Logic and Theoretical Computer Science

Bernd-Holger Schlingloff, Univ. Bremen

November 15, 1999

1 Limits of computability

Towards the end of the last century, A. Whitehead and B. Russell formalized the mathematics known at that time in their three-volume book *Principia Mathematica* (appeared 1910-13). In 1901, D. Hilbert proposed as a research topic for the century to find a calculus in which all true sentences of mathematics could be proved. Thirty years later, K. Gödel proved this to be impossible: there is no complete axiom system enumerating all true sentences of elementary arithmetik. In 1936, A. Church used this result to show that first order predicate calculus is undecidable. Independently, A. Turing proved that there is no algorithm to determine whether any given program will ever halt. Restrictions and extensions of his model of computation led to other important complexity classes.

1.1 Propositional and First Order Logic

For propositional logic **PL**, we assume that $\mathcal{P} = \{\mathtt{p}, \mathtt{q}, \mathtt{p}_1, ...\}$ is a set of proposition symbols which can be either true or false. *Propositional formulas* are built from $\mathcal{P}$ with the following syntax:

- Every $\mathtt{p} \in \mathcal{P}$ is a well-formed formula of propositional logic,
- $\bot$ is a well-formed formula ("the falsum"),
- if φ and ψ are well-formed formulae, then so is $(\varphi \to \psi)$, and
- nothing else is a propositional formula.

$\mathcal{P}$ is a parameter of the logic; the special case $\mathcal{P} = \emptyset$ is allowed. Other connectives can be defined as usual: $\neg\varphi \triangleq (\varphi \to \bot)$, $\top \triangleq \neg\bot$, $(\varphi \vee \psi) \triangleq (\neg\varphi \to \psi)$, $(\varphi \wedge \psi) \triangleq \neg(\neg\varphi \vee \neg\psi)$, and $(\varphi \leftrightarrow \psi) \triangleq ((\varphi \to \psi) \wedge (\psi \to \varphi))$. The precedence of these operators is fixed by $(\neg, \wedge, \vee, \to, \leftrightarrow)$, and parentheses are omitted in formulas whenever appropriate. A formula consisting of a proposition symbols or a negated proposition symbol is called a *literal*.

An *interpretation* $\mathcal{I}$ for the propositions is a function assigning a truth value from **true** or **false** to every proposition. A *propositional model* $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I})$ consists of the binary domain $\mathcal{U} \triangleq \{\mathbf{true}, \mathbf{false}\}$ and an interpretation for $\mathcal{P}$. The most basic semantical notion is the *validation relation* $\models$ between a model $\mathcal{M}$ and a formula φ. It is defined by the following clauses.

- $\mathcal{M} \models \mathtt{p}$ iff $\mathcal{I}(\mathtt{p}) = \mathbf{true}$,
- $\mathcal{M} \not\models \bot$, and
- $\mathcal{M} \models (\varphi \to \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$.

If $\mathcal{M} \models \varphi$, then we say that $\mathcal{M}$ *validates* φ or that φ is *valid in* $\mathcal{M}$.

Propositional logic is not well-suited to formalise statements involving all or some elements of a set. To express such quantitative dependencies, first order logic can be used.

A *first order signature* $\Sigma = (\mathcal{F}, \mathcal{R})$ consists of a finite set $\mathcal{F}$ of *function symbols*, and a finite set $\mathcal{R}$ of *relation symbols*. Associated with each function and relation symbol is its *arity* τ, which is a natural number ≥ 0. Propositions of propositional logic correspond to nullary relation symbols of first order logic. Unary relation symbols are called *predicate symbols*, and nullary function symbols are called *constant symbols*.

In first order logic, atomic formulas are $R(t_1 ... t_n)$, where R is an n-ary relation symbol and each t_i is an object term consisting of function symbols and individual variables. Formally, assume that $\mathcal{V} = \{x_1, x_2, ...\}$ is a countable set of *individual variables*. An *object term* t is

- x, where $x \in \mathcal{V}$ is an individual variable, or
- $f t_1 ... t_n$, where f is an n-ary function symbol and each t_i is an object term.

As a special case, each constant symbol is an object term. A *constant term* is an object term not containing any individual variables.

A *well formed formula* φ of first order logic is one of the following:

- $R(t_1 ... t_n)$, where R is an n-ary relation symbol, and each t_i is an object term,
- $t_1 = t_2$, where t_1 and t_2 are object terms, or
- $\bot$,
- $(\varphi \to \psi)$, where φ and ψ are well formed formulas,
- $\exists x\ \varphi$, where φ is a well formed formula, and x is an individual variable.

The formula $\forall x\ \varphi$ is an abbreviation for $\neg \exists x\ \neg \varphi$. When writing formulas, we often use infix notation for binary relations: $t_1 R t_2 \triangleq R(t_1, t_2)$. In the formula $\exists x\ \varphi$, each occurrence of the individual variable x in φ is said to be *bound*. Variables which are not bound by a quantification are *free*. *Sentences* or *closed formulas* are those not containing any free variables.

Assume that $\mathcal{U}$ is any nonempty set (the *universe of discourse*). An *interpretation* $\mathcal{I}$ for the signature Σ is a mapping $\mathcal{I} : \Sigma \to S$ assigning a function $\mathcal{I}(f) \in (\mathcal{I}(D_0) \times \cdots \times \mathcal{I}(D_{n-1}) \to \mathcal{I}(D_n))$ and relation $\mathcal{I}(R) \subseteq (\mathcal{I}(D_1) \times \cdots \times \mathcal{I}(D_n))$ of appropriate arity for each function and relation symbol, respectively. For nullary relations, we identify $\{\}$ (the empty set) with **false** and $\{()\}$ (the nonempty set containing the empty tuple) with **true**. A *variable valuation* $\mathbf{v}$ is a mapping assigning an object $\mathbf{v}(x) \in \mathcal{U}$ to every individual variable x. A *first order model* $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I}, \mathbf{v})$ for the signature Σ consists of universe $\mathcal{U}$, interpretation $\mathcal{I}$, and variable valuation $\mathbf{v}$.

Any first order model $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I}, \mathbf{v})$ determines a unique object $t^{\mathcal{M}}$ for every object term t. This *denotation* of terms is defined in the usual way:

- $x^{\mathcal{M}} \triangleq \mathbf{v}(x)$, if $x \in \mathcal{V}$ is an individual variable, and
- $(f t_1 ... t_n)^{\mathcal{M}} \triangleq \mathcal{I}(f)(t_1^{\mathcal{M}} ... t_n^{\mathcal{M}})$, i.e., the value of function $\mathcal{I}(f)$ at $(t_1^{\mathcal{M}} ... t_n^{\mathcal{M}})$.

Similar as for propositional logic, we define when a model $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I}, \mathbf{v})$ validates a formula φ.

- $\mathcal{M} \models R(t_1 ... t_n)$ iff $(t_1^{\mathcal{M}}, ..., t_n^{\mathcal{M}}) \in \mathcal{I}(R)$,

- $\mathcal{M} \models t_1 = t_2$ iff $t_1^{\mathcal{M}} = t_2^{\mathcal{M}}$; i.e., iff t_1 and t_2 denote the same object in $\mathcal{U}$,
- $\mathcal{M} \not\models \bot$,
- $\mathcal{M} \models (\varphi \to \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$,
- $\mathcal{M} \models \exists x\ \varphi$ iff $(\mathcal{U}, \mathcal{I}, \mathbf{v}') \models \varphi$ for some valuation $\mathbf{v}'$ which differs from $\mathbf{v}$ at most in x.

In first order logic it is not possible to uniquely characterize the equality relation; therefore we introduced it syntactically (not as one of the relations in the signature) with a special semantical clause. For closed formulas φ, the validation relation is independent of the variable valuation: $(\mathcal{U}, \mathcal{I}, \mathbf{v}) \models \varphi$ iff $(\mathcal{U}, \mathcal{I}, \mathbf{v}') \models \varphi$.

A *Hilbert system* is a set of formulas called *axioms*, and a set of *rules* of the form $\psi_1, ..., \psi_n \vdash \varphi$, where $\psi_1 ... \psi_n$ are the *premisses* and φ is the *consequence* of the rule. A *derivation* of φ from the *assumptions* Φ is a finite sequence of formulas, such that the last element of the sequence is φ, and every element of the sequence is either an element of Φ, or a substitution instance of an axiom, or a substitution instance of the consequence of a rule, where all premisses of the rule for this substitution appear already in the derivation. We write $\Phi \vdash \varphi$ if φ is derivable from Φ.

