1 \leq i \leq n_{\mathsf{E}}\}$,
- $M_{\mathsf{A}} = \{\mathsf{MA}_i \mid 1 \leq i \leq m_{\mathsf{A}}\}$, and

(2) C_{A} writes to M_{A} and a dedicated output h signaling whether $\mathcal{S}$ reached a halting configuration.

We also call σ the *resources* of $\mathcal{S}$. Thus, C_{A} computes a Boolean function $C_{\mathsf{A}} : \vec{N}_{\mathsf{E}} \times \vec{N}_{\mathsf{A}} \times \vec{M}_{\mathsf{A}} \to \vec{M}_{\mathsf{A}} \times \{0,1\}$. The *size* of $\mathcal{S}$ is defined as $|\mathcal{S}| = |C_{\mathsf{A}}|$. We say that $\mathcal{S}$ *is of type* $\widehat{\sigma}$ if, and only if, for each $n \in \mathbb{N}$, we have that if $|C_{\mathsf{A}}| = n$ then $\sigma = \widehat{\sigma}(n)$ and $\mathit{depth}(C_{\mathsf{A}}) = O(1)$. The structure of a sequential circuit machine is shown in Figure 3.1.

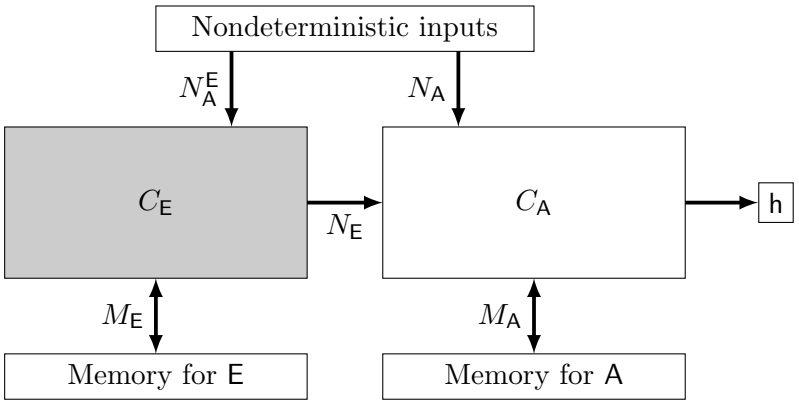


Figure 3.1: Structure of a sequential circuit machine.

3.3.2 Semantics

Intuitively, a machine $\mathcal{S} = (\sigma, C_{\mathsf{A}})$ is executed together with a combinatorial circuit C_{E} in a sequential manner. In each cycle of the execution, first, values for the inputs N_{A} are nondeterministically chosen. Then, C_{E} is executed and produces values for the circuit elements N_{E}, which, in turn, are read by C_{A} as inputs. The execution of C_{E} may only depend on the subset $N_{\mathsf{A}}^{\mathsf{E}}$ of N_{A}. After that, C_{A} is executed and produces a value for the output h. Both C_{E} and C_{A} are equipped with a read/write private memory M_{E} and M_{A}, respectively, from which they can recall information from earlier execution cycles.

We say that $\mathcal{S}$ *halts* iff there exists such a C_{E} with inputs $N_{\mathsf{A}}^{\mathsf{E}} \cup M_{\mathsf{E}}$ and outputs $N_{\mathsf{E}} \cup M_{\mathsf{E}}$ such that C_{A}, controlled by C_{E} via N_{E}, always reaches a halting configuration (i.e., a configuration where C_{A} computes a 1 for output h). Dually, we say that $\mathcal{S}$ *diverges* iff there exists a C_{E} that controls C_{A} such that no halting configuration is ever reached. In the following, we will formalize this informal description.

The semantics of a machine $\mathcal{S} = (((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (m_{\mathsf{E}}, m_{\mathsf{A}})), C_{\mathsf{A}})$ is formally defined as a finite game arena $[\![\mathcal{S}]\!] = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$, where

- $S_{\mathsf{E}} = \{\mathsf{E}\} \times \vec{M}_{\mathsf{A}} \times \vec{N}_{\mathsf{A}}$,
- $S_{\mathsf{A}} = \{\mathsf{A}\} \times \vec{M}_{\mathsf{A}} \times \{0, 1\}$,
- $s_0 = (\mathsf{A}, \vec{0}, 0)$,
- $\Sigma_{\mathsf{E}} = \vec{N}_{\mathsf{E}}$,
- $\Sigma_{\mathsf{A}} = \vec{N}_{\mathsf{A}}$,
- $\Delta((\mathsf{E}, (\vec{m_a}, \vec{n_a})), \vec{n_e}) = (\mathsf{A}, C_{\mathsf{A}}(\vec{n_e}, \vec{n_a}, \vec{m_a}))$, and
- $\Delta((\mathsf{A}, (\vec{m_a}, h)), \vec{n_a}) = (\mathsf{E}, (\vec{n_a}, \vec{m_a}))$.

The set of halting states is defined as

$$\mathsf{Halt}(\mathcal{S}) = \{\mathsf{A}\} \times \vec{M}_{\mathsf{A}} \times \{1\}.$$

3.3.3 Completeness

We now define the *divergence problem for sequential circuit machines* (i.e., deciding the unreachability of a configuration) as the canonical decision problem that characterizes completeness for signature classes. As we will show in Section 4.1 in more detail, we could have equivalently chosen the halting problem (i.e., deciding the reachability of a configuration) as the canonical decision problem. However, for the purpose of this thesis, regarding the satisfiability and synthesis problems for succinct formalisms, whose

complexity we are going to analyze in Chapters 5 and 7, choosing divergence over halting represents the more appropriate choice.

We define $\mathcal{S}' = \mathcal{S}[C_{\mathsf{E}} \leftarrow C, M_{\mathsf{E}} \leftarrow M]$ as the nonalternating sequential circuit machine that is obtained by replacing the existential nondeterminism in $\mathcal{S}$ with the concretely given combinatorial circuit C, which uses the private memory M. More precisely, $\mathcal{S}'$ is obtained by integrating C into C_{A} and M into M_{A} such that C deterministically generates values that are fed into the original C_{A} via the original N_{E}. Figure 3.2 shows such a composed machine.

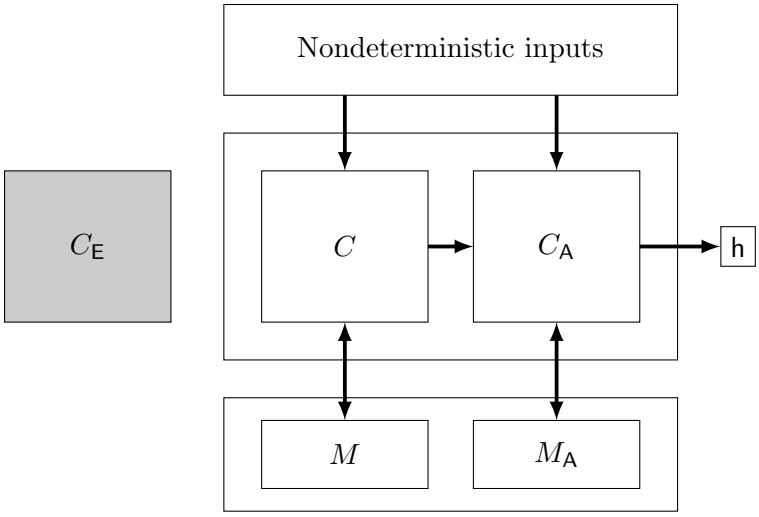


Figure 3.2: For a given sequential circuit machine $\mathcal{S}$, a circuit C with private memory M, the figure shows the structure of the composed sequential circuit machine $\mathcal{S}' = \mathcal{S}[C_{\mathsf{E}} \leftarrow C, M_{\mathsf{E}} \leftarrow M]$.

Definition 3.3.1. *For a sequential circuit machine $\mathcal{S}$ given in unary, the problem* DIVERGING *is to decide whether there exists a combinatorial circuit C with private memory M and $|M| \leq m_E$ such that $\mathcal{S}' = \mathcal{S}[C_E \leftarrow C, M_E \leftarrow M]$ and $[\![\mathcal{S}']\!] \models \mathsf{AG}(\overline{\mathsf{Halt}(\mathcal{S}')})$.*

Theorem 3.3.2. *For a sequential circuit machine $\mathcal{S} = (\sigma, C_A)$ given in unary with $\sigma = ((n_E, n_A^E, n_A), (m_E, m_A))$,* DIVERGING$(\mathcal{S})$ = **yes** *iff Player E wins $\big([\![\mathcal{S}]\!], \mathsf{Halt}(\mathcal{S}), \mathcal{V}_{\mathcal{S}}, m_E\big)$, where $\mathcal{V}_{\mathcal{S}} = (\vec{N_A^E}, vis)$ with $vis(\vec{n_a}) := \vec{n_a}(N_A^E)$.*

Proof. We show that a winning strategy f_{E} for Player E can be transformed into a circuit C with private memory M and $|M| \leq m_{\mathsf{E}}$ such that $\mathcal{S}' = \mathcal{S}[C_{\mathsf{E}} \leftarrow C, M_{\mathsf{E}} \leftarrow M]$ and $[\![\mathcal{S}']\!] \models \mathsf{AG}(\overline{\mathsf{Halt}(\mathcal{S}')})$, and vice versa. The rest of the claim then easily follow from the definition of (the constituent components of) $\big([\![\mathcal{S}]\!], \mathsf{Halt}(\mathcal{S}), \mathcal{V}_{\mathcal{S}}, m_{\mathsf{E}}\big)$.

Assume we have a winning strategy $f_{\mathsf{E}} \models \mathcal{V}_{\mathcal{S}}$ for Player E with existential memory bound m_{E}. Let $(Q, q_0, \Sigma_{\mathsf{A}}, \Sigma_{\mathsf{E}}, T)$ be the Mealy machine representing f_{E} with $\Sigma_{\mathsf{A}} = \vec{N}_{\mathsf{A}}^{\mathsf{E}}$ and $\Sigma_{\mathsf{E}} = \vec{N}_{\mathsf{E}}$. If $m_{\mathsf{E}} \neq \infty$, we have $|Q| \leq 2^{m_{\mathsf{E}}}$. We choose M to be the logarithmic encoding of Q. Then, we can construct C as a composition of Boolean functions each determining the value of a particular bit of M and the output of T depending on the bits encoding the current state and the input representing the current observation. Since f_{E} respects the (potentially partial) view on $[\![\mathcal{S}]\!]$, it is safe to ignore all unobservable bits in the construction of C.

For the other direction, we assume that there is a circuit C with private memory M and $|M| \leq m_{\mathsf{E}}$ such that $\mathcal{S}' = \mathcal{S}[C_{\mathsf{E}} \leftarrow C, M_{\mathsf{E}} \leftarrow M]$ and $[\![\mathcal{S}']\!] \models \mathsf{AG}(\overline{\mathsf{Halt}(\mathcal{S}')})$. Based on that, a winning strategy for E is implicitly given by the truth table of C seen as a Boolean function. □

For any given signature class $\widehat{\sigma}$, we define Diverging, restricted to machines of type $\widehat{\sigma}$, to be the most general problem for that signature class.

Definition 3.3.3. *A decision problem* P is complete for *a signature class* $\mathcal{C}$ *if, and only if, for every succinctness signature* $\widehat{\sigma}$ *in* $\mathcal{C}$, Diverging$_{\widehat{\sigma}}$ *and* P *are* LogSpace*-reducible to each other.*

Recall that Diverging$_{\widehat{\sigma}}$ is the restriction of Diverging to machines of type $\widehat{\sigma}$.

Example 3.3.4. Unreachability *from Example 2.1.1 is complete for* $\mathcal{C}((0,0,\mathcal{L}),(0,\mathcal{L}))$*:*

For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((0,0,\mathcal{L}),(0,\mathcal{L}))$*, the reduction from* Diverging$_{\widehat{\sigma}}$ *to* Unreachability *is just the application of* Unreachability *on the explicitly-represented directed graph (which is of polynomial size) that is given by the semantics (i.e., the single-player game arena) of the input of* Diverging*.*

The reduction from Unreachability *to* Diverging$_{\widehat{\sigma}}$ *follows directly from the fact that one can easily encode Jone's Algorithm [Jones, 1975] as a sequential circuit machine with a logarithmic amount of universal memory and universal nondeterminism. For this purpose, observe that one can encode the explicitly given edge relation of the directed graph, which is the input to* Unreachability*, as a DNF resulting in a combinatorial circuit of polynomial size and constant depth (recall that our circuits may have an unbounded fan-in).*

3.4 Succinct Circuit Representations

This section provides a technique to transform finite safety games with certain properties into sequential circuit machines. It will prove useful in establishing upper complexity bounds.

A finite safety game $(\mathcal{A}, B, \mathcal{V}, \beta)$ *allows a succinct circuit representation*, where $\mathcal{A} = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$ and $\mathcal{V} = (\Sigma^{\mathsf{obs}}, \mathit{vis})$, if it satisfies the following properties:

(1) There is a logarithmic encoding $(\!|\cdot|\!)$ of the positions S_{E} and S_{A}, and the decisions Σ_{E} and Σ_{A}.

(2) The move function Δ can be represented as a simple combinatorial circuit representing a function $\Delta^c : (\!|S|\!) \times (\!|\Sigma|\!) \to \{0,1\} \times (\!|S|\!)$ such that for any $s, s' \in S_p$ and $a \in \Sigma_p$, Δ^c outputs $(1, (\!|s'|\!))$ if $\Delta(s,a)$ is defined to be s', or Δ^c outputs $(0, \vec{x})$ for some arbitrary valuation $\vec{x}$, otherwise. We furthermore require that the size of Δ^c is at most polynomial in $|\Sigma| \cdot \log|S|$ while its depth is constant.

(3) B can be equivalently described as a simple combinatorial circuit $B^c : (\!|S|\!) \to \{0,1\}$ such that, for any $s \in S$, $B^c((\!|s|\!)) = 1$ iff $s \in B$. We furthermore require that the size of B^c is at most polynomial in $\log|S|$ while its depth is constant.

(4) $\mathcal{V}$ can be equivalently described as a subset of the variables (i.e. the bits) of $\widehat{(\!|\Sigma_{\mathsf{A}}|\!)}$.

As we will see later in Chapters 5 and 7 of this thesis, many well-known modeling formalisms, whose semantics is defined in terms of game arenas, indeed allow succinct circuit representations.

The following lemma will prove useful in establishing that a succinctly representable finite safety game can always be transformed into a sequential circuit machine of polynomial size and logarithmic resources.

Lemma 3.4.1. *For a finite safety game $\mathcal{G} = (\mathcal{A}, B, \mathcal{V}, \beta)$ that allows a succinct circuit representation, where $\mathcal{A} = (S_{\mathsf{A}}, S_{\mathsf{E}}, s_0, \Sigma_{\mathsf{A}}, \Sigma_{\mathsf{E}}, \Delta)$ and $\mathcal{V} = (\Sigma^{\mathsf{obs}}, \mathit{vis})$, one can construct in* $\textsc{Space}(\log|\mathcal{G}|)$ *a sequential circuit machine $\mathcal{S} = (\sigma, C_{\mathsf{A}})$ such that $|\mathcal{S}|$ is polynomial in $|\mathcal{G}|$,*

$$\sigma = \big((\lceil\log|\Sigma_{\mathsf{A}}|\rceil, \lceil\log|\Sigma^{\mathsf{obs}}|\rceil, \lceil\log|\Sigma_{\mathsf{A}}|\rceil),\ (\beta, O(\log|S|))\big),$$

and Player E wins $\mathcal{G}$ iff $\textsc{Diverging}(\mathcal{S}) = \mathbf{yes}$.

Proof. We construct $\mathcal{S} = (((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (m_{\mathsf{E}}, m_{\mathsf{A}})), C_{\mathsf{A}})$ in the following way. The existential and universal choices in $\mathcal{A}$ correspond to the existential and universal guessing bits in $\mathcal{S}$, i.e., $n_{\mathsf{E}} = |\widehat{(\!|\Sigma_{\mathsf{E}}|\!)}| = \lceil\log|\Sigma_{\mathsf{E}}|\rceil$ and $n_{\mathsf{A}} = |\widehat{(\!|\Sigma_{\mathsf{A}}|\!)}| = \lceil\log|\Sigma_{\mathsf{A}}|\rceil$. In case that Player E has partial (or no) observability we choose $n_{\mathsf{A}}^{\mathsf{E}} = |\widehat{(\!|\Sigma^{\mathsf{obs}}|\!)}|$, otherwise we choose $n_{\mathsf{A}}^{\mathsf{E}} = |\widehat{(\!|\Sigma_{\mathsf{A}}|\!)}|$. Here, recall that, by definition, $\mathcal{V}$ can be represented as a subset of the bits for representing a decision of A. We use the universal memory of $\mathcal{S}$ to represent the current game position in $S = S_{\mathsf{E}} \cup S_{\mathsf{A}}$, as well as a state counter that ranges

from 0 to $|S|-1$, i.e., $m_{\mathsf{A}} = |\widehat{[\![S]\!]}| + \lceil \log |S| \rceil = O(\log |S|)$. Furthermore, we choose $m_{\mathsf{E}} = \beta$.

We define C_{A} as follows. Recall that, by definition, the move function Δ of $\mathcal{A}$ can also be represented by a Boolean function Δ^c. We embed two versions of Δ^c in C_{A}: a sub-circuit Δ^c_{E} that reads from N_{E}, and a sub-circuit Δ^c_{A} that reads from N_{A}. Both sub-circuits read from M_{A}. Now, assuming that the current game position is stored in M_{A} and the existential and the universal nondeterminism determined their choices for N_{E} and N_{A}, respectively, C_{A} computes the next position by executing Δ^c_{E} and Δ^c_{A} (concurrently). Depending which valid bit is 1 (either the first output of Δ^c_{E} or the first output of Δ^c_{A}), a multiplexer selects the corresponding next state that is written into M_{A}.

Since sequential circuit machines have no possibility to model stuttering steps directly, with which Player A can execute arbitrarily many hidden moves before the turn changes to Player E again, in our construction of $\mathcal{S}$ we need to model them in a different way. The idea is to let Player A execute $|S|$ *obfuscating moves* in each round of the game before Player E can play a responding move. In an obfuscating move, Player A can either execute a move $d \in \Sigma_{\mathsf{A}}$ for which $\mathit{vis}(d) = \varepsilon$, or a stuttering move that does not change the contents of M_{A}. After the execution of the obfuscating moves (i.e., whenever the state counter overflows), A plays an ordinary move from Σ_{A} again. Observe that on each obfuscating move Player E (i.e., C_{E}) is activated but does not obtain any other information. Since the number of these activations is always $|S|$, Player E's strategy is indeed independent of the ε moves.

What remains is to let C_{A} produce $\mathsf{h} = 1$ whenever the current position is in B. This can be achieved by embedding B^c into C_{A} such that B^c reads from M_{A} and outputs h.

Overall, observe that C_{A} can be realized as a simple circuit of at most polynomial size and constant depth (even if $|S|$ is exponential), since we only use incrementers, comparators, multiplexers, and the simple circuits Δ^c and B^c, which have a polynomial size and constant depth by definition. □

3.5 Bibliographic Remarks

Sequential circuit machines can be seen as a continuation of several research directions, each investigating a particular aspect of computational complexity. While some aspects were introduced and analyzed individually, to the best of the author's knowledge, the combination of all in one uniform computation model has not been investigated yet. In the following, we give an overview on important works that are related to sequential circuit machines.

Turing machine extensions. Savitch [1977] extended Turing machines by allowing recursive calls. The introduction of a universal nondeterminism is due to Chandra et al. [1981] resulting in the *alternating Turing machine*. Reif [1984] introduced *private* and *blindfold Turing machines* and showed that there is a general exponential increase in complexity in the presence of *partial information*.

The impact of restricting the memory of the existential nondeterminism (differently to sequential circuit machines, though) was investigated by Cai and Furst [1991] by introducing *bottleneck Turing machines*, which extend ordinary Turing machines by an additional read/write "safe storage" to store a bounded amount of information. The execution of bottleneck Turing machines occurs in rounds. In each round, the machine can make polynomially many steps. From one round to the next, the machine can only retain the information in the safe storage.

Parallel computation models. Giving *sequential random access machines* (which are equivalent to Turing machines) the capability of accessing a shared memory in parallel results in *parallel random access machines* (PRAMs) [Fortune and Wyllie, 1978, Goldschlager, 1978, Savitch and Stimson, 1979]. PRAMs and sequential circuit machine share the ability to access many bits in one step. In the 1970s and 1980s, various PRAM models were proposed in the literature. Similar to alternating Turing machines, all these models have in common that they satisfy the *Parallel Computation Thesis*:

> *Parallel running time corresponds to sequential space consumption.*

A prominent PRAM representative is the *vector processing machine* introduced by Pratt and Stockmeyer [1976]. Vector processing machines are equipped with a collection of parallel processors together with a collection of registers that can hold bit vectors. All processors execute the same program whose instruction set consists of binary operations on the registers.

Savitch [1982] investigated the power of LPRAMs, parallel random access machines with instructions for string manipulation. It was observed that, in general, there is an exponential blow-up in the running times, when going from nondeterministic LPRAMs to nondeterministic Turing machines. Stockmeyer and Vishkin [1984] established the connection between PRAMs and (purely combinatorial, i.e., memoryless) Boolean circuits: Parallel time and number of processors for PRAMs correspond to depth and size for circuits, respectively. We refer to van Emde Boas [1990] for a comprehensive survey on sequential and parallel machine models.

Uniform circuit complexity. For characterizing complexity classes within PTIME, complexity classes based on the computational power of *com-*

binatorial Boolean circuits have been introduced. Borodin [1977] discovered the relation between space in Turing machines and depth in circuits.

Pippenger [1979] and Cook [1979] discovered the NC-hierarchy, comprising all complexity classes that are characterized by circuits with bounded fan-in, polynomial size, and polylogarithmic depth. The connection between NC and alternating Turing machines was shown by Ruzzo [1981]. Wolf [1994] investigated NC circuits with nondeterministic gates. We refer to Vollmer [1999] for a comprehensive survey on circuit complexity.

Succinct circuit representations. The research on the complexity of succinct versions of graph-theoretic problems was started in the early 1980s by Galperin and Wigderson [1983], where the exponential increase in the complexity was first observed. A more general theorem for lifting reductions (and therefore hardness proofs) from NPTime to NExpTime was given by Papadimitriou and Yannakakis [1986]. The generalization to arbitrary space- and time-bounded complexity classes was due to Lozano and Balcázar [1989].

While these works assumed the graph to be represented as a combinatorial circuit, Veith [1997] proved that the lifting argument (and thus the exponential increase in complexity) still holds even when the graph is given as a Boolean function. Similarly, Feigenbaum et al. [1999] proved that an OBDD-based representation of the graph suffices to cause an exponential blow-up in the complexity of both the nondeterministic and the alternating reachability problem. Borchert and Lozano [1996] made the connection between succinct circuit representations and *leaf language classes*.

A first connection between a succinct modeling formalism (specifically, Boolean automata) and the general lifting technique by Lozano and Balcázar [1989] was made by Chadha et al. [2010], who gave an explanation in terms of succinct circuit representations, why certain reachability-related problems on Boolean automata suffer from an exponential blow-up.

Descriptive complexity. In contrast to machine-based characterizations of complexity classes, a fundamentally different approach represents *descriptive complexity theory*, which was started by Ronald Fagin in the 1970s. Here, a complexity class is characterized by the expressiveness of a certain logic needed to specify the problems in that class. In a first paper, Fagin [1974] established the equivalence of NPTime and the set of problems describable in second-order existential logic. Characterizations of other major complexity classes followed such as PSpace [Immerman, 1980, Vardi, 1982], or NLogSpace and PTime [Grädel, 1992]. We refer to Immerman [1999] for a comprehensive survey on descriptive complexity theory.

Also from a pure logical perspective, Gottlob et al. [1999] showed that the expression complexity [Vardi, 1982] of logics with the power of replacing

inputs by Boolean combinations of inputs is generally exponentially harder than their data complexity.

Chapter 4

The Computational Power of Sequential Circuit Machines

This chapter investigates the impact of imposing structural restrictions on sequential circuit machines on their computational power.

Section 4.1 shows reductions between different classes of structurally restricted sequential circuit machines. Then, Section 4.1 draws the connection to Turing machines and identifies important structural restrictions to obtain a wide range of complexity classes. The chapter finishes with an overview on the results and a discussion in Section 4.3.

4.1 Reductions between Sequential Circuit Machines

In this section, we will prove some reductions between different classes of structurally restricted sequential circuit machines, which will turn out useful later in the thesis.

The first lemma ensures that one can always trade nondeterminism for the same amount of universal space.

Lemma 4.1.1. *Any given sequential circuit machine $\mathcal{S} = (\sigma, C_A)$, where σ is of the form*

$$\begin{gathered}\big((n_E, 0, 0),\ (m_E, m_A)\big),\\ \big((0, 0, n_A),\ (0, m_A)\big),\\ \big((n_E, 0, n_A),\ (m_E, m_A)\big),\\ \big((n_E, \infty, n_A),\ (m_E, m_A)\big),\ \textit{or}\\ \big((n_E, n_A^E, n_A),\ (m_E, m_A)\big),\end{gathered}$$

for $n_E, n_A \in \mathbb{N}_{>1}$, $n_A^E < n_A$, $m_E \in \mathbb{N} \cup \{\infty\}$, and $m_A \in \mathbb{N}$, can be transformed in SPACE$(\log |\mathcal{S}|)$ *into a sequential circuit machine $\mathcal{S}' = (\sigma', C_A{}')$ such that*

$\mathcal{S}$ diverges iff $\mathcal{S}'$ diverges, $|\mathcal{S}'| = O(|\mathcal{S}|)$, and σ' is of the form

$$\begin{aligned}
&\big((1,0,0),\,(m_E + O(\log n_E), m_A + O(n_E))\big),\\
&\big((0,0,1),\,(0, m_A + O(n_A))\big),\\
&\big((1,0,1),\,(m_E + O(\log n_E + \log n_A), m_A + O(n_E + n_A))\big),\\
&\big((1,1,1),\,(m_E + O(\log n_E + \log n_A), m_A + O(n_E + n_A))\big),\ \textit{or}\\
&\big((1,1,2),\,(m_E + O(\log n_E + \log n_A), m_A + O(n_E + n_A))\big),
\end{aligned}$$

respectively.

Proof. We construct $\mathcal{S}'$ as an extension of $\mathcal{S}$: Instead of letting the two types of nondeterminism determine the bits at once, we let the determination of the bits performed step-by-step. For this purpose, we have to (1) store the guessed bits in the universal memory, and (2) introduce counters to control the iterations. Note that, in the blindfold and private cases, no new information is leaked to C_{E} due to the newly introduced iterations in C_{A}, because the number of those iterations is always the same in each step. Also note that, in case of bounded existential memory, we need to give C_{E} the possibility to accommodate bits for counters used for iterating over the universal and existential choices. Overall, we need to introduce further $n_{\mathsf{E}} + n_{\mathsf{A}} + \lceil \log(n_{\mathsf{E}} + 1) \rceil + \lceil \log(n_{\mathsf{A}} + 1) \rceil$ universal memory bits, and the size of ${C_{\mathsf{A}}}'$ increases polynomially, while its depth remains constant, as we only introduce incrementers. □

With regard to the definition of the notion of *acceptance* for sequential circuit machines, we now show that whether choosing halting or diverging is not so important, as one can always obtain an equivalent machine with the dual acceptance at the expense of increasing the universal memory.

Lemma 4.1.2. *For every sequential circuit machine $\mathcal{S} = (((n_E, n_A^E, n_A), (m_E, m_A)), C_A)$, one can construct in* Space$(\log|\mathcal{S}|)$ *a sequential circuit machine $\mathcal{S}' = (((n_E, n_A^E, n_A), (m_E, {m_A}')), {C_A}')$ with $|{C_A}'| = O(|C_A|)$ and ${m_A}' = O(m_A)$ such that $\mathcal{S}$ diverges iff $\mathcal{S}'$ halts.*

Proof. If $m_{\mathsf{A}} = 0$, it suffices to choose ${m_{\mathsf{A}}}' = 0$ and let ${C_{\mathsf{A}}}'$ just invert the output h of C_{A}.

If $m_{\mathsf{A}} > 0$, we introduce a counter in $\mathcal{S}'$ with m_{A} bits. This counter deterministically (i.e., independently of the decisions of the two players) counts the number of execution cycles. Observe that there can be at most $2^{m_{\mathsf{A}}} - 1$ cycles between two equivalent configurations of C_{A}. Now, those configurations of $\mathcal{S}'$ in which the cycle counter overflows are those in which a configuration of $\mathcal{S}$ recurs. Hence, we can identify these configurations as diverging in $\mathcal{S}$ and, consequently, mark the corresponding configurations in $\mathcal{S}'$ as halting by letting ${C_{\mathsf{A}}}'$ produce an output $\mathsf{h} = 1$.

What remains is to convert the halting configurations in $\mathcal{S}$ to diverging ones in $\mathcal{S}'$. We achieve this by introducing an additional flag *div* that, once set, prevents the halting flag h of $\mathcal{S}'$ ever to become 1. When executing a cycle in $\mathcal{S}'$, ${C_\mathsf{A}}'$ sets *div* to 1 if either *div* was 1 in the previous cycle or C_A computed $\mathsf{h} = 1$. □

4.2 Reductions from and to Turing Machines

In this section, we investigate the relationship between the computational power of sequential circuit machines and Turing machines. We thereby flesh out our claim from Section 3.1 that any major complexity class can be characterized by a certain signature class.

4.2.1 Machines without Universal Memory

We start with the case, where sequential circuit machines have no universal memory. That is, the universal nondeterminism cannot accumulate any information over multiple cycles.

The first theorem establishes the connection to nondeterministic time-bounded complexity classes.

Theorem 4.2.1. *The following equivalences hold, even when we only assume simple circuits:*

$$\mathcal{C}((\mathcal{P},0,0),(0,0)) = \mathcal{C}((\mathcal{P},0,0),(\infty,0)) = \text{NPTime};$$
$$\mathcal{C}((0,0,\mathcal{P}),(0,0)) = \mathcal{C}((0,0,\mathcal{P}),(\infty,0)) = \text{coNPTime}$$

Proof. For proving the "$\subseteq$" direction, note that in all cases, only a single execution of C_A needs to be considered. This is because A cannot persist any information, and therefore, he finds himself always in the same configuration after executing C_A. For the same reason, E does not gain additional power by giving her more memory. Hence, by applying the definition of the semantics of sequential circuit machines, checking whether a given machine diverges that has a polynomial amount of (either existential or universal) nondeterminism but no universal memory, amounts to checking the satisfiability of the formulas

$$\exists \vec{N_\mathsf{E}} : C_\mathsf{A}(\vec{N_\mathsf{E}}) = 0 \text{ and} \tag{4.1}$$
$$\forall \vec{N_\mathsf{A}} : C_\mathsf{A}(\vec{N_\mathsf{A}}) = 0 \;\Leftrightarrow\; \neg(\exists \vec{N_\mathsf{A}} : C_\mathsf{A}(\vec{N_\mathsf{A}}) = 1). \tag{4.2}$$

Formula 4.1 can be decided by a simple nondeterministic algorithm that (1) first guesses values for $\vec{N_\mathsf{E}}$ in NPTime and (2) then validates the guess by evaluating $C_\mathsf{A}(\vec{N_\mathsf{E}}) = 0$. Formula 4.2 can be decided by a simple co-nondeterministic algorithm that (1) first guesses values for $\vec{N_\mathsf{A}}$ in NPTime, (2) then validates the guess by evaluating $C_\mathsf{A}(\vec{N_\mathsf{A}}) = 1$, and (3) negating

the overall result. Since computing the value of a circuit can be done in PTIME [Ladner, 1975] (so certainly in NPTIME or coNPTIME), we conclude that the two algorithms have an overall complexity of NPTIME and coNPTIME, respectively.

The "$\supseteq$" direction immediately follows by reduction from Boolean SAT and TAUTOLOGY, which are known to be hard for NPTIME and coNPTIME, respectively. For a given instance of SAT or TAUTOLOGY, i.e., a Boolean formula φ with free variables $\vec{x}$, we construct C_{A} such that $\mathsf{h} = 1$ iff $\neg\varphi(\vec{x})$. Now, in the case of SAT, we identify the free variables $\vec{x}$ with $\vec{N}_{\mathsf{E}}$, and in the case of TAUTOLOGY, we identify $\vec{x}$ with $\vec{N}_{\mathsf{A}}$. This way, by definition of the semantics of sequential circuit machines, the constructed machine diverges iff φ is satisfiable (in case of SAT) or φ is valid (in case of TAUTOLOGY). □

It turns out that when both the existential and universal nondeterminism can determine the values of some bits, we arrive in the polynomial hierarchy.

Theorem 4.2.2. *The following equivalences hold, even when we only assume simple circuits:*

$$\begin{aligned}
\mathcal{C}((\mathcal{P},0,\mathcal{P}),(0,0)) &= \mathcal{C}((\mathcal{P},0,\mathcal{P}),(\infty,0)) = \Sigma_2^P;\\
\mathcal{C}((\mathcal{P},\mathcal{P},\mathcal{P}),(0,0)) &= \mathcal{C}((\mathcal{P},\mathcal{P},\mathcal{P}),(\infty,0)) = \Pi_3^P;\\
\mathcal{C}((\mathcal{P},\infty,\mathcal{P}),(0,0)) &= \mathcal{C}((\mathcal{P},\infty,\mathcal{P}),(\infty,0)) = \Pi_2^P
\end{aligned}$$

Proof. For proving the "$\subseteq$" direction, similar to the proof for Theorem 4.2.1, observe that when having no universal memory, only a single execution of C_{A} needs to be considered and giving more memory to E does not make a difference. But now, by definition of the semantics of sequential circuit machines, the divergence problem for machines with both existential and universal nondeterminism, but without universal memory, corresponds to the satisfiability problem of Boolean formulas with more than one quantifier alternation:

$$\exists\vec{N_{\mathsf{E}}}\ \forall\vec{N_{\mathsf{A}}} : C_{\mathsf{A}}(\vec{N_{\mathsf{E}}},\vec{N_{\mathsf{A}}}) = 0; \tag{4.3}$$

$$\forall\vec{N_{\mathsf{A}}^{\mathsf{E}}}\ \exists\vec{N_{\mathsf{E}}}\ \forall\vec{N_{\mathsf{A}}} : C_{\mathsf{A}}(\vec{N_{\mathsf{E}}},\vec{N_{\mathsf{A}}^{\mathsf{E}}},\vec{N_{\mathsf{A}}}) = 0; \tag{4.4}$$

$$\forall\vec{N_{\mathsf{A}}}\ \exists\vec{N_{\mathsf{E}}} : C_{\mathsf{A}}(\vec{N_{\mathsf{E}}},\vec{N_{\mathsf{A}}}) = 0. \tag{4.5}$$

Formula 4.3 reflects the fact that E has no observability: she has to make her decisions (i.e., the values for $\vec{N_{\mathsf{E}}}$) independently from the decisions of A (i.e., the values for $\vec{N_{\mathsf{A}}}$). Formula 4.4 reflects the fact that E has only a partial observability: A has to provide values for the visible decisions first (i.e., the values for $\vec{N_{\mathsf{A}}^{\mathsf{E}}}$), before E can react by providing her decisions (i.e., the values for $\vec{N_{\mathsf{E}}}$), which must be independent of the unobservable decisions of A. Lastly, Formula 4.5 reflects the fact that E has full observability: A

has to make all of his decisions first (i.e., the values for $\vec{N}_{\mathsf{A}}$) before E can react by providing her decisions (i.e., the values for $\vec{N}_{\mathsf{E}}$).

Now, according to Wrathall [1976], deciding the satisfiability of the three quantified Boolean formulas (with bounded alternation) can be done in
(1) $\text{NP}\textsc{time}^{\text{coNP}\textsc{time}} = \Sigma_2^P$ for the blindfold case,
(2) $\text{coNP}\textsc{time}^{\text{NP}\textsc{time}^{\text{coNP}\textsc{time}}} = \Pi_3^P$ for the partially observable case, and
(3) $\text{coNP}\textsc{time}^{\text{NP}\textsc{time}} = \Pi_2^P$ for the fully observable case.
The "$\supseteq$" direction for the three cases follows by reduction from $\text{QBF}_2^{\exists}$, $\text{QBF}_3^{\forall}$, and $\text{QBF}_2^{\forall}$, which, according to Wrathall [1976], are known to be hard for Σ_2^P, Π_3^P, and Π_2^P, respectively (here, QBF_k^Q refers to the quantified Boolean formulas with k alternations and Q as the first quantifier). For a given QBF formula of the form

$$\forall x_0, \ldots, x_a \, \exists y_0, \ldots, y_b \, \forall z_0, \ldots, z_c : \\ \varphi(x_0, \ldots, x_a, y_0, \ldots, y_b, z_0, \ldots, z_c),$$

in our construction of a machine $\mathcal{S} = (((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (0,0)), C_{\mathsf{A}})$, we define $N_{\mathsf{A}}^{\mathsf{E}} = \{x_0, \ldots, x_a\}$, $N_{\mathsf{E}} = \{y_0, \ldots, y_b\}$, and $N_{\mathsf{A}} = \{z_0, \ldots, z_c\}$. Similar to the construction in the hardness proof of Theorem 4.2.1, we construct C_{A} such that $\mathsf{h} = 1 \Leftrightarrow \neg\varphi(x_0, \ldots, x_a, y_0, \ldots, y_b, z_0, \ldots, z_c)$. □

4.2.2 Machines with Unbounded Existential Memory

We now investigate the computational power of sequential circuit machines with no restrictions on the memory of the existential nondeterminism.

Before we come to the main theorems, we first state the following two lemmas, which will turn out to be useful. The first lemma is helpful in establishing upper complexity bounds.

Lemma 4.2.3. *For a given sequential circuit machine $\mathcal{S} = (\sigma, C_{\mathcal{A}})$, the following statements hold true:*

if $\sigma = ((0,0,0),(\infty, m_{\mathcal{A}}))$ *then*
$$\textsc{Diverging}(\mathcal{S}) \in \textsc{Space}(m_{\mathcal{A}})^{\textsc{Eval}(C_{\mathcal{A}})};$$
if $\sigma = ((1,0,0),(\infty, m_{\mathcal{A}}))$ *then*
$$\textsc{Diverging}(\mathcal{S}) \in \textsc{NSpace}(m_{\mathcal{A}})^{\textsc{Eval}(C_{\mathcal{A}})};$$
if $\sigma = ((0,0,1),(\infty, m_{\mathcal{A}}))$ *then*
$$\textsc{Diverging}(\mathcal{S}) \in \textsc{coNSpace}(m_{\mathcal{A}})^{\textsc{Eval}(C_{\mathcal{A}})};$$
if $\sigma = ((1,1,1),(\infty, m_{\mathcal{A}}))$ *then*
$$\textsc{Diverging}(\mathcal{S}) \in \textsc{ASpace}(m_{\mathcal{A}})^{\textsc{Eval}(C_{\mathcal{A}})}$$

Proof. We first prove that if $\sigma = ((1,1,1),(\infty, m_{\mathsf{A}}))$ then we have that $\textsc{Diverging}(\mathcal{S})$ is in $\textsc{ASpace}(m_{\mathsf{A}})^{\textsc{Eval}(C_{\mathsf{A}})}$. After that, we show how the construction can be adapted to the other three (nonalternating) cases.

For a given machine $\mathcal{S} = (((1,1,1),(\infty,m_{\mathsf{A}})),C_{\mathsf{A}})$, we construct an alternating Turing machine $\mathcal{T} = (Q, q_0, T, \delta)$ that uses at most $O(m_{\mathsf{A}})$ tape cells and that accepts iff $\mathcal{S}$ diverges. The basic idea of our construction is that $\mathcal{T}$ simulates the cycles of $\mathcal{S}$ in a sequential manner. Concurrently, $\mathcal{T}$ counts the cycles using a counter with m_{A} bits. Whenever this counter overflows, which happens after $2^{m_{\mathsf{A}}}$ cycles, we let $\mathcal{T}$ enter its accepting state. We use m_{A} tape cells to store the current contents of the universal memory.

In each cycle, first $\mathcal{T}$ makes a universal guess (0 or 1) and an existential guess (0 or 1), which are stored (as 2 bits) on the tape. Then, C_{A} is evaluated on these bits and on the universal memory contents. For each bit in the universal memory and the halting flag h, we obtain a new value by making $m_{\mathsf{A}} + 1$ oracle calls to $\textsc{Eval}(C_{\mathsf{A}})$. The result of the evaluation is the new contents of the universal memory, which is stored in another m_{A} bits on the tape, as well as the value of h. If $\mathsf{h} = 1$, we let $\mathcal{T}$ enter a dead-end state, where the cycle counter is not incremented anymore, and thus, the accepting state is unreachable. Note that, if always $\mathsf{h} = 0$, the cycle counter overflows after $2^{m_{\mathsf{A}}}$ cycles, exactly the number of maximal cycles when a particular universal memory contents recurs. Thus, $\mathcal{T}$ accepts iff $\mathcal{S}$ diverges, and deciding whether $\mathcal{T}$ accepts can be done in $\textsc{ASpace}(m_{\mathsf{A}})^{\textsc{Eval}(C_{\mathsf{A}})}$.

The adaptation to the three nonalternating cases is straightforward by just skipping the guessing of the existential and/or the universal decision. □

The second lemma establishes an upper bound for evaluating combinatorial circuits.

Lemma 4.2.4. *For a given combinatorial circuit C reading from inputs I and a valuation $\vec{i} \in \vec{I}$, evaluating C on $\vec{i}$ is in* $\textsc{Space}(\mathit{depth}(C) \cdot \log|C|)$.

