1

The Complexity of Temporal Logic Model Checking

Ph. Schnoebelen[1]

1 Introduction

Temporal logic. Logical formalisms for reasoning about time and the timing of events appear in several fields: physics, philosophy, linguistics, etc. Not surprisingly, they also appear in computer science, a field where logic is ubiquitous. Here temporal logics are used in automated reasoning, in planning, in semantics of programming languages, in artificial intelligence, etc.

There is one area of computer science where temporal logic has been unusually successful: the specification and verification of programs and systems, an area we shall just call "*programming*" for simplicity. In today's curricula, thousands of programmers first learn about temporal logic in a course on model checking!

Temporal logic and programming. Twenty five years ago, Pnueli identified temporal logic as a very convenient formal language in which to state, and reason about, the behavioral properties of parallel programs and more generally reactive systems [Pnu77, Pnu81]. Indeed, correctness for these systems typically involves reasoning upon related events at *different moments* of a system execution [OL82]. Furthermore, when it comes to liveness properties, the expected behavior of reactive systems cannot be stated as a static property, or as an invariant one. Finally, temporal logic is well suited to expressing the whole variety of fairness properties that play such a prominent role in distributed systems [Fra86].

For these applications, one usually restricts oneself to *propositional* temporal logic: on the one hand, this does not appear to be a severe limitation in practice, and on the other hand, this restriction allows decision procedures for validity and entailment, so that, at least in principle, the above-mentioned reasoning can be automated.

Model checking. Generally speaking, model checking is the algorithmic verification that a given logic formula holds in a given structure (the *model*

[1] Laboratoire Spécification & Vérification (LSV), ENS de Cachan & CNRS. Email: phs@lsv.ens-cachan.fr.

Advances in Modal Logic, Volume 4, 1–44.

that one *checks*). This concept is meaningful for most logics and classes of models but, historically, it was developed in the context of temporal logic formulae for finite Kripke structures, called "*temporal logic model checking*" in this survey. Temporal logic model checking has been a very active field of research for the last two decades because of its important applications in verification (see e.g. [CW96]).

A huge effort has been, and is being, devoted to the development of smarter and better model checking software tools, known as *model checkers*, that can verify ever larger models and deal with a wide variety of extended frameworks (real-time systems, stochastic systems, open systems, etc.). We refer to [Hol91, Kur95, CGP99, BBF$^+$01] for more details on the practical aspects of model checking.

Model checking for modal logicians. Temporal logic can be seen as some brand of modal logic, but it seems fair to say that model checking is not a popular problem in the modal logic community: for example it is not mentioned in standard textbooks such as [Ben85, HC96, BRV01]. This is probably because model checking is too trivial a problem for the standard modal logics based on immediate successors (it is easy even for *PDL*, see [CS93]) and only becomes interesting when richer temporal logics are considered. However, standard texts on temporal logic aimed at logicians (e.g. [Ben89, Sti92, GRF00]) just briefly mention that model checking is possible and do not really deal with the computational issues involved. A recent exception is [BS01] where a few pages are devoted to model checking for μ-calculi since, quoting [BS01, p. 315]:

> *Decidability and axiomatization are standard questions for logicians; but for the practitioner, the important question is model-checking.*

The complexity of model checking. Once decidability has been proved, the next basic problem in the theory of model checking is measuring its *complexity* [1].

The point is that, when the precise complexity of some computational problem has been established, it can be said that the *optimal algorithm* for the problem has been identified and proved optimal. Here "optimal" has a precise meaning: one only considers what computing resources are *asymptotically* necessary and sufficient for solving all instances, including the hardest ones (i.e., *in the worst case*). These simplifications and abstractions about the cost of algorithms lead to a surprisingly powerful theory

[1] We assume the reader has some basic knowledge of the theoretical framework of computational complexity, and refer him to standard texts like [Sto87, Joh90, Pap94] for more motivations and details.

that has been applied successfully in a huge number of fields.

In the field of temporal logic model checking, this research program led to a clearer understanding of why model checking works so well (or does not work). It also added a new dimension on which to compare different logical formalisms (alongside the more classical dimensions of expressive power and complexity of validity).

The goals of this survey. We present the main results on the complexity of model checking and the underlying algorithmic ideas. This covers the main temporal logics encountered in the programming literature: *LTL* (from [GPSS80]), *CTL* (from [CE81]), CTL^* (from [EH86]), their fragments, and their extensions with past-time modalities [2]. The presentation is mainly focused on complexity results, not on the usefulness, or elegance, or expressive power, of the temporal logics we consider [3]. However, when complexity of model checking is concerned, we (try to) explain the ideas behind the algorithms and hardness proofs, in the hope that these techniques can be useful in other subfields of modal logic.

Outline of the paper. We start with the basic concepts and definitions (temporal logics, their models, the model checking problem) in Section 2. Then Section 3 gives the main results on model checking for our three logics, before we consider fragments (in Section 4) and past-time extensions (in Section 5). Finally, we discuss more advanced questions (parameterized view of complexity in Section 6, and complexity of symbolic model checking in Section 7) that help bridge the gap between complexity theory and actual practice.

2 Basic notions

2.1 Temporal modalities

Temporal logic [Pri67] is a brand of modal logic tailored to temporal reasoning, i.e. reasoning with modalities like “sometimes”, “now”, “often”, “later”, “while”, “always”, “inevitably”, etc.

Temporal logics are usually interpreted in modal structures where the nodes (the modal *worlds*) are positions in time, often called *instants*. These need not be just points but can be, for example, time intervals (periods).

Usually, the modal relation relates two positions when the second lies in

[2] We decided to omit mentioning μ-calculi because, even though they are popular in the programming community, we think they are less temporal logics than languages in which one can define temporal logics (much like monadic second-order logics). The interested reader may consult [AN01].

[3] We refer the reader to standard texts, like [Eme90, MP92], for motivation and examples.

the future of the first. This provides a model for *qualitative* aspects of time, dealing with "before and after". More elaborate notions like, for example, metrics on time (durations) could be modeled as well but this survey limits itself to the simpler frameworks.

Thus a time frame is usually taken to be an ordered set $\langle T, \leq_T\rangle$, with $t \leq_T t'$ denoting that instant $t \in T$ precedes instant $t' \in T$ in time.

2.2 Linear-time and branching-time

A major distinction between temporal logics is whether they see time as *linear* or *branching* [4]. This is reflected in the classes of time frames they consider: linear orderings or trees.

In linear-time logics, all instants are linearly ordered from past to future and there is only one possible future: future is determined. In programming, this viewpoint is convenient for deterministic programs. For nondeterministic programs, the linear-time viewpoint applies to the runs of the system: any given run determines a single future. The nondeterminism of the system is taken into account at a different level, by considering all the runs.

In branching-time logics, the future is not determined and any given instant may have several distinct immediate successors [5], hence the tree-like structure of the frame. This viewpoint is probably the more appropriate when it comes to nondeterministic systems [Mil89, Gla01] but linear-time is often preferred for its simplicity, both notational (see below) and conceptual.

2.3 Kripke structures

A feature of model checking that explains its successes is that it mostly deals with finite structures displaying infinite behaviors: the finiteness of the structures entails the (efficient) decidability of model checking, while the non-finiteness of the behaviors allows one to model interesting situations.

Here it is crucial to distinguish between the *computational structure* and the *behavioral structure*.

The computational structure is a model of the program at hand, describing its possible configurations and the possible steps between them. Even if one only considers *finite* computational structures, it is still possible to model interesting programming problems [6], as the last twenty years of model

[4] Not all structures for time fall into the linear or the branching category (see e.g. [Wol89]) but these certainly are the two most often used in the programming literature.

[5] All through this survey we assume time is discrete.

[6] This is the case of *protocols*, where several small finite state machines interact in

checking have demonstrated.

The behavioral structure is a model of the behavior of the system modeled by a computational structure. It is usually infinite (systems are not supposed to terminate) but displays some regularity since it is obtained from the computational structure by some kind of traversal or unfolding.

Temporal logic is used to reason about the behavioral structure, where there is a clear notion of before and after along the runs of the system, in particular between different instants that correspond to a same, recurring, configuration. In summary, the temporal structure for our temporal logics is the behavioral structure, and the computational structure is just a finite-state model of the internal architecture of the system, from which the behavioral structure is derived by some kind of operational semantics.

With this in mind, it is unfortunate, and confusing for modal logicians, that computational structures are called *Kripke structures* in the model checking community! [7] This usage is so widespread that we follow it in the rest of this survey, and hope that the previous paragraph is sufficient warning against possible misunderstandings.

Formally, given a set $AP = \{P_1, P_2, \ldots\}$ of *atomic propositions*, a *Kripke structure* over AP is a tuple $S = \langle Q, R, l, I \rangle$ where

- $Q = \{q, r, s, \ldots\}$ is a set of *states* (the configurations of the system).
- $R \subseteq Q \times Q$ is a *transition relation* (the possible steps). For simplification purposes, we require that R is *total*, i.e. for any $q \in Q$ there is at least one q' s.t. $q \, R \, q'$.
- $l : Q \to 2^{AP}$ is a *labeling* of states with propositions. $P \in l(q)$ encodes the fact that P holds in state q.
- $I \subseteq Q$ is a non-empty set of *initial configurations*. Often I is a singleton and we just write q_I for the initial state.

We say S is a *finite* Kripke structure when Q is finite.

tricky combinatorial ways, of *reactive systems* where the system under study reacts to the stimuli of its environment and where temporal logic can state assumptions about the environment, of *hardware circuits* where finite-state gates are combined, and even of arbitrary *programs* after some abstraction (usually on their variables) has made them finite-state.

[7] Admittedly, it can be argued that Kripke structures are *bona fide* temporal structures for state-based branching-time pure-future logics like *CTL*. We do not favor this viewpoint since it leads to allow cycles in time.

2.4 Behaviors

A *path* through S is a sequence $q_0, q_1, q_2, \ldots$ of states s.t. $q_i \, R \, q_{i+1}$ for $i = 0, 1, \ldots$ A path may be finite or infinite. A *fullpath* is a maximal path and a *run* of S is a fullpath that starts from an initial state.

We use π, π', etc. to denote fullpaths and write $\Pi_S(q)$, or just $\Pi(q)$, for the set of fullpaths that start from some $q \in Q$. Then $\Pi(S) \stackrel{\text{def}}{=} \bigcup_{q \in I} \Pi(q)$ is the set of runs of S. A run is a possible behavior of the Kripke structure, and $\Pi(S)$ is the "linear-time behavior" of S. Note that, since we only allowed Kripke structures with a total R, all fullpaths are infinite (which is a welcome simplification).

For a state $q \in Q$ of S, the *tree* rooted at $q \in Q$ is the infinite tree $T_S(q)$, often simply denoted $T(q)$, obtained by unfolding S from q (formally, the nodes of $T(q)$ are the finite paths starting from q ordered by the prefix relation). Then $T(S) \stackrel{\text{def}}{=} \{T(q) \mid q \in I\}$ is the set of *computation trees* of S. A tree $T(q)$ gives the full branching structure of the behaviors issued from q, and $T(S)$ is the "branching-time behavior" of S.

Figure 1 displays an example of a simple Kripke structure S_{exm} with its runs $\Pi(S_{\text{exm}})$ and its computation trees $T(S_{\text{exm}})$. In this figure, time flows from left to right along the runs and the computation trees.

The example uses a set $AP = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ of atomic propositions. S_{exm} has $Q = \{q_1, q_2, q_3, q_4\}$, $I = \{q_1, q_2\}$ (indicated by the incoming arrows) and l given by $l(q_1) = \{\mathtt{a}\}$, $l(q_2) = \emptyset$, $l(q_3) = \{\mathtt{b}, \mathtt{e}\}$, and $l(q_4) = \{\mathtt{c}\}$. The transitions in R are all the directed edges between states.

The runs and computation trees of a Kripke structure S are structures collecting states of S. This definition is convenient for algorithms (as we see later). However, from a semantical viewpoint, runs and computation trees are considered up-to isomorphism: what particular state of S appears at some position is irrelevant, only the labeling with propositions from AP and the ordering relation between positions are meaningful. This is the reason why Figure 1 does not carry state names in $T(S_{\text{exm}})$ and $\Pi(S_{\text{exm}})$.