The following axiom system for first order logic is sound and complete:

(PL1)	$(\varphi \to (\psi \to \varphi))$
(PL2)	$((\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi)))$
(PL3)	$((\neg\varphi \to \neg\psi) \to (\psi \to \varphi))$
(inst)	$(\forall x\ \varphi \to \varphi[x := t])$
(eq1)	$x = x$
(eq2)	$(x = y \to (\varphi \to \varphi[x := y])$
(MP)	$\varphi,\ (\varphi \to \psi) \vdash \psi$
(Gen)	$(\varphi \to \psi) \vdash (\varphi \to \forall x\ \psi)$, if x is not free in φ

In axioms **(PL1)** and **(eq2)**, the formula $\varphi[x := t]$ is obtained from φ by replacing every free occurrence of the individual variable x by the term t. If φ contains the free variables $x_1, x_2, ...$, then we write $\varphi[t_1, t_2, ...]$ for $\varphi[x_1 := t_1][x_2 := t_2][...]$.

1.2 Incompleteness of First Order Arithmetik

Assume that the signature consists of the 0-ary function symbol 0, unary function symbol *succ*, and binary function symbols + and $*$. The *Peano axioms* describe important arithmetical properties of these functions:

(S1)	$\forall x\ succ(x) \neq 0$
(S2)	$\forall x_1 \forall x_2 (succ(x_1) = succ(x_2) \to x_1 = x_2)$
(A1)	$\forall x\ x + 0 = x$
(A2)	$\forall x_1 \forall x_2\ (x_1 + succ(x_2)) = succ(x_1 + x_2)$
(M1)	$\forall x\ (x * 0) = 0$
(M2)	$\forall x_1 \forall x_2\ (x_1 * succ(x_2)) = ((x_1 * x_2) + x_1)$
(Ind)	$\big(\varphi[0] \wedge \forall x(\varphi[x] \to \varphi[succ(x)]) \to \forall x\ \varphi[x]\big)$

We write n for $succ^n(0)$, i.e. for $succ(succ(\ldots(succ(0)...)$. Each formula consists of a finite string of symbols in the alphabet $\mathcal{A} = \{\bot,\ \to,\ (,\ ),\ \exists,\ =,\ 0,\ succ,\ +,\ *,\ x_1,\ x_2,\ ...\}$. Assume a fixed numbering γ for these symbols, e.g.,

$\gamma(\bot) \triangleq 1,\ \gamma(\rightarrow) \triangleq 2,\ \gamma(\,(\,) \triangleq 3,\ \gamma(\,)\,) \triangleq 4,\ \gamma(\exists) \triangleq 5,\ \gamma(=) \triangleq 6,\ \gamma(0) \triangleq 7,\ \gamma(succ) \triangleq 8,\ \gamma(+) \triangleq 9,\ \gamma(*) \triangleq 10,\ \gamma(x_1) \triangleq 11,\ \gamma(x_2) \triangleq 12,\ ...$

With this coding, each formula can be described by a finite string of (positive) natural numbers. Let p_i be the ith prime number, and consider the coding γ^* from sequences of positive numbers into positive numbers:

$$\gamma^*((n_1, ..., n_m)) \triangleq 1 \cdot 2^{n_1} \cdot 3^{n_2} \cdot 5^{n_3} \cdot \ldots \cdot p_m^{n_m}$$

γ^* is bijective, that is, for each sequence of numbers there is exactly one number encoding this sequence. This property guarantees that the inverse of γ^* is a unique decoding of numbers into sequences of numbers. Using γ and γ^*, each formula φ can be coded as a single number φ^γ. Again, using γ^*, even a whole derivation Φ (which is a sequence of formulas) can be coded as a natural number Φ^γ. Now, the syntactical property of being derivable from the Peano axioms can be described within the theory of natural numbers. Gödel proved that there exists a formula Γ_0 with two free individual variables x_1, x_2 such that $\Gamma_0[m, n]$ is derivable from the Peano axioms iff $m = \varphi^\gamma$ is an encoding of a well-formed formula φ and $n = \Phi^\gamma$ is an encoding of a valid derivation Φ of φ.

Consider the formula $\Gamma_1 \triangleq \neg\exists x_2\ \Gamma_0$, and let Γ_1^γ be the Gödel number of Γ_1. Now consider the formula $\Gamma_2 \triangleq \Gamma_1[\Gamma_1^\gamma](\triangleq \neg\exists x_2\ \Gamma_0[\Gamma_1^\gamma])$. Informally, Γ_2 states that there is no valid derivation of itself.

It can be shown that neither Γ_2 nor $\neg\Gamma_2$ can be derivable. This means, that there are formulas which are true in the theory of natural numbers, but cannot be derived (proved) from the Peano axioms. More generally, this holds for any theory which is sufficiently expressive, i.e., contains the Peano axioms as a subset.

1.3 Halting Problem for Turing Machines

Let $\mathcal{A}$ be any finite alphabet. A *Turing machine* is a tuple $(S, \Delta, S_0, S_{acc})$ consisting of

- a finite set of *control states* S
- a *transition relation* $\Delta \subseteq (S \times \mathcal{A}) \times (S \times \mathcal{A} \times \{-1, 0, 1\})$
- a set of initial states $S_0 \subseteq S$
- a set of accepting states $S_{acc} \subseteq S$

Turing machines operate on an infinite *tape*, which is a mapping $\tau : \mathbf{Z} \rightarrow \mathcal{A}$ assigning each integer numbers a letter from the alphabet. A *configuration* of a Turing machine is a tuple (s, τ, i), where $s \in S$ is a control state, τ is a tape and $i \in \mathbf{Z}$ is a position on the tape. An *execution* of a Turing machine on the *input* τ_0 is a finite or infinite sequence of configurations $\rho_0, \rho_1, \rho_2, ...$ such that

- $\rho_0 = (s_0, \tau_0, 0)$, where $s_0 \in S_0$,
- If $\rho_i = (s, \tau, k)$ and $\rho_{i+1} = (s', \tau', k')$, then there is a transition $(s, \alpha, s', \alpha', j) \in \Delta$ such that $\tau(k) = \alpha$, $\tau'(k) = \alpha'$, $\tau'(x) = \tau(x)$ for all $x \neq k$, and $k' = k + j$.

Informally, when the machine is in state s and reads input symbol α at the current position k of tape τ, then it transitions into control state s', writes the output symbol α' into the current tape position k and sets the new tape position to $k - 1$, k, or $k + 1$, respectively.

The Turing machine M *accepts* the input ρ, if there is a finite execution of M on ρ ending in an accepting state. The Turing machine *halts* on the input ρ, if any execution of M on ρ is finite. It *rejects* an input, if it halts but does not accept.

Note that we defined Turing machines in a full generality which is not necessary for expressivity: Firstly, each finite alphabet can be encoded by the binary alphabet $\mathcal{A} = \{0, 1\}$. Secondly, for each of our Turing machines, there is an "equivalent" one which is *deterministic*: with a single initial state s_0 and a single accepting state s_{acc}, where the transition relation is a function $\Delta : S \times \mathcal{A} \to S \times \mathcal{A} \times \{-1, 1\}$. A deterministic Turing machine has exactly one execution on any given input.

Turing's definition of a theoretical machine is ingenious since it captures the fundamentals of computation: A finite control device with an (in principle unlimited) storage. Note that the word "machine" nowadays should not be taken literally: A Turing machine can be regarded as a program in a very primitive (assembly-like) programming language. In fact, for each computer program in any given high-level language like C, Java, or SML there exists an equivalent Turing machine computing the same result.

In particular, there is a universal Turing UTM machine which can "emulate" all other Turing machines. UTM has two inputs: the description a Turing machine M, and an input τ for M. (We assume an appropriate encoding for the Turing machine M here). UTM simulates the behaviour of M in the following way: It accepts/rejects/loops on input (M, τ) iff M accepts/rejects/loops on τ. This is similar to a BASIC interpreter which can be written in BASIC.

Given any alphabet $\mathcal{A}$, a *language* is a set of words (sequences of letters). A language L is called *recursively enumerable* or *semi-decidable*, if there is a Turing machine accepting exactly the words in L. (For words outside of L, the machine may reject or loop forever). A language is called *recursive* or *decidable*, if there is a Turing machine which accepts all words in L and rejects all words not in L. In other words, L is decidable iff there is a Turing machine which halts on any input x and accepts x iff $x \in L$. It is not hard to see that a language L is recursive iff both L and its complement are recursively enumerable.

The most famous undecidable problem is the halting problem for Turing machines: There cannot be a Turing machine which decides for an input pair (M, τ) such that M is (the encoding of) a Turing machine and τ is an input tape for M whether M halts on τ. Assume for contradiction that HTM ("halting Turing machine") would be such a machine. We modify HTM such that it replaces the second argument with the first one. That is, HTM' is a machine which always halts and accepts (the encoding of) a Turing machine M iff M halts on the input M, on its own encoding. It is easy to modify HTM' into NTM ("nonhalting Turing machine") such that NTM loops on M iff M halts on input M. This yields the contradiction that NTM loops on NTM iff NTM halts on NTM.

Turing machines can be described in first order logic. More precisely, for any Turing machine M there exists a closed formula φ_M such that φ_M is valid iff M halts on any input. From this, it follows that first order satisfiability is not recursively enumerable, i.e., first order validity is not decidable.

