Verification of Finite State Systems with Temporal Logic Model Checking

Bernd–Holger Schlingloff
Institut für Informatik der Technischen Universität München, Germany
schlingl@informatik.tu-muenchen.de

Abstract

These tutorial notes contain an introduction to the logical theory and computational aspects of computer aided verification of finite state reactive systems with linear and branching temporal logic model checking. As a general recipe, computer science applications and algorithms are derived from logical notions and proofs. First, the expressivity of various temporal logics is compared to first and second order logic, and to ω-automata and formal languages. Then, temporal safety and liveness properties are reviewed. From the completeness proof for natural and tree models, local and global decision procedures are developed. These in turn give rise to the corresponding model checking procedures. Various modelling techniques for reactive systems are presented. Finally, symbolic techniques for global model checking with binary decision diagrams, and partial order techniques for local model checking with stubborn sets are discussed.
Keywords: *Computer Aided Verification, Finite State Systems, Temporal Logic, Model Checking, Modal Logic, Expressivity, Expressiveness, ω-Languages, Büchi-Automata, Completeness, Decision Procedures, Tableaus, Specification, Binary Decision Diagrams (BDDs), Partial Order Methods, Stubborn Sets*
Computing Review Categories:

1 Introduction

A *reactive system* consists of several components which are designed for continuous interaction with one another and with the system's environment. A *property* is a set of desired behaviours which the system is supposed to show.

Formal semantics treats such systems as *mathematical structures* (e.g., as functions, relations, streams etc.), and properties as *mathematical theorems* (e.g., finiteness, injectivity, monotonicity etc.) about these structures. From a logical viewpoint, a system is a semantical *model*, and a property is a logical *formula*. Arguing about system correctness therefore amounts to *verifying formulas in models*.

To be able to prove that a system satisfies a property, one needs a *modelling language* to describe the system, a *logical language* in which the property can be formulated, and a *deductive calculus* and *algorithms* with which verification can be performed. Examples for modelling languages include process calculi (e.g. CCS and CSP), state machine description languages (e.g. StateCharts), protocol specification languages (e.g., Lotos or Esterelle), and simple shared variables programming languages (C, Basic, ...). Logical languages which are considered in these notes are e.g. first and second order logic, and propositional modal and temporal logic. For verifying whether a model satisfies a formula we derive *model checking algorithms* based on *canonical Hilbert axioms* and *tableau decision procedures*.

Interactive vs. Automatic Verification
Verification algorithms can be classified as *interactive* or *automatic*. Interactive methods are more general but harder to use; automatic methods have a limited range but are more likely to be accepted.

In interactive verification, the user provides the overall proof strategy; the machine augments the user by

- checking the correctness of each step,
- maintaining a list of assumptions and subgoals,
- applying the rules and substitutions which the user indicates, and by
- searching for applicable transformation rules and assumptions.

Sophisticated tools also are able to prove certain easy lemmas by themselves, usually by applying a heuristic search. However, since most interactive provers are designed for undecidable languages (e.g., first or higher order logic), the proof process is never completely automatic. User interaction is required, e.g., to find certain loop invariants or inductive hypothesis, and it requires a highly skilled experienced user to perform a nontrivial proof.

On the other side, with automatic verification all the user has to do is to provide a formal model of the *system* he or she wants to verify, and a formulation of the *properties* he or she thinks that this system satisfies. The verification tool then checks these properties, and either succeeds in finding that they are valid, or giving an indication why the properties fail to hold in the specific system.

Finite State Systems
The completely automatic approach generally requires that the automatic verifier traverses *all* states of the given system, which is only possible if the state space is finite.

Because of the undecidability of the halting problem, programs operating on natural or real data values can never be verified fully automatic. But, this is not quite correct:

- All physical machines have a limited memory, and op-

erate with bounded-width arithmetic. Thus, natural and real values are not machine representable, and on a "deeper" level, all programs are finite-state. (On a yet "deeper" level, when machine cycles are broken into voltage increases, there might again be an infinite number of states. However, on the next level, there is a finite number of electrons, and ultimatively the question whether the real world is finite state or not seems to be beyond our imagination.)

- In many cases it is possible to separate between the control structure and the data structure of a given program. Branches **if** c **then goto** a **else goto** b **fi**, where c depends on data from an infinite domain, can be abstracted as nondeterministic choice **goto** a **or goto** b. In a network communications protocol, often the correctness of the protocol does not depend on the specific type (finite or infinite) of the transmitted data; thus data values can be abstracted as tokens from a finite (or even binary) domain.
- Several interesting systems actually *are* finite state, even though the systems designer used infinite domains. E.g., a **boolean** C++ variable can only take the values 0 and 1 even though it is declared **integer**. Of course, proving that a system is finite state can be difficult.
- Even if the program inherently uses infinite domains, errors are likely to be preserved when these infinite domains are replaced by appropriate finite domains. E.g., a faulty sorting algorithm for natural numbers is almost certainly also incorrect for sorting values, say, between 0 and 10. Therefore, even if no *proof* can be automatically obtained, automatic verification tools can be used to *debug* the system.

Given a formal model of the system to be verified, and a formulation of the properties the system should satisfy, there are three possible results which an automated model checker can produce:

1. either it finds a *proof* for the formula in the model and outputs "verified", or
2. it constructs a *refutation*, i.e., an execution of the (model of the) system which dissatisfies the (formulation of the) property, or
3. the complexity of the verification procedure exceeds the given memory limit or time bound.

If there is not sufficient space or time, we can increase these resources, provided there is enough money available. Alternatively, we can use a coarser abstraction of the system and its properties. The third possibility is to employ heuristics which improve the performance of the verifier. Some of these heuristics are discussed in sections 9 and 10.

In some sense it is more interesting to get a refutation than to get a proof. Whenever we get a refutation, we can decide whether it is due to our abstraction in modelling and formulation, or whether this undesired sequence of events could indeed happen in reality. In the former case, the unrealistic behaviour can be eliminated by additional assumptions on the model or formula. In the latter case, we have found a bug, and something should be changed in the system.

If the model checker is able to prove a formula for a given model, there is nothing left to do. But, what has been achived in this case?

The Role of Formal Methods in Engineering

There can never be any guarantee that a system which has been verified by a computer aided tool will function correctly in reality. Even if we could assume that the verifiers hard- and software is correct (which we can not), there is a fundamental source of inaccuracy involved. Verification proves theorems about models of systems and formulations of properties, not about physical systems and desired behaviour; we can never know to which extent our models and formulations reflect physical reality and intuitions.

Nobody can ever guarantee that a physical system will not fail in unexpected (i.e., unmodeled) situations. It would be stupid, however, to reject formal methods because they cannot offer such guarantees. Civil engineering can never *prove* that a certain building will not collapse. Nevertheless it uses mathematical models to calculate loads and wall thicknesses and so on.

Similarly, we can never *prove* that our model adequately represents the reality. Therefore we can never *prove* that a system will function as planned. Nevertheless, computer aided verification can help *locating errors* already during the design of a complex system, and it can help to *increase reliability* of these systems. In the future, formal verification will augment classical software design tools like structured analysis, code review and automated testing.

2 Logical Languages, Expressiveness

Propositional Logic

Given a set $\mathcal{P} = \{\mathtt{p}, \mathtt{q}, \mathtt{p}_1, \ldots\}$ of (atomic) propositions which can be either true or false. E.g., proposition `sun_is_shining` denotes the fact that "the sun is shining".

Propositional logic **PL** is built from $\mathcal{P}$ with the following syntax:

$$\mathbf{PL} \quad ::= \quad \mathcal{P} \quad | \quad \bot \quad | \quad (\mathbf{PL} \to \mathbf{PL})$$

That is,

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

Note that $\mathcal{P}$ is a parameter of the logic! The special case $\mathcal{P} = \emptyset$ is allowed. Other connectives can be defined as usual:

- $\neg\varphi \triangleq (\varphi \to \bot)$
- $\top \triangleq \neg\bot$
- $(\varphi \vee \psi) \triangleq (\neg\varphi \to \psi)$
- $(\varphi \wedge \psi) \triangleq \neg(\neg\varphi \vee \neg\psi)$
- $(\varphi \leftrightarrow \psi) \triangleq ((\varphi \to \psi) \wedge (\psi \to \varphi))$

Semantics: Assume an *interpretation* $\mathcal{I}$ of propositions is given, which assigns a truth value $\{\mathbf{true}, \mathbf{false}\}$ to every proposition. (E.g., proposition `sun_going_down` is interpreted differently at the shore, in front of a computer terminal, or at the stock exchange :)

$\mathcal{I}$ can be extended to the set of all propositional formulas as follows:

- $\mathcal{I}(\bot) = \mathbf{false}$, and
- $\mathcal{I}((\varphi \to \psi)) = \mathbf{true}$ iff
  $\mathcal{I}(\varphi) = \mathbf{true}$ implies $\mathcal{I}(\psi) = \mathbf{true}$.