Remark 2.1 The dashed arrow in Figure 1 emphasizes the fact that the set $\Pi(S)$ of can be derived from $T(S)$. Since state names have been forgotten, it is not possible in general to reconstruct $T(S)$ from $\Pi(S)$: the branching-time semantics of a system provides more information about its behavior than the linear-time semantics does [Gla01]. $\square$

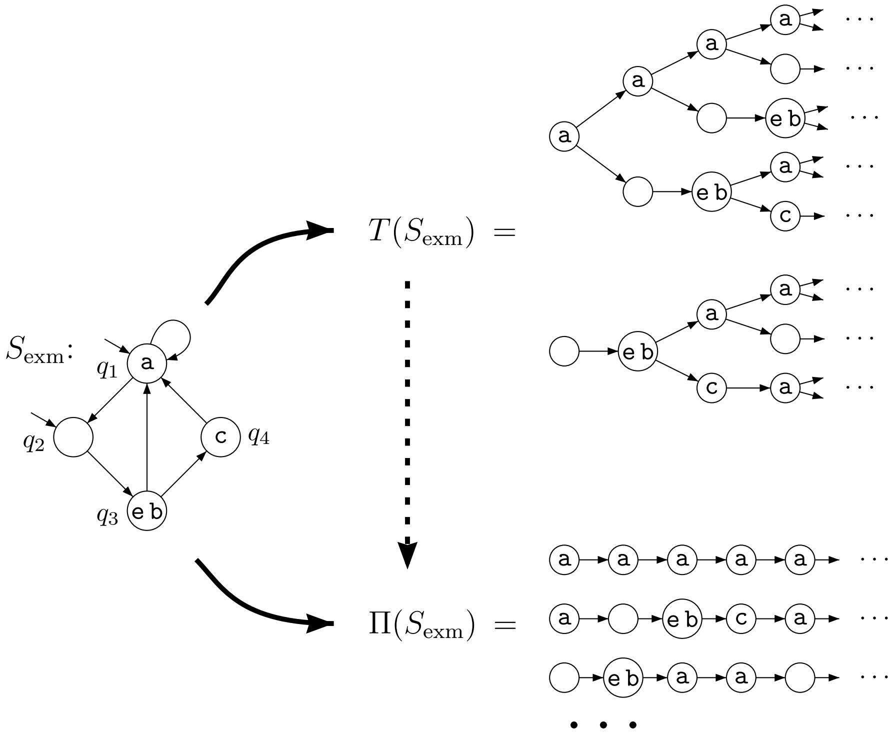

Figure 1. Branching-time and linear-time behaviors of S_{exm}

2.5 Three temporal logics: *LTL*, *CTL**, and *CTL*

2.5.1 *LTL*

LTL, for *Linear Temporal Logic*, is the temporal logic with Until and Next interpreted over runs, i.e. over any linearly ordered structure of type ω.

Syntax. Assuming a set $AP = \{P_1, P_2, \ldots\}$ of atomic propositions, the set of *LTL* formulae is given by the following abstract grammar:

$$\varphi, \psi \ ::= \ \varphi \, \mathsf{U} \, \psi \mid \mathsf{X} \, \varphi \mid \varphi \wedge \psi \mid \neg \varphi \mid P_1 \mid P_2 \mid \cdots \qquad (\textit{LTL} \text{ syntax})$$

Other Boolean connectives $\top$, $\bot$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$ and $\varphi \Leftrightarrow \psi$ are defined via the usual abbreviations. A formula like $\mathsf{X}\,\texttt{rain}$ reads "*it will rain (just next)*", while $\texttt{happy}\ \mathsf{U}\ \texttt{rain}$ reads "*(I) will be happy until it (eventually) rains*". The classical temporal modalities F ("sometimes in the future") and G ("always in the future") are obtained by

$$\mathsf{F}\,\varphi \ \equiv \ \top \, \mathsf{U} \, \varphi, \qquad\qquad \mathsf{G}\,\varphi \ \equiv \ \neg \mathsf{F} \neg \varphi. \tag{1}$$

Semantics. Formally, a *run* π is an ω-sequence $\sigma_\pi = s_0, s_1, s_2, \ldots$ of *states* with a labeling $l_\pi : \{s_0, s_1, \ldots\} \to 2^{AP}$ with propositions from AP. One defines when an *LTL* formula φ holds at position i of π, written $\pi, i \models \varphi$, by induction over the structure of φ:

$$\pi, i \models \varphi \mathsf{U} \psi \overset{\text{def}}{\Leftrightarrow} \exists j \geq i \text{ s.t. } \begin{cases} \pi, j \models \psi \\ \text{and} \\ \pi, k \models \varphi \text{ for all } i \leq k < j, \end{cases} \tag{S1}$$

$$\pi, i \models \mathsf{X}\,\varphi \overset{\text{def}}{\Leftrightarrow} \pi, i+1 \models \varphi, \tag{S2}$$

$$\pi, i \models \varphi \wedge \psi \overset{\text{def}}{\Leftrightarrow} \pi, i \models \varphi \text{ and } \pi, i \models \psi, \tag{S3}$$

$$\pi, i \models \neg\varphi \overset{\text{def}}{\Leftrightarrow} \pi, i \not\models \varphi, \tag{S4}$$

$$\pi, i \models P \overset{\text{def}}{\Leftrightarrow} P \in l_\pi(s_i) \qquad (\text{for } P \in AP). \tag{S5}$$

Observe that *LTL* is a *future-only* logic, i.e. a logic where whether $\pi, i \models \varphi$ only depends on the future $s_i, s_{i+1}, s_{i+2}, \ldots$ of the current situation.

Remark 2.2 Our definition assumes a *reflexive* U (and F), where the present is part of the future. This is standard in programming, but some works in temporal logic consider a strict, irreflexive, U_{irr} (that would be equivalent to our $\mathsf{X}\,\mathsf{U}$) [Kam68]. The irreflexive U_{irr} strives for minimality: it is a smart way of encoding both U and X in a single modality. The reflexive U is easier to use when writing real formulae, and allows considering the stutter-insensitive fragment of *LTL* formulae that do not use X [Lam83]. $\square$

Remark 2.3 *LTL* being future-only, one often finds in the literature an equivalent definition for its semantics, where the pair π, i is replaced by the i-th suffix of π (seen as word).

There are two reasons why we did not follow that style of definitions and notations in this survey. Firstly, we preferred be as close as possible to the standard semantics of modal logics, where formulae are evaluated at a position in a structure, and where modalities refer to other positions in the same structure. Secondly, we want our definitions to be easily adapted when we consider logics with past-time modalities in Section 5. $\square$

Remark 2.4 *LTL* being future-only, one often finds in the literature an equivalent definition for its semantics, where the pair π, i is replaced by the i-th suffix of π (seen as word).

There are two reasons why we did not follow that style of definitions and notations in this survey. Firstly, we preferred be as close as possible to the standard semantics of modal logics, where formulae are evaluated at a position in a structure, and where modalities refer to other positions in the same structure. Secondly, we want our definitions to be easily adapted when we consider logics with past-time modalities in Section 5. $\square$

The fundamental semantical definition leads to derived notions: we say that *a run* π *satisfies* φ, written $\pi \models \varphi$, when $\pi, 0 \models \varphi$. Furthermore, and since runs often come from a Kripke structure S, we say that S *satisfies* φ, written $S \models \varphi$, when $\pi \models \varphi$ for all $\pi \in \Pi(S)$, i.e. when φ holds in all runs issued from initial states of S. Finally, for a state $q \in Q$, we write $q \models \varphi$ when φ holds in the structure $S_{\{I \leftarrow q\}}$ obtained from S by assuming that q is the initial state.

Remark 2.5 These notions of satisfaction "in a run" and "in a Kripke structure" consider that temporal specifications apply to the initial states. This viewpoint, sometimes called the "anchored viewpoint" [MP89], is the most natural for programming. A possible alternative, more natural for logicians, is to consider that φ holds in π when it holds at all positions along π (the "floating viewpoint"). It is easy to translate the floating viewpoint into the anchored viewpoint (with a G) and the anchored viewpoint into the floating viewpoint (with some extra labeling for the initial state, or with one of the past-time modalities we define in Section 5). $\square$

Other *LTL*-like logics. There is a convenient way of denoting linear-time temporal logics like *LTL*: we write $L(\mathsf{H}_1, \ldots)$ for the logic with $\mathsf{H}_1, \ldots$ as modalities. For example, *LTL* is $L(\mathsf{U}, \mathsf{X})$. This notation assumes that the H_is are equipped with a semantical definition like (S1) and (S2) above.

2.5.2 *CTL**

*CTL** extends *LTL* with quantification over runs. It is interpreted over computation trees, i.e. well-founded trees where the branches are ω-type runs.

Syntax. The set of *CTL** formulae is given by the following abstract grammar:

$$\varphi, \psi \ ::= \ \underline{\mathsf{E}\varphi} \mid \varphi \,\mathsf{U}\, \psi \mid \mathsf{X}\, \varphi \mid \varphi \wedge \psi \mid \neg\varphi \mid P_1 \mid P_2 \mid \cdots \qquad (\mathit{CTL}^* \text{ syntax})$$

where we underlined the extension of *LTL*.

E is the existential path quantifier (see semantical definition below) and the universal quantifier is defined as usual as an abbreviation: $\mathsf{A}\varphi \equiv \neg\mathsf{E}\neg\varphi$.

Semantics. We omit the formal definition of what is a computation tree T in general and assume that T is a tree $T(q) \in T(S)$. The branches of T are runs $\pi \in \Pi(q)$, called "runs in T". Below we let $\pi[0, \ldots, i]$ denote the sequence $s_0, \ldots, s_i$ of the first $i + 1$ states of π.

One defines when a CTL^* formula φ holds at position i of run π in computation tree T, written $T, \pi, i \models \varphi$ (or $\pi, i \models \varphi$ when T is clear from the context), by induction over the structure of φ: clauses (S1–S5) apply unchanged (thanks to our notational choices), and we add

$$\pi, i \models \mathsf{E}\varphi \overset{\text{def}}{\Leftrightarrow} \pi', i \models \varphi \text{ for some } \pi' \text{ in } T \text{ s.t. } \pi[0, \ldots, i] = \pi'[0, \ldots, i]. \quad \text{(S6)}$$

Then come the usual derived notions of satisfaction: a CTL^* formula φ holds in a tree T, written $T \models \varphi$, when $T, \pi, 0 \models \varphi$ for all runs π in T. We further say that φ holds in a Kripke structure S, written $S \models \varphi$, when $T \models \varphi$ for all $T \in T(S)$.

Remark 2.6 Any LTL formula φ is also a CTL^* formula. The semantical definitions are coherent since

$$S \models_{LTL} \varphi \text{ iff } S \models^*_{CTL} \mathsf{A}\,\varphi \text{ iff } S \models^*_{CTL} \varphi$$

holds for any S. $\square$

Like LTL, CTL^* is a future-only logic. Therefore, it is possible to read $\mathsf{E}\varphi$ as "there exists a branch starting from the current situation, and where φ holds".

CTL^* **is more expressive than** LTL**.** The possibility of referring to the alternative runs that branch off from the current state is used e.g. in a formula like

$$\mathsf{A}\big[(\mathsf{G}\,\mathtt{life}) \Rightarrow (\mathsf{G}\,\mathsf{E}\,\mathsf{X}\,\mathtt{death})\big] \qquad (\varphi_{\mathrm{br}})$$

stating that along all runs ("A") with eternal life ("$\mathsf{G}\,\mathtt{life}$") it is always ("$\mathsf{G}$") possible ("$\mathsf{E}$") to meet death at the next moment ("$\mathsf{X}\,\mathtt{death}$"). Obviously, this possible death assumes that we branch off and follow a different path!

Lamport [Lam80] showed that a formula like φ_{br} has no LTL equivalent, that is, there is no LTL formula ψ such that $S \models \varphi_{\mathrm{br}}$ iff $S \models \psi$ for all Kripke structures S. Here is why: consider the Kripke structures S_1 and S_2 from Figure 2. Compared to S_1, S_2 has additional runs with eternal life and no escape to death, thus $S_1 \models \varphi_{\mathrm{br}}$ while $S_2 \not\models \varphi_{\mathrm{br}}$. However, since the two structures have isomorphic sets of runs (see Remark 2.1), $S_1 \models \psi$ iff $S_2 \models \psi$ for any linear-time formula ψ.

Other CTL^***-like logics.** Emerson and Halpern introduced a convenient notation, "$B(L)$", for branching-time logics like CTL^* that extend a linear-time logic L with quantification over branches. For example, CTL^* is $B(LTL)$, and we introduce other instances in the rest of this survey. The

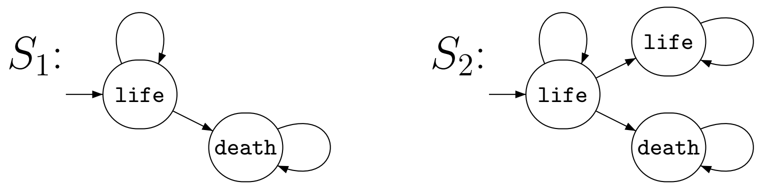

Figure 2. Two Kripke structures with $\Pi(S_1) = \Pi(S_2)$ and $T(S_1) \neq T(S_2)$

$B(L)$ notation assumes L was interpreted over runs, adds a "E" path quantifier, and interpret the resulting formulae over computation trees with the semantical clause (S6) above.

2.5.3 *CTL*

CTL is the fragment of CTL^* where every temporal modality (U or X) must be under the immediate scope of a path quantifier (E or A). The semantics is inherited from CTL^*.

An alternative view. While we have just completely defined *CTL* with the above paragraph, we should mention that this does not adopt the standard way of presenting *CTL*. For one thing, and as the names indicate, *CTL* was introduced before CTL^*, not as a fragment of it.

Requiring that temporal modalities be immediately under a path quantifier entails that they appear in pairs. Thus *CTL* can be defined with four modalities: EU, AU, EX and AX.

$$\begin{aligned} \varphi, \psi \ ::= {} & \mathsf{E}(\varphi \,\mathsf{U}\, \psi) \mid \mathsf{A}(\varphi \,\mathsf{U}\, \psi) \mid \mathsf{EX}\, \varphi \mid \mathsf{AX}\, \varphi \\ & \mid \varphi \wedge \psi \mid \neg\varphi \mid P_1 \mid P_2 \mid \cdots \end{aligned} \qquad (CTL \text{ syntax})$$

Additional modalities like EF, AF, EG and AG are defined as abbreviations.

