# Logics of Time and Action

Frank Wolter and Michael Wooldridge

Department of Computer Science,
University of Liverpool
Liverpool L69 3BX, UK
{wolter,mjw}@liverpool.ac.uk

March 2, 2012

**Abstract**

We present an introductory survey of temporal and dynamic logics: logics for reasoning about how environments change over time, and how processes change their environments. We begin by introducing the historical development of temporal and dynamic logic, starting with the seminal work of Prior. This leads to a discussion of the use of temporal and dynamic logic in computer science. We describe three key formalisms used in computer science for reasoning about programs (LTL, CTL, and PDL), and illustrate how these formalisms may be used in the formal specification and verification of computer systems. We then discuss interval temporal logics. We conclude with some pointers for further reading.

# 1 Introduction

Mathematical logic was originally developed with the goal of formalising mathematical reasoning – to apply mathematical rigour to notions such as truth and proof. One important property of mathematical expressions such as theorems and their proofs is that they are inherently *timeless*: a result such as Fermat's Theorem is true now, always has been true, and always will be true – irrespective of when it was actually proved. In this sense, mathematical logic was conceived with the goal of developing formal languages for representing a fixed, non-changing world, and the semantics of classical logic reflect this assumption. In the semantics of classical logic, it is assumed that there is exactly one world (called a model), which satisfies or refutes any given sentence. But this limits the applicability of such logics for reasoning about *dynamic* domains of discourse, where the truth status of statements can change over time.

It is of course possible to reason about time-varying domains using classical first-order logic. One obvious approach is to use a two-sorted language, in which we have

one sort for the domain of discourse, and a second sort for points in time. Variables $t, t', \ldots$ are used to denote time points, and an in-place binary "earlier than" predicate, "$<$", is used to capture the temporal ordering of time points. First-order predicates of arity $n$ are translated into predicates of arity $n+1$, with the additional argument being a term denoting a time point. Using this approach, for example, the English sentence "It is never hot in Liverpool" might be translated into the following first-order formula:

$$\forall t. \neg Hot(Liverpool, t)$$

The statement "Frank and Michael will eventually be rich" might be translated into:

$$\exists t, t'. (now < t) \wedge (now < t') \wedge Rich(Frank, t) \wedge Rich(Michael, t')$$

Notice that we use the term $now$ to denote the current time point.

This technique is sometimes called the *method of temporal arguments* [50], or simply the *first-order approach* [25]. The advantage of the approach is that no extra logical apparatus needs to be introduced to deal with time: the entire machinery of standard first-order logic can be brought to bear directly. The obvious disadvantage is that the approach leads to first-order formulae that are difficult for people to interpret. First-order formulae representing quite trivial temporal properties are often large, complicated, and hard to understand. For example, the English sentence "we are not friends until you apologise", when translated to first-order logic, becomes something like the following:

$$\begin{array}{l} \exists t. [(now < t) \wedge Apologise(you, t)] \wedge \\ \quad \forall t'. [(now < t' < t) \rightarrow \neg Friends(us, t')]. \end{array}$$

The structure of the original (rather simple!) English sentence is cleary lost, and some thought is required to recover the meaning of the formula.

The desire for logics that are capable of naturally and transparently capturing the meaning of statements such as those above, in dynamic environments, led to the development of specialised *temporal* and *dynamic* logics. This article is intended as a high-level introductory survey of such logics. Contemporary research in temporal and dynamic logics is a huge and very active enterprise, with participation from disciplines ranging from philosophy and linguistics to computer science [29]. Within the latter, it is especially the application of temporal and dynamic logics to verifying the correctness of computer systems that had a huge impact on the field. In an amazing example of technology transfer, this application has transformed purely philosophical logics into industrial-strength software analysis tools [57].

Before taking a look at such recent developments, however, we reflect on the origins of temporal logic, and in particular, the contributions of Arthur Prior.

# 2 The Origins of Temporal Logic

As we noted in the introduction, in classical logic it is implicitly assumed that formulae are interpreted with respect to a single model. But this inherently static view of logic

and its subject matter makes it awkward to apply classical logic to the analysis of everyday sentences such as "Barack Obama will win the election", since this statement might be true if evaluated now, but false if evaluated next week. It was concerns like this that led Arthur Prior, a philosopher and logician born in New Zealand in 1914, to start working on logics intended to facilitate reasoning about such statements. In Prior's words [49]: "Certainly there are unchanging truths, but there are changing truths also, and it is a pity if logic ignores these, and leaves it ... to comparatively informal dialecticians to study the more dynamic aspects of reality." Prior's work, mainly carried out in the 1950s and 1960s, is regarded as the foundation of the area now called temporal and dynamic logic [47, 48].

To analyse sentences such as "Barack Obama will win the election", Prior proposed the idea of regarding tense as a species of *modality*. He took classical propositional logic with its connectives $\vee$ (for "or"), $\neg$ ("not"), and $\rightarrow$ ("if ..., then ...") and extended it with modal *tense operators*, $F$ and $P$. In Prior's notation, $Fp$ stands for "it will be the case that $p$" and $Pp$ stands for "it was the case that $p$". In a similar way as in classical logic, one can define other basic tense operators as composed formulae. For example, the expression $Gp$ (read "generally, $p$") is defined as $\neg F \neg p$ (meaning "$p$ will always be the case") and $Hp$ ("heretofore, $p$") is defined as $\neg P \neg p$ (meaning "$p$ has always been the case"). Other temporal operators are also sometimes used, such as $\neg F \neg F p$ ("$p$ will be the case again and again").

Given this set-up, in addition to classical tautologies, (sentences like $p \rightarrow p$, which are true independently of the model under consideration), one should also consider temporal tautologies: formulae using tense operators that are true independently from the truth of its propositional atoms. Priors first axiomatization of temporal tautologies included formulae such as

$$FFp \rightarrow Fp, \quad \text{and} \quad F(p \vee q) \rightarrow (Fp \vee Fq).$$

The first formula, for example, states "if it will be the case that it will be the case that $p$, then it will be the case that $p$". A less obvious candidate for a temporal tautology is its "converse" $Fp \rightarrow FFp$. A moment's reflection should convince the reader that the truth of this formula depends on precisely how $Fp$ is interpreted. In other words, to decide whether this formula is a temporal tautology one has to define a formal semantics for the temporal language. Of course, one cannot interpret Prior's formulae in the static, non-changing models of classical logic. Instead one has to develop models that capture the evolution of reality over time. Leaving aside the question of what a physicist might have to say about this issue, one obvious and mathematically simple approach is to interpret time as a linear sequence of time points:

$$t_0, t_1, t_2, \ldots$$

These time points can be naturally interpreted as, say, dates in a calendar. Mathematically, we can view such a flow of time as the natural numbers $\mathbb{N}$, ordered by the usual "less than" relation, "$<$". Many other models of time are also possible, with correspondingly different properties. For example, a model of *dense time* is given by letting

the set of time points be the real numbers $\mathbb{R}$ ordered by the less-than relation "$<$". In such a dense model of time, for every pair of time points $t_1, t_2$ such that $t_1 < t_2$, there exists a time point $t_3$ such that $t_1 < t_3 < t_2$.

Now, in contrast to classical logic, a temporal logic formula can be true at one time point, but false at another time point. Thus, we obtain a time-dependent notion of truth: a formula might be true when evaluated at $t_i$ and false when evaluated at $t_{i+1}$. Formally, each time point $t_i$ comes with a truth assignment stating which propositional atoms $p$ are true at $t_i$. The propositional connectives are interpreted as in classical logic (for example $\phi \wedge \psi$ is true at $t_i$ if, and only if, $\phi$ and $\psi$ are both true at $t_i$). Finally,

$Fp$ is true at $t_i$ if, and only if, there exists $j > i$ such that $p$ is true at $t_j$;

$Pp$ is true at $t_i$ if, and only if, there exists $j < i$ such that $p$ is true at $t_j$.

Assuming, for example, that $p$ is true at $t_{100}$ and no other time point, then $Fp$ is true at $t_0$. In fact, it is true at all time points between (and including) $t_0$ and $t_{99}$, but not at $t_{100}$ nor any time point after $t_{100}$. Let us check that $FFp \rightarrow Fp$ is true in every time point no matter at which points $p$ is true. To this end, assume that $FFp$ is true at, say, $t_n$. Then $Fp$ is true at some time point after $t_n$, say $t_m$. Similarly, this means that $p$ is true at some time point after $t_m$, say $t_k$. But then $t_k$ is a time point after $t_n$ and we obtain that $Fp$ is true at $t_n$. We have shown that $Fp$ is true at any given time point if $FFp$ is true at that time point. Thus, $FFp \rightarrow Fp$ is true at every time point, independently from $p$. Note that what we have applied here is the natural assumption that temporal precedence is transitive: if $t_k$ is after $t_m$ and $t_m$ is after $t_n$, then $t_k$ is after $t_n$. In contrast, the truth of $Fp \rightarrow FFp$ in this model of time depends on $p$. For example, assume that $p$ is true at $t_5$ and no other time point. Then $Fp$ is true at $t_4$, but $FFp$ is not true at $t_4$: it is not possible to find a time point after $t_4$ that is before $t_5$. In fact, it is not difficult to see that a time model $(T, <)$ with set of time points $T$ and temporal precedence relation $<$ validates $Fp \rightarrow FFp$ if, and only if, it is *dense*, in the sense that we described above: for any two time points $t_1 < t_2$ there is a time point $t'$ such that $t_1 < t' < t_2$. Thus, if we move from the *discrete* time model $t_0, t_1, \ldots$ to a *dense* model of time (e.g., based on the rational or real numbers), we obtain new temporal tautologies (and loose others).