1.4 SAT and NP

In contrast to first order logic, propositional logic is decidable. However, in computer science the complexity of a decision procedure is very important. The complexity of a Turing machine computation can be measured as the length of the shortest execution which leads to acceptance. Usually, this length depends on the input τ_0, i.e., on the number of nonblank letters which are initially on the tape. If this number is x, then a *polynomial* computation takes at most $P(x)$ steps for some polynomial P, i.e., some fixed function $P(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x + a_0$. For example, if we can guarantee that for any input of length x the machine stops after at most $17 \cdot x^{23}$ steps, then this is a polynomially bounded computation. Although this sounds like a very large number, typically the bounds associated with

polynomial computations are small (quadratic or cubic). The complexity class NP consists of all languages which can be decided by a polynomially bounded nondeterministic Turing machine. The class P is the class of all languages which can be decided by a polynomially bounded deterministic Turing machine. Perhaps the most famous open problem in theoretical computer science is whether these classes are identical or not. All known implementations of nondeterministic algorithms on deterministic computers use some form of backtracking for the nondeterministic alternatives; hence the complexity increases by an exponential (i.e. nonpolynomial) factor.

For problems in the class P efficient decision algorithms can be given; these problems are "computationally feasible". To handle NP-hard problems which are relevant for practical applications, usually some form of heuristics is employed which constructs an approximative solution.

A language is called *NP-hard*, if there is a polynomial encoding of NP-bounded Turing machines in terms of the language. A language is called *NP-complete*, if it is both in NP and NP-hard.

The prototypical NP-complete problem is the question whether a given propositional formula is satisfiable. This problem is referred to as SAT. To prove that SAT is in NP, we have to give a polynomially time bounded Turing machine which accepts a boolean formula iff it is satisfiable. In essence, this machine nondeterministically chooses any interpretation and then checks whether the chosen interpretation validates the formula. The choice can be done linear in the number of variables, and the checking takes time linear in the length of the formula.

Cook (1971) proved that SAT is NP-hard. That is, for any given (nondeterministic) Turing machine M with time bound $P(x)$ and any finite input τ there is a boolean expression $\varphi_{M,\tau}$ such that $\varphi_{M,\tau}$ is satisfiable iff M accepts τ. For simplicity, consider a binary alphabet. Note that during the execution the machine can visit only the tape positions from $-P(\tau)$ to $P(\tau)$. The constructed formula contains a proposition p_{ij} for each of these positions i and each time point $j \leq P(\tau)$. It describes the value these propositions must take in each configuration of the Turing machine during the execution.

Virtually hundreds of other tasks in computer science have been shown to be NP-complete. Among these are a number of graph problems such as to find a clique of size k, finding Hamiltonian cycles, the travelling salesman problem, and others. Mostly, the proof is done by reducing the problem to boolean satisfiability.

2 Program construction

NOTE: This section is yet to be written.

3 Software Specification

A main topic in philosophical logic is the formalization of natural language. In the 1960s, D. Montague, A. Prior, H. Kamp and others investigated modal and temporal logics which provide operators similar to those in English. In the late 1970s, Burstall, Kröger and Pnueli suggested to use these logics for the specification and verification of programs. Later, connections to the theory of ω-automata and certain second order logics, which had been investigated by Büchi and Rabin, were discovered. These connections form the basis for a number of verification tools.

3.1 Temporal logic

First order logic has been criticised by theoretical linguists for not being intuitive. Except from text in mathematical books, one can hardly find English sentences which explicitly use variables to refer to objects. Natural language statements use modal adverbs like "possibly" and "necessarily" to refer to an alternative state of affairs. Temporal phrases in natural language use the adverbs "eventually" and "constantly" (or "sometime" and "always") to refer to future points in time. Modal logic was invented to formalise these modal and temporal adverbs. The idea is to suppress first-order variables $t \in \mathcal{T}$. In modal logics, the meaning of a proposition $\mathtt{p}$ is intended to be "$\mathtt{p}$ holds *now*". Thus, in a temporal interpretation, every formula describes a certain state of affairs *at a given point*.

To be able to describe properties depending on the relations between points, in multimodal logic for every $R \in \mathcal{R}$ a new operator $\langle R \rangle\ \varphi$ is introduced. The meaning of $\langle R \rangle\ \varphi$ is "possibly φ", i.e., "there exists some t accessible via R such that φ holds at t". Dually, $[R]\ \varphi \triangleq \neg\ \langle R \rangle\ \neg\varphi$ means "necessarily φ"; "for all t accessible via R, it is the case that φ holds at t".

Assume that $\mathcal{U}$ is a nonempty set of *points in time* (or "possible worlds"). An *interpretation* $\mathcal{I}$ for multimodal logic assigns to every $\mathtt{p} \in \mathcal{P}$ and $R \in \mathcal{R}$ a subset $\mathcal{I}(\mathtt{p}) \subseteq \mathcal{U}$ and a relation $\mathcal{I}(R) \subseteq \mathcal{U} \times \mathcal{U}$, respectively. The tuple $\mathcal{F} \triangleq (\mathcal{U}, \mathcal{I})$ is called a *frame* for $\mathcal{P}$ and $\mathcal{R}$. A (Kripke-) *model* $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I}, w_0)$ for multimodal logic consists of a frame $(\mathcal{U}, \mathcal{I})$ and a *current point* $w_0 \in \mathcal{U}$. If $\mathcal{M} = (\mathcal{U}, \mathcal{I}, w_0)$, we say that $\mathcal{M}$ is *based on* the frame $\mathcal{F} = (\mathcal{U}, \mathcal{I})$. Thus, a Kripke model for multimodal logic is similar to a first order model, where the variable valuation $\mathbf{v}$ is replaced by a single designated point w_0.

Validity of a modal formula in a Kripke model $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I}, w_0)$ is defined as follows.

- $\mathcal{M} \models \mathtt{p}$ iff $w_0 \in \mathcal{I}(\mathtt{p})$;
- $\mathcal{M} \not\models \bot$, and
- $\mathcal{M} \models (\varphi \to \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$.
- $\mathcal{M} \models \langle R \rangle\ \varphi$ iff there exists $w_1 \in \mathcal{U}$ with $(w_0, w_1) \in \mathcal{I}(R)$ and $(\mathcal{U}, \mathcal{I}, w_1) \models \varphi$.

We write $w \models \varphi$ instead of $(\mathcal{U}, \mathcal{I}, w) \models \varphi$ whenever the frame $(\mathcal{U}, \mathcal{I})$ is given. A formula φ is *universally* valid (or *frame-valid*) in $(\mathcal{U}, \mathcal{I})$, if for all $w \in \mathcal{U}$ it holds that $w \models \varphi$.

The following axioms and rules are complete to derive all universally valid formulas:

(taut)	propositional tautologies
(MP)	$p,\ (p \to q) \vdash q$
(N)	$q \vdash [R]\ q$
(K)	$\vdash ([R]\ (p \to q) \to ([R]\ p \to [R]\ q))$

Given any Kripke model $\mathcal{M}$, assume that $\prec$ is the *transition relation*, i.e., the union of all accessibility relations. Let $<$ be the transitive closure of $\prec$, and $\leq$ the reflexive transitive closure (the *reachability relation*).

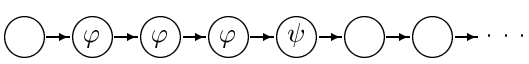
Modal operators cannot express statements about intervals. To remedy this lack of expressiveness, Kamp (1968) introduced a binary operator $(\varphi\ \mathbf{U}^+\ \psi)$ meaning "φ holds until ψ holds". The syntax of *linear temporal logic* (**LTL**) is defined as follows:

$$\mathbf{LTL} \quad ::= \quad \mathcal{P} \ \mid\ \bot \ \mid\ (\mathbf{LTL} \to \mathbf{LTL}) \ \mid\ (\mathbf{LTL}\ \mathbf{U}^+\ \mathbf{LTL})$$

The semantics is:

- $\mathcal{M} \models (\varphi\ \mathbf{U}^+\ \psi)$ iff there exists $w_1 \in \mathcal{U}$ with $w_0 < w_1$ and $w_1 \models \psi$, and for all $w_2 \in \mathcal{U}$ with $w_0 < w_2$ and $w_2 < w_1$, we have $w_2 \models \varphi$.

This situation can be illustrated with a picture.



Various other operators can be defined via $\mathbf{U}^+$. Sometime-operator and nexttime operators are defined as follows:

- $\mathbf{X}\ \varphi \triangleq (\bot\ \mathbf{U}^+\ \varphi)$
- $\mathbf{F}^+\ \varphi \triangleq (\top\ \mathbf{U}^+\ \varphi)$

Let $\overline{\mathbf{X}}\ \varphi \triangleq \neg\ \mathbf{X}\ \neg\varphi$ be the *weak nexttime* operator, and $\mathbf{G}^+\ \varphi \triangleq \neg\ \mathbf{F}^+\ \neg\varphi$ be the *always operator*. The following axiom system for **LTL** is sound and complete.