All *CTL* formulae are *state formulae*: whether φ holds in some T, π, i only depends on the current state, s_i, and not the future that is being considered, i.e. π. Consequently, we often simply write $T, s_i \models \varphi$, or $s_i \models \varphi$ when T is clear.

Remark 2.7 In programming, temporal logic first considered linear runs. There F, the classical modality for "eventually" or "sometimes in the future", had the obvious definition. When tree-like structures were considered, it was not clear what "eventually" should mean, as pointed out by [Lam80]. One possibility, perhaps the simplest, was to equate F with "in some future". The other possibility was to read F more as "eventually", i.e. as "inevitably", or "at some point in all futures". No single choice was expressive enough, and branching-time logics offer the two options, with EF for possibility, and AF for inevitability [BPM83]. Applied to Until, this line of thought leads to *CTL*. (On the quirkiness of *CTL* combinators, see [RM01].) $\square$

EX and AX are dual modalities, and any one of them can be defined from the other. Observe that EF and AF are not dual modalities. The dual of EF is AG, a modality for "at all points in all futures", that could be rendered by "permanently" ("always" is possible but more ambiguous). The dual of AF is EG, a modality for "at all points along one future", which is hard to render in English (but perhaps "possibly always" is a decent attempt?). $\{\mathsf{EU}, \mathsf{AF}, \mathsf{EX}\}$ is a minimal complete set of modalities for *CTL*, while $\{\mathsf{AU}, \mathsf{EF}, \mathsf{EX}\}$ is not complete [Lar95].

Other *CTL*-like logics. When $\mathsf{H}_1,\ldots$ are linear-time modalities, we write $B(\mathsf{H}_1,\ldots)$ to denote the branching-time logic where the modalities are EH_1, AH_1, ... Hence $B(\mathsf{H}_1,\ldots)$ is the fragment of $B(L(\mathsf{H}_1,\ldots))$ where every modality must appear under the immediate scope of a path quantifier. For example, *CTL* is $B(\mathsf{U},\mathsf{X})$ while CTL^* is $B(L(\mathsf{U},\mathsf{X}))$, i.e. $B(LTL)$.

2.6 The model checking problem

The *standard model checking problem* for a temporal logic L is the decision problem associated with the language (set)

$$MC(L) \stackrel{\text{def}}{=} \{\langle S, \varphi\rangle \mid \varphi \in L \text{ and } S \models \varphi\}$$

where S ranges over all finite Kripke structures. Less formally, $MC(L)$ is the problem of deciding, for any inputs S and $\varphi \in L$, whether $S \models \varphi$ or not.

There exists many possible variants (checking that *some* state of a given S satisfies a given φ, ...) and/or restrictions (to *acyclic* structures, ...) motivated by practical or by theoretical considerations: all these problems can be called "model checking problems" and can often be addressed with the techniques we survey below.

For the huge majority of (propositional) temporal logics L found in the programming literature, $MC(L)$ is decidable.

It is not fair to say that "decidability is obvious since the Kripke structures are finite" [8]: while S is finite, the models in which L is interpreted,

[8] It is fair to say that model checking of first-order formulae over finite first-order structures is obviously decidable (and can be done in polynomial-space). Over finite structures, decidability for higher-order formulae is equally obvious.

$\Pi(S)$ or $T(S)$, are not! However, $\Pi(S)$ and $T(S)$ are ω-regular objects (in the automata-theoretic sense, see [Tho90]) that admit general decidability results for monadic second-order logics in which our temporal logics are easily defined [Rab69, GS85].

Once the model checking problem for some temporal logic L is seen to be decidable, the next question is to find actual algorithms solving the problem, to evaluate their cost (essentially, their running time), and to try to prove that no better algorithm exists.

As we explained in the introduction, computational complexity provides a powerful framework for proving optimality of algorithms, at the cost of some simplifying abstractions (asymptotic measures, comparing performance on the worst cases, up-to polynomial transformations).

The cost of algorithms deciding whether $S \models \varphi$ for some Kripke structure S and temporal formula φ is given in terms of the sizes $|S|$ and $|\varphi|$ of the inputs.

In practice we assume that $|\varphi|$ is the number of symbols in φ seen as a string, and $|S|$ is the size of the underlying graph, that is, the sum $|Q|+|R|$ of the number of nodes and the number of edges [9].

2.7 Model checking *vs.* validity

Before we start explaining optimal algorithms for model checking, let us observe that model checking is usually easier than validity or satisfiability, and is almost never harder. The underlying reason is that, for many temporal logics, model checking reduces to validity since it is possible to describe a finite Kripke structure with a succinct temporal formula.

Formally, with a Kripke structure $S = \langle Q, R, l, I \rangle$ we associate a temporal formula φ_S that describe the runs of S. If the set of states is $Q = \{r, s, \ldots\}$, φ_S will use fresh propositions $P_r, P_s, \ldots$, one for each state of S, and is given by:

$$\begin{aligned} &\bigvee_{q \in I} P_q \wedge \mathsf{G}\Big(\bigvee_{q \in Q} P_q \wedge \bigwedge_{q \neq q' \in Q} (\neg P_q \vee \neg P_{q'})\Big) \\ \wedge &\bigwedge_{q \in Q} \mathsf{G}\Big(P_q \Rightarrow \Big(\mathsf{X} \bigvee_{q' \in R(q)} P_{q'}\Big) \wedge \bigwedge_{P \in l(q)} P \wedge \bigwedge_{P \notin l(q)} \neg P\Big) \end{aligned} \qquad (\varphi_S)$$

[9] These definitions could be discussed (e.g. why not also consider the size of the node labeling when defining $|S|$?). Let us just claim authoritatively that they assume the right level of abstraction for the kind of complexity results we give in the rest of this survey.

Now, for any *LTL* formula ψ

$$S \models \psi \text{ iff } \varphi_S \Rightarrow \psi \text{ is valid.}$$

This provides a logspace reduction from *LTL* model checking to *LTL* validity. Similar reductions exist for CTL^* (left as an exercise) and other logics.

For linear-time logics, validity can be reduced to model checking by considering a Kripke structure where all possible valuations are represented and connected. This construction helps one to understand why model checking and validity are so similar in linear-time logics. It does not *prove* they are equivalent since the Kripke structure has size exponential in the number of propositions we use. For branching-time logics, there is no such easy reduction, and validity is often much more complex than model checking.

3 Model checking for the main temporal logics

3.1 Upper bounds

We start with upper bounds, i.e. results stating that there exists some algorithm running inside the required complexity bounds.

3.1.1 *CTL*

The early model checkers from [QS82, CES86] could only be implemented and successfully tackle non-trivial problems because they considered temporal logics with a polynomial-time model checking problem:

Theorem 3.1 [CES86, AC88] *The model checking problem for CTL can be solved in time* $O(|S|.|\varphi|)$.

An $O(|S|.|\varphi|)$ algorithm was published in [CES86] and is reproduced in [CG87, CGP99, CS01]. It can be seen as a dynamic programming algorithm where one computes (and records in some array) whether $q \models \psi$ for all states q of S and all subformulae ψ of φ. The bilinear time is achieved since, using standard graph algorithms for reachability and strongly connected components, and treating the subformulae of ψ as mere additional propositions, one can decide whether $q \models \psi$ in linear-time.

Later, Arnold and Crubillé gave a more elegant algorithm for model checking *CTL* and more generally the alternation-free fragment of the branching-time μ-calculus [AC88, CS93]: see [BBF$^+$01, § 3.1] for an exposition geared towards *CTL*.

3.1.2 *LTL*

Model checking linear-time formulae is more difficult and this explains why *LTL* model checkers were not available immediately. It is interesting to note that decidability was proved as early as [Pnu77] [10] since at that time model checking was certainly not yet widely recognized as a worthy problem.

Theorem 3.2 [SC85] *The model checking problem for LTL is in* PSPACE.

Sistla and Clarke show that satisfiability of *LTL* formulae is in PSPACE, and then obtain Theorem 3.2 via the reduction from model checking to satisfiability.

Their algorithm for satisfiability relies on a *small model theorem*: they show that a satisfiable *LTL* formula φ has an ultimately periodic model of size $2^{O(|\varphi|)}$. Their nondeterministic algorithm is simply to guess the model (a path) and check it step by step: one starts by guessing what subformulae of φ hold in the initial state. This set can be stored in polynomial-space. Then (the valuation of) the next state and the set of subformulae it satisfies are guessed. A local consistency check allows one to forget the subformulae of the previous set and go on to the next state. At some point, the algorithm guesses that the current state will be the one where we loop back to in the ultimately periodic path, and simply records it before going on with the next state. Eventually, the next state turns out to be what has been recorded: the loop is closed and φ has been proved satisfiable. The small-model theorem is important for unsatisfiable formulae: the algorithm needs some criterion to eventually terminate on negative instances without missing positive ones. A polynomial-space counter is enough for visiting at most $2^{O(|\varphi|)}$ states.

This gives an algorithm in NPSPACE, one concludes using NPSPACE = PSPACE.

The above algorithm is not practical: it is nondeterministic and (more importantly) it reduces model checking to satisfiability. This would lead to a deterministic algorithm running in time $2^{(|\varphi|+|S|)^{O(1)}}$.

A better method was needed:

Theorem 3.3 [LP85, VW86] *The model checking problem for LTL can be solved in time* $2^{O(|\varphi|)}O(|S|)$.

The first practical algorithm for *LTL* model checking was given in [LP85] and had the $2^{O(|\varphi|)}O(|S|)$ running time. Vardi and Wolper then described how their Büchi automata approach for modal logics could provide the same running time but with a clearer and conceptually simpler algorithm [VW86].

This approach is now well-known: one associates with any *LTL* formula φ a Büchi automaton $\mathcal{A}_\varphi$ that accepts exactly the models of φ (seen as

[10] The result was given for the $L(\mathsf{F})$ fragment (the logic used in [Pnu77]) and relied on the now standard reduction to inclusion between ω-languages ("does $L(\varphi) \subseteq L(S)$?").

infinite words of valuations). One can then use $\mathcal{A}_\varphi$ to check for satisfiability of φ or for the existence of a run satisfying φ in some Kripke structure S. The size of $\mathcal{A}_\varphi$ is $2^{O(|\varphi|)}$, hence Theorem 3.3. See [Var96, Wol01] for more details.

There now exists even more direct approaches based on alternating automata [Var95] but the algorithms implemented in popular *LTL* model checkers (such as Spin [Hol97]) are based on standard nondeterministic automata.

3.2.3 *CTL**

In principle, model checking *CTL** is an easy adaptation of the model checking algorithms for *LTL*, as was observed by Emerson and Lei:

Theorem 3.4 [EL87, CES86] *For any linear-time future-only logic L, there is a polynomial-time Turing reduction from model checking $B(L)$ to model checking L.*

Hence if L has model checking in some complexity class $\mathcal{C}$, then $B(L)$ has model checking in $\mathrm{P}^{\mathcal{C}}$.

This applies to *CTL** since *CTL** is $B(LTL)$: we only need to remember that *LTL* has PSPACE-complete model checking.

Corollary 3.5 [EL87, CES86] *The model checking problem for CTL* is in* $\mathrm{P}^{\mathrm{PSPACE}}$*, that is, in* PSPACE*.*

The algorithm underlying Theorem 3.4 is quite simple: one uses a dynamic programming approach *à la CTL* and first computes in which states the subformulae are satisfied before dealing with the superformula. For a formula of the form $\mathsf{E}\varphi_l$ where φ_l is a linear-time formula, it is enough to use an *LTL* model checking algorithm.

We illustrate this on a simple example: imagine φ is $\mathsf{AFG}\,\mathsf{E}[(P_1 \Rightarrow \mathsf{X}P_2)\mathsf{U}P_3]$. We replace the subformulae starting with a universal path quantifier by fresh propositions (a form of renaming). Here φ is rewritten as $\mathsf{AFG}\neg P$ where P stands for $\mathsf{A}\neg[(P_1 \Rightarrow \mathsf{X}P_2)\mathsf{U}P_3]$. Then an *LTL* model checking algorithm computes which states satisfies $\mathsf{A}\neg[(P_1 \Rightarrow \mathsf{X}P_2)\mathsf{U}P_3]$ and label them with the new proposition P. We then reuse the *LTL* model checker on the modified Kripke structure, checking where $\mathsf{AFG}\neg P$ holds. Finally, one can perform *CTL** model checking with $O(|S| \times |\varphi|)$ invocations of an *LTL* model checker on subformulae of φ:

Corollary 3.6 *The model checking problem for CTL* can be solved in time* $2^{O(|\varphi|)}O(|S|^2)$.

The $2^{O(|\varphi|)}O(|S|^2)$ bound assume that we invoke the *LTL* model checker for all subformulae and all states of S. But the automata-theoretic method underlying Theorem 3.3 easily gives us *all the states* in S where a same subformula holds in just one invocation.

Corollary 3.7 [EL87, KVW00] *The model checking problem for CTL^* can be solved in time $2^{O(|\varphi|)}O(|S|)$.*

Hence model checking for CTL^* is really no harder than for *LTL*.

It is surprising that no real CTL^* model checker has yet been made available (but see [VB00]). The underlying reasons probably have to do with the hiatus between the pronouncements of theoretical complexity and what is observed in practice. Another factor is that, for non-specialists, branching-time logics are less natural than linear-time ones: witness the $\mathsf{AX}\,\mathsf{AF}\,P \not\equiv \mathsf{AF}\,\mathsf{AX}\,P$ conundrum [Var01].