Proof. Consider an algorithm that, starting with its output, recursively evaluates C in a depth-first manner. At each level of the recursion, we iteratively recur on the inputs of the current gate. If the current gate is an *AND* gate and the propagated value is **false**, then we terminate the recursion for the current gate and propagate **false** to the calling instance, otherwise (if all inputs yield **true**) we return **true**. Dually, if the current gate is an *OR* gate and the propagated value is **true**, then we terminate the recursion for the current gate and propagate **true** to the calling instance, otherwise (if all inputs yield **false**) we return **false**. If the recursion reaches an input i, then $\vec{i}(i)$ is propagated to the calling instance.

The recursion can be implemented using a stack with a height in the length of the longest path in C, which corresponds to $\mathit{depth}(C)$. The elements in the stack are used to remember the position (i.e., the index of a particular gate of the circuit) and the last branching decision (i.e., the index of an input to the gate) at each level of the recursion (i.e., a particular depth

in the circuit). As the indexes can be represented only using $O(\log n)$ bits, it can be easily seen that the algorithm is in $\textsc{Space}(\mathit{depth}(C) \cdot \log |C|)$. □

We now come to the main theorems of this section. The following establishes the fact that universal memory in sequential circuit machines corresponds to tape cells in (otherwise unbounded) Turing machines, assuming no bound on the existential memory.

Theorem 4.2.5. *For any nondecreasing class of functions $\mathcal{F} \geq \log n$, the following equivalences hold, even when we only assume simple circuits:*

$$\mathcal{C}((0,0,0),(\infty,\mathcal{F})) = \textsc{Space}(\mathcal{F});$$
$$\mathcal{C}((1,0,0),(\infty,\mathcal{F})) = \textsc{NSpace}(\mathcal{F});$$
$$\mathcal{C}((0,0,1),(\infty,\mathcal{F})) = \textsc{coNSpace}(\mathcal{F});$$
$$\mathcal{C}((1,1,1),(\infty,\mathcal{F})) = \textsc{ASpace}(\mathcal{F})$$

Proof. We first prove the "$\subseteq$" direction. Let $\mathcal{S} = (\sigma, C_{\mathsf{A}})$ and $n = |\mathcal{S}| = |C_{\mathsf{A}}|$. Recall that C_{A} is of constant depth. Thus, according to Lemma 4.2.4, we have that $\textsc{Eval}(C_{\mathsf{A}})$ is in $\textsc{Space}(\log n)$. Moreover, since $O(\log n) \leq \mathcal{F}$, we have that $\textsc{Eval}(C_{\mathsf{A}})$ is in $\textsc{Space}(\mathcal{F})$. Hence, according to Lemma 4.2.3, the upper bound complexities for deciding Diverging for the four cases are

$$\begin{aligned}
\textsc{Space}(\mathcal{F})^{\textsc{Space}(\mathcal{F})} &= \textsc{Space}(\mathcal{F}),\\
\textsc{NSpace}(\mathcal{F})^{\textsc{Space}(\mathcal{F})} &= \textsc{NSpace}(\mathcal{F}),\\
\textsc{coNSpace}(\mathcal{F})^{\textsc{Space}(\mathcal{F})} &= \textsc{coNSpace}(\mathcal{F}),\\
\textsc{ASpace}(\mathcal{F})^{\textsc{Space}(\mathcal{F})} &= \textsc{ASpace}(\mathcal{F}),
\end{aligned}$$

respectively.

For proving the "$\supseteq$" direction, we first show the $\textsc{ASpace}(\mathcal{F})$-hardness of the alternating case $\mathcal{C}((1,1,1),(\infty,\mathcal{F}))$. For a given alternating Turing machine $\mathcal{T} = (Q, q_0, T, \delta)$ that, without loss of generality, has a tape bound $b = 2^k$ and $|Q| = 2^q$, for some $k, q \in \mathbb{N}$, we construct in LogSpace a sequential circuit machine $\mathcal{S} = (((1,1,1),(\infty, m_{\mathsf{A}})), C_{\mathsf{A}})$ such that $\mathcal{T}$ accepts iff $\mathcal{S}$ diverges. We use the universal memory of $\mathcal{S}$ to store the contents of the b tape cells (using b bits), the position of the tape head ranging from 0 to $b-1$ (using k bits), the state $\mathcal{T}$ is currently in (using q bits), a timeout counter that counts the executions of $\mathcal{T}$ (using b bits), and a 1-bit flag *div* that indicates whether an accepting configuration have been reached. Overall, we have $m_{\mathsf{A}} = O(b)$.

The transition function δ as well as the contents of the input tape are encoded as C_{A}: Based on the type of the state $\mathcal{T}$ is currently in (either E, A, or D), C_{A} either selects NE_0, NA_0, or constant 0, respectively, to resolve the nondeterminism in δ. We let C_{A} update the contents of the current tape

cell and move the tape head. If the type of the current state is Acc, we let C_{A} set $div = 1$, otherwise we increment the timeout counter. If $div = 0$ and the counter overflows, we let C_{A} produce $\mathsf{h} = 1$.

The translation of δ (which is explicitly given) into C_{A} (which can be of polynomial size, i.e., of size polynomial in $|\delta|$) is straightforward, as we can just obtain a DNF from δ of polynomial size. Overall, observe that C_{A}, besides the DNF for δ, only contains incrementer, decrementer, comparator, and multiplexer circuits. While the first three types of circuits are used to move the tape head and to update the counters, multiplexers are used to read the contents of the currently selected tape cell and to look up the type of the state $\mathcal{T}$ is currently in. Recall that we allow an unbounded fan-in for combinatorial circuits, hence, all the sub-circuits have constant depth. This concludes the proof of the ASpace($\mathcal{F}$)-hardness.

What remains is to adapt the construction to the three nonalternating cases. Note that NE_0 and NA_0 are only used in the sub-circuit of C_{A} representing T. Since the mode of acceptance of $\mathcal{T}$ only depends on the definition of T, it is easy to see that the other cases are hard for Space($\mathcal{F}$), NSpace($\mathcal{F}$), and coNSpace($\mathcal{F}$), respectively. □

When restricting the existential nondeterminism to partial or no observability, we encounter the same exponential blow-up that was already observed by Reif [1984] for alternating Turing machines in the nonsuccinct case.

Theorem 4.2.6. *For any nondecreasing class of functions $\mathcal{F} \geq \log n$, the following equivalences hold, even when we only assume simple circuits:*

$$\mathcal{C}((1,0,1),(\infty,\mathcal{F})) = \text{NSpace}(2^{\mathcal{F}});$$
$$\mathcal{C}((1,1,2),(\infty,\mathcal{F})) = \text{ASpace}(2^{\mathcal{F}})$$

Proof. For proving the "$\subseteq$" direction, we first show the upper bound for the private case: $\mathcal{C}((1,1,2),(\infty,\mathcal{F})) \subseteq \text{ASpace}(2^{\mathcal{F}})$. The adaptation to the blindfold case is straightforward.

For a given machine $\mathcal{S} = (((1,1,2),(\infty,m_{\mathsf{A}})),C_{\mathsf{A}})$, we show that Diverging($\mathcal{S}$) can be decided by an alternating algorithm with a memory consumption of $O(2^{m_{\mathsf{A}}})$ bits. Recall that the semantics of $\mathcal{S}$ is defined in terms of a private game: the existence of a feasible C_{E} corresponds to the existence of a strategy for Player E that respects the partial observability on the decisions of Player A. Now, by applying the *belief set construction* by Reif [1984], we obtain a game of perfect information. In this construction, the positions of the perfect-information game represent *beliefs*: over-approximations of the positions in which the actual game might be currently in. Thus, compared to the original imperfect-information game, the perfect-information game has an exponential number of positions. Recall

that games induced by sequential circuit machines are strictly turn-based and the possible moves the players can play do not depend on the current position. Hence, each belief can be clearly associated to a particular player and the available moves are always the same for each belief. The set of bad beliefs contains those beliefs that subsume a position with a halting configuration. This concludes the proof of the $\text{ASPACE}(2^{\mathcal{F}})$ upper bound.

The adaptation to the blindfold case is straightforward. Observe that, in this case, the belief set construction completely removed the universal nondeterminism, so that the game of perfect information only contains the existential nondeterminism. Hence, the algorithm runs in $\text{NSPACE}(2^{\mathcal{F}})$.

Similar to the upper bound, for proving the "$\supseteq$" direction, we first show the $\text{ASPACE}(2^{\mathcal{F}})$-hardness of the alternating case $\mathcal{C}((1,1,2),(\infty,\mathcal{F}))$. For a given alternating Turing machine $\mathcal{T} = (Q, q_0, T, \delta)$ that, without loss of generality, has a tape bound $b = 2^k$ and $|Q| = 2^q$, for some $k, q \in \mathbb{N}$, we construct in LOGSPACE a sequential circuit machine $\mathcal{S} = (((1,1,2),(\infty, m_{\mathsf{A}})), C_{\mathsf{A}})$ such that $\mathcal{T}$ accepts iff $\mathcal{S}$ diverges and $m_{\mathsf{A}} = O(k)$. The basic idea is to let C_{E} sequentially simulate the steps of $\mathcal{T}$ so that $\mathcal{T}$ eventually reaches an accepting state. A single step of $\mathcal{T}$ is communicated by C_{E} to C_{A} in a sequence of bits with constant length: q bits representing the next state, 1 bit representing the contents to write into the current tape cell, 1 bit indicating in which direction the input tape head moves, and 1 bit indicating in which direction the work tape head moves.

Similar to a proof presented by Rintanen [2004] in the planning setting, instead of storing the contents of the whole tape, we let C_{A}, unobservable for C_{E}, nondeterministically select a dedicated tape cell that is *watched* by C_{A}. We use k bits in the universal memory to represent the bits of some integer variable ranging from 0 to $b-1$. We introduce an integer variable *watch* to store the index of the watched cell and another two integer variables *input* and *work* to store the current positions of the tape heads of the input and work tape, respectively. As an initialization step, unobservable for C_{E}, C_{A} nondeterministically chooses a value for *watch* and sets *input* and *work* to 0. Then, C_{A} keeps track of the movement of the tape head by decrementing or incrementing *input* and *work*. Here, C_{A} produces $\mathsf{h} = 1$ whenever C_{E} proposes to move the head to the left and it is over the first cell, or to move to the right and the head is over the last cell. If $work \neq watch$, C_{A} ignores the correctness of the tape operations (writing in a cell and moving the head) determined by C_{E}. If $work = watch$, the tape operations are verified (C_{A} checks if the proposed step is consistent with δ) and the writing of the new tape symbol is memorized. The state $\mathcal{T}$ is currently in is represented using q bits, and the contents of the watched tape cell is represented using 1 bit.

The transitions in δ are encoded in the following way. In case $\mathcal{T}$ is in an existential state, C_{A} expects that C_{E} resolves the branching decision (0 or 1).

In case $\mathcal{T}$ is in a universal state, C_{A} first provides the branching decision, which should then be considered by C_{E}. Note that the latter decision is visible for C_{E}.

So far, $\mathcal{S}$ diverges whenever C_{E} does faithfully simulate the steps of $\mathcal{T}$. However, what remains is to enforce that C_{E} actually reaches an accepting state of $\mathcal{T}$ eventually. We do this by introducing a timeout counter that is incremented after a step of $\mathcal{T}$ is simulated, and in case it overflows, C_{A} produces $\mathsf{h} = 1$. Observe that the maximal number of steps without visiting a state twice corresponds to the number of possible configurations of $\mathcal{T}$. That is, we need to count steps up to a number bounded by 2^{2^k}. As this number is double exponential in the number of bits, we cannot use just another b-bit integer variable to represent the timeout counter. Instead, we construct an incrementation gadget that implements a carry chain adder of length 2^b. Now, the trick is to let C_{E} do the incrementation: Before C_{E} performs a step of $\mathcal{T}$, C_{E} has to additionally produce a sequence of 2^b bits representing the current value of the timeout counter (ranging from 0 to $2^{2^k} - 1$). To verify that C_{E} increments the timeout counter correctly, similar to the watched tape cell explained above, we let C_{A} nondeterministically and unobservably for C_{E} select a dedicated bit j, $0 \leq j < 2^b$, whose correct incrementation is verified. The actual gadget contains a loop that iterates an integer variable i from 0 to $2^b - 1$. In each iteration, C_{E} has to provide the carry flag before bit i is incremented, the new (incremented) value of bit i, and the carry flag for bit $i + 1$. If $i \neq j$, we require that C_{E} provides *some* value for the carry flags and the new value of bit i. If $i = j$, we actually check that Player E provides correct values. It is easy to see that the new value of bit j only depends on its last value and the incoming carry flag. Furthermore, if $i = 0$, we always require that the first carry flag is 1. If $i = 2^b - 1$ and the next carry flag is 1, the counter overflows, which means that we have reached the timeout and let C_{A} produce $\mathsf{h} = 1$. This concludes the proof of the $\text{ASPACE}(2^{\mathcal{F}})$-hardness.

Now, the adaptation to the blindfold case is straightforward: Without having access to any decisions of C_{A}, C_{E} can only simulate an existential nondeterministic Turing machine $\mathcal{T}$. However, we can still use the hidden universal nondeterminism to check whether C_{E} simulates $\mathcal{T}$ correctly. Hence, the $\text{NSPACE}(2^{\mathcal{F}})$-hardness. □

Finally, combining Theorems 4.2.5 and 4.2.6 immediately reveals the fact that privateness can always be traded for exponential universal space.

Corollary 4.2.7. *For any nondecreasing class of functions $\mathcal{F} \geq \log n$, the following equivalences hold, even when we only assume simple circuits:*

$$\mathcal{C}((1,0,1),(\infty,\mathcal{F})) = \mathcal{C}((1,0,0),(\infty,2^{\mathcal{F}}));$$
$$\mathcal{C}((1,1,2),(\infty,\mathcal{F})) = \mathcal{C}((1,1,1),(\infty,2^{\mathcal{F}}))$$

4.2.3 Machines with Bounded Existential Memory

In this section, we make the surprising discovery that bounding the memory of the existential nondeterminism always dominates any restriction on the observability or the power of both the universal and existential nondeterminism in general.

The first theorem establishes the fact that if the existential memory bound of a sequential circuit machine is at most logarithmic in the universal memory, then we have the same computational power as a nondeterministic space-bounded Turing machine.

Theorem 4.2.8. *For any nondecreasing class of functions $\mathcal{F} \geq \log n$, the following equivalence holds, even when we only assume simple circuits:*

$$\begin{aligned}
&\mathcal{C}((1,0,0),(0,2^{\mathcal{F}})) = \mathcal{C}((1,0,0),(\mathcal{F},2^{\mathcal{F}})) = \\
&\mathcal{C}((1,0,1),(0,2^{\mathcal{F}})) = \mathcal{C}((1,0,1),(\mathcal{F},2^{\mathcal{F}})) = \\
&\mathcal{C}((1,1,1),(0,2^{\mathcal{F}})) = \mathcal{C}((1,1,1),(\mathcal{F},2^{\mathcal{F}})) = \\
&\mathcal{C}((1,1,2),(0,2^{\mathcal{F}})) = \mathcal{C}((1,1,2),(\mathcal{F},2^{\mathcal{F}})) = \mathrm{NSPACE}(2^{\mathcal{F}})
\end{aligned}$$

Proof. For showing the "$\subseteq$" direction, we prove the containment in $\mathrm{NSPACE}(2^{\mathcal{F}})$ for the most general case $\mathcal{C}((1,1,2),(\mathcal{F},2^{\mathcal{F}}))$, where we have full alternation (i.e., we have both existential and universal nondeterminism) and where we assume partial information, which is clearly more general than the blindfold and fully informed cases. For a given machine $\mathcal{S} = (((1,1,2),(m_{\mathsf{E}},m_{\mathsf{A}})),C_{\mathsf{A}})$, let $n = |\mathcal{S}|$, $b = f(n)$, for some f from $\mathcal{F}$, $m_{\mathsf{E}} = O(b)$, and $m_{\mathsf{A}} = 2^{O(b)}$. We show that $\textsc{Diverging}(\mathcal{S})$ can be decided by a nondeterministic algorithm with a memory consumption of $O(m_{\mathsf{A}})$ bits.

First, observe that if there is a feasible C_{E}, then it can be represented by a truth table that, depending on the current contents of the existential memory and the current universal decision, defines the existential decision. Since there can be at most $2^{m_{\mathsf{E}}} = 2^{O(b)}$ distinct configurations of the existential memory but only one binary universal choice visible to C_{E}, the table has at most $2^{O(b)}$ number of rows. And since there is only one binary existential choice, the table has only a single column. Hence, the total number of entries in the table is $2^{O(b)}$, which can be guessed in $2^{O(b)}$ steps. The validation of the guess is done by integrating C_{E} into $\mathcal{S}$ and deciding $\textsc{Diverging}$ on the machine so obtained (which has only universal nondeterminism). Since, according to Theorem 4.2.5, deciding $\textsc{Diverging}$ for such machines can be done in $\mathrm{coNSPACE}(2^{\mathcal{F}})$, the overall complexity is in

$$\mathrm{NTIME}(2^{\mathcal{F}})^{\mathrm{coNSPACE}(2^{\mathcal{F}})} = \mathrm{NTIME}(2^{\mathcal{F}})^{\mathrm{NSPACE}(2^{\mathcal{F}})} = \mathrm{NSPACE}(2^{\mathcal{F}}).$$

The "$\supseteq$" direction easily follows from the $\mathrm{NSPACE}(2^{\mathcal{F}})$-hardness of the most restricted case $\mathcal{C}((1,0,0),(0,2^{\mathcal{F}}))$, where we only have existential but no

universal nondeterminism. An even weaker case is when we assume $\mathcal{F}$ to be minimal (i.e., $\mathcal{F} = \mathcal{L}$ and thus $2^{\mathcal{F}} = \mathcal{P}$) and we neither have existential nor universal nondeterminism $\mathcal{C}((0,0,0),(0,\mathcal{P}))$, which is, according to Theorem 4.2.5 and Savitch [1970], already hard for $\textsc{Space}(\mathcal{P}) = \textsc{NSpace}(2^{\mathcal{F}})$. □

The following theorem states that, if the existential memory bound is in the order of the universal memory bound, we always obtain the computational power of nondeterministic time-bounded Turing machines.

Theorem 4.2.9. *Let $\mathcal{F}$ be either $\mathcal{L}$ or $\mathcal{P}$, the following equivalence holds, even when we only assume simple circuits:*

$$\begin{aligned}
&\mathcal{C}((1,0,0),(\mathcal{F},\mathcal{F})) = \\
&\mathcal{C}((1,0,1),(\mathcal{F},\mathcal{F})) = \\
&\mathcal{C}((1,1,1),(\mathcal{F},\mathcal{F})) = \\
&\mathcal{C}((1,1,2),(\mathcal{F},\mathcal{F})) = \textsc{NTime}(2^{\mathcal{F}}),
\end{aligned}$$

where $\textsc{NTime}(2^{\mathcal{F}})$ *is either* NP Time *or* NExpTime*, respectively.*

Proof. For showing the "$\subseteq$" direction, we prove the containment in $\textsc{NTime}(2^{\mathcal{F}})$ for the most general case $\mathcal{C}((1,1,2),(\mathcal{F},\mathcal{F}))$, where we have full alternation (i.e., we have both existential and universal nondeterminism) and where we assume partial information, which is clearly more general than the blindfold and fully informed cases. For a given machine $\mathcal{S} = (((1,1,2),(m_{\mathsf{E}},m_{\mathsf{A}})),C_{\mathsf{A}})$, let $n = |\mathcal{S}|$, $b = f(n)$, for some f from $\mathcal{F}$, and $m_{\mathsf{E}}, m_{\mathsf{A}} \in O(b)$. We prove that $\textsc{Diverging}(\mathcal{S})$ can be decided by the following nondeterministic algorithm that requires at most $O(2^{m_{\mathsf{A}}})$ steps.

Analogously to the proof of the upper bound of Theorem 4.2.8, we guess C_{E} in form of a truth table. However, here, the number of rows of the table is exponential in m_{E} (while it still has only a single column). Hence, guessing the table can be done in time exponential in m_{E}. Also observe that the table can be represented as a DNF of exponential size. We validate the guess by integrating C_{E} into $\mathcal{S}$ and deciding Diverging on the machine so obtained $\mathcal{S}'$ (which has only universal nondeterminism). According to Lemma 4.2.4, evaluating C_{E} can be done in $\textsc{Space}(m_{\mathsf{E}}) \subseteq \textsc{Space}(\mathcal{F})$. Moreover, according to 4.2.3, deciding Diverging for $\mathcal{S}'$ can thus be done in $\textsc{coNSpace}(\mathcal{F})^{\textsc{Space}(\mathcal{F})} = \textsc{NSpace}(\mathcal{F})$. Thus, the overall complexity is in $\textsc{NTime}(2^{\mathcal{F}})^{\textsc{NSpace}(\mathcal{F})} = \textsc{NTime}(2^{\mathcal{F}})$.

The "$\supseteq$" direction follows from the $\textsc{NTime}(2^{\mathcal{F}})$-hardness, $\mathcal{F} \in \{\mathcal{L},\mathcal{P}\}$, of the most restricted case $\mathcal{C}((1,0,0),(\mathcal{F},\mathcal{F}))$, where we only have a (minimal) existential nondeterminism but no universal nondeterminism. Our hardness proof is a generalization of the NExpTime-hardness proof by Kupferman

and Sheinvald-Faragy [2006, Theorem 5]. We give a reduction from HAMILTONIAN CYCLE if $\mathcal{F} = \mathcal{L}$ or SUCCINCT HAMILTONIAN CYCLE if $\mathcal{F} = \mathcal{P}$, which are known to be NPTIME-complete [Karp, 1972] and NEXPTIME-complete [Galperin and Wigderson, 1983], respectively, to DIVERGING.

An instance of HAMILTONIAN CYCLE is a graph G, where we use $n \in \mathbb{N}$ as the number of bits needed to represent a node of G. Without loss of generality, we assume that $n = 2^k$, for some $k \in \mathbb{N}$. In a succinctly specified instance, G is given implicitly as a combinatorial circuit with $2n$ inputs (or, alternatively, as a Boolean function with arity $2n$ [Veith, 1997]). To unify our reduction for both the enumerative case ($\mathcal{F} = \mathcal{L}$) and the succinct case ($\mathcal{F} = \mathcal{P}$), we assume that the explicitly given graph is also represented as a Boolean function. Hence, G is a Boolean function $G : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, where, if $\mathcal{F} = \mathcal{L}$, $n = \lceil \log |Q| \rceil$ assuming that the instance is explicitly given as a tuple (Q, E). Two nodes v, v' of G, represented using n bits each, are connected iff $G(v, v') = 1$.

For a given G, we construct a sequential circuit machine $\mathcal{S} = (\sigma, C_{\mathsf{A}})$ with $\sigma = ((1,0,0),(m_{\mathsf{E}}, m_{\mathsf{A}}))$, where $m_{\mathsf{E}} = O(n)$ and $m_{\mathsf{A}} = O(n)$, such that G has a Hamiltonian cycle iff DIVERGING$(\mathcal{S}) = \mathbf{yes}$. The idea of the construction of $\mathcal{S}$ is to let C_{E} produce a path of length 2^n and to let C_{A} (which is purely deterministic) validate whether this path is (1) a valid path and (2) a Hamiltonian cycle in G. More precisely, C_{A} expects the path as a sequence of nodes provided by C_{E}. A path contains 2^n nodes represented as $n2^n$ bits, which are communicated from C_{E} to C_{A} in a cyclic iteration. At any point in this iteration, C_{A} maintains the current edge (a pair of nodes) in its memory. Whenever a new edge is complete (i.e., the n bits of the succeeding node are completely provided), C_{A} checks whether the edge actually exists by checking whether G evaluates to 1. If this is not the case, $\mathcal{S}$ halts by letting C_{A} produce $\mathsf{h} = 1$, otherwise the iteration continues.

By limiting the number of bits per path to $n2^n$, we make sure that the path must cycle exactly according to the length of a (potential) Hamiltonian cycle. To make sure that the proposed path is indeed Hamiltonian, we have to enforce that every node of G is visited eventually. Observe that it would be infeasible to introduce 2^n bits to memorize whether a node was visited. Instead, we adopt a technique proposed by Kupferman and Sheinvald-Faragy [2006, Theorem 5] and introduce a node counter with n bits that, starting with 0, is incremented whenever the value of the next node provided by C_{E} equals the current counter value. What remains to do is to introduce a watchdog counter with n bits that makes sure that the node counter ultimately overflows after that 2^n cycles have been accomplished. If the watchdog counter overflows without having seen an overflow of the node counter, we let C_{A} produce $\mathsf{h} = 1$ again.

More technically, the current edge and the various counters can be represented using a linear amount of bits, thus $m_{\mathsf{A}} = O(n)$. By choosing $m_{\mathsf{E}} = \log(n2^n) = n + k$, we bound the length of the candidate path. Ob-

serve that a path that visits less than 2^n nodes (which, of course, can never be a Hamiltonian cycle) would be rejected by the watchdog counter.

In the actual construction of C_{A}, besides an embedding of G, we also need incrementer, multiplexer, and comparator sub-circuits, which are all of polynomial size and constant depth (since we allow an unbounded fan-in). However, we have to be a bit more careful with the embedding of the Boolean function G into C_{A}. If $\mathcal{F} = \mathcal{L}$ and G is represented explicitly, we can represent the edge relation of G as a simple DNF of polynomial size. If $\mathcal{F} = \mathcal{P}$, we can apply the well-known technique by Tseitin [1968] to transform G (which might be an arbitrary Boolean formula) into a CNF (which is of constant depth), only suffering from a linear blow-up in the size of the formula and the number of the node bits. The length of a sub sequence, where C_{E} communicates a node to C_{A}, needs to be extended accordingly (which also only requires a polynomial blowup). □

Combining Theorems 4.2.1 and 4.2.9 immediately reveals the fact that bounded existential memory can be traded for an exponential amount of nondeterminism.

Corollary 4.2.10. *The following equivalence holds, even when we only assume simple circuits:*

$$\begin{aligned}
&\mathcal{C}((1,0,0),(\mathcal{L},\mathcal{L})) = \\
&\mathcal{C}((1,0,1),(\mathcal{L},\mathcal{L})) = \\
&\mathcal{C}((1,1,1),(\mathcal{L},\mathcal{L})) = \\
&\mathcal{C}((1,1,2),(\mathcal{L},\mathcal{L})) = \mathcal{C}((\mathcal{P},0,0),(0,0))
\end{aligned}$$

Concerning the size of the representation of the smallest feasible C_{E}, if there is one at all, the following two theorems state that it is highly unlikely[1] that a compact C_{E} always exists.

The first theorem considers the case of a logarithmic amount of existential memory.

Theorem 4.2.11. *For a given sequential circuit machine $\mathcal{S} = (\sigma, C_{\mathsf{A}})$ with $\sigma = ((1,0,0),(m_{\mathsf{E}}, m_{\mathsf{A}}))$, $m_{\mathsf{E}} = O(\log |C_{\mathsf{A}}|)$, and $m_{\mathsf{A}} \geq m_{\mathsf{E}}$, if there is a combinatorial circuit C that uses a memory M with $|M| \leq m_{\mathsf{E}}$ such that $\mathcal{S}[C_{\mathsf{E}} \leftarrow C, M_{\mathsf{E}} \leftarrow M]$ diverges, then C cannot always be represented logarithmically, unless* PTime = NPTime.

Proof. We consider the most restricted case $m_{\mathsf{A}} = \log |\mathcal{S}|$: if C_{E} cannot (most probably) be of logarithmic size for a logarithmic amount of universal memory, then it certainly cannot be of logarithmic size for a polynomial amount of universal memory.

[1] as it is common belief that PTime ⊊ NPTime as well as PSpace ⊊ ExpTime ⊊ NExpTime

We assume that C_{E} can always be represented logarithmically (i.e., $|C_{\mathsf{E}}| = O(\log |\mathcal{S}|)$ and can thus be represented using logarithmically many bits). Then, we can guess C_{E} in logarithmically many steps and represent C_{E} using logarithmic space. Hence, C_{E} can be guessed in NLogSpace. Validating the guess, according to Lemma 4.2.3 and Ladner [1975], can be done in $\text{coNLogSpace}^{\text{PTime}} = \text{PTime}$. Thus, the overall complexity is $\text{NLogSpace}^{\text{PTime}} = \text{PTime}$. However, according to Theorem 4.2.9, synthesizing a C_{E} is complete for NPTime. Thus, NPTime = PTime (under the assumption that C_{E} can always be represented logarithmically). □

The second theorem considers the case of a polynomial amount of existential memory.

Theorem 4.2.12. *For a given sequential circuit machine $\mathcal{S} = (\sigma, C_{\mathsf{A}})$ with $\sigma = ((1,0,0),(m_{\mathsf{E}}, m_{\mathsf{A}}))$, $m_{\mathsf{E}} = |\mathcal{S}|^{O(1)}$, and $m_{\mathsf{A}} = |\mathcal{S}|^{O(1)}$, if there is a combinatorial circuit C that uses a memory M with $|M| \leq m_{\mathsf{E}}$ such that $\mathcal{S}[C_{\mathsf{E}} \leftarrow C, M_{\mathsf{E}} \leftarrow M]$ diverges, then C cannot always be represented polynomially, unless* PSpace = NExpTime.

Proof. We assume that C_{E} can always be represented polynomially (i.e., $|C_{\mathsf{E}}| = |\mathcal{S}|^{O(1)}$ and thus can be represented using polynomially many bits). Then, we can always guess C_{E} in polynomially many steps and represent C_{E} using polynomial space. Hence, C_{E} can be guessed in NPSpace. Validating the guess, according to Lemma 4.2.3 and Ladner [1975], can be done in $\text{coNPSpace}^{\text{PTime}} = \text{NPSpace}$. Thus, the overall complexity is $\text{NPSpace}^{\text{NPSpace}} = \text{NPSpace} = \text{PSpace}$. However, according to Theorem 4.2.9, synthesizing a circuit C_{E} is complete for $\text{NTime}(2^{\mathcal{P}}) = \text{NExpTime}$. Thus, NExpTime = PSpace (under the assumption that C_{E} can always be represented polynomially). □

4.3 Succinctness Unifies Space and Time

Putting it all together, it turns out that the space of signature classes whose unary representation is at most polynomial spans precisely the computational power characterized by the (Turing machine-based) complexity classes between LogSpace and 2ExpTime. This confirms our claim from Section 3.1 that many natural problems can be classified according to the amount of succinctness granted to the existential and universal nondeterminism.

Tables 4.1 and 4.2 summarize the equivalences between signature classes and complexity classes characterized by space- or time-bounded Turing machines, respectively. In the tables, $\exists$ and $\forall$ represent the existential and universal nondeterminism, respectively. From left to right, each row shows the (Turing-machine based) complexity class, the power and the observability of $\exists$, the power of $\forall$, the memory available to $\exists$, and the memory available

to ∀. The power of both the universal and existential nondeterminism, the observability, as well as the memory are shown in terms of number of bits.

Complexity class	Signature class				
	∃ nondet.	obs.	∀ nondet.	∃ mem.	∀ mem.
LogSpace	0	0	0	∞	$\mathcal{L}$
NLogSpace	1	0	0	∞	$\mathcal{L}$
	$\mathcal{L}$	0	0	∞	$\mathcal{L}$
	0	0	1	∞	$\mathcal{L}$
	0	0	$\mathcal{L}$	∞	$\mathcal{L}$
PSpace	1	0	1	∞	$\mathcal{L}$
	$\mathcal{L}$	0	$\mathcal{L}$	∞	$\mathcal{L}$
	0	0	0	0	$\mathcal{P}$
	$\mathcal{L}$	0	0	$\mathcal{L}$	$\mathcal{P}$
	$\mathcal{L}$	0	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{P}$
	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{P}$
	$\mathcal{P}$	0	0	∞	$\mathcal{P}$
	0	0	$\mathcal{P}$	∞	$\mathcal{P}$
ExpSpace	1	0	1	∞	$\mathcal{P}$
	$\mathcal{P}$	0	$\mathcal{P}$	∞	$\mathcal{P}$

Table 4.1: Equivalences between signature classes and complexity classes characterized by space-bounded Turing machines.

Our results show that an unbounded amount of existential memory always corresponds to the computational power of space-bounded Turing machines. In case of full observability, one obtains alternating space-bounded complexity classes, which, according to Chandra et al. [1981], correspond to deterministic time-bounded complexity classes. When the amount of universal memory is at least exponential in the existential memory, one also obtains the power of space-bounded computations. The general exponential increase in complexity due to partial observability, which was already observed by Reif [1984] for alternating Turing machines, can also be shown for the succinct case.

We also reveal the insight that a symmetric amount of existential and universal memory always corresponds to the computational power of nondeterministic time-bounded Turing machines. Interestingly, this relationship turns out to be invariant of the actual observational power of the existential nondeterminism, which is in contrast to space-bounded computations.

Complexity class	Signature class				
	$\exists$ nondet.	obs.	$\forall$ nondet.	$\exists$ mem.	$\forall$ mem.
PTime	1	1	1	∞	$\mathcal{L}$
	$\mathcal{L}$	∞	$\mathcal{L}$	∞	$\mathcal{L}$
NPTime	$\mathcal{P}$	0	0	0	0
	1	0	0	$\mathcal{L}$	$\mathcal{L}$
	$\mathcal{L}$	0	0	$\mathcal{L}$	$\mathcal{L}$
	$\mathcal{L}$	0	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$
	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$
	$\mathcal{L}$	∞	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$
coNPTime	0	0	$\mathcal{P}$	0	0
Σ_2^P	$\mathcal{P}$	0	$\mathcal{P}$	0	0
Π_3^P	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$	0	0
Π_2^P	$\mathcal{P}$	∞	$\mathcal{P}$	0	0
ExpTime	1	1	2	∞	$\mathcal{L}$
	$\mathcal{L}$	$\mathcal{L}$	$\mathcal{L}$	∞	$\mathcal{L}$
	1	1	1	∞	$\mathcal{P}$
	$\mathcal{P}$	∞	$\mathcal{P}$	∞	$\mathcal{P}$
NExpTime	1	0	0	$\mathcal{P}$	$\mathcal{P}$
	$\mathcal{P}$	0	0	$\mathcal{P}$	$\mathcal{P}$
	$\mathcal{P}$	0	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$
	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$
	$\mathcal{P}$	∞	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$
2ExpTime	1	1	2	∞	$\mathcal{P}$
	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$	∞	$\mathcal{P}$

Table 4.2: Equivalences between signature classes and complexity classes characterized by time-bounded Turing machines.

Chapter 5

The Ubiquity of Sequential Circuit Machines

We are now ready to demonstrate the relevance of our new universal computation model by investigating the relation to other well-known formalisms from the literature. Many formalisms that are unrelated at first sight share a close connection to sequential circuit machines, in the sense that they can be easily reduced to each other. In this chapter, we exploit these connections to achieve two goals: (1) We first recover known complexity results from the literature with drastically simpler proofs than the ones given in the original papers. In fact, once the connection between a formalism and sequential circuit machines is established, many results immediately follow as corollaries. (2) Using this new proof technique, we are also able to establish matching lower and upper complexity bounds of some open problems.

The proposed approach is uniform for each formalism under consideration: We pick a general decision problem P (one that subsumes many other problems of interest) and define its succinctness signature $\widehat{\sigma}$ as a purely syntactic function of the instance description. We then prove that P and $\textsc{Diverging}_{\widehat{\sigma}}$ are LogSpace-reducible to each other, thereby establishing the completeness of P for the signature class that contains all succinctness signatures subsumed by $\widehat{\sigma}$. Thanks to the syntactic connection between the instances of P and succinctness signatures, we can now easily relate P and its subproblems to signature classes, which, in turn, correspond to standard complexity classes.

In Section 5.1, we start our investigation by relating the fundamental principle of concurrency to succinctness. As a fundamentally different formalism, Section 5.2 demonstrates how a specification logic can be seen from a succinctness perspective. Section 5.3 discusses further relations to other succinct modeling formalisms.

5.1 Communicating State Machines

We start with communicating state machines, a well-known and general computation model for asynchronous systems. For this thesis, this formalism is interesting for two reasons: First, it represents a canonical formalism to capture concurrency, which, historically, was one of the first sources of complexity for which the state explosion problem was observed. Second, other formalisms (such as timed automata) introduce parallelism by adopting the general principle behind communicating state machines. Understanding the succinctness of this model will give a clearer picture of the succinctness of timed automata in Part II of this thesis.

We first define the computation model, choose controller synthesis as the general decision problem, and then, by establishing the connection to sequential circuit machines, we uniformly obtain complexity results.

5.1.1 Definition

Syntax. A *communicating state machine* (also known as *concurrent state machine*) (CSM) M is represented by a tuple (Q, q_0, Σ, E), where

- Q is a finite set containing the *states*,
- $q_0 \in Q$ is the initial state,
- Σ is a finite and nonempty set of *events*, and
- $E \subseteq Q \times \Sigma \times Q$ is a transition relation.

For two states $q, q' \in Q$ and an a, we write $q \xrightarrow{a} q'$ iff (1) $a \in \Sigma$ and $(q, a, q') \in E$, or (2) $a \notin \Sigma$ and $q = q'$. We require CSMs to be *event-deterministic*: for any two edges $(q_1, a_1, q_1'), (q_2, a_2, q_2') \in E$, we require

$$(q_1 = q_2 \wedge a_1 = a_2) \Rightarrow (q_1' = q_2').$$

A *network of communicating state machines* is a finite set of CSMs $\mathcal{N} = \{M_1, M_2, \ldots, M_n\}$ that run asynchronously and synchronize on shared events. We also call the elements of a network *components* and use

$$M_1 \| M_2 \| \ldots \| M_n$$

as an alternative notation. In the following, we define

$$Q = \prod_{i=1}^{n} Q_i \qquad \text{and} \qquad \Sigma = \bigcup_{i=1}^{n} \Sigma_i$$

as those sets that contain the global states and the decisions of $\mathcal{N}$, respectively. We assume that a *global state* $\vec{q} \in Q$ is represented as an n-dimensional vector. For two global states $\vec{q}, \vec{q}' \in Q$, and an event $a \in \Sigma$,

we write $\vec{q} \xrightarrow{a} \vec{q}'$ iff, for all $1 \leq i \leq n$, $\vec{q}[i] \xrightarrow{a} \vec{q}'[i]$. A situation in which no further transition can be executed is called a *deadlock*. The set of all deadlock states is defined as

$$\mathsf{Deadlock}(\mathcal{N}) = \left\{\vec{q} \in Q \mid \neg\exists a\ \exists \vec{q}' : \vec{q} \xrightarrow{a} \vec{q}'\right\}.$$

A *controllable network of CSMs* is a network of CSMs $\mathcal{N}$ (also called a *plant*) whose events are partitioned into controllable events $\Sigma_{\mathsf{in}} \subseteq \Sigma$ (over which $\mathcal{N}$ can be influenced) and into uncontrollable events $\Sigma_{\mathsf{out}} \subseteq \Sigma$ (which are emitted by $\mathcal{N}$ and which can be observed externally). We assume that each component $(Q_i, q_0^i, \Sigma_i, E_i)$ in $\mathcal{N}$ is *input enabled*: for each $q \in Q_i$ and each $a \in \Sigma_{\mathsf{in}}$, we require that

$$a \in \Sigma_i \Rightarrow \exists q' \in Q_i : (q, a, q') \in E_i.$$

Example 5.1.1. *Figure 5.1 shows an example network of two state machines communicating via actions a, c, and d. Observe that the network runs into a deadlock state after two a events are produced.*

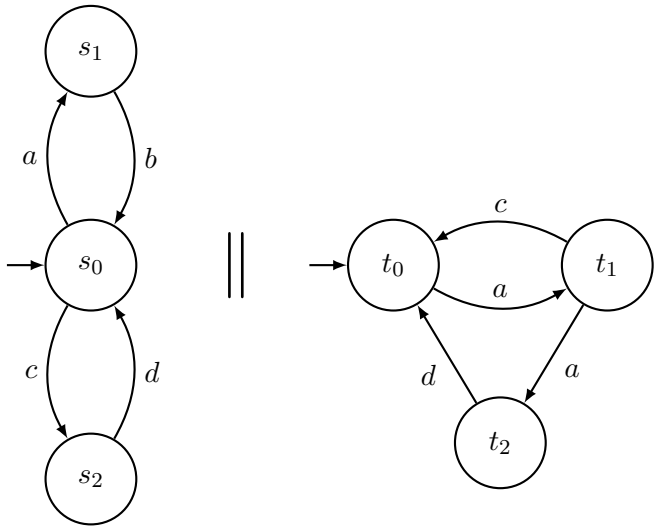


Figure 5.1: Example network of communicating state machines.

Semantics. The semantics of a controllable network of CSMs $\mathcal{N}$ with events $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$ is formally described as the finite game arena $[\![\mathcal{N}]\!] = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$, where

- $S_{\mathsf{E}} = \{\mathsf{E}\} \times (Q_1 \times \ldots \times Q_n)$,
- $S_{\mathsf{A}} = \{\mathsf{A}\} \times (Q_1 \times \ldots \times Q_n)$,
- $s_0 = (\mathsf{A}, (q_0^1, \ldots, q_0^n))$,

- $\Sigma_{\mathsf{E}} = \Sigma_{\mathsf{in}} \uplus \{\tau\}$,
- $\Sigma_{\mathsf{A}} = \Sigma_{\mathsf{out}} \uplus \{\tau\}$,
- $\Delta((p, \vec{q}), a) = s'$, where

$$s' = \begin{cases} (\overline{p}, \vec{q}) & \text{if } a = \tau; \\ (\mathsf{A}, \vec{q}') & \text{if } \vec{q} \xrightarrow{a} \vec{q}'. \end{cases}$$

For a set of states $Y \subseteq Q$, we write $[\![\mathcal{N}]\!] \models \mathsf{AG}(Y)$ as a shorthand for $[\![\mathcal{N}]\!] \models \mathsf{AG}(\{\mathsf{E}, \mathsf{A}\} \times Y)$.