We write $\mathcal{I} \models \varphi$ iff $\mathcal{I}(\varphi) = \mathbf{true}$. Thus the above is equivalent to:

- $\mathcal{I} \not\models \bot$, and
- $\mathcal{I} \models (\varphi \to \psi)$ iff $\mathcal{I} \models \varphi$ implies $\mathcal{I} \models \psi$.

Propositional logic is not well-suited to formalise statements about events in time. Even though the interpretation of a statement can be fixed, its truth value may vary in time.

First Order Logic

To express temporal dependencies, the set $\mathcal{P}$ is used as a set of *monadic predicates*, i.e., each $\mathtt{p} \in \mathcal{P}$ is augmented with an additional parameter denoting time, for example, $\mathtt{sun\_is\_shining}(t)$.

What kind of language elements should be permitted? For sake of simplicity, we do not include function symbols (or constants) in the first order language. Assume in addition to the set $\mathcal{P}$ of unary predicates a fixed set $\mathcal{R} \triangleq \{R, a, b, \ldots\}$ of *accessibility relations* or *actions*, and let $\mathcal{R}^+ \triangleq \mathcal{R} \cup \{<, =\}$. Furthermore, let $\mathcal{T}$ be a set of *first order variables* $\mathcal{T} \triangleq \{t, t_0, \ldots\}$ for moments in time (which is assumed to be infinite unless stated otherwise).

$$\mathbf{FOL} \quad ::= \quad \mathcal{P}(\mathcal{T}) \mid \bot \mid (\mathbf{FOL} \to \mathbf{FOL}) \mid \mathcal{R}^+(\mathcal{T}, \mathcal{T}) \mid \exists \mathcal{T}\ \mathbf{FOL}$$

Again, we do not assume $\mathcal{P}$ or $\mathcal{R}$ to be nonempty. $\forall t\ \varphi$ is short for $\neg \exists t\ \neg \varphi$. Relations are denoted in infix notation as well: $t_1 R t_2 \triangleq R(t_1, t_2)$.

Semantics: To assign a truth value to a formula containing (free) variables, we assume that we are given a nonempty universe U of time-points, and that the *interpretation* $\mathcal{I}$ assigns to every proposition $\mathtt{p} \in \mathcal{P}$ a subset of points $\mathcal{I}(\mathtt{p}) \subseteq U$, and to every action $R \in \mathcal{R}$ a binary relation $\mathcal{I}(R) \subseteq U \times U$. For the special relation signs $<$ and $=$ we require that $\mathcal{I}(=) \triangleq \{(w, w) \mid w \in U\}$, and $\mathcal{I}(<)$ is the transitive closure of $\bigcup\{\mathcal{I}(R) \mid R \in \mathcal{R}\}$. A *variable valuation* $\mathbf{v}$ assigns to any variable $t \in \mathcal{T}$ a time-point $w \in U$. A first-order model $\mathcal{M} \triangleq (U, \mathcal{I}, \mathbf{v})$ consists of a universe U, an interpretation $\mathcal{I}$, and a variable valuation $\mathbf{v}$. Similar to above, we define when a formula holds in a model:

- $\mathcal{M} \models \mathtt{p}(t)$ iff $\mathbf{v}(t) \in \mathcal{I}(\mathtt{p})$;
- $\mathcal{M} \not\models \bot$, and
- $\mathcal{M} \models (\varphi \to \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$;
- $\mathcal{M} \models R(t_0, t_1)$ iff $(\mathbf{v}(t_0), \mathbf{v}(t_1)) \in \mathcal{I}(R)$;
- $\mathcal{M} \models \exists t\ \varphi$ iff $(U, \mathcal{I}, \mathbf{v}') \models \varphi$ for some $\mathbf{v}'$ which differs from $\mathbf{v}$ at most in t.

This language is rather expressive:

(1) $\mathtt{nil}(t_0) \to \exists t_1 (put(t_0, t_1) \wedge \neg \mathtt{nil}(t_1))$
If `nil` holds, then it is possible to perform a put and become not `nil`.

(2) $\forall t_1 ((t_0 < t_1 \wedge \mathtt{req}(t_1)) \to \exists t_2 (t_1 < t_2 \wedge \mathtt{ack}(t_2)))$
Every request is eventually acknowledged.

(3) $\forall t_1 ((t_0 < t_1 \wedge \mathtt{req}(t_1)) \to \exists t_2 ((t_1 < t_2 \wedge \mathtt{ack}(t_2)) \wedge \forall t_3 ((t_1 < t_3 \wedge t_3 < t_2) \to \mathtt{req}(t_3))))$
No request is withdrawn before it is acknowledged.

But is first-order logic intuitive?

Multimodal Logic

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 alternative state of affairs. Temporal phrases in natural language use the adverbs "eventually" and "constantly" (or "sometime" and "always") to refer to future time points. Modal logic was invented to formalise these modal and temporal adverbs: the idea is to suppress first order variables $t \in \mathcal{T}$; propositions $\mathtt{p} \in \mathcal{P}$ are again without argument. The meaning of a proposition like `sun_is_shining` is "the sun is shining *now*". Thus, in a temporal interpretation, every formula describes a certain state of affairs *at a given moment*.

What about accessibility relations? For every $R \in \mathcal{R}$, multimodal logic introduces a new operator $\langle R \rangle \varphi$ meaning "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".

$$\mathbf{ML} \quad ::= \quad \mathcal{P} \mid \bot \mid (\mathbf{ML} \to \mathbf{ML}) \mid \langle \mathcal{R} \rangle \mathbf{ML}.$$

Intuitively, the above example (1) could be written

$$\mathtt{nil} \to \langle put \rangle \neg \mathtt{nil}.$$

A (Kripke-) *model* $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ for multimodal logic again consists of a universe U of points, an interpretation $\mathcal{I}$ assigning to every $\mathtt{p} \in \mathcal{P}$ and $R \in \mathcal{R}$ a subset $\mathcal{I}(\mathtt{p}) \subseteq U$ and a relation $\mathcal{I}(R) \subseteq U \times U$, respectively. Instead of a valuation for free variables, a model designates a *current* or *initial* point $w_0 \in U$.

- $\mathcal{M} \models \mathtt{p}$ 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 $w_1 \in U$ with $(w_0, w_1) \in \mathcal{I}(R)$ and $(U, \mathcal{I}, w_1) \models \varphi$.

A formula φ is *universally* valid in $(U, \mathcal{I})$, if for all $w \in U$ it holds that $(U, \mathcal{I}, w) \models \varphi$.

By definition, multimodal logic has no operator for $<$, the transitive closure relation of all actions. Define $\mathcal{M} \models \Diamond \varphi$ iff there exists $w_1 > w_0$ such that $(U, \mathcal{I}, w_1) \models \varphi$; that is, $\Diamond \triangleq \langle < \rangle$, and $\Box \triangleq [<]$. With these operators, example (2) could be written

$$\Box(\mathtt{req} \to \Diamond \mathtt{ack}).$$

Temporal Logic

However, modal operators cannot express statements about intervals (example (3) of the above). *Temporal logic* is based on a binary operator $\mathcal{U}(\varphi, \psi)$ meaning "until φ, also ψ holds".

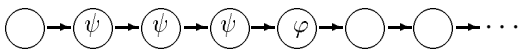
Syntax of linear temporal logic (**LTL**):

$$\textbf{LTL} \quad ::= \quad \mathcal{P} \mid \bot \mid (\textbf{LTL} \to \textbf{LTL}) \mid \langle\mathcal{R}\rangle\textbf{LTL} \mid \mathcal{U}(\textbf{LTL}, \textbf{LTL}).$$

The above definition of validity of a formula in a model is extended for formulas involving $\mathcal{U}$-operators as follows:

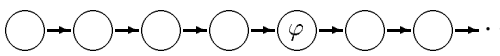
- $\mathcal{M} \models \mathcal{U}(\varphi, \psi)$ iff there exists $w_1 \in U$ with $w_0 < w_1$ and $(U, \mathcal{I}, w_1) \models \varphi$, and for all $w_2 \in U$ with $w_0 < w_2$ and $w_2 < w_1$, we have $(U, \mathcal{I}, w_2) \models \psi$.