This small example illustrates one of the main distinctions between classical and temporal logics: even if the language is fixed and as simple as the basic tense logic with operators $F$ and $P$, there remains a number of choices to be made with respect to the model used to represent time. Depending on the temporal domain and discourse of interest, one can choose between flows of time that are discrete, dense, or continuous; time can be cyclic or cycle-free; time can be bounded or unbounded in the past, and unbounded or bounded in the future. The are many possibilities, and in the late 1960s and early 1970s it became something of an industry to axiomatize and analyse the temporal tautologies of such time models.

In addition to varying the flow of time, additional tense operators were introduced and investigated. Of particular interest are the binary operators $S$ (for *since*) and $U$

(for *until*) whose semantics is defined as follows:

> $pUq$ is true in $t$ if, and only if, there exists $t' > t$ such that $q$ is true in $t'$ and $p$ is true in $t''$ for all $t''$ such that $t < t'' < t'$;

> $pSq$ is true in $t$ if, and only if, there exists $t' < t$ such that $q$ is true in $t'$ and $p$ is true in $t''$ for all $t''$ such that $t' < t'' < t$.

Note that $Fp$ and $Pp$ can be expressed using since and until as

$$Fp = \top Up \quad \text{and} \quad Pp = \top Sp,$$

where $\top$ is a propositional constant standing for a propositional tautology. For axiomatizations of temporal tautologies for languages with operators $F$, $P$, $S$, and $U$ for various time flows see [9].

# 3 Temporal *versus* Predicate Logic

In our introduction to Prior's basic tense logic, we emphasised that the main difference between classical and temporal logic is time-dependence: in classical logic truth is time-independent, whereas in temporal logic it is not. Under this view, temporal languages are extensions of propositional logic by means of temporal operators. There is, however, a very different and equally important interpretation of temporal logic, namely as a fragment of predicate logic (see [5] for a general discussion of these two different views in modal logic). To achieve this, propositional atoms are identified with unary predicates and complex temporal formulae become predicate logic sentences with exactly one free variable $x$ that ranges over time points. Consider, for example, the sentence "the mail will be delivered". In Prior's tense logic, this is formalised as $Fp$, where $p$ stands for "the mail is delivered". In contrast, in predicate logic one introduces a unary predicate $P$ ranging over time points for "the mail is delivered" and the sentence is formalised as $\exists y(x < y \wedge P(y))$, where $<$ stands for temporal precedence.

To make the connection between temporal and predicate logic precise, consider a flow of time $(T, <)$ which can be, for example, the discrete time flow $t_0, t_1, \ldots$ or a copy of the rational or real numbers. A valuation $v$ determines at which time points $t \in T$ an atom $p$ from a given set $\Phi$ of propositional atoms is true. Equivalently, we can describe the resulting model by identifying $(T, <, v)$ with the first-order relational model

$$M = (T, <, (p^M \mid p \in \Phi)),$$

where $p^M \subseteq T$ denotes the set of time points at which $p$ is true. Thus, now we regard the $p \in \Phi$ as unary predicates that have as extensions a set of time points. Inductively,

we can translate every temporal logic formula $\phi$ in the language with, say, $S$ and $U$, as a first-order predicate logic formula $p^\sharp(x)$, where $x$ is a fixed individual variable:

$$\begin{array}{rcl} p^\sharp & = & p(x) \\ (\phi \wedge \psi)^\sharp & = & \phi^\sharp(x) \wedge \psi^\sharp(x) \\ (\neg\phi)^\sharp & = & \neg\phi^\sharp(x) \\ (\phi S \psi)^\sharp & = & \exists y(y < x \wedge \psi^\sharp(y) \wedge \forall z((y < z < x) \rightarrow \phi^\sharp(z))) \\ (\phi U \psi)^\sharp & = & \exists y(y > x \wedge \psi^\sharp(y) \wedge \forall z((x < z < y) \rightarrow \phi^\sharp(z))) \end{array}$$

where $y, z$ are fresh individual variables and $\phi^\sharp(y)$ and $\phi^\sharp(z)$ are obtained from $\phi^\sharp(x)$ by replacing $x$ with $y$ and $z$, respectively (see [29] for details).

By definition of $\cdot^\sharp$, we have that a temporal logic formula $\phi$ is true at a time point $t$ in a model $M$ if, and only of $M \models \phi^\sharp[t]$, where $\models$ is the standard truth-relation of first-order predicate logic. Thus, modulo the translation $\cdot^\sharp$, we can regard the temporal language with operators $S$ and $U$ as a fragment of first-order predicate logic. The formulae obtained as translations of temporal formulae are, of course, only a tiny subset of the set of all first-order predicate logic formulae (even those with one free variable $x$ using only unary predicates $p \in \Phi$ and the binary predicate $<$). Moreover, for arbitrary time flows, there are many first-order predicate formulae of this form that are *not equivalent* to any translation of a temporal formula. However, a fundamental result proved by Hans Kamp [33] established that, for some important flows of time, every first-order predicate logic formula in the language with unary predicates $p \in \Phi$ and the binary predicate $<$ and having exactly one free variable is indeed equivalent to a temporal formula using since and until. The precise formulation is as follows:

**Theorem 1 (Kamp)** *Let $(T, <)$ be a flow of time consisting of the natural or real numbers. Then one can construct for every first-order formula $\varphi(x)$ using $<$ and $p \in \Phi$ and with one free variable $x$, a temporal formula $\varphi^T$ using the operators $S$ and $U$ such that the following holds for every model $M = (T, <, (p^M \mid p \in \Phi))$ and every $t \in T$:*

$$\varphi^T \textit{ is true at } t \quad \Leftrightarrow \quad M \models \varphi[t]$$

Kamp's theorem explains why there are only very few important distinct temporal operators for linear time: any first-order definable temporal operator can be expressed using just since and until. We would like to stress here that it would be wrong to conclude from Kamp's result that temporal logic has nothing useful to offer compared to predicate logic. As we pointed out in the introduction, the crucial difference between temporal and predicate logic is that temporal logic is much closer to natural language than predicate logic and, therefore, much easier for people to read and understand. Thus, for the same reason that programming languages such as C or JAVA are not useless just because any program in C or JAVA is equivalent to a Turing Machine, temporal logics do not become useless just because they have the same expressive power as first-order formulae.

Interestingly, the difference between temporal and first-order logic can also be described in technical terms. The translation from first-order predicate logic to temporal logic introduces temporal formulae of non-elementary size (i.e., their size cannot be bounded by a tower of exponentials) [23] and for standard linear time flows the satisfiability problem for temporal logic with $S$ and $U$ is PSPACE-complete (and even coNP-complete with operators $F$ and $P$ only), but it is non-elementary for the corresponding fragment of first-order predicate logic [23, 29, 35].

Kamp's result was the beginning of a long and ongoing research tradition. Results such as Kamp's are nowadays known as *expressive completeness* results. A typical expressive completeness result states that a certain temporal language is equivalent to (some fragment of) first-order predicate logic over a certain class of time flows. For example, Prior's original language with the tense operators $F$ and $P$ only and without since and until is expressively complete for the *two-variable fragment* of first-order predicate logic (i.e., first-order predicate sentences using only two individual variables) on arbitrary linear time flows [15, 36]. An overview of recent extensions and variations of Kamp's theorem is given in [29].

# 4 Temporal Logic in Computer Science

At this point in our story, computer science enters the scene. A key research topic in computer science is the *correctness problem*: crudely, the problem of showing that computer programs operate correctly [7]. Two key issues associated with correctness are the related problems of *specification* and *verification*. A specification is an exact description of the behaviour that we want a particular computer system to exhibit. Verification is the problem of demonstrating that a particular program does or does not behave as a particular specification says it should.

Temporal logic has proved to be an extremely valuable formalism for the specification and verification of computer systems. This application of temporal logic is largely due to the work of Amir Pnueli, an Israeli logician born in 1941. In 1977, Pnueli was considering the problem of specifying a class of computer programs known as *reactive systems*. A reactive system is one that does not simply compute some function and terminate, but rather has to maintain an *ongoing interaction* with its environment. Examples of reactive systems include computer operating systems and process control systems. Typical properties that we might find in the specification of a reactive system are *liveness* and *safety* properties. Intuitively, liveness properties relate to programs correctly progressing, while safety properties relate to programs avoiding undesirable situations. In a seminal paper [43], Pnueli observed that temporal logics of the type introduced by Prior provide an elegant and natural formal framework with which to specify and verify liveness and safety of reactive systems. For example, suppose $p$ is a predicate describing a particular undesirable property of a program (a "system crash", for example). Then the temporal formula $G\neg p$ formally expresses the requirement that the program does not crash. The requirement $G\neg p$ is an example of a safety property.