5.1.2 Controller Synthesis

Problem definition. An input to the *safety controller synthesis problem for controllable networks of CSMs* CsmSynth is represented by a tuple $(\mathcal{N}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, q^{\mathsf{max}})$, where

- $\mathcal{N}$ is a controllable network of CSMs,
- $\Sigma_{\mathsf{out}}^{\mathsf{obs}} \subseteq \Sigma_{\mathsf{out}}$ are the observable output events,
- $q_b \in \bigcup_{1 \leq i \leq n} Q_i$ is a dedicated bad state, and
- $q^{\mathsf{max}} \in \mathbb{N} \cup \{\infty\}$ is a bound on the number of states of the controller.

Deciding CsmSynth on x is to check whether there exists a controller component $C = (Q_c, q_0^c, \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}}, E_c)$ such that the following conditions are satisfied:

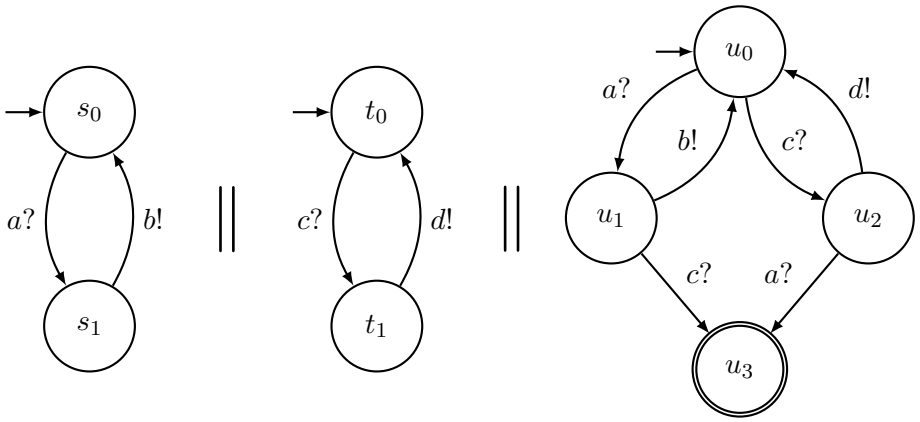
(1) C satisfies the *state bound*: $|Q_c| \leq q^{\mathsf{max}}$;

(2) C is *nonblocking*: if $[\![\mathcal{N}]\!] \models \mathsf{AG}(\overline{\mathsf{Deadlock}(\mathcal{N})})$ then $[\![\mathcal{N} \| C]\!] \models \mathsf{AG}(\overline{\mathsf{Deadlock}(\mathcal{N} \| C)})$;

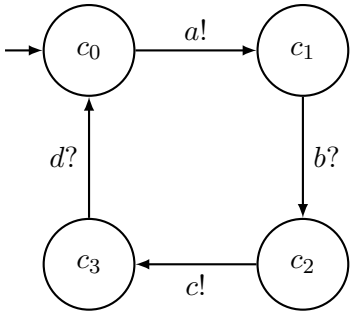
(3) C is *safe*: $[\![\mathcal{N} \| C]\!] \models \mathsf{AG}(\overline{S_b})$, where S_b is the set of those states, where the component that contains q_b is in q_b.

For convenience, we define $\Sigma_{\mathsf{out}}^{\mathsf{unobs}} = \Sigma \setminus (\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}})$ to refer to all unobservable events. We say that CsmSynth is *under full observability* if $\Sigma_{\mathsf{out}}^{\mathsf{unobs}} = \emptyset$. We say that CsmSynth is *under no observability* if $\Sigma_{\mathsf{out}}^{\mathsf{obs}} = \emptyset$.

Example 5.1.2. *Figure 5.2(a) shows an example network of communicating state machines. The input actions are a and c, while the output actions are b and d. The state u_3 of the third component is the dedicated bad state, which is reached whenever the first two components enter the states s_1 and t_1 at the same time. A feasible four-state controller is shown in Figure 5.2(b), which ensures that u_3 is never reached.*



(a) Example network of CSMs.



(b) Example controller.

Figure 5.2: Example network of communicating state machines with a controller that ensures that u_3 is never reached.

Game-theoretic solution. An input x to CsmSynth, where $x = (\mathcal{N}, \Sigma^{\mathsf{obs}}_{\mathsf{out}}, q_b, q^{\mathsf{max}})$ induces a finite safety game

$$\mathcal{G}(\textsc{CsmSynth}, x) = (\mathcal{A}, B, \mathcal{V}, \beta),$$

which is defined as follows:

- $\mathcal{A} = [\![\mathcal{N}]\!]$.
- B is defined as those states, where the component that contains q_b is in q_b.
- The fact that the controller can only observe the last visible event emitted by $\mathcal{N}$ (but not the current state of the components) is reflected in the definition of the view of Player E on $\mathcal{A}$:

$$\mathcal{V} = (\Sigma^{\mathsf{obs}}_{\mathsf{out}} \uplus \{\tau\}, \mathit{vis}),$$

where

$$vis(d) = \begin{cases} d & \text{if } d \in \Sigma_{\mathsf{out}}^{\mathsf{obs}} \cup \{\tau\}; \\ \varepsilon & \text{otherwise.} \end{cases}$$

- $\beta = q^{\mathsf{max}}$.

Theorem 5.1.3. *For a given input x,* CsmSynth(x) = **yes** *iff Player* E *wins* $\mathcal{G}(\text{CsmSynth}, x)$.

Proof. The statement immediately follow from the definition of (the constituent components of) $(\mathcal{A}, B, \mathcal{V}, \beta)$. □

5.1.3 Complexity Analysis

For an input $x = (\mathcal{N}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, q^{\mathsf{max}})$ to CsmSynth, we define the instance signature $sig(\text{CsmSynth}, x)$ as

$$\log\big((|\Sigma_{\mathsf{in}}| + 1,\ |\Sigma_{\mathsf{out}}^{\mathsf{obs}}| + 1,\ |\Sigma_{\mathsf{out}}| + 1),\ \ (q^{\mathsf{max}},\ |\prod_{M \in \mathcal{N}} M_Q|)\big).$$

Note that we have not yet specified whether q^{max} is given in unary or binary.

As a basis for obtaining the complexity of model checking and synthesis for communicating state machines, we first show that one can use sequential circuit machines to decide CsmSynth.

Lemma 5.1.4. *For an input x, $\mathcal{G}(\text{CsmSynth}, x)$ allows a succinct circuit representation of signature $sig(\text{CsmSynth}, x)$.*

Proof. We show that the definition of $\mathcal{G}(\text{CsmSynth}, x)$ matches the properties required for succinct circuit representations from Section 3.4.

(1) There is a logarithmic encoding of the controllable decisions $\Sigma_{\mathsf{in}} \cup \{\tau\}$ since Σ_{in} is explicitly given. We introduce the bits $[\![\widehat{\Sigma_{\mathsf{out}}^{\mathsf{obs}} \cup \{\tau\}}]\!]$ for representing the observable uncontrollable events. We also introduce the bits $[\![\widehat{\Sigma_{\mathsf{out}}^{\mathsf{unobs}}}]\!]$ for representing the unobservable uncontrollable events. Then, the bits for the logarithmic encoding of all uncontrollable decisions are $[\![\widehat{\Sigma_{\mathsf{out}}^{\mathsf{obs}} \cup \{\tau\}}]\!] \uplus [\![\widehat{\Sigma_{\mathsf{out}}^{\mathsf{unobs}}}]\!]$.

The logarithmic encoding of the positions can be obtained by combining the local logarithmic encodings of the components, whose states are given explicitly. One extra bit is additionally needed to encode the information, which player can move. A position can thus be represented as a bit string whose length is bounded by $|\mathcal{N}| \cdot \lceil \log Q_{max} \rceil + 1$, where $Q_{max} = \max_{1 \le i \le |\mathcal{N}|}(|Q_i|)$.

(2) Concerning the move function Δ, we first obtain $|\mathcal{N}|$ sub-circuits representing the local transition relations of the individual components. Let

$b = \lceil \log |Q_i| \rceil$ be the number of node bits for component $\mathcal{N}_i$, $1 \leq i \leq |\mathcal{N}|$. Since each E_i is explicitly given, for each E_i, we can obtain a DNF of size polynomial in $|Q_i| \cdot |\Sigma|$ that computes a Boolean function $E_i^c(s, d) = (v, s')$ that, for a state $s \in Q_i$ and a decision $d \in \Sigma$, computes a valid flag $v \in \{0, 1\}$ and a successor state $s' \in Q_i$ such that $v = 1$ iff $(s, d, s') \in E_i$. The simple circuit Δ^c representing the move function of $[\![\mathcal{N}]\!]$ can now be obtained by executing the various E_i^c in parallel for computing the bits of the global successor state. The global valid flag is just the conjunction of all local valid flags. Hence, it can be easily seen that the size of Δ^c is bounded by $O(|Q_{max}| \cdot |\Sigma|)$ and its depth is constant.

(3) Similar to (2), we construct B^c as a DNF that yields a 1 whenever the component that contains state q_b is in q_b.

(4) We represent $\mathcal{V}$ as the subset $(\!|\widehat{\Sigma_{\mathsf{out}}^{\mathsf{obs}} \cup \{\tau\}}|\!) \subseteq (\!|\widehat{\Sigma_{\mathsf{out}} \cup \{\tau\}}|\!)$. □

By combining Lemma 5.1.4 and Lemma 3.4.1, we immediately obtain that every given instance of CsmSynth can be transformed into an instance of Diverging for sequential circuit machines with the same signature.

Lemma 5.1.5. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((\mathcal{L}, \mathcal{L}_\infty, \mathcal{L}), (\mathcal{P}_\infty, \mathcal{P}))$, $\textsc{CsmSynth}_{\widehat{\sigma}}$ is* LogSpace*-reducible to $\textsc{Diverging}_{\widehat{\sigma}}$.*

With the following lemma, we establish the opposite direction:

Lemma 5.1.6. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((\mathcal{L}, \mathcal{L}_\infty, \mathcal{L}), (\mathcal{P}_\infty, \mathcal{P}))$, $\textsc{Diverging}_{\widehat{\sigma}}$ is* LogSpace*-reducible to $\textsc{CsmSynth}_{\widehat{\sigma}}$.*

Proof. For a given machine $\mathcal{S} = (((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (m_{\mathsf{E}}, m_{\mathsf{A}})), C_{\mathsf{A}})$, we construct an input $(\mathcal{N}, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, q^{\mathsf{max}})$ to CsmSynth in the following way. Without loss of generality, we assume C_{A} has no shared sub-circuits and, thus, is represented as a Boolean expression, where $C_{\mathsf{A}}(\mathsf{MA}_i)$ is the Boolean function that computes the new value for MA_i, $1 \leq i \leq m_{\mathsf{A}}$, and $C_{\mathsf{A}}(\mathsf{h})$ computes the value of the halting flag.

The idea of our construction is to use independent components to represent bits. For representing the bits of the circuit elements, for each element x in $N_{\mathsf{A}} \cup N_{\mathsf{E}} \cup M_{\mathsf{A}} \cup \{\mathsf{h}\}$, and the updated universal memory $M_{\mathsf{A}}{}'$, we introduce a component $M^x = (Q^x, q_0^x, \Sigma^x, E^x)$, which is defined as follows:

$$
\begin{aligned}
Q^x &= \{m_0^x, m_1^x\};\\
q_0^x &= m_0^x;\\
\Sigma^x &= \{r(x,0), r(x,1), w(x,0), w(x,1)\};\\
E^x &= \{\ (m_0^x, w(x,0), m_0^x),\ (m_0^x, w(x,1), m_1^x),\ (m_0^x, r(x,0), m_0^x),\\
&\qquad (m_1^x, w(x,0), m_0^x),\ (m_1^x, w(x,1), m_1^x),\ (m_1^x, r(x,1), m_1^x)\ \}
\end{aligned}
$$

We define the set of observable uncontrollable events as $\Sigma_{\mathsf{out}}^{\mathsf{obs}} = \vec{N_{\mathsf{A}}^{\mathsf{E}}}$, the set of unobservable uncontrollable events as $\Sigma_{\mathsf{out}}^{\mathsf{unobs}} = \{\epsilon\}$, and the set of

controllable events as $\Sigma_{\mathsf{in}} = \vec{N}_{\mathsf{E}}$. To simulate the actual sequential execution of $\mathcal{S}$, we introduce a control component $M^c = (Q^c, q_0^c, \Sigma^c, E^c)$, which is constructed as follows. First, for determining values for the unobservable universal guessing bits, we let M^c nondeterministically branch with an ϵ-edge to a particular chain of write events of the form

$$g_1^u \xrightarrow{w(\mathsf{NA}_1, v_1)} \ldots \rightarrow g^u_{n_{\mathsf{A}}-n^{\mathsf{E}}_{\mathsf{A}}} \xrightarrow{w(\mathsf{NA}_{n_{\mathsf{A}}-n^{\mathsf{E}}_{\mathsf{A}}}, v_{n_{\mathsf{A}}-n^{\mathsf{E}}_{\mathsf{A}}})} g_f^u,$$

where $(v_1, \ldots, v_{n_{\mathsf{A}}-n^{\mathsf{E}}_{\mathsf{A}}})$ represents a vector from $N_{\mathsf{A}} \vec{\setminus} N^{\mathsf{E}}_{\mathsf{A}}$. Here, we consider $2^{n_{\mathsf{A}}-n^{\mathsf{E}}_{\mathsf{A}}}$ branching decisions to cover all possible choices. Then, for determining values for the observable universal guessing bits, M^c nondeterministically branches with an edge labeled with a $\vec{n_a^e} \in \vec{N^{\mathsf{E}}_{\mathsf{A}}}$ to a particular chain of widgets of the form

$$g_1^o \xrightarrow{w(\mathsf{NO}_1, \vec{n_a^e}[1])} \ldots \rightarrow g^o_{n^{\mathsf{E}}_{\mathsf{A}}} \xrightarrow{w(\mathsf{NO}_{n^{\mathsf{E}}_{\mathsf{A}}}, \vec{n_a^e}[n^{\mathsf{E}}_{\mathsf{A}}])} g_f^o,$$

Here, we consider $2^{n^{\mathsf{E}}_{\mathsf{A}}}$ branching decisions to cover all possible choices. After the universal guessing bits are determined, M^c awaits the values for the circuit elements N_{E} from the controller. For this purpose, we let P receive a controllable event $\vec{n_e} \in \Sigma_{\mathsf{in}}$ and then branch to a chain of widgets of the form

$$g_1^e \xrightarrow{w(\mathsf{NE}_1, \vec{n_e}[1])} \ldots \rightarrow g^e_{n_{\mathsf{E}}} \xrightarrow{w(\mathsf{NE}_{n_{\mathsf{E}}}, \vec{n_e}[n_{\mathsf{E}}])} g_f^e,$$

Since we have only a logarithmic number of guessing bits, the exponential blowup due the nondeterministic branching results only in a polynomial control structure.

Now, for each Boolean formula $C_{\mathsf{A}}(y)$, $y \in M_{\mathsf{A}} \cup \{\mathsf{h}\}$, we embed its formula DAG $D = dag(C_{\mathsf{A}}(y))$ (recall Section 2.2) into the control structure of M^c: For each node in D, we add a state to Q^c, and for each edge in D, we add an edge to E^c, such that for some circuit element $z \in N_{\mathsf{A}} \cup N_{\mathsf{E}} \cup M_{\mathsf{A}}$, if the original edge is labeled with the literal z then the corresponding edge in E^c carries the event $r(z, 1)$, and if the original edge is labeled with the literal $\neg z$ then the corresponding edge in E^c carries the event $r(z, 0)$. We identify $root(D)$ with c^y, $true(D)$ with s_0^y, and $false(D)$ with s_1^y. We connect the various sub components with each other by adding the following edges to E^c:

(1) $(g_f^y, w(y, v), g_1^{y'})$, where $v \in \{0, 1\}$ and $y, y' \in N_{\mathsf{E}} \cup N_{\mathsf{A}}$ assuming that y' succeeds y in the total order of the circuit elements $\mathsf{NE}_1, \ldots, \mathsf{NE}_{n_{\mathsf{E}}}, \mathsf{NA}_1, \ldots, \mathsf{NA}_{n_{\mathsf{A}}}$;

(2) $(s_v^y, w(y'', v), c^{y'})$, where $v \in \{0, 1\}$, and $y, y' \in M_{\mathsf{A}} \cup \{\mathsf{h}\}$ assuming that y' succeeds y in the total order of the circuit elements $\mathsf{MA}_1, \ldots, \mathsf{MA}_{m_{\mathsf{A}}}, \mathsf{h}$,

and that y'' is either the corresponding updated memory element if $y \in M_{\mathsf{A}}$, or h otherwise;

(3) $(s_v^{\mathsf{h}}, w(\mathsf{h}, v), g_1)$, where $v \in \{0, 1\}$ and assuming that g_1 is the first guessing state or c^{MA_1} if $N_{\mathsf{A}} = N_{\mathsf{E}} = \emptyset$.

Note that the only existential nondeterminism in M^c occurs for resolving the branching decisions for the existential guessing bits. The universal nondeterminism is used for resolving the branching decisions for the universal guessing bits and for resolving the nondeterminism in the DAGs of the Boolean formulas. Before the execution of a cycle finishes, we copy the contents of the updated memory $M_{\mathsf{A}}{}'$ to M_{A} by the following chain of events:

$$d_1 \xrightarrow{r(M_{\mathsf{A}1}', v_1)} d_1^{v_1} \xrightarrow{w(M_{\mathsf{A}1}, v_1)} \ldots \rightarrow$$
$$d_{m_{\mathsf{A}}} \xrightarrow{r(M_{\mathsf{A}m_{\mathsf{A}}}', v_{m_{\mathsf{A}}})} d_{m_{\mathsf{A}}}^{v_{m_{\mathsf{A}}}} \xrightarrow{w(M_{\mathsf{A}m_{\mathsf{A}}}, v_{m_{\mathsf{A}}})} d_f$$

We finish our construction by defining the set of processes

$$\mathcal{N} = \{M^c, M^x \mid x \in N_{\mathsf{E}} \cup N_{\mathsf{A}} \cup M_{\mathsf{A}} \cup \{\mathsf{h}\}\}$$

and the bad state q_b as m_1^{h}. □

The combination of Lemma 5.1.5 and Lemma 5.1.6 gives the desired connection between CSMs and SCMs:

Theorem 5.1.7. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((\mathcal{L}, \mathcal{L}_\infty, \mathcal{L}), (\mathcal{P}_\infty, \mathcal{P}))$,* CsmSynth$_{\widehat{\sigma}}$ *and* Diverging$_{\widehat{\sigma}}$ *are* LogSpace*-reducible to each other.*

When we assume a one-player setting (by either choosing $\Sigma_{\mathsf{in}} = \emptyset$ or $\Sigma_{\mathsf{out}} = \emptyset$), controller synthesis reduces to the safety model checking problem. The combination of Theorem 5.1.7, Lemma 4.1.1, and Theorem 4.2.5 yields the complexity for safety model checking of CSMs:

Corollary 5.1.8. *The* AG*-model checking problem for networks of CSMs is complete for*

- $\mathcal{C}((0, 0, \mathcal{L}), (0, \mathcal{L})) =$ NLogSpace*, if only a constant number of components is allowed, and*
- $\mathcal{C}((0, 0, \mathcal{L}), (0, \mathcal{P})) =$ PSpace*, in the general case, respectively.*

The EG*-model checking problem for networks of CSMs is complete for*

- $\mathcal{C}((\mathcal{L}, 0, 0), (\infty, \mathcal{L})) =$ NLogSpace*, if only a constant number of components is allowed, and*
- $\mathcal{C}((\mathcal{L}, 0, 0), (\infty, \mathcal{P})) =$ PSpace*, in the general case, respectively.*

When we assume two players, by combining Lemma 4.1.1 and Theorems 4.2.5, 4.2.6, and 5.1.7, we obtain the complexities for controller synthesis under no, partial, and full observability:

Corollary 5.1.9. *The unbounded safety controller synthesis problem for controllable networks of CSMs is complete for*

- $\mathcal{C}((\mathcal{L}, 0, \mathcal{L}), (\infty, \mathcal{P})) = \text{ExpSpace}$*, assuming no observability,*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\infty, \mathcal{P})) = 2\text{ExpTime}$*, assuming partial observability, and*
- $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\infty, \mathcal{P})) = \text{ExpTime}$*, assuming full observability, respectively.*

By imposing a bound on the number of processes in the plant and/or on the states of the controller, by combining Theorem 5.1.7, Lemma 4.1.1, and Theorems 4.2.8 and 4.2.9, we obtain the complexities for finding small witnesses and bounded controller synthesis:

Corollary 5.1.10. *The* EG*-small witness problem for networks of CSMs is complete for*

- $\mathcal{C}((\mathcal{L}, 0, 0), (\mathcal{L}, \mathcal{L})) = \text{NPTime}$ *if only a constant number of components is allowed and the bound on the length of the witness is encoded in unary,*
- $\mathcal{C}((\mathcal{L}, 0, 0), (\mathcal{L}, \mathcal{P})) = \text{PSpace}$ *if the bound on the length of the witness is encoded in unary, and*
- $\mathcal{C}((\mathcal{L}, 0, 0), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$ *if the bound on the length of the witness is encoded in binary, respectively.*

Corollary 5.1.11. *The bounded safety controller synthesis problem for controllable networks of CSMs is complete for*

- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\mathcal{L}, \mathcal{L})) = \text{NPTime}$ *if only a constant number of components is allowed and the bound on the states of the controller is encoded in unary,*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\mathcal{L}, \mathcal{P})) = \text{PSpace}$ *if the bound on the states of the controller is encoded in unary, and*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$ *if the bound on the states of the controller is encoded in binary, respectively.*

5.1.4 Bibliographic Remarks

The NLogSpace-completeness result of deciding (co-)reachability in an explicitly given (i.e., unary-encoded) directed graph is due to Jones [1975]. The unavoidable exponential increase to PSpace in the model checking complexity for CSMs was shown by Harel et al. [1997]. Also, the ExpTime-completeness result for the alternating (i.e., safety synthesis) case can easily be deduced from that work. Kupferman and Sheinvald-Faragy [2006] established the NPTime-, PSpace-, and NExpTime-completeness for the small witness problem for nondeterministic, alternating, and concurrent Büchi word automata[1], respectively, where the bound on the length of the witness is given in binary. The NPTime-completeness of bounded synthesis against a specification in form of an automaton, for a bound given in unary, was shown by Schewe and Finkbeiner [2007] and Kupferman et al. [2011]. The latter work also proved that when imposing unary bounds on the controller *and* the environment, the problem becomes Σ_2^P-complete, a result that, in principle, can also be easily established using sequential circuit machines.

Based on the connection to sequential circuit machines, we are able to fill in the missing gaps: For instance, the Corollaries 5.1.10 and 5.1.11 establish the PSpace-completeness of both finding shortest witnesses and bounded synthesis for CSMs, where the bounds are given in unary.

5.2 Linear-time Temporal Logic

In the previous section, we saw how one can use sequential circuit machines to uniformly obtain various complexity results for an automata-based modeling formalism. One might ask the question, whether one could apply the same technique to obtain complexity results for specification logics, a fundamentally different kind of formalism.

As a proof of concept, in this section, we make the connection between sequential circuit machines and *Linear-time Temporal Logic* (LTL) [Pnueli, 1977]. Using sequential circuit machines, the PSpace-hardness proof of LTL satisfiability gets surprisingly simple and intuitive.

We first recall the definition of the safety fragment of LTL and choose satisfiability as the main decision problem. We then establish the connection to sequential circuit machines and finally obtain a new PSpace-completeness proof.

5.2.1 Definition

For a comprehensive overview on LTL, we refer to Baier and Katoen [2008, Chapter 5].

[1] Even though Büchi acceptance is more general than safety acceptance, it is easily seen that their proofs of the lower bounds also apply to the safety case.

Syntax. Let AP be a finite set of atomic propositions. The set of all *LTL formulas* $\mathrm{LTL}(AP)$ is inductively defined by the following rules:

- **true**;
- **false**;
- a, for a $a \in AP$;
- $\neg a$, for a $a \in AP$;
- $\varphi \wedge \psi$, for $\varphi, \psi \in \mathrm{LTL}(AP)$;
- $\varphi \vee \psi$, for $\varphi, \psi \in \mathrm{LTL}(AP)$;
- $\mathsf{X}\varphi$, for $\varphi \in \mathrm{LTL}(AP)$;
- $\mathsf{G}\varphi$, for $\varphi \in \mathrm{LTL}(AP)$;
- $\mathsf{F}\varphi$, for $\varphi \in \mathrm{LTL}(AP)$;
- $\varphi \mathsf{U} \psi$, for $\varphi, \psi \in \mathrm{LTL}(AP)$;
- $\varphi \mathsf{R} \psi$, for $\varphi, \psi \in \mathrm{LTL}(AP)$.

Here, X, G, U, and R are called the *temporal operators*. We write $[\varphi]$ to refer to the *set of subformulas* of φ. The *temporal depth* of φ is the maximal number of nestings of temporal operators. We say that a formula φ is in the *syntactic safety fragment* if the only temporal operators occurring in φ are X and G. Let $\textsc{SafeLTL}(AP)$ contain all formulas in the syntactic safety fragment.

Semantics. For a given finite set of atomic propositions AP, a *path* over AP is defined as an infinite sequence of sets of atomic propositions. We define the set of all paths as $\textsc{Paths}(AP) = \left(2^{AP}\right)^{\omega}$. If $p \in \textsc{Paths}(AP)$ is of the form

$$A_1 \; A_2 \; A_3 \; \ldots$$

then, for an $i \in \mathbb{N}_{\geq 1}$, we write $p[i]$ to refer to A_i and $p[i \ldots]$ to refer to the infinite subsequence

$$A_i \; A_{i+1} \; A_{i+2} \; \ldots.$$

The semantics of LTL-formulas is defined in terms of the satisfaction relation $\models$. For a given path $p \in \textsc{Paths}(AP)$ and an LTL-formula $\varphi \in \mathrm{LTL}(AP)$, we define $p \models \varphi$ as follows:

- $p \models$ **true**;

- $p \not\models \mathbf{false}$;
- $p \models a$ iff $a \in p[1]$;
- $p \models \neg a$ iff $a \notin p[1]$;
- $p \models \varphi \wedge \psi$ iff $p \models \varphi$ and $p \models \psi$;
- $p \models \varphi \vee \psi$ iff $p \models \varphi$ or $p \models \psi$;
- $p \models \mathsf{X}\varphi$ iff $p[2\ldots] \models \varphi$;
- $p \models \mathsf{G}\varphi$ iff for all $i > 0$, $p[i\ldots] \models \varphi$;
- $p \models \varphi \mathsf{U} \psi$ iff there is an $i > 0$ such that $p[i\ldots] \models \psi$ and for all $0 < j < i$, $p[j\ldots] \models \varphi$;
- $p \models \mathsf{F}\varphi$ iff $p \models \mathbf{true}\mathsf{U}\varphi$;
- $p \models \varphi \mathsf{R} \psi$ iff $p \models \mathsf{G}\psi \vee \psi \mathsf{U}(\varphi \wedge \psi)$.

5.2.2 Satisfiability

Problem definition. For a given set of atomic propositions AP and an LTL formula $\varphi \in \mathrm{LTL}(AP)$, the problem LtlSat$(AP, \varphi)$ is to decide whether there exists a path $p \in \mathrm{Paths}(AP)$ such that $p \models \varphi$.

Example 5.2.1. *The LTL formula* $a\mathsf{U}(b\mathsf{X}(\mathsf{G}c))$ *is satisfied by the path* $\{a\}\{a\}\{a\}\{b\}\{c\}^\omega$. *The LTL formula* $(\mathsf{G}\neg a) \wedge (b\mathsf{U}a)$ *is unsatisfiable.*

The problem SafeLtlSat(AP, φ) is the restriction of LtlSat to the syntactic safety fragment (i.e., assuming that $\varphi \in \mathrm{SafeLTL}(AP)$).

Game-theoretic solution. We give a solution for SafeLtlSat in terms of a blindfold game that is played between Players E and A. It can be seen as a game-theoretic extension of the approach based on alternating automata [Vardi, 1997] proposed by Manna and Sipma [2000]. To the best of the author's knowledge, this is the first such interpretation of LTL satisfiability.

For a given formula φ, the idea of our construction is that Player E stepwise proposes a path p and resolves the disjunctions in φ. Player A resolves the conjunctions in φ and checks whether the proposed p satisfies φ. We use the blindfoldedness of E to hide all the verification steps that A nondeterministically performs to validate p.

The actual interplay between E and A alternates between a *proposal* and a *verification phase*. In the proposal phase, E determines which atomic propositions are part of the next step of p. A nondeterministically (and

unobservably for E) chooses a certain subformula that should be faithfully checked in the verification phase.

In the verification phase, A nondeterministically (and, again, unobservably for E) navigates through adjacent subformulas until he reaches (1) a subformula **true** or **false**, (2) an atomic proposition check (of the form a or $\neg a$, for an $a \in AP$), or (3) a next operator (of the form $\mathsf{X}\psi$, for a $\psi \in \mathrm{LTL}(AP)$). In cases (1) or (2), either the game stops by entering a dedicated bad state if the propositional check is negative or it enters a state from which the bad state is not reachable. In case (3), A remains at the current subformula $\mathsf{X}\psi$ and waits until the verification phase ends, whereupon ψ is chosen as the next subformula. In each round of the verification phase, E proposes a sequence of bits of constant length to resolve the disjunctions in φ. This way, E does not need to know which subformula A has chosen.

More formally, a set of atomic propositions $AP = \{a_0, a_1, \ldots, a_n\}$ and an LTL formula $\varphi \in \textsc{SafeLTL}(AP)$ induce an unbounded blindfold safety game

$$\mathcal{G}(\textsc{SafeLtlSat}, (AP, \varphi)) = (\mathcal{A}, B, \mathcal{V}_\tau, \infty),$$

which is defined in the following.

$\mathcal{A} = (S_\mathsf{E}, S_\mathsf{A}, s_0, \Sigma_\mathsf{E}, \Sigma_\mathsf{A}, \Delta)$ is a game arena that models the game construction explained above, where

- the positions are divided into the proposal and the verification phase:

$$\begin{aligned} S_\mathsf{E} = {} & (\{P\} \times \{\mathsf{E}\} \times [\varphi] \times AP \times \{0,1\}^2 \times \{0, \ldots, |AP|\}) \cup \\ & (\{V\} \times \{\mathsf{E}\} \times [\varphi] \times AP \times \{0,1\} \times \{0, \ldots, |[\varphi]|\}^2) \cup \\ & \{\textbf{true}\}, \\ S_\mathsf{A} = {} & (\{P\} \times \{\mathsf{A}\} \times [\varphi] \times AP \times \{0,1\}^2 \times \{0, \ldots, |AP|\}) \cup \\ & (\{V\} \times \{\mathsf{A}\} \times [\varphi] \times AP \times \{0,1\} \times \{0, \ldots, |[\varphi]|\}^2) \cup \\ & \{\textbf{false}\}, \end{aligned}$$

- $s_0 = (P, \mathsf{E}, \varphi, a, 0, 0, 0)$ for some arbitrary $a \in AP$,
- $\Sigma_\mathsf{E} = \{0, 1\}$,
- $\Sigma_\mathsf{A} = \{0, 1\}$,
- the proposal phase is defined as $\Delta((P, p, \varphi_1, a, q, q', c), d) = s'$, where

$$s' = \begin{cases} (P, \mathsf{A}, \varphi_1, a, q, d, c) & \text{if } p = \mathsf{E},\ c < |AP|; \\ (P, \mathsf{E}, \varphi_1, a_c, q', q', c+1) & \text{if } p = \mathsf{A},\ c < |AP|,\ d = 0; \\ (P, \mathsf{E}, \varphi_1, a, q, q', c+1) & \text{if } p = \mathsf{A},\ c < |AP|,\ d = 1; \\ (V, \mathsf{E}, \varphi_1, a, q, 0, 0) & \text{if } c = |AP|; \end{cases}$$

- the proposal of the disjunctions in the verification phase is defined as $\Delta((V, \mathsf{E}, \varphi_1, a, q, c_e, c_a), d) = s'$, where

$$s' = \begin{cases} (V, \mathsf{E}, \varphi_1, a, q, c_e + 1, c_a) & \text{if } c_e < |[\varphi]|,\ c_e \neq idx(\varphi_1); \\ (V, \mathsf{E}, \varphi_2, a, q, c_e + 1, c_a) & \text{if } c_e = idx(\varphi_1),\ \varphi_1 \xrightarrow{d}_{\mathsf{E}} \varphi_2; \\ (V, \mathsf{A}, \varphi_1, a, q, 0, c_a) & \text{if } c_e = |[\varphi]|; \end{cases}$$

- the actual verification is defined as $\Delta((V, \mathsf{A}, \varphi_1, a, q, c_e, c_a), d) = s'$, where

$$s' = \begin{cases} (V, \mathsf{E}, \varphi_2, a, q, 0, c_a + 1) & \text{if } c_a < |[\varphi]|,\ \varphi_1 \xrightarrow{d}_{\mathsf{A}} \varphi_2; \\ (P, \mathsf{E}, \psi, a, q, q, 0) & \text{if } c = |[\varphi]|,\ \varphi_1 \equiv \mathsf{X}\psi; \\ a' = a \Rightarrow q = 1 & \text{if } c = |[\varphi]|,\ \varphi_1 \equiv a'; \\ a' = a \Rightarrow q = 0 & \text{if } c = |[\varphi]|,\ \varphi_1 \equiv \neg a'; \\ \mathbf{true} & \text{if } c = |[\varphi]|,\ \varphi_1 \equiv \mathbf{true}; \\ \mathbf{false} & \text{if } c = |[\varphi]|,\ \varphi_1 \equiv \mathbf{false}. \end{cases}$$

Here, the relations $\rightarrow_{\mathsf{E}}$ and $\rightarrow_{\mathsf{A}}$ connect adjacent subformulas in φ and thus constitute its *subformula tree*. Let φ_1 be a subformula of φ and $p \in \{\mathsf{E}, \mathsf{A}\}$, then we generally have $\varphi_1 \xrightarrow{d}_p \varphi_1$, for a $d \in \{0, 1\}$, except for the following cases:

- if $\varphi_1 \equiv \varphi_2 \wedge \psi_2$ and $p = \mathsf{A}$ then $\varphi_1 \xrightarrow{0}_{\mathsf{A}} \varphi_2$ and $\varphi_1 \xrightarrow{1}_{\mathsf{A}} \psi_2$;
- if $\varphi_1 \equiv \varphi_2 \vee \psi_2$ and $p = \mathsf{E}$ then $\varphi_1 \xrightarrow{0}_{\mathsf{E}} \varphi_2$ and $\varphi_1 \xrightarrow{1}_{\mathsf{E}} \psi_2$;
- if $\varphi_1 \equiv \mathsf{G}\varphi_2$ and $p = \mathsf{A}$ then $\varphi_1 \xrightarrow{0}_{\mathsf{A}} \varphi_2$ and $\varphi_1 \xrightarrow{1}_{\mathsf{A}} \mathsf{X}\varphi_1$.

Note that we assume for $[\varphi]$, without loss of generality, for each subformula $\mathsf{G}\varphi_1$ there is also a subformula $\mathsf{XG}\varphi_1$.

We finish our game construction by defining the set of bad states as the singleton set $B = \{\mathbf{false}\}$.

Theorem 5.2.2. *For a given set of atomic propositions AP and an LTL formula* $\varphi \in \textsc{SafeLTL}(AP)$, $\textsc{LtlSat}(AP, \varphi) =$ **yes** *iff Player* E *wins* $\mathcal{G}(\textsc{SafeLtlSat}, (AP, \varphi))$.

Proof. The statement immediately follows from the definition of (the constituent components of) $\mathcal{G}(\textsc{SafeLtlSat}, (AP, \varphi))$. □

5.2.3 Complexity Analysis

Similar to the previous section, we first define the instance signature of a dedicated problem, which is SafeLtlSat in this case. For a set of atomic

propositions AP and an LTL formula $\varphi \in \text{SafeLTL}(AP)$, we define the instance signature $sig(\text{SafeLtlSat}, (AP, \varphi))$ as

$$\big((1,\ 0,\ 1),\ \ (\infty,\ O(\log|[\varphi]| + \log|AP|))\big).$$

Hence, a candidate signature class for which we want to prove completeness is $\mathcal{C}((1,0,1),(\infty,\mathcal{L}))$.

We first show that we can use sequential circuit machines to encode the game construction described above.

Lemma 5.2.3. *For a set of atomic propositions AP and an LTL formula $\varphi \in \text{SafeLTL}(AP)$, $\mathcal{G}(\text{SafeLtlSat}, (AP, \varphi))$ allows a succinct circuit representation of signature $sig(\text{SafeLtlSat}, (AP, \varphi))$.*

Proof. Since every component of $\mathcal{G}(\text{SafeLtlSat}, (AP, \varphi))$ is given explicitly, it can be easily seen that its definition matches the properties required for succinct circuit representations from Section 3.4. The move function Δ and the bad states B can be represented as DNFs. □

The combination of Lemma 5.2.3 and Lemma 3.4.1, we immediately obtain that every given instance of SafeLtlSat can be transformed into an instance of Diverging for sequential circuit machines with an equivalent signature.

Lemma 5.2.4. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((1,0,1),(\infty,\mathcal{L}))$, $\text{SafeLtlSat}_{\widehat{\sigma}}$ is LogSpace-reducible to $\text{Diverging}_{\widehat{\sigma}}$.*

With the following lemma, we establish the opposite direction:

Lemma 5.2.5. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((1,0,1),(\infty,\mathcal{L}))$, $\text{Diverging}_{\widehat{\sigma}}$ is LogSpace-reducible to $\text{SafeLtlSat}_{\widehat{\sigma}}$.*

Proof. For a given sequential circuit machine $\mathcal{S} = (\sigma, C_{\mathsf{A}})$, we describe a LogSpace reduction that constructs a set of atomic proposition AP and an LTL formula $\varphi \in \text{SafeLTL}(AP)$ such that

$$sig(\text{SafeLtlSat}, (AP, \varphi)) = \sigma$$

and $\text{SafeLtlSat}(AP, \varphi) = \textbf{yes}$ iff $\mathcal{S}$ diverges. Since, according to Corollary 4.2.7, $\mathcal{C}((1,0,1),(\infty,\mathcal{L})) = \mathcal{C}((1,0,0),(\infty,\mathcal{P}))$, we can also assume $\sigma = ((1,0,0),(\infty,m_{\mathsf{A}}))$ with m_{A} being polynomial in the size of AP and φ.

The basic idea of our reduction is to use AP to represent the universal memory and to use φ to encode the semantics of $\mathcal{S}$. Note that, without loss of generality, we assume that C_{A} can be represented by a Boolean formula, which can thus be embedded into φ. Formally, we define $AP = M_{\mathsf{A}}$ and

$$\varphi \equiv \mathsf{G} \bigvee_{e\in\{0,1\}} \Big(\neg C_{\mathsf{A}}(\mathsf{h}) \quad \wedge \quad \bigwedge_{a\in M_{\mathsf{A}}} (\mathsf{X}a \Leftrightarrow C_{\mathsf{A}}(a)) \Big),$$

where $C_{\mathsf{A}}(x)$ is the Boolean formula with free variables $M_{\mathsf{A}} \cup \{e\}$ that computes $x \in M_{\mathsf{A}} \cup \{\mathsf{h}\}$. Furthermore, note that the negation in front of the next operator does not do any harm since $\neg \mathsf{X} a \equiv \mathsf{X} \neg a$. □

The combination of Lemma 5.2.4 and Lemma 5.2.5 gives the desired connection between LTL and SCMs:

Theorem 5.2.6. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((1,0,1),(\infty,\mathcal{L}))$,* SafeLtlSat$_{\widehat{\sigma}}$ *and* Diverging$_{\widehat{\sigma}}$ *are* LogSpace*-reducible to each other.*

Finally, the combination of Theorem 5.2.6 and Theorem 4.2.5 yields the complexity for satisfiability of the syntactic safety fragment of LTL:

Corollary 5.2.7. SafeLtlSat *is complete for $\mathcal{C}((1,0,1),(\infty,\mathcal{L}))$, that is* PSpace*-complete, even when we assume that formulas have a maximal temporal depth of 2.*

5.2.4 Bibliographic Remarks

The PSpace-completeness of LTL satisfiability was first established by Sistla and Clarke [1985]. The more precise result of the PSpace-hardness even when formulas may only have a temporal depth of at most 2 is due to Demri and Schnoebelen [2002] [see Schnoebelen, 2002, for a comprehensive summary]. However, our hardness proof (i.e., the proof of Lemma 5.2.5) is much more succinct than the classical proof.

5.3 Further Succinct Formalisms

In this section, we discuss the relationship of sequential circuit machines to some other well-known formalisms from the literature.

Boolean automata. A *Boolean automaton* is a finite state automaton extended with Boolean variables. The variables may be referenced in constraints that are used as guards defining when an edge can be executed. When executing an edge, the values of some variables can be set to a new value.

The correspondence between sequential circuit machines and Boolean automata is obvious: One can use the Boolean variables to represent the values of the various circuit elements. Combinatorial circuits (for which we assume that they are simple, without loss of generality) are represented as guards. Now, when we ask for the complexity of (un)bounded (alternating) reachability or safety under no, partial, or full observability, respectively, instead of establishing every single result individually, we just need to carry over all the results for sequential circuit machines from Section 4.2.

We note that Laroussinie and Schnoebelen [2000] first showed that, for checking bisimilarity, there is a complexity-theoretic connection between Boolean automata, timed automata, and one-safe Petri nets. Later, Chadha et al. [2010] made the connection between Boolean automata and the more general lifting lemma by Lozano and Balcázar [1989] for establishing the complexity of problems defined on succinct instances.

Propositional planning. For a given set of discrete state variables, a *planning task* describes update rules in form of $p \to q$, where p is a conjunctive condition on the values of the state variables and q is a list of variable assignments. A solution to a given planning task is a *plan* that defines an execution sequence of rules so that some goal state is reached from a given initial variable valuation.