  The following picture illustrates this situation:

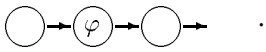


Various other operators can be expressed using $\mathcal{U}$:

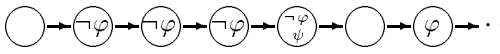
- $\Diamond\varphi \triangleq \mathcal{U}(\varphi, \top)$ (*Sometime*)



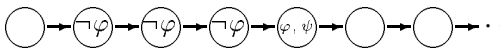
- $\circ\varphi \triangleq \mathcal{U}(\varphi, \bot)$ (*Nexttime*)



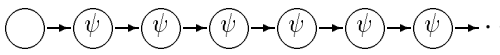
- $\mathcal{B}(\varphi, \psi) \triangleq \mathcal{U}(\neg\varphi \wedge \psi, \neg\varphi)$ (*Before*)



- $\mathcal{A}(\varphi, \psi) \triangleq \mathcal{U}(\varphi \wedge \psi, \neg\varphi)$ (*Atnext*)



- $\mathcal{V}(\varphi, \psi) \triangleq \neg\mathcal{U}(\neg\psi, \varphi)$ (*Unless*)



Note: these definitions are made up with a linear, irreflexive and discrete $<$–relation in mind. But, this is not required so far.

As an example, the above formula (3) can be expressed as

$$\Box(\texttt{req} \to \mathcal{U}(\texttt{ack}, \texttt{req})).$$

How can we compare first order and temporal logic? First order logic formulas can mention several reference points (free variables), which is not possible in modal or temporal logic. To be able to compare the expressiveness of both logics, we restrict **FOL** to formulas with at most one free variable.

Moreover, first order logic can use reverse relations: $x > y$ iff $y < x$. Therefore, we introduce a temporal *past* operator $\mathcal{S}$ (since), with the following semantics:

- $\mathcal{M} \models \mathcal{S}(\varphi, \psi)$ iff $\exists w_1(w_1 < w_0 \wedge \varphi(w_1) \wedge \forall w_2(w_1 < w_2 < w_0 \to \psi(w_2)))$.

We write $\Diamond^-\varphi$ and $\Box^-\varphi$ for $\mathcal{S}(\varphi, \top)$ and $\neg\Diamond^-\neg\varphi$, respectively. Intuitively, these operators refer to "sometime in the past" and "always in the past".

The semantic clause induces a translation from temporal to first order logic.

E.g., $\mathcal{U}(\texttt{ack}, \mathcal{S}(\texttt{req}, \neg\texttt{ack}))$ is translated as

$$\exists t_1(t_0 < t_1 \wedge \texttt{ack}(t_1) \wedge \forall t_2(t_0 < t_2 < t_1 \to \exists t_3(t_3 < t_2 \wedge \texttt{req}(t_3) \wedge \forall t_4(t_3 < t_4 < t_2 \to \neg\texttt{ack}(t_4))))).$$

Hence, **FOL** is at least as expressive as **LTL**. Formally, this means: for every $\varphi \in$ **LTL** (with or without past operators) there exists a formula $\varphi^\tau \in$ **FOL** (with exactly one free variable t_0), such that for every model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ and valuation $\mathbf{v}$ for which $\mathbf{v}(t_0) = w_0$ we have $(U, \mathcal{I}, w_0) \models \varphi$ iff $(U, \mathcal{I}, \mathbf{v}) \models \varphi^\tau$.

A logic is called *expressive* (or *expressively complete*), if there exists also a translation in the other direction: given any first-order formula, does an equivalent temporal formula exist?

Expressive Completeness of Temporal Logic

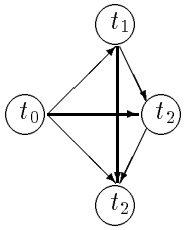
Observation: Only 3 variables are really needed in the translation of any given temporal formula; other variables can be "reused".

E.g., $\mathcal{U}(\texttt{ack}, \mathcal{S}(\texttt{req}, \neg\texttt{ack}))$ can be translated as

$$\exists t_1(t_0 < t_1 \wedge \texttt{ack}(t_1) \wedge \forall t_2(t_0 < t_2 < t_1 \to \exists t_0(t_0 < t_2 \wedge \texttt{req}(t_0) \wedge \forall t_1(t_0 < t_1 < t_2 \to \neg\texttt{ack}(t_1))))).$$

As an immediate consequence, **LTL** cannot express properties which "inherently" use four variables, like, e.g., the statement "there are three different connected points reachable from the current point",

$$\exists t_1, t_2, t_3(t_0 < t_1 \wedge t_0 < t_2 \wedge t_0 < t_3 \wedge t_1 < t_2 \wedge t_1 < t_3 \wedge t_2 < t_3)$$



In case that $<$ is a linear order (antisymmetric and total) this is equivalent to

$$\exists t_1(t_0 < t_1 \wedge \exists t_2(t_1 < t_2 \wedge \exists t_3(t_2 < t_3)))$$
$$\exists t_1(t_0 < t_1 \wedge \exists t_2(t_1 < t_2 \wedge \exists t_0(t_2 < t_0)))$$

which can be expressed temporally as $\Diamond\Diamond\Diamond\top$.

Therefore, attention is restricted to certain classes of structures, like e.g. Dedekind-complete linear orders, or finitely-branching trees, and others.

A *path* in a model is a (finite or infinite) sequence of points and actions $w_0 R_0 w_1 R_1 \ldots$, where $(w_i, w_{i+1}) \in \mathcal{I}(R_i)$ for each i. A path is *maximal*, if each of its points which has a successor in the model also has a successor in the path; that is, a maximal path is either infinite, or for its final point w_n there is no w such that $w_n < w$. *Natural*

models are Kripke models for which there exists exactly one maximal path, and every point $w \in U$ is contained exactly once in this path. That is, in natural models $(U, <)$ is isomorphic to the natural numbers or an initial segment of the natural numbers, with its usual ordering. Additionally, for every point $w \in U$ there exists at most one $R \in \mathcal{R}$ and $w' \in U$ such that $(w, w') \in \mathcal{I}(R)$.

Theorem(Kamp, Gabbay): Temporal logic is expressive for natural models.

The proof follows [6] and uses a certain property called *separation*. Call a temporal formula *pure future*, if it is of form $\langle R\rangle\varphi$ or $\mathcal{U}(\varphi, \psi)$, where in both φ and ψ no $\mathcal{S}$-operator occurs, and *pure past*, if it is of form $\mathcal{S}(\varphi, \psi)$, where in both φ and ψ no $\mathcal{U}$-operator occurs, and *pure present*, if it contains no $\mathcal{U}$ or $\mathcal{S}$-operators.

A formula is *separated* if it is a boolean combination of pure future, pure present and pure past formulas. A logic has the *separation property* (for a given class of models), if for every formula there exists an separated formula which is equivalent for all models under consideration.

Lemma: The separation property implies expressive completeness.

Proof is by induction on the structure of those **FOL**-formulas which have exactly one free variable t_0. Translation of $\mathtt{p}(t_0)$ is $\mathtt{p}$, and boolean connectives are immediate. We don't have to consider formulas like $t_0 < t_1$ by themselves, since they involve two free variables. The only remaining case are formulas $\varphi = \exists t_1 \psi(t_0, t_1)$. Without loss of generality we can assume that ψ does not contain any $\mathtt{p}(t_0)$. That is, $\psi(t_0, t_1)$ is a boolean combination of formulas $\mathtt{p}(t_1)$, $t_0 R t_1$, $t_1 R t_0$, $t_1 < t_0$, $t_1 = t_0$, $t_0 < t_1$, and φ', where in $\varphi' \triangleq \exists t_2 \psi'(t_0, t_1, t_2)$ again we can assume that ψ' does not contain any $\mathtt{p}(t_0)$ or $\mathtt{p}(t_1)$. Replace in φ every $t_0 R t$ by a new unary proposition $X_R(t)$, replace every $t R t_0$ by a new unary $Y_R(t)$, replace $t_0 < t$ by $F(t)$, replace $t_0 = t$ by $N(t)$, and replace $t < t_0$ by $P(t)$. That is, φ now does not contain any t_0. Since the nesting depth of existential quantifiers in $\varphi' = \exists t_2 \psi'(t_1, t_2)$ is smaller than that of φ, we can apply the induction hypothesis to get a temporal formula $\tau(\varphi')$. Reinserting this into ψ and replacing $\mathtt{p}(t_1)$ by $\mathtt{p}$, and $X_R(t_1)$ by X_R and $Y_R(t_1)$ by Y_R, and $F(t_1)$ by F, and $N(t_1)$ by N, and $P(t_1)$ by P gives $\tau(\psi)$. Now separate the formula $\Diamond\tau(\psi) \vee \tau(\psi) \vee \Diamond^- \tau(\psi)$. The resulting formula is a boolean combination of pure future, pure past and pure present formulas. Replace in this formula every F inside a pure future formula by $\top$, every other F by $\bot$, and similar for X_R, Y_R, N and P. The resulting formula is the required translation $\tau(\varphi)$.