3.2 Lower bounds

Lower bounds on the complexity of a computational problem state that solving this problem requires at least a given amount of computing power. Such lower bounds are used to prove that a problem is inherently difficult, or that a known algorithm is "optimal" and cannot be improved (made more efficient) in an essential way.

3.2.1 *CTL*

Polynomial-time algorithms for model checking *CTL* are "optimal" since the problem is P-complete:

Theorem 3.8 *The model checking problem for CTL is* P*-hard.*

While this result is not unknown in the model checking community, we failed to find any mention of it in the early literature.

In fact, model checking is already P-hard for the $B(\mathsf{X})$ and $B(\mathsf{F})$ fragments of *CTL*. A direct proof is by reduction from CIRCUIT-VALUE, well-known to be P-complete even when restricted to *monotone* (no negation) *synchronized* (connections between gates respect layers) circuits with *proper alternation* [GHR95]. We illustrate the reduction on an example: Consider the circuit C from Fig. 3. C can be seen as a Kripke structure S_C where the initial state q_I is at the top and where transitions go downward. Then

$$C \text{ evaluates to 1 iff } S_C \models \overbrace{\mathsf{AX}\,\mathsf{EX}\,\mathsf{AX}\,\mathsf{EX}\,1}^{\varphi_C}.$$

Observe that, in this construction, the $B(\mathsf{X})$ formula φ_C depends on (the

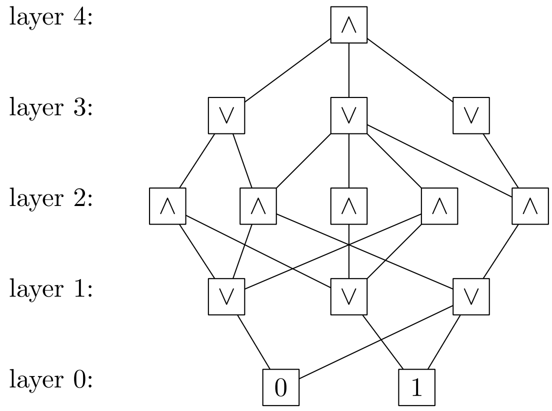

Figure 3. C, an instance of CIRCUIT-VALUE.

depth of) C [11].

According to standard opinion in complexity theory, proving that model checking for *CTL* is P-hard means that, very probably,[12] it cannot be solved using only polylogarithmic space and does not admit efficient parallel algorithms: the problem is inherently sequential and requires storing a polynomial number of intermediary results.

3.2.2 *LTL*

Polynomial-space algorithms for model checking of *LTL* are "optimal" since the problem is PSPACE-complete:

Theorem 3.9 [SC85] *The model checking problem for LTL is* PSPACE-*hard.*

We give a proof based on reduction from a tiling problem. This was inspired by Harel's proof (see [Har85]) that satisfiability of *LTL* formulae is PSPACE-hard (a result due to [HR83]): as is common with linear-time temporal logics, model checking and satisfiability are closely related problems, and proofs can often be transfered from one problem to the other.

[11] This is inescapable in view of Theorem 6.3.2 below. A reduction using always the same formula is possible with the alternation-free fragment of the branching-time μ-calculus, an extension of *CTL* for which Theorem 6.3.2 does not apply.

[12] "Very probably" because it has not yet been proved that POLYLOG − SPACE or NC do not coincide with P (even though most researchers believe this is the case) [Joh90, Pap94]. The situation here is like with the P $\neq$ NP? question.

Assume $C = \{c_1, c_2, \ldots, c_l\}$ is a set of *colors* and $D \subseteq C^4$ is a set of *tile types*: $d \in D$ has the form $d = \langle c_{\mathtt{up}}, c_{\mathtt{right}}, c_{\mathtt{down}}, c_{\mathtt{left}} \rangle$. A *tiling* of some region $\mathcal{R} \subseteq \mathbb{Z}^2$ is a mapping $t : \mathcal{R} \to D$ s.t. neighboring tiles have matching colors on shared edges. The PSPACE-complete problem we reduce from is: given some D of size n and two colors $c_0, c_1 \in C$, is there some $m \in \mathbb{N}$ and a tiling of the $n \times m$ grid s.t. the bottom edge of the grid is colored with c_0 and the top edge with c_1.

With an instance $D = \{d_1, \ldots, d_n\}$ of the problem we associate the Kripke structure S_D depicted in Fig. 4. The states are labeled with tile

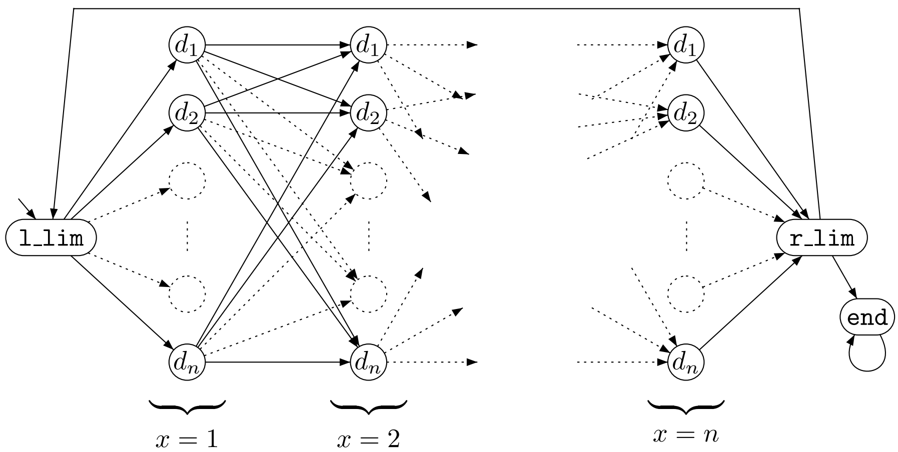

Figure 4. The Kripke structure S_D associated with a tiling problem D

types from D and with additional propositions like "$x = k$" (depending on position) or "$\mathtt{up} = c$" (depending on tile type). A path π through S_D lists the tiles on the first row, then loops back to the leftmost state and lists the tiles on the second row, etc., until it perhaps decides to stop and loop forever in the end state. It remains to state that such a path is indeed a tiling:

1. The lower edge and the upper edge (exist and) have colors c_0 and c_1:

$$\bigwedge_{k=1}^{n} \mathsf{X}^k(\mathtt{down} = c_0) \;\; \bigwedge \;\; \mathsf{F}\Big(\mathtt{l\_lim} \wedge \bigwedge_{k=1}^{n} \mathsf{X}^k(\mathtt{up} = c_1) \wedge \mathsf{X}^{n+2}\mathtt{end}\Big) \qquad (\varphi_1)$$

2. and neighboring tiles have matching edges:

$$\mathsf{G} \bigwedge_{c \in C} \begin{pmatrix} & \mathtt{right} = c \;\Rightarrow \mathsf{X}\,(\mathtt{r\_lim} \vee \mathtt{left} = c) \\ \wedge & \mathtt{up} = c \;\Rightarrow \mathsf{X}^{n+2}\,(\mathtt{end} \vee \mathtt{down} = c) \end{pmatrix} \qquad (\varphi_2)$$

We now have the required reduction since obviously

$$D \text{ is not solvable iff } S_D \models \neg(\varphi_1 \wedge \varphi_2).$$

3.2.3 *CTL**

Since *LTL* is a fragment of *CTL**, Theorem 3.9 entails:

Corollary 3.10 *The model checking problem for CTL* is* PSPACE*-hard.*

3.3 Completeness results

When lower bounds and upper bounds coincide we obtain completeness in a given complexity class. This is indeed the case with the three model checking problems we considered in Section 3, as summarized in Table 1.

CTL	P-complete
LTL	PSPACE-complete
*CTL**	PSPACE-complete

Table 1. Model checking the main temporal logics

The table shows a big contrast between *CTL* and *LTL*! In the early days of model checking, such results were used to argue that branching-time logics are preferable to linear-time logics.

In reality, the situation is not so clear-cut, as we see in Sections 6 and 7.

4 Model checking fragments of temporal logic

Once the cost of model checking the main temporal logics has been precisely measured, it is natural, for the theoretician and the practitioner alike, to look at *fragments*, i.e. sublanguages defined in some way or other. The goal is to better understand what makes model checking difficult or easy, and to ascertain the scope of the results we presented in the previous sections. For the practitioner, it is useful to identify fragments for which the complexity is reduced: if these fragments occur naturally in practice, the corresponding specialized algorithms may be worth implementing.

Remark 4.1 Clearly, similar motivations exist for considering *extensions*. For the practitioner, it is useful to know that an implemented algorithm can in fact be used for a richer and more expressive logic. Indeed there exists many proposals for logics that extend *LTL* (sometimes greatly) without being essentially more difficult for model checking. Four well-known examples are *CTL** (as seen in Section 3.2), *LTL* extended with (essentially) Büchi automata [Wol83], *LTL* with existential quantification over propositions [SVW87], and $LTL + Past$ that we consider in Section 5 below. $\square$

4.1 Fragments of LTL

4.1.1 Restricted set of modalities

Some fragments of LTL are obtained by restricting the allowed modalities. E.g. $L(\mathsf{U})$ is LTL where U is the only allowed modality (X is not allowed but F is since it can be expressed with U). Thus LTL is $L(\mathsf{U},\mathsf{X})$ and $L(\mathsf{F})$ is a fragment of $L(\mathsf{U})$ that coincides with $L(\mathsf{G})$ (since F and G are dual modalities).

Theorem 4.2 **[SC85]** *The model checking problem for $L(\mathsf{F},\mathsf{X})$ is* PSPACE-*complete.*

Indeed our proof of Theorem 3.2 only used $L(\mathsf{F},\mathsf{X})$ formulae!

Theorem 4.3 **[SC85]** *The model checking problem for $L(\mathsf{U})$ is* PSPACE-*complete.*

One just has to show that the lower bound still applies. For this we modify the reduction from Theorem 3.2 so that it uses $L(\mathsf{U})$ formulae:

$$\begin{array}{c} (\mathtt{l\_lim} \vee \mathtt{down}{=}c_0)\ \mathsf{U}\ \mathtt{r\_lim} \\ \wedge\ \ \mathsf{F}\Big(\mathtt{l\_lim} \wedge (\mathtt{l\_lim} \vee \mathtt{up}{=}c_1 \vee \mathtt{r\_lim})\ \mathsf{U}\ \mathtt{end}\Big) \end{array} \qquad (\varphi_1')$$

$$\mathsf{G} \bigwedge_{k=1}^{n-1} x{=}k \Rightarrow \bigwedge_{c\in C} \left[\begin{array}{l} \mathtt{up}{=}c \ \Rightarrow\ x{=}k\ \mathsf{U}\Big[x{\neq}k\ \mathsf{U}\Big[\begin{array}{l}\mathtt{end}\ \vee \\ x{=}k \wedge \mathtt{down}{=}c\end{array}\Big]\Big] \\ \wedge\ \mathtt{right}{=}c\ \Rightarrow\ x{=}k\ \mathsf{U}\ \mathtt{left}{=}c \end{array}\right] \qquad (\varphi_2')$$

With Theorems 4.2 and 4.3 we have two instances of situations where considering strict fragments [13] of LTL does not simplify the model checking problem in an essential way. But further restricting the set of allowed modalities decreases the complexity of model checking, as the next two results show:

Theorem 4.4 **[SC85]** *The model checking problem for $L(\mathsf{F})$ is* coNP-*complete.*

Membership in coNP is a consequence of a small model theorem: a satisfiable $L(\mathsf{F})$ formula φ has an ultimately periodic model of size $O(|\varphi|)$ [ON80, SC85]. The reduction from model checking to satisfiability (section 2.6)

[13] That $L(\mathsf{F},\mathsf{X})$ and $L(\mathsf{U})$ are less expressive than LTL is well-known. See [EVW02, KS02] for recent results on these issues.

translates this as: if $S \not\models \varphi$ then S has an ultimately periodic path of polynomial-size that does not satisfy φ. It is enough to guess this counter-example path and check it.

Hardness for coNP can be explained by reduction from the validity problem for Boolean formulae in clausal form: assume θ is a 3SAT instance with variables among $X_n = \{x_1, \ldots, x_n\}$, e.g. θ is "$\big(x_1 \wedge \overline{x_2} \wedge \overline{x_4}\big) \vee \big(\overline{x_1} \wedge \cdots\big) \vee \cdots$". Consider the structure S_n from Fig. 5. There is a one-to-one correspondence between runs in S_n and valuations for X_n. Thus

$$\theta \text{ is valid iff } S_n \models \big(\mathsf{F}\, x_1 \wedge \mathsf{F}\, \overline{x_2} \wedge \mathsf{F}\, \overline{x_4}\big) \vee \big(\mathsf{F}\, \overline{x_1} \wedge \cdots\big) \vee \cdots,$$

providing the required reduction.