Thus, Pnueli's idea was to formally specify the desirable behaviour of a reactive computer system as a formula $\Psi$ of temporal logic. Such a *formal* specification is valuable its own right, as a precise, mathematical description of the intended behaviour of the program. But it also opens up the possibility of formal verification, as follows. Suppose we are given a computer program $\mathcal{P}$, (written in a programming language such as PASCAL, C, or JAVA), and a specification $\Psi$, expressed as a formula of temporal logic. Then the idea of *deductive temporal verification* is to first derive the *temporal theory* $Th(\mathcal{P})$ of $\mathcal{P}$, i.e., a logical theory which expresses the actual behaviour of $\mathcal{P}$. The temporal theory $Th(\mathcal{P})$ of the program $\mathcal{P}$ is derived from the text of the program $\mathcal{P}$. For example, for each program statement in $\mathcal{P}$, there will typically be a collection of axioms in the temporal theory $Th(\mathcal{P})$, which collectively characterise the effect of that statement. To do verification, we attempt to show that $Th(\mathcal{P}) \vdash \Psi$. If we succeed, then we say that $\mathcal{P}$ *satisfies* the specification $\Psi$; it is *correct* with respect to $\Psi$. In this way, verification reduces to a proof problem in temporal logic.

Pnueli's insight led to enormous interest in the use of temporal logic in the formal specification and verification of reactive systems, and ultimately, to software verification tools that are used in industry today. One question that attracted considerable interest in the 1980s was that of exactly what kind of temporal logic is best suited to program specification and verification. Although one can use Prior's logics for reasoning about programs, they are not well suited to talking about the "fine structure" of the state sequences generated by programs as they execute. For this reason, a great many different proposals were made with respect to temporal logics for reasoning about programs (see, e.g., [34, 3, 52, 59, 1, 14]). In the end, two key formalisms emerged from this debate: *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL). These two formalisms are intended to capture different aspects of computation. LTL describes properties of a single run of a reactive program. Hence it is interpreted over a *linear* sequence of successive machine states. In contrast, CTL describes properties of the *branching* structure of the set of all possible runs of the program.

## 4.1 From Programs to Flows of Time

Before presenting the semantics of LTL and CTL, let us pause to consider in a little more detail exactly how computer programs give rise to flows of time. Consider the following (admittedly rather pointless) computer program, written in a PASCAL/C-like language.

```
x = y = true;
while (true) do
  if x == false then
    x = true;
  else
    x = false;
  end-if;
```

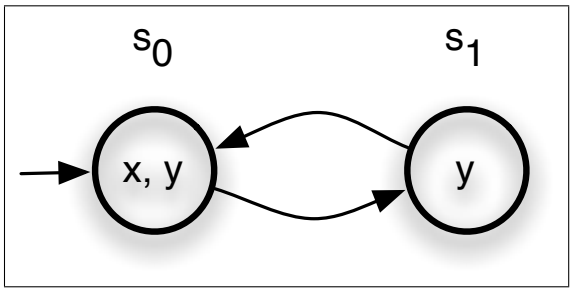


Figure 1: A state transition graph.

```
end-while;
```

Thus, this program manipulates two Boolean-valued program variables, `x` and `y`; the variable `x` is initialised to the value `true`, and then its value is subsequently flipped between the values `true` and `false`. The variable `y` is initialised to `true` and remains unchanged subsequently. Notice that the program never terminates – it is an infinite loop. Now, we can understand the behaviour of this program as a *state transition system*, as illustrated in Figure 1. The state transition graph contains the possible states, or configurations, of the program; edges between states correspond to the execution of individual program instructions. For the program above, there are just two possible system states, labelled $s_0$ and $s_1$. The variables `x` and `y` are both true in $s_0$, while `x` is false and `y` is true in $s_1$; the arrow to state $s_0$ indicates that this is the *initial* state of the system. The other edges in the graph indicate that, when the program is in state $s_0$, then the only possible next state of the system is $s_1$, while when the program is in state $s_1$, the only possible next state of the system is $s_0$. Now, if we want to reason about the program given above, then we can focus on the state transition graph: this graph completely captures the behaviour of the program.

A little more formally, a state transition system is a triple:

$$M = (S, R, V)$$

where:

- $S$ is a non-empty set of *states*;
- $R \subseteq S \times S$ is a total[1] binary relation on $S$, which we refer to as the *transition relation*; and
- $V : S \to 2^{\Phi}$ labels each state with the set of propositional atoms true in that state.

State transition systems are fundamental to the use of temporal logics for reasoning about programs. To make the link to temporal logic, we need a little more notation and

[1]By totality we here mean the property that every state has a successor, i.e., for every $s \in S$ there is a $t \in S$ such that $(s, t) \in R$.

terminology. A *path*, $\rho$, over $M$ is an infinite sequence of states $\rho = s_0, s_1, \ldots$ which must satisfy that $(s_n, s_{n+1}) \in R$ for all natural numbers $n$. In the program given above, the state transition system is completely deterministic, in the sense that there is only ever one possible next state of the system, and so there is in fact only one path possible through the transition system of the program:

$$\rho : s_0, s_1, s_0, s_1, \ldots$$

However, in general, state transition systems are *non-deterministic*, in the sense that, for any given state $s_i$ in the state transition graph, there can be multiple outgoing edges from $s_i$. This non-determinism can be thought of as reflecting the choices available to the program itself, or as the program's environment (e.g., its user) interacting with the program. So, in general, there may be more than one possible path through a transition system. The set of all possible paths through a state transition system will completely characterise the behaviour of the program: the paths in a transition system are exactly the possible *runs* of the program. Figure 2 shows how a transition system (Figure 2(a)) can be "unravelled" into a set of paths (Figure 2(b)). Of course, we do not show all the paths of the transition system in Figure 2 – the reader should be able to easily convince themselves that there are an infinite number of such paths, and these paths are infinitely long – even though the transition system that generates them is finite. Now, going back to temporal logic, a path is simply a linear, discrete sequence of time points (now called states), and we can think of such a path as a flow of time, in exactly the way that we discussed earlier. Thus temporal formulae of the kind studied by Prior's logics can be used to express properties of the runs of programs.

So far, we have three different views of programs: (*i*) the original program text, written in a programming language like PASCAL or C, above; (*ii*) the state transition diagram of the program, as in Figure 1 and Figure 2(a); and (*iii*) the runs obtained from the state transition diagram by "unravelling" it (Figure 2(b)). However, a third view is also possible: we can unravel the transition system into a *computation tree*, as shown in Figure 2(c). The key difference between the logics LTL and CTL is that the language of LTL is intended for representing properties of individual computation paths, while the language of CTL is intended for representing properties of computation trees of the type shown in Figure 2(c).

In the following subsections, we will take a closer look at the technical frameworks of LTL and CTL.

## 4.2 Linear Temporal Logic – LTL

In this section, we will present and investigate the framework of LTL in a little more detail. In the particular version that we work with, we will only consider temporal operators that refer to the future; it is also possible to consider LTL operators that refer to the past, although we will not do so here [12]. LTL extends classical propositional logic with the unary modal operator $X$ ("next") and the binary operator $U$ ("until").

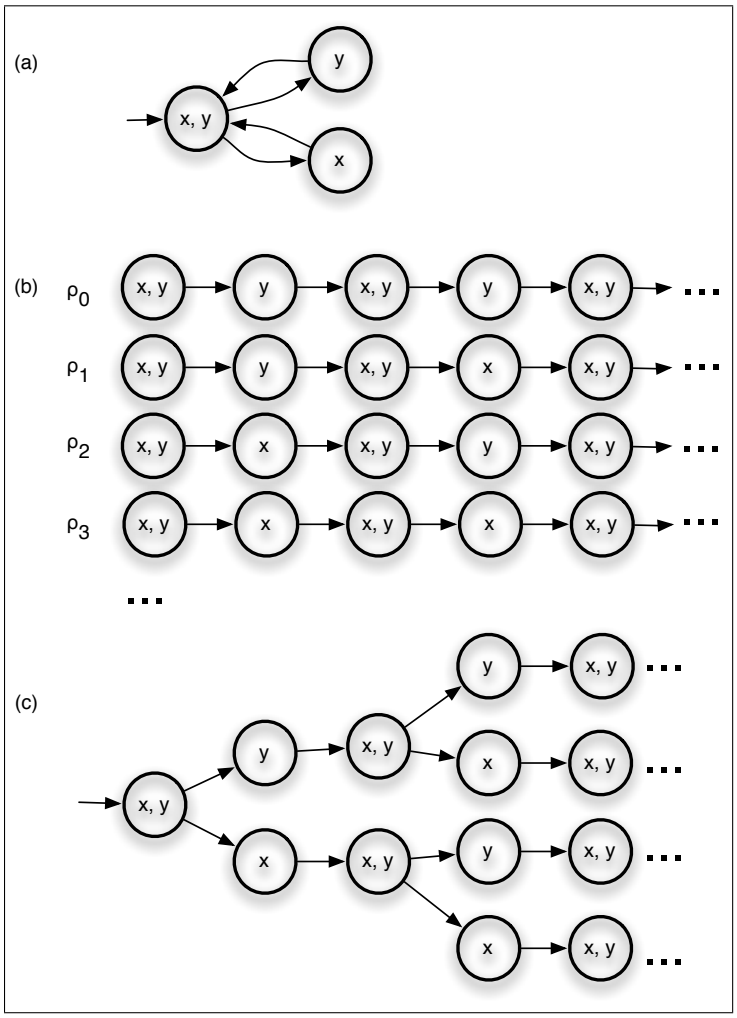


Figure 2: A state transition diagram (a), can be "unravelled" into a set of runs (b), or viewed as a tree-like branching model of time (c).