To illustrate this construction, let us find the temporal equivalent of $\varphi = \exists t_1(t_0 < t_1 \wedge \mathtt{p}(t_1) \wedge \forall t_2(t_0 < t_2 < t_1 \rightarrow \mathtt{q}(t_2)))$. (We already know that the outcome should be $\mathcal{U}(\mathtt{p}, \mathtt{q})$!) The first replacement results in $\exists t_1(F(t_1) \wedge \mathtt{p}(t_1) \wedge \neg\exists t_2(F(t_2) \wedge t_2 < t_1 \wedge \neg\mathtt{q}(t_2)))$. The formula $\varphi'(t_1) = \exists t_2(t_2 < t_1 \wedge F(t_2) \wedge \neg\mathtt{q}(t_2)))$ inductively translates to $\tau(\varphi') = \Diamond^-(F \wedge \neg\mathtt{q})$. Separating $\Diamond(F \wedge \mathtt{p} \wedge \Box^-(F \rightarrow \mathtt{q}))$ gives $\Box^-(F \rightarrow \mathtt{q}) \wedge (F \rightarrow \mathtt{q}) \wedge \mathcal{U}((F \wedge \mathtt{p}), (F \rightarrow \mathtt{q}))$ (see below). With the final replacement, the disjuncts $\Diamond^-\tau(\psi)$ and $\tau(\psi)$ reduce to $\bot$, and $\Box^-(F \rightarrow \mathtt{q}) \wedge (F \rightarrow \mathtt{q})$ reduces to $\top$, and $\mathcal{U}((F \wedge \mathtt{p}), (F \rightarrow \mathtt{q}))$ reduces to $\mathcal{U}(\mathtt{p}, \mathtt{q})$, so that we are left with the desired result.

The above proof could be slightly simplified if multimodal diamond operators were encoded as special propositions: In natural models, $\langle R\rangle\varphi \leftrightarrow \circ\varphi \wedge \mathtt{p}_R$, where $\mathtt{p}_R \triangleq \langle R\rangle\top$. A similar encoding will be used in sections 3 and 4.

To show expressive completeness, it remains to prove that **LTL** has the separation property for natural models. This can be done by systematically considering equivalences like

$\Diamond(\varphi \wedge \Box^-\psi) \leftrightarrow \Box^-(\psi) \wedge \psi \wedge \mathcal{U}(\varphi, \psi)$.
$\Diamond(\varphi \wedge \Diamond^-\psi) \leftrightarrow \Diamond(\psi \wedge \Diamond\varphi) \vee (\psi \wedge \Diamond\varphi) \vee (\Diamond^-\psi \wedge \Diamond\varphi)$.
$\mathcal{U}(\varphi, \psi \wedge \mathcal{S}(\varphi_1, \psi_1)) \leftrightarrow$
$\circ\varphi \vee \mathcal{U}(\circ\varphi, \psi \wedge (\varphi_1 \vee \psi_1) \wedge (\varphi_1 \vee \psi_1 \wedge \mathcal{S}(\varphi_1, \psi_1))$.

To complete this proof, the reader has to go through the tedious exercise of giving such clauses for all possible occurrences of a since-operator in a propositional formula inside an until-operator.

3 Second Order Languages

Linear and Branching time Logics

As we have seen, linear temporal logic is expressive for natural models. The same result can be proved (with minor modifications) for finitely branching trees. In computer science, the set of executions of a program can be modelled as a set of execution sequences or as an execution tree, where branches denote nondeterministic decisions.

Statements about correctness of program can involve talking about *all maximal paths* in a tree.

Syntax of computation tree logic (**CTL**):

CTL $::= \mathcal{P} \mid \bot \mid (\mathbf{CTL} \rightarrow \mathbf{CTL}) \mid \langle R\rangle\mathbf{CTL} \mid E\mathcal{U}(\mathbf{CTL}, \mathbf{CTL}) \mid A\mathcal{U}(\mathbf{CTL}, \mathbf{CTL})$.

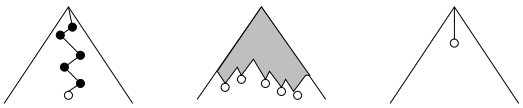
This definition is slightly nonstandard: to give a uniform treatment for modal, linear and branching temporal logic we include $\langle R\rangle$-operators referring to the action, i.e. labelling of a transition.

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 symbol $<$ then denotes the usual tree-order: $w_1 < w_2$ iff w_1 is on the (unique) path from the root w_0 up to w_2.

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

The following operators are defined as abbreviations:

- $EX\varphi \triangleq E\mathcal{U}(\varphi, \bot)$
- $AX\varphi \triangleq \neg EX\neg\varphi$
- $EF\varphi \triangleq E\mathcal{U}(\varphi, \top)$
- $AG\varphi \triangleq \neg EF\neg\varphi$
- $AF\varphi \triangleq A\mathcal{U}(\varphi, \top)$
- $EG\varphi \triangleq \neg AF\neg\varphi$



$E\mathcal{U}(\psi_1,\psi_2)$ $A\mathcal{U}(\psi_1,\psi_2)$ $EX\psi$

How can we compare the expressivity of **CTL** with (the future fragment of) **LTL**? Direct comparison is difficult, since models are different: On natural models, $A\mathcal{U}$ and $E\mathcal{U}$-operators coincide; on tree models, **LTL** trivially cannot express $A\mathcal{U}$.

Therefore, one considers (nonlinear, nontree) *Kripke-models* $(U,\mathcal{I},w_0)$. In contrast to natural or tree models, Kripke models can contain reflexive points, loops or even dense relations. On Kripke models, an **LTL**-formula is valid, if it is valid for all natural models *generated* from the Kripke model. (This definition revises the original definition given above!) Similarly, a **CTL**-formula is valid in a Kripke model, if it is valid for the unique maximal tree generated from it.

Now, the **LTL**-formula $\Diamond\Box p$ is not expressible in **CTL** (it is *not* the same property as $AFAGp$), and $AGEFp$ is not expressible in **LTL** (it is *not* the same as $\Box\Diamond p$); for more information on the expressiveness of linear versus branching time see [4].

On Kripke models, the more expressive logic **CTL*** subsumes **CTL** and **LTL** by separating path quantification (E) from temporal quantification ($\mathcal{U}$):

$$\textbf{CTL}^* \ ::= \ \mathcal{P} \mid \bot \mid (\textbf{CTL}^* \to \textbf{CTL}^*) \mid \langle R\rangle\textbf{CTL} \mid \mathcal{U}(\textbf{CTL}^*,\textbf{CTL}^*) \mid E\ \textbf{CTL}^*.$$

In the semantics, validity is declared relative to a current point w_0 *and current path* p_0:

- $(U,\mathcal{I},w_0,p_0)\models\mathcal{U}(\varphi,\psi)$ iff $\exists w_1(w_0<w_1\wedge p_0(w_1)\wedge(U,\mathcal{I},w_1,p_0)\models\varphi\wedge\forall w_2(w_0<w_2<w_1\to(U,\mathcal{I},w_2,p_0)\models\psi)))$.
- $(U,\mathcal{I},w_0,p_0)\models E\varphi$ iff $\exists$ path p_1 from w_0, $(U,\mathcal{I},w_0,p_1)\models\varphi$.

(Here $p_0(w_1)$ indicates that point w_1 is on path p_0.) A **CTL***-formula φ is defined to be valid for $(U,\mathcal{I},w_0)$, if it is valid for all models $(U,\mathcal{I},w_0,p_0)$, where p_0 is a path from w_0. Therefore, for Kripke models, **CTL*** is strictly more expressive than **CTL** and **LTL**.

Propositionally Quantified Logics

The second of the above clauses uses quantification over paths, which is not a first-order notion. For linear time, this is not very useful. But why should second-order quantification be restricted to paths? Wolper remarked that "temporal logic can be more expressive".

