Chapter 1

A language-theoretic view of verification

1.1 Setting the stage

The use of automata in verification goes back a long way, to Büchi [Bü60], Elgot [Elg61] and Trakhtenbrot [Tra61] who, in the early 1960s, used the theory of automata on finite words to give an algorithm to check whether a sentence of monadic second order logic on such structures is valid, true in all models. They showed that this validity problem can be reduced to checking whether the language accepted by such a finite automaton is empty. Büchi went on to develop [Bü62] a theory of finite automata on infinite words and proved the same result (which was considerably harder). This would come into use in verification nearly 25 years later!

These days, the verification question is posed as follows. We are given a model of a system (which can be hardware, software, or even mixed), typically as some kind of transition system. More precisely, we might be provided with a run of the system, modelled as a word over a suitable alphabet. We are given a system property, specified by a logical formula, typically in a propositional framework such as temporal logic (rather than monadic second order logic). Does the run satisfy this property? Or, do all runs of the system satisfy this property? These **truth checking** and **model checking** problems can be posed as membership or inclusion problems in automata theory, since the set of models of a temporal logic formula is a set of words—a language.

One of the highlights of the modern approach to automata theory is a careful consideration of algorithmic questions and the determination of their precise complexity. We refer the reader to the many textbooks available on complexity theory, but we give here a primer on complexity classes pertaining to this chapter. The reader familiar with basic complexity notions is advised to skip to the next section.

Acknowledgements. Thanks to Simoni Shah and Paul Gastin for their detailed comments, and to Nutan Limaye and A.V. Sreejith for their help.

1.1.1 A primer on complexity classes

The complexity of a problem can be measured in terms of the number of computation steps (this is abbreviated as time) when the input is of a particular size, say n, but also in terms of the memory used (abbreviated space) as a function of n.

Nondeterministic algorithms. The algorithms we consider can be either deterministic, which is usual, or nondeterministic, which is not so usual. A standard paradigm of how a nondeterministic algorithm works is exemplified by:

Problem 1 (Directed graph accessibility, DGAP)
Instance: A directed graph G with n vertices, a source vertex s and a target vertex t
Question: Is there a path from s to t?
Complexity: NLogSpace

Here is how a nondeterministic algorithm might work: it writes out a string of vertices $v_1 \dots v_m$ of length $m \leq n$. This is called a guess. Now the algorithm checks whether the first vertex v_1 is s, each pair of vertices (v_i, v_{i+1}) is an edge in the graph and if the last vertex v_m is t. If this is indeed the case, the algorithm says "Yes" (and returns the guess if required), otherwise it says "No" (there is no path from s to t).

Surprisingly, this is indeed a nondeterministic *algorithm* because if there is a path from s to t, there is *one* run of the algorithm which is correct, and if there is no path from s to t, *all* the possible runs put together have examined all the possible paths of length $\leq n$ from s and not reached t, which is sufficient to conclude the "No" answer.

The time complexity of the algorithm depends upon the way the graph is represented, but certainly it can be done in time polynomial in n. We can say the algorithm is nondeterministic polynomial time or in NPtime.

If we analyze the memory usage, we find that we can optimize the algorithm a little: instead of guessing the whole path, the algorithm first guesses the pair (v_1, v_2) and verifies that this is an edge with $v_1 = s$, then it erases v_1 and guesses v_3 to form the pair (v_2, v_3), verifies that this is an edge, and so on. (v_3 might be the same as v_1, but the existence of a walk of length $< n$ guarantees a path of length $< n$.) Since writing out a vertex of a graph with n nodes and managing a counter takes $O(\log n)$ bits, this is a nondeterministic logarithmic space or NLogSpace algorithm. Now we have a complexity upper bound of NLogSpace for DGAP, which is better than NPtime.

Upper bounds. Complexity theorists have defined classes of algorithms like we did above, and shown that the following inclusions hold between them. None of them is known to be strict, but we do know of exponential separation between classes (for example, NLogSpace $\neq$ NPspace, Ptime $\neq$ ExpTime).

ALogTime $\subseteq$ LogSpace $\subseteq$ NLogSpace $\subseteq$ Ptime $\subseteq$ NPtime $\subseteq$ Pspace $=$ NPspace
Pspace $=$ NPspace $\subseteq$ ExpTime $\subseteq$ NExpTime $\subseteq \dots \subseteq$ Elementary $\subset$ Computable

More precisely, complexity theory classifies *problems* rather than *algorithms*, since it is possible to write very inefficient algorithms! So the class LogSpace consists of all problems which have deterministic algorithms using space bounded by a polynomial in $\log n$, where n is the size of the input of the problem instance. The even more restrictive class ALogTime is sometimes also called NC1, but there are some technicalities involved and it is best to consult a textbook for the details. We will use this class only for the reason that it exactly matches the complexity of evaluation of propositional logic (or Boolean sentences), a result shown by Buss [Buss87].

Problem 2 (Boolean sentence value, BSVP)
Instance: A tree with vertices labelled from $\{\wedge, \vee, 0, 1\}$*, where the vertices labelled* 0 *or* 1 *are sources.*
Question: Does this "formula", with the vertex labels being given their usual boolean interpretation, evaluate to 1 *at the root?*
Complexity: ALogTime

The class NLogSpace has all problems which have *nondeterministic* algorithms with this space requirement. Since a deterministic algorithm is a special case of a nondeterministic one, the inclusion of a deterministic class in a nondeterministic class with the same requirements follows. What about the other way? Savitch [Sav70] gave a clever divide-and-conquer technique which shows that any NLogSpace algorithm can be converted to one which is deterministic, but the space usage is squared.

The classes Ptime and NPtime contain deterministic and nondeterministic algorithms whose *time* requirement is a polynomial in n. If an algorithm takes space $s(n)$, after time exponential in $s(n)$ it will return to a configuration which it had already seen earlier. A more formal proof based on this argument, which can be found in a textbook, shows the inclusion of a class taking space $O(s(n))$ in a deterministic class taking time $2^{O(s(n))}$.

The class Pspace contains problems which have algorithms taking space polynomial in n. Applying Savitch's theorem, we see that NPspace turns out to be the same as Pspace. An algorithm taking time order $t(n)$ can only use space order $t(n)$, since it does not have the time to reach more space! This gives inclusion of the time classes in space classes with the same requirements.

The classes ExpTime and NExpTime have algorithms whose time requirement is $2^{p(n)}$, for some function $p(n)$ polynomial in n. Higher classes can also be defined similarly, using towers of exponentials. The class Elementary includes algorithms whose space is a tower of exponentials of any constant height. Finally, the class Computable includes all problems which have any kind of algorithm at all.

Lower bounds. We are not yet done with DGAP, we want to find a lower bound for it. Complexity theory really took off in the 1970s when Steve Cook discovered how to give lower bounds to a problem.

A problem P is said to reduce to a problem Q (we write $P \leq Q$) if, suppose we are given an algorithm for Q, we can find a "preprocessing" function f which will take any instance

I of the problem P and translate it into an instance $f(I)$ of the problem Q, such that $f(I)$ has a solution if, and only if, I has a solution. Since it does not make sense for f to take more time or space than the algorithm for Q itself, we will assume that f is itself computed by an algorithm which takes logarithmic space. A textbook on complexity would call this a "many-one logspace reduction".