Formally, starting with a set $\Phi$ of propositional atoms, the syntax of LTL is defined by the following grammar:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \vee \phi \mid X\phi \mid \phi U \phi.$$

where $p \in \Phi$. A model for LTL is a path $\rho$ in a transition system $M = (S, R, V)$. If $u \in \mathbb{N}$, then we denote by $\rho[u]$ the element indexed by $u$ in $\rho$ (thus $\rho[0]$ denotes the first element, $\rho[1]$ the second, and so on). The satisfaction relation "$\rho, u \models \phi$" between pairs $\rho, u$ and LTL formulae $\phi$ formalises the condition "after $u$ steps of the computation given by $\rho$, the formula $\phi$ holds" and is inductively defined via the following rules:

$$\rho, u \models \top$$

$\rho, u \models p$ iff $p \in V(\rho[u])$ (where $p \in \Phi$)

$\rho, u \models \neg\phi$ iff $\rho, u \not\models \phi$

$\rho, u \models \phi \vee \psi$ iff $\rho, u \models \phi$ or $\rho, u \models \psi$

$\rho, u \models X\phi$ iff $\rho, u+1 \models \phi$

$\rho, u \models \phi U \psi$ iff there exists $v \geq u$ such that $\rho, v \models \psi$ and for all $w$ such that $u \leq w < v$, we have $\rho, w \models \psi$.

(It is worth mentioning that the semantics of LTL can equivalently be defined using a flow of time $t_1, t_2, \ldots$ and without introducing an underlying transition system $M$. This is the viewpoint taken in our discussion of Prior's temporal logics. The main purpose of introducing state transition systems is to make explicit the computational interpretation of LTL and to enable the comparison with CTL in the next section.)

The reader will have noticed that the temporal operator $U$ has been given a slightly different interpretation here: previously we had the "strict" interpretation of $\phi U \psi$ according to which $\phi U \psi$ is true at $t$ if $\psi$ is true some time $t'$ later than $t$ and $\phi$ is true at all time points properly between $t$ and $t'$. This interpretation is typically seen in philosophical temporal logic motivated by capturing the semantics of natural language tense constructs. In contrast, in typical computer science temporal logic $t'$ can be $t$ itself or later. The same applies to the definition of $F$ and $G$ in terms of $U$. As before we define the temporal connectives $F$ and $G$ by setting

$$F\phi = \top U \phi \qquad G\phi = \neg F \neg \phi.$$

As the truth condition of $U$ has changed, so have the truth conditions of $F\phi$ and $G\phi$. $F\phi$ means "either now or at some time later $\phi$ hold" and $G\phi$ means "now and always in the future $\phi$ holds". In what follows these distinctions will not play any important role.

Referring back to Figure 2(a), consider the path $\rho_0$. The following temporal properties may be seen to hold:

- $\rho_0, 0 \models x \wedge y$
- $\rho_0, 0 \models X(y \wedge \neg x)$
- $\rho_0, 0 \models XXX(y \wedge \neg x)$
- $\rho_0, 1 \models y \wedge \neg x$
- $\rho_0, 0 \models F(y \wedge \neg x)$
- $\rho_0, 0 \models GF(x \wedge y)$

However, considering path $\rho_1$, we have for example:

- $\rho_1, 0 \models X(y \wedge \neg x)$
- $\rho_1, 0 \models XXX(x \wedge \neg y)$

| | |
|---|---|
| Axioms: | |
| $(LAX1)$ | propositional tautologies |
| $(LAX2)$ | $\neg X\phi \leftrightarrow X\neg\phi$ |
| $(LAX3)$ | $X(\phi \rightarrow \psi) \rightarrow (X\phi \rightarrow X\psi)$ |
| $(LAX4)$ | $G(\phi \rightarrow \psi) \rightarrow (G\phi \rightarrow G\psi)$ |
| $(LAX5)$ | $G\phi \rightarrow (\phi \wedge XG\phi)$ |
| $(LAX6)$ | $G(\phi \rightarrow X\phi) \rightarrow (\phi \rightarrow G\phi)$ |
| $(LAX7)$ | $(\phi U\psi) \rightarrow F\psi$ |
| $(LAX8)$ | $(\phi U\psi) \leftrightarrow (\psi \vee (\phi \wedge X(\phi U\psi))$ |
| Inference Rules: | |
| $(LIR1)$ | From $\vdash \phi \rightarrow \psi$ and $\vdash \phi$ infer $\vdash \psi$ |
| $(LIR2)$ | From $\vdash \phi$ infer $\vdash G\psi$ |

Table 1: A complete axiomatization for LTL.

### 4.2.1 Liveness and Safety Properties

Let us take a moment to consider the types of properties that LTL may be used to specify. As we noted above, it is generally accepted that such properties fall into two categories: *safety* and *liveness* properties[2]. Informally, a safety property can be interpreted as saying that "something bad won't happen". For obvious reasons, safety properties are sometimes called invariance properties. The simplest kind of safety property is a *global invariant*, expressed by a formula of the form: $G\phi$. A *mutual exclusion* property is a global invariant of the form: $G\left(\sum_{i=1}^{n} \phi_i \leq 1\right)$. This formula states that at most one of the properties $\phi_i \in \{\phi_1, \ldots, \phi_n\}$ should hold at any one time. (The $\Sigma$ notation is readily understood if one thinks of truth being valued at 1, falsity at 0.) A *local invariant*, stating that whenever $\phi$ holds, $\psi$ must hold also, is given by the following formula: $G(\phi \rightarrow \psi)$. Where a system terminates, *partial correctness* may be specified in terms of a precondition $\phi$, which must hold initially, a postcondition $\psi$, which must hold on termination, and a condition $\varphi$, which indicates when termination has been reached: $\phi \rightarrow G(\varphi \rightarrow \psi)$. A *liveness* property is one that states that "something good will eventually happen". The simplest liveness properties have the form $F\phi$, stating that eventually, $\phi$ will hold. *Termination* is an example of liveness. The basic termination property is: $\phi \rightarrow F\varphi$ which states that every run which initially satisfied the property $\phi$ eventually satisfied the property $\varphi$, where $\varphi$ is the property which holds when a run has terminated. Another useful liveness property is *temporal implication*: $G(\phi \rightarrow F\psi)$ which states that "every $\phi$ is followed by a $\psi$". *Responsiveness* is a classic example of temporal implication: suppose $\phi$ represents a "request", and $\psi$ a "response". The above temporal implication would then state that every request is followed by a response.

[2]The material in this section was adapted from [12, pp.1049–1054].

### 4.2.2 Technical Results Relating to LTL

A great many technical results have been obtained with respect to LTL. A complete axiomatization was given in [24], and several different axiomatizations have subsequently been presented – we present one such axiomatization in Table 1. The axiom (LAX1) expresses the fact that LTL subsumes classical propositional logic, and so all tautologies of propositional logic can be taken as validities of LTL. Axiom (LAX2) can be understood as expressing the fact that the flows of time underpinning LTL are indeed linear: it says that if it is not the case that $\phi$ is true in the next state, then in the next state $\phi$ is false. (To see that this is only valid in linear models of time, consider the possibility that there are multiple next states, in some of which $\phi$ is true and in some of which $\phi$ is false.) Axioms (LAX3) and (LAX4) show that $X$ and $G$ correspond to the universal quantifier of first-order logic: if $\forall x(\phi \rightarrow \psi)$ and $\forall x\phi$ are true, then $\forall x\psi$ is true. This principle is also known from modal logic where (LAX3) and (LAX4) are called the K axioms for the "next" and "always" operators, respectively. Axiom (LAX5) is what is known as a *fixpoint* axiom. Notice that the converse implication is also easily seen to be true, and so we have as an axiom $G\phi \leftrightarrow (\phi \wedge XG\phi)$. This tells us that $G\phi$ is a maximum fixed point of $\phi \wedge XG\phi$. Axiom (LAX6) is an *induction* axiom: to show that $\phi$ is true forever, it is sufficient to show that $\phi$ is true in the present state and it is always the case that if $\phi$ is true now then $\phi$ is true in the next state. Axiom (LAX7) gives us a relationship between the $U$ and $F$ operators: if $\phi U\psi$ is true now, then this implies that $\psi$ is eventually true. Finally, axiom (LAX8) is a fixpoint axiom for the $U$ operator: it says that $\phi U\psi$ is a least fixpoint of $\psi \vee (\phi \wedge X(\phi U\psi))$. With respect to the inference rules, (LIR1) tells us that the modus ponens, the most basic inference rule of classical logic, holds in LTL. Rule (LIR2) is a version of the modal logic necessitation rule and states that if $\phi$ is a LTL tautology then $G\phi$ is a LTL tautology.

Tableau-based proof methods for LTL were introduced by Wolper [60], and resolution proof methods were developed by Fisher [18]. The computational complexity of satisfiability checking for LTL was investigated by Sistla and Clarke, who showed that the problem is PSPACE-complete [53]. This latter result is particularly interesting in that PSPACE-completeness is the complexity of the modal logic K [4, p.387], which would at first sight appear to be less expressive power than LTL.