(taut)	propositional tautologies
(MP)	$p,\ (p \to q) \vdash q$
(N)	$q \vdash (\overline{\mathbf{X}}\ q)$
(K)	$\vdash (\overline{\mathbf{X}}\ (p \to q) \to (\overline{\mathbf{X}}\ p \to \overline{\mathbf{X}}\ q))$
(Rec)	$\vdash (\mathbf{X}\ (q_2 \vee q_1 \wedge (q_1\ \mathbf{U}^+\ q_2)) \to (q_1\ \mathbf{U}^+\ q_2))$
(Ind)	$(\mathbf{X}\ (q_2 \vee q_1 \wedge p) \to p) \vdash ((q_1\ \mathbf{U}^+\ q_2) \to p)$

(Rec) and **(Ind)** are the *recursion axiom* and *induction rule*, resp. Recursion allows to unfold an $\mathbf{U}^+$ operator as

$$(\varphi\ \mathbf{U}^+\ \psi) \leftrightarrow \mathbf{X}\ (\psi \vee \varphi \wedge\ \mathbf{X}\ (\psi \vee \varphi \wedge\ \mathbf{X}\ (\psi \vee \varphi \wedge \cdots)...)$$

Induction allows to derive $\mathbf{G}^+\ \varphi$ from an *invariant*, i.e., from a formula ψ such that ψ holds initially and $(\psi \to \overline{\mathbf{X}}\ (\varphi \wedge \psi))$ holds universally.

3.2 ω-automata and -languages

Given a (finite or infinite) natural model $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I}, w_0)$, the interpretation $\mathcal{I}$ defines a mapping $\mathcal{I} : \mathcal{P} \to 2^{\mathcal{U}}$ from propositions into subsets of the universe. Define a *labelling function* $\mathcal{L} : \mathcal{U} \to 2^{\mathcal{P}}$ by

$$\mathtt{p} \in \mathcal{L}(w) \qquad \text{iff} \qquad w \in \mathcal{I}(\mathtt{p})$$

That is, $\mathcal{L}(w) \triangleq \{\mathtt{p} \mid w \in \mathcal{I}(\mathtt{p})\}$ is the *label* of point $w \in \mathcal{U}$. If $\mathcal{U} = (w_0, w_1, w_2, ...)$, then the sequence $\sigma = (\mathcal{L}(w_0), \mathcal{L}(w_1), \mathcal{L}(w_2), ...)$ is called the *ω-word* of $\mathcal{M}$ over the *alphabet* $\Sigma \triangleq 2^{\mathcal{P}}$. A set of ω-words is called an *ω-language*.

Let $\mathcal{F} \triangleq (\mathcal{U}, \mathcal{I})$ be the frame of a natural model. Formula φ is *initially valid* in $\mathcal{F}$, if $(\mathcal{U}, \mathcal{I}, w_0) \models \varphi$, where w_0 is the unique initial point of $\mathcal{U}$ (which has no predecessors).

We say that a linear-time logic formula *defines* the set of all natural frames in which it is initially valid. Thus every such formula defines the ω-language given by these frames.

Languages can also be defined by *(ω-)regular expressions* and by *finite (ω-) automata*

The language of *(ω-)regular expression* is defined similar to the language of usual regular expressions, with an additional operation denoting infinite repetition of a subexpression.

- Every letter from the alphabet is an ω-regular expression.
- If α and β are ω-regular expressions, then so are ε, $(\alpha + \beta)$, $(\alpha; \beta)$ and α^+.
- If α is an ω-regular expression, then so is α^ω.

Every ω-regular expression defines an ω-language: The letter $a \subseteq \mathcal{P}$ defines $\{(a)\}$, i.e., a one-word language (one-element set) consisting of a one-letter word (one-element sequence). ε denotes the empty language, and $(\alpha + \beta)$, $(\alpha; \beta)$ and α^+ denote union, sequential composition and finite iteration of languages. α^ω denotes the language of all words consisting of an infinite concatenation of words from α. A language is called ω-regular if it can be defined by an ω-regular expression.

We use boolean terms over $\mathcal{P}$ to denote (unions of) letters. For example, if $\mathcal{P} = \{\mathtt{p1}, \mathtt{p2}\}$ then $(\neg \mathtt{p1} \wedge \mathtt{p2})$ denotes the letter $\{\mathtt{p2}\}$, and $(\neg \mathtt{p1} \vee \mathtt{p2})$ denotes $\{\} + \{\mathtt{p2}\} + \{\mathtt{p1,p2}\}$.

As an example for an ω-regular expression, consider $(\neg \mathtt{p1})^\omega + (\top^+; \mathtt{p2})^\omega$. This expression defines the set of all infinite words $(\sigma_0, \sigma_1, \sigma_2, ...)$ such that either for all i it holds that $\mathtt{p1} \notin \sigma_i$, or for infinitely many i it holds that $\mathtt{p2} \in \sigma_i$. That is, it defines the set of natural models $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{G}^* (\neg \mathtt{p1} \wedge \mathbf{X}\ \top) \vee \mathbf{G}^* \mathbf{F}^+\ \mathtt{p2}$. Since this formula implies $\mathbf{G}^* \mathbf{X}\ \top$, each of its natural models must be infinite.

An *ω-automaton* or *fair transition system* over the alphabet $\Sigma = 2^\mathcal{P}$ is defined like a usual (nondeterministic) automaton with an additional recurrence set ("fairness constraint"); it is a tuple $(S, \Delta, S_0, S_{acc}, S_{rec})$, where

- S is a set of states,
- $\Delta \subseteq S \times \Sigma \times S$ is the transition relation,
- $S_0 \subseteq S$ is the set of initial states,
- $S_{acc} \subseteq S$ is the set of accepting states (for finite words), and
- $S_{rec} \subseteq S$ is the set of recurring states (for infinite words).

A *Büchi-automaton* is a finite ω-automaton, that is, a fair transition system where the set S of states is finite. A *transition system* is a fair transition system where $S_{acc} = S_{rec} = S$. A *weakly fair transition system* is an ω-automaton where $S_{rec} = S$ and $S_{acc} = \{s \mid \forall a, s'(s, a, s') \notin \Delta\}$. That is, in a weakly fair transition system all states are recurring, and states are accepting iff they are terminal.

A (finite or infinite) nonempty word $\sigma \triangleq (\sigma_0, \sigma_1, ...)$ is *accepted* by an automaton $(S, \Delta, S_0, S_{acc}, S_{rec})$, if there is a function ρ assigning to any letter σ_i a state $\rho(\sigma_i) \in S$ of the automaton such that

- $\rho(\sigma_0) \in S_0$,
- For all $0 \leq i < n$, $(\rho(\sigma_i), \sigma_i, \rho(\sigma_{i+1})) \in \Delta$, and
- $(\rho(\sigma_n), \sigma_n, s) \in \Delta$ for some $s \in S_{acc}$, if σ is finite with last letter w_n, and
- $\mathit{inf}(\rho) \cap S_{rec} \neq \emptyset$, if σ is infinite, where $\mathit{inf}(\rho)$ is the set of states that appear infinitely often in the range of ρ. That is, at least one recurring state must be selected infinitely often.

There are a number of alternative acceptance conditions, see the survey of Thomas in the Handbook of Theoretical Computer Science. We say that an automaton accepts a natural model $\mathcal{M}$, if it accepts the ω-word of $\mathcal{M}$. The language of a transition system consists of all paths through the transition graph; this language is prefix-closed (for any word in the language, all of its prefixes are also contained).

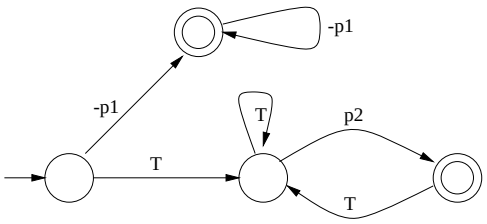


Figure 1: A Büchi automaton accepting $(\neg \mathtt{p1})^\omega + (\top^+; \mathtt{p2})^\omega$

The language defined by a weakly fair transition system consists of all maximal paths through the graph. As an example of a Büchi-automaton, consider Figure 1.

This automaton accepts (i.e., defines) exactly the same language as the example ω-regular expression above. In general, for any ω-regular expression we can construct such a Büchi-automaton and vice versa; Büchi-automata can define all and only ω-regular languages.

ω-regular expressions and Büchi-automata are of equal expressive power. The proof of this statement is similar as for automata on finite words: For one direction, we have to show that the Büchi acceptance condition can be captured by an appropriate regular expression. Let $L(s_i, s_j)$ be a regular expression for the language of finite nonempty words sending an automaton from state s_i into state s_j. Then the ω-regular expression associated with any Büchi-automaton is

$$\Sigma\{L(s_0, s) \mid s_0 \in S_0,\ s \in S_{acc}\} + \Sigma\{L(s_0, s); L(s, s)^\omega \mid s_0 \in S_0,\ s \in S_{rec}\}$$

For the other direction it must be shown that Büchi-automata are closed under single letters, the empty language, union, concatenation, and finite and infinite repetition. All of these constructions are straightforward.