Here is Cook's insight. Suppose we find a problem Q in NLogSpace such that *every* problem in NLogSpace reduces to Q (we write NLogSpace $\leq Q$). Indeed, DGAP is such a problem, which is said to be NLogSpace-hard. To prove that NLogSpace is a lower bound for some problem R, it is sufficient to show that DGAP reduces to R. In symbols, NLogSpace $\leq$ DGAP and DGAP $\leq R$ shows NLogSpace $\leq R$.

DGAP is also NLogSpace-complete: in NLogSpace as well as NLogSpace-hard, giving both a lower and an upper bound. Cook's theorem showed that NPtime $\leq$ SAT, the problem of checking whether a propositional logic formula is satisfiable. (Since SAT is also in NPtime, SAT is NPtime-complete.) We can say that the complexity of DGAP *is* NLogSpace, since NLogSpace is both an upper and a lower bound. Similarly, the monotone circuit value problem MCVP defined below is a problem whose complexity *is* Ptime.

It is not known whether there is a *deterministic* LogSpace algorithm for DGAP. By a simple analysis, you can work out that showing that there is a LogSpace algorithm for the DGAP problem amounts to showing that LogSpace=NLogSpace. This question, like the more famous Ptime = NPtime question, is yet unsolved.

Problem 3 (Monotone circuit value, MCVP)
Instance: A directed acyclic graph G with vertices labelled from $\{\wedge, \vee, 0, 1\}$ and a sink vertex v, where the vertices labelled 0 or 1 are sources.
Question: Does this "circuit", with the vertex labels being given their usual boolean interpretation, evaluate to 1 at the output v?
Complexity: Ptime

The circuit may be assumed to be layered, with alternate layers being labelled $\wedge$ and $\vee$, until we reach the input layer which has 0 or 1 nodes.

Lower bounds using simulations. To give a Pspace lower bound, we again need a "hard" problem as earlier. We show next how this is done using a "simulation" technique.

A computation of a nondeterministic algorithm can be thought of as a sequence of configurations $\#c_0\#c_1\#\ldots$. We consider the case of a *linear space* computation. That is, given an input w of size n to the algorithm, over an alphabet A, each configuration c_i is a word of length $n+1$ over a finite alphabet (the extra letter encodes some "state" information).

Now comes the simulation: *the whole computation itself* is a word over a larger alphabet $B \supset A$ (B includes symbols like # and state information). This word can be of arbitrary length. The initial configuration consists of an *initial state* (of the computation) and the input w provided to the computation. The final configuration is decided by a *final state* of the computation. Moving from the configuration c_i of the computation to c_{i+1}, the next

configuration in the sequence, is represented by changing just three consecutive letters in the word c_i in accordance with the algorithm. This surprising representation was discovered by Alan Turing and formalized in his now-famous Turing machine. A textbook would call our description that of a "linear bounded machine". The problem which we now consider is whether such a simulation can be found.

Problem 4 (Valid computation for a linear space algorithm)
Instance: A word w over a finite alphabet $A \subset B$
Question: Is there a valid linear space computation, as a word over B, of the algorithm on input w?
Complexity: PSPACE

Notice that the algorithm itself is implicit in the problem. It might be desirable to have a fixed alphabet. This can be easily done using a two-letter alphabet since the encoding only adds a constant factor.

1.2 Membership, emptiness and inclusion

This section is an introduction to the complexity of three fundamental problems of language theory. We sketch a couple of lower bound proofs which are published but might not be easily accessible. We also cast our results in a logical setting by introducing an interval logic.

1.2.1 The membership problem

In formal language theory, when we specify a membership problem between a word and a language, we have to present the input language, which might be infinite, in a finite way. For example, one might provide a machine description.

Problem 5 (NFA membership)
Instance: A finite word w of length n and a nondeterministic finite automaton M of size m
Size of instance: $n + m$
Question: Is w accepted by M?
Complexity: NLOGSPACE

In this case, since a nondeterministic finite automaton is a labelled directed graph, it is an exercise to work out that the complexity of this problem matches that of the graph accessibility problem (DGAP) and the membership problem NFA is NLOGSPACE-complete.

1.2.2 Different kinds of regular expressions

Specifying the input language by a syntactic entity like a regular expression, rather than a graphical one such as an automaton, must have been a natural afterthought of the development of program verification in the late 1960s and early 1970s.

First attempts at formulating the problem of verifying a program were made by Naur [Naur66] and Floyd [Flo67], and Hoare developed the first programming logic [Hoa69] where the program was presented as a syntactic expression. Soon after, Stockmeyer and Meyer observed that the membership problem for a language given by a regular expression (including complement operations) is in PTIME [SM73]. This is a textbook exercise (for example, in Hopcroft and Ullman) using a dynamic programming algorithm.

Here are the definitions for these extended regular expressions (ERE), along with regular expressions (RE) and starfree expressions (SF)—the last-named are regular expressions with complement, but without the Kleene star operator.

$e ::= a \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e_1^* \mid \overline{e_1}$

$r ::= \emptyset \mid a \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r_1^*$

$s ::= \varepsilon \mid a \mid s_1 \cdot s_2 \mid s_1 + s_2 \mid \overline{s_1}$

As usual, $e_1 \cdot e_2$ is written $e_1 e_2$ where no confusion arises. $L(e)$ is the language defined by the expression e: thus all words are in $L(\overline{\emptyset})$. We will use the notation $e_1 \cap e_2$ for the derived operator $\overline{\overline{e_1} + \overline{e_2}}$.

Next we enumerate, following Schützenberger [Sch65], all ways in which words in the language of a starfree expression, of length at least two, can be broken up into a prefix and a suffix, again described using a pair of starfree expressions.

$Br(\varepsilon) = Br(a) = \emptyset$

$Br(s_1 s_2) = \{(s_1, s_2)\} \cup \{(s_{11}, s_{12}s_2) \mid (s_{11}, s_{12}) \in Br(s_1)\} \cup \{(s_1 s_{21}, s_{22}) \mid (s_{21}, s_{22}) \in Br(s_2)\}$

$Br(s_1 + s_2) = Br(s_1) \cup Br(s_2)$

Suppose $Br(s) = \{(e_1, f_1), \ldots, (e_k, f_k)\}$. Then:

$$Br(\overline{s}) = \bigcup_{I \subseteq \{1,\ldots,k\}} \{(\bigcap_{i \in I} \overline{e_i}, \bigcap_{j \notin I} \overline{f_j})\}$$

By induction on expressions s, for any word w of length at least two, it can be shown that $w \in L(s)$ if and only if there is an (e, f) in $Br(s)$ such that $w \in L(e \cdot f)$.

The tricky case is complementation. Suppose $Br(s) = \{(e_1, f_1), \ldots, (e_k, f_k)\}$. Let $w = uv \in L(\overline{s})$, that is, $uv \notin L(s)$. Using the induction hypothesis on s, for $1 \leq i \leq k$, either $u \notin L(e_i)$ or $v \notin L(f_i)$. Let $I = \{i \mid u \notin L(e_i)\}$. Then $u \in L(\bigcap_{i \in I} \overline{e_i})$ and $v \in L(\bigcap_{j \notin I} \overline{f_j})$.