### 4.2.3 LTL, Automata, and Model Checking

One interesting aspect of temporal languages over the natural numbers (and, in particular, LTL) which has turned out to be of great practical and theoretical value in computer science, is the relationship to *automata for infinite words* [42]. Of particular interest in temporal logic are a class of automata known as *Büchi automata*. Büchi automata are those that can recognise *$\omega$-regular expressions*: regular expressions that may contain infinite repetition. A fundamental result in temporal logic theory is that for every LTL formula $\phi$ one can construct a Büchi automaton $\mathcal{A}_\phi$ that accepts ex-

actly the models of $\phi$ (more precisely, the infinite words corresponding to models of $\phi$). The technique for constructing $\mathcal{A}_\phi$ from $\phi$ is closely related to Wolper's tableau proof method for temporal logic [60]. This result yields a decision procedure for satisfability of LTL formulae: to determine whether a formula $\phi$ is satisfiable, construct the automaton $\mathcal{A}_\phi$ and check whether this automaton accepts at least one word (the latter problem is well-understood and can be solved in polynomial time). In addition, however, this connection between Büchi automata and LTL also provides a route to automating the verification of computer programs, in an approach known as *automata theoretic model checking* [11]. The basic idea is the following. First, recall that we described above how the verification of computer systems can be treated as a proof problem in temporal logic: given a program $\mathcal{P}$ and a temporal logic specification $\psi$ of the desired behaviour of $\mathcal{P}$, derive the temporal theory $Th(\mathcal{P})$ of $\mathcal{P}$, and attempt to show that $Th(\mathcal{P}) \vdash \psi$. To make this approach work, we have two problems that we need to solve. The first is the process of deriving the temporal theory $Th(\mathcal{P})$ of the program $\mathcal{P}$; this is an area known as the *temporal semantics of programming languages*, and is itself a deeply technical area (see [44, 37, 38]). The second issue is that of actually finding a temporal proof. Here, the computational complexity of temporal logic is a problem: for example, temporal logics with quantification, of a form suitable for use in the verification of real programs, are highly undecidable [1]. This greatly limits the applicability of this approach for directly verifying systems, and in particular, has made it difficult to automate temporal verification using this approach. With model checking, we use an alternative approach to verification. The general idea is that since the state transition graph of a program $\mathcal{P}$ can be understood as a model $M_\mathcal{P}$ for a temporal logic, verifying that $\mathcal{P}$ satisfies $\psi$ can be done by checking that $M_\mathcal{P} \models \psi$. In general, the problem of checking whether or not $M_\mathcal{P} \models \psi$ is much easier to automate than deductive proof. The link between Büchi automata and LTL provides one such method for automation.

First, a little notation: where $\mathcal{A}$ is a (Büchi) automaton, then let $\mathcal{L}_{\mathcal{A}_\psi}$ denote the language accepted by $\mathcal{A}$; thus $\mathcal{L}_{\mathcal{A}_\psi}$ will be a set of infinite words over the alphabet of $\mathcal{A}$. Now, the first step to automata-theoretic model checking is to model $\mathcal{P}$ as a Büchi automaton $\mathcal{A}_\mathcal{P}$; the language $\mathcal{L}_{\mathcal{A}_\mathcal{P}}$ recognised by $\mathcal{A}_\mathcal{P}$ will represent the set of possible computations of $\mathcal{P}$. Given the specification $\psi$ as a LTL formula, one constructs a Büchi automaton $\mathcal{A}_\psi$ such that $\mathcal{L}_{\mathcal{A}_\psi}$ is exactly the set of models of $\psi$. It should then be clear that $\mathcal{P}$ satisfies $\psi$ iff

$$\mathcal{L}_{\mathcal{A}_\mathcal{P}} \subseteq \mathcal{L}_{\mathcal{A}_\psi} \tag{1}$$

That is, $\mathcal{P}$ satisfies $\psi$ if the set of possible computations of $\mathcal{P}$ are a subset of the computations that satisfy $\psi$. We can rewrite (1) as

$$\mathcal{L}_{\mathcal{A}_\mathcal{P}} \cap \mathcal{L}_{\mathcal{A}_{\neg\psi}} = \emptyset \tag{2}$$

That is, $\mathcal{P}$ satisfies $\psi$ if the intersection of the set of computations of $\mathcal{P}$ and the set of

computations *disallowed* by $\psi$ is empty. We can further rewrite (2) as:

$$\mathcal{L}_{\mathcal{A}_{\mathcal{P}} \cap \mathcal{A}_{\neg\psi}} = \emptyset \tag{3}$$

So, checking that $\mathcal{P}$ satisfies $\psi$ involves first generating $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\neg\psi}$, then checking whether $\mathcal{L}_{\mathcal{A}_{\mathcal{P}} \cap \mathcal{A}_{\neg\psi}} = \emptyset$. This approach has been implemented in a practical model checking tool for LTL called SPIN, which has been widely used in the formal analysis of computer protocols and programs [30, 31]. We refer the reader to [42] for an overview of automata-based techniques for temporal reasoning.

### 4.2.4 Synthesis and Direct Execution

A different approach to the issue of program correctness is provided by the paradigm of *automated synthesis from temporal specifications*. Here, the idea is that, rather than generating a program and then attempting to check that the program satisfies the specification, we start with a specification $\Psi$ and *automatically generate* a program $\mathcal{P}_{\Psi}$ from $\Psi$ such that $\mathcal{P}_{\Psi} \vdash \Psi$. Synthesis from temporal specifications is discussed in [39, 46, 45, 10]. Although this approach seems intuitively very appealing, synthesis is a computationally rather complex process, which has to date hampered attempts to automate the process.

Another approach that attempts to avoid working with programs $\mathcal{P}$ is the paradigm of *direct execution*. Here, a temporal specification $\Psi$ is viewed as a program, which is directly executed by an interpreter. What does it mean, to execute a formula $\Psi$? It means generating a model $M$ such that $M \models \Psi$ [20]. If this could be done without interference from the environment — if the program had complete control over its environment — then execution would reduce to constructive theorem proving, where we show that $\Psi$ is satisfiable by building a model for $\Psi$. In reality of course, programs are *not* interference-free: they must iteratively construct a model in the presence of input from the environment. Execution can then be seen as a two-way iterative process:

- environment makes something true;
- program responds by doing something, i.e., making something else true in the model;
- environment responds, making something else true;
- ...

A useful way to think about execution is thus as if the program is *playing a game* against the environment. The specification $\Psi$ represents the goal of the game: the program must keep the goal satisfied, while the environment tries to prevent the program doing so. The game is played by program and environment taking it in turns to build a little more of the model. If the specification ever becomes false in the (partial)

model, then the program loses. A *winning strategy* for building models from (satisfiable) specifications $\Psi$ in the presence of arbitrary input from the environment is an execution algorithm.

In the context of LTL, the METATEM framework represents the best-known attempt to turn this vision of directly executing temporal formula into reality [19]. In METATEM, the idea is to require specifications $\Psi$ to take the form:

$$\Psi = \bigwedge_i past_i \rightarrow future_i$$

where $past_i$ is a temporal formula that refers to the past, and $future_i$ is a temporal formula that refers to the future. The $past_i \rightarrow future_i$ formulae are called *rules*. Executing such a set of rules is very intuitive: whenever we see that the past looks like $past_i$, then we must commit to making the future look like $future_i$.

## 4.3 Branching Temporal Logic – CTL

If we are interested in the branching structure of reactive programs and their possible computations, then LTL does not seem a very appropriate language. With linear time, there is just one path, so an event either happens or it doesn't happen. But for a reactive program there may be *multiple* possible computations, and an event may occur on some of these, but not on others. How to capture this type of situation? The basic insight in CTL is to talk about possible computations (or futures) by introducing two operators "A" ("on all paths ..."") and "E" ("on some path ..."), called *path quantifiers*, which can be prefixed to a temporal (LTL) formula. For example, the CTL formula $\mathsf{A}Fp$ says "on all possible futures, $p$ will eventually occur", while the CTL formula $\mathsf{E}Fq$ says "there is at least one possible future on which $q$ eventually occurs". CTL imposes one important syntactic restriction on the structure of formulae: a temporal (LTL) operator must be prefixed by a path quantifier. The language without this restriction is known as CTL* [14]: it is much more expressive than CTL, but also much more complex. For simplicity, we will here stick with CTL.

Starting from a set $\Phi$ of propositional atoms, the syntax of CTL is defined by the following grammar:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{E}X\phi \mid \mathsf{E}(\phi U \phi) \mid \mathsf{A}X\phi \mid \mathsf{A}(\phi U \phi)$$

where $p \in \Phi$. Given these operators, we can derive the remaining CTL temporal operators as follows:

$$\begin{array}{lcllcl} \mathsf{A}F\phi & \equiv & \mathsf{A}(\top U \phi) & \quad & \mathsf{E}F\phi & \equiv & \mathsf{E}(\top U \phi) \\ \mathsf{A}G\phi & \equiv & \neg\mathsf{E}F\neg\phi & & \mathsf{E}G\phi & \equiv & \neg\mathsf{A}F\neg\phi \end{array}$$

As in the case of LTL, the semantics of CTL is defined with respect to transition systems. For a state $s$ in a transition system $M = (S, R, V)$ we say that a path $\rho$ is a *s-path* if $\rho[0] = s$. Let $paths(s)$ denote the set of $s$-paths over $M$.