The automaton resulting from this proof is highly nondeterministic. An automaton is called *deterministic*, if its transition relation is a function $\Delta : S \times \Sigma \to S$. For each nondeterministic finite automaton on finite words an equivalent deterministic one is given by the well known powerset construction of Rabin and Scott. The same holds for finite transition systems. In contrast, for nondeterministic Büchi-automata it is not always possible to construct an equivalent deterministic one. For example, consider the language $\mathcal{L}$ of all words containing only finitely many $\mathtt{p}$. This language is defined by the formula $\mathbf{F}^*\mathbf{G}^+ \neg\mathtt{p}$ or the ω-regular expression $(\top^+ + (\top^+; \neg\mathtt{p}^\omega))$. However, there is no determinictic Büchi-automaton defining $\mathcal{L}$: Assume for contradiction that $\mathcal{L}$ is the language of $\mathcal{A}$. Then $\mathcal{A}$ must accept $(\sigma; (\neg\mathtt{p})^\omega)$ for any finite word σ. In particular, from any reachable state some recurring state is reached by a finite number of $\neg\mathtt{p}$-transitions. Let m be the maximum of these numbers. Therefore, in the run of $\mathcal{A}$ on the word $(\mathtt{p}; (\neg\mathtt{p})^m)^\omega$ infinitely often recurring states are visited. Thus, this word also is accepted by $\mathcal{A}$. This is a contradiction, since it is not in $\mathcal{L}$.

3.3 Automata and Logics

Trivially, ω-automata are closed under projection onto a smaller alphabet. Büchi (1962) showed that his automata are closed under complement; this is a highly nontrivial proof. The best known construction for complementing Büchi-automata was given by Safra (1988).

Closure under complement can be used to show that Büchi-automata can express all **LTL** properties: For every **LTL** formula there is a Büchi-automaton defining the same language. A more direct proof uses a tableau decomposition of **LTL** formulas.

The converse does not hold: there are languages definable by Büchi automata which cannot be defined in temporal logic. Therefore, several extensions of **LTL** have been suggested, most notably the propositional μ-calculus, and temporal logic with propositional quantification.

The embedding of temporal logic into automata theory allows to use automata theoretic tools for the verification of temporal logic formulas. For example, the formula φ is valid, iff $\neg\varphi$ defines the empty language. This can be checked by constructing the automaton for $\neg\varphi$ and checking whether the accepted language is empty.

We now give some examples of **LTL** formulas. The following properties are typical correctness requirements that might arise in the verification of a finite state concurrent program.

— $\mathbf{F}^+$ (`started` $\wedge \neg$`empty`): Eventually a state will be reached where `started` holds but `empty` does not hold.

— $\mathbf{G}^+$ (`req` $\rightarrow \mathbf{F}^+$ `ack`): If a request occurs, then it will be eventually acknowledged

— $\mathbf{G}^+ \ \mathbf{F}^+$ `stack_is_empty`: The statement `stack_is_empty` holds infinitely often.

— $\mathbf{F}^+ \ \mathbf{G}^+$ `stack_is_empty`: At some point, `stack_is_empty` holds forever after.

4 Theorem Proving and Model Checking

Since the advent of the computer, theoreticians have tried to use it as a tool for proving logical formulas. Interest in this research has be driven by two motivations: firstly, the hope to be able to prove mathematical theorems which are too complex or too tedious for humans. Secondly, many technical systems, especially software systems, can be described by logical formulas, and logical reasoning can be used to establish the correctness of these systems.

Whereas for several decades the methods could only be applied to small examples, in revent years theorem proving and model checking could be used for highly complex theorems and systems. This is partly due to the increase of available computing resources, and partly due to sophisticated algorithms and data structures.

4.1 Automated theorem proving

The most successful methods in automated first order theorem proving are based on a method called *resolution* which was developed by A. Robinson in 1965. The following definitions are from the chapter on "Resolution Decision Procedures" by C. Fermüller, A. Leitsch and T. Tammet, to appear in the Handbook of Automated Deduction.

A first order *atom* is a formula $R(t_1, ..., t_n)$; a *literal* is either an atom or an atom preceded by a negation sign. An *expression* is either a term or a literal. A *clause* is a finite set of literals; the empty clause is denoted by $\Box$. The *dual* L^d of a literal L is $\neg L$, if L is unsigned; otherwise, if L is of the form $\neg\varphi$, then $L^d \triangleq A$. For a set of literals $C = \{L_1, ..., L_n\}$, we define $C^d \triangleq \{L_1^d, ..., L_n^d\}$. Additionally, let C_+ be the set of positive (unsigned) literals of a clause C, and C_- the set of negative literals (negated atoms) in C.

C is a *Horn clause*, if it contains at most one positive literal, i.e., if $|C_+| \leq 1$.

The *term depth* $\tau(t)$ of a term t is defined by

- $\tau(t) = 0$ if t is a variable or a constant

- $1 + \max\{\tau(t_i) \mid 1 \leq i \leq n\}$, if $t = f\ t_1, ..., t_n$

The term depth of a literal is the maximal term depth of its arguments. Likewise, the term depth of a clause or set of clauses is the maximal term depth of its constituents.

The set of variables occurring in E is called $\mathcal{V}(E)$; if C is a clause, then $\mathcal{V}(C)$ is the union over all $\mathcal{V}(p_i)$ for all atoms $p_i \in C$. We define E_1 and E_2 to be *variable disjoint* if $\mathcal{V}(E_1) \cap \mathcal{V}(E_2) = \emptyset$. An expression or a clause is called *ground* if no variables occur in it. We call it *constant free* if no constants occur in it, and *function free* if it does not contain function symbols.

Let $\mathcal{V}$ be the set of variables and T be the set of terms. A *substitution* is a mapping $\sigma : \mathcal{V} \to T$ such that $\sigma(x) = x$ almost everywhere. We call the set $\{x|\sigma(x) \neq x\}$ the *domain* $dom(\sigma)$ of σ and $\{\sigma(x)|x \in dom(\sigma)\}$ the *range* $rg(\sigma)$ of sigma. By ε we denote the *empty substitution*, i.e., $\varepsilon(x) = x$ for all variables x. We say that a substitution σ is *based on a clause set* S if no other constants and function symbols besides that in S occur in the terms of $rg(\sigma)$. A *ground substitution* is a substitution σ such that there are only ground terms in $rg(\sigma)$.

An expression E_1 is an *instance* of another expression E_2 if there is a substitution σ such that $E_1 = E_2\sigma$, and similar for clauses. Expressions, substitutions and clauses can be compared using the following ordering relation. For expressions E_1 and E_2, we say that E_2 is *more general than* E_1 ($E_1 \leq_s E_2$) if there is a substitution σ such that $E_1\sigma = E_2$. For substitutions ρ and *thetea* we define analogously $\rho \leq_s \theta$ iff there exists a substitution σ such that $\rho\sigma = \theta$. Similar for clauses C and D; if $C \leq_s D$ we say that C *subsumes* D.

A set of expressions M is *unifiable* by a substitution σ if $E_i\sigma = E_j\sigma$ for all E_i, $E_j \in M$. Substitution σ is called *most general unifier* (mgu) of M if for every other unifier ρ of M it holds that $\sigma \leq_s \rho$. We say that E_1 is *unifiable with* E_2 if $\{E_1, E_2\}$ is unifiable. Any two different mgu's of a set of expressions can only differ in the names of the variables.

A *factor* of a clause C is a clause $C\theta$, where θ is a mgu of some $C' \subseteq C$. In case $|C\theta| < |C|$ we call the factor *nontrivial*. A clause C is *condensed* if there exists no nontrivial factor of C which is a subclause of C. If C' is a condensed factor of C such that $C' \subseteq C$, then C' is called a condensation of C. Condensations are unique up to renaming. As an example, consider $\{P(x, y), P(y, x)\}$. This set is condensed. $\{P(x, y), P(x, a)\}$ is not condensed; its condensation is $\{P(x, a)\}$.

If C and D are variable disjoint clauses and M and N are subsets of C and D, respectively, such that $N^d \cup M$ is unifiable by the mgu θ, then $E = (C - M)\theta \cup (D - N)\theta$ is a *Robinson-resolvent* of C and D. If M and N are singleton sets, then E is called *binary resolvent* of C and D. The atom A of $(N^d \cup M)\theta$ is called the *resolved atom*. We also say that E is *generated via* A. The elements of N and M are called the *literals resolved upon*.