Conversely suppose some I, $u \in L(\bigcap_{i \in I} \overline{e_i})$ and $v \in L(\bigcap_{j \notin I} \overline{f_j})$. For contradiction suppose $uv \notin L(\overline{s})$, that is, $uv \in L(s)$. By the induction hypothesis there is some $1 \leq l \leq k$ such that $u \in L(e_l)$ and $v \in L(f_l)$ which contradicts our supposition.

1.2.3 The membership problem for regular expressions

Having introduced the different kinds of regular expressions, we now proceed to consider their membership problems.

Problem 6 (ERE, RE, SF membership)
Instance: A finite word w and a regular expression e in the class ERE/RE/SF

Question: Is w in the language defined by e?
Complexity: NLogSpace *for RE,* Ptime *for ERE and SF*

The membership problem RE is NLogSpace-complete. This was shown by Jiang and Ravikumar [JR91]. It is in NLogSpace since the usual inductive Kleene construction of a finite automaton from a regular expression can be thought of as a logspace reduction from RE to FA. That NLogSpace is a lower bound (even without using Kleene star and complement) is established by a reduction from DGAP.

Consider the graph accessibility problem (DGAP) where the input graph G has self-loops for all nodes. This subcase is also NLogSpace-complete. Encode the vertices of the graph in unary as $a, a^2, \ldots, a^n$. Let the complement $\overline{a^i}$ of the node a^i be defined as a^{n-i}.

Define words $w_1 \ldots w_n$ by taking edges (u, v) as substrings $u \cdot b \cdot \overline{v}$. A path from s to t is described by the expression $e = start \cdot (middle)^{n-2} \cdot end$, with $start = s \cdot b \cdot (\Sigma_{(s,u)\in G}\overline{u})$, $middle = \Sigma_{(u,v)\in G}(u \cdot b \cdot \overline{v})$ and $end = (\Sigma_{(v,t)\in G} v) \cdot b \cdot \overline{t}$.

Now a word $w = s \cdot (ba^n)^{n-2} \cdot \overline{t}$ is in the language defined by e iff G has a path from s to t. The expression e and word w can be constructed from the description of DGAP in LogSpace, so we have that NLogSpace is a lower bound for RE membership.

Petersen showed that the complexity of ERE and SF membership is Ptime [Pet00], by a reduction from MCVP. Assume that the gates in the circuit are enumerated so that the inputs to each gate occur earlier in the enumeration. The reduction inductively constructs starfree expressions s_k whose language L_k is such that for every $j \leq k$, the word a^{2j} is in L_k if and only if the output of gate j is 1. Let All_k be an expression whose language contains all the a^{2j} for $j \leq k$.

For the base case, $s_0 = \emptyset$. The induction step for gate k being labelled $\vee$ and having inputs from the gates i, j is

$s_k = All_k \cap (((s_{k-1}(a^{2(k-i)-1} + a^{2(k-j)-1} + \varepsilon)) \cap (All_k + a^{2k-1}))(a + \varepsilon))$

The idea is that all words in L_{k-1} have even length, and so a^{2k-1} can be formed only by concatenating $a^{2(k-i)-1}$ or $a^{2(k-j)-1}$ with some word in L_{k-1}. Solving the linear equations thus formed will be possible only if a^{2i} or a^{2j} are in L_{k-1}. The remaining part of s_k adds a^{2k} to L_k if and only if the previous part generated a^{2k-1}.

A similar expression (using negations) takes care of the $\wedge$ case. The reduction can be carried out in LogSpace by reading the circuit description from right to left, generating the parentheses in the expression, and then filling in the rest of the expression from left to right. Storing the numbers and expanding them out as a's in unary notation can be done in logarithmic space.

1.2.4 The nonemptiness and inclusion problems

The other problems typically considered in formal language theory are emptiness and inclusion.

Problem 7 (RE nonemptiness)
Instance: An expression $e \in RE$

Question: Is the language defined by e nonempty?
Complexity: ALogTime

Problem 8 (NFA nonemptiness)
Instance: A finite automaton M
Question: Is the language defined by M nonempty?
Complexity: NLogSpace

Nonemptiness checking of finite automata is another textbook construction, and the problem again matches DGAP. With reference to emptiness, regular expressions can be mapped rather directly to boolean formulae by translating a letter (or the empty word) to 1 and the empty set to 0, with concatenation serving as $\wedge$, union as $\vee$ and star as a constant function returning 1. Now we have that a regular expression is nonempty if and only if the translated formula has value 1.

Problem 9 (RE/NFA inclusion)
Instance: An expression $e_1 \in RE$ or a finite automaton M_1, another expression $e_2 \in RE$ or a finite automaton M_2
Question: Is L_1, the language defined by e_1 or M_1, included in L_2, the language defined by e_2 or M_2?
Complexity: Pspace

We use a reduction to emptiness. Since $L_1 \subseteq L_2$ is equivalent to $L_1 \cap \overline{L_2} = \emptyset$ we use the Rabin-Scott subset construction to complement an automaton, causing an exponential blow-up in size, and solve the problem in Pspace, and because of the Kleene reduction it does not matter whether the input is a regular expression in RE or an automaton.

Meyer and Stockmeyer observed [MS72] that those words which are *not* valid computations of a linear space nondeterministic algorithm can be described by a regular expression which says that either the initial configuration is wrong, or the final configuration is wrong, or a move from some c_i to c_{i+1} is wrong. This last bit can be done since the distance in the word between the three letters which change from c_i to c_{i+1} is exactly n. Hence an $O(n)$ regular expression r can be written to describe these three cases. The expression constructed satisfies $Lang(r) = B^*$ (which is the same as $B^* \subseteq Lang(r)$) if and only if the computation is not valid. This is a reduction from the validity problem of linear space computations to the inequivalence and non-inclusion problems of regular expressions, making them Pspace-complete.

Things change dramatically when the language L_2 is presented as a regular expression with complement (ERE or SF).

Problem 10 (ERE/SF nonemptiness)
Instance: An expression $e \in ERE$
Question: Is the language defined by e nonempty?
Complexity: above Elementary*, even if $e \in SF$*

Problem 11 (ERE/SF inclusion)
Instance: An expression $e_1 \in ERE$ or a finite automaton M_1, another expression $e_2 \in ERE$
Question: Is L_1, the language defined by e_1 or M_1, included in L_2, the language defined by e_2?
Complexity: above ELEMENTARY*, even if $e_2 \in SF$*

By the Kleene construction we know the problem can be solved by constructing a finite automaton for L_2 whose size is a tower of exponentials of height n determined by the nesting depth of negations in the expression for L_2, but the ELEMENTARY class is a *lower bound* for this problem, and no algorithm with space bounded by a tower of exponentials of constant height suffices! A sketch of the proof ideas is given in the book of Aho, Hopcroft and Ullman [AHU74]. The details are in the PhD thesis of Stockmeyer [Sto74], and a proof for the related satisfiability problem of monadic second order logic is in [Mey75].

1.2.5 Interval temporal logic

Since a syntax such as the starfree expressions has union as well as complementation, it can be reformulated as a logic. We describe below a variant of the propositional interval temporal logic (ITL) defined by Moszkowski and Manna [MM83], where the propositions are related to the letters of the alphabet. B stands for a set of letters from the alphabet A, and the special proposition pt denotes a point, a single letter. The ";" operator is called chop (sometimes also called fusion) and is defined slightly differently from concatenation in starfree expressions. A closely related syntax is called starfree chop expressions (SFCE) in Chapter **??** by Ajesh Babu and Pandya.