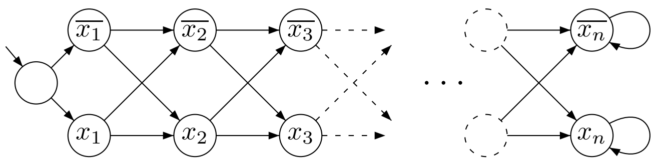

Figure 5. S_n, a structure for picking Boolean valuations of $\{x_1, \ldots, x_n\}$

Theorem 4.5 [DS02] *The model checking problem for* $L(\mathsf{X})$ *is* coNP*-complete.*

Here the small model theorem is trivial: if φ has temporal depth k then only the k first states of a linear-time model for φ are relevant. For coNP-hardness, we proceed as we just did:

$$\theta \text{ is valid iff } S_n \models \big(\mathsf{X}^1 x_1 \wedge \mathsf{X}^2 \overline{x_2} \wedge \mathsf{X}^4 \overline{x_4}\big) \vee \big(\mathsf{X}^1 \overline{x_1} \wedge \cdots\big) \vee \cdots$$

It is possible to consider further natural "subsets" of temporal modalities among $\{\mathsf{X}, \mathsf{F}, \mathsf{U}\}$:

$L(\overset{\infty}{\mathsf{F}})$ is the fragment of $L(F)$ where F can only be used in the form $\overset{\infty}{\mathsf{F}}$ (that is, GF). This fragment is useful for stating fairness properties and it does not have the full power of $L(\mathsf{F})$ (e.g., ordering constraints cannot be stated).

We leave the reader to adapt the proof of Theorem 4.4 and show that the model checking problem for $L(\overset{\infty}{\mathsf{F}})$ is coNP-complete [EL87].

$L(\mathsf{U}^-)$ is the fragment of $L(\mathsf{U})$ where only "flat Until" is allowed. Flat Until is Until where only propositional formulae (no temporal modality) is allowed in the left-hand side. E.g. $a\mathsf{U}(b\mathsf{U}c)$ uses flat Until but $(a\mathsf{U}b)\mathsf{U}c$

does not. In expressive power, flat Until lies strictly between F and U [Dam99].

Since the proof of Theorem 4.3 only uses $L(\mathsf{U}^-)$ formulae, we conclude that model checking for $L(\mathsf{U}^-)$ is PSPACE-complete [DS02].

The results of this section are summarized in Table 2.

<table>
<tr><td>LTL</td><td>PSPACE-complete</td></tr>
<tr><td>$L(\mathsf{U})$</td><td rowspan="3">PSPACE-complete</td></tr>
<tr><td>$L(\mathsf{U}^-)$</td></tr>
<tr><td>$L(\mathsf{F},\mathsf{X})$</td></tr>
<tr><td>$L(\mathsf{F})$</td><td rowspan="3">coNP-complete</td></tr>
<tr><td>$L(\overset{\infty}{\mathsf{F}})$</td></tr>
<tr><td>$L(\mathsf{X})$</td></tr>
</table>

Table 2. Model checking fragments of *LTL* $(= L(\mathsf{U},\mathsf{X}))$

4.1.2 Restricted temporal depth

Let $\mathsf{H}_1, \ldots$ be some temporal modalities. For $k \in \mathbb{N}$, we write $L^k(\mathsf{H}_1, \ldots)$ for the fragment of $L(\mathsf{H}_1, \ldots)$ where only formulae of temporal depth at most k are allowed.

It is natural to ask whether enforcing such a depth restriction can make model checking easier. For example, for many modal logics considered in [Hal95], satisfiability becomes polynomial-time when the modal depth is bounded.

The situation is different with *LTL*: for modalities like U and F, (satisfiability and) model checking is already at its hardest with a low temporal depth [14], as stated by the next two results.

Theorem 4.6 [DS02] *The model checking problem for* $L^2(\mathsf{U})$ *is* PSPACE-*complete.*

The proof of Theorem 4.3 uses $\varphi'_1, \varphi'_2 \in L^3(\mathsf{U})$, so we already know PSPACE-hardness for $L^3(\mathsf{U})$! We let the reader adapt this and prove PSPACE-hardness for $L^2(\mathsf{U})$ as well (or even for $L^2(\mathsf{U}^-)$).

Theorem 4.7 [DS02] *The model checking problem for* $L^1(\mathsf{F})$ *is* coNP-*complete.*

[14] By contrast, expressive power always increases with temporal depth [EVW02, KS02].

Indeed the proof of Theorems 4.4 shows hardness for the $L^1(\mathsf{F})$ fragment. This also applies to $L^1(\overset{\infty}{\mathsf{F}})$.

Below these thresholds, complexity is decreased:

Theorem 4.8 *The model checking problem for* $L^1(\mathsf{U}, \overset{\infty}{\mathsf{F}}, \mathsf{X})$ *is* coNP*-complete.*

This result is important since, as we see in Section 4.2 below, it has applications for branching-time logics like *CTL* and *CTL*$^+$, where path quantification only allows path formulae without nesting of temporal modalities, i.e. path formulae in some $L^1(\mathsf{H}_1, \ldots)$.

The ideas behind Theorem 4.8 are easy to summarize. Hardness was proved with Theorem 4.4. Membership in coNP comes from a small model theorem: a satisfiable $L^1(\mathsf{U}, \overset{\infty}{\mathsf{F}}, \mathsf{X})$ is satisfiable in a linear-sized model (which lets us proceed as in the proof of Theorem 4.4). For the $L^1(\mathsf{U}, \mathsf{X})$ fragment, the small model theorem is proved in [DS02, Section 7]. As shown in [LMS01, Section 6], this can further be extended to the $L^1(\mathsf{U}, \mathsf{X}, \overset{\infty}{\mathsf{F}})$ fragment, and in fact to any fragment that can be translated efficiently in $FO_2(<)$, the two-variables fragment of the first-order logic of linear orderings [RS00].

Theorem 4.9 [DS02] *For any* $k \in \mathbb{N}$*, the model checking problem for* $L^k(\mathsf{X})$ *can be done in polynomial time.*

This result does not have much practical use. A possible polynomial-time (in fact, logspace) algorithm is simply to look at all tuples of $k+1$ consecutive states in the Kripke structure. This runs in time $O(|S|^{k+1} \times |\varphi|)$ which, for fixed k, is polynomial-time.

4.1.3 Other fragments of *LTL*

Other restrictions have been investigated but they do not seem as interesting as far as model checking is concerned. These may have to do with bounding the number of atomic propositions allowed in temporal formulae, or restricting the use of negations, or forbidding that a given modality occurs in the scope of another given modality (most notably, forbidding future-time modalities to appear under the scope of past-time modalities). Examples of such investigations are reported in [SC85, DS02, Mar02].

4.2 Fragments of CTL^*

There does not seem to be much to say about model checking [15] fragments of CTL: they cannot be harder than CTL itself and the P-hardness proof of Theorem 3.8 only needs EX and one proposition, or EF and two propositions!

Much more interesting are the fragments of CTL^*. In fact, a good deal of effort has been spent trying to extend CTL (because it lacks expressive power, most notably for fairness properties) without losing the efficient model checking algorithms that CTL permits [16].

4.2.1 A selection of branching-time logics

Several natural fragments of CTL^* have been identified and named [ES89, Eme90]. Below we consider:

CTL^+: an extension of CTL, introduced in [EH86], and where Boolean combinations (not nesting) of temporal modalities are allowed under the scope of a path quantifier. E.g. CTL^+ allows formulae like $\mathsf{A}(a\mathsf{U}b \Rightarrow c\mathsf{U}d)$. There exists a translation from CTL^+ to CTL [EH85] (hence the two logics have the same expressive power) but such a translation must produce exponential-size formulae [Wil99b, AI01].

CTL^+ is $B(\mathsf{U}, \mathsf{X}, \wedge, \neg)$ in the notation of [EH86], but we prefer to see it as $B(L^1(\mathsf{U}, \mathsf{X}))$.

$ECTL$: an extension of CTL, introduced in [EH86], where the $\mathsf{E}\overset{\infty}{\mathsf{F}}$ and $\mathsf{A}\overset{\infty}{\mathsf{F}}$ modalities are allowed, providing some expressive power for fairness properties [17]. Thus $ECTL$ is $B(\mathsf{U}, \mathsf{X}, \overset{\infty}{\mathsf{F}})$ in the notation of [EH86, Eme90]. Emerson and Halpern showed that $ECTL$ is not sufficiently expressive for typical situations where fairness issues appear, as when one needs to write $\mathsf{A}((\overset{\infty}{\mathsf{F}}a_1 \wedge \overset{\infty}{\mathsf{F}}a_2 \wedge \ldots \wedge \overset{\infty}{\mathsf{F}}a_n) \Rightarrow \mathsf{F}\ \mathtt{end})$.

$ECTL^+$: was introduced in [EH86] and is $B(\mathsf{U}, \mathsf{X}, \overset{\infty}{\mathsf{F}}, \wedge, \neg)$ in Emerson's notation, or equivalently $B(L^1(\mathsf{U}, \mathsf{X}, \overset{\infty}{\mathsf{F}}))$. It allows one to relativize CTL formulae with any kind of fairness properties.

[15] The situation is more interesting for the *satisfiability* problem. It is EXPTIME-complete for CTL [EH85] and this leaves much room for fragments that could be less intractable, see [ESS92] for example.

[16] The choice of what are the CTL modalities is mostly a historical accident (see Remark 2.7) and it is possible to define more expressive logics for which essentially the same model checking algorithm still applies. Examples are $ECTL$ (see below), or the logic from [KG96].

[17] Only $\mathsf{E}\overset{\infty}{\mathsf{F}}$ is a real addition to CTL since $\mathsf{A}\overset{\infty}{\mathsf{F}}$ can be defined as $\mathsf{AG}\,\mathsf{AF}$.

BT^*: is $B(L(\mathsf{F}))$, i.e. a fragment of CTL^* where F is the only allowed modality. Thus BT^* combines full branching-time power *à la* CTL^* but only based on Eventually, the classical temporal modality, instead of Until and Next. The name BT^* is from [CES86].

BX^*: is $B(L(\mathsf{X}))$, i.e. a fragment of CTL^* where X is the only allowed modality. It is even more basic than BT^* and, to the best of our knowledge, had not been identified in the literature.

4.2.2 Model checking fragments of CTL^*

As far as model checking is concerned, $ECTL$ behaves like CTL:

Theorem 4.10 *The model checking problem for ECTL is* P*-complete, and can be solved in time* $O(|S| \times |\varphi|)$.

This is because the standard CTL model checking algorithm is easily extended to deal with the $\mathsf{E}\overset{\infty}{\mathsf{F}}\varphi$ subformulae: one simply has to find the strongly connected components in the Kripke structure, select those where at least one node satisfies φ, and gather all states from which these connected components can be reached [CES86].

In fact, the $O(|S| \times |\varphi|)$ time can be achieved for any $B(\mathsf{H}_1, \mathsf{H}_2, \ldots, \mathsf{H}_k)$ logic based on a finite set of "existential path modalities" (see [RM01, Rab02]).

When it comes to the other branching-time logics in our selection, the situation is not so simple. They all have the form $B(L(\ldots))$ for a linear-time temporal logic $L(\ldots)$ with a coNP-complete model checking problem: (see Theorem 4.8 for CTL^+, and $ECTL^+$, Theorem 4.4 for BT^*, and Theorem 4.5 for BX^*). Thus Theorem 3.4 applies and gives a $\mathrm{P}^{\mathrm{coNP}}$, or equivalently P^{NP}, model checking procedure for these logics. Of course the coNP lower bound still applies. Hence

Corollary 4.11 *The model checking problem for* CTL^+*,* $ECTL^+$*,* BT^* *and* BX^* *is* NP*-hard,* coNP*-hard and can be solved in* P^{NP}*, i.e. in* Δ_2^p.

Δ_2^p is one of the levels in Stockmeyer's polynomial-time hierarchy [Sto76]. Formally, a problem is in P^{NP} if it can be solved by a deterministic polynomial-time Turing machine making (adaptive) queries to an NP-complete set, e.g. to an oracle for satisfiability [18]. That model checking CTL^+ and BT^* is

[18] Observe that the number of queries and their lengths are polynomially bounded since the algorithm has to produce them in polynomial-time. By "*adaptive* queries" we mean that the queries are produced and answered sequentially, so that the formulation of any query may depend on the answers that were received for the previous queries.

in Δ_2^p was already observed in [CES86, Theorem 6.2]. It is widely believed that Δ_2^p is strictly below PSPACE (inside which the whole polynomial-time hierarchy resides) so that temporal logics like $ECTL^+$ or BT^* are believed to have easier model checking problems than LTL or CTL^*.

The gap between the lower bound (NP-hard and coNP-hard) and the upper bound (in Δ_2^p) has only been recently closed:

Theorem 4.12 [LMS01] *The model checking problem for CTL^+, $ECTL^+$ and BT^* is Δ_2^p-complete.*

In fact the hardness proofs in [LMS01] even apply to $B(L^1(\mathsf{F}))$ and $B(L^1(\overset{\infty}{\mathsf{F}}))$. Theorem 4.12 is mainly interesting for complexity theorists: there exist very few problems known to be complete for Δ_2^p (see [Wag87, Kre88]), in particular none from temporal logic model checking [19], and any addition from a new field helps understand the class. The techniques from [LMS01] also have more general relevance: they have been used to show Δ_2^p-completeness of model checking for some temporal logics featuring quantitative informations about time durations [LMS02a].

In the case of BX^*, the Δ_2^p upper bound can be improved:

Theorem 4.13 [Sch03] *Model checking for BX^* is $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$-complete.*

Here $\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$, from [CS96], is the class of problems that can be solved by a deterministic polynomial-time Turing machine that only makes $O(\log^2 n)$ queries to an NP-complete set (where n is the size of the input). It should be noted that, to the best of our knowledge, this is the first example of a problem complete for this class!