In **LTL** (or **FOL**), is it possible to specify that a certain property holds on every *second* point of an execution sequence?

$$\textit{even}\ \varphi \triangleq \varphi\wedge\Box(\varphi\to\circ\circ\varphi)$$
$$\textit{even}\ \varphi(t_0) \triangleq \varphi(t_0)\wedge\forall t>t_0(\varphi(t)\to\exists t_1,t_2(tSt_1St_2\wedge\varphi(t_2)))$$

Not correct: if φ holds twice in a row, it must hold always.

$$\textit{even}\ \varphi \triangleq (\Box((\mathtt{q}\leftrightarrow\circ\neg\mathtt{q})\wedge(\mathtt{q}\to\varphi))\wedge\mathtt{q})$$
$$\textit{even}\ \varphi(t_0) \triangleq (\forall t>t_0,tSt_1((\mathtt{q}(t)\leftrightarrow\neg\mathtt{q}(t_1))\wedge(\mathtt{q}(t)\to\varphi(t)))\wedge\mathtt{q}(t_0))$$

Almost correct: q is an auxiliary variable ("implementation detail").

$$\textit{even}\ \varphi \triangleq \exists q(\Box((q\leftrightarrow\circ\neg q)\wedge(q\to\varphi))\wedge q)$$
$$\textit{even}\ \varphi(t_0) \triangleq \exists q(\forall t>t_0,tSt_1((q(t)\leftrightarrow\neg q(t_1))\wedge(q(t)\to\varphi(t)))\wedge q(t_0))$$

That's it! The language used in the first item is called quantified temporal logic **qTL**, the language of the second item is *monadic second order logic* **MSOL**.

$$\textbf{qTL} \ ::= \ \mathcal{P} \mid \mathcal{Q} \mid \bot \mid (\textbf{qTL}\to\textbf{qTL}) \mid \langle\mathcal{R}\rangle\ \textbf{qTL} \mid \mathcal{U}(\textbf{qTL},\textbf{qTL}) \mid \exists\mathcal{Q}\ \textbf{qTL}.$$
$$\textbf{MSOL} \ ::= \ \mathcal{P}(\mathcal{T})\mid\mathcal{Q}(\mathcal{T})\mid\bot\mid(\textbf{MSOL}\to\textbf{MSOL})\mid \mathcal{R}^+(\mathcal{T},\mathcal{T}) \mid \exists\mathcal{T}\ \textbf{MSOL} \mid \exists\mathcal{Q}\ \textbf{MSOL}$$

To define this syntax, we used another syntactic category $\mathcal{Q}=\{q,q_0,\ldots\}$ of *proposition variables*. Any valuation in a model assigns a subset of U to each of these (second order) variables. The formula $\exists q\ \varphi$ is true in a model $\mathcal{M}$ if it is true in a model which differs from $\mathcal{M}$ at most in the valuation of the proposition variable q.

On natural models, the $\mathcal{U}$-operator in **qTL** is definable by $\Box$ and $\circ$ with the following clause:

$$\mathcal{U}(\varphi,\psi)\leftrightarrow\exists q(\circ q\wedge\Box(q\to(\varphi\vee\psi\wedge\circ q))\wedge\Diamond\neg q).$$

For complexity reasons, it is not always advisable to allow full monadic second order quantification (that is, quantifiers on arbitrary subsets of U). Therefore, we introduce *fixpoint quantification*: quantification on sets which can be obtained as fixed points of recursive definitions.

Examples:

$$\Box\varphi\leftrightarrow\circ(\varphi\wedge\Box\varphi)\leftrightarrow\circ(\varphi\wedge\circ(\varphi\wedge\Box\varphi))\leftrightarrow\cdots$$
$$\mathcal{U}(\varphi,\psi)\leftrightarrow\circ(\varphi\vee(\psi\wedge\mathcal{U}(\varphi,\psi)))\leftrightarrow\circ(\varphi\vee(\psi\wedge\circ(\varphi\vee(\psi\wedge\mathcal{U}(\varphi,\psi)))))$$
$$\Box\varphi\leftrightarrow\nu q\ \circ(\varphi\wedge q)$$
$$\mathcal{U}(\varphi,\psi)\leftrightarrow\mu q\ \circ(\varphi\vee(\psi\wedge q))$$

Syntax of the propositional μ-calculus μ**TL**:

$$\mu\textbf{TL} \ ::= \ \mathcal{P} \mid \mathcal{Q} \mid \bot \mid (\mu\textbf{TL}\to\mu\textbf{TL}) \mid \langle\mathcal{R}\rangle\ \mu\textbf{TL} \mid \nu\mathcal{Q}\ \mu\textbf{TL}.$$

The formula $\mu q\ \varphi(q)$ is short for $\neg\nu q\ \neg\varphi(\neg q)$. Semantics can again be defined by a translation into **qTL** (or **MSOL**):

- $\mathcal{M}\models\nu q\,\varphi$ iff $\exists q(q(w_0)\wedge\forall w_1(q(w_1)\to\varphi(w_1)))$.

Hence, μ**TL** is at most as expressive as **qTL** and **MSOL**.

Theorem (Büchi, Wolper): On natural models, μ**TL** is as expressive as **qTL** and **MSOL**.

For the proof of this theorem, we need the notion of ω-*regular languages* and ω-*automata*.

ω-automata and -languages

Any linear-time logic formula specifies the set of all natural models (execution sequences) in which it is valid. Consider an infinite sequence $(U=(w_0,w_1,w_2,\ldots),\mathcal{I},w_0)$, where $\mathcal{I}$ defines a mapping from $\mathcal{P}$ into subsets of U, and for a mapping from $\mathcal{R}$ into subsets of $U\times U$ such that for any i there is exactly one R such that $(w_i,w_{i+1})\in\mathcal{I}(R)$. Equivalently, we can introduce a *labelling function* $\mathcal{L}$ from U into subsets of $\mathcal{P}$:

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

Define a function w from the natural numbers into $2^{\mathcal{P}} \times \mathcal{R}$ by $w(i) \triangleq (\mathcal{L}(w_i), R_i)$, where R_i is the unique action such that $(w_i, w_{i+1}) \in \mathcal{I}(R_i)$. (For finite sequences $(w_0, ..., w_n)$, we can either leave the last action R_n arbitrary, or introduce a special null action. Details will not be elaborated here.) The function w is called an *ω-word* over the *alphabet* $2^{\mathcal{P}} \times \mathcal{R}$. Since a set of ($\omega$-)words is usually called an *(ω-)language*, every **LTL**-formula defines such a language.

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

(ω-)regular expressions are defined like usual regular expressions, with additional operation denoting infinite repetition of a subexpression:

- Every letter from the alphabet is an ω-regular expression (letters can be seen as words of length one, and words can be seen as languages with one element)
- If α and β are ω-regular, then so are $(\alpha; \beta)$, $(\alpha + \beta)$ and α^*.
- If α is ω-regular, then so is α^ω.

We use boolean expressions over $\mathcal{P} \cup \mathcal{R}$ to denote (unions of) letters; e.g. $(\neg \mathtt{p1} \vee \mathtt{p2})$ over $\mathcal{P} = \{\mathtt{p1}, \mathtt{p2}\}$ denotes $\{\} + \{\mathtt{p2}\} + \{\mathtt{p1}, \mathtt{p2}\}$.

Example for an ω-regular expression: $(\neg \mathtt{p1})^\omega + (\top^*; \mathtt{p2})^\omega$.

An *ω-automaton* over the alphabet $2^{\mathcal{P}} \times \mathcal{R}$ is defined like a usual finite automaton; it is a tuple $(S, \Delta, S_0, F_f, F_i)$, where

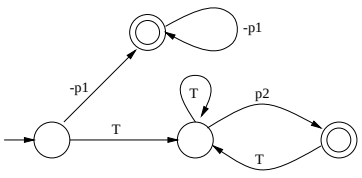
- S is a set of states,
- $\Delta \subseteq S \times 2^{\mathcal{P}} \times \mathcal{R} \times S$ is the transition relation,
- $S_0 \subseteq S$ is the set of starting states,
- $F_f \subseteq S$ is the acceptance set for finite words, and
- $F_i \subseteq S$ is the recurrence set for infinite words.

A *Büchi-automaton* is an ω-automaton where the set of states is finite.

When does such an automaton accept a word? A (finite or infinite) word $(w_0, w_1, ...)$ is accepted by the automaton, if there is a labelling ρ assigning to any point w_i a state $\rho(w_i)$ of the automaton such that

- $\rho(w_0) \in S_0$,
- For all $0 \leq i < n$, $(s_i, w_i, s_{i+1}) \in \Delta$, and
- $\rho(w_n) \in F_f$, if the word is finite with last letter w_n, and
- *inf*$(\rho) \cap F_i \neq \emptyset$, if it is infinite, where *inf*(ρ) is the set of states that appear infinitely often in the range of ρ. That is, at least one state from the recurrence set must be selected infinitely often.