$\phi ::= \lceil\lceil B \rceil\rceil, B \subseteq A \mid pt \mid \neg\ \phi \mid \phi \vee \psi \mid \phi; \psi$

Temporal logics are interpreted on state sequences $\pi = s_1 s_2 \ldots$, where each state s_i is a set of atomic propositions. In the present case, since we want to mimic the starfree expressions, we let $\pi : N \to A$ be a nonempty word over the alphabet A.

For an interval logic, when a word satisfies a formula is inductively defined using intervals $[b, e]$ where $b \leq e$ are natural numbers. Alternatively they could be thought of as substrings $\pi(b) \ldots \pi(e)$ of a word.

$\pi, [b, e] \models \lceil\lceil B \rceil\rceil$ iff for all $m : b \leq m \leq e : \pi(m) \in B$

$\pi, [b, e] \models pt$ iff $b = e$

$\pi, [b, e] \models \phi; \psi$ iff $\exists m : b \leq m \leq e : \pi, [b, m] \models \phi$ and $\pi, [m, e] \models \psi$

We now restrict ourselves to finite word models and say that $\pi \models \phi$ if $\pi, [1, |\pi|] \models \phi$ holds. $Lang(\phi)$, the language defined by ϕ, is the set of words $\{\pi \mid \pi \models \phi\}$. Now we consider the truth checking or membership problem for this logic.

Problem 12 (ITL truth checking)
Instance: A finite word u over alphabet A and an ITL formula ϕ
Question: Does $u \models \phi$? Alternately, is u in $Lang(\phi)$?
Complexity: PTIME

Because of their similarity, the dynamic programming algorithm for the membership problem of starfree expressions can be used for the membership problem of a word against an ITL formula. Now we use the results of Petersen [Pet00] to show that the complexity of this problem *is* PTIME. Similarly ITL satisfiability corresponds to the nonemptiness problem of starfree expressions and we can use the results of [Sto74].

Problem 13 (SAT(ITL), ITL satisfiability)
Instance: An ITL formula ϕ
Question: Does ϕ *have a word model? Alternately, is* $Lang(\phi)$ *nonempty?*
Complexity: above ELEMENTARY

1.3 The membership problem for linear temporal logic

The propositional temporal logic of linear time LTL was introduced in the verification setting by Pnueli [Pnu77]. Let us backtrack a bit to establish how this came about.

We saw that Hoare introduced a programming logic [Hoa69]. Hoare's notation $\alpha\{Program\}\beta$ specified the property that all runs of $Program$, when started in a state satisfying the pre-assertion α, would end in a state satisfying the post-assertion β, provided the run terminated.

These "invariant" assertions were complemented by Burstall's "intermittent" assertions for proving termination itself [Bur74], and Manna and Pnueli realized that the program could be kept implicit, its run described as a sequence of states, and an assertion α could be made in a logic of sequences, namely, linear temporal logic (LTL) in which both kinds of assertions could be described.

Over the same period, several people working on extending Hoare logic to concurrent programs (Owicki and Gries [OG76]; Apt, Francez and de Roever [AFdR80]; Misra and Chandy [MC81]; Soundararajan [Sou83]) realized that the completeness proofs for their logics demanded an encoding of "histories" or sequences inside the assertion language. Meanwhile, Pnueli demonstrated [Pnu77] that interesting properties of concurrent protocols (such as mutual exclusion) could be stated and proved in temporal logic. His book with Manna [MP92] is a repository of such properties. Hence model checking came to be based on temporal logic.

1.3.1 LTL

We consider a basic temporal logic LTL. The operator X below is read as "next", and the operator U as "until". We refer to the book of Huth and Ryan [HR00] for motivating examples.

$\alpha ::= p \in Prop \mid \neg\alpha \mid \alpha \vee \beta \mid \mathsf{X}\alpha \mid \alpha\mathsf{U}\beta$

The formula $\mathsf{F}\alpha$ is defined as $true\mathsf{U}\alpha$ and $\mathsf{G}\alpha$ as $\neg\mathsf{F}\neg\alpha$.

LTL is interpreted on an infinite state sequence $\pi = s_1 s_2 \ldots$, where each state s_i is labelled by a function $\pi : N \to \wp(Prop)$ with a set of atomic propositions. Alternatively π could be read as an infinite word over the alphabet $A = \wp(Prop)$. One can also consider nonempty finite words as models of LTL, but we will not need them in this chapter.

When a word satisfies a formula is defined traditionally. Instead of using the interval $[k, \infty)$ we just use the index k.

$\pi, k \models p$ iff $p \in \pi(k)$

$\pi, k \models \mathsf{X}\alpha$ iff $\pi, k+1 \models \alpha$

$\pi, k \models \mathsf{F}\alpha$ iff for some $m \geq k : \pi, m \models \alpha$

$\pi, k \models \alpha \ \mathsf{U} \ \beta$ iff for some $m \geq k : \pi, m \models \beta$
and for all $l : k \leq l < m : \pi, l \models \alpha$

This time we say $\pi \models \alpha$ holds when $\pi, 1 \models \alpha$ holds, that is, the whole infinite word is a model of the formula. This is the same thing as whether an infinite word is a member of a language of infinite words, the latter given by a logical formula.

We next indicate how LTL could be translated into starfree expressions. To be more precise, we should use starfree expressions which describe languages of infinite words, but here we only sketch the ideas using the usual starfree expressions over finite words.

$sf(p) = (\Sigma_{p \in a} a) \cdot \overline{\emptyset}$

$sf(\mathsf{X}\alpha) = (\Sigma_{a \in A} a) \cdot sf(\alpha)$

The translation of until requires more work. From the semantics of the operator, we know that a word $w[1, \infty)$ satisfies the formula $\alpha \mathsf{U} \beta$ if it has a suffix $w[m, \infty)$ which satisfies β and the "intermediate" suffixes $w[l, \infty)$ for $1 \leq l < m$ satisfy α.

So consider that $w[1, \infty)$ satisfies α. Inductively, the starfree expression $sf(\alpha)$ will describe such words. But we have to restrict this set of words to those where α holds for some suffixes after which β holds. To describe this, Pnueli and Zuck [PZ93] found a technique of forcing prefixes which satisfy α until a suffix which satisfies β is reached. The translation of an until formula uses Schützenberger's breakups which we saw earlier. The expression $e \cap \overline{(\overline{\emptyset} \cdot \overline{e})}$ denote all words described by e all whose suffixes are again in $L(e)$.

$$sf(\alpha \mathsf{U} \beta) = \Sigma_{B \subseteq Br(sf(\alpha))} \Sigma_{(e,f) \in B} (e \cap \overline{(\overline{\emptyset} \cdot \overline{e})})(f \cap sf(\beta))$$

This translation is not terribly efficient, hopefully a better one is yet to be found.

1.3.2 Truth checking a path as membership

We now consider the truth checking question for this logic, which has also been given the name of *path checking* [MS03]. Since we are dealing with infinite words, we restrict ourselves to ultimately periodic words uv^ω which can be described by giving the pair of finite words u and v as input.