Again, the correspondence to sequential circuit machines is obvious: One can use the state variables to model the values of the circuit elements, and the choice of the rules to model the existential nondeterminism. Combinatorial circuits (for which we assume that they are simple, without loss of generality) are represented using the preconditions of the update rules. In the presence of *nondeterministic operators* (i.e., in an alternating planning setting), we just need to add universal nondeterminism in the sequential circuit machine. The extension to partial information and/or boundedness can also be achieved easily.

We note that Bylander [1994] established the PSpace-completeness of the basic version of the problem (with only one player and no bounds). The extension to the alternating case was considered by Littman [1997], where the ExpTime-completeness result was established. Bäckström and Nebel [1995] investigated the impact of imposing a (polynomial, i.e., unary-encoded) bound on the length of the plan and established several completeness results between NPTime and PSpace. The extension to the setting of partial information was investigated by Rintanen [2004].

Now, by carrying over the complexity results from Section 4.2, we are able to extend that line of research by, e.g., introducing the NExpTime-completeness of deciding the existence of deterministic or nondeterministic bounded plans under no, partial, or full observability, respectively, where the bound is encoded in binary.

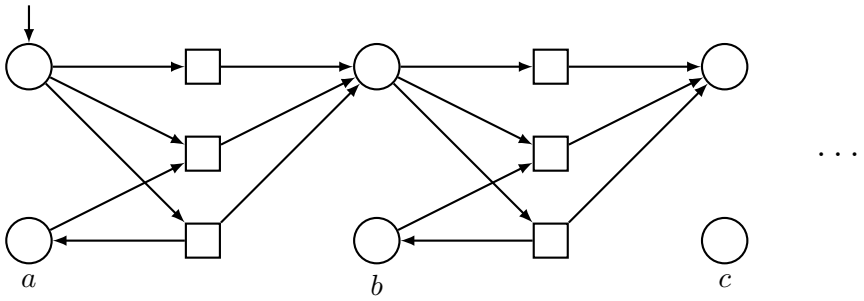
One-safe Petri nets. *Petri nets* [see Esparza, 1996, for a survey] are a popular formalism for modeling concurrent systems. A Petri net is basically a directed graph with two kinds of nodes: *places* and *transitions*. Two places are connected via a transition. The purpose of places is to accumulate *tokens*, while transitions remove tokens from their source places and add new tokens to their target places. An execution of a Petri net is a sequence of *transition firings*. Now, *one-safe Petri nets* are a special case, where each

place may only accumulate a single token.

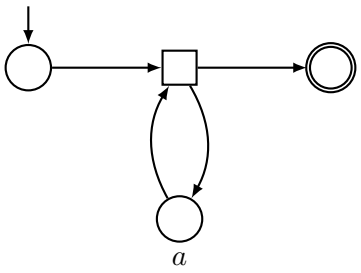
The connection to sequential circuit machines can be established by representing the value of the circuit elements as tokens on places. Combinatorial circuits (for which we assume that they are simple, without loss of generality) are simulated as a propagation of a value through a Boolean expression, which corresponds to propagating a token through a directed acyclic net. Figure 5.3 depicts the important constructions for simulating computation steps of sequential circuit machines by transition firings. The hardness proof (that shows that one-safe Petri nets can simulate sequential circuit machines) uses these constructions as modular widgets, which are connected via their entry places (drawn as circles with an incoming arrow) and exit places (drawn as circles with a double border).

The PSpace-completeness of deciding whether a state (including deadlocks) is reachable in a given one-safe Petri net was shown by Cheng et al. [1995]. The ExpTime-completeness of the global controller synthesis problem (i.e., the fully informed alternating extension) was established by Katz et al. [2011].

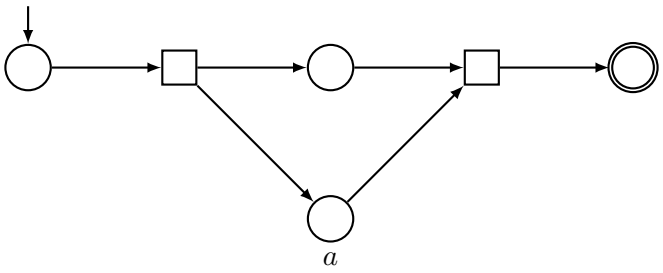
Again, by carrying over the complexity results from Section 4.2, we are able to fill in the missing gaps. For instance, to the best of the author's knowledge, there has not been a result published on the bounded synthesis problem for one-safe Petri nets: Having the connection to sequential circuit machines, we can immediately obtain the PSpace- and NExpTime-completeness for both the smallest witness and the bounded synthesis problem for unary and binary encodings of the bound, respectively.



(a) Nondeterministic choice of the values for the circuit elements a, b, c.



(b) A token reaches the exit place iff circuit element a has value 1.



(c) A token reaches the exit place iff circuit element a has value 0.

Figure 5.3: Constructions for simulating sequential circuit machines using one-safe Petri nets.

Part II

The Succinctness of Timed Automata

Chapter 6

Controllable Timed Automata

In this chapter, we recall the timed automaton formalism by Alur and Dill [1994]. We also introduce the extension to the turn-based, alternating case assuming full [Maler et al., 1995, Asarin et al., 1998] and partial information [Bouyer et al., 2003, Bouyer and Chevalier, 2006].

For a comprehensive and recent survey on timed automata, we refer to Waez et al. [2011].

6.1 Syntax

Intuitively, timed automata extend finite state machines with real-valued clock variables.

6.1.1 Granularity and Constraints

The *granularity* of a timed automaton defines its *timing resources*, i.e., the clocks and the constants against which the clocks are compared to [D'Souza and Madhusudan, 2002]. Formally, a granularity is represented by a tuple $\mu = (X, m, c^{\mathsf{max}})$, where X is a finite set of real-valued clocks, $m \in \mathbb{N}_{\geq 1}$, and $c^{\mathsf{max}} \in \mathbb{Q}_{\geq 0}$. We call the value of a clock $x \in X$ *maximal* if it is strictly greater than c^{max}. The *combination* of two granularities $\mu_1 = (X_1, m_1, c_1^{\mathsf{max}})$ and $\mu_2 = (X_2, m_2, c_2^{\mathsf{max}})$ is defined as

$$\mu_1 \otimes \mu_2 = \big(X_1 \cup X_2,\ \mathrm{lcm}(m_1, m_2),\ \max(c_1^{\mathsf{max}}, c_2^{\mathsf{max}})\big).$$

We say that μ_1 is finer than or equal to μ_2, written as $\mu_1 \leq \mu_2$, iff

$$X_1 \supseteq X_2 \ \wedge\ m_1 \geq m_2 \ \wedge\ c_1^{\mathsf{max}} \geq c_2^{\mathsf{max}}.$$

We say that μ_1 is strictly finer (or just finer) than μ_2, written as $\mu_1 < \mu_2$, iff $\mu_1 \leq \mu_2$ and $\mu_1 \neq \mu_2$.

A (rectangular) *clock constraint* $\varphi \in \mathsf{CC}(\mu)$ is of the form

$$\textbf{true} \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2,$$

where x is a clock from X, φ_1 and φ_2 are clock constraints from $\mathsf{CC}(\mu)$, and c is a constant from $\mathbb{Q}_{\geq 0}$ that satisfies the following constraints: (1) c can be represented as $c = k \cdot m^{-1}$, for some $k \in \mathbb{N}$, and (2) c is less than or equal to c^{max}. We write $\mathsf{CC}_{\leq}(\mu)$ to refer all constraints φ of the form

$$\textbf{true} \mid x \leq c \mid \varphi_1 \wedge \varphi_2,$$

where x, c, φ_1, and φ_2 are defined analogously to the definition from above. We assume that constants are always encoded in binary unless stated otherwise.

6.1.2 Timed Automata

For a granularity $\mu = (X, m, c^{\mathsf{max}})$, a *$\mu$-granular timed automaton* $\mathcal{T}$ is a tuple (Q, q_0, Σ, E, I), where

- Q is a finite set of *control locations*,
- $q_0 \in Q$ is the initial location,
- Σ is a finite set of *events*,
- $E \subseteq Q \times \Sigma \times \mathsf{CC}(\mu) \times 2^X \times Q$ is an edge relation defining the *control structure* of $\mathcal{T}$, and
- $I : Q \to \mathsf{CC}_{\leq}(\mu)$ is a total function that assigns an *invariant* to each location.

Sometimes, for the sake of simplifying the illustration, we also use disjunctions in clock guards, which can be easily resolved by splitting the corresponding edge. We write $\mathcal{T}_\mu$ to refer to μ. Without loss of generality, we assume that it is syntactically ensured that taking an edge always ends up in a location whose invariant is satisfied. We require timed automata to be *event-deterministic*: for any two edges $e_1 = (q_1, a_1, \varphi_1, r_1, q_1')$ and $e_2 = (q_2, a_2, \varphi_2, r_2, q_2')$ from E, we require

$$(q_1 = q_2 \wedge a_1 = a_2) \Rightarrow (e_1 = e_2 \ \vee \ \varphi_1 \wedge \varphi_2 \equiv \textbf{false}).$$

Example 6.1.1. *Figure 6.1 shows an example timed automaton with four locations and two clocks. While the locations l_1 and l_3 have invariants $y \leq 5$ and $y \leq 15$, the other locations l_2 and l_4 have no invariants (i.e., they have the invariant* **true***).*

Intuitively, the execution starts in l_1 with $x = y = 0$. Due to l_1's invariant, the execution has to leave l_1 within 5 time units. There is an edge

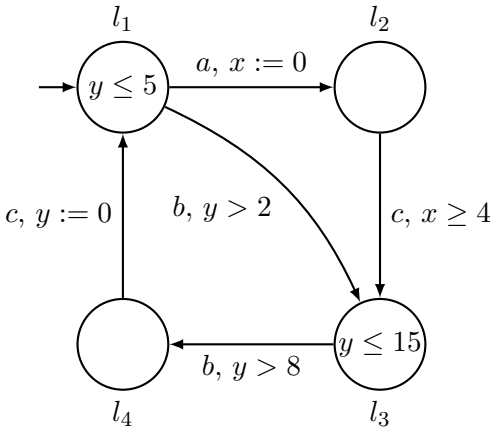


Figure 6.1: Example timed automaton with clocks x and y, and events a, b, and c.

leading to l_2, which is labeled with event a and resets the clock x. Another edge, labeled with event b and performing no resets, leads to l_3, and, due to its guard $y > 2$, can only be taken after strictly 2 time units. In this manner, the execution continues either at l_2 or l_3.

We refer to Section 6.2 for a formal definition of the semantics of timed automata.

6.1.3 Networks

Similar to communicating state machines, timed automata can be syntactically composed into *networks*, in which the automata run in parallel and synchronize on shared events. For a μ_1-granular timed automaton $\mathcal{T}_1 = (Q_1, q_0^1, \Sigma_1, E_1, I_1)$ and a μ_2-granular timed automaton $\mathcal{T}_2 = (Q_2, q_0^2, \Sigma_2, E_2, I_1)$, we define the *parallel composition* of $\mathcal{T}_1$ and $\mathcal{T}_2$, written as $\mathcal{T}_1 \| \mathcal{T}_2$, as the $(\mu_1 \otimes \mu_2)$-granular timed automaton $(Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, E, I)$, where $I(q_1, q_2) = I_1(q_1) \wedge I_2(q_2)$, for all $q_1 \in Q_1$ and $q_2 \in Q_2$, and E is the smallest set that satisfies the following conditions:

$$
\begin{aligned}
E \supseteq \{&((q_1, q_2), a, \varphi_1 \wedge \varphi_2, r_1 \cup r_2, (q_1', q_2')) \mid \\
&\quad a \in \Sigma_1 \cap \Sigma_2 \wedge \\
&\qquad (q_1, a, \varphi_1, r_1, q_1') \in E_1 \wedge (q_2, a, \varphi_2, r_2, q_2') \in E_2\}; \\
E \supseteq \{&((q_1, q_2), a, \varphi_1, r_1, (q_1', q_2)) \mid \\
&\quad a \in \Sigma_1 \setminus \Sigma_2 \wedge (q_1, a, \varphi_1, r_1, q_1') \in E_1\}; \\
E \supseteq \{&((q_1, q_2), a, \varphi_2, r_2, (q_1, q_2')) \mid \\
&\quad a \in \Sigma_2 \setminus \Sigma_1 \wedge (q_2, a, \varphi_2, r_2, q_2') \in E_2\}.
\end{aligned}
$$

Example 6.1.2. *Figure 6.2 shows a network of two timed automata. They synchronize on the events a and c: Provided that the clock guards are satisfied, an automaton can only execute an edge labeled with an event a or c if the other automaton executes an edge with the same label concurrently. Otherwise, the two automata run asynchronously: They can independently execute edges labeled with an event b or d.*

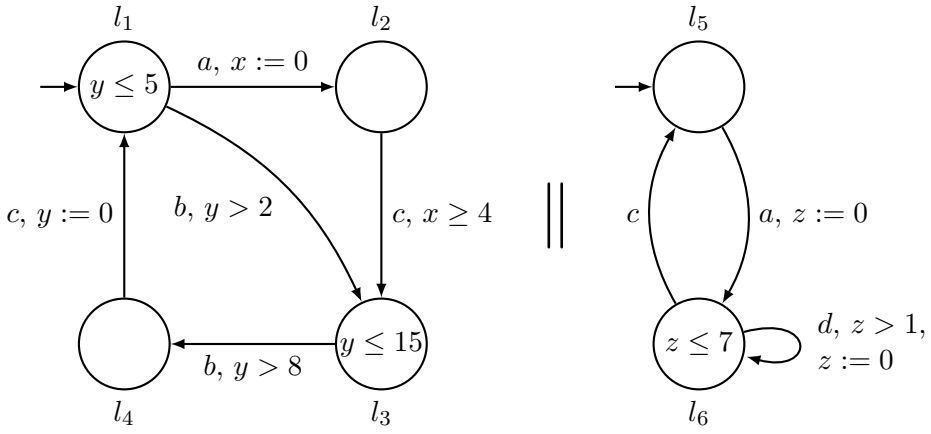


Figure 6.2: Example network comprising two timed automata that synchronize on events a and c.

6.2 Infinite Semantics

Due to the continuous value domain of the clocks, in the analysis of timed automata, one has to deal with an uncountable number of states.

6.2.1 Timed States and Transitions

For a set of clocks X, a *clock valuation* $\vec{t} : X \to \mathbb{R}_{\geq 0}$ assigns a nonnegative value to each clock and can also be represented by a $|X|$-dimensional vector $\vec{t} \in \mathbb{R}^X_{\geq 0}$. For a $d \in \mathbb{R}_{>0}$, we write $\vec{t} + d$ as a shorthand for a vector $\vec{t}'$, where $\vec{t}'(x) = \vec{t}(x) + d$, for every $x \in X$. Also, for an $r \subseteq X$, we write $\vec{t}[r := 0]$ as a shorthand for a vector $\vec{t}'$, where

$$\vec{t}'(x) = \begin{cases} 0 & \text{if } x \in r; \\ \vec{t}(x) & \text{otherwise.} \end{cases}$$

For a given timed automaton $\mathcal{T} = (Q, q_0, \Sigma, E, I)$, a *timed state* of $\mathcal{T}$ is a tuple $(q, \vec{t})$ comprising a location $q \in Q$ and a clock valuation $\vec{t} \in \mathbb{R}^X_{\geq 0}$.

Two timed states $s = (q, \vec{t})$ and $s' = (q', \vec{t'})$ are connected via the following two kinds of *timed transitions*:

(1) *Discrete transitions*: for an edge $(q, d, \varphi, r, q') \in E$, there is a discrete transition between s and s', written as $s \xrightarrow{a} s'$, if $\vec{t} \models \varphi$, $\vec{t'} = \vec{t}[r := 0]$, and $\vec{t'} \models I(q')$.

(2) *Delay transitions*: for a delay $d \in \mathbb{R}_{>0}$, there is a delay transition between s and s', written as $s \xrightarrow{d} s'$, if $q' = q$, $\vec{t'} = \vec{t} + d$, and $\vec{t'} \models I(q)$;

A situation in which a further progress of time would violate the current location invariant but no edges are enabled is called a *timelock*. The set of all timelock states is defined as

$$\mathsf{Timelock}(\mathcal{T}) = \big\{(q, \vec{t}) \mid \forall \epsilon > 0 : \vec{t} + \epsilon \not\models I(q) \wedge \neg\exists a \, \exists q' \, \exists \vec{t'} : (q, \vec{t}) \xrightarrow{a} (q', \vec{t'})\big\}.$$

6.2.2 Infinite Game Arena

The infinite semantics of a controllable timed automaton $\mathcal{T}$ with $\mathcal{T} = (Q, q_0, \Sigma, E, I)$ and $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$ is formally described as an infinite game arena $[\![\mathcal{T}]\!] = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$, where

- $S_{\mathsf{E}} = \{\mathsf{E}\} \times \mathbb{R}_{>0} \times (Q \times \mathbb{R}^X_{\geq 0})$,
- $S_{\mathsf{A}} = \{\mathsf{A}\} \times \mathbb{R}_{>0} \times (Q \times \mathbb{R}^X_{\geq 0})$,
- $s_0 = (\mathsf{A}, 0, (q_0, \vec{0}))$,
- $\Sigma_{\mathsf{E}} = \Sigma_{\mathsf{in}} \uplus \mathbb{R}_{>0}$,
- $\Sigma_{\mathsf{A}} = \Sigma_{\mathsf{out}} \uplus \mathbb{R}_{>0}$,
- $\Delta((p, d, t), m) = s'$, where

$$s' = \begin{cases} (\mathsf{A}, 0, t') & \text{if } p = \mathsf{A},\ m \in \Sigma_{\mathsf{A}} \text{ and } t \xrightarrow{m} t'; \\ (\mathsf{E}, m, t) & \text{if } p = \mathsf{A} \text{ and } m \in \mathbb{R}_{>0}; \\ (\mathsf{E}, d - m, t') & \text{if } p = \mathsf{E},\ m \in \mathbb{R}_{>0},\ m < d \text{ and } t \xrightarrow{m} t'; \\ (\mathsf{A}, 0, t') & \text{if } p = \mathsf{E},\ m \in \mathbb{R}_{>0},\ m \geq d \text{ and } t \xrightarrow{d} t'; \\ (\mathsf{A}, 0, t') & \text{if } p = \mathsf{E},\ m \in \Sigma_{\mathsf{E}} \text{ and } t \xrightarrow{m} t'. \end{cases}$$

For a set of states $Y \subseteq Q \times \mathbb{R}^X_{\geq 0}$, we write $[\![\mathcal{T}]\!] \models \mathsf{AG}(Y)$ as a shorthand for $[\![\mathcal{T}]\!] \models \mathsf{AG}(\{\mathsf{E}, \mathsf{A}\} \times \mathbb{R}_{>0} \times Y)$.

6.3 Controller Synthesis

6.3.1 Plants and Controllers

A (X, m, c^{max})-granular timed automaton $P = (Q, q_0, \Sigma, E, I)$ is called a *(timed) plant* if it satisfies the following conditions:

(1) P has an *input/ouput interface*: the events Σ are partitioned into controllable events $\Sigma_{\mathsf{in}} \subseteq \Sigma$ (over which P can be influenced) and into uncontrollable events $\Sigma_{\mathsf{out}} \subseteq \Sigma$ (which are emitted by P and can be observed externally);

(2) P is *input enabled*: for each $q \in Q$ and each $a \in \Sigma_{\mathsf{in}}$, we require that $I(q) \;\Rightarrow\; \Big(\bigvee_{(q,d,\varphi,r,q') \in E} \varphi\Big)$.

A $(X_c, m_c, c_c^{\mathsf{max}})$-granular timed automaton $C = (Q_c, q_0^c, \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}}, E_c)$ is called a *controller* for P if it satisfies the following conditions:

(1) C is *nonintrusive*: for each $(q, d, \varphi, r, q') \in E_c$ we have $r \cap X = \emptyset$;

(2) C is *nonblocking*:
if $[\![P]\!] \models \mathsf{AG}(\overline{\mathsf{Timelock}(P)})$ then $[\![P \| C]\!] \models \mathsf{AG}(\overline{\mathsf{Timelock}(P \| C)})$;

(3) C is *nonrestricting*: for each $\pi \in \mathsf{Traces}([\![P \| C]\!])$ and each $1 \leq i < |\pi|$ with $\pi[1..i] \in \mathsf{Traces}([\![P]\!])$ and $\pi[i{+}1] \in \Sigma_{\mathsf{out}}$, we require that $\pi[1..i{+}1] \in \mathsf{Traces}([\![P \| C]\!])$.

6.3.2 Problem Definition

We now formally define the *safety controller synthesis problem for controllable timed automata* Timed synthesis.

An input to Timed synthesis is a tuple $(P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, q^{\mathsf{max}})$, where

- $P = (Q, q_0, \Sigma, E, I)$ is a plant with $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$,
- $\Sigma_{\mathsf{out}}^{\mathsf{obs}} \subseteq \Sigma_{\mathsf{out}}$ are the observable output events,
- $q_b \in Q$ is a dedicated bad location,
- $\mu_c = (X_c, m_c, c_c^{\mathsf{max}})$ is the controller granularity, and
- $q^{\mathsf{max}} \in \mathbb{N} \cup \{\infty\}$ is a bound on the number of locations of the controller, given in binary unless stated otherwise.

Definition 6.3.1. *For a given input $(P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, q^{\mathsf{max}})$, the problem* Timed synthesis *is to check whether there exists a timed automaton $C = (Q_c, q_0^c, \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}}, E_c)$ that satisfies the following conditions:*

(1) C is a μ_c-granular controller for P;

(2) C satisfies the location bound: $|Q_c| \leq q^{\mathsf{max}}$*;*

(3) C is safe: $[\![P \| C]\!] \models \mathsf{AG}(\overline{\{q_b\}} \times Q_c \times \mathbb{R}^X_{\geq 0})$.

For convenience, we define $X^{\mathsf{obs}} = X \cap X_c$ as the observable clocks, $\Sigma^{\mathsf{unobs}}_{\mathsf{out}} = \Sigma \setminus (\Sigma_{\mathsf{in}} \cup \Sigma^{\mathsf{obs}}_{\mathsf{out}})$ as the unobservable events, and $X^{\mathsf{unobs}} = X \setminus X^{\mathsf{obs}}$ as the unobservable clocks. We say that TIMED SYNTHESIS is *under full observability* if $X^{\mathsf{unobs}} = \emptyset$ and $\Sigma^{\mathsf{unobs}}_{\mathsf{out}} = \emptyset$. We say that TIMED SYNTHESIS is *under no observability* if $X^{\mathsf{obs}} = \emptyset$ and $\Sigma^{\mathsf{obs}}_{\mathsf{out}} = \emptyset$.

6.4 Finite Semantics

The decidability of the (alternating) reachability problem of timed automata relies on the existence of the *region equivalence relation* [Alur and Dill, 1994] on $\mathbb{R}^X_{\geq 0}$, which has a finite index.

6.4.1 The Region Abstraction

In the following, we fix a μ-granular timed automaton $\mathcal{T}$ with $\mu = (X, m, c^{\mathsf{max}})$ and $\mathcal{T} = (Q, q_0, \Sigma, E, I)$. We say that two clock valuations $\vec{t}_1, \vec{t}_2 \in \mathbb{R}^X_{\geq 0}$ are in the same *clock region*, denoted $\vec{t}_1 \sim_\mu \vec{t}_2$, if the following conditions are satisfied:

- the set of clocks with maximal value is the same in $\vec{t}_1$ and in $\vec{t}_2$: $\forall x \in X : \vec{t}_1(x) > c^{\mathsf{max}} \Leftrightarrow \vec{t}_2(x) > c^{\mathsf{max}}$;
- $m \cdot \vec{t}_1$ and $m \cdot \vec{t}_2$ agree (1) on the integer parts of the clock values, (2) on the relative order of the fractional parts of the clock values, and (3) on the equality of the fractional parts of the clock values with 0. That is, for all clocks x and y in X with nonmaximal value, it holds that
  (1) $\lfloor m \cdot \vec{t}_1(x) \rfloor = \lfloor m \cdot \vec{t}_2(x) \rfloor$,
  (2) $fr(m \cdot \vec{t}_1(x)) \leq fr(m \cdot \vec{t}_1(y)) \Leftrightarrow fr(m \cdot \vec{t}_2(x)) \leq fr(m \cdot \vec{t}_2(y))$, and
  (3) $fr(m \cdot \vec{t}_1(x)) = 0$ iff $fr(m \cdot \vec{t}_2(x)) = 0$,
  where $fr(m \cdot \vec{t}_i(x)) = m \cdot \vec{t}_i(x) - \lfloor m \cdot \vec{t}_i(x) \rfloor$ for $i \in \{1, 2\}$.

The set of all clock regions of a granularity μ is defined as $[\mu]$. We denote by

$$[\vec{t}]_\mu = \{\vec{t}' \in \mathbb{R}^X_{\geq 0} \mid \vec{t} \sim_\mu \vec{t}'\}$$

the clock region $\vec{t} \in \mathbb{R}^X_{\geq 0}$ belongs to. We say that two timed states $s_1 = (q_1, \vec{t}_1)$ and $s_2 = (q_2, \vec{t}_2)$ of $\mathcal{T}$ are *region-equivalent*, denoted by $s_1 \sim_\mu s_2$, if (1) their locations are the same: $q_1 = q_2$; and (2) the clock valuations are in the same clock region: $\vec{t}_1 \sim_\mu \vec{t}_2$. For a timed state $s \in Q \times \mathbb{R}^X_{\geq 0}$, we denote by

$$[s]_\mu = \{s' \in Q \times \mathbb{R}^X_{\geq 0} \mid s \sim_\mu s'\}$$

the equivalence class of region-equivalent states (or just the *region*) that s belongs to. The set of all regions of $\mathcal{T}$ is defined as $[\mathcal{T}]_\mu = Q \times [\mu]$. Alur and Dill [1994] showed that, the number of regions for a given timed automaton is linear in its locations and exponential in its granularity:

Lemma 6.4.1. *[Alur and Dill, 1994] For a μ-granular timed automaton $\mathcal{T} = (Q, q_0, \Sigma, E, I)$ with $\mu = (X, m, c^{\mathsf{max}})$, we have the following upper bound on its number of regions:*

$$\begin{aligned}|[\mathcal{T}]_\mu| &\leq |Q| \cdot |X|! \cdot 2^{|X|-1} \cdot \prod_{x \in X} O(m \cdot c^{\mathsf{max}}) \\ &= |Q| \cdot |X|! \cdot O(m \cdot c^{\mathsf{max}})^{|X|}\end{aligned}$$

For convenience, for a set of timed states $S \subseteq Q \times \mathbb{R}^X_{\geq 0}$, we write

$$[S]_\mu = \{r \in [\mathcal{T}]_\mu \mid r \subseteq S\}$$

to refer to the set of regions that constitute S, and for a location $q \in Q$, we write

$$[q]_\mu = \{r \in [\mathcal{T}]_\mu \mid \forall (q', \vec{t}) \in r : q' = q\}$$

to refer to the set of regions with location q. Note that, in the rest of this thesis, we always assume that each set of timed states S is expressible as a partitioning of regions $r_1, \ldots, r_n \in [\mathcal{T}]_\mu$ of the form $S = \biguplus_{1 \leq i \leq n} r_i$. For a coarser granularity $\mu' \geq \mu$ with clocks $X' \subseteq X$ and for a timed state $(q, \vec{t}) \in Q \times \mathbb{R}^X_{\geq 0}$, by abuse of notation, we define

$$\begin{aligned}[(q, \vec{t})]_{\mu'} = \{(q', \vec{t}') \in Q \times \mathbb{R}^X_{\geq 0} \mid q' = q \wedge \\ \exists \vec{t}'' \in [\vec{t}\,\downharpoonright X']_{\mu'} : \forall x \in X' : \vec{t}'(x) = \vec{t}''(x)\}\end{aligned}$$

as the *widening* of $(q, \vec{t})$ on μ'.

Regions are a suitable semantics for the abstraction of timed automata because they essentially preserve the time-abstracted behavior: If there is a discrete transition $s \xrightarrow{a} s'$ from a state s to a state s' of a timed automaton, then there is, for all states t with $t \sim_\mu s$, a state t' with $t' \sim_\mu s'$ such that $t \xrightarrow{a} t'$ is a discrete transition with the same event. For delay transitions, a slightly weaker property holds: If there is a delay transition $s \xrightarrow{d} s'$ from a state s to a state s', then there is, for all states t with $t \sim_\mu s$, a state t' with $t' \sim_\mu s'$ such that there is a timed transition $t \xrightarrow{d'} t'$ (but possibly with $d' \neq d$).

Formally, two regions $r, r' \in [\mathcal{T}]_\mu$ are connected via the following two kinds of steps:

(1) *Discrete steps*: If

$$\forall s \in r : \exists a \in \Sigma : \exists s' \in r' : s \xrightarrow{a} s'$$

then there is a discrete step between r and r', written as $r \xrightarrow{a} r'$.

(2) *Delay steps*: If

$$\forall s \in r : \exists d \in \mathbb{R}_{>0} : \exists s' \in r' : s \xrightarrow{d} s' \wedge \forall d' < d : s + d' \in r \cup r'$$

then there is a delay step between r and r', written as $r \xrightarrow{\tau} r'$.

Timed automata are *time-deterministic*: any given region has exactly one successor region, except for the region in which all clock values are maximal, which has no successor.

For a granularity $\mu' \geq \mu$ and a region $r \in [\mu]$, we define

$$r \models \mu' \ :\Longleftrightarrow\ \forall (q, \vec{t}) \in r : \forall d \in \mathbb{R}_{>0} : (q, \vec{t} - d) \notin [(q, \vec{t})]_{\mu'} \setminus r,$$

Intuitively, $r \models \mu'$ if, and only if, entering r by letting time pass is observable through granularity μ'.

6.4.2 Finite Game Arena

For a granularity $\mu = (X, m, c^{\mathsf{max}})$, the μ-granular finite semantics of a μ'-granular controllable timed automaton $\mathcal{T} = (Q, q_0, \Sigma, E, I)$ with $\mu' \leq \mu$ and events $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$ is formally described as the finite game arena $[\![\mathcal{T}]\!]_\mu = (S_{\mathsf{E}}, S_{\mathsf{A}}, s_0, \Sigma_{\mathsf{E}}, \Sigma_{\mathsf{A}}, \Delta)$, where

- $S_{\mathsf{E}} = \{\mathsf{E}\} \times [\mathcal{T}]_\mu$,
- $S_{\mathsf{A}} = \{\mathsf{A}, \mathsf{D}\} \times [\mathcal{T}]_\mu$,
- $s_0 = (\mathsf{A}, [(q_0, \vec{0})]_\mu)$,
- $\Sigma_{\mathsf{E}} = \Sigma_{\mathsf{in}} \uplus \{\tau\}$,
- $\Sigma_{\mathsf{A}} = \Sigma_{\mathsf{out}} \uplus \{\tau\} \uplus [\mu]$,
- $\Delta((p, r), m) = s'$, where

$$s' = \begin{cases} (\mathsf{E}, r) & \text{if } p = \mathsf{A} \text{ and } m = \tau; \\ (\mathsf{D}, r) & \text{if } p = \mathsf{E} \text{ and } m = \tau; \\ (\mathsf{A}, r') & \text{if } p = \mathsf{D},\ m = r' \text{ and } r \xrightarrow{\tau} r'; \\ (\mathsf{A}, r') & \text{if } m \in \Sigma \text{ and } r \xrightarrow{m} r'. \end{cases}$$

For the one-player case, we have the following classical result:

Lemma 6.4.2. *[Alur and Dill, 1994] For a μ-granular timed automaton $\mathcal{T}$ and a set of timed states Y, the following statements hold:*

- $[\![\mathcal{T}]\!] \models \mathsf{EF}(Y)$ *if, and only if,* $[\![\mathcal{T}]\!]_\mu \models \mathsf{EF}([Y]_\mu)$*;*
- $[\![\mathcal{T}]\!] \models \mathsf{EG}(Y)$ *if, and only if,* $[\![\mathcal{T}]\!]_\mu \models \mathsf{EG}([Y]_\mu)$*;*
- $[\![\mathcal{T}]\!] \models \mathsf{AF}(Y)$ *if, and only if,* $[\![\mathcal{T}]\!]_\mu \models \mathsf{AF}([Y]_\mu)$*;*
- $[\![\mathcal{T}]\!] \models \mathsf{AG}(Y)$ *if, and only if,* $[\![\mathcal{T}]\!]_\mu \models \mathsf{AG}([Y]_\mu)$*.*

Note that the successor of a state (p, r), where $p \in \{\mathsf{E}, \mathsf{A}\}$ and $r \in [\mathcal{T}]_\mu$, is determined by Player p, who chooses among polynomially many moves. Due to the fact that timed automata are time-deterministic, the successor of a state (D, r) is determined deterministically (and does not depend on a choice of a particular player).

6.4.3 A Game-Theoretic Solution to Controller Synthesis

An input $i = (P, \Sigma^{\mathsf{obs}}_{\mathsf{out}}, q_b, \mu_c, q^{\mathsf{max}})$ to Timed synthesis induces a finite safety game $\mathcal{G}(\textsc{Timed synthesis}, i) = (\mathcal{A}, B, \mathcal{V}, \beta)$, which is defined as follows:

- $\mathcal{A} = [\![\mathcal{T}]\!]_{\mu'}$ with the combined granularity $\mu' = \mathcal{T}_\mu \otimes \mu_c$.
- $B = \{\mathsf{E}, \mathsf{A}, \mathsf{D}\} \times ([q_b]_{\mu'} \cup [\mathsf{Timelock}(\mathcal{A})]_{\mu'})$.
- The fact that the controller can only observe some clocks and some events emitted by $\mathcal{T}$ is reflected in the definition of the view $\mathcal{V}$ of Player E on $\mathcal{A}$:

  $$\mathcal{V} = \big(\Sigma^{\mathsf{obs}}_{\mathsf{out}} \uplus \{\tau\}, \mathit{vis}\big),$$

  where

  $$\mathit{vis}(m) = \begin{cases} m & \text{if } m \in \Sigma^{\mathsf{obs}}_{\mathsf{out}}; \\ \tau & \text{if } m \in [\mu'] \text{ and } m \models \mu_c; \\ \varepsilon & \text{otherwise.} \end{cases}$$

- $\beta = q^{\mathsf{max}} \cdot |[\mu_c]|$ is the existential memory bound.

Indeed, solving this safety game is a decision procedure for Timed synthesis.

Theorem 6.4.3. *[Bouyer et al., 2003] For a given input x,*
Timed synthesis$(x) =$ **yes** *iff*
Player E wins $\mathcal{G}(\textsc{Timed synthesis}, x)$.

Chapter 7

The Complexity of Timed Controller Synthesis

This chapter investigates the complexity of several important analysis problems for timed automata.[1]

We first present two ways to exploit the succinctness of timed automata to simulate the manipulation of bits in Section 7.1. Based on that, we make the connection to sequential circuit machines in Section 7.2. We are then ready to establish complexity results in Section 7.3.

7.1 Using Clocks to Represent Bits

In this section, we will present how to represent a set of bits B using clocks assuming a unary or a binary encoding of the constants. We will also show the construction of two widgets $\mathsf{TEST}(b, v)$ and $\mathsf{SET}(b, v)$, $v \in \{0, 1\}$, for testing and setting a particular bit $b \in B$, respectively. Later, in the proof of Lemma 7.2.3, we abstract from the actual representation of the bits by treating the two widgets as partial black-box timed automata that can be embedded into the control structure of another timed automaton.

Independent of the actual representation, we always assume that a $\mathsf{TEST}(b, v)$ widget has a unique entry location and two exit locations: (1) a **false** location that is entered when the value of b not equals v, and (2) a **true** location that is entered when the value of b equals v. For a $\mathsf{SET}(b, v)$ widget, we always assume that there is a unique entry and a unique exit location. As a general pre- and postcondition, we always assume that the bits are in their respective normal form representation before or after executing a widget. Furthermore, we will construct the widgets in such a way that they are completely deterministic.

[1] The results of this chapter on bounded synthesis for timed automata have been published in [Peter and Finkbeiner, 2012].

7.1.1 Unary Encoding of Constants

In this subsection, we will demonstrate how to represent and manipulate an array of n bits using $n + 1$ clocks. The constraints in the TEST and SET widgets only use the constants 0 and 1.

For a set of bits $B = \{b_1, \ldots, b_n\}$, we introduce a set of clocks $X = \{x_1, \ldots, x_n, z\}$, where z is an auxiliary clock. For a clock valuation $\vec{x} \in \mathbb{R}^X_{\geq 0}$ and a bit valuation $\vec{b} \in \vec{B}$, we say that $\vec{x}$ *represents* $\vec{b}$ *in unary normal form* iff

$$\forall b \in B : \vec{b}(b) = 0 \;\Leftrightarrow\; \vec{x}(x_b) = 0,$$

where we write x_b to refer to the clock that represents b.

For this kind of encoding, the TEST(b, v) widget, for a bit $b \in B$ and a value $v \in \{0, 1\}$, is a simple construction that is shown in Figure 7.1. We have an edge with guard $x_b = 0$ to test whether $b = 0$, and we have an edge with guard $x_b > 0$ to test whether $b = 1$. For all locations, we have the invariant $z = 0$ in order to ensure that executing TEST does not harm the unary normal form representation. We use a double border to indicate exits.

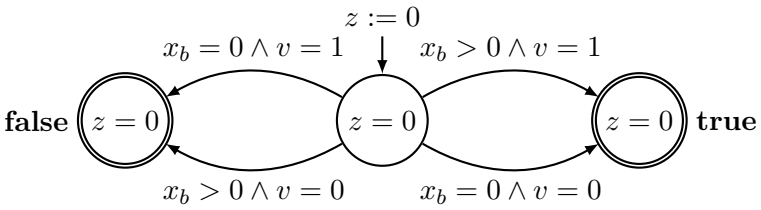


Figure 7.1: The TEST(b, v) widget for testing whether bit b has value v, assuming a unary encoding of the constants.

The SET(b, v) widget, for a bit $b \in B$ and a value $v \in \{0, 1\}$, is constructed as follows. The basic idea is to let exactly one time unit elapse at the beginning, and then, to reset all clocks whose value is exactly 1, except for x_b if $v = 1$. More technically, we construct the SET widget as a chain of conditional resets, which is shown in Figure 7.2. Clearly, on exiting the SET widget, we have the unary normal form representation again.

Observe that in both widgets, except for the entry location of the SET widget, we prevent time from elapsing by assigning the invariant $z = 0$ to every location. Also, observe that all widgets are deterministic.

7.1.2 Binary Encoding of Constants

In this subsection, we will demonstrate how to represent and manipulate an array of n bits only using three clocks, which can be compared to arbitrary

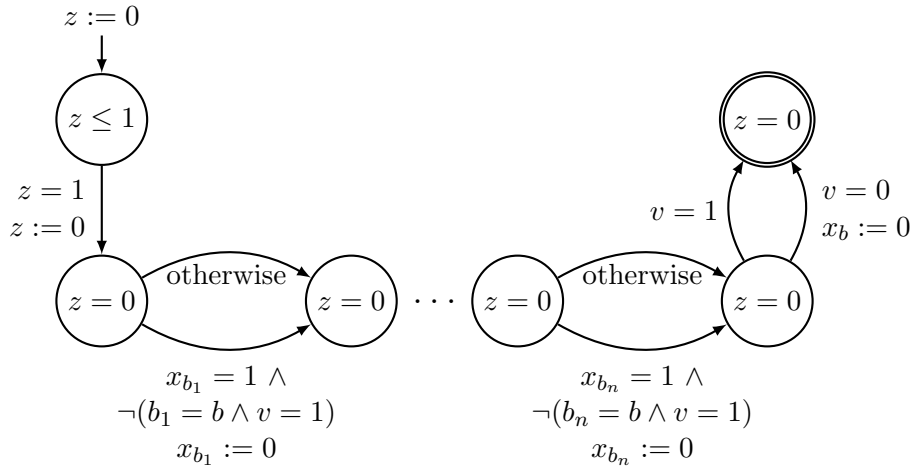


Figure 7.2: The $\mathsf{SET}(b, v)$ widget for setting bit b to v, assuming a unary encoding of the constants.

constants encoded in binary. The maximal constant that is used in the constraints of the TEST and SET widgets is 2^n.

For a set of bits $B = \{b_1, \ldots, b_n\}$, we introduce a set of clocks $X = \{x, y, z\}$, where the value of x represents the values of the bits in B, and y and z are auxiliary clocks. For a clock valuation $\vec{x} \in \mathbb{R}^X_{\geq 0}$ and a bit valuation $\vec{b} \in \vec{B}$, we say that $\vec{x}$ *represents* $\vec{b}$ *in binary normal form* iff

$$\vec{x}(x) \;=\; \sum_{i=1}^{n} \vec{b}(b_i) \cdot 2^{i-1}.$$

Before we come to the TEST and SET widgets, we first introduce three auxiliary widgets. First, we introduce a widget $\mathsf{ADD}(c)$ for adding a constant $c \in \{0, \ldots, 2^n\}$ to x. Here, the idea is to use z to let exactly 2^n time units elapse. At the same time, we reset x whenever x reaches $2^n - c$. By resetting y whenever y reaches 2^n, we make sure that y will have the same value as before. The construction of ADD is shown in Figure 7.3. We easily obtain a widget $\mathsf{SUB}(c)$ for subtracting a constant $c \in \{0, \ldots, 2^n\}$ from x by using $\mathsf{ADD}(2^n - c)$ instead.

A last auxiliary widget we need to introduce is the $\mathsf{COPY}(p, q)$ widget that copies the value of clock $q \in \{x, y\}$ to the value of clock $p \in \{x, y\} \setminus \{q\}$. Here, the construction is similar to the one of the ADD widget. But now, whenever q reaches 2^n, we reset q *and* p. The construction of COPY is shown in Figure 7.4.