For a clause set S we define $RES(S)$ as the set of Robinson-resolvents of S. Furthermore, let $R(S) \triangleq S \cup RES(S)$ and

- $R^0(S) \triangleq S$
- $R^{i+1}(S) = R(R^i(S))$

Finally, $R^*(S) \triangleq \bigcup_i R^i(S)$. The clause C is derivable from a clause set S iff $C \in R^*(S)$. The most basic theorem about resolution is:

> A set S of clauses is unsatisfiable iff $\Box$ is derivable from S.

Thus, resolution theorem provers work by converting a given formula into clause form and deriving resolvents from it. If the empty clause is derived, the original formula is refuted; otherwise, the process may go on forever.

4.2 Program Verification by Model Checking

In contrast to resolution theorem proving, *model checking* is a decision procedure for the correctness of finite state programs. To establish the correctness of a software system, the system to be verified is modeled as a finite state transition graph, and the properties are formulated in an appropriate propositional temporal logic. An efficient search procedure is then used to determine whether or not the state transition graph satisfies the temporal formulas. When model checking was first developed in 1981, it was only possible to handle concurrent systems with a few thousand states. In the last few years, however, the size of the concurrent systems that can be handled has increased dramatically. By using sophisticated data structures and heuristic search procedures, it is now possible to check systems many orders of magnitude larger.

For model checking often a branching time temporal logic is used as specification language. Statements about correctness of program can involve assertions about *all maximal paths* in a tree. A *path* in a model is a (finite or infinite) nonempty sequence of points $\sigma = (w_0, w_1, ...)$, where for each i with $0 \leq i < |\sigma|$ there exists an $R_i \in \mathcal{R}$ such that $(w_i, w_{i+1}) \in \mathcal{I}(R_i)$. A path is *maximal*, if each of its points which has a successor in the model also has a successor in the path. In other words, a maximal path is either infinite, or its final point w_n is terminal (there is no w such that $w_n \prec w$). Computation tree logic (**CTL**) (developed by Clarke and Emerson in 1981) has the following syntax:

$$\mathbf{CTL} \quad ::= \quad \mathcal{P} \mid \bot \mid (\mathbf{CTL} \rightarrow \mathbf{CTL}) \mid \mathbf{E}(\mathbf{CTL}\ \mathbf{U}^+\ \mathbf{CTL}) \mid \mathbf{A}(\mathbf{CTL}\ \mathbf{U}^+\ \mathbf{CTL}).$$

CTL is interpreted on *tree models*. A tree is defined as usual: It has a single root w_0, and every node w_n can be reached from w_0 by exactly one finite path. The transitive closure "$<$" of the successor relation "$\prec$" then denotes the usual tree-order: $(w_1, w_2) \in \mathcal{I}(<)$ iff w_1 is on the (unique) path from the root w_0 up to w_2.

- $w_0 \models \mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ iff there exists $w_1 > w_0$ such that $w_1 \models \psi$, and for all $w_2 \in \mathcal{U}$, if $w_0 < w_2 < w_1$ then $w_2 \models \varphi$.
- $w_0 \models \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$ iff *for all maximal paths* p from w_0 there exists $w_1 > w_0$ *on path* p such that $w_1 \models \psi$, and for all $w_0 < w_2 < w_1$, $w_2 \models \varphi$.

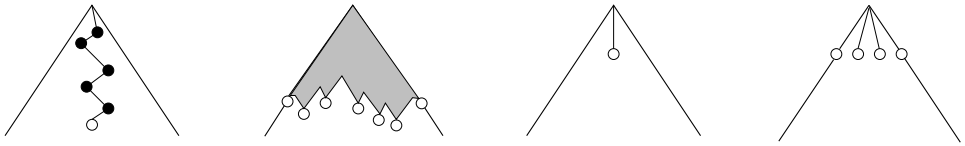
Thus, the $\mathbf{E}\ \mathbf{U}^+$-operator is defined similar to the **LTL** until-operator. However, the intended models for **CTL** are trees, whereas **LTL** usually is interpreted on natural models. In **CTL** weak and derived operators can also be defined as abbreviations. However, in branching time, there are two variants of each derived operator.

$$\begin{array}{cc}
\mathbf{E}\ \mathbf{X}\ \psi \triangleq \mathbf{E}(\bot\ \mathbf{U}^+\ \psi), & \mathbf{A}\ \mathbf{X}\ \psi \triangleq \mathbf{A}(\bot\ \mathbf{U}^+\ \psi), \\
\mathbf{E}\ \not\mathbf{X}\ \psi \triangleq \neg\mathbf{A}\ \mathbf{X}\ \neg\psi, & \mathbf{A}\ \not\mathbf{X}\ \psi \triangleq \neg\mathbf{E}\ \mathbf{X}\ \neg\psi, \\
\mathbf{E}\ \mathbf{F}^+\ \psi \triangleq \mathbf{E}(\top\ \mathbf{U}^+\ \psi), & \mathbf{A}\ \mathbf{F}^+\ \psi \triangleq \mathbf{A}(\top\ \mathbf{U}^+\ \psi), \\
\mathbf{E}\ \mathbf{G}^+\ \psi \triangleq \neg\mathbf{A}\ \mathbf{F}^+\ \neg\psi, & \mathbf{A}\ \mathbf{G}^+\ \psi \triangleq \neg\mathbf{E}\ \mathbf{F}^+\ \neg\psi, \\
\mathbf{E}(\varphi\ \mathbf{U}^*\ \psi) \triangleq (\psi \vee \varphi \wedge \mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)), & \mathbf{A}(\varphi\ \mathbf{U}^*\ \psi) \triangleq (\psi \vee \varphi \wedge \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)), \\
\mathbf{E}\ \mathbf{F}^*\ \psi \triangleq (\psi \vee \mathbf{E}\ \mathbf{F}^+\ \psi), & \mathbf{A}\ \mathbf{F}^*\ \psi \triangleq (\psi \vee \mathbf{A}\ \mathbf{F}^+\ \psi), \\
\mathbf{E}\ \mathbf{G}^*\ \psi \triangleq (\psi \wedge \mathbf{E}\ \mathbf{G}^+\ \psi), & \mathbf{A}\ \mathbf{G}^*\ \psi \triangleq (\psi \wedge \mathbf{A}\ \mathbf{G}^+\ \psi)
\end{array}$$

Informally, $\mathbf{E}\ \mathbf{X}\ \psi$ means that some successor node satisfies ψ, and $\mathbf{A}\ \not\mathbf{X}\ \psi$ holds if all successors are ψ. In a terminal point, $\mathbf{A}\ \not\mathbf{X}\ \bot$ is valid, but $\mathbf{A}\ \mathbf{X}\ \bot$ not: If w_0 has no successors, then the only maximal path p from w_0 is the one-element sequence $\sigma = (w_0)$. On this unique path σ there is no $w_1 > w_0$, therefore each

formula $\mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$ and $\mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ must be invalid. As a special case, in such a point $\mathbf{E}\ \mathbf{X}\ \top$ is not valid, but $\mathbf{E}\ \not\mathbf{X}\ \top$ and $\mathbf{E}\ \not\mathbf{X}\ \bot$ are valid. In a nonterminal point, $(\mathbf{E}\ \mathbf{X}\ \varphi \leftrightarrow \mathbf{E}\ \not\mathbf{X}\ \varphi)$ and $(\mathbf{A}\ \mathbf{X}\ \varphi \leftrightarrow \mathbf{A}\ \not\mathbf{X}\ \varphi)$. Thus, if we restrict attention to models without terminal points, these operators coincide. The operators $\mathbf{A}\ \mathbf{X}$ and $\mathbf{E}\ \not\mathbf{X}$ can be expressed by $\mathbf{E}\ \mathbf{X}$ and $\mathbf{A}\ \not\mathbf{X}$ (with at most linear increase of formula length) via $(\mathbf{A}\ \mathbf{X}\ \varphi \leftrightarrow \mathbf{A}\ \not\mathbf{X}\ \varphi \wedge \mathbf{E}\ \mathbf{X}\ \top)$ and $(\mathbf{E}\ \not\mathbf{X}\ \varphi \leftrightarrow \mathbf{E}\ \mathbf{X}\ \varphi \vee \mathbf{A}\ \not\mathbf{X}\ \bot) \leftrightarrow (\mathbf{E}\ \mathbf{X}\ \top \rightarrow \mathbf{E}\ \mathbf{X}\ \varphi))$. Thus, all **CTL** nexttime-operators can be expressed in terms of $\mathbf{E}\ \mathbf{X}$.

The formula $\mathbf{E}\ \mathbf{F}^*\ \psi$ means that some node in the computation tree satisfies ψ, and $\mathbf{A}\ \mathbf{F}^*\ \psi$ specifies that ψ must hold somewhere along every maximal computation path. Dually, $\mathbf{A}\ \mathbf{G}^*\ \psi$ means that every node in the (sub-) tree satisfies ψ, whereas $\mathbf{E}\ \mathbf{G}^*\ \psi$ indicates that ψ is globally valid along some path.