Problem 14 (LTL truth checking)
Instance: Two finite words u and v over the alphabet A and an LTL formula ϕ
Question: Does $uv^\omega \models \phi$ hold?
Complexity: lower bound ALogTime*, upper bound* AC1(LogDCFL)

Since ALogTime lower bounds the evaluation of propositional logic formulas, it also serves as a lower bound for LTL. A polynomial time dynamic programming algorithm taking time $O(|w||\phi|)$ for the problem is easy to see and is sketched below. An exciting recent

theoretical development was a more efficient parallel algorithm (for LTL truth checking over finite words) found by Kuhtz and Finkbeiner [KF09]. The complexity class mentioned above for the upper bound is defined using circuit families which use oracle gates: for an input of length n they construct a circuit of depth order $\log n$ which uses LogDCFL oracle gates. The class LogDCFL consists of problems which reduce in logarithmic space to membership in a deterministic context-free language. It is known that LogSpace $\subseteq$ LogDCFL $\subseteq$ AC1(LogDCFL) $\subseteq$ Ptime. Hopefully this can be extended to general LTL truth checking over infinite words as well.

The Ptime algorithm proceeds by successively, at all states, extending the labelling $\pi : N \to A$ to a new labelling $\widehat{\pi} : N \to B$, where $B = \wp(Sub(\phi))$ and $Sub(\phi)$ is the set of subformulas of ϕ. The number of subformulas of ϕ is linear in the size of ϕ. The algorithm starts with the smallest subformulas and works outwards towards ϕ.

The inductive construction can be described as follows. If the formula being tackled is

p**:** nothing needs to be done since the state s is already labelled if $p \in \pi(s)$.

$\alpha \wedge \beta$**:** label s with $\alpha \wedge \beta$ if s is already labelled with α and β.

$\neg\alpha$**:** label s with $\neg\alpha$ if s is not already labelled with α.

$\mathbf{X}\alpha$**:** label s with $\mathsf{X}\alpha$ if its successor is already labelled with α.

$\alpha\mathbf{U}\beta$**:** do a backward pass through the word, labelling states already labelled with β with $\alpha\mathsf{U}\beta$, and propagating this formula backwards so long as a state is labelled with α and its successor is labelled with $\alpha\mathsf{U}\beta$.

1.4 The model checking problem for computation tree logic

The motivation for the logic CTL defined below came from the notion of "safety" and "liveness" properties, due to Lamport [Lam80]. Two groups working independently, Emerson and Clarke in the US [EC82], and Queille and Sifakis in France [QS82], came up with algorithms for checking temporal logic properties of a system, described as a tree. (We used their algorithm in the previous section, restricted to a word.) CTL is due to Emerson and Clarke and, like Queille and Sifakis's logic, it was based on UB, another of Manna and Pnueli's temporal logics, developed with Ben-Ari [BMP83]. The advantage of these historically first approaches is that the model checking algorithm continues to be linear time $O(|M||\phi|)$.

$\alpha ::= p \in Prop \mid \neg\alpha \mid \alpha \vee \beta \mid \mathsf{EX}\alpha \mid \mathsf{EG}\alpha \mid \mathsf{E}[\alpha\mathsf{U}\beta]$

Similar to before, we define $\mathsf{EF}\alpha = \mathsf{E}[true\mathsf{U}\alpha]$ (these are called weak liveness properties) and its dual $\mathsf{AG}\neg\alpha = \neg\mathsf{EF}\alpha$ (safety properties, which say that no "bad" α happens in any run). $\mathsf{EG}\neg\alpha$ is called a weak safety property and the dual liveness property $\mathsf{AF}\alpha = \neg\mathsf{EG}\neg\alpha$ says

that some "good" α happens in every run. We also define $\mathsf{A}[\alpha\mathsf{U}\beta] = \neg(\mathsf{EG}\neg\beta \vee \mathsf{E}[\neg\beta\mathsf{U}\neg\alpha \wedge \neg\beta])$.

Again we refer to Huth and Ryan's book [HR00] for motivating examples. The idea is that a system is modelled as a tree $\tau = (T, t_0, \rightarrow, L)$ of all its runs, together with a state labelling function. $(T, \rightarrow)$ is the tree, t_0 the root and $L : T \rightarrow A$ is the labelling function which specifies which atomic propositions hold at the tree nodes. This semantics is what gives the name computation tree logic. The definition of satisfaction formalizes the intuition.

$\tau, t \models p$ iff $p \in L(t)$

$\tau, t \models \mathsf{EX}\alpha$ iff for some t' such that $t \rightarrow t'$, we have $\tau, t' \models \alpha$

$\tau, t \models \mathsf{EG}\alpha$ iff for some path $\pi = t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$ where $t = t_1$, $\pi, 1 \models \mathsf{G}\alpha$

$\tau, t \models \mathsf{E}[\alpha\mathsf{U}\beta]$ iff for some path $\pi = t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$ where $t = t_1$, $\pi, 1 \models \alpha\mathsf{U}\beta$

From its definition, you can work out that $\mathsf{A}[\alpha\mathsf{U}\beta]$ checks the until property along *all* paths from the root. It should be clear that Lamport's criteria of describing safety and liveness properties of a system are met by this logic.

But there is a question: how is a tree with infinite paths to be presented as input to an algorithm? The solution is to again restrict ourselves to regular infinite trees, which have finitely many subtrees upto isomorphism. Such trees can be described by a **transition system** $M = (S, s_1, \rightarrow, L)$ (also called a Kripke structure), where S is a finite set of states with a labelling function $L : S \rightarrow A$ and s_1 the root state from which infinite paths ("runs") of the system start off. The transition system M with start state s_1 *unfolds* into a tree model $Unf(M)$ with $root(M)$ as its root.

Problem 15 (MC(CTL), CTL model checking)
Instance: A transition system M over the alphabet A and a CTL formula ϕ
Question: Does $Unf(M), root(M) \models \phi$ hold?
Complexity: PTIME

Observe that this is an alternate way of framing the question of checking whether a given infinite tree (represented as a finite structure) is a member of a language of infinite trees (described again by a finite logical formula). So this could be called the truth checking, or membership, problem for CTL. But by now the name *model checking* is firmly established for this problem when the model is given as a transition system, and we use the popular terminology.

To solve the problem we look at extending the algorithm for truth checking a word, and this can be done with minor changes. As before, the states of M are labelled by the alphabet B of sets of subformulas of a formula ϕ. Tarjan's algorithm is used to find strongly connected components (SCCs) for EG subformulas, thus maintaining linear time.

If the formula being tackled is

$\mathsf{EX}\alpha$: label s with $\mathsf{EX}\alpha$ if it has a successor which is already labelled with α

$\mathsf{E}[\alpha\mathsf{U}\beta]$: label states already labelled with β with $\mathsf{E}[\alpha\mathsf{U}\beta]$, and propagate this formula using a backwards breadth-first search, that is, so long as a state is labelled with α and has a successor state labelled with $\mathsf{E}[\alpha\mathsf{U}\beta]$

EGα: to deal with this case efficiently, restrict the graph to states satisfying α (consider other states and their transitions to be deleted), find the maximal SCCs, and use backwards breadth-first search to find states which can reach an SCC, all of these states are labelled with $\mathsf{EG}\alpha$

While CTL model checking is in $O(|S||\phi|)$ time, one can also verify that it has a PTIME lower bound, by a reduction from the monotone (layered) circuit value problem MCVP. For example, if the layers begin with "and"s and end with "or"s, the CTL formula (which does not even use until) $\mathsf{AXEX} \ldots \mathsf{AXEX}true$ holds iff the circuit evaluates to true. Thus membership in a tree language, or model checking a transition system against a formula, is a PTIME-complete problem.