With these auxiliary widgets, we can now give the construction of the TEST and SET widgets for binary encodings. In the construction of the

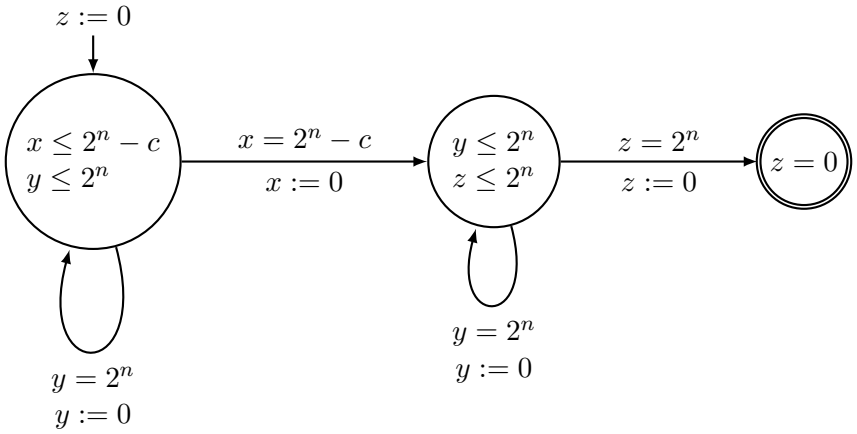


Figure 7.3: The $\mathsf{ADD}(c)$ widget for adding a nonnegative constant c to clock x.

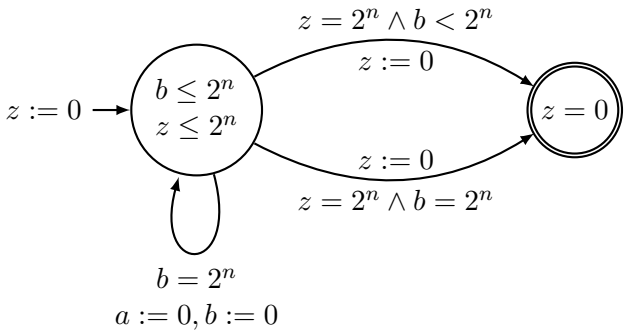


Figure 7.4: The $\mathsf{COPY}(a, b)$ widget that copies the current value of clock b to clock a.

$\mathsf{TEST}(b_i, v)$ widget, for the i^{th} bit b_i and a value $v \in \{0, 1\}$, we first backup the current values of all bits, using the auxiliary clock y. Then, for accessing the i^{th} bit, we reset all bits above i in x, so that we can use the simple clock constraints $x > 2^i$ and $x \leq 2^i$ to check whether the i^{th} bit is set or not, respectively. Finally, we have to restore the original value from y in x again. The construction is shown in Figure 7.5.

Now, the construction of the $\mathsf{SET}(b_i, v)$ widget, for the i^{th} bit b_i and a value $v \in \{0, 1\}$, is straightforward. If $v = 0$, we check whether the i^{th} bit is set and, if this is the case, we reset the bit by calling $\mathsf{SUB}(2^i)$. If $v = 1$, we check whether the i^{th} bit is not set and, if this is the case, we set the bit by calling $\mathsf{ADD}(2^i)$. The construction is shown in Figure 7.6.

Observe that during an execution of an auxiliary widget ADD, SUB, or

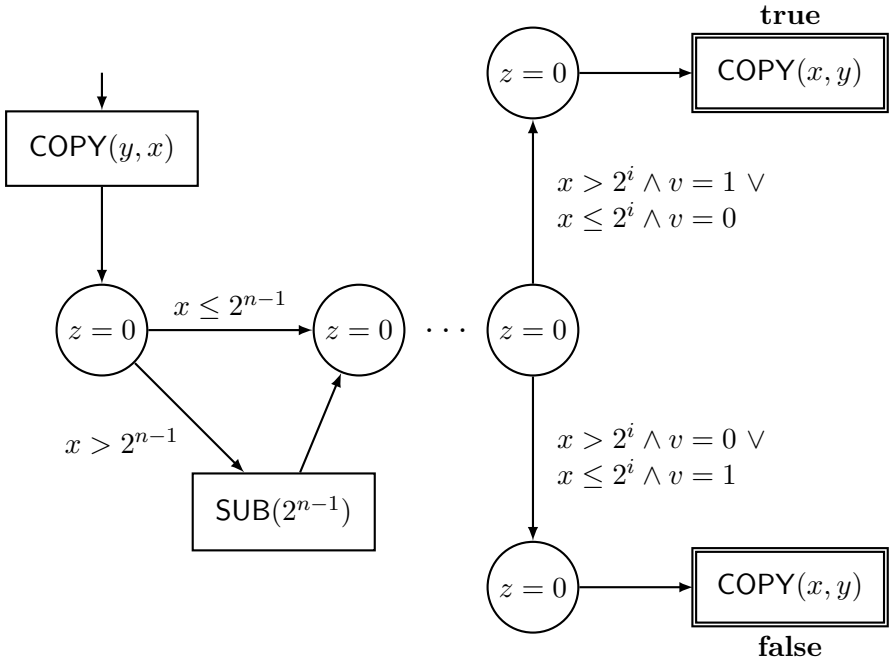


Figure 7.5: The TEST(b_i, v) widget for testing whether the i^{th} bit is v, assuming a binary encoding of the constants.

COPY, exactly 2^n time units elapse. When executing SET or TEST, no time elapses. Thus, it can be easily seen that the binary normal form is always maintained outside an execution of SET or TEST. Also, observe that all widgets are deterministic.

7.2 Timed Automata and Sequential Circuit Machines

In this section, we show the equivalence of timed automata with sequential circuit machines based on the constructions from the previous section.

For an input $\mathcal{P} = (P, \Sigma^{\mathsf{obs}}_{\mathsf{out}}, q_b, \mu_c, q^{\mathsf{max}})$ to Timed synthesis, where $\mathcal{T} = (Q, q_0, \Sigma, E, I)$ is a μ-granular controllable timed automaton with $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$, we define the instance signature

$$sig(\textsc{Timed synthesis}, \mathcal{P}) = \log\Big(\big(|\Sigma_{\mathsf{in}}| + 1,\ |\Sigma^{\mathsf{obs}}_{\mathsf{out}}| + 2,\ |\Sigma_{\mathsf{out}}| + 2\big),\ \big(q^{\mathsf{max}} \cdot |[\mu_c]|,\ |[\mathcal{T}]_{\mu'}|\big)\Big),$$

where $\mu' = \mu \otimes \mu_c$.

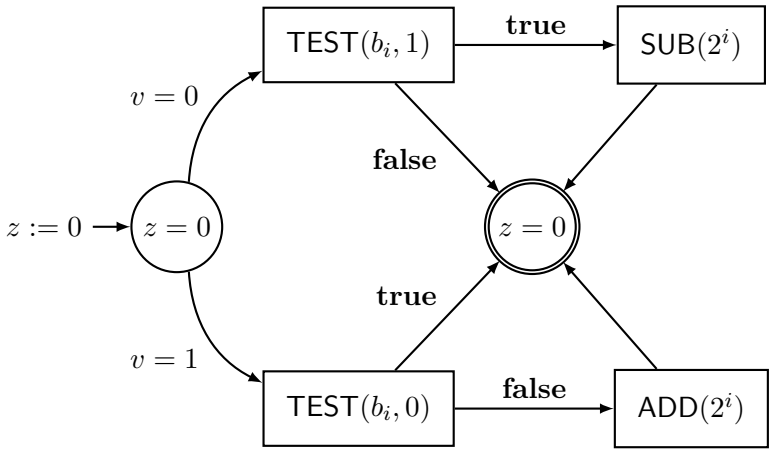


Figure 7.6: The $\mathsf{SET}(b_i, v)$ widget for setting the i^{th} bit to v, assuming a binary encoding of the constants.

We first show that one can use sequential circuit machines to decide Timed synthesis.

Lemma 7.2.1. *For an input y, $\mathcal{G}(\text{Timed synthesis}, y)$ allows a succinct circuit representation of signature $sig(\text{Timed synthesis}, y)$.*

Proof. Let $y = (P, \Sigma^{\mathsf{obs}}_{\mathsf{out}}, q_b, \mu_c, q^{\mathsf{max}})$ with $P = (Q, q_0, \Sigma, E, I)$, we show that the definition of $\mathcal{G}(\text{Timed synthesis}, y)$ matches the properties required for succinct circuit representations from Section 3.4.
(1) There is a logarithmic encoding of the controllable decisions $\Sigma_{\mathsf{in}} \cup \{\tau\}$ since Σ_{in} is explicitly given. We introduce the bits $[\![\widehat{\Sigma^{\mathsf{obs}}_{\mathsf{out}} \cup \{\tau, \tau^o_t\}}]\!]$ for representing the observable uncontrollable events, where τ^o_t represents a visible elapsing of time (i.e., advancing to the successor region). We also introduce the bits $[\![\widehat{\Sigma^{\mathsf{unobs}}_{\mathsf{out}} \cup \{\tau^u_t\}}]\!]$ for representing the unobservable uncontrollable events, where τ^u_t represents an invisible elapsing of time. The idea is that whenever Player A proposes a delay step, he must play a move τ^o_t if this step is observable through the controller's granularity, or a move τ^u_t if it is unobservable. Then, the bits for the logarithmic encoding of all uncontrollable decisions are $[\![\widehat{\Sigma^{\mathsf{obs}}_{\mathsf{out}} \cup \{\tau, \tau^o_t\}}]\!] \uplus [\![\widehat{\Sigma^{\mathsf{unobs}}_{\mathsf{out}} \cup \{\tau^u_t\}}]\!]$.

The logarithmic encoding of the positions (i.e., the regions of P) looks as follows. Notice that we can identify a region from a granularity $\mu = (X, m, c^{\mathsf{max}})$ as a tuple containing (i) the location P is currently in, (ii) an $|X|$-dimensional vector over tuples of the form $(v, \prec, p)$, where $v \in \{0, \ldots, k\}^{|X|}$ and $k = m \cdot c^{\mathsf{max}}$ (here called the *clock's value*), $\prec \in \{=, <\}$ (here called the *clock's sign*), and $p \in \{1, \ldots, |X|\}$ (here called the *clock's*

position). Hence, we can represent a region using an array of

$$\lceil \log |Q| \rceil \ + \ |X| \cdot (\lceil \log(k+1) \rceil + 1 + \lceil \log |X| \rceil)$$

(i.e., polynomially many) bits. One extra bit is additionally needed to encode the information, which player can move.

(2) Concerning the move function Δ, we construct Δ^c using the following sub-circuits. We implement the discrete steps of P by constructing a DNF of polynomial size from E, similar to the construction in the proof of Lemma 5.1.4. The guards can be implemented by independent comparator circuits whose output bits can be used in the DNF. Executing resets can be done by the following simple sub-circuit:

$$\mathit{reset}(x, (v, \prec, p), r) = \begin{cases} (0, =, p') & \text{if } x \in r; \\ (v, \prec, p + |r|) & \text{if } x \notin r, \end{cases}$$

where p' is the position of x in the ordered set r. Notice that we statically create such a sub-circuit *reset* for each clock x and for each distinct reset appearing in E. Each *reset* sub-circuit has a constant depth. We embed all *reset* sub-circuits in Δ^c in the following way: To update each clock x, we use a multiplexer that selects for a given decision the corresponding *reset* sub-circuit whose output is used to update the game position for x.

For executing the delay step, we introduce the following simple sub-circuits:

$$\mathit{advance}(x) :\Leftrightarrow \mathit{sign}(0) = \text{“}<\text{”} \ \wedge \bigwedge_{p \geq pos(x)} (\mathit{sign}(p) = \text{“}=\text{”})$$

where $pos(x)$ and $sign(x)$ are simple sub-circuits of constant depth that compute the current position and sign of x, respectively. Now, we update the entries of each clock using the following simple sub-circuit:

$$\mathit{delay}(x, (v, \prec, p)) = \begin{cases} (v, <, p) & \text{if } p = 0 \text{ and } \prec = \text{“}=\text{”}; \\ (v+1, =, p + a - |X|) & \text{if } p \geq |X| - a; \\ (v, \prec, p + a) & \text{otherwise,} \end{cases}$$

where $a = |\{x \mid \mathit{advance}(x) = \mathbf{true}\}$ is the number of clocks that advance to their next value. Overall, it can be easily seen that the size of Δ^c is polynomial in $|P|$ and its depth is at most logarithmic in $|P|$.

(3) We can construct B^c as a sub-circuit of polynomial size and constant depth: The discrete part of B can be easily checked by a DNF over the location bits. For the timed part of B, observe that for each location of P, the time lock region can be characterized by a conjunction of clock constraints.

Hence, one can construct a simple circuit of constant depth that checks time locks.

(4) We represent $\mathcal{V}$ as the subset $(\!|\Sigma_{\mathsf{out}}^{\mathsf{obs}} \cup \widehat{\{\tau, \tau_t^o\}}|\!) \subseteq (\!|\Sigma_{\mathsf{out}} \cup \widehat{\{\tau, \tau_t^o\}}|\!)$. □

Combining Lemma 7.2.1 and Lemma 3.4.1, we immediately obtain that every given instance of TIMED SYNTHESIS can be transformed into an instance of DIVERGING for sequential circuit machines with the same signature.

Lemma 7.2.2. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((\mathcal{L}, \mathcal{L}_\infty, \mathcal{L}), (\mathcal{P}_\infty, \mathcal{P}))$,* TIMED SYNTHESIS$_{\widehat{\sigma}}$ *is* LOGSPACE*-reducible to* DIVERGING$_{\widehat{\sigma}}$.

With the following lemma, we establish the opposite direction:

Lemma 7.2.3. *For a succinctness signature $\widehat{\sigma} \in \mathcal{C}((\mathcal{L}, \mathcal{L}_\infty, \mathcal{L}), (\mathcal{P}_\infty, \mathcal{P}))$,* DIVERGING$_{\widehat{\sigma}}$ *is* LOGSPACE*-reducible to* TIMED SYNTHESIS$_{\widehat{\sigma}}$.

Proof. For a given machine $\mathcal{S} = (((n_{\mathsf{E}}, n_{\mathsf{A}}^{\mathsf{E}}, n_{\mathsf{A}}), (m_{\mathsf{E}}, m_{\mathsf{A}})), C_{\mathsf{A}})$, we construct an input $(P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, q^{\mathsf{max}})$ to TIMED SYNTHESIS in the following way. Without loss of generality, we assume C_{A} has no shared sub-circuits and, thus, is represented as a Boolean expression, where $C_{\mathsf{A}}(\mathsf{MA}_i)$ is the Boolean function that computes the new value for MA_i, $1 \leq i \leq m_{\mathsf{A}}$, and $C_{\mathsf{A}}(\mathsf{h})$ computes the value of the halting flag.

We construct the plant $P = (Q, q_0, \Sigma, E, I)$ in such a way that it simulates the sequential execution of $\mathcal{S}$. Depending on the type of encoding of the constants (either unary or binary, see Section 7.1), we represent the n bits of the circuit elements N_{A}, N_{E}, M_{A}, h, and the updated universal memory $M_{\mathsf{A}}{}'$, either as (1) n clocks or (2) clock values ranging from 0 to $2^n - 1$, respectively. For a circuit element $x \in N_{\mathsf{A}} \cup N_{\mathsf{E}} \cup M_{\mathsf{A}} \cup M_{\mathsf{A}}{}' \cup \{\mathsf{h}\}$ and a $v \in \{0, 1\}$, we assume we have the widgets $\mathsf{SET}(x, v)$ and $\mathsf{TEST}(x, v)$ at our disposal.

We define the set of observable uncontrollable events as $\Sigma_{\mathsf{out}}^{\mathsf{obs}} = \vec{N}_{\mathsf{A}}^{\mathsf{E}}$, the set of unobservable uncontrollable events as $\Sigma_{\mathsf{out}}^{\mathsf{unobs}} = \{\epsilon\}$, and the set of controllable events as $\Sigma_{\mathsf{in}} = \vec{N}_{\mathsf{E}}$. Now, the simulation of a cycle looks as follows. First, for determining values for the unobservable universal guessing bits, we let P nondeterministically branch with an ϵ-edge to a particular chain of widgets of the form

$$\mathsf{SET}(\mathsf{NA}_1, v_1) \to \ldots \to \mathsf{SET}(\mathsf{NA}_{n_{\mathsf{A}} - n_{\mathsf{A}}^{\mathsf{E}}}, v_{n_{\mathsf{A}} - n_{\mathsf{A}}^{\mathsf{E}}}),$$

where $(v_1, \ldots, v_{n_{\mathsf{A}} - n_{\mathsf{A}}^{\mathsf{E}}})$ represents a vector from $N_{\mathsf{A}} \setminus \vec{N}_{\mathsf{A}}^{\mathsf{E}}$. Here, we consider $2^{n_{\mathsf{A}} - n_{\mathsf{A}}^{\mathsf{E}}}$ branching decisions to cover all possible choices. Then, for determining values for the observable universal guessing bits, P nondeterministically

branches with an edge labeled with a $\vec{n_a^e} \in \vec{N_{\mathsf{A}}^{\mathsf{E}}}$ to a particular chain of widgets of the form

$$\mathsf{SET}(\mathsf{NO}_1, \vec{n_a^e}[1]) \to \ldots \to \mathsf{SET}(\mathsf{NO}_{n_{\mathsf{A}}^{\mathsf{E}}}, \vec{n_a^e}[n_{\mathsf{A}}^{\mathsf{E}}]).$$

Here, we consider $2^{n_{\mathsf{A}}^{\mathsf{E}}}$ branching decisions to cover all possible choices. After the universal guessing bits are determined, P awaits the values for the circuit elements N_{E} from the controller. For this purpose, we let P receive a controllable event $\vec{n_e} \in \Sigma_{\mathsf{in}}$ and then branch to a chain of widgets of the form

$$\mathsf{SET}(\mathsf{NE}_1, \vec{n_e}[1]) \to \ldots \to \mathsf{SET}(\mathsf{NE}_{n_{\mathsf{E}}}, \vec{n_e}[n_{\mathsf{E}}]).$$

Since we have only a logarithmic number of guessing bits, the exponential blowup due the nondeterministic branching results only in a polynomial control structure.

The actual execution of C_{A} is simulated by P as follows. For each Boolean formula $C_{\mathsf{A}}(y)$, $y \in M_{\mathsf{A}} \cup \{\mathsf{h}\}$, we embed its formula DAG $D = \mathit{dag}(C_{\mathsf{A}}(y))$ (recall Section 2.2) into the control structure of P: For each edge in D, we execute a widget for testing the corresponding bit: For some circuit element $z \in N_{\mathsf{A}} \cup N_{\mathsf{E}} \cup M_{\mathsf{A}}$, if the original edge in D is labeled with the literal z then the widget $\mathsf{TEST}(z, 1)$ is executed, and if the original edge in D is labeled with the literal $\neg z$ then the widget $\mathsf{TEST}(z, 0)$ is executed. If the execution of D in P ends up in $\mathit{true}(D)$ the widget $\mathsf{SET}(z', 0)$ is executed, if the execution ends up in $\mathit{false}(D)$ the widget $\mathsf{SET}(z', 1)$ is executed, where z' is either the corresponding updated memory element if $z \in M_{\mathsf{A}}$, or h otherwise. We execute these testing and setting widgets in a sequential manner. Note that the only nondeterminism in P occurs for resolving the branching decisions for the guessing bits. Note that the only existential nondeterminism in P occurs for resolving the branching decisions for the existential guessing bits. The universal nondeterminism is used for resolving the branching decisions for the universal guessing bits and for resolving the nondeterminism in the DAGs of the Boolean formulas. Before the execution of a cycle finishes, we copy the contents of the updated memory $M_{\mathsf{A}}{}'$ to M_{A} by the following chain of widgets:

$$\begin{aligned}&\mathsf{TEST}(\mathsf{MA}'_1, v_1) \to \mathsf{SET}(\mathsf{MA}_1, v_1) \to \ldots \to \\ &\quad \mathsf{TEST}(\mathsf{MA}'_{m_{\mathsf{A}}}, v_{m_{\mathsf{A}}}) \to \mathsf{SET}(\mathsf{MA}_{m_{\mathsf{A}}}, v_{m_{\mathsf{A}}})\end{aligned}$$

We finish our construction by defining the bad location q_b to be $\mathit{false}(\mathit{dag}(C_{\mathsf{A}}(\mathsf{h})))$, the controller granularity μ_c as $(\emptyset, 1, 0)$, and the bound on the locations of the controller q^{max} as m_{E}. Note that we require the controller to have no clocks at all. This way, we make sure that the controller only uses locations to represent its memory. □

The combination of Lemma 7.2.2 and Lemma 7.2.3 gives the desired connection between timed automata and sequential circuit machines:

Theorem 7.2.4. *For a succinctness signature* $\widehat{\sigma} \in \mathcal{C}((\mathcal{L}, \mathcal{L}_\infty, \mathcal{L}), (\mathcal{P}_\infty, \mathcal{P}))$, TIMED SYNTHESIS$_{\widehat{\sigma}}$ *and* DIVERGING$_{\widehat{\sigma}}$ *are* LOGSPACE*-reducible to each other.*

7.3 Complexity Analysis

Based on the connection to sequential circuit machines established in the previous section, we are now ready to obtain complexity bounds.

7.3.1 Model Checking

When we assume a one-player setting (by either choosing $\Sigma_{\mathsf{in}} = \emptyset$ or $\Sigma_{\mathsf{out}} = \emptyset$), controller synthesis boils down to the safety model checking problem. The combination of Theorem 7.2.4, Lemma 4.1.1, and Theorem 4.2.5 yields the complexity for safety model checking of timed automata:

Corollary 7.3.1. *The* EG*-model checking problem for timed automata is complete for* $\mathcal{C}((\mathcal{L}, 0, 0), (\infty, \mathcal{P})) = \text{PSPACE}$*. The* AG*-model checking problem for timed automata is complete for* $\mathcal{C}((0, 0, \mathcal{L}), (\infty, \mathcal{P})) = \text{PSPACE}$*.*

The combination of Theorem 7.2.4, Lemma 4.1.1, and Theorem 4.2.9 reveals the complexities of finding small witnesses for timed automata:

Corollary 7.3.2. *The* EG*-small witness problem for timed automata is complete for*

- $\mathcal{C}((\mathcal{L}, 0, 0), (\mathcal{L}, \mathcal{P})) = \text{PSPACE}$ *if the bound on the length of the witness is encoded in unary, and*
- $\mathcal{C}((\mathcal{L}, 0, 0), (\mathcal{P}, \mathcal{P})) = \text{NEXPTIME}$ *if the bound on the length of the witness is encoded in binary, respectively.*

7.3.2 Control with Full Observability

Assuming two players and full observability, by combining Lemma 4.1.1, Theorem 4.2.5, and Theorem 7.2.4, we obtain the complexity for unbounded synthesis for timed automata under full observability:

Corollary 7.3.3. *The unbounded safety controller synthesis problem for controllable timed automata under full observability is complete for* $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\infty, \mathcal{P})) = \text{EXPTIME}$*.*

The combination of Theorem 7.2.4, Lemma 4.1.1, and Theorem 4.2.9 reveals the complexities of bounded synthesis:

Corollary 7.3.4. *The bounded safety controller synthesis problem for controllable timed automata with full observability is complete for*

- $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\mathcal{L}, \mathcal{P})) = \text{PSPACE}$ *if the bound on the number of locations of the controller is encoded in unary, and*
- $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NEXPTIME}$ *if the bound on the number of locations of the controller is encoded in binary, respectively.*

7.3.3 Control with Partial Observability

Assuming partial observability without imposing a bound on neither the existential memory nor the observability results in undecidability.

Theorem 7.3.5. *[Bouyer et al., 2003] For a given timed plant* $P = (Q, q_0, \Sigma, E, I)$ *with* $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$*, a set of observable events* $\Sigma_{\mathsf{out}}^{\mathsf{obs}} \subseteq \Sigma_{\mathsf{out}}$*, and a dedicated bad location* $q_b \in Q$*, deciding whether there is a granularity* μ_c *such that*

$$\text{TIMED SYNTHESIS}(P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, \infty) = \textbf{yes}$$

is undecidable.

Only bounding the number of clocks of the controller, while leaving the number of locations unrestricted, does not suffice to obtain decidability, since the existential memory and the observability are still unbounded.

Theorem 7.3.6. *For a given timed plant* $P = (Q, q_0, \Sigma, E, I)$ *with* $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$*, a set of observable events* $\Sigma_{\mathsf{out}}^{\mathsf{obs}} \subseteq \Sigma_{\mathsf{out}}$*, a dedicated bad location* $q_b \in Q$*, and a bound on the number of clocks of the controller* $k \in \mathbb{N}_{\geq 1}$*, deciding whether there is a granularity* $\mu_c = (X_c, m_c, c_c^{\mathsf{max}})$ *with* $|X_c| \leq k$ *such that*

$$\text{TIMED SYNTHESIS}(P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, \infty) = \textbf{yes}$$

is undecidable.

Proof. We show undecidability by a reduction from the halting problem of a given two-counter Minsky machine, which is known to be undecidable [Minsky, 1967]. The basic idea is to let the synthesis algorithm generate a controller that simulates an accepting run of the machine, or to report that no such controller/run exists. Following the construction proposed by Bouyer and Chevalier [2006], which, in turn, is an extension of the one proposed by Alur and Dill [1994], we let the plant nondeterministically and unobservably for the controller verify that it faithfully performs the simulation. We refer to Bouyer and Chevalier [2006] for details on the modeling of the verification widgets. Note that the goal state of the machine is reached after finitely many steps $m \in \mathbb{N}_{\geq 1}$ for a configuration with some maximal counter value

bounded by m. Hence, the synthesis algorithm must somehow determine m (or report that no such m exists) and fix a sufficiently large number of locations and fine granularity $(\{z\}, 4m, 1)$ to accommodate the necessary information to keep track of the machine's configurations arising during its execution. We let the controller and the plant communicate via the actions a, b, c, and d. Without loss of generality, we assume that the states of the given machine have a unique index and their number is less than m.

As usual for such proofs, a configuration is encoded as a sequence of actions $d^s a^{c_1} b^{c_2} c^{c_3}$ representing the current state of the machine with index s, the current values of the two machine counters c_1 and c_2, and the value of a step counter c_3. Here, c_3 is necessary to force the controller to reach the goal state within a finite amount of steps. After the i^{th} step of the machine, we let the controller produce the current configuration sequence within the time interval $[i, i+1)$. Here, we force the controller that the delay between corresponding actions in two successive configurations is exactly one time unit. This can be achieved by a plant component that nondeterministically chooses an action, waits exactly one time unit, and then verifies whether the controller immediately produces that action. Hereby, we exploit the partial observability to completely hide these verification steps. In the first configuration, we let the controller choose an appropriate value for $c_3 > 0$. After each step, we require that the controller decrements c_3 by one. If c_3 becomes 0, we let the plant go into a bad state.

Now, assuming that the goal state of the given machine is reachable, let us fix some m. It is easy to see that the number of distinct configurations of the machine along with a certain step count is bounded by m^4. Hence, a feasible controller could have $O(m^8)$ locations, i.e., $O(m^2)$ locations to represent and produce the current configuration of the machine as well as the step count. If the machine is in a certain configuration after a certain number of steps, the corresponding part of the controller's control structure is of the following form:

$$l_d \underbrace{\xrightarrow{d, z=\frac{1}{4m}, z:=0} \cdots \xrightarrow{d, z=\frac{1}{4m}, z:=0}}_{s \text{ times}} l'_d \underbrace{\xrightarrow{\epsilon, z=\frac{1}{4m}, z:=0} \cdots \xrightarrow{\epsilon, z=\frac{1}{4m}, z:=0}}_{m-s \text{ times}} l''_d,$$

for representing the current machine state s;

$$l_a \underbrace{\xrightarrow{a, z=\frac{1}{4m}, z:=0} \cdots \xrightarrow{a, z=\frac{1}{4m}, z:=0}}_{c_1 \text{ times}} l'_a \underbrace{\xrightarrow{\epsilon, z=\frac{1}{4m}, z:=0} \cdots \xrightarrow{\epsilon, z=\frac{1}{4m}, z:=0}}_{m-c_1 \text{ times}} l''_a,$$

for representing the current value of the machine counter c_1;

$$l_b \underbrace{\xrightarrow{b, z=\frac{1}{4m}, z:=0} \cdots \xrightarrow{b, z=\frac{1}{4m}, z:=0}}_{c_2 \text{ times}} l'_b \underbrace{\xrightarrow{\epsilon, z=\frac{1}{4m}, z:=0} \cdots \xrightarrow{\epsilon, z=\frac{1}{4m}, z:=0}}_{m-c_2 \text{ times}} l''_b,$$

for representing the current value of the machine counter c_2;

$$l_c \underbrace{\xrightarrow{c,z=\frac{1}{4m},z:=0} \dots \xrightarrow{c,z=\frac{1}{4m},z:=0}}_{c_3 \text{ times}} l'_c \underbrace{\xrightarrow{\epsilon,z=\frac{1}{4m},z:=0} \dots \xrightarrow{\epsilon,z=\frac{1}{4m},z:=0}}_{m-c_3 \text{ times}} l''_c,$$

for representing the current step count c_3.

Thus, there is a controller iff there is an accepting run, and furthermore, if there is a controller at all then there is one that can be represented using a single clock. □

On the other hand, only bounding the number of locations of the controller, while leaving the number of clocks unrestricted, does not suffice to obtain decidability either.

Theorem 7.3.7. *For a given timed plant $P = (Q, q_0, \Sigma, E, I)$ with $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$, a set of observable events $\Sigma_{\mathsf{out}}^{\mathsf{obs}} \subseteq \Sigma_{\mathsf{out}}$, a dedicated bad location $q_b \in Q$, and a bound on the number of locations of the controller $q^{\mathsf{max}} \in \mathbb{N}_{\geq 1}$, deciding whether there is a granularity μ_c such that*

$$\textsc{Timed synthesis}(P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, q^{\mathsf{max}}) = \mathbf{yes}$$

is undecidable.

Proof. To show undecidability, we give a similar reduction from the halting problem of a given two-counter Minsky machine as the one used in the proof of Theorem 7.3.6. But now, the synthesized controller generating an accepting run (if there is one) has only one location and uses its clocks to represent all information necessary to keep track of the machine's current configuration. Such a one-location controller is a translation of the one-clock controller from above, where the location-based control structure is simulated by a pure clock-based control structure. In the following, we explain this translation.

Without loss of generality, we assume that the one-clock controller contains 2^b locations, for some $b \in \mathbb{N}$. We introduce b clocks in the one-location controller $x_1, \dots, x_b$. Also, we assume that each location has a unique index between 0 and $2^b - 1$. The one-clock controller is in location with index l iff $\vec{x} = l$, where

$$\vec{x} = l \quad :\Longleftrightarrow \quad \bigwedge_{i=1}^{b} x_i \leq \frac{1}{4m} \Leftrightarrow l_i = 0$$

and l_i refers to the i^{th} bit of l. Since each step in the one-clock controller takes exactly $\frac{1}{4m}$ time units, we also have an auxiliary clock z in the one-location controller that is reset on every discrete step. For each edge between two locations l and l' in the one-clock controller, we introduce a

corresponding (self-looping) edge in the one-location controller with guard $z = \frac{1}{4m} \wedge \vec{x} = l$ that resets all clocks in $\vec{x}$ whose corresponding bit in l' is zero.

Thus, if there is a controller at all then there is one that can be represented using a single location. □

However, when bounding the granularity of the controller one obtains decidability, since the existential memory (which is still unbounded) only needs to remember a finite amount of observations. By combining Theorem 7.2.4, Lemma 4.1.1, and Theorem 4.2.6, we obtain the complexities for controller synthesis under no and partial observability:

Corollary 7.3.8. *The unbounded safety controller synthesis problem for controllable timed automata is complete for*

- $\mathcal{C}((\mathcal{L}, 0, \mathcal{L}), (\infty, \mathcal{P})) = \text{ExpSpace}$*, assuming no observability, and*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\infty, \mathcal{P})) = 2\text{ExpTime}$*, assuming partial observability, respectively.*

We note that the second claim was already proven by Bouyer et al. [2003]. The first claim, to the best of the author's knowledge, is a new result that is firstly proven in this thesis.

If we impose a bound on the number of locations of the controller, the combination of Theorem 7.2.4, Lemma 4.1.1, and Theorem 4.2.9 reveals the complexity of bounded synthesis for timed automata under no and partial observability:

Corollary 7.3.9. *The bounded safety controller synthesis problem for controllable timed automata with no or partial observability, where the bound on the number of locations of the controller is either given in unary or binary, is complete for*

- $\mathcal{C}((\mathcal{L}, 0, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$*, assuming no observability, and*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$*, assuming partial observability, respectively.*

7.3.4 Discrete Controllers

In the previous subsections, we have seen that even under the assumption of full observability, timed controller synthesis is an inherently intractable problem.[2] In this subsection, we identify *discrete controllers* that communicate synchronously with arbitrary timed plants as a subclass of the synthesis problem which exhibits a better worst-case complexity. In contrast

[2] In the sense that, for some instances, there is always an exponential blow up in the running time, as it is a well-known fact that PTime ≠ ExpTime [Hartmanis and Stearns, 1965].

to general timed controllers, discrete controllers may only react to discrete observations of the plant; they are not allowed to measure the time between two observed events.

More formally, for a safety controller synthesis problem for controllable timed automata i, where $i = (P, \Sigma_{\mathsf{out}}^{\mathsf{obs}}, q_b, \mu_c, q^{\mathsf{max}})$, where $P = (Q, q_0, \Sigma, E, I)$ with $\Sigma = \Sigma_{\mathsf{in}} \uplus \Sigma_{\mathsf{out}}$ and $P_\mu = (X, m, c^{\mathsf{max}})$, we call a controller timed automaton $C = (Q_c, q_0^c, \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}^{\mathsf{obs}}, E_c)$, representing a solution to i, *discrete* if the following conditions are satisfied:

(1) $\mu_c = (X_c, 1, 0)$ and $|X_c| = 1$;

(2) for each $\pi \in \mathsf{Traces}([\![P \| C]\!])$ and each $1 \leq i < |\pi|$, whenever $\pi[i+1] \in \Sigma_{\mathsf{in}}$ we require that $\pi[i] \in \Sigma_{\mathsf{out}}^{\mathsf{obs}}$.

We point out that discrete controllers differ from controllers with a fixed sampling rate considered by Henzinger and Kopke [1999] or Cassez et al. [2002].

Example 7.3.10. *Figure 7.7 shows a controllable timed automaton with output action a and input actions b and c modeling a plant, and a discrete controller that ensures that the bad location (drawn with a double border) is never reached.*

Obviously, the only meaningful bound which one can impose on discrete controllers is to restrict the number of locations. In the following, we investigate the complexity of the unbounded and the bounded case.

It turns out that the restriction to discrete controllers does not pay off in the case with an unbounded number of locations. The combination of Theorem 7.2.4, Lemma 4.1.1, and Theorems 4.2.5 and 4.2.6 reveals the following complexity results:

Corollary 7.3.11. *The unbounded safety controller synthesis problem for discrete controllable timed automata is complete for*

- $\mathcal{C}((\mathcal{L}, 0, \mathcal{L}), (\infty, \mathcal{P})) = \textsc{ExpSpace}$*, assuming no observability,*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\infty, \mathcal{P})) = 2\textsc{ExpTime}$*, assuming partial observability, and*
- $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\infty, \mathcal{P})) = \textsc{ExpTime}$*, assuming full observability, respectively.*

We point out that, for establishing the lower bounds, a direct adaptation of the classical ExpTime- and 2ExpTime-hardness proofs given by Henzinger and Kopke [1999] and D'Souza and Madhusudan [2002], respectively, is not possible, as these proofs rely on the assumption of a timed controller.

By imposing a bound on the number of locations of a discrete controller, we can combine Theorem 7.2.4, Lemma 4.1.1, and Theorem 4.2.9 to obtain the following complexity results:

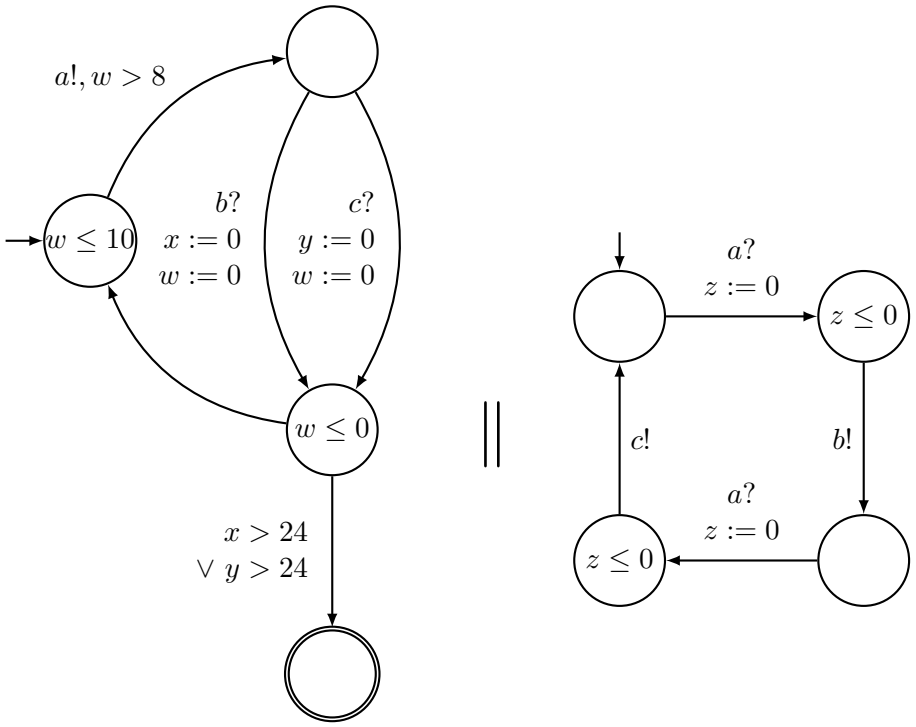


Figure 7.7: Plant timed automaton (on the left) controlled by a discrete controller (on the right) that ensures that the bad location (drawn with a double border) is never reached.

Corollary 7.3.12. *The bounded safety controller synthesis problem for discrete controllable timed automata, where the bound on the number of locations of the controller is given in binary, is complete for*

- $\mathcal{C}((\mathcal{L}, 0, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$*, assuming no observability,*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$*, assuming partial observability, and*
- $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\mathcal{P}, \mathcal{P})) = \text{NExpTime}$*, assuming full observability, respectively.*

By assuming a unary encoding of the bound on the number of locations, the controller can only access a logarithmic number of bits in its memory. We thus obtain:

Corollary 7.3.13. *The bounded safety controller synthesis problem for discrete controllable timed automata, where the bound on the number of locations of the controller is given in unary, is complete for*

- $\mathcal{C}((\mathcal{L}, 0, \mathcal{L}), (\mathcal{L}, \mathcal{P})) = \text{PSpace}$*, assuming no observability,*
- $\mathcal{C}((\mathcal{L}, \mathcal{L}, \mathcal{L}), (\mathcal{L}, \mathcal{P})) = \text{PSpace}$*, assuming partial observability, and*

- $\mathcal{C}((\mathcal{L}, \infty, \mathcal{L}), (\mathcal{L}, \mathcal{P})) = \text{PSPACE}$, *assuming full observability, respectively.*

In conclusion, whenever one restricts the controller to use only a logarithmic amount of memory, synthesis becomes as easy as model checking. As we will see in the next chapters, we can indeed exploit the drastic decrease in complexity from 2EXPTIME to PSPACE and devise an effective synthesis algorithm based on model checking that can be efficiently implemented using symbolic data structures.

7.4 Bibliographic Remarks

To the best of the author's knowledge, this thesis presents the first complexity-theoretic analysis of bounded synthesis for timed automata, providing matching lower and upper complexity bounds for various relevant problems. Moreover, this thesis also presents the first EXPSPACE-completeness proof of timed controller synthesis under no observability and the first NEXPTIME-completeness proof of the small witness problem for timed automata. Tables 7.1 and 7.2 provide an overview.

Branching \ Witness	Unconstrained	Binary bounded	Unary bounded
Nondeterministic	PSPACE-c	**NExpTime-c**	**PSpace-c**
Alternating	EXPTIME-c	**NExpTime-c**	**PSpace-c**
Blindfold alternating	**ExpSpace-c**	**NExpTime-c**	**PSpace-c**
Private alternating	UNDECIDABLE	**NExpTime-c**	**PSpace-c**

Table 7.1: Overview on the complexities of the reachability problem for timed automata. The results written in **bold face** are established in this thesis, the PSPACE-completeness result is due to Alur et al. [1990], the EXPTIME-completeness result is due to Henzinger and Kopke [1999], and the undecidability result is due to Bouyer et al. [2003].

Since our analysis uniformly bases on a connection to sequential circuit machines, every result that we establish is precise[3] with respect to the number of clocks and the encoding of the constants.

Timed automata were introduced in a sequence of papers in the early 1990s [Alur et al., 1990, Alur and Dill, 1990, Alur et al., 1993a, Alur and Dill, 1994]. PSPACE-completeness of the model checking problem was established by Alur et al. [1990]. Courcoubetis and Yannakakis [1992] showed that the PSPACE-hardness of the reachability problem for timed automata already

[3] there are still many open problems for timed automata with two clocks and constants encoded in binary

Timing \ Locations	Unconstrained	Binary bounded	Unary bounded
Unconstrained	UNDECIDABLE	**Undecidable**	**Undecidable**
Bounded	2EXPTIME-C	**NExpTime-c**	**NExpTime-c**
None (discrete)	**2ExpTime-c**	**NExpTime-c**	**PSpace-c**

Table 7.2: Overview on the complexities of the bounded synthesis problem for timed controllers with partial observability. The results written in **bold face** are established in this thesis, the two other results are due to Bouyer et al. [2003].

occurs for either a polynomial number of clocks with integer constants 0 and 1, or for timed automata with precisely three clocks and integer constants encoded in binary. The complexity of model checking timed automata with one or two clocks was investigated by Laroussinie et al. [2004]. Tripakis [2003] and Finkel [2005] proved the undecidability of various problems concerning the minimization and determinization of timed automata.