For alternative acceptance conditions, see [11]. As an example of a Büchi-automaton, consider the following:



This automaton accepts (i.e., defines) exactly the same language as the example ω-regular expression above. In general, Büchi-automata can define all and only ω-regular languages. The proof of this statement is similar as for automata on finite words: For one direction, we have to show that Büchi-automata are closed under concatenation, union, Kleene-star and ω-repetition. All of these constructions are straightforward extensions of the appropriate constructions for automata on finite words.

The other direction is to show that the Büchi acceptance condition can be captured by an appropriate regular expression. Let L_{ij} be the regular language of finite words sending an automaton from state s_i into state s_j. Then the ω-regular language associated with any Büchi-automaton is

$$\Sigma_{w_0 \in S_0,\ w_f \in F_f} L_{0f} + \Sigma_{w_0 \in S_0,\ w_i \in F_i} L_{0i}; (L_{ii})^\omega$$

Trivially, ω-automata are closed under projection onto a smaller alphabet. Büchi 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.

Automata and Logics

Closure under complement can be used to show that Büchi-automata are at least as expressive as **qTL**: Automata for propositions are trivial two-state machines; from an automaton for φ, an automaton for $\langle R \rangle \varphi$, $\circ\varphi$ and $\Diamond\varphi$ can be built by an appropriate prefixing with a single step or loop on the initial states; as already mentioned, $\mathcal{U}$ can be expressed with $\circ$, $\Diamond$ and second order quantification; implications $(\varphi \rightarrow \psi)$ can be written as $(\neg\varphi \vee \psi)$ and thus be reduced to unions and complements; and existential second order quantification amounts to the projection of the automaton onto a smaller alphabet.

To close the circle, we show that for every ω-regular expression there exists a μ**TL**-formula describing the same language. This proof associates with every ω-regular expression φ a μ**TL**-formula $\mu\mathbf{TL}(\varphi, q)$ with at most one free proposition variable q indicating the end of the sequence:

- For $\varphi \in 2^{\mathcal{P}} \times \mathcal{R}$, say $\varphi = (P, R)$, define $\mu\mathbf{TL}(\varphi, q) \triangleq (\bigwedge_{\mathtt{p} \in P} \mathtt{p} \wedge \bigwedge_{\mathtt{p} \notin P} \neg\mathtt{p} \wedge \langle R \rangle q)$
- If $\mu\mathbf{TL}(\varphi, q)$ and $\mu\mathbf{TL}(\psi, q)$ are given (with the same free q), then $\mu\mathbf{TL}(\varphi + \psi, q) \triangleq \mu\mathbf{TL}(\varphi, q) \vee \mu\mathbf{TL}(\psi, q)$.
- If $\mu\mathbf{TL}(\varphi, q_1)$ and $\mu\mathbf{TL}(\psi, q_2)$ are given (with $q_1 \neq q_2$), then $\mu\mathbf{TL}(\varphi; \psi, q_2) \triangleq \mu\mathbf{TL}(\varphi[q_1 / \mu\mathbf{TL}(\psi, q_2)])$
- If $\mu\mathbf{TL}(\varphi, q_1)$ is given with $q_2 \notin \mu\mathbf{TL}(\varphi, q_1)$, then $\mu\mathbf{TL}(\varphi^*, q_2) \triangleq (\mu q_1 (\mu\mathbf{TL}(\varphi, q_1) \vee q_2))$
- Finally, $\mu\mathbf{TL}(\varphi^\omega, q) \triangleq (\nu q\ \mu\mathbf{TL}(\varphi, q))$.

If there is a free proposition variable q in the result, it can be eliminated by an empty quantification νq.

As an example for this construction, consider

$$\mu\mathbf{TL}((\neg\mathtt{p})^\omega + (\top^*; \mathtt{q})^\omega)$$
$$= \nu q_1 (\neg\mathtt{p} \wedge \circ q_1) \vee \nu q_3 \mu q_2 ((\top \wedge \circ q_2) \vee (\mathtt{q} \wedge \circ q_3))$$

Thus we have shown: For every μ**TL**-formula there exists an equivalent **qTL**-formula by definition; on natural models **qTL** is equal in expressiveness to **MSOL** by (an obvious extension of) Kamp's theorem; for every **qTL** (or **MSOL**) formula there is a Büchi-automaton defining the set of its models; Büchi-automata are equivalent to ω-regular expressions; and these in turn can be described by

μ**TL**-formulas.

$$\mu\mathbf{TL} = \mathbf{qTL} = \mathbf{MSOL} = \omega\text{-}automata = \omega\text{-}regular$$

Similar results can be proved about branching time logic (μ**TL**/**qTL** on tree models) and tree automata, which accept computation trees ($\Delta \subseteq S \times 2^{\mathcal{P}} \times (\mathcal{R} \times S)^n$) (see [11]).

4 Transition Systems and Properties

As we have seen, linear temporal formulas and ω-automata both can be used to describe sets of infinite sequences. The practical difference is, that logic tends to be more "descriptive", describing *what* a system should do, whereas automata tend to be more "machine-oriented", describing *how* it should be done. Logical formulas are "global", they are interpreted on the whole structure, whereas automata are "local", describing single states and transitions.

Therefore, traditionally automata or related models are used to give an abstract account of the *system* to be verified, whereas formulas are used to specify *properties* of these systems. But, since it is possible to translate between automata and formulas and back, the choice is a matter of complexity, of available algorithms and of taste. We could equally well define both system and properties in temporal logic; in this case we would have to prove an implication formula (section 6 will explain how to do this). Another alternative is that both the implementation and the specification are given as automata, where the latter is more "abstract" than the former. Then we have to prove that one can *simulate* the other.

Transition Systems

Any Kripke model can be regarded as an automaton without acceptance or recurrence conditions (i.e., as a *transition system*): Given a model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ over $\mathcal{P}$ and $\mathcal{R}$, let $\mathcal{L}(w) \triangleq \{\mathtt{p} \mid \mathtt{p} \in \mathcal{I}(w)\} \subseteq 2^{\mathcal{P}}$ be the label of a point. The alphabet of the automaton corresponding to $\mathcal{M}$ is $2^{\mathcal{P}} \times \mathcal{R}$. The set of states is the set of arcs in the model, i.e., the set of tuples (w, w') such that $(w, w') \in \mathcal{I}(R)$ for some $R \in \mathcal{R}$, plus an additional initial state s_0. Two states (w_1, w_1') and (w_2, w_2') are related by (P, R) (where $P \subseteq \mathcal{P}$) iff $w_1' = w_2$, $(w_2, w_2') \in \mathcal{I}(R)$ and $\mathcal{L}(w_1') = \mathcal{L}(w_2) = P$. Similarly, the initial state s_0 is related to (w_0, w_0') by (P, R), iff w_0 is the current point of the model, $(w_0, w_0') \in \mathcal{I}(R)$ and $\mathcal{L}(w_0) = P$. The set of acceptance and recurrence states of the automaton are defined to be the set of all of its states. Then, this automaton accepts exactly the set of all natural models which can be generated from the Kripke model. Note that this set is a prefix-closed language, that is, one which contains with every word also all of its prefixes. In general, we can not go back from an automaton to a *single* Kripke model, since automata correspond to formulas, i.e. *sets* of models.

Thus, models can be seen as automata. Likewise, formulas can be seen as automata: the result of the previous section implies that for every **LTL** formula there exists an equivalent automaton. Because this proof is constructive, it yields a method to obtain such an automaton; however, a much more concise way of constructing it is the tableau construction sketched in section 6.

A Kripke model $\mathcal{M}$ satisfies a given **LTL**-formula φ iff the language of $\mathcal{M}$ is a subset of the language of the automaton $\mathcal{M}_\varphi$ for φ. That is, $\mathcal{M} \models \varphi$ iff $L(\mathcal{M}) \subseteq L(\mathcal{M}_\varphi)$. (We assume that the alphabet of $\mathcal{M}$ is including the alphabet of φ, and that the language of $\mathcal{M}$ is obtained by projection onto the alphabet of φ.) The latter condition is equivalent to $L(\mathcal{M}) \cap -L(\mathcal{M}_\varphi) = \emptyset$. Therefore, a feasible way to check whether $\mathcal{M} \models \varphi$ is to construct the automaton $\mathcal{M}_{\neg\varphi}$ for $\neg\varphi$, and to check whether the language of the product automaton $\mathcal{M} \times \mathcal{M}_{\neg\varphi}$ is empty.

If both system $\mathcal{M}$ and property φ are given as automata, then "specification" φ can be regarded as a "more abstract version" of the "implementation" $\mathcal{M}$. We write $\mathcal{M}_I \models \mathcal{M}_S$ if $L(\mathcal{M}_I) \subseteq L(\mathcal{M}_S)$, i.e., if the language of $\mathcal{M}_I$ is a subset of the language of $\mathcal{M}_S$. A *property* φ is defined to be just any ω-language $\varphi \subseteq \Sigma^\omega$, where $\Sigma = 2^{\mathcal{P}} \times \mathcal{R}$.