Inclusion of tree languages is a much harder problem, and we refer to Chapter **??** by Christof Löding for details. The satisfiability question of CTL (which can also be seen as nonemptiness of the tree language) is correspondingly hard, as was shown by Fischer and Ladner [FL79].

Problem 16 (SAT(CTL), CTL satisfiability)
Instance: A CTL formula ϕ
Question: Is there a tree model for ϕ?
Complexity: EXPTIME

The equivalences below can be used to handle boolean equivalences over paths, but we cannot reduce nested path formulas.

$\mathsf{E}[\neg\mathsf{X}\alpha] \equiv \mathsf{EX}\neg\alpha,\ \mathsf{E}[\neg(\alpha\mathsf{U}\beta)] \equiv \mathsf{EG}\neg\beta \vee \mathsf{E}[\neg\beta\mathsf{U}\neg\alpha \wedge \neg\beta]$

$\mathsf{E}[(\alpha_1\mathsf{U}\beta_1) \wedge (\alpha_2\mathsf{U}\beta_2)] \equiv \mathsf{E}[(\alpha_1 \wedge \alpha_2)\mathsf{U}(\beta_1 \wedge \mathsf{E}[\alpha_2\mathsf{U}\beta_2])] \vee \mathsf{E}[(\alpha_1 \wedge \alpha_2)\mathsf{U}(\beta_2 \wedge \mathsf{E}[\alpha_1\mathsf{U}\beta_1])]$

The "formula" $\mathsf{E}[\mathsf{GF}p]$, which intuitively says that there is a path along which p is infinitely often true, does not have any CTL equivalent at all.

1.5 Model checking temporal logics

Once algorithms for CTL model checking were developed, those for checking LTL properties were not far behind. Instead of interpreting a transition system $M = (S, s_1, \rightarrow, L)$ by its tree unfolding, one could consider $Lang(M)$, the set of words which are runs from its initial state, and ask whether all these are models of an LTL formula. We return to our LTL notation, and extend it momentarily:

$\alpha ::= p \in Prop \mid \neg\alpha \mid \alpha \vee \beta \mid \mathsf{X}\alpha \mid \alpha\mathsf{U}\beta$

$M, s \models \mathsf{A}[\alpha]$ iff for all paths $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ where $s = s_1$, we have $\pi, 1 \models \alpha$

Manna and Pnueli took the decision to interpret *every* LTL formula using such a $\forall$path-semantics, hence we do not explicitly write the $\mathsf{A}[\ ]$-operator and continue to use the old LTL syntax.

Problem 17 ($\mathrm{MC}^{\forall}$(LTL), LTL model checking)
Instance: A transition system M and an LTL formula ϕ

Question: Does $Unf(M), root(M) \models \mathsf{A}[\phi]$ *hold?*
Question: Alternately, does $\pi, 1 \models \phi$ *hold for every* $\pi \in Lang(M)$*?*
Complexity: PSPACE

The basic strategy for the LTL model checking problem is different from that for CTL: it changes from checking membership of a tree (presented finitely) in a language of trees (described by a formula) to checking inclusion of a set of words $Lang(M)$ (implicitly presented by the transition system M, assuming all states are accepting) in another set of words (described by a formula).

Historically, a key rôle here was played by dynamic logic (PDL), invented by Pratt [Pra76] as a generalization of traditional modal logic with program-indexed modalities. PDL was proved decidable by a filtration argument of Fischer and Ladner [FL79]. A *tour de force* completeness-cum-decidability proof of PDL by Kozen and Parikh [KP81] and the tight "tableau" construction of Pratt [Pra80] laid the seeds of the "formula automaton" construction of Vardi and Wolper [VW94], which is today the standard way of implementing LTL model checking algorithms. We only sketch this theory, more details can be found in Chapter **??** by Stéphane Demri and Paul Gastin.

We first describe how to construct a formula automaton, also called a tableau, for a formula ϕ, which is exponential in the size of ϕ, with the property that the language accepted by this automaton is precisely the set of models of ϕ.

Since a formula automaton is a finite automaton (over infinite words) and the given transition system is also a finite automaton, we can perform a "product" construction to recognize the intersection of the accepted paths. Since we are concerned with infinite paths, the product is of automata over infinite words (here we witness the return of the automata of Büchi [Bü62]) rather than the usual one of automata over finite words. Chapter **??** by Madhavan Mukund describes this construction.

The model checking algorithm consists of taking the product of the automaton for the *complement* formula $\neg\phi$ with M, and checking the product for *nonemptiness*. If there is a path in the product of this kind, we exhibit this as a counterexample to M satisfying ϕ; if there isn't, we declare that M is a model for ϕ. In effect, we have checked whether the language of M is included in the "language" of ϕ by checking the emptiness of $L(M) \cap L(\neg\phi) = L(M) \cap \overline{L(\phi)}$.

The states of the formula automaton are all subsets a of $Sub(\phi)$ which could be potential states of the transition system, in the following sense:

Consistent: For every subformula ψ, both ψ and $\neg\psi$ are not in a.

Downward saturated: If $\alpha \vee \beta$ is in a, one of α or β is in a.

Maximal: For every subformula ψ, either ψ or $\neg\psi$ is in a.

Let $\widehat{a}$ be the conjunction of formulas in a. The transitions are defined so that $\widehat{a} \wedge \mathsf{X}\widehat{b}$ is consistent when there is a transition from a to b. Since there are exponentially many subsets of $Sub(\phi)$, which is itself linear in the size of ϕ, it is possible to give an exponential

$|M|.2^{O(|\phi|)}$ time model checking algorithm. Using the fact that nonemptiness of the language of a finite state Büchi automaton can be checked in nondeterministic logarithmic space, we get an NPSPACE algorithm, which we know is the same as PSPACE, so that is an upper bound for the complexity of model checking LTL.

Before considering the lower bound, we introduce another problem:

Problem 18 (SAT(LTL), LTL satisfiability)
Instance: An LTL formula ϕ
Question: Is there a word model for ϕ?
Complexity: PSPACE

PSPACE is a lower bound for both model checking and satisfiability, as was shown by Sistla and Clarke [SC85]. We refer to their paper, or the survey by Schnoebelen [Sch03], for proofs. Here is an idea of how a reduction from the valid linear space computation problem to either model checking or satisfiability can be done.

A letter of the word representing a configuration c_i can take as many possible values as the size of the alphabet B, which is represented by transitions branching to that many states where a proposition coding that letter is true, and then coming together again, forming a diamond-shaped structure. The whole word c_i of length n can be coded as that many "diamonds" in sequence. After the last "diamond" we have a transition cycling back to the beginning. The LTL formula now describes the initial configuration, and specifies using $2n$ nested X operators, how a configuration c_i can change to the next one c_{i+1}, and asserts using an F operator the existence of a final configuration. A model of this formula (notice that it does not even use the until operaor) describes a valid computation and conversely, every valid computation is described by a model of this formula.