The results of Section 4.2 are summarized in Table 3. This does not provide the good compromise between expressive power and low model checking complexity we were looking for: $ECTL$ suffers from essentially the same expressivity limitations that plague CTL (the same is true of the alternation-free fragment of the branching-time μ-calculus).

5 Temporal logic with past

The temporal logics we have considered until now only had *future-time* modalities, dealing with future events. In fact, in most of programming, temporal logic, i.e. "logic of time", means "logic of the future".

[19] But recently some model checking problems for propositional default logics have been found Δ_2^p-complete [BG02].

CTL^*	$= B(L(\mathsf{U},\mathsf{X}))$	PSPACE-complete
$ECTL^+$	$= B(L^1(\mathsf{U},\mathsf{X},\overset{\infty}{\mathsf{F}}))$	Δ_2^p-complete
CTL^+	$= B(L^1(\mathsf{U},\mathsf{X}))$	Δ_2^p-complete
BT^*	$= B(L(\mathsf{F}))$	Δ_2^p-complete
BX^*	$= B(L(\mathsf{X}))$	$\mathrm{P}^{\mathrm{NP}[O(\log^2 n)]}$-complete
$ECTL$	$= B(\mathsf{U},\mathsf{X},\overset{\infty}{\mathsf{F}})$	P-complete
CTL	$= B(\mathsf{U},\mathsf{X})$	P-complete

Table 3. Model checking fragments of CTL^*

At first sight, this situation seems a bit strange. For logicians and philosophers, temporal logic deals with both past and future, often treating them symmetrically. Furthermore, it is easy to observe that natural languages have a richer set of constructs (tenses, adverbs, ...) dealing with past than with future, and that our own sentences, in everyday speech or mathematical papers, use more past than future.

There are two explanations usually put forward when it comes to explaining this apparently slanted view from programming. Firstly, computer scientists deal with dynamical systems (e.g. Turing machines) that *move forward in time*, and the questions they are interested in concern what will happen in the future (e.g. will the Turing machines halt?). Secondly, a famous result by Gabbay states that past can be dispensed with, in a way that we make completely precise below, after the necessary definitions.

5.1 Past-time modalities

The past-time modalities most commonly used in programming are S ("Since"), F^{-1}, G^{-1} and X^{-1}, i.e. they are the past-time counterparts of U, F, G and X [20].

Syntactically, one defines *PLTL* [21], or *LTL* + *Past*, with the following grammar:

$$\varphi,\psi \;::=\; \varphi\,\mathsf{U}\,\psi \mid \underline{\varphi\,\mathsf{S}\,\psi} \mid \mathsf{X}\,\varphi \mid \underline{\mathsf{X}^{-1}\,\varphi} \mid \varphi\wedge\psi \mid \neg\varphi \mid P_1 \mid \cdots \qquad (\textit{PLTL}\text{ syntax})$$

where we underlined what has been added to the *LTL* syntax from Sec-

[20] Sometimes the names P ("Past") and Y ("Yesterday") are used instead of F^{-1} and X^{-1} but we prefer the emphasis on symmetry. Admittedly, we would have been more consistent if we had used U^{-1} instead of S.

[21] Naming conventions are not yet 100% stable, and some works can be found where *PLTL* stands for "**P**ropositional Linear temporal Logic", which is just plain *LTL*.

tion 2.5. One then defines F^{-1} and G^{-1} as abbreviations:

$$\mathsf{F}^{-1}\,\varphi \;\equiv\; \top\,\mathsf{S}\,\varphi \qquad\qquad \mathsf{G}^{-1}\,\varphi \;\equiv\; \neg\mathsf{F}^{-1}\neg\varphi \tag{2}$$

Semantically, *PLTL* formulae are still interpreted at positions along linear runs, and we complement (S1–5) from Section 2.5 with the following:

$$\pi, i \models \varphi\,\mathsf{S}\,\psi \overset{\text{def}}{\Leftrightarrow} \exists j \in \{0,1,\ldots,i\} \text{ s.t. } \begin{cases} \pi, j \models \psi \\ \text{and} \\ \pi, k \models \varphi \text{ for all } j < k \leq i, \end{cases} \tag{S7}$$

$$\pi, i \models \mathsf{X}^{-1}\varphi \overset{\text{def}}{\Leftrightarrow} i > 0 \text{ and } \pi, i-1 \models \varphi. \tag{S8}$$

Thus S and X^{-1} behaves as mirror images of U and X, so that $\mathtt{sad}\,\mathsf{S}\,\mathtt{rain}$ reads "*(I) have been sad since it rained*", $\mathsf{X}^{-1}\,\mathtt{rain}$ reads "*it rained (just previously)*", $\mathsf{F}^{-1}\,\mathtt{rain}$ reads "*it rained (sometimes in the past)*", and $\mathsf{G}^{-1}\,\mathtt{rain}$ reads "*it has always rained*".

A *pure-past* formula (resp. *pure-future*) is a formula that only uses the past-time modalities X^{-1} and S (resp. the future-time X and U). Hence *LTL* is the pure-future fragment if *LTL* + *Past*.

Remark 5.1 The only asymmetry between past and future is that, since the models are ω-words, there is a definite starting point (past is finite) and no end point (future is forever). This difference is imposed by the applications in programming, but it is unimportant for the main theoretical results. □

5.2 The expressive power of *LTL* + *Past*

Certainly, as was observed in [LPZ85], some natural language statements are easier to translate in *LTL* + *Past* than in *LTL*. For example, while a specification like "*any fault eventually raises the alarm*" is translated as the *LTL* formula

$$\mathsf{G}\,(\mathtt{fault} \Rightarrow \mathsf{F}\,\mathtt{alarm}), \tag{$\varphi_\rightarrow$}$$

we would naturally translate "*any alarm is due to some (earlier) fault*" as

$$\mathsf{G}\,(\mathtt{alarm} \Rightarrow \mathsf{F}^{-1}\,\mathtt{fault}), \tag{$\varphi_\leftarrow$}$$

an *LTL* + *Past* formula.

Observe that both formulae start with a G because our informal "any fault" and "any alarm" mean a fault at any time and an alarm at any time, and more precisely at any time along the runs (the future) of the system.

There is an equivalent way of stating the second property with only future-time constructs:

$$\neg\big[(\neg\mathtt{fault})\,\mathsf{U}\,(\mathtt{alarm} \wedge \neg\mathtt{fault})\big], \qquad (\varphi'_{\leftarrow})$$

which could be read "*one never sees an alarm after no fault*". This equivalent formula is only equivalent at the start of the system:

Definition 5.2 (Global vs. initial equivalence)
1. Two PLTL formulae are (globally) equivalent*, written $\varphi \equiv \psi$, when $\pi, i \models \varphi$ iff $\pi, i \models \psi$ for all linear-time models π, and positions $i \in \mathbb{N}$.*
2. Two PLTL formulae are initially equivalent*, written $\varphi \equiv_i \psi$, when $\pi, 0 \models \varphi$ iff $\pi, 0 \models \psi$ for all linear-time models π.*

Clearly global equivalence entails initial equivalence but the converse is not true: e.g. $\mathsf{F}^{-1}\, a \equiv_i a$ but $\mathsf{F}^{-1}\, a \not\equiv a$. In our earlier example $\varphi_{\leftarrow} \equiv_i \varphi'_{\leftarrow}$ but the two formulae are not globally equivalent. Since $\varphi \equiv_i \psi$ entails ($S \models \varphi$ iff $S \models \psi$) for any S, initial equivalence is enough for comparing model checking specifications. Of course, for the future-only *LTL* logic, initial and global equivalence coincide.

Now Gabbay's Theorem compares *PLTL* and *LTL* with *initial equivalence*:

Theorem 5.3 [Gab89, GPSS80] *Any PLTL formula is* globally *equivalent to a* separated *PLTL formula, i.e. a Boolean combination of pure-past and pure-future formulae.*

Corollary 5.4 (Gabbay's Theorem) *Any PLTL formula is* initially *equivalent to an LTL formula.*

As was alluded to in the introduction of Section 5, Gabbay's Theorem is one more explanation of why computer scientists often neglect past-time modalities in their temporal logics.

It turns out that there are good reasons to rehabilitate past-time modalities: not only does it make formulae easier to write (compare $\varphi_{\leftarrow}$ with the indirect $\varphi'_{\leftarrow}$), but it also gives added expressive power when one takes succinctness into account:

Theorem 5.5 [LMS02b] *PLTL can be exponentially more succinct than LTL.*

More precisely, [LMS02b] considers the following sequence $(\psi_n)_{n=1,2,\ldots}$ of formulae:

$$\mathsf{G}\Big[\Big[\bigwedge_{i=1}^{n}\big(P_i \Leftrightarrow \mathsf{F}^{-1}(P_i \wedge \neg\mathsf{X}^{-1}\top)\big)\Big] \Rightarrow \big(P_0 \Leftrightarrow \mathsf{F}^{-1}(P_0 \wedge \neg\mathsf{X}^{-1}\top)\big)\Big]. \qquad (\psi_n)$$

ψ_n states that every future state that agrees with the initial state on propositions $P_1, \ldots, P_n$ also agree on P_0. The colloquialism $\mathsf{F}^{-1}(\ldots \wedge \neg\mathsf{X}^{-1}\top)$ is used to talk about the initial state, the only state where $\mathsf{X}^{-1}\top$ does not hold. Then, using results from [EVW02], it can be shown that any *LTL* formulae initially equivalent to the ψ_ns have size in $\Omega(2^n)$.

At the moment, it is not known if *PLTL* formulae can be translated into *LTL* formulae with a single exponential blowup, or if the gap in Theorem 5.5 can be further widened [22].

Remark 5.6 The succinctness gap between $LTL + Past$ and *LTL* is important for model checking but not for validity. There exists a succinct satisfiability-preserving reduction from $LTL + Past$ to *LTL* (left as an exercise), using fresh propositions that we call "history variables" in the programming community. However, introducing polynomially-many fresh propositions leads to an exponential blowup when we look at the size of Kripke structures, so that this translation is not practical for model checking. □

5.3 $CTL + Past$ and $CTL^* + Past$

Past-time modalities can be added to branching-time logics too. Here we concentrate on proposals where future is branching but past is linear. While this may seem like just one more arbitrary (and awkward) choice, it is actually the one more consistent with the view that a Kripke structure actually describes a computation tree, and that temporal logic is a logic for describing properties of this tree.

In classical temporal logic, this is referred to as "unpreventability of the past" and leads to so-called Ockhamist temporal logics [Pri67, Bur84]. Logics with a branching past do not adhere to this view and really talk about something other than the behavior, most often the internal structure of the system [23].

[22] Gabbay's effective procedure is not known to be elementary, however an elementary upper bound on formula size can be obtained by combining the standard translation from *PLTL* to counter-free Büchi automata and the elementary translation from these automata to *LTL* using results from [Wil99a].

[23] The interested, or unconvinced, reader will find a longer argumentation in [LS00a]. Proposals with branching past can be found e.g. in [Rei89, Wol89, Sti92, Kam94, KP95].

Syntactically, one defines $PCTL^*$, or $CTL^* + Past$, with the following grammar:

$$\varphi, \psi \ ::= \ \varphi \,\mathsf{S}\, \psi \mid \mathsf{X}^{-1} \varphi \mid \ \cdots \ \textit{usual } CTL^* \textit{ syntax} \ \cdots \qquad (PCTL^* \text{ syntax})$$

The defining abbreviations for F^{-1} and G^{-1} (2) still apply. The semantics is just obtained by combining clauses (S7) and (S8) from Section 5.1 with (S1–6), the semantics of CTL^*. $PCTL$ is the fragment of $PCTL^*$ where every future-time modality (U or X) is immediately under the scope of path quantifier (A or E) [24].

As far as expressive power is concerned, and comparing logics with the same *initial equivalence* criterion we used in Section 5.2, the following results are proved in [LS95]:

- $PCTL^*$ can be translated to CTL^*,
- $PCTL$ is strictly more expressive than CTL,
- the $CTL + \mathsf{F}^{-1}$ fragment of $PCTL$ can be translated to CTL.

The translations are not succinct: $CTL + \mathsf{F}^{-1}$ can be exponentially more succinct than CTL (a consequence of [Wil99a]) and $PCTL^*$ can be exponentially more succinct than CTL^* (a consequence of Theorem 5.5).

5.4 Model checking $LTL + Past$

While $LTL + Past$ is more succinctly expressive than LTL, this does not entail obviously harder model checking problems.

Theorem 5.7 [SC85] *The model checking problem for PLTL is* PSPACE-*complete.*

In fact [SC85] directly gave their small model theorem for $PLTL$, and the proof of Theorem 3.2 can be extended to $PLTL$.

The automata-theoretic approach of Vardi and Wolper extends as well, and with a $PLTL$ formula φ, one can associate a Büchi automaton with $2^{O(|\varphi|)}$ states (see [LPZ85, VW86]), so that $PLTL$ model checking can be done in time $2^{O(|\varphi|)}O(|S|)$ as for LTL and CTL^*.