$\mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ $\mathbf{A}(\varphi\ \mathbf{U}^*\ \psi)$ $\mathbf{E}\ \mathbf{X}\ \psi$ $\mathbf{A}\ \mathbf{X}\ \psi$

In the above picture, nodes satisfying φ are shown solid (or as a shaded area), whereas ψ nodes are indicated by a circle.

Given a model $\mathcal{M}$ and a formula φ, the model checking problem is to decide whether $\mathcal{M} \models \varphi$. Given a Kripke frame $\mathcal{F} = (\mathcal{U}, \mathcal{I})$ and a multimodal formula φ, the set $\varphi^{\mathcal{F}} \triangleq \{w \in \mathcal{U} \mid w \models \varphi\}$ of points validating φ can be calculated by a recursive descent on the structure of φ. If $\mathtt{p}$ is an atomic proposition, then $\mathtt{p}^{\mathcal{F}} \triangleq \mathcal{I}(\mathtt{p})$. Furthermore, $\bot^{\mathcal{F}} \triangleq \{\}$ and $(\varphi \rightarrow \psi)^{\mathcal{F}} \triangleq \mathcal{U} \setminus \varphi^{\mathcal{F}} \cup \psi^{\mathcal{F}}$. Finally, $(\langle R \rangle\ \psi)^{\mathcal{F}} \triangleq \{w \in \mathcal{U} \mid \exists w' \in \psi^{\mathcal{F}}, (w, w') \in \mathcal{I}(R)\}$.

This algorithms seems to be just a trivial reformulation of the semantical definition for the logical operators. However, there are some important observations. Firstly, $(\langle R \rangle\ \psi)^{\mathcal{F}}$ can be calculated from $\psi^{\mathcal{F}}$ in two ways: We can either check for each $w \in \mathcal{U}$, whether the intersection of $\psi^{\mathcal{F}}$ and $R(w)$ is nonempty. Alternatively, we can calculate $\bigcup\{R^{-1}(w') \mid w' \in \psi^{\mathcal{F}}\}$, where $R^{-1}(w') \triangleq \{w \mid (w, w') \in \mathcal{I}(R)\}$ is the *inverse image* of point w under the relation R. This inverse image calculation can be accomplished by a traversal of all arcs $(w, w') \in \mathcal{I}(R)$: If $w' \in \psi^{\mathcal{F}}$, then $w \in (\langle R \rangle\ \psi)^{\mathcal{F}}$. Secondly, to avoid recalculation of common subformulas, we use a table, where for each sub-formula ψ the set $\psi^{\mathcal{F}}$ is stored. Thus, we need an efficient data structure for large sets of points. Thirdly, the overall *complexity* of this algorithm is linear in the number of different sub-formulas and in the size of the model. However, even for infinite models which are given by some symbolic description (e.g., C programs or Turing machines), the model checking problem can be decidable.

Similar to the above modal logic procedure, the **CTL** model checking algorithm proceeds by marking each point with the set of sub-formulas which are valid for this point. Suppose we have already marked the set of points satisfying ψ_1 and the points satisfying ψ_2. To label the set of points satisfying $\varphi \triangleq \mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1)$ or $\varphi \triangleq \mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1)$, we use the fixpoint unfoldings

$$\mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \mathbf{E}\ \mathbf{X}\ (\psi_1 \vee \psi_2 \wedge \mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1))$$

$$\mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \mathbf{A}\ \mathbf{X}\ (\psi_1 \vee \psi_2 \wedge \mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1))$$

For $\varphi \triangleq \mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1)$, we label all points with φ which have a successor that is labelled with ψ_1, or with ψ_2 and also φ. This process is repeated until stabilisation

is reached. For $\varphi \triangleq \mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1)$, we label all points with φ for which all successors are labelled with ψ_1, or with ψ_2 and also φ. Again, this process must be repeated until no new points can be marked. A recursive formulation of this algorithm is given in Fig. 2.

```
program CTL_check (Model (𝒰, ℐ, w0), Formula φ) =
    if w0 ∈ eval(φ)
    then print("φ is satisfied at w0 in (𝒰, ℐ)")
    else print("φ not satisfied at w0 in (𝒰, ℐ)");
procedure eval (Formula φ): Pointset =
    case φ of
        p : return ℐ(p);
        ⊥ : return {};
        (ψ1 → ψ2) : return 𝒰 \ eval(ψ1) ∪ eval(ψ2);
        E(ψ2 U+ ψ1) : E1 := eval(ψ1); E2 := eval(ψ2); E := {};
            repeat until stabilization
                E := E ∪ {w | (succ(w) ∩ (E1 ∪ E2 ∩ E)) ≠ {}};
            return E;
        A(ψ2 U+ ψ1) : E1 := eval(ψ1); E2 := eval(ψ2); E := {};
            repeat until stabilization
                E := E ∪ {w | succ(w) ⊆ E1 ∪ E2 ∩ E};
            return E;
procedure succ (Point w): Pointset = return {w' | (w, w') ∈ ℐ(≺)};
```

Figure 2: naïve **CTL** model checking algorithm

Since the Kripke-model has a finite number of points, each **repeat** stabilises after at most $|\mathcal{U}|$ passes. In the worst case, each pass searches the whole model, hence the complexity is linear in the number of different sub-formulas, and cubic in $|\mathcal{U}|$.

This bound can be improved if the search is organised better. Clarke, Emerson and Sistla (1986) give an algorithm which is linear in the size of the model as well. For the $\mathbf{E}\ \mathbf{F}^+$-operator, the problem of marking all points for which $\mathbf{E}\ \mathbf{F}^+\ \varphi$ holds, given the set of point satisfying φ, is equivalent to the *inverse reachability problem*: Given a set of points, mark all points from which any finite path leads into the given set. Assuming that for any two points we can decide in constant time whether they are connected by an arc, this can be done with time complexity quadratic in the number of points.

```
procedure reach (Pointset Target): Pointset =
    Source := {}; Search := Target;
    while Search ≠ {} do
        Search := pred (Search) \ Source;
        Source := Source ∪ Search
    enddo;
    return Source;
procedure pred (Point w): Pointset = return {w' | (w', w) ∈ ℐ(≺)};
```

Figure 3: Inverse reachability calculation

The algorithm given in Fig. 3 calculates the set *Source* of all points from which any point in given set *Target* is reachable. In this algorithm, every point enters the set *Search* in the **while** loop at most once. Moreover, all set operations can be performed in time linear in the size of these sets, i.e., in the number of points; thus the overall complexity is quadratic in $|\mathcal{U}|$ or linear in the size of the Kripke-model.

For the $\mathbf{E}\ \mathbf{U}^+$-operator, this idea can be refined to give an evaluation procedure of linear complexity. The $\mathbf{A}\ \mathbf{U}^+$-operator can be expressed by

$$\mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \neg(\mathbf{E}(\neg\psi_1\ \mathbf{U}^+\ (\neg(\psi_1 \wedge \psi_2)) \vee \mathbf{E}\ \mathbf{G}^+\ \neg\psi_1)$$

Thus, we only need a procedure marking all points for which $\mathbf{E}\ \mathbf{G}^+\ \varphi$ holds. This can be done as follows:

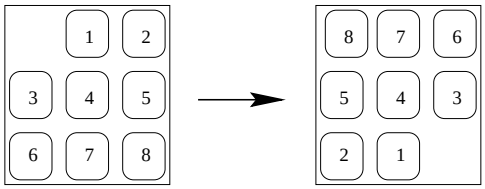
- restrict the model to those states satisfying φ
- find the maximal strongly connected components in the restriction
- mark all points in the original model from which a nontrivial SCC or a point without successors can be reached by a path in the restricted model.

These operations can be accomplished with time complexity which is quadratic in $\mathcal{U}$. Thus, the overall complexity of CTL model checking is linear in the size of the formula and in the size of the model.

4.3 Example Applications

A Combinatorial Game

As a first example, we describe the use of model checking in a combinatorial search. Although this example is not very typical for real applications, it can demonstrate the capabilities and limits of present technology. A well-known puzzle from 1870 by the American Sam Loyd consists of a $h \times v$ grid in which there are $(h \cdot v) - 1$ numbered tiles and one blank space. A move consists in moving any tile into the position of the blank. The goal is to achieve a certain predetermined order on the tiles.



This puzzle can be described by a finite state programming language as follows. For each tile there is a program variable which notes its horizontal and vertical position. Furthermore, there is a program variable `move` indicating whether the next move will be a shift up, down, left or right of the blank space. If the move would bring it out of the borders, nothing is changed; otherwise, its position is swapped with the respective adjacent tile.