Sistla and Clarke showed that if an LTL formula is satisfiable, it is satisfiable in a word uv^ω where u and v are at most exponential in the size of the formula [SC85]. Guessing this model and model checking ϕ along it would give an NEXPTIME algorithm. But as in the case of our algorithm for DGAP, instead of guessing the whole model, it can be guessed and verified on the fly. This gives an NPSPACE algorithm, and by Savitch's theorem the problem is in PSPACE.

1.5.1 CTL*

Recall that the formula $\mathsf{E}[\mathsf{GF}p]$ says there is a path along which p is infinitely often true. This does not have an equivalent in the LTL $\forall$-semantics either. This motivated the development of the logic CTL*.

$\alpha ::= p \in Prop \mid \neg\alpha \mid \alpha \vee \beta \mid \mathsf{X}\alpha \mid \alpha\mathsf{U}\beta$

$\phi ::= \mathsf{A}[\alpha] \mid \neg\phi \mid \phi \vee \psi$

Now we can define $\mathsf{E}[\alpha] = \neg\mathsf{A}[\neg\alpha]$, yielding the semantics:

$M, s \models \mathsf{E}[\alpha]$ iff for some path $\pi = s_1 \to s_2 \to s_3 \to \ldots$ where $s = s_1$, we have $\pi, 1 \models \alpha$

The PSPACE model checking complexity of LTL can be lifted to CTL* as well. Chapter **??** by Demri and Gastin has the details.

1.5.2 Model checking ITL

It is possible to describe paths much more succinctly. We return to the interval logic defined in Section 1.2.5 and recall its syntax and semantics.

$\phi ::= \lceil\!\lceil B \rceil\!\rceil, B \subseteq A \mid pt \mid \neg\phi \mid \phi \vee \psi \mid \phi;\psi$

$\pi, [b,e] \models \lceil\!\lceil B \rceil\!\rceil$ iff for all $m : b \leq m \leq e : \pi(m) \in B$

$\pi, [b,e] \models pt$ iff $b = e$

$\pi, [b,e] \models \phi;\psi$ iff $\exists m : b \leq m \leq e : \pi, [b,m] \models \phi$ and $\pi, [m,e] \models \psi$

Problem 19 (MC$^\forall$(ITL), ITL model checking)
Instance: A transition system M and an ITL formula ϕ
Question: Does $\pi, 1 \models \phi$ hold for every $\pi \in Lang(M)$?
Complexity: above ELEMENTARY

ITL formulas are succinct compared to LTL formulas, although the class of languages covered is the same. Theoretically, this power derives from the negation $\neg(\phi;\psi)$ of a chop formula. As can be seen from the semantics, this ranges over all possible ways of breaking up an interval into subintervals. In fact, this combination means that we can obtain lower bounds from the inclusion problem for starfree expressions, and we have that MC$^\forall$(ITL), model checking an ITL formula against (all runs of) a transition system, has an ELEMENTARY lower bound. It is difficult to find natural properties which require several nestings of such operators and model checkers for ITL [Pan01] work reasonably well in practice.

1.5.3 Unambiguous interval logic

Recently, motivated by the good performance of an ITL model checker [KP05], with Pandya and Shah we considered a logic UITL [LPS08] where the chops are "marked" in a deterministic manner. In the syntax below, a stands for a letter and B for a set of letters from the alphabet.

$\phi ::= \lceil\!\lceil B \rceil\!\rceil \mid pt \mid \neg\phi \mid \phi \vee \psi \mid \phi F_a \psi \mid \phi L_a \psi \mid \oplus\phi \mid \ominus\phi$

The operators F_a and L_a can be read "first a" and "last a". They chop an interval into two subintervals at a point determined by either the first or the last occurrence of the specified letter a in the interval (provided it exists). The last two operators provide successor and predecessor modalities at the interval level.

$$
\begin{array}{l}
\pi, [b,e] \models \phi F_a \psi \text{ iff for some } m : b \leq m \leq e. \quad \pi(m) = a \text{ and} \\
\qquad (\text{for all } l : b \leq l < m.\ \pi(l) \neq a) \text{ and} \\
\qquad \pi, [b,m] \models \phi \text{ and } \pi, [m,e] \models \psi \\
\pi, [b,e] \models \phi L_a \psi \text{ iff for some } m : b \leq m \leq e. \quad \pi(m) = a \text{ and} \\
\qquad (\text{for all } l : m < l \leq j.\ \pi(l) \neq a) \text{ and} \\
\qquad \pi, [b,m] \models \phi \text{ and } \pi, [m,e] \models \psi \\
\pi, [b,e] \models \oplus\phi \text{ iff } b < e \text{ and } w, [b+1,e] \models \phi \\
\pi, [b,e] \models \ominus\phi \text{ iff } b < e \text{ and } w, [b,e-1] \models \phi
\end{array}
$$

This logic is much less expressive than full ITL, and matches a fragment of LTL. Interestingly, the complexity of model checking comes down: checking a UITL formula ϕ against a word w is in $O(|w||\phi|^2)$ and also in the complexity class LogDCFL (which we saw earlier, recall that LogDCFL $\subseteq$ Ptime), while checking whether all runs of a transition system satisfy the formula is in NPtime. We refer to the paper [LPS08] for some of the details.

Problem 20 (UITL truth checking)
Instance: A word w and a UITL formula ϕ
Question: Does $w \models \phi$ hold?
Complexity: lower bound ALogTime*, upper bound* LogDCFL

Problem 21 (MC$^\forall$(UITL), UITL model checking)
Instance: A transition system M and a UITL formula ϕ
Question: Does $\pi, 1 \models \phi$ hold for every $\pi \in Lang(M)$?
Complexity: NPtime

Problem 22 (SAT(UITL), UITL satisfiability)
Instance: A UITL formula ϕ
Question: Does ϕ have a word model?
Complexity: NPtime

1.6 Reading ahead

In this chapter we considered the simple case where the system model is finite. The more difficult case when the model is infinite is considered in the articles by Javier Esparza (Chapter **??**) and by Wolfgang Thomas (Chapter **??**) in this volume.

The alphabet we consider for labelling the transitions is just a finite set. A more distributed structure on the alphabet is considered by Madhavan Mukund in Chapter **??**, whereas the representation of abstract data values is in Chapter **??** by M. Amaldev and R. Ramanujam.

As should be clear by now, verification theory using automata is developing in many directions. Our article is conceptually closer to the early theory of the 1980s and we have used the chapter in the *Handbook of Theoretical Computer Science* by Emerson [Eme90]. The material on model checking is covered in detail in the books by Huth and Ryan [HR00] and by Clarke, Grumberg and Peled [CGP99]. The book by Stirling [Sti01] uses an alternative treatment using a rich formalism called the modal μ-calculus, and reduces model checking to problems of finding the winner in a suitable game. The complexity results for most of these logics and more detailed fragments are surveyed in the long article by Schnoebelen [Sch03], and the journal article by Demri and Schnoebelen [DS02].

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[AFdR80] Krzysztof R. Apt, Nissim Francez and Willem-Paul de Roever. A proof system for communicating sequential processes, *ACM Trans. Prog. Lang. Syst.* 2:3, 359–385, Jul 1980.

[BMP83] Mordechai Ben-Ari, Zohar Manna and Amir Pnueli. The temporal logic of branching time, *Acta Inf.* 20, 207–226, 1983.

[Bü60] J. Richard Büchi. Weak second-order arithmetic and finite automata, *Z. Math. Logik Grundl. Math.* 6, 66–92, 1960.