An empirical observation is that, in most cases, linear-time logic with past is not more difficult than its pure-future fragment. For example, Theorem 4.4 can be extended to:

[24] These definitions for $PCTL$ and $PCTL^*$ are from [LS95]. These logics are equivalent to the CTL_{lp} and CTL^*_{lp} (lp for "linear past") from [KP95]. Our $PCTL^*$ further coincides with the OCL logic from [ZC93]. The $PCTL^*$ from [HT87] differs from our $PCTL^*$ since its path quantifiers always forget the past (see [LS95, LS00a] for a formalism allowing both cumulative and forgettable past).

Theorem 5.8 **[Mar02]** *The model checking problem for* $L(\mathsf{F}, \mathsf{F}^{-1})$ *is* coNP-*complete.*

Here too the coNP algorithm given in [Mar02] relies on a small model theorem: satisfiable $L(\mathsf{F}, \mathsf{F}^{-1})$ formulae admit a linear-sized ultimately periodic model.

5.5 Model checking $CTL + Past$ and $CTL^* + Past$

"Very probably" $CTL+Past$ does not allow polynomial-time model checking algorithms:

Theorem 5.9 **[LS00a]** *The model checking problem for PCTL is* PSPACE-*hard.*

This can be shown via a direct reduction from QBF (a.k.a. Quantified Boolean Formula, a well-known PSPACE-complete problem). We illustrate this on an example: consider a QBF formula θ of the form

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \exists x_n \Big[\big(x_1 \wedge \overline{x_2} \wedge \overline{x_4}\big) \vee \big(\overline{x_1} \wedge \cdots\big) \vee \cdots \Big] \qquad (\theta)$$

Then obviously θ is valid iff

$$S_n \models \mathsf{EX}\,\mathsf{AX}\,\mathsf{EX}\,\mathsf{AX} \cdots \mathsf{EX} \Big[\big(\mathsf{F}^{-1} x_1 \wedge \mathsf{F}^{-1}\overline{x_2} \wedge \mathsf{F}^{-1}\overline{x_4}\big) \vee \big(\mathsf{F}^{-1}\overline{x_1} \wedge \cdots\big) \vee \cdots \Big]$$

in the structure S_n (see Figure 5) we used in Section 4.1. The reduction applies to restricted fragments of $PCTL$: one future-time modality and one past-time modality is enough.

At the other end of the spectrum, model checking for $CTL^* + Past$ is in PSPACE:

Theorem 5.10 **[KPV98]** *Model checking for* $PCTL^*$ *is* PSPACE-*complete.*

As a corollary, we obtain PSPACE-completeness for $PCTL$ too.

The results of Section 5 are summarized in Table 4.

LTL	PSPACE-complete	$LTL + Past$	PSPACE-complete
CTL	P-complete	$CTL + Past$	PSPACE-complete
CTL^*	PSPACE-complete	$CTL^* + Past$	PSPACE-complete

Table 4. Model checking temporal logics with past

6 The two parameters of model checking

Table 1 that concludes Section 3.3 shows that model checking CTL formulae is much easier than model checking LTL formulae, so that it seems CTL is perhaps the better choice when it comes to picking a temporal logic in which to state behavioral properties. Indeed this line of argument was widespread in the early days of model checking.

However, complexity results like the ones we surveyed in previous sections are not the most relevant for assessing the cost of model checking *in practical situations*. For these situations, it is sensible to distinguish between how much each of the two parameters contribute separately to the complexity of model checking. These two parameters are the Kripke structure S and the temporal formula φ. In practical situations, $|S|$ is usually quite large and $|\varphi|$ is often small, so that what matters most is the impact $|S|$ has on the overall cost.

When Lichtenstein and Pnueli published their LTL model checking algorithm that runs in time $2^{O(|\varphi|)}O(|S|)$ [LP85], they explained that the algorithm is *linear-time w.r.t. the Kripke structure*, so that it is comparable to CTL model checking. They further said that the $2^{O(|\varphi|)}$ factor is not so important in practice, most formulae being short anyway, and this is validated by current practice showing that LTL model checking is very feasible.

6.1 Program-complexity and formula-complexity

These considerations can be stated and studied in the complexity-theoretic framework. The *program-complexity* of a model checking problem $MC(L)$ is its computational complexity measured as a function of the Kripke structure S (the program) only, with the temporal formula being fixed. The *formula-complexity* of $MC(L)$ is its computational complexity measured as a function of φ only, with S fixed. For model checking, these notions were introduced in [VW86], inspired from similar ideas from database querying [Var82].

Characterizing the program- and the formula-complexity of model checking allows one to measure the impact each input must have on the overall cost. In such investigations, it is natural to look for specialized algorithms that are asymptotically optimal in their handling of one parameter, perhaps to the detriment of the other parameter.

As an illustration, let us mention the following two results based on algorithms that try to reduce the space required for CTL model checking:

Theorem 6.1 [KVW00] *The model checking problem for CTL can be solved in space* $O\big(|\varphi| \times \log^2(|\varphi| \times |S|)\big)$.

Theorem 6.2 [Sch01] *The model checking problem for CTL can be solved in space* $O(|S| \times \log|\varphi|)$.

6.2 A new view of the cost of model checking

The main results on the program-complexity and the formula-complexity of model checking are listed in the following theorem:

Theorem 6.3 (See Table 5)
1. The program-complexity of LTL model checking is NLOGSPACE-*complete, its formula-complexity is* PSPACE-*complete.*
2. The program-complexity of CTL model checking is NLOGSPACE-*complete, its formula-complexity is in time* $O(|\varphi|)$ *and in space* $O(\log|\varphi|)$.
3. The program-complexity of CTL^* *model checking is* NLOGSPACE-*complete, its formula-complexity is* PSPACE-*complete.*

	program-complexity	formula-complexity	(overall complexity)
LTL	NLOGSPACE-complete	PSPACE-complete	PSPACE-complete
CTL	NLOGSPACE-complete	LOGSPACE	P-complete
CTL^*	NLOGSPACE-complete	PSPACE-complete	PSPACE-complete

Table 5. Three measures for the complexity of model checking

Remark 6.4 Saying that the program- (or formula-) complexity is $\mathcal{C}$-complete means that:

1. for any formula φ (resp. structure S) the problem of deciding whether $S \models \varphi$, a problem where S (resp. φ) is the only input, belongs to $\mathcal{C}$;
2. there exist formulae (resp. structures) for which the problem is $\mathcal{C}$-hard.
□

Thus the overall complexity can be higher than the maximum of the program-complexity and the formula-complexity.

We now briefly explain how the results in Theorem 6.3 can be obtained:

Lower bound for program-complexity: In all three logics one can write a fixed formula stating that a given state is reachable (or is not reachable, in the case of *LTL*). This reachability problem is well-known to be NLOGSPACE-complete [Jon75].

Upper bound for program-complexity: For a fixed *LTL* formula φ, telling whether some Kripke structure S satisfies φ reduces to a reachability problem in the product of S and the (fixed) Büchi automaton for $\neg\varphi$, a question that can be solved in NLOGSPACE.

For CTL^* and *CTL*, the same kind of reasoning applies [KVW00] but one now reduces to an emptiness test for hesitating alternating tree-automata on a one-letter alphabet.

Formula-complexity for ***CTL*****:** The $O(|\varphi|)$ time comes from Theorem 3.1, the $O(\log|\varphi|)$ space is from Theorem 6.2.

Formula-complexity for ***LTL*** **and** CTL^***:** If we fix a finite set AP of atomic propositions, there exists a fixed Kripke structure S_{AP} where all possible AP-labeled runs can be found: if AP has k propositions, then S_{AP} is a clique with 2^k states, one for each Boolean valuation on AP. Now, checking whether $S_{AP} \models \varphi$ for an *LTL* formula φ amounts to deciding whether φ is valid, which is a PSPACE-hard problem.

Note that this uses the fact that *LTL* validity is PSPACE-hard even when AP is finite [DS02].

As a conclusion, the program-complexity of model checking is NLOGSPACE-complete, and this does not depend on whether we consider *LTL*, *CTL* or CTL^* formulae. There is a parallel here with the results from Section 3.1 where we saw that model checking for these three logics can be done in time that linearly depends on $|S|$. Finally, Table 1 is not a fair comparison of the relative merits of *CTL* and *LTL* [25].

7 The complexity of symbolic model checking

We already observed that, in practical situations, model checking mostly has to deal with large Kripke structures and small temporal formulae.

Kripke structures are frequently used to model the configurations of systems where several components interact, leading to a combinatorial explosion of the number of possible configurations. By *components*, we mean agents in a protocol, or logic gates in a circuit, or simply variables in a program (among many other possibilities). Each component can be identified and easily understood independently of the others, but the combination of their interacting behaviors makes the whole system hard to analyze without the help of a model checker.

[25] Even in terms of computational complexity, *LTL* behaves better than *CTL* on many verification problems (see [Var98, Var01]) and it seems that model checking is the only place where *CTL* can hold its ground.

Most model checkers let the user define his Kripke structures in a compositional way. But when n components interact together, and assuming that each of them only has a finite number of possible states, the resulting system has $2^{O(n)}$ possible configurations. It quickly becomes impossible to build it (e.g., to store its transitions) and one faces what is now called the *state explosion problem*.

There exist several ways to tackle the state explosion problem in model checking: the most prominent ones today are *compositional reasoning*, *abstraction methods*, *on the fly* model checking, and *symbolic* model checking [CGP99, BBF$^+$01]. In the best approaches, all these methods are combined.

7.1 Symbolic model checking

By "symbolic model checking", we mean any model checking algorithm where the Kripke structure is not described in extension (by a description having size $|S|$, as in *enumerative* methods) but is rather handled via more succinct data structures, most often some kind of restricted logical formulae for which efficient constraint-solving techniques apply [26].

Symbolic algorithms can often verify systems that defy enumerative methods (see [BCM$^+$92, McM93]). But there are also systems on which they do not perform better than the naive non-symbolic approach (an approach that can be defined as "build the structure enumeratively and then use the best model checking algorithm at hand".)

7.2 A complexity-theoretic viewpoint

From a complexity-theoretic viewpoint, there is no reason why symbolic model checking could not be solved more efficiently, even in the "worst cases", than with the naive non-symbolic approach (that always builds the resulting structure). Indeed, symbolic model checkers only deal with a very special kind of huge structures, those that have a succinct representation as a combination of small components, while the naive non-symbolic approach is not so specific and, in particular, is tuned to perform as well as possible on all huge structures. (Observe that very few huge structures have a succinct representation, as a simple cardinality argument shows).

A possible formalization of this issue is as follows: given a sequence $S_1, \ldots, S_k$ of Kripke structures, we define their composition $S = S_1 \otimes \cdots \otimes S_k$ via some combining mechanism denoted $\otimes$. In practice, some kind of

[26] The most famous example is of course the OBDD's (Ordered Binary Decision Diagrams) that made symbolic model checking so popular [BCM$^+$92]. There had been earlier attempts at symbolic model checking but their choice of data structure and constraint-solving system was less successful.

parallel combination with prescribed synchronization rules can be used: this mechanism is powerful enough to represent other ways of combining systems, like refinements *à la* Statecharts, or program with Boolean variables *à la* SMV. For the purposes of this survey, it is enough to know that in all these cases $|S|$ is in $O(\prod_{i=1}^{k}|S_i|)$, and S can be constructed in PSPACE.

Then, the *symbolic model checking problem* for a temporal logic L is the decision problem associated with the language

$$MC_{\text{symb}}(L) \stackrel{\text{def}}{=} \{\langle S_1, \dots, S_k, \varphi\rangle \mid k \in \mathbb{N}, \varphi \in L \text{ and } (S_1 \otimes \cdots \otimes S_k) \models \varphi\}.$$

The cost of algorithms solving $MC_{\text{symb}}(L)$ must be evaluated in terms of the size n of its inputs, i.e. $n = |\varphi| + \sum_i |S_i|$.

Since $|\bigotimes_i S_i|$ is in $2^{O(\sum_i |S_i|)}$, the naive non-symbolic approach (Theorems 3.1, 3.3, and Corollary 3.7) provides upper bounds of time $2^{O(n)}$ for symbolic model checking of *CTL*, *LTL*, and *CTL**.

Theorem 6.1 further provides a non-symbolic algorithm running in polynomial-space for *CTL* symbolic model checking, and this approach extends to *CTL* and *CTL** as well.

It can be proved that this upper bound is optimal:

Theorem 7.1 [KVW00] *For LTL, CTL and CTL***, symbolic model checking is* PSPACE*-complete, and its program complexity is* PSPACE*-complete.*

Remark 7.2 Let us note that PSPACE-completeness is not the universal measure for symbolic model checking: for the branching-time μ-calculus, symbolic model checking is EXPTIME-complete [Rab00].

What seems to be universal is that, given our simplifying assumptions on the cost of algorithms, non-symbolic methods perform optimally on verification problems. Indeed, this has been observed in many instances, ranging from reachability problems to equivalence problems, and can now be called an empirical fact (see [LS00b, DLS02] and the references therein). In other words, Kripke structures that admit a succinct representation are not simpler for model checking purposes than arbitrary Kripke structures. □

Table 6 contrasts the cost of model checking with that of symbolic model checking. It provides one more argument against the view that model checking is easier for *CTL* than for *LTL*.