For Büchi-automata $\mathcal{M}_1$ and $\mathcal{M}_2$ it holds that $\mathcal{M}_1 \models \mathcal{M}_2$

iff for all properties φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$
iff for all ω-regular φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.

Safety and Liveness Properties

A similar characterisation holds for finite transition systems and safety-properties: For a natural model $\mathcal{M}$, let $\mathcal{M}^{[..i]}$ be the model consisting of the first i points of $\mathcal{M}$, and $\mathcal{M}_1 \circ \mathcal{M}_2$ be the concatenation of the two models $\mathcal{M}_1$ and $\mathcal{M}_2$.

- φ is a *safety property*, iff for every natural model $\mathcal{M}$,

$$\mathcal{M} \models \varphi \text{ if } \forall i \exists \mathcal{M}' : \mathcal{M}^{[..i]} \circ \mathcal{M}' \models \varphi$$

- φ is a *liveness property*, iff for every natural model $\mathcal{M}$,

$$\forall i \exists \mathcal{M}' : \mathcal{M}^{[..i]} \circ \mathcal{M}' \models \varphi$$

Thus, φ is a safety property if for every model *not* satisfying φ there is a finite prefix $\mathcal{M}^{[..i]}$ which can not be completed by any continuation $\mathcal{M}'$ such that $\mathcal{M}^{[..i]} \circ \mathcal{M}' \models \varphi$. In other words, for every model dissatisfying φ something "bad" must have happened after some finite number of steps which cannot be remedied by any future (well-) behaviour. Hence, in Lamport's popular characterisation, safety properties express that "something bad never happens".

A liveness property φ, on the other hand, can never be refuted by observing only a finite prefix of some run. It holds, if and only if every finite sequence can be completed to a model satisfying φ, hence φ states that "something good eventually happens". Notice, however, that in contrast to the "bad thing" referred to above, the occurrence of the "good thing" does not have to be observable in any fixed time interval.

Facts about safety and liveness:

- Safety properties are closed under finite unions and arbitrary intersections.

- Liveness properties are closed under arbitrary unions, but not under intersections (e.g., $(\Diamond\Box \mathtt{p} \wedge \Box\Diamond\neg \mathtt{p}) \leftrightarrow \bot$).
- The complement of safety properties are the open sets, and liveness properties are the dense sets in the product topology on finite and infinite words.
- $\top$ is the only property which is both a safety and a liveness property.
- For any φ there exists a safety property φ_S and a liveness property φ_L such that $\varphi \leftrightarrow (\varphi_S \wedge \varphi_L)$.

A Kripke model or automaton is called *image finite*, if every point has only finitely many successors. In particular, finiteness implies image finiteness.

For image finite transition systems $\mathcal{M}_1$ and $\mathcal{M}_2$ it holds that $\mathcal{M}_1 \models \mathcal{M}_2$ iff for all safety properties φ it holds that $\mathcal{M}_1 \models \varphi$ if $\mathcal{M}_2 \models \varphi$. To see why this is so, assume that $\mathcal{M}_1 \models \mathcal{M}_2$, and that $\mathcal{M}_1 \not\models \varphi$. Then there exists a natural model generated from $\mathcal{M}_1$ dissatisfying φ. Since $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$, this countermodel can also be generated from $\mathcal{M}_2$, hence $\mathcal{M}_2 \not\models \varphi$. For the other direction, note that the set of all natural models generated from an image finite transition system is a safety property. This can be proved using König's lemma. Therefore, by the fact that $\mathcal{M}_2 \models \mathcal{M}_2$ the assumption immediately reduces to $\mathcal{M}_1 \models \mathcal{M}_2$.

Characterisation of Safety Properties

In modal logic, safety properties are exactly those not involving any diamond operator. More precisely, propositions and $\top, \bot$ are modal safety properties, and if φ and ψ are modal safety properties, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ and $[R]\varphi$ are modal safety properties. Any formula of modal logic which is a safety property can be written as a modal safety property.

For linear temporal logic, an **LTL** safety property can be defined as a formula $\Box\psi$, where ψ is a boolean combination of pure present and pure past formulas. It is not difficult to show that any such formula defines a safety property; moreover, also the other direction holds: all temporal formulas which are safety formulas can be written as an **LTL** safety property.

For **qTL**, a definition of **qTL** safety property is immediate from the definition of safety property. For μ**TL**, we take all formulas of the kind $\nu q(\psi \wedge [R]q)$, where ψ is a pure present formula.

When does an ω-automaton specify a safety property? We call a strongly connected component (SCC) in a Büchi-automaton *bad*, if it is either trivial and non-accepting, or nontrivial and without any recurrence state. We call it *good*, if it is trivial and accepting or nontrivial and contains a recurrence state. A *safety automaton* has for any bad SCC B, and any good SCC G, that either B is not reachable from the initial state, or G is not reachable from B.

Any language accepted by a safety automaton is a safety property, and for any ω-regular safety property there is a safety automaton accepting it. Furthermore, by removing nonreachable parts and bad SCCs, for any safety automaton we can obtain an equivalent transition system. Given any automaton, it can be decided with linear time complexity whether this automaton presents a safety property or not.

Classification of Temporal Properties

Manna and Pnueli [8] gives a hierarchy of property classes, which is derived from an identical hierarchy in topology. Assume that φ and ψ are boolean combinations of pure present and pure past temporal formulas.

- Safety: $\Box\varphi$
- Guarantee: $\Diamond\varphi$
- Obligation: $(\Box\varphi \vee \Diamond\psi)$ (or $(\Diamond\varphi \rightarrow \Diamond\psi)$)
- Response: $\Box\Diamond\varphi$
- Persistence: $\Diamond\Box\varphi$
- Reactivity: $(\Box\Diamond\varphi \vee \Diamond\Box\psi)$ (or $(\Box\Diamond\varphi \rightarrow \Box\Diamond\psi)$)

These are only normal forms! E.g., $\Diamond(\mathtt{p} \wedge \Diamond\mathtt{q})$ is equivalent to the guarantee formula $\Diamond(\mathtt{q} \wedge \mathcal{S}(\mathtt{p}, \top))$. As another example, $\Box\varphi \rightarrow \Diamond\psi$ is equivalent to $\Box\Diamond\mathcal{S}(\neg\varphi \wedge \psi, \neg\varphi)$

All safety and guarantee properties are also obligations; the class of obligation properties is properly included in the response and persistence classes. Response and persistence are closed under disjunctions and conjunctions, and are properly included in reactivity. Every **LTL**-property can be written as a conjunction of reactivity formulas, since these can be used to describe the acceptance condition of ω-automata.

5 Completeness

Logicians are interested in logical truths, i.e., in the set of formulas which are valid in *all* models of the logic.

What about specific theories like the theory of groups, or the theory of a specific given model? How does it help to know about the set of *all* valid formulas when we want to find out whether a particular formula φ holds for a given model or theory?

Answer: encode the model or theory as a set of *assumptions* Φ and check whether the formula in question *follows* from Φ!

In fact, a *logic* can be defined to be any set of well-formed formulas which is closed under provable consequence; and a *theory* is a set of well-formed formulas which is closed under semantical consequence.

Thus there are three notions of consequence involved here:

- $\Phi \mid\!\!\sim \varphi$ if from Φ follows φ,
  i.e. if any model in which all formulas from Φ are valid also validates φ,
- $\Phi \vdash \varphi$ if Φ proves φ,
  i.e. if there is a proof of φ which uses only assumptions from Φ, and
- $\Phi \rightarrow \varphi$ if Φ implies φ,
  i.e. this is a statement of the object language which is only defined if Φ is a single formula. To be liberal, we can identify a finite set of formulas $\{\varphi_1, \ldots, \varphi_n\}$ with the conjunction $(\varphi_1 \wedge \ldots \wedge \varphi_n)$.

Note that $\Phi \mid\!\!\sim \varphi$ is different from $\mathcal{M} \models \varphi$! The notations $\mid\!\!\sim \varphi$ and $\vdash \varphi$ are short for $\{\} \mid\!\!\sim \varphi$ and $\{\} \vdash \varphi$, respectively.

Of course, the semantical notion of *validity* sometimes is restricted to certain classes of models, e.g., to those satisfying certain axioms, or to natural or tree models. For the sake of simplicity, we will restrict ourself to natural models in this section. However, all results apply to tree models as well. To be able to talk about propositions *and* actions, we assume that the transition from one point to the next carries a unique label $a \in \mathcal{R}$.