The satisfaction relation "$M, s \models \phi$" between between pairs $M, s$ and CTL formulae $\phi$ formalizes the condition "at state $s$ in the transition system $M$ the formula $\phi$ holds" and is inductively defined via the following rules:

$M, s \models \top$

$M, s \models p$ iff $p \in V(s)$ (where $p \in \Phi$);

$M, s \models \neg\phi$ iff $M, s \not\models \phi$

$M, s \models \phi \vee \psi$ iff $M, s \models \phi$ or $M, s \models \psi$

$M, s \models \mathsf{A}X\phi$ iff $\forall \rho \in paths(s) : M, \rho[1] \models \phi$

$M, s \models \mathsf{E}X\phi$ iff $\exists \rho \in paths(s) : M, \rho[1] \models \phi$

$M, s \models \mathsf{A}(\phi U \psi)$ iff $\forall \rho \in paths(s), \exists u \in \mathbb{N}$, s.t. $M, \rho[u] \models \psi$ and $\forall v, (0 \leq v < u) : M, \rho[v] \models \phi$

$M, s \models \mathsf{E}(\phi U \psi)$ iff $\exists \rho \in paths(s), \exists u \in \mathbb{N}$, s.t. $M, \rho[u] \models \psi$ and $\forall v, (0 \leq v < u) : M, \rho[v] \models \phi$

Referring back to the branching time model given in Figure 2(c), we leave the reader to verify that in the initial state, the following formulae are satisfied:

- $\mathsf{E}Xx$
- $\neg\mathsf{A}Xx$
- $\mathsf{A}Fy$
- $\mathsf{E}(xUy)$

At this point, it is a useful exercise to convince oneself that one cannot interpret in any meaningful way LTL formulae in pairs $M, s$ consisting of a model and a state: a run is required to interpret pure temporal formulae without prefixed path quantifiers. Conversely, one cannot interpret CTL formulae in pairs $\rho, u$ consisting of a run and a number: the whole transition system is required to interpret path quantifiers.

As with LTL, many technical results have been obtained with respect to CTL. A complete axiomatization is given in Table 2 (see [13]; for discussion and further references, see [12, p.1040]). Satisfiability of CTL formulas is harder than that of LTL: the problem is EXPTIME-complete [12, p.1037]. As with linear time temporal logic, the relationship to automata is fundamental for understanding the behaviour of CTL and other branching time logics. Here one employs automata for *infinite trees* rather than infinite words. We refer the reader to [42] for an overview.

Model checking techniques have also been developed for CTL, with great success. However, the main approaches used in CTL model checking are not based on

| | |
|---|---|
| Axioms: | |
| $(BAX1)$ | propositional tautologies |
| $(BAX2)$ | $\mathsf{E}F\phi \leftrightarrow \mathsf{E}(\top U \phi)$ |
| $(BAX2b)$ | $\mathsf{A}G\phi \leftrightarrow \neg\mathsf{E}F\neg\phi$ |
| $(BAX3)$ | $\mathsf{A}F\phi \leftrightarrow \mathsf{A}(\top U \phi)$ |
| $(BAX3b)$ | $\mathsf{E}G\phi \leftrightarrow \neg\mathsf{A}F\neg\phi$ |
| $(BAX4)$ | $\mathsf{E}X(\phi \vee \psi) \leftrightarrow (\mathsf{E}X\phi \vee \mathsf{E}X\psi)$ |
| $(BAX5)$ | $\mathsf{A}X\phi \leftrightarrow \neg\mathsf{E}X\neg\phi$ |
| $(BAX6)$ | $\mathsf{E}(\phi U \psi) \leftrightarrow (\psi \vee (\phi \wedge \mathsf{E}X\mathsf{E}(\phi U \psi)))$ |
| $(BAX7)$ | $\mathsf{A}(\phi U \psi) \leftrightarrow (\psi \vee (\phi \wedge \mathsf{A}X\mathsf{A}(\phi U \psi)))$ |
| $(BAX8)$ | $\mathsf{E}X\top \wedge \mathsf{A}X\top$ |
| $(BAX9)$ | $\mathsf{A}G(\phi \rightarrow (\neg\psi \wedge \mathsf{E}X\phi)) \rightarrow (\phi \rightarrow \neg\mathsf{A}(\gamma U \psi))$ |
| $(BAX9b)$ | $\mathsf{A}G(\phi \rightarrow (\neg\psi \wedge \mathsf{E}X\phi)) \rightarrow (\phi \rightarrow \neg\mathsf{A}F\psi)$ |
| $(BAX10)$ | $\mathsf{A}G(\phi \rightarrow (\neg\psi \wedge (\gamma \rightarrow \mathsf{A}X\phi))) \rightarrow (\phi \rightarrow \neg\mathsf{E}(\gamma U \psi))$ |
| $(BAX10b)$ | $\mathsf{A}G(\phi \rightarrow (\neg\psi \wedge \mathsf{A}X\phi)) \rightarrow (\phi \rightarrow \neg\mathsf{E}F\psi)$ |
| $(BAX11)$ | $\mathsf{A}G(\phi \rightarrow \psi) \rightarrow (\mathsf{E}X\phi \rightarrow \mathsf{E}X\psi)$ |
| | |
| Inference Rules: | |
| $(BIR1)$ | From $\vdash \phi$ and $\vdash \phi \rightarrow \psi$ infer $\vdash \psi$ |
| $(BIR2)$ | From $\vdash \phi$ infer $\vdash \mathsf{A}G\phi$ |

Table 2: A complete axiomatization for CTL.

automata theory, but are known as *symbolic* model checking. The basic idea is as follows. A fundamental problem in model checking is known as the *state explosion problem*. Suppose the computer program we want to analyse has $n$ bits of memory. In a typical cheap laptop available as we write (early 2011), $n$ might be $10^9$, for example. Then the number of possible states of the program might be as high as $2^n$. Now, $2^{10^9}$ is a number so large as to be very hard to contemplate: certainly we could never hope to manipulate such a large set of states within a computer. For a long time, this *state explosion problem* hampered attempts to develop usable model checking tools. In the early 1990s, however, a researcher at Carnegie Mellon University called Ken McMillan realised that it was possible to represent and manipulate such large state structures very efficiently, by using a data structure known as a *binary decision diagram* (BDD). McMillan implemented the SMV model checking system for CTL, using BDDs as the data structure for representing the system state space and transition relation [32]. SMV proved enormously influential, and has been very widely used. BDD-based model checking techniques are a highly active area of research today.

# 5 Interval Temporal Logics

In all temporal logics we considered so far, temporal formulae were evaluated at *time points* or *states*. An alternative, and typically more powerful, way of evaluating formulae is in *intervals*, sets $i$ of time points with the property that if $t_1 \leq t_2 \leq t_3$ and $t_1, t_3 \in i$, then $t_2 \in i$. For example, the sentence "Mary often visits her mother" can be true in a certain interval, say from 2006 to 2008, but it does not make sense to say that it is true at a certain time point or state. Many different interval based temporal logics have been introduced [27, 54, 41]. When designing such a language, the first decision to take is the set of temporal operators. Between time points, there are only three distinct qualitative relations: before, after, and equal. This might be the reason that point-based temporal logics typically employ (some subset) of the rather small set of operators discussed above (Kamp's Theorem provides another explanation). In contrast, there are thirteen distinct qualitative relations between time intervals, known as Allen's relations [2] and depicted in Figure 3. Assuming that intervals are closed (i.e., contain their endpoints), we give formal definitions for two such relations: for intervals $i$ and $j$, we say that $i$ is before $j$, in symbols $\mathsf{before}(i, j)$, if for all $t \in i$ and $t' \in j$ we have $t < t'$ and we say that $i$ meets $j$, in symbols $\mathsf{meets}(i, j)$, if $i \cap j$ consists of exactly one element and for all $t \in i$ and $t' \in j$ we have $t \leq t'$. The possible choices of temporal operators for interval-based logics reflect these relations. For example, for the relation "before" one can introduce a temporal operator $<\mathsf{before}>$ whose truth condition is as follows:

> $<\mathsf{before}>\ \phi$ is true in interval $i$ if, and only if, there exists an interval $j$ such that $i$ is before $j$ and $\phi$ is true in $j$.

Operators $<\mathsf{meets}>$, $<\mathsf{during}>$, etc. can be introduced in the same way. The resulting temporal language with operators for all 13 Allen relations (or some subset thereof of equal expressivity) has been investigated extensively [27, 54]. In contrast to point-based temporal logics, in interval temporal logic it is typically undecidable whether a formula is a temporal tautology. Often (e.g., for the discrete time flow consisting of a copy of the natural numbers and for the time flow consisting of the reals) temporal tautologies are even not recursively enumerable and, therefore, non-axiomatizable [27]. Such a negative result can give rise to an interesting new research program: to classify the fragments of the undecidable/non-axiomatizable logic into those that are decidable/axiomatizable and those that are still undecidable/non-axiomatizable. The paradigmatic example of such a program is the undecidability of classical predicate logic that has transformed Hibert's original Entscheidungsproblem into a classification problem asking which fragments of classical predicate logic are decidable [6]. For interval temporal logics a similar (but smaller scale) program has been launched. The recent state of the art for the classification problem for interval temporal logics is described in [8].