8 Concluding remarks

So what does complexity theory have to say about model checking? A lot! For one thing, it helps identify barriers to the efficiency of model checking

	model checking	symbolic model checking
LTL	PSPACE-complete	PSPACE-complete
CTL	P-complete	PSPACE-complete
CTL^*	PSPACE-complete	PSPACE-complete

Table 6. Contrasting symbolic and non-symbolic model checking

algorithms (e.g., symbolic model checking inherently requires the full power of PSPACE). It also helps compare different temporal logics (e.g., while much more expressive than LTL, $LTL + Past$ or CTL^* are not essentially harder for model checking). Finally, it provides explanations of why there is no real difference in practice between the costs of LTL and CTL model checking: program-complexity is the important parameter! This argument applies equally to the enumerative, old-style, and the symbolic, new-style, approaches.

When does complexity theory miss the point? Many users (and most developers :–) of model checkers claim that computational complexity is only an abstract theory that bears little relationship with the actual difficulty of problems in practice. We see two situations where this criticism applies: (1) when costs must be measured precisely (i.e., when polynomial transformations are too brutal), and (2) when instances met in practice are nowhere like the hardest cases.

A fair answer to these objections is that, in principle, complexity theory is equipped with the concepts that are required in such situations (as we partly illustrated with our developments on program-complexity and complexity of symbolic model checking). The obstacles here are practical, not conceptual: fine-grained complexity and average complexity, are extremely difficult to measure.

Acknowledgements

This survey borrows a lot from investigations that I conducted with some of my colleagues: S. Demri, F. Laroussinie and N. Markey at LSV, and A. Rabinovich from Tel-Aviv.

BIBLIOGRAPHY

[AC88] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29(2):57–66, 1988.

[AI01] M. Adler and N. Immerman. An $n!$ lower bound on formula size. In *Proc. 16th IEEE Symp. Logic in Computer Science (LICS'2001)*, pages 197–206. IEEE Comp. Soc. Press, 2001.

[AN01] A. Arnold and D. Niwiński. *Rudiments of μ-Calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2001.

[BBF$^+$01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Ben85] J. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Naples, 1985.

[Ben89] J. van Benthem. Time, logic and computation. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes in Comp. Sci.*, pages 1–49. Springer, 1989.

[BG02] R. Baumgartner and G. Gottlob. Propositional default logics made easier: computational complexity of model checking. *Theoretical Computer Science*, 289(1):591–627, 2002.

[BPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BRV01] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univ. Press, 2001.

[BS01] J. C. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier Science, 2001.

[Bur84] J. P. Burgess. Basic tense logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II: Extensions of Classical Logic*, chapter 2, pages 89–133. D. Reidel Publishing, 1984.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Logics of Programs Workshop*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CG87] E. M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. In *Ann. Rev. Comput. Sci.*, volume 2, pages 269–290. Annual Reviews Inc., 1987.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[CS93] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[CS96] J. Castro and C. Seara. Complexity classes between Θ^p_k and Δ^p_k. *RAIRO Informatique Théorique et Applications*, 30(2):101–121, 1996.

[CS01] E. M. Clarke and B.-H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, vol. 2*, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Dam99] D. R. Dams. Flat fragments of CTL and CTL*: Separating the expressive and distinguishing powers. *Logic Journal of the IGPL*, 7(1):55–78, 1999.

[DLS02] S. Demri, F. Laroussinie, and Ph. Schnoebelen. A parametric analysis of the state explosion problem in model checking. In *Proc. 19th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2002)*, volume 2285 of *Lect. Notes in Comp. Sci.*, pages 620–631. Springer, 2002.

[DS02] S. Demri and Ph. Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Information and Computation*, 174(1):84–103, 2002.

[EH85] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EL87] E. A. Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science, 1990.

[ES89] E. A. Emerson and J. Srinivasan. Branching time temporal logic. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes in Comp. Sci.*, pages 123–172. Springer, 1989.

[ESS92] E. A. Emerson, T. Sadler, and J. Srinivasan. Efficient temporal satisfiability. *Journal of Logic and Computation*, 2(2):173–210, 1992.

[EVW02] K. Etessami, M. Y. Vardi, and T. Wilke. First order logic with two variables and unary temporal logic. *Information and Computation*, 179(2):279–295, 2002.

[EW00] K. Etessami and T. Wilke. An until hierarchy and other applications of an Ehrenfeucht-Fraïssé game for temporal logic. *Information and Computation*, 160(1/2):88–108, 2000.

[Fra86] N. Francez. *Fairness*. Springer, 1986.

[Gab89] D. M. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In *Proc. Workshop Temporal Logic in Specification*, volume 398 of *Lect. Notes in Comp. Sci.*, pages 409–448. Springer, 1989.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford Univ. Press, 1995.

[Gla01] R. J. van Glabbeek. The linear time – branching time spectrum I. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier Science, 2001.

[GPSS80] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. Principles of Programming Languages (POPL'80)*, pages 163–173, 1980.

[GRF00] D. M. Gabbay, M. A. Reynolds, and M. Finger. *Temporal Logic: Mathematical Foundations and Computational Aspects. Vol. 2*, volume 40 of *Oxford Logic Guides*. Clarendon Press, 2000.

[GS85] Y. Gurevich and S. Shelah. The decision problem for branching time logic. *The Journal of Symbolic Logic*, 50(3):668–681, 1985.

[Hal95] J. Y. Halpern. The effect of bounding the number of primitive propositions and the depth of nesting on the complexity of modal logic. *Artificial Intelligence*, 75(2):361–372, 1995.

[Har85] D. Harel. Recurring dominos: Making the highly undecidable highly understandable. *Annals of Discrete Mathematics*, 24:51–72, 1985.

[HC96] G. E. Hughes and M. J. Cresswell. *A new introduction to modal logic*. Routledge, London, 1996.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Int., 1991.

[Hol97] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HR83] J. Y. Halpern and J. H. Reif. The propositional dynamic logic of deterministic, well-structured programs. *Theoretical Computer Science*, 27(1–2):127–165, 1983.

[HT87] T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Proc. 14th Int. Coll. Automata, Languages, and Programming (ICALP'87)*, volume 267 of *Lect. Notes in Comp. Sci.*, pages 269–279. Springer, 1987.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. A*, chapter 2, pages 67–161. Elsevier Science, 1990.

[Jon75] N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975.

[Kam68] J. A. W. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, UCLA, Los Angeles, CA, USA, 1968.

[Kam94] Michael Kaminski. A branching time logic with past operators. *Journal of Computer and System Sciences*, 49(2):223–246, 1994.

[KG96] O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996.

[KP95] O. Kupferman and A. Pnueli. Once and for all. In *Proc. 10th IEEE Symp. Logic in Computer Science (LICS'95)*, pages 25–35. IEEE Comp. Soc. Press, 1995.

[KPV98] O. Kupferman, A. Pnueli, and M. Y. Vardi. Unpublished proof. Private communication with O. Kupferman, January 1998.

[Kre88] M. W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509, 1988.

[KS02] A. Kučera and J. Strejček. The stuttering principle revisited: On the expressiveness of nested X and U operators in the logic LTL. In *Proc. 16th Int. Workshop Computer Science Logic (CSL'2002)*, volume 2471 of *Lect. Notes in Comp. Sci.*, pages 276–291. Springer, 2002.

[Kur95] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1995.

[KVW00] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[Lam80] L. Lamport. "Sometimes" is sometimes "Not Never". In *Proc. 7th ACM Symp. Principles of Programming Languages (POPL'80)*, pages 174–185, 1980.

[Lam83] L. Lamport. What good is temporal logic? In *Information Processing'83. Proc. IFIP 9th World Computer Congress*, pages 657–668. North-Holland, 1983.

[Lar95] F. Laroussinie. About the expressive power of CTL combinators. *Information Processing Letters*, 54(6):343–345, 1995.

[LMS01] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Model checking CTL^+ and $FCTL$ is hard. In *Proc. 4th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2001)*, volume 2030 of *Lect. Notes in Comp. Sci.*, pages 318–331. Springer, 2001.

[LMS02a] F. Laroussinie, N. Markey, and Ph. Schnoebelen. On model checking durational Kripke structures (extended abstract). In *Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2002)*, volume 2303 of *Lect. Notes in Comp. Sci.*, pages 264–279. Springer, 2002.

[LMS02b] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *Proc. 17th IEEE Symp. Logic in Computer Science (LICS'2002)*, pages 383–392. IEEE Comp. Soc. Press, 2002.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Principles of Programming Languages (POPL'85)*, pages 97–107, 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proc. Logics of Programs Workshop*, volume 193 of *Lect. Notes in Comp. Sci.*, pages 196–218. Springer, 1985.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148(2):303–324, 1995.

[LS00a] F. Laroussinie and Ph. Schnoebelen. Specification in CTL+Past for verification in CTL. *Information and Computation*, 156(1/2):236–263, 2000.

[LS00b] F. Laroussinie and Ph. Schnoebelen. The state explosion problem from trace to bisimulation equivalence. In *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000)*, volume 1784 of *Lect. Notes in Comp. Sci.*, pages 192–207. Springer, 2000.

[Mar02] N. Markey. Past is for free: on the complexity of verifying linear temporal properties with past. In *Proc. 9th Int. Workshop on Expressiveness in Concurrency (EXPRESS'2002)*, volume 68.2 of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 2002.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall Int., 1989.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes in Comp. Sci.*, pages 201–284. Springer, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[ON80] H. Ono and A. Nakamura. On the size of refutation Kripke models for some linear modal and tense logics. *Studia Logica*, 39(4):325–333, 1980.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[Pri67] A. N. Prior. *Past, Present, and Future.* Clarendon Press, Oxford, 1967.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Int. Symp. on Programming*, volume 137 of *Lect. Notes in Comp. Sci.*, pages 337–351. Springer, 1982.

[Rab69] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[Rab00] A. Rabinovich. Symbolic model checking for μ-calculus requires exponential time. *Theoretical Computer Science*, 243(1–2):467–475, 2000.

[Rab02] A. Rabinovich. Expressive power of temporal logics. In *Proc. 13th Int. Conf. Concurrency Theory (CONCUR'2002)*, volume 2421 of *Lect. Notes in Comp. Sci.*, pages 57–75. Springer, 2002.

[Rei89] W. Reisig. Towards a temporal logic for causality and choice in distributed systems. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes in Comp. Sci.*, pages 603–627. Springer, 1989.

[RM01] A. Rabinovich and S. Maoz. An infinite hierarchy of temporal logics over branching time. *Information and Computation*, 171(2):306–332, 2001.

[RS00] A. Rabinovich and Ph. Schnoebelen. BTL_2 and expressive completeness for $ECTL^+$. Research Report LSV-00-8, Lab. Specification and Verification, ENS de Cachan, Cachan, France, October 2000.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Sch01] Ph. Schnoebelen. Spécification et vérification des systèmes concurrents. Mémoire d'habilitation à diriger des recherches, Université Paris 7, October 2001.

[Sch03] Ph. Schnoebelen. Oracle circuits for branching-time model checking. In *Proc. 30th Int. Coll. Automata, Languages, and Programming (ICALP'2003)*, to appear in *Lect. Notes in Comp. Sci.*. Springer, July 2003.

[Sti92] C. Stirling. Modal and temporal logics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, vol.2. Background: Computational Structures*, pages 477–563. Oxford Univ. Press, 1992.

[Sto76] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[Sto87] L. J. Stockmeyer. Classifying the computational complexity of problems. *The Journal of Symbolic Logic*, 52:1–43, 1987.

[SVW87] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3):217–237, 1987.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 4, pages 133–191. Elsevier Science, 1990.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. Theory of Computing (STOC'82)*, pages 137–146, 1982.

[Var95] M. Y. Vardi. Alternating automata and program verification. In *Computer Science Today. Recent Trends and Developments*, volume 1000 of *Lect. Notes in Comp. Sci.*, pages 471–485. Springer, 1995.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure Versus Automata*, volume 1043 of *Lect. Notes in Comp. Sci.*, pages 238–266. Springer, 1996.

[Var98] M. Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *Proc. 13th IEEE Symp. Logic in Computer Science (LICS'98)*, pages 394–405. IEEE Comp. Soc. Press, 1998.

[Var01] M. Y. Vardi. Branching vs. linear time: Final showdown. In *Proc. 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2001)*, volume 2031 of *Lect. Notes in Comp. Sci.*, pages 1–22. Springer, 2001.

[VB00] W. Visser and H. Barringer. Practical CTL* model checking: Should SPIN be extended? *Journal of Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Comp. Soc. Press, 1986.

[Wag87] K. W. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theoretical Computer Science*, 51(1–2):53–80, 1987.

[Wil99a] T. Wilke. Classifying discrete temporal properties. In *Proc. 16th Ann. Symp. Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *Lect. Notes in Comp. Sci.*, pages 32–46. Springer, 1999.

[Wil99b] T. Wilke. CTL$^+$ is exponentially more succint than CTL. In *Proc. 19th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'99)*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 110–121. Springer, 1999.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

[Wol89] P. Wolper. On the relation of programs and computations to models of temporal logic. In *Proc. Workshop Temporal Logic in Specification*, volume 398 of *Lect. Notes in Comp. Sci.*, pages 75–123. Springer, 1989.

[Wol01] P. Wolper. Constructing automata from temporal logic formulas: A tutorial. In *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lect. Notes in Comp. Sci.*, pages 261–277. Springer, 2001.

[ZC93] A. Zanardo and J. Carmo. Ockhamist computational logic: Past-sensitive necessitation in CTL*. *Journal of Logic and Computation*, 3(3):249–268, 1993.