The extension to controller synthesis, i.e., the turn-based two-player case, is due to Maler et al. [1995] and Asarin et al. [1998], who introduced turn-based timed game automata. The decidability of the safety controller synthesis problem was shown by demonstrating that the attractor construction [see, e.g., Grädel et al., 2002] can be carried out using *clock zones* (polyhedra that symbolically represent sets of clock values). Henzinger and Kopke [1999] showed that the discrete attractor construction on the region graph of the plant is theoretically optimal by proving that the problem is EXPTIME-complete, assuming a polynomial number of clocks. Later, Chen and Lu [2008] made the lower bound more precise and proved that the exponential blow-up already occurs assuming only three clocks and a binary encoding of the constants.

D'Souza and Madhusudan [2002] investigated the complexity of timed controller synthesis against external specifications (i.e., specifications given as another timed automaton). Cassez et al. [2002] showed that the controller synthesis problem with an unknown sampling rate is undecidable for timed automata. Bouyer et al. [2003] were the first who investigated the impact of partial observability. Besides the fundamental undecidability result for the general problem, they also showed that one obtains a 2EXPTIME-complete synthesis procedure when the granularity of the controller is fixed in advance. Recently, Peter and Finkbeiner [2012] extended this line of research and established complexity bounds for various bounded synthesis problems for timed automata.

Chapter 8

Template-based Timed Controller Synthesis

At the end of the previous chapter, we have seen that the complexity of synthesis boils down to the complexity of model checking whenever we assume that the memory of the controller is at least exponentially smaller than the memory of the plant. In this chapter, we build upon that observation and propose a new synthesis approach, where the size and general shape of the controller is fixed in advance in the form of a *template*.

Template-based synthesis has several attractive features: Since the observations of the controller are limited by the template, template-based synthesis naturally solves the controller synthesis problem with partial observability. The size of the controller is also limited by the size of the template. Because the templates model standard types of controllers, the synthesized controllers are well-structured, resembling a manually built controller.[1]

We first give an overview on different types of templates in Section 8.1. Then, in Section 8.2, we give a formal definition of the template-based synthesis problem and provide a complexity analysis. In Section 8.3, we describe effective algorithms to find feasible parameter instantiations. The chapter concludes with an overview on related works in Section 8.4.

8.1 Template Types

A template is a timed or untimed automaton with parametric control structure. To illustrate the basic idea behind template-based controller synthesis, in the following, we present some standard template types.

Cyclic executive. In the *cyclic-executive template*, shown in Figure 8.1, the controller implements some schedule according to which the events emit-

[1] The template-based synthesis approach has been published in [Finkbeiner and Peter, 2012].

ted by the plant are handled; the controller alternates between waiting for an uncontrollable observation and responding with some controllable action. We use the parametric control structure to abstract from the actual events that are communicated between plant and controller. The template families are organized according to the number of *phases*. Typically, we start the synthesis process with small templates and then iteratively increase the size until an optimal controller is found.

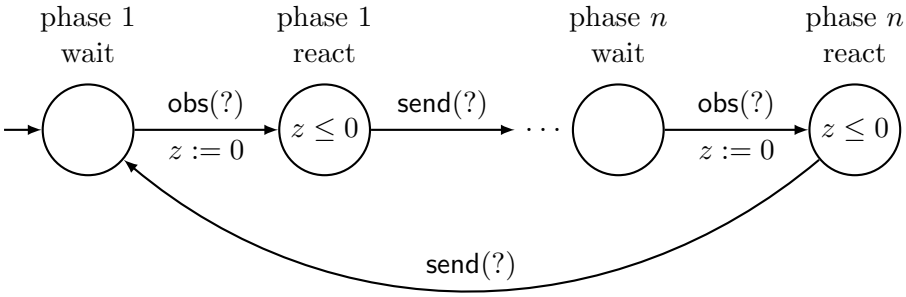


Figure 8.1: Cyclic executive template with n phases.

Wait for. In the *wait-for template*, shown in Figure 8.2, the controller is represented as a sequence of *control points*. For each control point, we have a Boolean expression over discrete plant variables that are visible to the controller. Whenever the condition is satisfied, a given effect is executed that changes the value of some controllable variable that affects the behavior of the plant. We use the parametric control structure to abstract from the actual integer values in the conditions and the effects.

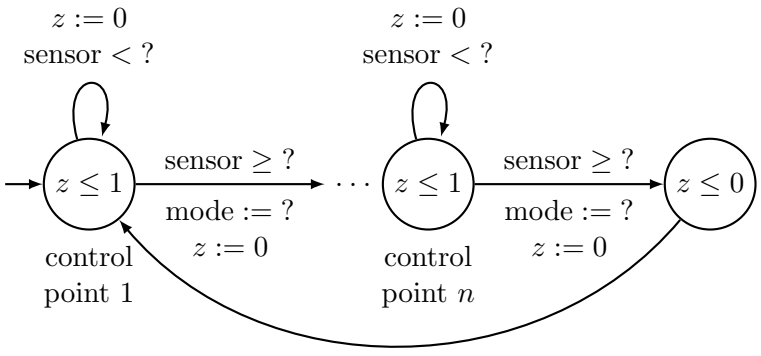


Figure 8.2: Wait-for template with n control points.

Distributed controllers. A template is not restricted to be a monolithic automaton; we can also use the template-based synthesis approach to synthesize a *distributed controller*, shown in Figure 8.3, which is given as a network of monolithic template automata. Following this modeling pattern, it is possible to introduce individual controllers each capturing a particular aspect of the plant. In principle, one can assume an arbitrary communication topology among the plant and the controllers. We note that the general distributed synthesis problem is undecidable even for pure discrete systems [Pnueli and Rosner, 1990, Finkbeiner and Schewe, 2005].

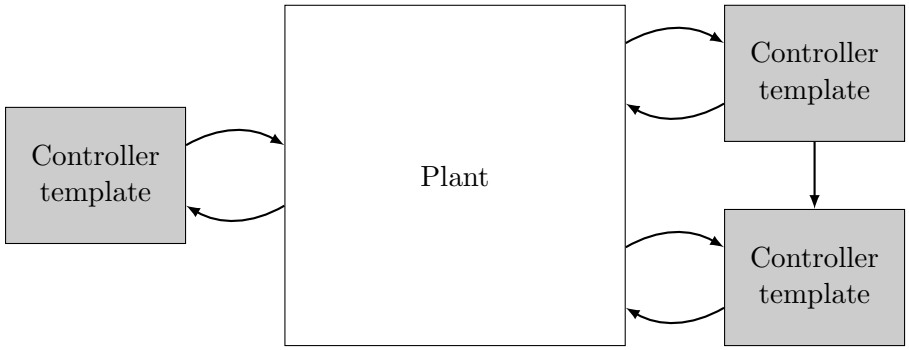


Figure 8.3: For a network containing several monolithic template automata and a timed automaton representing the plant, template-based synthesis yields a distributed controller.

8.2 Definition and Complexity

In this section, we formalize the template-based synthesis problem and analyze its complexity.

A *controller template* is a tuple $(\mathcal{T}, P, \Pi)$ consisting of a timed automaton $\mathcal{T} = (Q, q_0, \Sigma, E, I)$, a finite set of Boolean parameters P, and a total function $\Pi : \vec{P} \to 2^E$ defining which edges are enabled for a given parameter valuation, where $\vec{P} = P \to \{\mathbf{false}, \mathbf{true}\}$ is the set of all parameter valuations. In the following, we will assume that the timed automaton modeling the environment (or plant) is already integrated (by parallel composition) in $\mathcal{T}$. As usual, we assume that the controller does neither reset plant clocks, inhibit plant actions, nor introduce timelocks.

Definition 8.2.1. *For a controller template $(\mathcal{T}, P, \Pi)$ and a set of bad states B, where $\mathcal{T} = (Q, q_0, \Sigma, E, I)$, the* instantiation problem *asks for a parameter valuation $\vec{p} \in \vec{P}$ such that $\mathcal{I} = (Q, q_0, \Sigma, \Pi(\vec{p}), I)$ and $\mathcal{I} \not\models B$.*

We call an instantiation of the template that satisfies the condition of the definition *feasible*. Synthesizing a template-based controller corresponds to

statically finding a feasible instantiation. Observe that this is in contrast to the classical formulation of the timed controller synthesis problem by Maler et al. [1995], Asarin et al. [1998], where the controller is an arbitrary timed automaton whose behavior depends *dynamically* on the observed events of the plant.

In terms of sequential circuit machines, template-based synthesis corresponds to a setting, where the universal memory contains a polynomial number of bits while the existential player has no memory at all and can only observe logarithmically many bits.

Theorem 8.2.2. *The instantiation problem for a controller template and a set of bad states is complete for* $\mathcal{C}((1, \mathcal{L}, \mathcal{L}), (0, \mathcal{P})) = \text{PSPACE}$.

Proof. According to Corollary 7.3.1, AG-model checking for timed automata is complete for $\mathcal{C}((0, 0, \mathcal{L}), (0, \mathcal{P})) = \text{PSPACE}$, which, according to Theorem 4.2.8, coincides with $\mathcal{C}((1, \mathcal{L}, \mathcal{L}), (0, \mathcal{P}))$. Thus, the hardness for the template instantiation problem follows immediately, since timed model checking is just the special case, where we have no parameters at all (i.e., if for all controller templates $(\mathcal{T}, P, \Pi)$ we have $P = \emptyset$).

For the upper bound, observe that we can transform a given template instantiation problem into a bounded controller synthesis problem, where the plant queries the controller whenever a value for a parameter needs to be determined. By disallowing any existential memory, we make sure that the controller always gives consistent answers. Since there can be at most a polynomial number of parameters, the plant (i.e., Player A) only needs logarithmically many bits to communicate the parameter index to the controller (i.e., Player E). In turn, Player E responds with the value of the parameter, which just requires a single bit. Hence, containment in $\mathcal{C}((1, \mathcal{L}, \mathcal{L}), (0, \mathcal{P}))$. □

We note that for the general timed synthesis problem as introduced by Maler et al. [1995] and Bouyer et al. [2003], the size of the controller can be exponential or even, in case of partial observability, doubly-exponential in the size of the plant. This contrasts with template-based synthesis, which has not only a much smaller worst-case complexity, but also has the advantage that the size of the controller is fixed in advance.

Template-based synthesis thus provides a much more promising setting for effective controller synthesis than the standard approach. The remainder of this chapter is devoted to the development of an efficient template-based synthesis algorithm, which is supplemented with an experimental evaluation in the next chapter.

8.3 Symbolic Parameter Synthesis

We now present a symbolic algorithm for finding feasible instantiations for a given controller template $(\mathcal{T}, P, \Pi)$ with $\mathcal{T} = (Q, q_0, \Sigma, E, I)$ and a set of bad states $B \subseteq S$, where S is the set of states of $\mathcal{T}$. In the rest of this section, we assume that $\mathcal{T}$, P, Π, S, and B are fixed.

We develop the algorithm in three steps: first, we describe the immediate, exact, computation of the set of feasible instantiations based on forward and backward propagation; then we give an approximate computation based on an abstraction of the template; finally, we describe an abstraction refinement procedure, which increases the precision of the approximate computation until either a feasible instantiation has been found, or it has been shown that no feasible instantiation exists.

8.3.1 Precise Computation of the Feasible Instantiations

The precise set of feasible instantiations can be computed in a standard fixed point construction that either starts from the initial state and propagates, in a forward manner, the reachable combinations of states and parameter valuations, or starts with the bad states and propagates, in a backward manner, those combinations of states and parameter valuations that have a path to the bad states.

To accommodate both directions, we define a successor and a predecessor propagation function $\mathsf{Succ}, \mathsf{Pred} : 2^{S \times \vec{P}} \to 2^{S \times \vec{P}}$ with

$$\begin{aligned}
\mathsf{Succ}(Y) = \big\{ (s', \vec{p}) \in S \times \vec{P} \mid \\
\exists \delta \in \Pi(\vec{p}) : \exists s \in S : (s, \vec{p}) \in Y \wedge s \xrightarrow{\delta} s' \big\} \text{ and} \\
\mathsf{Pred}(Y') = \big\{ (s, \vec{p}) \in S \times \vec{P} \mid \\
\exists \delta \in \Pi(\vec{p}) : \exists s' \in S : (s', \vec{p}) \in Y' \wedge s \xrightarrow{\delta} s' \big\}.
\end{aligned}$$

The set FR of forward-reachable states and parameter valuations and the set BR of backward-reachable states and parameter valuations are obtained by the following fixed point computations (the index identifies the round of the fixed point iteration):

$$\begin{array}{lll}
\mathsf{FR}_0 & = \{(l_0, \vec{0})\} \times \vec{P} & \mathsf{BR}_0 = B \times \vec{P} \\
\mathsf{FR}_{i+1} & = \mathsf{Succ}(\mathsf{FR}_i) \cup \mathsf{FR}_i & \mathsf{BR}_{i+1} = \mathsf{Pred}(\mathsf{BR}_i) \cup \mathsf{BR}_i \\
\mathsf{FR} & = \lim_i \mathsf{FR}_i & \mathsf{BR} = \lim_i \mathsf{BR}_i.
\end{array}$$

Clearly, if there is some $(s, \vec{p}) \in \mathsf{FR}_i$ then this means that state s is reached after $i \in \mathbb{N}$ forward steps for parameter valuation $\vec{p}$, which corresponds to a path $s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_i} s$, where each $\delta_1, \delta_2, \ldots, \delta_i$ is in $\Pi(\vec{p})$. Dually, if there is some state $(s, \vec{p}) \in \mathsf{BR}_i$ then this means that state s is reached

after $i \in \mathbb{N}$ backward steps for parameter valuation $\vec{p}$, which corresponds to a path $s \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_i} b$, where $b \in B$ and each $\delta_1, \delta_2, \ldots, \delta_i$ is in $\Pi(\vec{p})$.

We can obtain the feasible instantiations either by looking for parameter valuations in FR that are not paired up with bad states, or by looking for parameter valuations in BR that are not paired up with the initial state. Both constructions identify the same set of feasible instantiations.

Theorem 8.3.1. *The set*

$$G = \{\vec{p} \in \vec{P} \mid (B \times \{\vec{p}\}) \cap \mathsf{FR} = \emptyset\} = \{\vec{p} \in \vec{P} \mid ((l_0, \vec{0}), \vec{p}) \notin \mathsf{BR}\}$$

consists of exactly the feasible instantiations.

Proof. It is easy to see that $G_f = \{\vec{p} \in \vec{P} \mid B \times \{\vec{p}\} \cap \mathsf{FR} = \emptyset\}$ contains exactly those parameter valuations for which no bad state is forward reachable. Hence, by definition, $G = G_f$.

On the other hand, because for every forward reachable state s, the initial state s_0 is also backward reachable from s, we have that $G_b = \{\vec{p} \in \vec{P} \mid ((l_0, \vec{0}), \vec{p}) \notin \mathsf{BR}\}$ coincides with G_f. Hence, $G = G_f = G_b$. □

In practice, neither construction performs well. The problem is that it is difficult and expensive to maintain the correlation between parameter valuations and reachable states; typically, each parameter valuation results in a different set of states.

Instead of directly computing the precise set of parameter valuations, in the next subsection, we will present an abstraction technique that allows us to reason about approximations of parameter valuations.

8.3.2 The Focus Abstraction

We now consider an abstraction of the template based on a given set $P \subseteq \vec{P}$ of parameter valuations, which we call *focus*. We use the parameter valuations in P to obtain an over- or underapproximation of the sets FR and BR, by considering P as an equivalence class: we require that a transition must exist for some or all parameter valuations in P, respectively. In the following, we use an overapproximation for the forward construction and an underapproximation for the backward construction; obviously, all constructions can also be dualized. We obtain the following approximate successor and predecessor functions: $\overline{\mathsf{Succ}}^P, \underline{\mathsf{Pred}}^P : 2^S \to 2^S$ with

$$\overline{\mathsf{Succ}}^P(Y) = \left\{s' \in S \mid \exists \vec{p} \in P : \exists \delta \in \Pi(\vec{p}) : \exists s \in Y : s \xrightarrow{\delta} s'\right\} \text{ and}$$
$$\underline{\mathsf{Pred}}^P(Y') = \left\{s \in S \mid \forall \vec{p} \in P : \exists \delta \in \Pi(\vec{p}) : \exists s' \in Y' : s \xrightarrow{\delta} s'\right\}.$$

Replacing the precise Succ and Pred operators in the fixed point construction from Subsection 8.3.1, we obtain two new fixed point constructions for the approximations $\overline{\mathsf{FR}}^P$ and $\underline{\mathsf{BR}}^P$:

$$\begin{array}{ll} \overline{\mathsf{FR}}_0^P = \{(l_0, \vec{0})\} & \underline{\mathsf{BR}}_0^P = B \\ \overline{\mathsf{FR}}_{i+1}^P = \overline{\mathsf{Succ}}^P(\overline{\mathsf{FR}}_i^P) \cup \overline{\mathsf{FR}}_i^P & \underline{\mathsf{BR}}_{i+1}^P = \underline{\mathsf{Pred}}^P(\underline{\mathsf{BR}}_i^P) \cup \underline{\mathsf{BR}}_i^P \\ \overline{\mathsf{FR}}^P = \lim_i \overline{\mathsf{FR}}_i^P & \underline{\mathsf{BR}}^P = \lim_i \underline{\mathsf{BR}}_i^P. \end{array}$$

Clearly, if there is some state $s \in \overline{\mathsf{FR}}_i^P$ then this means that state s is reached after $i \in \mathbb{N}$ forward steps for a set of parameter valuations P, which corresponds to a path

$$s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_i} s,$$

where, for each δ_i, there is a $\vec{p}_i \in P$ such that δ_i in $\Pi(\vec{p}_i)$. Dually, if there is some state $s \in \underline{\mathsf{BR}}_i^P$ then this means that state s is reached after $i \in \mathbb{N}$ backward steps for a set of parameter valuations P, which corresponds to a path

$$s \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \ldots \xrightarrow{\delta_i} b,$$

where $b \in B$ and, for each δ_i and each $\vec{p} \in P$, we have δ_i in $\Pi(\vec{p})$.

The following lemma clarifies the relationships between the approximate and precise versions of FR and BR: $\overline{\mathsf{FR}}^P$ overapproximates FR on P, $\underline{\mathsf{BR}}^P$ underapproximates BR on P.

Lemma 8.3.2. *For every set $P \subseteq \vec{P}$ of parameter valuations, it holds that*

$$\overline{\mathsf{FR}}^P \supseteq \{s \in S \mid \exists \vec{p} \in P : (s, \vec{p}) \in \mathsf{FR}\} \text{ and}$$
$$\underline{\mathsf{BR}}^P \subseteq \{s \in S \mid \exists \vec{p} \in P : (s, \vec{p}) \in \mathsf{BR}\}.$$

Proof. We first show $\overline{\mathsf{FR}}^P \supseteq Y = \{s \in S \mid \exists \vec{p} \in P : (s, \vec{p}) \in \mathsf{FR}\}$. By definition, for every state s contained in Y, there is a $\vec{p} \in P$ such that there is a path

$$s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_n} s,$$

where, for each $0 \leq i \leq n$, $\delta_i \in \Pi(\vec{p})$. On the other hand, by definition, for every state s contained in $\overline{\mathsf{FR}}^P$, there is a path

$$s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_n} s,$$

where, for each δ_i, $0 \leq i \leq n$, there is a $\vec{p}_i \in P$ such that δ_i in $\Pi(\vec{p}_i)$. Now, when we fix each $\vec{p}_i = \vec{p}$, for some $\vec{p} \in P$, then we get Y. Hence, $\overline{\mathsf{FR}}^P \supseteq Y$.

Secondly, we show $\underline{\mathsf{BR}}^P \subseteq Y = \{s \in S \mid \exists \vec{p} \in P : (s, \vec{p}) \in \mathsf{BR}\}$. By definition, for every state s contained in Y, there is a $b \in B$ and a $\vec{p} \in P$ such that there is a path

$$s \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_n} b,$$

where, for each $0 \leq i \leq n$, $\delta_i \in \Pi(\vec{p})$. On the other hand, by definition, for every state s contained in $\underline{\mathsf{BR}}^P$, there is a $b \in B$ such that there is a path

$$s \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_n} b,$$

where, for each δ_i, $0 \leq i \leq n$, and each $\vec{p} \in P$, we have $\delta_i \in \Pi(\vec{p})$. Now, it is easy to see that every state s in $\underline{\mathsf{BR}}^P$, is backward reachable for an arbitrary $\vec{p} \in P$, and thus, also contained in Y. Hence, $\underline{\mathsf{BR}}^P \subseteq Y$. □

Combining Lemma 8.3.2 with Theorem 8.3.1, we obtain that the focus abstraction allows us to approximate the set of feasible instantiations: A set of parameter valuations P definitely represents feasible instantiations if no bad states appear in $\overline{\mathsf{FR}}^P$. Dually, the parameter valuations in P definitely represent infeasible instantiations if the initial state appears in $\underline{\mathsf{BR}}^P$. Hence, we obtain the following lower and upper bounds for the set of feasible instantiations.

Theorem 8.3.3. *Let G be the precise set of feasible instantiations. For every set $P \subseteq \vec{P}$, it holds that*

$$\left\{\vec{p} \in P \mid B \cap \overline{\mathsf{FR}}^P = \emptyset\right\} \subseteq G \subseteq \left\{\vec{p} \in \vec{P} \mid \vec{p} \in P \Rightarrow (l_0, \vec{0}) \notin \underline{\mathsf{BR}}^P\right\}.$$

Proof. We first show that $Y = \left\{\vec{p} \in P \mid B \cap \overline{\mathsf{FR}}^P = \emptyset\right\} \subseteq G$. By Lemma 8.3.2, $\overline{\mathsf{FR}}^P$ overapproximates those states that are precisely forward reachable for P. By definition of Y, either (1) $Y = \emptyset \subseteq G$, if $B \cap \overline{\mathsf{FR}}^P \neq \emptyset$, or (2) $Y = P$, if $B \cap \overline{\mathsf{FR}}^P = \emptyset$, i.e., surely no bad state is reachable for P. By Theorem 8.3.1, a parameter valuation is feasible if no bad state is forward reachable. Hence, also in case (2), $Y = P \subseteq G$.

Secondly, we prove that $G \subseteq Y = \left\{\vec{p} \in \vec{P} \mid \vec{p} \in P \Rightarrow (l_0, \vec{0}) \notin \underline{\mathsf{BR}}^P\right\}$ by showing that every $\vec{p} \in G$ is also contained in Y. We distinguish two cases.
Case 1: assume $\vec{p} \in G \cap P$. By Theorem 8.3.1, since $\vec{p} \in G$, the initial state is not backward reachable for $\vec{p}$. By Lemma 8.3.2, $\underline{\mathsf{BR}}^P$ underapproximates those states that are precisely backward reachable for P. Since $\vec{p} \in P$, we have that the premise and the implicant of the definition of Y are true, and thus, $\vec{p}$ is also contained in Y.
Case 2: assume $\vec{p} \in G \setminus P$. $\vec{p}$ violates the premise in the implication of the definition of Y, and thus, is trivially contained in Y.
Hence, $G \subseteq Y$. □

In the next subsection, we will describe an automatic refinement algorithm for the Focus abstraction.

8.3.3 Abstraction Refinement

We now describe a refinement procedure that computes an increasingly precise approximation of the set of feasible instantiations. The procedure starts with the set $\vec{P}$ of all parameter valuations, and then splits the set into smaller and smaller subsets, until either a feasible instance is found, or it is established that no feasible instance exists.

Algorithm 1 $\mathsf{Solve}(P)$: The algorithm computes a safe subset of a given set P of parameter valuations, or returns fail if no safe subset exists.

```
1: if P = ∅ then
2:    return fail
3: else if (l_0, 0⃗) ∈ BR^P then
4:    return fail
5: else if FR̄^P ∩ BR^P = ∅ then
6:    return P
7: else
8:    P_1 := Refine(P)
9:    R_1 := Solve(P_1)
10:   if R_1 ≠ fail then
11:      return R_1
12:   else
13:      P_2 := P \ P_1
14:      return Solve(P_2)
15:   end if
16: end if
```

The procedure is shown as Algorithm 1. The input to the procedure is the current focus P, for which we initially use $\vec{P}$. Unless the (un)reachability of some bad state can be surely established, after each refinement step, Solve recurs on the refined focus. In each call of Solve, the set of bad states are augmented with the states in $\underline{\mathsf{BR}}^P$. This is justified by the following lemmas, which state that the old underapproximation $\underline{\mathsf{BR}}^P$ is a subset of the new underapproximation $\underline{\mathsf{BR}}^{P'}$ for a refinement $P' \subset P$, and that excluding $\underline{\mathsf{BR}}^P$ from $\overline{\mathsf{FR}}^P$ does not affect the resulting upper bound on the feasible instantiations.

Lemma 8.3.4. *For two sets $P, P' \subseteq \vec{P}$ of parameter valuations such that $P' \subset P$, it holds that $\underline{\mathsf{BR}}^P \subseteq \underline{\mathsf{BR}}^{P'}$.*

Proof. We assume $P' \subset P$ and prove $\underline{\mathsf{BR}}^P \subseteq \underline{\mathsf{BR}}^{P'}$ by showing that every state in $\underline{\mathsf{BR}}^P$ is also contained in $\underline{\mathsf{BR}}^{P'}$. By definition, for every state $s \in \underline{\mathsf{BR}}^P$, there is a $b \in B$ such that there is a path

$$s \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_n} b,$$

where, for each δ_i, $0 \leq i \leq n$, and each $\vec{p} \in P$, we have $\delta_i \in \Pi(\vec{p})$. Now, it is easy to see that the same path also exists for the smaller set $P' \subset P$. □

Lemma 8.3.5. *For every set $P \subseteq \vec{P}$ of parameter valuations, it holds that*

$$\{\vec{p} \in P \mid B \cap \overline{\mathsf{FR}}^P = \emptyset\} = \{\vec{p} \in P \mid \underline{\mathsf{BR}}^P \cap \overline{\mathsf{FR}}^P = \emptyset\}.$$

Proof. Let $Y = \{\vec{p} \in P \mid B \cap \overline{\mathsf{FR}}^P = \emptyset\}$ and $Y' = \{\vec{p} \in P \mid \underline{\mathsf{BR}}^P \cap \overline{\mathsf{FR}}^P = \emptyset\}$. We distinguish two cases:
Case 1: assume $B \cap \overline{\mathsf{FR}}^P \neq \emptyset$. In this case, since $B \subseteq \underline{\mathsf{BR}}^P$, $Y = Y' = \emptyset$.
Case 2: assume $B \cap \overline{\mathsf{FR}}^P = \emptyset$. Observe that, for each state $s \in \underline{\mathsf{BR}}^P$, there is a path from s to some state $b \in B$, for all parameter valuations in P. On the other hand, by assumption, $\overline{\mathsf{FR}}^P$ does not contain any states leading to some bad state for some parameter valuation in P. Hence, $\underline{\mathsf{BR}}^P \cap \overline{\mathsf{FR}}^P = \emptyset$, and thus, $Y = Y'$. □

It remains to specify the function Refine, which is called in procedure Solve to find an appropriate subset of $\vec{P}$ to split on. Since $\vec{P}$ is finite, we could, in principle, choose any strict (and non-empty) subset of $\vec{P}$ during the refinement step. In the following we describe a heuristic choice that has proved useful in practice: we choose a set of parameter valuations that are guaranteed to increase $\underline{\mathsf{BR}}^P$ in the next iteration.

Suppose the termination conditions of procedure Solve are not true yet, i.e., the initial state is not in $\underline{\mathsf{BR}}^P$ and there are still states in $\overline{\mathsf{FR}}^P \cap \underline{\mathsf{BR}}^P$. We choose a state $s \in \overline{FR}^P \setminus \underline{\mathsf{BR}}^P$ and a state $s' \in \underline{BR}^P$, such that there exists a transition $\delta \in \Pi(\vec{p})$ that leads from s to s' for some $\vec{p} \in P$, but not for all $\vec{p} \in P$. The refinement proceeds with the parameter valuations that allow a transition from s to s':

$$\mathsf{Refine}(P) = \{\vec{p} \in P \mid \exists \delta \in \Pi(\vec{p}) : s \xrightarrow{\delta} s'\}$$

Since such a pair s, s' of states can be found until the termination conditions of procedure Solve become true, we obtain that Refine always ensures progress of our refinement algorithm.

Lemma 8.3.6. *For every set of parameter valuations $P \subseteq \vec{P}$, if $P \neq \emptyset$, $(l_0, \vec{0}) \notin \underline{\mathsf{BR}}^P$, and $\overline{\mathsf{FR}}^P \cap \underline{\mathsf{BR}}^P \neq \emptyset$, then there is a state $s \in \overline{\mathsf{FR}}^P \setminus \underline{\mathsf{BR}}^P$ and a state $s' \in \underline{\mathsf{BR}}^P$ such that*

$$\emptyset \neq \mathsf{Refine}(P) \subsetneq P.$$

Proof. By assumption, since $\overline{\mathsf{FR}}^P \cap \underline{\mathsf{BR}}^P \neq \emptyset$, there is a path from $s_0 = (l_0, \vec{0})$ to some state $b \in \underline{\mathsf{BR}}^P$ of the form

$$s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} \dots s_n \xrightarrow{\delta_n} b,$$

where, for each δ_i, $0 \leq i \leq n$, there is a $\vec{p}_i \in P$ such that δ_i in $\Pi(\vec{p}_i)$. Since $s_0 \notin \underline{\mathsf{BR}}^P$, there is surely some $s_i \in \overline{\mathsf{FR}}^P \setminus \underline{\mathsf{BR}}^P$, whose successor s_{i+1} is in $\underline{\mathsf{BR}}^P$. In fact, if b is the first state from $\underline{\mathsf{BR}}^P$, which appears on the path, then a candidate edge for refinement is $s_n \xrightarrow{\delta_n} b$, i.e., $s = s_n$ and $s' = b$. Thus, $\emptyset \subset \mathsf{Refine}(P)$ as $\mathsf{Refine}(P)$ contains those $\vec{p} \in P$, for which $\delta_n \in \Pi(\vec{p})$. On the other hand, if $\mathsf{Refine}(P)$ subsumes exactly those $\vec{p} \in P$, for which $\delta_n \in \Pi(\vec{p})$, we can conclude that $\mathsf{Refine}(P) \subset P$, as s_n would be contained in $\underline{\mathsf{BR}}^P$ in the first place if δ_n would be available for all $\vec{p} \in P$. □

Putting everything together, we obtain the following correctness theorem for $\mathsf{Solve}(\vec{P})$, where Lemma 8.3.6 guarantees termination and Theorem 8.3.3 guarantees soundness of the result.

Theorem 8.3.7. *Called with the set $\vec{P}$ of parameter valuations, Procedure $\mathsf{Solve}(\vec{P})$ terminates after at most $|\vec{P}|$ refinement steps and either computes a feasible template instantiation or reports failure, in which case no feasible template instantiation exists.*

Proof. Termination of $\mathsf{Solve}(\vec{P})$ is easily established by Lemma 8.3.6, which ensures progress of each refinement step. Since there are only finitely many parameter valuations in $\vec{P}$, the focus P will eventually, after at most $|\vec{P}|$ refinement steps, end up with $\emptyset$ or some parameter valuations, for which one of the termination conditions becomes true.

Soundness of $\mathsf{Solve}(\vec{P})$ follows immediately from Theorem 8.3.3.

Completeness of $\mathsf{Solve}(\vec{P})$ is established by the fact that Solve does not miss any parameter valuation: in the lines 9 and 14 of Algorithm 1, Solve recurs on the refinement P_1 and on $P \setminus P_1$. □

8.3.4 Towards an Efficient Implementation

In conclusion, the focus abstraction reduces controller synthesis to an abstraction refinement-based reachability analysis. Here, we extend timed automata by discrete parameter variables that succinctly represent sets of template instantiations. An efficient implementation of Algorithm 1 thus requires an efficient treatment of both sets of clock and parameter values.

While specialized data structures exist for efficiently representing sets that consists only of clock values or only of discrete values, the combined treatment of both is a challenging technical problem that arises in the analysis of timed systems with a succinctly specified control structure. The next chapter addresses this problem and presents a general technique to efficiently treat orthogonal sources of complexity induced by independent forms of succinctness. For timed automata and template-based synthesis in particular, in Chapter 10, we will present the tool Synthia and report on an experimental evaluation.

8.4 Bibliographic Remarks

A first more practical approach to timed controller synthesis, implemented in the tool SynthKro, was proposed by Altisen and Tripakis [2002]. The approach requires, however, an expensive preprocessing step. Cassez et al. [2005] presented a symbolic algorithm, implemented in the tool Uppaal-Tiga [Behrmann et al., 2007], that avoids the upfront state explosion by combining the backward attractor construction with a forward zone graph exploration. The first practical work on timed controller synthesis under incomplete information is due to Cassez et al. [2007], who proposed to restrict the choices and the observability of the controller so that a zone-based synthesis algorithm remains possible. An extension of this work uses alternating timed simulation relations to efficiently control partially observable systems [Chatain et al., 2009]. This approach has also been implemented in Uppaal-Tiga. The template-based synthesis approach presented in this chapter was introduced by Finkbeiner and Peter [2012].

Template-based synthesis is related to the *bounded synthesis* approach [Schewe and Finkbeiner, 2007], where one fixes the size (but not the structure) of the controller. Bounded synthesis has so far been limited to purely discrete systems. There are efficient algorithms for bounded synthesis based on SMT-solving [Finkbeiner and Schewe, 2007], antichains [Filiot et al., 2009], and BDDs [Ehlers, 2010], which, however, unfortunately do not seem to have straight-forward extensions to the timed case. Another interesting restriction on the type of controllers to be considered has been proposed by Lustig and Vardi [2009]: *synthesis from component libraries* attempts to construct a controller by assembling routines from a given library. The difference to template-based synthesis is that the synthesized controller is a combination of predefined components rather than an instantiation of a parametric template. Currently, this approach is also limited to discrete systems.

Chapter 9

Combined-Symbolic Analysis of Timed Systems

In Chapter 7, we have seen that the source of the exponential blow-up in the analysis of timed automata is the fact that clocks can be used to succinctly encode state spaces of exponential size. Hence, from a theoretical point of view, unless PTime = PSpace, any actual analysis algorithm will intrinsically suffer from an exponential blow-up in at least some cases. Left with this rather pessimistic insight, the best one can achieve is to devise heuristics that avoid the exponential blow-up, maybe not for all, but hopefully for many practically relevant cases.

Independently of the complexity induced by clocks, timed systems can orthogonally suffer from an exponential blow-up caused by a succinctly specified control structure. For example, in Section 5.1, we have seen that networks of communicating state machines, which represent an untimed subclass of networks of timed automata, are already succinct enough to cause an exponential blow-up in their analysis.

A crucial point in any analysis algorithm for timed automata is how sets of states are symbolically represented.[1] The choice of the right data structure is essential for *preserving the succinctness* during the analysis. While there exist efficient techniques for representing sets that contain either only clock values or only discrete values, the efficient representation and manipulation of sets of pairs of continuous and discrete state information is widely regarded as an important open problem since the introduction of timed automata.

This chapter addresses this practical challenge and, based on an insightful circuit-based interpretation of the semantics of timed automata in Section 9.1, in Section 9.2, we present a solution that yields promising experimental results, which will be presented in Chapter 10. This chapter

[1] Necessarily, one has to employ some symbolic representation as there are uncountably many timed states.

concludes with an overview on efficient techniques for the analysis of timed automata in Section 9.3.

9.1 The Combined Succinctness of Timed Automata

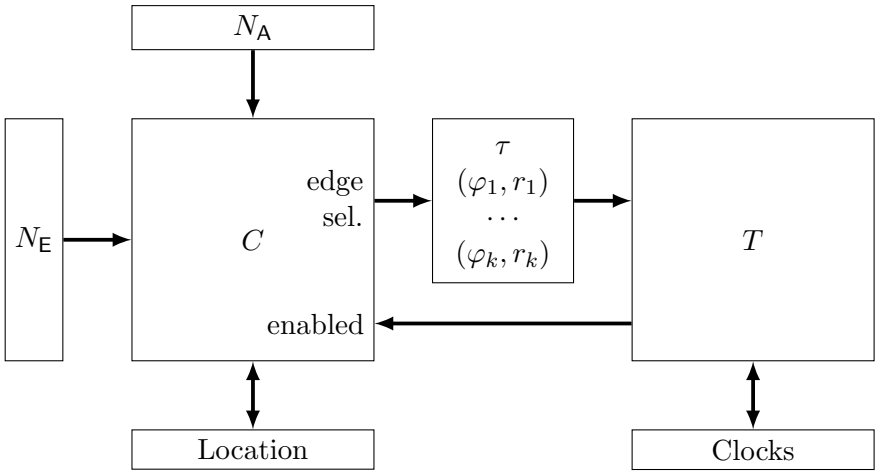


Figure 9.1: The main circuit C_{A} of the finite semantics of timed automata can be divided into a sub-circuit C representing the discrete behavior and a sub-circuit T representing the timed behavior.

A closer look on the circuit-based interpretation of the semantics of timed automata reveals a minimal cut in the communication structure: As shown in Figure 9.1, when we divide the combinatorial circuit representing the transitions of the finite semantics (1) into a sub-circuit C that operates on the state bits representing the control locations and (2) into a sub-circuit T that operates on the state bits representing the clocks, we notice that C only needs $\lceil \log(k+1) \rceil$ bits to communicate which clock operation should be executed to T, where k is the number of distinct pairs of clock constraints and resets that appear in the given automaton. For the other direction, T only needs 1 bit to communicate whether the operation could be executed (i.e., whether the current clock values satisfy the selected guard). Recall from Chapter 7 that the number of state bits of T can be polynomial in the number of clocks and the size of the binary representation of the constants of the given timed automaton. According to Section 5.1, in case of *networks* of timed automata or due to the introduction of discrete parameter variables, the number of state bits of C can be polynomial in the number of the components.

The separation of C and T also makes sense concerning the way how both circuits manipulate their respective state bits: When executing a dis-

crete step, C only manipulates the local state bits of those components that synchronize. Also, in the presence of discrete parameter variables, whose values are determined statically (i.e., they are initially chosen and remain fixed), executing a discrete step only changes the state bits of some components, but not the bits of the parameter values. On the other hand, the computation of the timed successor states is more of arithmetic nature and may change all timed state bits at once: While elapsing of time corresponds to an incrementation operation, as clocks (whose values need to be preserved in the state bits) are de facto integer counters, resetting corresponds to an addition / subtraction operation, as clock differences (whose values also need to be preserved in the state bits) are de facto integer counters as well.

Being aware of this difference, it gets clear that an efficient treatment of one part might be very inefficient for the other one. Therefore, instead of aiming for a monolithic approach that represents every aspect of the state space in a single data structure, we rather propose to exploit the (hopefully) slack interface between the discrete and the timing part and aim for a *combination* of two different symbolic representations, each *specialized* for a particular aspect.

In the rest of this chapter, we will work out such a combination approach for the reachability analysis of timed automata that can be used to combine BDDs with DBMs.

9.2 Combining Symbolic Data Structures

Based on the fundamental considerations in Section 9.1, we now describe our combination approach in more technical detail.[2]

In the following, for the sake of simplicity, we assume that discrete parameter variables are incorporated as additional locations in the control structure of the network of timed automata under consideration. For this entire section, we fix a network of timed automata

$$\mathcal{N} = \mathcal{T}_1 \| \mathcal{T}_2 \| \dots \| \mathcal{T}_n$$

and an n-dimensional location vector $\vec{q}_b$ representing the global bad location. For each $1 \leq i \leq n$, we assume $\mathcal{T}_i = (Q_i, q_0^i, \Sigma_i, E_i, I_i)$ and we define

$$Q = \prod_{i=1}^{n} Q_i \qquad \text{and} \qquad \Sigma = \bigcup_{i=1}^{n} \Sigma_i$$

[2]The idea of analyzing timed systems in sequences of syntactic abstractions with increasing precision, which can be seen as a precursor work to the combination approach, has been published in [Peter and Mattmüller, 2009]. The idea of strictly separating the discrete and the timing part along with a first experimental evaluation has been published in [Ehlers et al., 2010b]. The combination approach itself has been published in [Ehlers et al., 2010c].

9.2.1 Overview

The idea of our combination approach is to use a data structure specialized for Boolean functions to produce a sequence of *syntactic abstractions* of $\mathcal{N}$ with increasing precision. We obtain the abstractions by merging locations such that the abstract control structure either under- or overapproximates the behavior of $\mathcal{N}$. For each abstraction, we apply the *standard zone-based reachability algorithm* (to be defined in Section 9.2.4) to obtain an under- or an overapproximation of the reachable states.

Instead of computing the reachable states on the most precise control structure of $\mathcal{N}$ directly, our key idea is to analyze a sequence of simpler timed automata, where each analysis process reuses the approximations obtained from the previous one. That is, we use reachable state set approximations computed on coarse (and therefore simple) abstractions to (1) obtain refinements that increase the precision of the abstractions, and (2) identify irrelevant parts of $\mathcal{N}$ that do not need to be analyzed. Both soundness and effectiveness of our approach rely on the fact that whenever an abstract state appears in an underapproximation, all subsumed concrete states are surely reachable, and dually, whenever an abstract state is not contained in an overapproximation, all subsumed concrete states are surely unreachable.

9.2.2 Representing the Edge Relation

We introduce the following Boolean variables.

- $\widehat{[\![Q_i]\!]}$ and $\widehat{[\![Q_i]\!]}'$, for every Q_i,
- $\widehat{[\![\Sigma]\!]}^d$, for the decisions Σ,
- $\widehat{[\![C_i]\!]}^c$, for every set C_i that contains all the clock constraints that appear in E_i or I_i, and
- $\widehat{[\![R_i]\!]}^r$, for every set R_i that contains all clock resets r that appear in E_i.