From a philosophical as well as mathematical perspective it is also of interest to regard intervals not as derived objects from time points but as primitive objects [54].

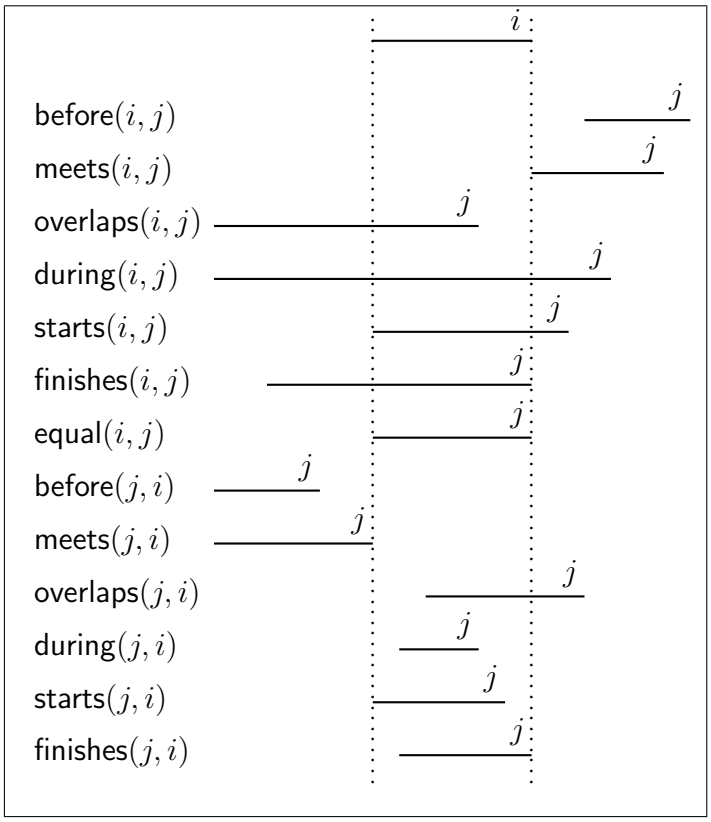


Figure 3: Allen's relations between intervals.

The models in which temporal formulae are interpreted are then not collections of time points, but collections of intervals with temporal relations between them. Typical temporal relations one can consider are (subsets of) the set of thirteen Allen relation, however, now one has to explicitly axiomatize their properties rather than derive them from the underlying point based structure. This then opens the door for representation theorems: when is an abstract structure of primitive intervals representable as a concrete structure of intervals induced by time points? Can one describe those structures axiomatically? We refer the reader to [54, 55] for a discussion of this approach and results.

# 6 Dynamic Logic

At about the same time that Pnueli was first investigating temporal logic, a different class of logics, also based on modal logic, were being developed for reasoning about actions in general, and computer programs in particular. The starting point for this research is the following observation. Temporal logics allow us to describe the time-varying properties of dynamic domains, but they have nothing to say about the *actions* that *cause* these changes; that is, in the language, we have no direct way of expressing

things like "Michael turned the motor on". Here "turning the motor on" is an action, and the performance of this action changes the state of the world. There are many situations, however, where it is desirable to be able to explicitly refer to actions and the effects that they have. One such domain that is particularly well-suited for formal representation and reasoning is computer programs. A computer program can be regarded as a list of actions which the computer must execute one after the other. Note that in contrast to many other domains, there is little or no ambiguity about what the actions are; the computer programming language makes the meaning and effect of such actions precise, and this enables us to develop and use formalisms for reasoning about them. Dynamic logics arose from the desire to establish the correctness of computer programs using a logic that explicitly refers to the actions the computer is executing.

An important question to ask about terminating programs is what properties they guarantee, i.e., what properties are guaranteed to hold after they have finished executing. This type of reasoning can be captured using modal operators: we might interpret the formula $[\mathcal{P}]\phi$ to mean that "after all possible terminating runs of the program $\mathcal{P}$, the formula $\phi$ holds". Given a conventional Kripke semantics, possible worlds are naturally interpreted as the states of a machine executing a program. However, this modal treatment of programs has one key limitation. It treats programs as *atomic*, whereas in reality, programs are highly structured, and this structure is central to understanding their behaviour. So, rather than using a *single* modal "box" operator, the idea in dynamic logic is to use a collection of operators $[\pi]$, one for each program $\pi$, where $[\pi]\phi$ then means "on all terminating executions of program $\pi$, the property $\phi$ holds". Crucially, $\pi$ is allowed to contain *program constructs* such as selection ("if") statements, loops, and the like; the overall behaviour of programs is derived from their component programs. The resulting formalism is known as *dynamic logic*; it was originally formulated by Vaughan Pratt in the late 1970s. Here, we will introduce the best-known variant of dynamic logic, known as *Propositional Dynamic Logic* (PDL), introduced by Fischer and Ladner [17].

Formally, we define the syntax of programs $\pi$ and formulae $\phi$ with respect to a set $A$ of atomic actions and a set $\Phi$ of propositional atoms by mutual induction through the following grammar:

$$\begin{array}{lcl} \pi & ::= & \alpha \mid \pi;\pi \mid \pi^* \mid \pi \cup \pi \mid \phi? \\ \phi & ::= & p \mid \neg\phi \mid \phi \vee \phi \mid [\pi]\phi \end{array}$$

where $\alpha \in A$ and $p \in \Phi$.

The program constructs ";", "$\cup$", "$\cdot^*$", and "?" are known as *sequence*, *choice*, *iteration*, and *test*, and closely reflect the basic constructs found in programming languages:

$\pi_1;\pi_2$ means "execute program $\pi_1$ and then execute program $\pi_2$";

$\pi_1 \cup \pi_2$ means "either execute program $\pi_1$ or execute program $\pi_2$";

$\pi^*$ means "repeatedly execute $\pi$ (an undetermined number of times)"; and

$\phi?$ means "only proceed if $\phi$ is true."

Let us see a few examples of PDL formulae, and the properties they capture.

$$p \rightarrow [\pi]q$$

This asserts that if $p$ is true, then after we have executed program $\pi$, we are guaranteed to have $q$ true.

$$p \rightarrow [\pi_1 \cup \pi_2]q$$

This asserts that if $p$ is true, then no matter whether we execute program $\pi_1$ or program $\pi_2$, $\phi$ will be true.

The program constructs provided in PDL may seem rather strange to those familiar with programming languages such as C, PASCAL, or JAVA. In particular, we do not seem to have in PDL operators for `if...else` constructs, or the familiar loop constructs such as `while` and `repeat`. However, this is not the case: they can be defined in terms of PDL constructs, as follows. First, consider the following definition of an `if...else` construction:

$$\texttt{if}\ \phi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2 \quad = \quad ((\phi?;\pi_1) \cup (\neg\phi?;\pi_2))$$

A `while` construct can be defined:

$$\texttt{while}\ \phi\ \texttt{do}\ \pi \quad = \quad ((\phi?;\pi)*;\neg\phi?)$$

A `repeat` construct can be defined:

$$\begin{array}{rcl} \texttt{repeat}\ \pi\ \texttt{until}\ \phi & = & \pi;\texttt{while}\ \neg\phi\ \texttt{do}\ \pi \\ & = & \pi;((\neg\phi?;\pi)*;\phi?) \end{array}$$

We invite the reader to convince themselves that these definition do indeed capture the meaning of these operators in C/JAVA-like languages.

The semantics of PDL are somewhat more involved than the logics we have looked at previously. It is based on the semantics of normal modal logic; we have a set $S$ of states, and for each atomic action $\alpha$ we have a relation $R_\alpha \subseteq S \times S$, defining the behaviour of $\alpha$, with the idea being that $(s, s') \in R_\alpha$ if state $s'$ is one of the possible outcomes that could result from performing action $\alpha$ in state $s$. Given these atomic relations, we can then obtain accessibility relations for arbitrary programs $\pi$, as follows. Let the composition of relations $R_1$ and $R_2$ be denoted by $R_1 \circ R_2$, and the reflexive transitive closure (ancestral) of relation $R$ by $R^*$. Then the accessibility relations for complex programs are defined [28]:

$$\begin{array}{rcl} R_{\pi_1;\pi_2} & = & R_{\pi_1} \circ R_{\pi_2} \\ R_{\pi_1 \cup \pi_2} & = & R_{\pi_1} \cup R_{\pi_2} \\ R_{\pi^*} & = & (R_\pi)^* \\ R_{\phi?} & = & \{(s,s) \mid M, s \models \phi\}. \end{array}$$

Notice that the final clause refers to a satisfaction relation for PDL, $\models$, which has not yet been defined. So let us define this relation. A model for PDL is a structure:

$$M = \langle S, \{R_\alpha\}_{\alpha \in A}, V \rangle$$

where:

- $S$ is a set of states;
- $\{R_\alpha\}_{\alpha \in A}$ is a collection of accessibility relations, one for each atomic program $\alpha \in A$; and
- $V : S \to 2^\Phi$ gives the set of propositional atoms true in each state.