[Bü62] J. Richard Büchi. On a decision method in restricted second-order arithmetic, *Proc. 1960 Congr. Logic, Methodology, Philosophy and Science*, Stanford (Ernest Nagel, Patrick Suppes and Alfred Tarski, eds.), Stanford Univ Press, 1–11, 1962.

[Bur74] Rod M. Burstall. Program proving as hand simulation with a little induction, *Proc. 6th IFIP Congress*, Stockholm (Jack L. Rosenfeld, ed.), North-Holland, 308–312, 1974.

[Buss87] Samuel R. Buss. The Boolean formula value problem is in ALogTime, *Proc. 19th Symp. Th. Comput.*, New York, ACM, 123–131, 1987.

[CGP99] Edmund M. Clarke Jr., Orna Grumberg and Doron Peled. *Model checking*, MIT Press, 1999.

[DS02] Stéphane Demri and Philippe Schnoebelen. The complexity of propositional linear temporal logic in simple cases, *Inf. Comput.* 174:1, 84–103, 2002.

[Elg61] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics, *Trans. AMS* 98, 21–52, 1961.

[EC82] E. Allen Emerson and Edmund M. Clarke Jr. Using branching time temporal logic to synthesize synchronization skeletons, *Sci. Comp. Program.* 2, 241–266, 1982.

[Eme90] E. Allen Emerson. Modal and temporal logics, in *Handbook of theoretical computer science* B, (Jan van Leeuwen, ed.), Elsevier, 995–1072, 1990.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs, *J. Comput. Syst. Sci.* 18:2, 194–211, 1979.

[Flo67] Robert W. Floyd. Assigning meanings to programs, *Mathematical aspects of computer science*, Providence (J.T. Schwartz, ed.), *Proc. Symp. Appl. Math.* 19, AMS, 19–32, 1967.

[Hoa69] C. Anthony R. Hoare. An axiomatic basis for computer programming, *Commun. ACM* 12:10, 576–580, Oct 1969.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in computer science: modelling and reasoning about computer systems*, Cambridge, 2000.

[JR91] Tao Jiang and Bala Ravikumar. A note on the space complexity of some decision problems for finite automata, *Inf. Proc. Lett.* 40:1, 25–31, 1991.

[KP05] S.N. Krishna and Paritosh K. Pandya. Modal strength reduction in quantified discrete duration calculus, *Proc. FSTTCS*, Hyderabad (R. Ramanujam and Sandeep Sen, eds.), *LNCS* 3821, 444–456, 2005.

[KP81] Dexter C. Kozen and Rohit J. Parikh. An elementary proof of the completeness of PDL, *Theoret. Comp. Sci.* 14, 113–118, 1981.

[KF09] Lars Kuhtz and Bernd Finkbeiner. LTL path checking is efficiently parallelizable, *Proc. 36th Int. Conf. Autom. Lang. Program., Part II*, Rhodes (Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas and Wolfgang Thomas, eds.), *LNCS* 5556, 235–246, 2009.

[Lam80] Leslie Lamport. 'Sometime' is sometimes 'not never', *Proc. 7th Symp. Princ. Program. Lang.*, Las Vegas, ACM, 174–185, 1980.

[LPS08] Kamal Lodaya, Paritosh K. Pandya and Simoni S. Shah. Marking the chops: an unambiguous temporal logic, *Proc. 5th IFIP TCS*, Milano (Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri and C.-H. Luke Ong, eds.), *IFIP Series* 273, Springer, 461–476, 2008.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: specification*, Springer, 1992.

[MS03] Nicolas Markey and Philippe Schnoebelen. Model checking a path, *Proc. 14th Concur*, Marseilles (Roberto M. Amadio and Denis Lugiez, eds.), *LNCS* 2761, 251–265, 2003.

[MS72] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space, *Proc. 13th Switch. Autom. Theory*, IEEE, 125–129, 1972.

[Mey75] Albert R. Meyer. Weak monadic second order theory of successor is not elementary recursive, *Proc. Logic Colloq.*, Boston (Rohit J. Parikh, ed.), *LNM* 453, 132–154, 1975.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes, *IEEE Trans. Softw. Engg.* SE-7:4, 417–426, Jul 1981.

[MM83] Ben C. Moszkowski and Zohar Manna. Reasoning in interval temporal logic, *Proc. Logics of programs*, Pittsburgh (Edmund M. Clarke Jr. and Dexter C. Kozen, eds.), *LNCS* 164, 371–382, 1983.

[Naur66] Peter Naur. Proof of algorithms by general snapshots, *BIT* 6:4, 310–316, Jul 1966.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I, *Acta Inf.* 6, 319–340, 1976.

[Pan01] Paritosh K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID: an automata theoretic approach, *Proc. RTTOOLS*, Aalborg, Aug 2001.

[Pet00] Holger Petersen. Decision problems for generalized regular expressions, *Proc. 2nd Descr. Compl. Aut. Gram. Related Str.*, London (CA), 22–29, 2000.

[Pnu77] Amir Pnueli. The temporal logic of programs, *Proc. 18th Found. Comp. Sci.*, Providence , IEEE, 46–57, 1977.

[PZ93] Amir Pnueli and Lenore D. Zuck. In and out of temporal logic, *Proc. 3rd LICS*, IEEE, 124–135, 1993.

[Pra76] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic, *Proc. 17th Found. Comp. Sci.*, Houston , IEEE, 109–121, 1976.

[Pra80] Vaughan R. Pratt. A near-optimal method for reasoning about action, *J. Comput. Syst. Sci.* 20:2, 231–254, 1980.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of computer systems in CESAR, *Proc. 5th Symp. Program.*, Torino (Mariangiola Dezani-Ciancaglini and Ugo Montanari, eds.) *LNCS* 137, 337–351, 1982.

[Sav70] Walter J. Savitch. Relationship between nondeterministic and deterministic tape classes, *J. Comp. Syst. Sci.* 4, 177–192, 1970.

[Sch03] Philippe Schnoebelen. The complexity of temporal logic model checking, *Proc. 4th Adv. Modal Log. '02*, Toulouse (Philippe Balbiani, Nobu-Yuki Suzuki, Frank Wolter and Michael Zakharyaschev, eds.), King's College, 393–436, 2003.

[Sch65] Marcel-Paul Schützenberger. On finite monoids having only trivial subgroups, *Inf. Contr.* 8:2, 190–194, 1965.

[SC85] A. Prasad Sistla and Edmund M. Clarke Jr. The complexity of propositional linear temporal logics, *J. ACM* 32:3, 733–749, 1985.

[Sou83] Neelam Soundararajan. Correctness proofs of CSP programs, *Theoret. Comp. Sci.* 24, 131–141, 1983.

[Sti01] Colin P. Stirling. *Modal and temporal properties of processes*, Springer, 2001.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time, *Proc. 5th Symp. Th. Comput.*, Austin, ACM, 1–9, 1973.

[Sto74] Larry J. Stockmeyer. *The complexity of decision problems in automata theory and logic*, PhD thesis, MIT, 1974.

[Tra61] Boris A. Trakhtenbrot. Finite automata and the logic of monadic predicates, *Dokl. Akad. Nauk SSSR* 140, 326–329, 1961.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations, *Inf. Comput.* 115:1, 1–37, 1994.