The SMV code corresponding to this description[1] is shown in Figure 4. For $h = 3$ and $v = 3$, the internal representation of the transition relation takes about 3KB. There are $4 \cdot (h \cdot v)! = 1.4 \cdot 10^6$ states, of which 50% are reachable from any initial state. The specification claims that a certain final state is *not* reachable;

[1] In the actual SMV code, variable array bounds or indices, e.g., vpos[i], are not allowed and have to be replaced by the respective constant values vpos[1],vpos[2],...

```
MODULE main
DEFINE h := 3; v := 3;
VAR move:  u,d,l,r;
    hpos:  array 0..(h*v-1) of 1..h;
    vpos:  array 0..(h*v-1) of 1..v;
ASSIGN
next(vpos[0]) := case
  (move=l) & !(vpos[0]=1) :  vpos[0] - 1;
  (move=r) & !(vpos[0]=v) :  vpos[0] + 1;
  1:  vpos[0]; esac;
next(hpos[0]) := case
  (move=u) & !(hpos[0]=1) :  hpos[0] - 1;
  (move=d) & !(hpos[0]=h) :  hpos[0] + 1;
  1:  hpos[0]; esac;
for all i:
next(vpos[i]) := case
  (move=l) & !(vpos[0]=1) & hpos[i]=hpos[0] & vpos[i]=vpos[0]+1 |
  (move=r) & !(vpos[0]=v) & hpos[i]=hpos[0] & vpos[i]=vpos[0]-1 :  vpos[0];
  1:  vpos[i]; esac;
next(hpos[i]) := case
  (move=u) & !(hpos[0]=1) & vpos[i]=vpos[0] & hpos[i]=hpos[0]-1 |
  (move=d) & !(hpos[0]=h) & vpos[i]=vpos[0] & hpos[i]=hpos[0]+1 :  hpos[0];
  1:  hpos[i]; esac;
init(hpos[i]) := i div v; init(vpos[i]) := i mod v;
DEFINE goal := (hpos[i] = 3 - (i div v) & vpos[i] = 3 - (i mod v))
SPEC !EF goal
```

Figure 4: SMV Code for Loyds Puzzle

the model checker contradicts this claim by showing a sequence of moves (rrddlluurrddlluurrddlluurrdd) which gives a solution to the puzzle. The solution is found within a couple of minutes on a 32 MB Pentium 133.

For $h = 4$, $v = 3$, there are approximately 10^9 reachable states. Although the model checker detects rather quickly that some solution must exist, for the construction of a concrete solution sequence the state space has to be partitioned into strongly connected components. This requires several days of CPU time and approximately 1GB RAM on a Sparc Ultra. For model checking applications, virtual memory is not very useful; if the representation of the reachable state space exceeds the available main memory, then constant swapping occurs. To find a solution for $h = 4$, $v = 4$ by exhaustive state space exploration seems to be beyond the limits of present technology.

A Sequential Circuit

Our second example is from hardware verification. We consider a shift register for interfacing a parallel data bus. It is used to exchange data between the bus and a serial device. It thus acts as parallel-serial converter and vice versa. A functional diagram of the register is given in Figure 5.

The register has a *mode control* input `mc` to choose between parallel or serial access mode. For each mode, there is a corresponding input clock (`pc` and `sc`). Parallel loading is performed if `mc` is high and a `pc` clock pulse arrives. In this case, data is read from the bus into the associated flip-flops. The data appears at the `Q` outputs at the pulse of the `pc` clock.
For serial loading, mode control should be low. Data is input serially with every tick of the `sc` clock. At each pulse the state of all flip-flops is transferred one stage to the right. After n cycles, the data is positioned at the parallel output and can

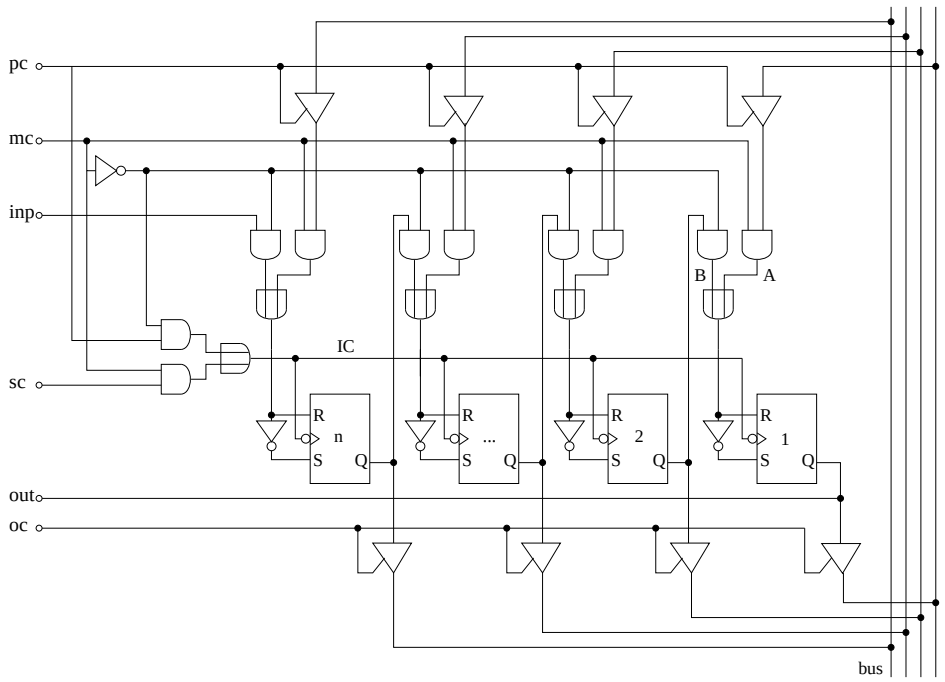


Figure 5: A shift register for data bus interfacing

be sent to the bus by an `oc` command. A right shift occurs if the serial input `inp` is held low. By a sequence of n right shifts, data which has been obtained in parallel from the bus can be written serially to the `out` port.

The register is implemented with SR-bistables which have the following characterictic function. If both inputs are low, the bistable keeps its state. The output `Q`

S	R	Q'
0	0	Q
1	0	1
0	1	0
1	1	-

is set if input `S` is high, and reset if input `R` is high. If both `S` and `R` are high, then `Q` is undefined. This can be modelled by a nondeterministic internal choice between high and low output. The latch is triggered by a negative edge of the clock pulse. That is, a change of output occurs only at the time instant when the clock line goes from high to low. If the value of the clock line is part of the state space, then the clock value would be low in every new state. For an accurate state-based model (e.g., of an asynchronous circuit), we would have to include timing information of all gates. However, if the clock is only used as trigger, an event based modelling is more adequate: The high-to-low change of the clock line is considered as an event occurrence. In each state, this event may or may not occur. To prevent executions in which the input or output clocks are indefinitely blocked, we require infinitely many input and output clock ticks in every infinite run.

The model is just a representation of the circuit's truth table, where the outputs are a boolean function of inputs and latch states. It can be derived automatically from any standard hardware description language; in fact, several model checkers

```
MODULE main
VAR Q, bus:  array 1..n of boolean;  -- n SR-latches, n databits
    inp, mc, pc, sc, oc:  boolean;   -- input lines
DEFINE out := Q[1]; ic := ((mc & pc)  |  (!mc & sc));
       A[i] := mc & pc & bus[i]; B[i] := !mc & Q[i + 1];
       R[i] := !(A[i] | B[i]); S[i] := !R[i];
ASSIGN next(Q[i]) := case ic:  case
                                !S[i] & !R[i]:  Q[i];        --hold
                                 S[i] & !R[i]:  1;           --set
                                !S[i] &  R[i]:  0;           --reset
                                 S[i] &  R[i]:  {0,1}; esac; --undef
                           !ic:  Q[i]; esac; -- unchanged if no input
       next(bus[i]) := case oc:  Q[i];  !oc:  {0, 1}; esac;
FAIRNESS ic FAIRNESS oc
```

Figure 6: Model of shift register

support such front-end translations. Correctness of parallel and sequential input is expressed by the following formulas:

$$\mathbf{A}\ \mathbf{G}^{*}\ (\mathtt{mc} \wedge \mathtt{pc} \rightarrow (\mathtt{bus[i]} \leftrightarrow \mathbf{A}((\mathtt{oc} \rightarrow \mathbf{A}\ \mathbf{X}\ \mathtt{bus[i]})\ \mathbf{U}^{+}\ \mathtt{ic})))$$

$$\mathbf{A}\ \mathbf{G}^{*}\ (\neg\mathtt{mc} \wedge \mathtt{sc} \rightarrow (\mathtt{Q[i]} \leftrightarrow \mathbf{A}(\mathtt{Q[i-1]}\ \mathbf{U}^{+}\ \mathtt{ic})))$$

Intuitively, these formulas assure that data which is input into the register remains there until a new input occurs. With appropriate ordering on the BDD variables, the model checker verifies these formulas for a bus width of 32 bit in less than a second. Similar formulas can be used to verify that after a sequence of n sequential load operations, the correct data word will be put onto the bus on a subsequent output pulse.

If the connection structure of wires within the circuit is "well-behaved", then automatic verification is successful even on much bigger circuits. A circuit is "well-behaved" if there exists an ordering of all wires such that the value of a wire only depends on the value of wires which are close in the ordering. A large number of circuits with hundreds of storage places have been verified automatically in this way.