For any two distinct decisions $d, d' \in \Sigma$, we require $[\![d]\!]^d \wedge [\![d']\!]^d \equiv \mathbf{false}$. Also, for two locations $q, q' \in Q_i$ of the same component $\mathcal{T}_i$, we require $[\![q]\!] \wedge [\![q']\!] \equiv \mathbf{false}$ and $[\![q]\!]' \wedge [\![q']\!]' \equiv \mathbf{false}$. Since the components of $\mathcal{N}$ are explicitly given, the number of Boolean variables is only logarithmic per component.

Now, we define the Boolean function that encodes the unsynchronized global edge relation as

$$\Delta_e \equiv \bigwedge_{1 \le i \le n} \bigvee_{(q,d,\varphi,r,q') \in E_i} [\![q]\!] \wedge [\![d]\!]^d \wedge [\![\varphi]\!]^c \wedge [\![r]\!]^r \wedge [\![q']\!]'.$$

Since we only consider (un)reachability properties, we construct the following Boolean function to incorporate the invariants into the guards:

$$I \equiv \bigwedge_{1 \le i \le n} \bigwedge_{q \in Q_i} [\![q]\!] \Rightarrow [\![I_i(q)]\!]^c$$

We combine both Boolean functions by taking their conjunction and existentially quantifying over the Boolean variables that are used to define the synchronization events. We thus obtain the synchronized global edge relation:

$$\Delta \equiv \exists [\![\cdot]\!]^d : \Delta_e \wedge I$$

Obviously, $\mathcal{N}$ can take a discrete step between a global location $\vec{q}$ to another global location $\vec{q}'$, with guard φ and clock resets r, iff

$$\big(\Delta \wedge [\![\vec{q}]\!] \wedge [\![\varphi]\!]^c \wedge [\![r]\!]^r \wedge [\![\vec{q}']\!]'\big) \not\equiv \textbf{false}.$$

For a set of locations L, the *discrete successors* $\mathsf{Succ}_b([\![L]\!])$ of L are represented by the following Boolean function, which is defined over the $[\![\cdot]\!]'$ variables:

$$\mathsf{Succ}_b([\![L]\!]) \equiv \exists [\![\cdot]\!] \; \exists [\![\cdot]\!]^c \; \exists [\![\cdot]\!]^r : \Delta \wedge [\![L]\!] \wedge [\![\varphi]\!]^c \wedge [\![r]\!]^r$$

Similarly, the *discrete predecessors* $\mathsf{Pred}_b([\![L]\!])$ of L are represented by the following Boolean function, which is defined over the $[\![\cdot]\!]$ variables:

$$\mathsf{Pred}_b([\![L]\!]) \equiv \exists [\![\cdot]\!]' \; \exists [\![\cdot]\!]^c \; \exists [\![\cdot]\!]^r : \Delta \wedge [\![L]\!]' \wedge [\![\varphi]\!]^c \wedge [\![r]\!]^r$$

9.2.3 Syntactic Abstractions of Timed Automata

We obtain syntactic abstractions of $\mathcal{N}$ by merging its locations according to a partitioning

$$\Pi \subseteq 2^Q \text{ such that } \biguplus \Pi = Q.$$

For a partition $\pi \in \Pi$, we write $[\![\pi]\!]$ or $[\![\pi]\!]'$ to refer to the Boolean function that characterizes π, defined over unprimed or primed Boolean location variables, respectively. We define

$$\Pi_0^{\mathcal{N}} = \big\{\{\vec{q}_0\}, \{\vec{q}_b\}, Q \setminus \{\vec{q}_0, \vec{q}_b\}\big\}$$

as the *initial partition* of $\mathcal{N}$ that just separates the global initial location $\vec{q}_0$ and the bad location $\vec{q}_b$ from each other and the remaining locations of $\mathcal{N}$.

We define the *quotient of $\mathcal{N}$ with respect to* Π as the timed automaton $(Q, q_0, \Sigma, E, I) = \mathcal{N}/_\Pi$, where

- $Q = \Pi$,

- $q_0 = \pi_0$, where π_0 is the unique partition from Π that contains $\vec{q}_0$,
- $\Sigma = \{d\}$, and
- $I = [\, q \mapsto \mathbf{true} \mid q \in Q \,]$.

The definition of the edge relation E depends on whether the *may quotient* $\lceil \mathcal{N}/_\Pi \rceil$ or the *must quotient* $\lfloor \mathcal{N}/_\Pi \rfloor$ should be constructed. The edge relation of the may quotient is defined as

$$E_{may} = \{(\pi, d, \varphi, r, \pi') \mid [\![\pi]\!] \wedge [\![\varphi]\!]^c \wedge [\![r]\!]^r \wedge [\![\pi']\!]' \wedge \Delta \not\equiv \mathbf{false}\}.$$

The definition of the edge relation of the must quotient E_{must} is a bit more involved. We define E_{must} to be the set that contains those edges $(\pi, d, \varphi, r, \pi')$, for which the following conditions hold:

(1) π, π' are partitions from Π, r is a set of clock resets, and $\varphi_1, \ldots, \varphi_k$ are clock constraints;

(2) $\varphi \equiv \bigwedge_{1 \le j \le k} \varphi_j \not\equiv \mathbf{false}$;

(3) $\Big([\![\pi]\!] \Rightarrow \bigvee_{1 \le i \le k} \exists [\![\cdot]\!]' \, \exists [\![\cdot]\!]^c \, \exists [\![\cdot]\!]^r : \big([\![\varphi_i]\!]^c \wedge [\![r_i]\!]^r \wedge [\![\pi']\!]' \wedge \Delta \big) \Big) \equiv \mathbf{true}$.

Whenever we have two edges $e_1 = (q, d, \varphi_1, r, q')$ and $e_2 = (q, d, \varphi_2, r, q'')$ in E, where E is either E_{may} or E_{must}, with $\varphi_1 \Rightarrow \varphi_2$, we remove e_1 from E.

Clearly, the number of edges in an abstraction scales with the number of distinct clock guard/reset pairs $\mathcal{N}$ exhibits. Note that this performance bottleneck of our approach corresponds to the communication bandwidth between the control and the timing part mentioned in Section 9.1. We refer to Section 10.1.1 on Page 137 for a more detailed discussion concerning this issue.

9.2.4 Computing the Reachable States

For a given abstract timed automaton, we compute the set of reachable states using the *zone graph*[3] construction [Henzinger et al., 1994, Alur, 1999]. We now briefly recall the basic definitions for clock zones, difference bound matrices, and the semi-symbolic reachability analysis on timed automata. For a detailed overview, we refer to Clarke et al. [2001b, Section 17.5].

Clock zones and difference bound matrices. We fix a granularity $\mu = (X, m, c^{\mathsf{max}})$. A *convex clock zone* (or just a *clock zone*) z over μ is represented by a $|X'| \times |X'|$ *difference bound matrix* [Dill, 1989] (DBM) M, where X' is the ordered set $\{x_0\} \uplus X$ and x_0 is a dedicated clock whose value

[3] also sometimes called the *simulation graph* [e.g., by Bouajjani et al., 1997]

is always 0. Each column x and each row y of M correspond to a clock in X'. Semantically, M represents a conjunction of inequalities of clock differences of the form

$$\bigwedge_{y,x \in X'} y - x \prec^M_{(y,x)} M(y,x),$$

where each $\prec^M_{(y,x)} \in \{<, \leq\}$ and $M(y,x) \in \mathbb{Q} \cup \{\infty\}$ with $-c^{\mathsf{max}} \leq M(y,x) \leq c^{\mathsf{max}}$ and $M(y,x) = k \cdot m^{-1}$, for some $k \in \mathbb{Z}$, unless $M(y,x) = \infty$. We write $\mathcal{Z}$ to refer to the set of all clock zones.

DBMs are a convenient data structure for computing fixed points, because they have a normal form property. We say that a DBM M is in *canonical normal form* iff for every $x, y, z \in X'$ we have that

$$\big(M(y,x), \prec^M_{(y,x)}\big) \leq \big(M(y,z), \prec^M_{(y,z)}\big) + \big(M(z,x), \prec^M_{(z,x)}\big),$$

where

$$(c_1, \prec_1) + (c_2, \prec_2) := \begin{cases} (c_1 + c_2, \leq) & \text{if } \prec_1 = \leq \text{ and } \prec_2 = \leq; \\ (c_1 + c_2, <) & \text{otherwise;} \end{cases}$$

and

$$\begin{aligned} &(c_1, \prec_1) \leq (c_2, \prec_2) \ :\Leftrightarrow \\ &\qquad (c_1 < c_2) \ \vee \ (c_1 = c_2) \wedge (\prec_1 = \leq \vee \ \prec_2 = <). \end{aligned}$$

One can use the Floyd-Warshall algorithm [Floyd, 1962, Warshall, 1962] [Cormen et al., 2009, Section 25.2], which runs in $O(|X'|^3)$, to transform a DBM into its canonical normal form. We always assume that every DBM is in canonical normal form (i.e., we always execute the canonization algorithm whenever the normal form property is violated).

The *conjunction* of two DBMs M and N is obtained by taking the conjunction of the individual inequalities:
$M \wedge N := M'$, where, for each $x, y \in X'$,

$$\big(M'(y,x), \prec^{M'}_{(y,x)}\big) = \begin{cases} \big(M(y,x), \prec^M_{(y,x)}\big) & \text{if } M(y,x) < N(y,x); \\ \big(N(y,x), \prec^N_{(y,x)}\big) & \text{if } N(y,x) < M(y,x); \\ \big(M(y,x), \prec^M_{(y,x)}\big) & \text{if } M(y,x) = N(y,x) \\ & \quad \text{and } \prec^M_{(y,x)} = \prec^N_{(y,x)}; \\ \big(M(y,x), <\big) & \text{if } M(y,x) = N(y,x) \\ & \quad \text{and } \prec^M_{(y,x)} \neq \prec^N_{(y,x)}. \end{cases}$$

The *future* operation for a DBM M widens M such that, after the widening, it contains all states that are reachable by letting time elapse:

$M^{\Uparrow} := M'$, where, for each $x, y \in X'$,

$$\left(M'(y,x), \prec^{M'}_{(y,x)}\right) = \begin{cases} \left(\infty, <\right) & \text{if } y \neq x_0 \text{ and } x = x_0; \\ \left(M(y,x), \prec^{M}_{(y,x)}\right) & \text{otherwise.} \end{cases}$$

The *reset* operation for a DBM M resets the clocks $r \subseteq X$ in M: $M[r := 0] := M'$, where, for each $x, y \in X'$,

$$\left(M'(y,x), \prec^{M'}_{(y,x)}\right) = \begin{cases} \left(0, \leq\right) & \text{if } y \in r \text{ and } x \in r; \\ \left(M(x_0,x), \prec^{M}_{(x_0,x)}\right) & \text{if } y \in r \text{ and } x \notin r; \\ \left(M(y,x_0), \prec^{M}_{(y,x_0)}\right) & \text{if } y \notin r \text{ and } x \in r; \\ \left(M(y,x), \prec^{M}_{(y,x)}\right) & \text{if } y \notin r \text{ and } x \notin r. \end{cases}$$

The *existential quantification* (or *inverse reset*) operation for a DBM M existentially quantifies over the clocks in $r \subseteq X \setminus \{x_0\}$: $\exists r : M := M'$, where, for each $x, y \in X'$,

$$\left(M'(y,x), \prec^{M'}_{(y,x)}\right) = \begin{cases} \left(\infty, <\right) & \text{if } y \in r \text{ and } x \in r; \\ \left(\infty, <\right) & \text{if } y \in r \text{ and } x \notin r; \\ \left(M(y,x_0), \prec^{M}_{(y,x_0)}\right) & \text{if } y \notin r \text{ and } x \in r; \\ \left(M(y,x), \prec^{M}_{(y,x)}\right) & \text{if } y \notin r \text{ and } x \notin r. \end{cases}$$

The result of a *negation* or a *disjunction* operation applied on DBMs might not be convex anymore, and thus, cannot be represented as a single DBM. That is why we introduce *clock federations* (or just *federations*) to represent sets of DBMs. The set of all clock federations is $\mathcal{F} = 2^{\mathcal{Z}}$. The convex operations on DBMs introduced above can be extended to federations in a straightforward way. As all the operations are distributive with respect to disjunction, for a federation $f \in \mathcal{F}$, it suffices to perform a particular operation on f just by applying the operation on the subsumed DBMs in f.

Semi-symbolic reachability analysis. As we assume a coarse abstraction of the exponential control structure of $\mathcal{N}$, we employ the *semi-symbolic* representation, where the set of reachable states R is represented as an explicit mapping of locations to clock federations. That is, R is a mapping $R : Q \to \mathcal{F}$. We note that standard timed model checking tools such as KRONOS [Yovine, 1997] or UPPAAL [Bengtsson et al., 1995, Behrmann et al., 2004] are implemented based on this representation.[4]

For a federation f and an edge $e = (q, d, \varphi, r, q')$, the *strongest timed postcondition of f after e* is defined as

$$\mathsf{Succ}_t(f, e) := \bigvee_{z \in f} (z \wedge \varphi)[r := 0]^{\Uparrow} \wedge I(q').$$

[4] To the best of the author's knowledge, without a symbolic treatment of the discrete state information.

For a given (abstract) timed automaton, Algorithm 2 computes the least fixed point of Succ_t to obtain the set of *forward* reachable states based on a semi-symbolic representation. In order to ensure termination, we implicitly apply *maximal constant widening* [Behrmann et al., 2006] on the result of Succ_t. In our case, such a widening exists as we only allow rectangular clock constraints in the definition of our timed automata [Bouyer, 2003].

Algorithm 2 $\mathsf{ReachForward}(\mathcal{T})$: For a given timed automaton $\mathcal{T} = (Q, q_0, \Sigma, E, I)$, the algorithm returns the *forward* reachable states of $(q_0, \vec{0})$ based on a semi-symbolic representation.

```
1: R := [ q_0 ↦ 0⃗⇑ ]
2: Q := ∅
3: push(Q, q_0)
4: while Q ≠ ∅ do
5:    q := pop(Q)
6:    for all e ∈ E with source location q do
7:       f := Succ_t(R[q], e)
8:       if f ⇏ R[q'] then
9:          R[q'] := R[q'] ∨ f
10:         push(Q, q')
11:      end if
12:   end for
13: end while
14: return R
```

The *weakest timed precondition of f before e* is defined as

$$\mathsf{Pred}_t(f, e) := \bigvee_{z \in f} \big(\exists r : (z \wedge (r = 0) \wedge \varphi)\big)^{\Downarrow}.$$

Algorithm 3 computes the least fixed point of Pred_t to obtain the set of *backward* reachable states based on a semi-symbolic representation.

9.2.5 Local Refinement

Refining our syntactic abstractions corresponds to selecting a set of locations and split the partitions in Π according to this set. Clearly, any refinement that increases Π would ensure progress and would ultimately lead to the most precise partitioning whose elements are the singleton sets of the concrete locations of $\mathcal{N}$. However, in this section, we propose a refinement heuristic that is *guided by the current approximations* of the reachable states, which has proven effective in practice.

For a partitioning Π, an edge relation of a may quotient E_{may}, an overapproximation of the forward reachable states R_f, and an underapproximation

Algorithm 3 $\mathsf{ReachBackward}(\mathcal{T}, q_b)$: For a given timed automaton $\mathcal{T} = (Q, q_0, \Sigma, E, I)$ and a location $q_b \in Q$, the algorithm returns the *backward* reachable states of $(q_b, \mathbf{true})$ based on a semi-symbolic representation.

```
1:  R := [ q_b ↦ true ]
2:  Q := ∅
3:  push(Q, q_b)
4:  while Q ≠ ∅ do
5:    q' := pop(Q)
6:    for all e ∈ E with target location q' do
7:      f := Pred_t(R[q'])
8:      if f ⇏ R[q] then
9:        R[q] := R[q] ∨ f
10:       push(Q, q)
11:     end if
12:   end for
13: end while
14: return R
```

of the backward reachable states R_b, where $R = R_f \wedge R_b \not\equiv \mathbf{false}$, we define the *refinement of* Π as the function $\mathsf{Refine}(\Pi, \mathcal{T}_{may}, \mathcal{T}_{must}, R_f, R_b)$.

We pick some $e = (\pi, d, \varphi, r, \pi')$ from E_{may} such that

$$\big(\mathsf{Pred}_t(R[\pi'], e) \wedge R_f[\pi]\big) \not\equiv \mathbf{false}$$

and

$$\mathsf{Pred}_t(R[\pi'], e) \not\Rightarrow R_b[\pi].$$

Then, the refinement is defined as

$$\mathsf{Refine}(\Pi, \mathcal{T}_{may}, \mathcal{T}_{must}, R_f, R_b) := [\![\pi]\!] \wedge \mathsf{Pred}_b([\![\pi']\!]', e).$$

Note that this kind of refinement can be seen as a precise backward propagation of the bad states that is guided by an overapproximated forward analysis. This resembles the on-the-fly algorithm proposed by Cassez et al. [2005] for computing winning strategies in timed games. However, the crucial improvement of the approach presented here is that both locations and clock values are represented symbolically.

9.2.6 Abstraction Refinement

The core part of our combination approach is the abstraction refinement loop shown as Algorithm 4.

In Lines 1–3, Reachable first initializes the partitioning Π and constructs the initial abstractions $\mathcal{T}_{may}$ and $\mathcal{T}_{must}$ with respect to Π. Then, in the

Algorithm 4 Reachable$(\mathcal{N}, B)$: The algorithm decides whether a set of timed states B is reachable in a network of timed automata $\mathcal{N}$, where $\vec{q}_0$ is the global initial location of $\mathcal{N}$.

```
1:  Π := Π_0^N
2:  T_may := ⌈N/Π⌉
3:  T_must := ⌊N/Π⌋
4:  while true do
5:    R_b := ReachBackward(T_must, q_b)
6:    if (q_0, 0) ∈ R_b then
7:      return true
8:    end if
9:    R_f := ReachForward(T_may)
10:   if R_f ∧ R_b ≡ false then
11:     return false
12:   end if
13:   Π := Refine(Π, T_may, T_must, R_f, R_b)
14: end while
```

actual loop, in Line 5, Reachable computes an underapproximation of the backward reachable states R_b by running Algorithm 3 on $\mathcal{T}_{must}$. At this point, in Line 6, we check whether the initial state of $\mathcal{N}$ is already contained in the underapproximation. If this is the case, we terminate the loop and report that some of the bad states are reachable. Otherwise, we continue and, in Line 9, we compute an overapproximation of the forward reachable states R_f by running Algorithm 2 on $\mathcal{T}_{may}$. During this analysis, if no state from R_b is visited, we terminate the loop and report that no bad state is reachable. If neither reachability nor unreachability can be established, we refine Π and continue the loop.

Note that, instead of underapproximating the backward reachable states while using an overapproximation of the forward reachable states for guiding the refinement, Algorithm 4 can also be dualized such that the forward reachable states are underapproximated while an overapproximation of the backward reachable states is used to guide the forward refinement.

9.2.7 Optimizations

Precomputational discrete analysis. The symbolic treatment of the locations allows us to apply computationally cheap but effective optimizations based on pure discrete analyses of the control structure of $\mathcal{N}$. For instance, we can restrict the locations in the partitioning Π in Algorithm 4 only to those locations which are both forward reachable from the initial locations and backward reachable from the bad locations.

Also, before constructing the initial abstraction, we can enlarge the set

of bad locations by those locations from which a bad state is discretely reachable. More precisely, each edge $e = (q, d, \varphi, r, q')$, for which

$$\mathsf{Pred}_t(\mathbf{true}, e) \equiv \mathbf{true},$$

can be used to identify additional bad locations. Figure 9.2 shows an example.

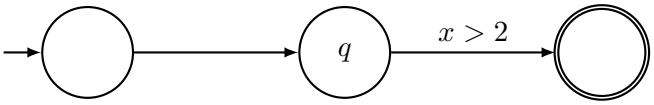


Figure 9.2: Location q can be added to the bad locations.

Reusing approximations. Whenever a timed state appears in an underapproximation of the reachable states, then it is certainly contained in the precisely reachable states. During the refinement, when splitting a partition π into finer partitions π_1 and π_2, we can exploit this fact and let π_1 and π_2 inherit the underapproximation of the reachable states for π. Also, we can reuse the underapproximation of all other locations of the abstraction.

For overapproximations, however, one has to be more careful. When a refinement changes the structure of the abstractions, we can only reuse the overapproximation for those locations that are not reachable from those locations affected by the refinement. The overapproximations for the locations reachable from the changed locations need to be discarded.

Lazy constraints. In case that $\mathcal{N}$ exhibits exponentially many distinct clock guards (see Section 10.1.1 on Page 137 for such a scenario), in order to avoid the up-front explosion in the construction of $\mathcal{T}_{may}$, one can approximate the guards of its edges. Recall from Section 9.2.2 that the global edge relation Δ is defined as a CNF of polynomial size. When constructing $\mathcal{T}_{may}$, for connecting two abstract locations π_1 and π_2, one first computes

$$[\![e]\!] \equiv \Delta \wedge [\![\pi_1]\!] \wedge [\![\pi_2]\!]'.$$

Now, for some clock resets r, instead of computing the precise guard

$$\varphi \equiv \bigvee_{i=1}^{k} \varphi_i$$

by enumerating all k combinations of clock guards φ_i, for which

$$\varphi_i \not\equiv \mathbf{false} \qquad \text{and} \qquad \left([\![e]\!] \wedge [\![\varphi_i]\!]^c \wedge [\![r]\!]^r\right) \not\equiv \mathbf{false},$$

one can obtain a structurally simpler but approximate guard $\varphi' \Leftarrow \varphi$ just by omitting some clauses in $[\![e]\!]$ interpreted as CNF.

9.3 Bibliographic Remarks

Symbolic data structures. The efficiency of an analysis algorithm for timed automata strongly depends on the way in which the state space is represented. In the last two decades, several techniques to represent state spaces that consist of a discrete and a continuous part were proposed. These techniques can broadly be classified into two categories: *semi-symbolic* and *fully symbolic* approaches [Henzinger et al., 1994, Seshia and Bryant, 2003].

A semi-symbolic state space representation completely focuses on efficiently representing and manipulating sets of clock values, while leaving the discrete state information explicit. KRONOS [Yovine, 1997] and UPPAAL [Bengtsson et al., 1995, Behrmann et al., 2004] are the oldest and most prominent representatives of this approach that use difference bound matrices [Dill, 1989] to implement (a heuristic variant of) the zone-based reachability algorithm [Henzinger et al., 1994, Alur, 1999]. Instead of a matrix-based representation of clock zones, Larsen et al. [1997] investigated the use of weighted directed graphs to store only a minimal set of constraints.

A first approach towards a diagram-based representation that also incorporates discrete state information was given by Asarin et al. [1997], who used BDDs to encode sets of clock valuations as *numerical decision diagrams* using a discretization scheme based on region equivalence. Similarly, Bozga et al. [1997] approximated the precise clock values to discrete time steps, resulting in a pure discrete semantics allowing a state space representation using a single BDD. In the same spirit, based on *closed timed automata*, a restricted form of classical timed automata where only nonstrict clock constraints are allowed, Beyer [2001] introduced an integer semantics where clock values and location information can be represented jointly in a single BDD. The latter approach was implemented in the RABBIT tool [Beyer et al., 2003]. Besides the fact that the performance of such pure BDD-based approaches is very sensitive to the magnitude of the clocks, it has been observed that the BDDs can blow-up significantly due to interdependencies in the timing behavior of the system.

Seshia and Bryant [2003] solved the TCTL model checking problem by representing sets of states by difference logic formulas which, in turn, are represented as BDDs using a binary encoding. The clock differences that need to be tracked in the fixed-point computation are encoded in so-called transitivity constraints, which are added on-the-fly during the model checking process. Even though they added some specialized optimizations for this case, the experimental results are inconclusive. Møller et al. [1999] introduced *difference decision diagrams*, a BDD-like data structure in which each diagram node is labeled with a difference constraint. Here, the Boolean constraints, represented as special differences, are interleaved with the clock constraints in the diagram structure. Larsen et al. [1999], Behrmann et al. [1999b] proposed *clock difference diagrams* (CDDs), a more space-efficient

data structure, which benefits from sharing clock constraints for several clock zones. CDDs store intervals of clock valuations in a BDD-like structure as a rooted, directed, and acyclic graph. As a further extension, Wang [2004] proposed *clock restriction diagrams* (CRDs), in which the disjointness requirement is dropped. In contrast to CDDs, CRDs only store upper bounds of clock differences. Location information is added to CRDs by adding Boolean variable nodes. Recently, Morbé et al. [2011] proposed *and-inverter graphs with linear constraints* as a diagram-based data structure to perform a fully symbolic reachability analysis of timed automata.

Yamane and Nakamura [2004] combined DBMs with BDDs for implementing an approximation technique proposed by Dill and Wong-Toi [1995]. More recently, Ehlers et al. [2010b] introduced a model checking approach based on *clock zone maps*, where clock zones, represented as DBMs, are mapped onto sets of locations, represented as BDDs. As a continuation of the latter approach, Ehlers et al. [2010a] introduced *constraint matrix diagrams*, a diagram-based data structure that generalizes CDDs, CRDs, and DBMs.

Abstraction refinement. Möller et al. [2002] investigated the theoretical foundations of applying predicate abstraction [Graf and Saïdi, 1997] for model checking dense real-time systems against μ-calculus specifications. The basic principle of counterexample-guided abstraction refinement (the so-called CEGAR approach) is due to Clarke et al. [2003]. From a more applied perspective, Dierks et al. [2007] proposed an abstraction refinement technique for *PLC automata*, a subclass of timed automata. Peter and Mattmüller [2009] presented a component-based abstraction refinement approach for timed controller synthesis (where reachability checking is a special case), which can be seen as a timed extension of the compositional analysis approach for pure discrete systems by Behrmann et al. [1999a]. The combination approach based on abstraction refinement, which is presented in this chapter, was introduced by Ehlers et al. [2010c] for timed games and implemented in the tool Synthia [Peter et al., 2011]. The idea of using fixed point approximations to guide a refinement was independently developed by Ganty et al. [2010] for purely discrete alternating automata.

Chapter 10

Experimental Evaluation

In this chapter, we report on the tool SYNTHIA[1] and an experimental evaluation of the techniques presented in the previous two chapters.

10.1 Efficiency Considerations

Before we come to the actual experimental evaluation, we first discuss more generally in which cases the combination approach from the last chapter pays off and where it does not. We fix a given network of timed automata $\mathcal{N}$.

10.1.1 Bad Cases

The number of edges in $\mathcal{T}_{may}$, and therefore also the efficiency of the whole approach, depends on the number of distinct clock guard/reset pairs k (recall from Section 9.1 that k is exponential in the bandwidth of the communication interface between the control and the timed part). Clearly, k is bounded by the number of synchronized discrete steps the components in $\mathcal{N}$ can jointly take. If $\mathcal{N}$ does not have any synchronization at all, k is bounded by the number of edges of all components (i.e., k is polynomial). If, on the other hand, all components share some common synchronization events, and the components' control structures are discretely nondeterministic, k can be exponential in the number of the components. Figure 10.1 depicts such a situation.

However, for realistic examples, a more natural assumption is that the number of possible nondeterministic synchronizations in $\mathcal{N}$ is bounded (e.g., if at most a constant number of components synchronize on a shared decision).

[1] SYNTHIA has been published in [Peter et al., 2011].

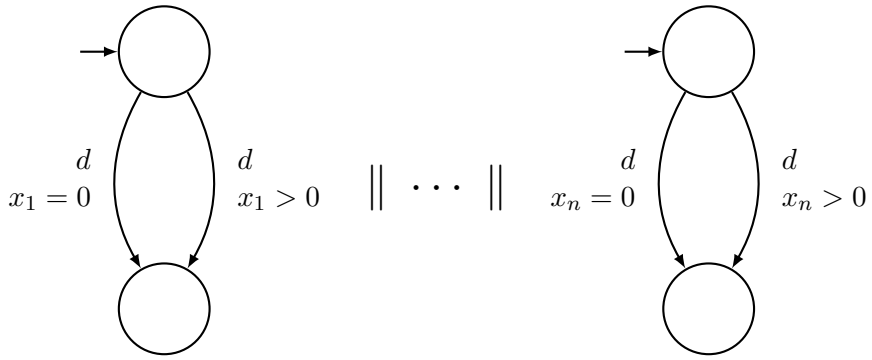


Figure 10.1: A network of timed automata with an exponential number of distinct clock guard/reset pairs.

10.1.2 Good Cases

Our combination approach based on abstraction refinement greatly unfolds its potential whenever we can identify irrelevant parts of the control structure of $\mathcal{N}$ in an early (and therefore coarse) abstraction. Such a situation is shown in Figure 10.2.

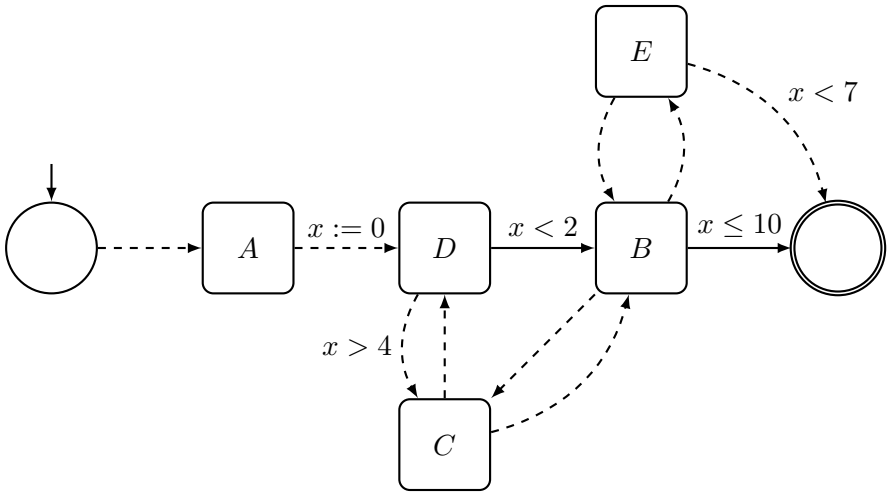


Figure 10.2: An abstract timed automaton with may and must edges, drawn as dashed and solid lines, respectively. The abstract locations C and E can be completely ignored, as both can be identified as not contributing to the backward reachable states at D.

In this example, the underapproximation of the backward reachable states for location B is $x \leq 10$, and for D it is $x < 2$. The only incom-

ing edge to D is from location A and resets x. Hence, the successor states after executing that edge are always subsumed by the underapproximation at D. Consequently, no further refinement beyond D is necessary anymore as we can conclude that location C does not contribute to the forward reachability of some bad state. Observe that a pure propagation-based approach that just incrementally computes the set of forward or backward reachable states would not be capable of identifying C and E as being redundant.

10.2 The Tool Synthia

We now present the tool Synthia, which we will use as a basis for our experimental evaluation.

Synthia is the first certifying model checker for open real-time systems modeled as networks of timed automata. The key innovation of Synthia is its ability to justify why a given system is correct by providing a correctness certificate to the user. Such certificates are easy-to-validate deductive proofs that only reflect the specification-critical properties of the system. Synthia can also handle partially implemented systems, in which case it certifies their realizability by synthesizing reference implementations for the unimplemented parts.

Synthia's core algorithm is the abstraction refinement approach presented in Chapter 9 that combines binary decision diagrams with difference bound matrices. The synthesized correctness certificates are the final syntactic abstractions exported as timed automata. As an extension, the template-based synthesis approach from Chapter 8 is available as a specialized refinement strategy.

10.2.1 Availability and Usage

Synthia is licensed under the GNU General Public License and available for download at

http://react.cs.uni-saarland.de/tools/synthia.

Providing a comprehensive reference manual for Synthia is out of the scope of this thesis. Instead, some standard usage scenarios are presented. A detailed description of the command line parameters, the file format, as well as a step-by-step tutorial can be found on the tool's website.

A specification is given in form of an XML file and essentially contains a plant model with requirements. Assuming that `fischer.xml` represents a specification, then the simplest way to execute Synthia is the following:

```
$ synthia  fischer.xml
```

This lets SYNTHIA check whether the given model satisfies its requirements, both specified in `fischer.xml`. Specifications can have parameters with default values which can be overridden using the `-D` command line argument:

```
-Dprocesses:2  -Ddelay:23  -Dtimeout:42
```

Requirements are given as conjunctions of assumptions and guarantees. A system does not satisfy its requirements if (1) there is a trace that eventually violates the guarantees, and (2) each prefix of that trace satisfies the assumptions. For example, the following lines of an XML specification file encode a requirement describing a location invariant and a bounded reachability guarantee:

```
<assume>
  in(loc) imply (x <= {delay})
</assume>

<guarantee>
  (not in(goal)) imply (y <= {timeout})
</guarantee>
```

Additionally to its model checking capabilities, SYNTHIA can also be used as a synthesis tool to generate controllers for open plants modeled as timed game automata. For example, to let SYNTHIA synthesize a controller for a partially specified plant given in `fischer.xml`, the following command line parameters can be used:

```
$ synthia  robot.xml
      --synth-cont controller.xml

$ synthia  robot.xml
      --synth-cont-plant controlled_plant.xml
```

The former call generates a model (in the SYNTHIA file format) that only comprises the controller, while the latter generates a model where the synthesized controller is embedded into the original plant.

10.2.2 Implementation Details

SYNTHIA is written in C^{++} and uses, besides some standard BOOST libraries, the CUDD BDD library [Somenzi, 2009] for representing transition relations and sets of locations, as well as the UPPAAL DBM library [David, 2011] for representing and manipulating clock zones / federations.

After parsing the specification, as explained in Section 9.2 on Page 125, SYNTHIA constructs a BDD-based representation of the control structure and sets up the initial abstraction. SYNTHIA's main analysis procedure

essentially bases on Algorithm 4 with the optimizations proposed in Section 9.2.7. In case a controller is to be generated for an open plant, instead of computing the backward reachable states, SYNTHIA computes the set of winning states (the so-called *attractor set* [Grädel et al., 2002]) of the reachability player (representing the hostile environment that tries to violate the guarantees) and deduces a safety controller.

10.3 Model Checking

This section presents the results of an experimental evaluation of SYNTHIA performing model checking. We compare SYNTHIA's performance with the model checker UPPAAL [Behrmann et al., 2004].

10.3.1 Benchmarks

To assess the practical efficiency of the combination approach from Chapter 9, we have chosen standard benchmarks from the literature that exhibit both timed *and* concurrent behavior as two independent sources of succinctness. While *Fischer's mutual exclusion protocol* and the *carrier sense, multiple access with collision detection protocol* are such benchmarks consisting of many concurrent processes running asynchronously, we also consider the *gear production stack* as a benchmark that is more representative for sequential real-time systems.

Fischer's mutual exclusion protocol. This is a standard benchmark (Fischer) from the real-time verification community devised by Michael Fischer [see Abadi and Lamport, 1991, for a detailed description]. It models a distributed mutual exclusion protocol, in which asynchronous processes access a common resource. Each process has a unique nonzero number. The processes communicate via a single shared variable id, which ranges over the various process numbers and 0. Before a particular process p accesses the resource, it checks whether $id = 0$. If this is the case, p nondeterministically waits at most D time units before it sets id to its own number j. If, after T more time units, id still equals j, p accesses the resource. When p releases the resource again, it sets id to zero.

The processes are parametrized in D and T. The size of a benchmark instance is measured in the number of processes. We check the safety property, whether it is possible that any two processes access the resource at the same time.

Carrier sense, multiple access with collision detection protocol. This benchmark (CSMA/CD) stems from a case study, where a communication protocol for a distributed network of stations communicating over

a shared bus is modeled. Whenever a particular station has data to sent and the bus is idle (i.e., no other station is transmitting), the station begins sending its message. If, on the other hand, the bus is busy, the station waits a nondeterministic amount of time until it retries sending a message. In case a collision occurs (because several stations transmit simultaneously), all transmissions are aborted and, after a nondeterministic amount of time, each station tries to transmit its message again.

The original model on which this benchmark is based was given by Yovine [1997]. In our evaluation, we use the version due to Möller [2001]. The size of a benchmark instance is measured in the number of stations. We check the safety property, whether a particular station A detects a collision timely when another station B is simultaneously transmitting.

Gear production stack. This case study (GPS) represents a manufacturing plant that consists of communicating processing stations for workpieces [Finkbeiner et al., 2008]. Whenever a workpiece is loaded into the plant, it gets processed by each station in a sequential manner. In principle, the stations, where each is specialized in a certain treatment of the workpiece, run asynchronously. However, they synchronize with their predecessor station, whenever they receive the workpiece.

The size of a benchmark instance is measured in the number of stations. We check the bounded liveness property whether workpieces are always processed within a certain amount of time.

10.3.2 Results

We now present the actual experimental results of the comparison of the performance of Synthia version 1.2.1 against Uppaal version 4.1.4 running on the benchmarks described in the previous section. Synthia was compiled using GCC version 4.4.3. We executed Uppaal with various combinations of command line options and have chosen the best one for the comparison: For all safe instances (where the bad state is unreachable), we either used no command line options (default mode), '`-C`' (disable most memory reduction techniques), '`-S2`' (optimize space consumption), or '`-C -S2`'. For all unsafe instances (where the bad state is reachable), we additionally executed Uppaal with the '`-o1`' command line option (perform a depth-first search).

Running times are given in seconds and the memory consumptions are given in MB. The time limit was set to 2 hours, and the memory limit was set to 4 GB. All experiments were conducted on a 2.6 GHz AMD Opteron computer running Ubuntu 10.04.

A direct comparison of the running times of Synthia and Uppaal running on Fischer with safe timing parameters is shown in Figure 10.3. As one can see, Synthia clearly outperforms Uppaal by several orders of magnitude. The reason for this tremendous gain of efficiency is the effectiveness

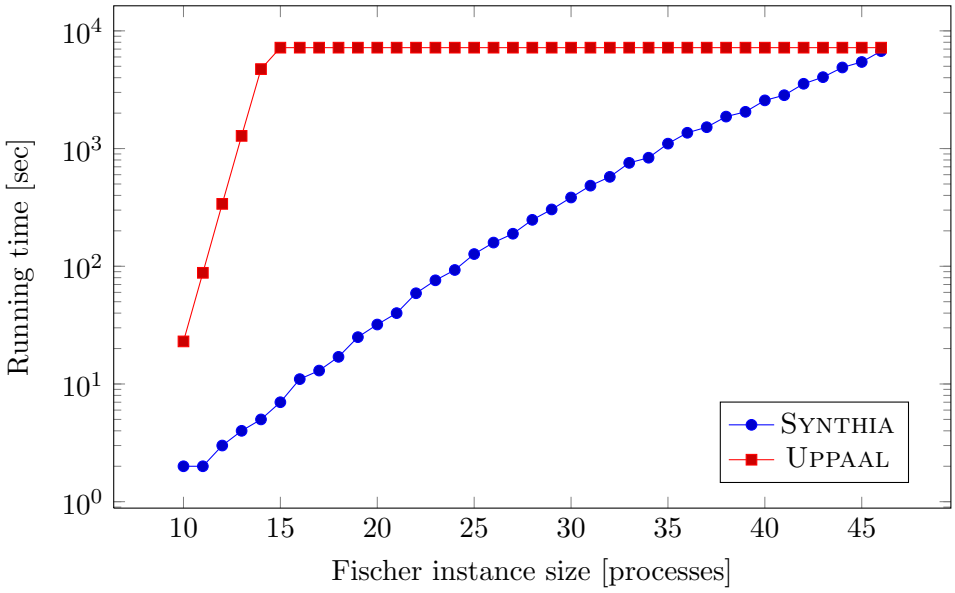


Figure 10.3: Comparison of the running times of Synthia and Uppaal running on Fischer with safe timing parameters.

of Synthia's abstraction refinement algorithm. As shown in Figure 10.4, Synthia produces abstractions of *only quadratic size*, which are already sufficient to establish the unreachability of the bad state.

Also interesting is that there is only a moderate increase in the memory consumption of Synthia for larger instances of Fischer, as shown in Figure 10.5. The small oscillation effect (i.e., the fact that some smaller instances have a higher memory consumption than some larger ones) is caused by the variable reordering and caching heuristics of the CUDD library. This BDD-related phenomenon is also observable in other contexts [see, e.g., Bloem et al., 2007].

The picture changes when we look at those instances of Fischer, where the bad state is reachable (by modeling the last process with an unsafe timing behavior). In this case, thanks to its capability to perform a depth-first search (manually enabled using the `-o1` option), Uppaal is able to detect the error in the model much faster than Synthia's abstraction refinement algorithm. However, sometimes (e.g., for 40 or especially 70 processes) Uppaal's internal search heuristics fail in finding the bad state quickly.

Table 10.1 shows a detailed summary of the experimental results for Fischer. From left to right, the columns show the size of the instance in terms of processes, whether it was an unsafe or safe instance, the number of refinement steps Synthia needed to produce the final abstraction, whose size in terms of locations is shown in the next column, the running time and

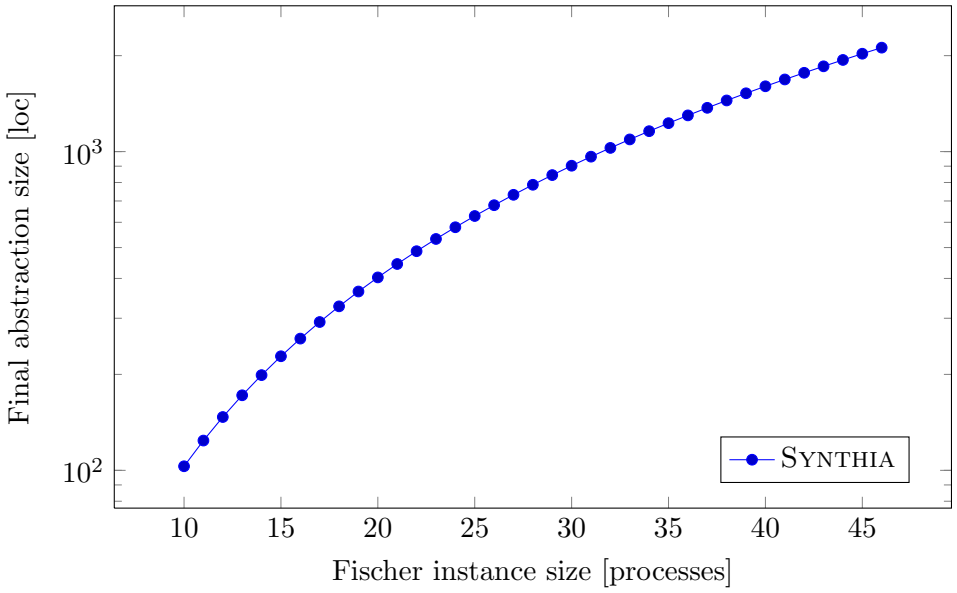


Figure 10.4: Size of the final abstraction generated by Synthia for Fischer with safe timing parameters.

memory consumption of Synthia, the command line arguments for which Uppaal showed the best results, the number of states Uppaal explored, and the running time and memory consumption of Uppaal.