Given these definitions, the satisfaction relation $\models$ for PDL holds between pairs $M, s$ and formulae:

$M, s \models p$ iff $p \in V(s)$ (where $p \in \Phi$);

$M, s \models \neg\phi$ iff not $M, s \models \phi$

$M, s \models \phi \vee \psi$ iff $M, s \models \phi$ or $M, s \models \psi$

$M, s \models [\pi]$ iff $\forall s' \in S$ such that $(s, s') \in R_\pi$ we have $M, s' \models \phi$

A complete axiomatization of PDL was first given by Segerberg [51] – see Table 3. Many axioms are similar to those used in the axiomatization of LTL. For example, (PAX8) is an induction axiom: to show that $\phi$ holds in all states that can be reached by executing $\pi$ finitely many times, it is sufficient to show that $\phi$ holds in the current state and that if $\phi$ holds in a state reachable by executing $\pi$ finitely many times, then $\phi$ holds in all states that can reached from that state by executing $\pi$ again.

The satisfiability problem for PDL is EXPTIME-complete [16]. Thus PDL has the same computational complexity as CTL. Numerous extensions of PDL with various additional program constructors such as loop, intersection, and converse have been considered. For an overview, we refer the reader to [28]. There also exist first-order versions of dynamic logics in which the abstract atomic actions of PDL are replaced by "real" atomic programs such as, for example, $x := x + 4$ [28].

PDL has been used to represent and reason about structured actions in more general domains than computer programs. One particularly interesting application is the formal study of normative concepts and sentences such as "it is permitted to do $\pi$" and "it is forbidden to do $\pi$". To formalise such sentences in PDL, it has been suggested to introduce a propositional constant $\mathsf{Good}$ that denotes all states that are regarded as desirable. Using this constant, one can then formalise "it is permitted to do $\pi$" using the PDL formula $[\pi]\mathsf{Good}$ and "it is forbidden to do $\pi$" using $[\pi]\neg\mathsf{Good}$. For a discussion of this and other applications of PDL that are not restricted to computer programs we refer the reader to [40].

| | |
|---|---|
| Axioms: | |
| $(PAX1)$ | propositional tautologies |
| $(PAX2)$ | $[\pi](\phi \rightarrow \psi) \rightarrow ([\pi]\phi \rightarrow [\pi]\psi)$ |
| $(PAX3)$ | $[\pi](\phi \wedge \psi) \leftrightarrow [\pi]\phi \wedge [\pi]\psi$ |
| $(PAX4)$ | $[\pi_1 \cup \pi_2]\phi \leftrightarrow [\pi_1]\phi \wedge [\pi_2]\phi$ |
| $(PAX5)$ | $[\pi_1; \pi_2]\phi \leftrightarrow [\pi_1][\pi_2]\phi$ |
| $(PAX6)$ | $[\psi?]\phi \leftrightarrow (\psi \rightarrow \phi)$ |
| $(PAX7)$ | $\phi \wedge [\pi][\pi*]\phi \leftrightarrow [\pi^*]\phi$ |
| $(PAX8)$ | $\phi \wedge [\pi^*](\phi \rightarrow [\pi]\phi) \rightarrow [\pi^*]\phi$ |
| | |
| Inference Rules: | |
| $(PIR1)$ | From $\vdash \phi \rightarrow \psi$ and $\vdash \phi$ infer $\vdash \psi$ |
| $(PIR2)$ | From $\vdash \phi$ infer $\vdash [\pi]\phi$ |

Table 3: A complete axiomatization for PDL.

# 7 Further Reading

We emphasise that temporal and dynamic logics are major research areas, with a vast literature behind them. In this short paper, we have been able to do no more than sketch some of the major directions and developments. For more reading, we recommend [58] as a gentle and short introduction to temporal logic, [26] as a mathematical introduction to temporal and dynamic logic, with particular emphasis on the use of such logics for reasoning about programs, and [12] for an excellent technical introduction to LTL and CTL. A recent collection of papers on temporal reasoning in AI is [22]; a comprehensive overview article, providing many pointers to further reading on temporal logic may be found in [21]. The debate on the relative merits of linear versus branching time logics to a certain extent continues today; see, e.g., [56] for a relatively recent contribution to the debate, with extensive references. The definitive reference to dynamic logic is [28].

# References


[1] M. Abadi. *Temporal Logic Theorem Proving*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA 94305, 1987.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Proceedings of the Eighth ACM Symposium on the Principles of Programming Languages (POPL)*, pages 164–176, 1981.

[4] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press: Cambridge, England, 2001.

[5] P. Blackburn, J. van Benthem, and F. Wolter. Preface. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*. Elsevier Science, 2006.

[6] E. Boerger, E. Graedel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[7] R. S. Boyer and J. S. Moore, editors. *The Correctness Problem in Computer Science*. The Academic Press: London, England, 1981.

[8] D. Bresolin, D. D. Monica, V. Goranko, A. Montanari, and G. Sciavicco. Decidable and undecidable fragments of halpern and shoham's interval temporal logic: Towards a complete classification. In *LPAR*, pages 590–604, 2008.

[9] J. Burgess. Basic tense logics. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical logic*. Reidel, 1984.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs — Proceedings 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Berlin, Germany, 1981.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[12] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[13] 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.

[14] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

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

[16] M. Fischer and N. J. Pippenger. Relations among complexity measures. *Journal of the ACM*, 26:361–381, 1979.

[17] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[18] M. Fisher. A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, Aug. 1991.

[19] M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Berlin, Germany, July 1994.

[20] M. Fisher. An introduction to executable temporal logic. *The Knowledge Engineering Review*, 11(1):43–56, 1996.

[21] M. Fisher. Temporal representation and reasoning. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 2008.

[22] M. Fisher, D. Gabbay, and L. Vila, editors. *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 2005.

[23] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*. Clarendon Press, 1994.

[24] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages (POPL '80)*, pages 163–173, New York, USA, Jan. 1980. ACM Press.

[25] A. Galton. Temporal logic and computer science: An overview. In A. Galton, editor, *Temporal Logics and their Applications*, pages 1–52. The Academic Press: London, England, 1987.

[26] R. Goldblatt. *Logics of Time and Computation (CSLI Lecture Notes Number 7)*. Center for the Study of Language and Information, Ventura Hall, Stanford, CA 94305, 1987. (Distributed by Chicago University Press).

[27] J. Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. *J. ACM*, 38(4):935–962, 1991.

[28] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press: Cambridge, MA, 2000.

[29] I. Hodkinson and M. Reynolds. Temporal logic. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*. Elsevier Science, 2006.

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

[31] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley: Reading, MA, 2003.

[32] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990.

[33] H. Kamp. *Tense logic and the theory of linear order*. PhD thesis, University of California, 1968.

[34] L. Lamport. Sometimes is sometimes not never — but not always. In *Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages (POPL)*, 1980.

[35] T. Litak and F. Wolter. All finitely axiomatizable tense logics of linear time flows are conp-complete. *Studia Logica*, 81(2):153–165, 2005.

[36] C. Lutz, U. Sattler, and F. Wolter. Modal logic and the two-variable fragment. In *CSL*, pages 247–261, 2001.

[37] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.

[38] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.

[39] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, Jan. 1984.

[40] J.-J. Meyer and F. Veltman. Intelligent agents and common sense reasoning. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*. Elsevier Science, 2006.

[41] B. C. Moszkowski. A complete axiomatization of interval temporal logic with infinite time. In *LICS*, pages 241–252, 2000.

[42] M.Y.Vardi. Automata-theoretic techniques for temporal reasoning. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*. Elsevier Science, 2006.

[43] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth IEEE Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[44] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:1–20, 1981.

[45] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on the Principles of Programming Languages (POPL)*, pages 179–190, Jan. 1989.

[46] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programs*, 1989.

[47] A. Prior. *Time and Modality*. Oxford University Press, 1957.

[48] A. Prior. *Past, Present, and Future*. Oxford University Press, 1967.

[49] A. Prior. A statement of temporal realism. In B.J.Copeland, editor, *Logic and Reality: Essays on the Legacy of Arthur Prior*. Clarendon Press, 1996.

[50] H. Reichgelt. A comparison of first-order and modal logics of time. In P. Jackson, H. Reichgelt, and F. van Harmelen, editors, *Logic Based Knowledge Representation*, pages 143–176. The MIT Press: Cambridge, MA, 1989.

[51] K. Segerberg. A completeness theorem in the modal logic of programs. *Not. Amer. Math. Soc.*, 24(6), 1977.

[52] A. Sistla. Theoretical issues in the design and verification of distributed systems. Technical Report CMU-CS-83-146, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1983.

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

[54] J. van Benthem. *The Logic of Time*. Kluwer, 1983.

[55] J. van Benthem. Temporal logic. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 241–351. Oxford University Press, 1995.

[56] M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001 (LNCS Volume 2031)*, pages 1–22. Springer-Verlag: Berlin, Germany, Apr. 2001.

[57] M. Y. Vardi. From philosophical to industrial logics. In *ICLA*, pages 89–115, 2009.

[58] Y. Venema. Temporal logic. In L. Goble, editor, *The Blackwell Guide to Philosophical Logic*, pages 203–223. Blackwell Publishers, 2001.

[59] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56, 1983.

[60] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 110–111, 1985.