Figure 10.6 shows the comparison of the running times of Synthia and Uppaal on CSMA/CD. While both tools suffer from an exponential blow-up in the number of stations, the increase in Synthia's running time is far less dramatic than the increase for Uppaal. The reason for this gain of efficiency is, again, due to the effectiveness of the abstraction refinement algorithm: As shown in Table 10.2 (which is analogously structured as Table 10.1), after only 4 refinement steps, Synthia is able to find a sufficiently precise abstraction with only 5 locations, *independent of the size of the instance*. The blow-up that still occurs is due to the exponential number of distinct guard/reset pairs.

The results for GPS are shown in Table 10.3 (which is analogously structured as Table 10.1). In the unsafe instances, we choose a too short time bound, in which the workpieces must be processed by the stations. Similar to Fischer, in these cases, when the bad state is reachable, manually setting Uppaal into depth-first search mode yields the best results. However, for the safe instances, already in the (optimized) initial abstraction, Synthia is able to detect the unreachability of the bad state, allowing to analyze instances with more than 300 stations.

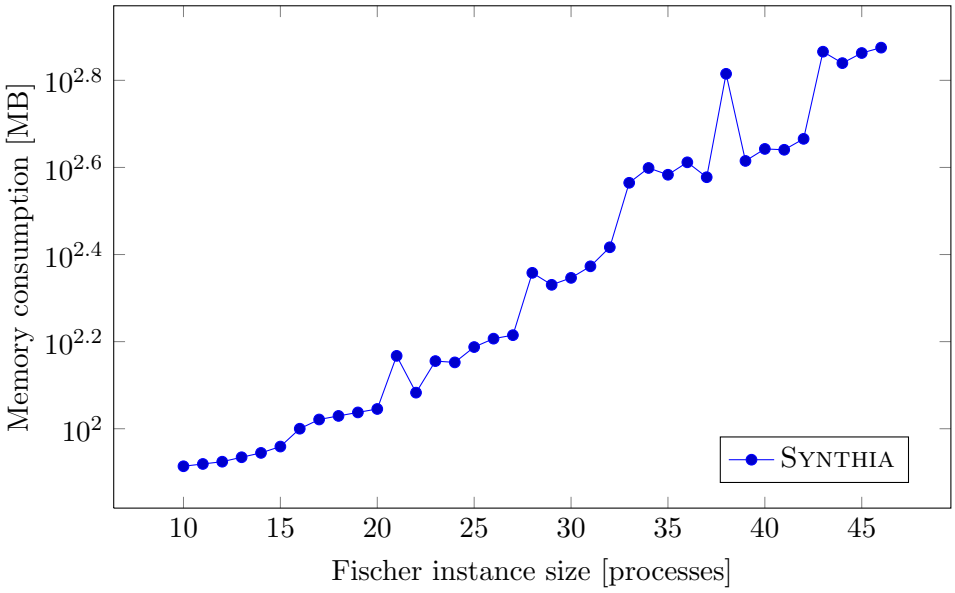


Figure 10.5: Memory consumption of Synthia running on Fischer with safe timing parameters.

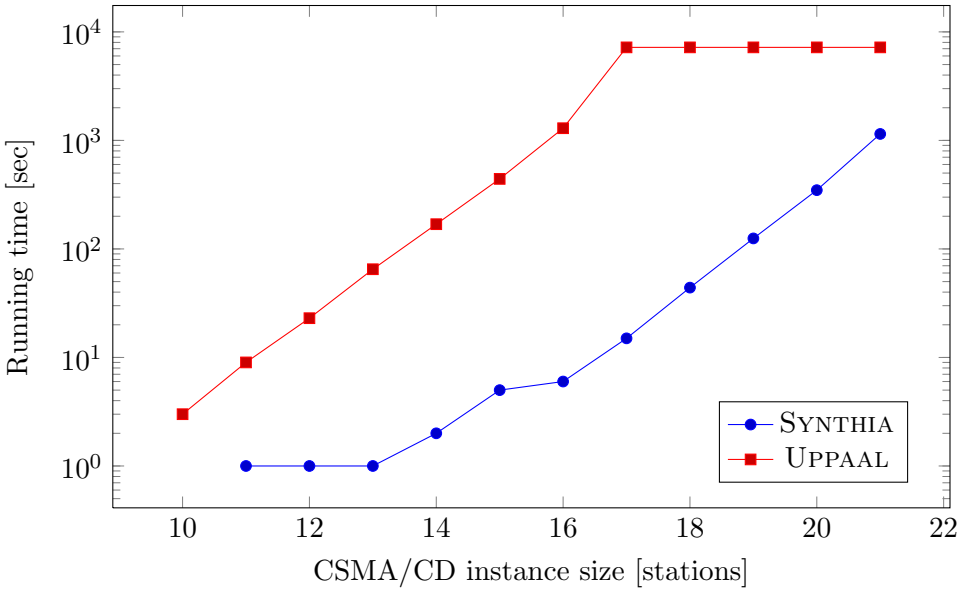


Figure 10.6: Running time comparison between Synthia and Uppaal for the CSMA/CD protocol.

Instance		Synthia				Uppaal			
Size	Safe	Steps	Abs	Time	Mem	Mode	States	Time	Mem
15	No	16	19	3	82	-o1	166	1	24
20	No	28	31	5	100	-o1	8	0	6
25	No	38	41	14	109	-o1	8	1	6
30	No	50	53	41	215	-o1	1648	1	28
35	No	60	63	91	218	-o1	239	1	28
40	No	72	75	181	363	-o1	60803	137	194
45	No	82	85	255	390	-o1	209	2	31
50	No	94	97	1247	650	-o1	224	1	33
55	No	104	107	1263	744	-o1	36	1	33
60	No	116	119	2563	1286	-o1	334	4	38
65	No	126	129	5029	1286	-o1	180	3	39
69	No	TIMEOUT				-o1	690	7	46
70	No	TIMEOUT				MEMOUT			
71	No	TIMEOUT				-o1	735	11	45
72	No	TIMEOUT				-o1	7	1	40
10	Yes	100	103	2	82	-C -S2	836128	23	51
11	Yes	121	124	2	83	-C -S2	2752774	88	108
12	Yes	144	147	3	84	-C -S2	8985344	339	351
13	Yes	169	172	4	86	-C -S2	29122758	1282	1127
14	Yes	196	199	5	88	-C -S2	93835680	4732	3501
15	Yes	225	228	7	91	MEMOUT			
16	Yes	256	259	11	100	MEMOUT			
17	Yes	289	292	13	105	MEMOUT			
18	Yes	324	327	17	107	MEMOUT			
19	Yes	361	364	25	109	MEMOUT			
20	Yes	400	403	32	111	MEMOUT			
25	Yes	625	628	127	154	MEMOUT			
30	Yes	900	903	384	222	MEMOUT			
35	Yes	1225	1228	1101	383	MEMOUT			
45	Yes	2025	2028	5440	729	MEMOUT			
46	Yes	2116	2119	6732	750	MEMOUT			
47	Yes	TIMEOUT				MEMOUT			

Table 10.1: Overview on the experimental results of Synthia and Uppaal running on Fischer.

Instance		Synthia				Uppaal			
Size	Safe	Steps	Abs	Time	Mem	Mode	States	Time	Mem
10	Yes	4	5	0	21	-C	123140	3	30
11	Yes	4	5	1	81	-C	316420	9	73
12	Yes	4	5	1	81	-C -S2	797700	23	164
13	Yes	4	5	1	84	-C	1978372	65	424
14	Yes	4	5	2	94	-C	4837380	169	1052
15	Yes	4	5	5	115	-C -S2	11681796	442	2639
16	Yes	4	5	6	156		27901956	1295	3073
17	Yes	4	5	15	247	MEMOUT			
18	Yes	4	5	44	438	MEMOUT			
19	Yes	4	5	125	870	MEMOUT			
20	Yes	4	5	348	1784	MEMOUT			
21	Yes	4	5	1147	3781	MEMOUT			
22	Yes	MEMOUT				MEMOUT			

Table 10.2: Overview on the experimental results of Synthia and Uppaal running on CSMA/CD.

Instance		Synthia				Uppaal			
Size	Safe	Steps	Abs	Time	Mem	Mode	States	Time	Mem
25	No	52	29	2	82	-o1	49	1	6
30	No	62	34	4	84	-o1	59	0	6
35	No	72	39	10	86	-o1	69	1	6
40	No	82	44	19	98	-o1	79	0	6
45	No	92	49	35	107	-o1	89	1	6
50	No	102	54	62	103	-o1	99	0	6
75	No	152	79	600	145	-o1	149	1	6
100	No	202	104	3072	225	-o1	199	0	6
125	No	TIMEOUT				-o1	249	1	6
150	No	TIMEOUT				-o1	299	2	39
13	Yes	0	3	1	53		69632	2	27
14	Yes	0	3	1	53	-C	147456	4	31
15	Yes	0	3	1	53	-C	311296	9	57
16	Yes	0	3	1	53	-C	655360	21	92
17	Yes	0	3	1	53	-C -S2	1376274	51	193
18	Yes	0	3	1	53	-C -S2	2883603	118	401
19	Yes	0	3	1	53	-C	6029312	275	848
20	Yes	0	3	1	21	-C -S2	12582933	626	1796
21	Yes	0	3	1	53	-C	26214400	1467	3834
22	Yes	0	3	1	53	MEMOUT			
23	Yes	0	3	1	53	MEMOUT			
24	Yes	0	3	1	53	MEMOUT			
25	Yes	0	3	1	53	MEMOUT			
50	Yes	0	3	4	89	MEMOUT			
75	Yes	0	3	16	118	MEMOUT			
100	Yes	0	3	55	159	MEMOUT			
150	Yes	0	3	282	282	TIMEOUT			
200	Yes	0	3	1033	450	TIMEOUT			
250	Yes	0	3	3178	758	TIMEOUT			
300	Yes	0	3	6751	1058	TIMEOUT			
350	Yes	TIMEOUT				TIMEOUT			

Table 10.3: Overview on the experimental results of Synthia and Uppaal running on GPS.

10.4 Template-based Synthesis

The template-based synthesis algorithm from Chapter 8 (i.e., the Focus Abstraction) is realized as a specialized refinement procedure for SYNTHIA's standard abstraction refinement loop. Whenever an edge for refinement is found, we identify the parameter valuations associated with that edge and split the global abstraction with these valuations. In the subsequent refinement step, we focus (i.e., we restrict the forward exploration) on the identified parameters of the last refinement.

This section presents the results of an experimental evaluation of SYNTHIA performing template-based synthesis. We compare SYNTHIA's performance with the controller synthesis tool UPPAAL-TIGA [Behrmann et al., 2007].

10.4.1 Benchmarks

As synthesis benchmarks, we have chosen standard examples from control theory.

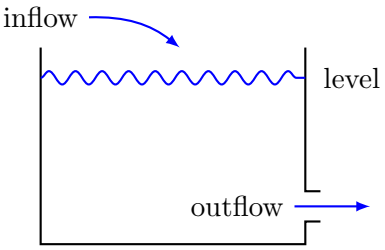
Chinese juggler. In the *Chinese juggler* benchmark [Larsen et al., 2004], a performer needs to stabilize spinning plates to prevent them from falling. After a certain amount of time has passed since a plate was stabilized, it can nondeterministically become unstable. If no restabilization takes place, it ultimately falls down. The plates have different sizes, and hence, different times to become unstable. It takes the performer one time unit to stabilize a certain plate. During that time, he cannot stabilize another plate. The controller synthesis task consists in finding a safe strategy for the performer such that no plate will ever fall down.

The benchmark size is parametrized in the number of plates n. For the template-based synthesis, we use a generic cyclic-executive template (see Section 8.1) with n phases. In each step of the cyclic execution, the controller decides which plate should be stabilized next.

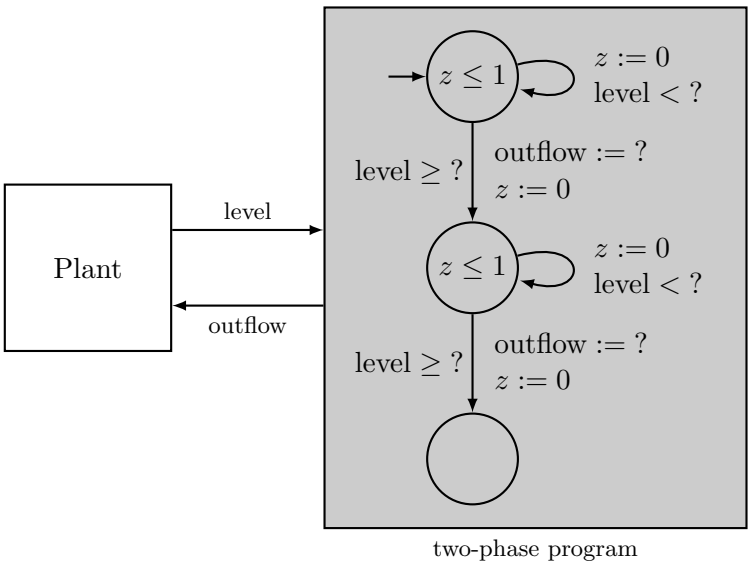
Dam controller. In the *dam controller* benchmark, depicted in Figure 10.7, a controller is to be synthesized that determines the speed of the inflow to a dam. The controller can either stop the inflow or choose between a slow or a fast inflow speed. The bounded reachability requirement is that the fill level should reach a certain value between a minimal and maximal bound. While a fast speed might reach the desired fill level more quickly, the variance of the actual inflow is larger so that the maximal level might be exceeded. On the other hand, being in slow mode, it takes longer to reach the desired fill level, but the variance is not so high so that it is always possible to exactly reach a desired fill level. Thus, being in one mode all the

time is not feasible, since a feasible controller must alternate between fast and slow at least once to fulfill the requirement.

The benchmark size is parametrized in the degree of precision in which the fill level and the inflow amount is digitized. For the template-based synthesis, we use a controller template that models a two-phase program: in the first phase, a certain inflow speed is set until a threshold of the current fill level is passed. Then, the controller enters the second phase with a possibly different speed. The controller stops as soon as a desired fill level is reached. The first and the second speed, as well as the phase-switching threshold are parameters, for which feasible instantiations are to be found.



(a) Principle setting.



(b) Provided template.

Figure 10.7: The Dam Controller benchmark.

10.4.2 Results

We now present the actual experimental results of the comparison of the performance of Synthia version 1.2.1 against Uppaal-Tiga version 4.1.4-0.16 running on the benchmarks described in the previous section. Synthia was compiled using GCC version 4.4.3. We executed Uppaal-Tiga with various command line options and have chosen the best one for the comparison. It turned out that we always obtained the best performance using the default options.

Running times are given in seconds and the memory consumptions are given in MB. The time limit was set to 2 hours, and the memory limit was set to 4 GB. All experiments were conducted on a 2.6 GHz AMD Opteron computer running Ubuntu 10.04.

Instance size	Template-based Synthia				Uppaal-Tiga		
	Steps	Abs	Time	Mem	States	Time	Mem
2	6	19	0	53	57	0	6
3	38	136	0	61	477	0	6
4	110	421	2	87	6755	6	57
5	423	1899	59	247	81292	1095	79
6	1445	8335	1932	1335	TIMEOUT		
7	TIMEOUT				TIMEOUT		

Table 10.4: Overview on the experimental results of Synthia and Uppaal-Tiga running on the Chinese Juggler benchmark.

Instance size	Template-based Synthia				Uppaal-Tiga		
	Steps	Abs	Time	Mem	States	Time	Mem
5	58	100	1	80	88592	2	65
25	268	380	13	87	3114648	307	443
50	530	730	87	105	13545848	5018	2355
75	793	1080	329	111	TIMEOUT		
100	1055	1430	927	143	TIMEOUT		
125	1318	1780	1949	149	TIMEOUT		
150	1580	2130	3483	153	TIMEOUT		
175	1843	2480	5127	213	TIMEOUT		
200	TIMEOUT				TIMEOUT		

Table 10.5: Overview on the experimental results of Synthia and Uppaal-Tiga running on the Dam Controller benchmark.

Tables 10.4 and 10.5 show the results for the Chinese Juggler and the Dam Controller benchmark, respectively. From left to right, the columns

show the size of the instance in terms of plates, the number of refinement steps Synthia needed to produce the final abstraction, whose size in terms of locations is shown in the next column, the running time and memory consumption of Synthia, the number of states Uppaal-Tiga explored, and the running time and memory consumption of Uppaal-Tiga.

For both benchmarks, template-based Synthia clearly outperforms the game-based synthesis techniques implemented in Uppaal-Tiga. A closer look at the Chinese Juggler example reveals that a major source of complexity results from the subtraction operation that occurs in the backwards computation of the winning states. Subtraction is expensive because it does not preserve convexity, and therefore requires a split into multiple zones [Cassez et al., 2005]. The much better performance of template-based synthesis is due to the fact that template-based synthesis is based on model checking, rather than game solving, and model checking does not require such non-convex operations.

In the Dam Controller example, we observe that the size of the abstraction, and, thus, the running time, of the template-based approach increases polynomially in the size of the benchmark, while Uppaal-Tiga suffers from an exponential blow-up in the number of explored states.

Our results thus demonstrate that template-based synthesis is an attractive alternative to the standard game-based approach to timed synthesis. Template-based synthesis has the better worst-case complexity, is easier to implement with symbolic data structures such as DBMs, and produces nicely structured controllers with a small number of locations.

Chapter 11

Conclusion and Outlook

11.1 Conclusion

This thesis introduces sequential circuit machines, a new universal computation model that focuses on succinctness as the central computational resource. As demonstrated in Chapters 5 and 7, many well-known modeling formalisms from the literature exhibit an immediate connection to our new machine model. Once a (syntactic) connection is established, many complexity bounds for structurally restricted sequential circuit machines, proven in Chapter 4, can be uniformly transferred to a specific formalism. As a consequence, besides a drastic unification of independent lines of research, this thesis also provides matching complexity bounds for various analysis problems, whose complexity was not known so far.

Beyond their applicability as a new lower-bound technique, sequential circuit machines per se represent an interesting object worth studying. In Chapter 4, we provide an insightful overview on the computational power of the existential and universal nondeterminism depending on the degree of succinctness granted to them. By seeing explicitness as succinctness over logarithmically many bits, we can characterize all major complexity classes between LogSpace and 2ExpTime in terms of structurally restricted sequential circuit machines.

For timed automata, as a particular important modeling formalism, our complexity-theoretic analysis in Chapter 7 leads to the discovery of tractable fragments of the timed synthesis problem. Specifically, we identify timed controller synthesis based on discrete or template-based controllers to be as complex as model checking. Based on this insight, in Chapter 8, we develop a new model checking-based abstraction refinement algorithm to efficiently find feasible template instantiations.

From a more practical perspective, this thesis also studies the preservation of succinctness in analysis algorithms using symbolic data structures. While efficient techniques exist for specific forms of succinctness considered

in isolation, Chapter 9 presents a general approach to combine off-the-shelf symbolic data structures. Especially in the analysis of timed automata, additionally to the exponential blow-up due to the introduction of clocks, one often also faces a discrete blow-up, e.g., when using networks of timed automata to succinctly model timed systems consisting of concurrent components. Also, due to the exponential number of feasible instantiations, template-based synthesis can cause a similar discrete blow-up. In Chapter 10, we report on an implementation of the combination approach in the tool Synthia. In an experimental evaluation, it turns out that our new approach dramatically outperforms Uppaal and Uppaal-Tiga running on standard model checking and synthesis benchmarks, respectively.

11.2 Outlook

Extending sequential circuit machines. Not in the scope of this thesis, but in principle also possible, is the syntactic characterization of other complexity classes using a slight modification of our machine model. For example, assuming reductions weaker than LogSpace, one could immediately characterize classes from the AC or NC hierarchy [Pippenger, 1979]. One could also prove that, by fixing a constant amount of universal memory, one obtains classes from the polynomial hierarchy [Stockmeyer, 1976].

Beyond 2ExpTime, one could also characterize classes in the exponential hierarchy [see Papadimitriou, 1994, Pages 497–498] by *cascading succinctness* (e.g., assuming a binary encoding of sequential circuit machines) or by *cascading observability* (e.g., by assuming multiple black-box circuits arranged in a pipelined architecture [Pnueli and Rosner, 1990]). In this manner, one could also introduce *information forks* [Finkbeiner and Schewe, 2005] in sequential circuit machines to arrive at full Turing completeness. Alternatively, one could think of an extension, where the amount of universal nondeterminism is generally unbounded and the existential quantification, additionally to the black-box circuit, also comprises the amount of existential nondeterminism.

Imposing more fine-grained restrictions. So far, we only considered restricting the computational power of sequential circuit machines by limiting the number of bits controllable by both the universal and existential nondeterminism. That is, we have seen that by restricting the *interface* (i.e., the number of inputs and outputs) of the combinatorial circuit C_{A}, one obtains the computational power equivalent to a particular major complexity class between LogSpace and 2ExpTime. An interesting direction for future research is to investigate the impact of restricting the structure of C_{A} on the computational power, which might lead to the discovery of new syntactic complexity classes with complete problems. This fundamentally

new characterization of complexity classes might shed new light on certain formidable problems for which the best known lower bound is NPTime and the best known upper bound is PSpace, such as the *reachability problem for timed automata with two-clocks* [Laroussinie et al., 2004] or the *interval-bound problem for weighted automata* [Bouyer et al., 2008].

Quasi-complete template-based synthesis. Beyond timed automata with a parametric control structure, in future work, one could expand the class of templates considered by the synthesis algorithm from Chapter 8. Particularly interesting is the introduction of parameters in the clock constraints. Results from parametric timed model checking [Alur et al., 1993b] indicate that the analysis of such templates is in general undecidable. However, subclasses of parametric timed automata, such as *L/U automata* [Hune et al., 2001], for which the emptiness problem is decidable, are promising candidates for a more expressive and yet computationally feasible class of templates.

The long-term goal is to obtain a succinct but comprehensive library of practically meaningful templates. Then, for a given plant, an automatic synthesis procedure could heuristically search that library for a feasible instantiation. We call such a procedure *quasi-complete* because it either finds a practically meaningful controller, or it reports that no such controller exists.

Extending the combination approach. In the abstraction refinement approach presented in Chapter 9, the refinement is computed based on a propagation of states. A natural extension would be to adapt advances from the field of model checking based on *counter example-guided abstraction refinement* [Clarke et al., 2003] and *interpolation* [McMillan, 2003, Brückner et al., 2008] to compute refinements for decreasing overapproximations. In some sense dual to that, one could develop techniques similar to *acceleration* [Boigelot and Wolper, 1994, Hendriks and Larsen, 2002, Bardin et al., 2005], for a faster increasing of underapproximations.

In the current implementation of Synthia, the CNF of the global edge relation (see Section 9.2.2 on Page 126) is represented explicitly as a list of clauses over Boolean functions and clock constraints. Instead, one could think of a *symbolic* representation using, e.g., diagram-based data structures [Ehlers et al., 2010a, Morbé et al., 2011]. Based on that, one could also think of a fully symbolic representation of the set of reachable states. Moreover, one could even think of a symbolic representation of the abstract control structures, avoiding the potentially expensive CNF to DNF conversion.

A more far-reaching extension is to generalize the combination approach from timed to arbitrary systems by allowing an arbitrary partitioning of the state bits. Having such a generalization, one could tackle other domains,

where one also has a combination of different forms of succinctness. For example, in the hardware verification domain, a design is usually divided into two parts: (1) the *datapath* for performing data processing operations (such as arithmetic operations), and (2) the *control unit* that determines the signals that control the functional units in the datapath (such as multiplexers, clock signals for registers, etc.). While efficient analysis techniques exists for each individual part, the combination of both still represents a challenging task.

Bibliography

Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 1991. ISBN 3-540-55564-1.

Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. In *2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, 2002.

Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999. ISBN 3-540-66202-2.

Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990. ISBN 3-540-52826-1.

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

Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS*, pages 414–425. IEEE Computer Society, 1990. ISBN 0-8186-2073-0.

Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993a.

Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *STOC*, pages 592–601. ACM, 1993b. ISBN 0-89791-591-7.

Eugene Asarin, Marius Bozga, Alain Kerbrat, Oded Maler, Amir Pnueli, and Anne Rasse. Data-structures for the verification of timed automata. In Oded Maler, editor, *HART*, volume 1201 of *LNCS*, pages 346–360. Springer, 1997. ISBN 3-540-62600-X.

Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In J.-F. Lafay, editor, *Proc. 5th IFAC*

Conference on System Structure and Control, pages 469–474. Elsevier, 1998.

Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11:625–656, 1995.

Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Ph. Schnoebelen. Flat acceleration in symbolic model checking. In Doron Peled and Yih-Kuen Tsay, editors, *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005. ISBN 3-540-29209-8.

Gerd Behrmann, Kim Guldstrand Larsen, Henrik Reif Andersen, Henrik Hulgaard, and Jørn Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 1999a. ISBN 3-540-65703-7.

Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *CAV*, volume 1633 of *LNCS*, pages 341–353, 1999b. ISBN 3-540-66202-2.

Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004. ISBN 3-540-23068-8.

Gerd Behrmann, Patricia Bouyer, Kim Guldstrand Larsen, and Radek Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *STTT*, 8(3):204–215, 2006.

Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-Tiga: Time for playing games! In Werner Damm and Holger Hermanns, editors, *Proc. 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 121–125. Springer, 2007.

Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995. ISBN 3-540-61155-X.

Dirk Beyer. Improvements in BDD-based reachability analysis of timed automata. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *LNCS*, pages 318–343. Springer, 2001. ISBN 3-540-41791-5.

Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for bdd-based verification of real-time systems. In Jr. and Somenzi [2003], pages 122–125. ISBN 3-540-40524-0.

Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994. ISBN 3-540-58179-0.

Bernd Borchert and Antoni Lozano. Succinct circuit representations and leaf language classes are basically the same concept. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(6), 1996.

Allan Borodin. On relating time and space to size and depth. *SIAM J. Comput.*, 6(4):733–744, 1977.

Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model checking for real-time systems. In Lin [1997], pages 25–.

Patricia Bouyer. Untameable timed automata! In Helmut Alt and Michel Habib, editors, *STACS*, volume 2607 of *Lecture Notes in Computer Science*, pages 620–631. Springer, 2003. ISBN 3-540-00623-0.

Patricia Bouyer and Fabrice Chevalier. On the control of timed and hybrid systems. *EATCS Bulletin*, 89:79–96, June 2006.

Patricia Bouyer, Deepak D'Souza, P. Madhusudan, and Antoine Petit. Timed control with partial observability. In Jr. and Somenzi [2003], pages 180–192. ISBN 3-540-40524-0.

Patricia Bouyer, Ulrich Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, and Jirí Srba. Infinite runs in weighted timed automata with energy constraints. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008. ISBN 978-3-540-85777-8.

Marius Bozga, Oded Maler, Amir Pnueli, and Sergio Yovine. Some progress in the symbolic verification of timed automata. In Orna Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 179–190. Springer, 1997. ISBN 3-540-63166-6.

Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing abstractions. *Fundam. Inform.*, 89(4):369–392, 2008.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

Tom Bylander. The computational complexity of propositional strips planning. *Artif. Intell.*, 69(1-2):165–204, 1994.

Jin-yi Cai and Merrick L. Furst. PSPACE survives constant-width bottlenecks. *Int. J. Found. Comput. Sci.*, 2(1):67–76, 1991.

Franck Cassez, Thomas A. Henzinger, and Jean-François Raskin. A comparison of control problems for timed and hybrid systems. In Claire Tomlin and Mark R. Greenstreet, editors, *HSCC*, volume 2289 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2002. ISBN 3-540-43321-X.

Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In Martín Abadi and Luca de Alfaro, editors, *Proc. 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.

Franck Cassez, Alexandre David, Kim Guldstrand Larsen, Didier Lime, and Jean-François Raskin. Timed control with observation based and stuttering invariant strategies. In Namjoshi et al. [2007], pages 192–206. ISBN 978-3-540-75595-1.

Rohit Chadha, Axel Legay, Pavithra Prabhakar, and Mahesh Viswanathan. Complexity bounds for the verification of real-time software. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 95–111. Springer, 2010. ISBN 978-3-642-11318-5.

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

Thomas Chatain, Alexandre David, and Kim G. Larsen. Playing games with timed games. In Alessandro Giua, Manuel Silva, and Janan Zaytoon, editors, *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09)*, Zaragoza, Spain, September 2009. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/CDL-adhs09.pdf.

Taolue Chen and Jian Lu. Towards the complexity of controls for timed automata with a small number of clocks. In Jun Ma, Yilong Yin, Jian Yu, and Shuigeng Zhou, editors, *FSKD (5)*, pages 134–138. IEEE Computer Society, 2008. ISBN 978-0-7695-3305-6.

Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. *Theor. Comput. Sci.*, 147(1&2):117–136, 1995.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. ISBN 3-540-11212-X.

Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001a.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2001b. ISBN 978-0-262-03270-4.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

Stephen A. Cook. Deterministic cfl's are accepted simultaneously in polynomial time and log squared space. In Michael J. Fischer, Richard A. DeMillo, Nancy A. Lynch, Walter A. Burkhard, and Alfred V. Aho, editors, *STOC*, pages 338–345. ACM, 1979.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. ISBN 978-0-262-03384-8.

Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1 (4):385–415, 1992.

Alexandre David. UPPAAL DBM Library release 2.0.8, 2011.

Stéphane Demri and Philippe Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Inf. Comput.*, 174(1):84–103, 2002.

Henning Dierks, Sebastian Kupferschmid, and Kim Guldstrand Larsen. Automatic abstraction refinement for timed automata. In Jean-François Raskin and P. S. Thiagarajan, editors, *FORMATS*, volume 4763 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007. ISBN 978-3-540-75453-4.

David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1989. ISBN 3-540-52148-8.

David L. Dill and Howard Wong-Toi. Verification of real-time systems by successive over and under approximation. In Pierre Wolper, editor, *CAV*, volume 939 of *LNCS*, pages 409–422. Springer, 1995. ISBN 3-540-60045-0.

Deepak D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In Helmut Alt and Afonso Ferreira, editors, *STACS*, volume 2285 of *Lecture Notes in Computer Science*, pages 571–582. Springer, 2002. ISBN 3-540-43283-3.

Rüdiger Ehlers. Symbolic bounded synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2010. ISBN 978-3-642-14294-9.

Rüdiger Ehlers, Daniel Fass, Michael Gerke, and Hans-Jörg Peter. Fully symbolic timed model checking using constraint matrix diagrams. In *RTSS*, pages 360–371. IEEE Computer Society, 2010a. ISBN 978-0-7695-4298-0.

Rüdiger Ehlers, Michael Gerke, and Hans-Jörg Peter. Making the right cut in model checking data-intensive timed systems. In Jin Song Dong and Huibiao Zhu, editors, *ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 565–580. Springer, 2010b. ISBN 978-3-642-16900-7.

Rüdiger Ehlers, Robert Mattmüller, and Hans-Jörg Peter. Combining symbolic representations for solving timed games. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *FORMATS*, volume 6246 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010c. ISBN 978-3-642-15296-2.

Javier Esparza. Decidability and complexity of petri net problems - an introduction. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996. ISBN 3-540-65306-6.

Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In Richard M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM AMS Proceedings*, pages 43–73, 1974.

Joan Feigenbaum, Sampath Kannan, Moshe Y. Vardi, and Mahesh Viswanathan. The complexity of problems on graphs represented as obdds. *Chicago J. Theor. Comput. Sci.*, 1999, 1999.

Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for ltl realizability. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2009. ISBN 978-3-642-02657-7.

Bernd Finkbeiner and Hans-Jörg Peter. Template-based controller synthesis for timed systems. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2012. ISBN 978-3-642-28755-8.

Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *LICS*, pages 321–330. IEEE Computer Society, 2005. ISBN 0-7695-2266-1.

Bernd Finkbeiner and Sven Schewe. SMT-based synthesis of distributed systems. In *Proceedings of the 2nd Workshop on Automated Formal Methods (AFM 2007), 6 November, Atlanta, Georgia, USA*, pages 69–76. ACM Press, 2007.

Bernd Finkbeiner, Hans-Jörg Peter, and Sven Schewe. Synthesizing certificates in networks of timed automata. In *IEEE Real-Time Systems Symposium*, pages 183–194. IEEE Computer Society, 2008. ISBN 978-0-7695-3477-0.

Olivier Finkel. On decision problems for timed automata. *Bulletin of the EATCS*, 87:185–190, 2005.

Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

Steven Fortune and James Wyllie. Parallelism in random access machines. In Lipton et al. [1978], pages 114–118.

Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56(3):183–198, 1983.

Pierre Ganty, Nicolas Maquet, and Jean-François Raskin. Fixed point guided abstraction refinement for alternating automata. *Theor. Comput. Sci.*, 411 (38-39):3444–3459, 2010.

Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In Lipton et al. [1978], pages 89–94.

Ganesh Gopalakrishnan and Shaz Qadeer, editors. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, 2011. Springer. ISBN 978-3-642-22109-5.

Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a source of complexity in logical formalisms. *Ann. Pure Appl. Logic*, 97(1-3):231–260, 1999.

Erich Grädel. Capturing complexity classes by fragments of second-order logic. *Theor. Comput. Sci.*, 101(1):35–57, 1992.

Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*, 2002. Springer. ISBN 3-540-00388-6.

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997. ISBN 3-540-63166-6.

David Harel, Orna Kupferman, and Moshe Y. Vardi. On the complexity of verifying concurrent transition systems. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 1997. ISBN 3-540-63141-0.

Juris Hartmanis and Richard Edwin Stearns. On the computational complexity of algorithms. *Transactions of the AMS*, 117:285–306, 1965.

Martijn Hendriks and Kim Guldstrand Larsen. Exact acceleration of real-time model checking. *Electr. Notes Theor. Comput. Sci.*, 65(6):120–139, 2002.

Thomas A. Henzinger and Peter W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221(1-2):369–392, 1999.

Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2): 193–244, 1994.

Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2001. ISBN 3-540-41865-2.

Neil Immerman. Upper and lower bounds for first order expressibility. In *FOCS*, pages 74–82. IEEE Computer Society, 1980.

Neil Immerman. Number of quantifiers is better than number of tape cells. *J. Comput. Syst. Sci.*, 22(3):384–406, 1981.

Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999. ISBN 978-0-387-98600-5.

Neil D. Jones. Space-bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci.*, 11(1):68–85, 1975.

Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-40524-0.

Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. ISBN 0-306-30707-3.

Gal Katz, Doron Peled, and Sven Schewe. Synthesis of distributed control through knowledge accumulation. In Gopalakrishnan and Qadeer [2011], pages 510–525. ISBN 978-3-642-22109-5.

Orna Kupferman and Sarai Sheinvald-Faragy. Finding shortest witnesses to the nonemptiness of automata on infinite words. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 492–508. Springer, 2006. ISBN 3-540-37376-4.

Orna Kupferman, Yoad Lustig, Moshe Y. Vardi, and Mihalis Yannakakis. Temporal synthesis for bounded systems and environments. In Thomas Schwentick and Christoph Dürr, editors, *STACS*, volume 9 of *LIPIcs*, pages 615–626. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. ISBN 978-3-939897-25-5.

Richard E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975.

François Laroussinie and Philippe Schnoebelen. The state explosion problem from trace to bisimulation equivalence. In Jerzy Tiuryn, editor, *FoSSaCS*, volume 1784 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2000. ISBN 3-540-67257-5.

François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Model checking timed automata with one or two clocks. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2004. ISBN 3-540-22940-X.

Kim G. Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

Kim G. Larsen, Gerd Behrmann, and Arne Skou. Exercises for Uppaal. http://www.cs.aau.dk/~bnielsen/TOV08/ESV04/exercises, 2004.

Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In Lin [1997], pages 14–24.

Kwei-Jay Lin, editor. *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, 1997. IEEE Computer Society.

Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors. *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, 1978. ACM.

Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In Benjamin Kuipers and Bonnie L. Webber, editors, *AAAI/IAAI*, pages 748–754. AAAI Press / The MIT Press, 1997. ISBN 0-262-51095-2.

Antoni Lozano and José L. Balcázar. The complexity of graph problems for succinctly represented graphs. In Manfred Nagl, editor, *WG*, volume 411 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 1989. ISBN 3-540-52292-1.

Yoad Lustig and Moshe Y. Vardi. Synthesis from component libraries. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2009. ISBN 978-3-642-00595-4.

Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In Ernst W. Mayr and Claude Puech, editors, *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 1995.

Zohar Manna and Henny Sipma. Alternating the temporal picture for safety. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 429–450. Springer, 2000. ISBN 3-540-67715-1.

Kenneth L. McMillan. Interpolation and sat-based model checking. In Jr. and Somenzi [2003], pages 1–13. ISBN 3-540-40524-0.

Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc, 1967.

Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. *ENTCS*, 23(2), 1999.

M. Oliver Möller. CSMA/CD model generator for Uppaal. http://www.it.uu.se/research/group/darts/uppaal/benchmarks/genCSMA_CD.awk, 2001.

M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate abstraction for dense real-time system. *Electr. Notes Theor. Comput. Sci.*, 65(6):218–237, 2002.

Georges Morbé, Florian Pigorsch, and Christoph Scholl. Fully symbolic model checking for timed automata. In Gopalakrishnan and Qadeer [2011], pages 616–632. ISBN 978-3-642-22109-5.

Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors. *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, volume 4762 of *Lecture Notes in Computer Science*, 2007. Springer. ISBN 978-3-540-75595-1.

Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. ISBN 978-0-201-53082-7.

Christos H. Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71(3):181–185, 1986.

Hans-Jörg Peter and Bernd Finkbeiner. The complexity of bounded synthesis for timed control with partial observability. accepted for publication. In Marcin Jurdzinski and Dejan Nickovic, editors, *FORMATS*, Lecture Notes in Computer Science. Springer, 2012.

Hans-Jörg Peter and Robert Mattmüller. Component-based abstraction refinement for timed controller synthesis. In Theodore P. Baker, editor, *IEEE Real-Time Systems Symposium*, pages 364–374. IEEE Computer Society, 2009. ISBN 978-0-7695-3875-4.

Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. Synthia: Verification and synthesis for timed automata. In Gopalakrishnan and Qadeer [2011], pages 649–655. ISBN 978-3-642-22109-5.

Nicholas Pippenger. On simultaneous resource bounds. In *FOCS*, pages 307–311. IEEE Computer Society, 1979.

Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society, 1990.

Vaughan R. Pratt and Larry J. Stockmeyer. A characterization of the power of vector machines. *J. Comput. Syst. Sci.*, 12(2):198–221, 1976.

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982. ISBN 3-540-11494-7.

John H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.

Jussi Rintanen. Complexity of planning with partial observability. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 345–354. AAAI, 2004. ISBN 1-57735-200-9.

Walter L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22 (3):365–383, 1981.

Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

Walter J. Savitch. Recursive turing machines. *Internat. J. Comput. Math.*, 6:3–31, 1977.

Walter J. Savitch. Parallel random access machines with powerful instruction sets. *Mathematical Systems Theory*, 15(3):191–210, 1982.

Walter J. Savitch and Michael J. Stimson. Time bounded random access machines with parallel processing. *J. ACM*, 26(1):103–118, 1979.

Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In Namjoshi et al. [2007], pages 474–488. ISBN 978-3-540-75595-1.

Philippe Schnoebelen. The complexity of temporal logic model checking. In Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter, and Michael Zakharyaschev, editors, *Advances in Modal Logic*, pages 393–436. King's College Publications, 2002. ISBN 0-9543006-2-9.

Sanjit A. Seshia and Randal E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In *CAV*, volume 2725 of *LNCS*, pages 154–166, 2003. ISBN 3-540-40524-0.

A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

Fabio Somenzi. CUDD: CU Decision Diagram package release 2.4.2, 2009.

Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis II. Hierarchies of memory limited computations. In *SWCT (FOCS)*, pages 179–190. IEEE Computer Society, 1965.

Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

Larry J. Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.

Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–99, 1987.

Stavros Tripakis. Folk theorems on the determinization and minimization of timed automata. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 182–188. Springer, 2003. ISBN 3-540-21671-5.

Grighori S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1968.

Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Society*, 2(42):230–265, 1937.

Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. Elsevier, 1990.

Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *STOC*, pages 137–146. ACM, 1982. ISBN 0-89791-070-2.

Moshe Y. Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997. ISBN 3-540-63104-6.

Helmut Veith. Languages represented by boolean formulas. *Inf. Process. Lett.*, 63(5):251–256, 1997.

Heribert Vollmer. *Introduction to circuit complexity - a uniform approach.* Texts in theoretical computer science. Springer, 1999. ISBN 978-3-540-64310-4.

Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. Timed automata for the development of real-time systems. Technical report, Queen's University, Ontario, Canada, 2011. URL http://research.cs.queensu.ca/TechReports/Reports/2011-579.pdf.

Farn Wang. Efficient verification of timed automata with BDD-like data structures. *STTT*, 6(1):77–97, 2004.

Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

Marty J. Wolf. Nondeterministic circuits, space complexity and quasigroups. *Theor. Comput. Sci.*, 125(2):295–313, 1994.

Celia Wrathall. Complete sets and the polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):23–33, 1976.

Satoshi Yamane and Kazuhiro Nakamura. Development and evaluation of symbolic model checker based on approximation for real-time systems. *Systems and Computers in Japan*, 35(10):83–101, 2004.

Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1 (1-2):123–133, 1997.