Also, the syntactical notion of *provability* sometimes is parametrised to a certain proof-system. In this section, we will use *Hilbert-style* proof-systems, consisting of a set of *axioms* and derivation rules. Usually, axioms and derivation rules contain *propositional variables* $q \in \mathcal{Q}$ and a substitution rule allowing consistent replacement of propositional variables with formulas. (Conceptually, propositional variables are not the same as propositions, though many authors do not distinguish between these syntactic categories.)

To complicate things even more, there are two notions of validity of a formula: *local validity* $(U, \mathcal{I}, w_0) \models \varphi$, where the evaluation point is given, and *universal validity* $(U, \mathcal{I}) \models \varphi$. Traditionally, focus has been on complete axioms for universal validity rather than for the local version; proofs are much simpler. Thus, we are interested in formulas which are valid *in all models at all points*.

One of the major concerns after defining a logical language and its models is to find an *adequate* proof-system for the logic, i.e. one which is both *correct* and *complete*. That is, for any Φ and φ,

- if $\Phi \vdash \varphi$, then $\Phi \mid\!\!\sim \varphi$ (Correctness), and
- if $\Phi \mid\!\!\sim \varphi$, then $\Phi \vdash \varphi$ (Completeness).

Why should these statements be valid?

Correctness should be clear: We don't want to be able to "prove" false statements. Usually correctness is very easy to show, we just have to show that the axioms are valid, and that the derivation rules only allow to deduce valid formulas from valid formulas.

Completeness is in most cases much harder to show, if not impossible. So, why is it important to show completeness? Firstly, we would like to make sure that any specification which is satisfied by a program can be proved from the program axioms, provided the specification is expressible in the logic. Secondly, and more important, in many cases decision algorithms for automated verification can be obtained from the completeness proofs.

Deductions in Multimodal Logic

To illustrate the basic idea, we start with a deductive system for multimodal logic for natural models. A number of similar proofs can be found in [2]. We use the following axioms and rules:

(taut)	propositional tautologies
(MP)	$p,\ (p \to q) \vdash q$
(N)	$q \vdash [a]q$
(K)	$\vdash [a]p \wedge [a](p \to q) \to [a]q$
(U)	$\vdash \langle a\rangle q \to [a]q$
(L)	$\vdash \langle a\rangle q \to [b]\neg q$ for $a \neq b$

To prove $\Phi \vdash \varphi$ we have to give a *derivation* of φ from the assumptions Φ, i.e., a sequence of formulas such that the last element of this sequence is φ, and every element of this sequence is either from Φ, or a substitution instance of an axiom, or the substitution instance of the consequence of a rule, where all premisses of the rule for this substitution appear already in the derivation.

As an example, let us assume $(\mathtt{p} \to \mathtt{q})$ and derive some consequences:

1.	$\mathtt{p} \to \mathtt{q}$	(ass)
2.	$[a](\mathtt{p} \to \mathtt{q})$	(1, **N**)
3.	$[a](\mathtt{p} \to \mathtt{q}) \to ([a]\mathtt{p} \to [a]\mathtt{q})$	(**K**, taut)
4.	$[a]\mathtt{p} \to [a]\mathtt{q}$	(2,3,MP)
5.	$\neg\mathtt{q} \to \neg\mathtt{p}$	(1)
6.	$[a]\neg\mathtt{q} \to [a]\neg\mathtt{p}$	(5)
7.	$\langle a\rangle\mathtt{p} \to \langle a\rangle\mathtt{q}$	(6)

Lines (4) and (7) form the basis for an inductive proof of the following replacement and monotonicity rules:

(repl) $p \leftrightarrow q \vdash \varphi(p) \leftrightarrow \varphi(q)$, and
(mon) $p \to q \vdash \varphi(p) \to \varphi(q)$.

(mon) requires that $\varphi(q)$ is *positive* in q, that is, that every occurrence of q is under an even number of negation signs (an exact definition will be given in section 9). For example, $[a]q$, $\langle a\rangle q$, $q \wedge [a](q \vee \langle a\rangle q)$ are all positive in q.

Using these rules, we prove that **(L)** is equivalent to $[a]\bot \vee [b]\bot$:

1.	$\langle a\rangle q \to [b]\neg q$	(**L**)
2.	$\langle a\rangle\neg\bot \to [b]\neg\neg\bot$	(1)
3.	$[a]\bot \vee [b]\bot$	(2,repl)

And the other direction:

4.	$\bot \to \neg q$	(taut)
5.	$[a]\bot \to [a]\neg q$	(4, mon)
6.	$\langle a\rangle q \to \neg[a]\bot$	(5)
7.	$\neg[a]\bot \to [b]\bot$	(3)
8.	$[b]\bot \to [b]\neg q$	(4, mon)
9.	$\langle a\rangle q \to [b]\neg q$	(6,7,8)

As a more practical example, let us derive from the assumptions

$\mathtt{set} \to \langle V\rangle\neg\mathtt{set}$ and
$\neg\mathtt{set} \to \langle P\rangle\mathtt{set} \vee \langle V\rangle\neg\mathtt{set}$ the property
$[P][P]\bot$.

The assumptions can be seen as describing the actions of a semaphore with two states, $\mathtt{set}$ and $\neg\mathtt{set}$, which allows to be set with a P-operation when it is not set, and to be freed with a V-operation when it is either set or not set. The given property describes that there are never two P operations in a row.

1.	$\bot \to [P]\bot$	(taut)
2.	$[P]\bot \to [P][P]\bot$	(1,mon)
3.	$\langle V\rangle\top \to [P]\bot$	(**L**)
4.	$\langle V\rangle\neg\mathtt{set} \to \langle V\rangle\top$	(mon)
5.	$\langle V\rangle\neg\mathtt{set} \to [P]\bot$	(3,4)
6.	$\langle V\rangle\neg\mathtt{set} \to [P][P]\bot$	(2,5)
7.	$\langle P\rangle\langle V\rangle\neg\mathtt{set} \to \langle P\rangle[P]\bot$	(5,mon)
8.	$\mathtt{set} \to \langle V\rangle\neg\mathtt{set}$	(ass)
9.	$\mathtt{set} \to [P][P]\bot$	(6,8)
10.	$\langle P\rangle\mathtt{set} \to \langle P\rangle\langle V\rangle\neg\mathtt{set}$	(8,mon)
11.	$\langle P\rangle\mathtt{set} \to \langle P\rangle[P]\bot$	(7,10)
12.	$\langle P\rangle[P]\bot \to [P][P]\bot$	(**U**)

13. $\langle P\rangle \mathtt{set} \rightarrow [P][P]\bot$ (11,12)
14. $(\langle P\rangle \mathtt{set} \vee \langle V\rangle \neg \mathtt{set}) \rightarrow [P][P]\bot$ (6,13)
15. $\neg \mathtt{set} \rightarrow \langle P\rangle \mathtt{set} \vee \langle V\rangle \neg \mathtt{set}$ (ass)
16. $\neg \mathtt{set} \rightarrow [P][P]\bot$ (14,15)
17. $(\mathtt{set} \vee \neg \mathtt{set}) \rightarrow [P][P]\bot$ (9,16)
18. $[P][P]\bot$ (17)

As we see, even in such relatively easy examples it can be quite cumbersome to find a Hilbert-style proof "by hand"; it should be possible to conduct these proofs automatically. This will be the topic of the next section!

Let us argue about the correctness of our deduction rules. **(taut)** and **(MP)** are immediately clear. **(N)** is the so called *necessitation rule*. Its validity depends on the universal interpretation of validity: If some formula is true in every point of a model, it is true in every point which is the a-successor of some other point in that model. **(K)** is the classical *Kripke*-axiom which holds for all normal modal logics. If in all accessible points p holds, and in all accessible points $(p \rightarrow q)$ holds, then in all accessible points q must hold. **(U)** is the axiom describing that the next-step relation is *univalent*: If there is any successor satisfying q, then all successors satisfy q. This holds because at any given moment, there is at most one successor which can be reached. While this is true for natural models, it does not hold for trees or other branching structures. **(L)** finally is an additional axiom for the labelling of the next-step relation by transition relations. If some a-successor satisfies q, then the next state is determined by an a-step, hence it is not a b-step, and all states reachable by a b-step are false. Again, this only holds because we are considering natural models (i.e. paths through a Kripke model, not the Kripke model as such).

Completeness of Multimodal Logic

The classical way to prove completeness is the so–called Henkin/Hasenjäger construction. A set Ψ of formulas is *inconsistent with* Φ, if there is a finite subset $\{\psi_1, \ldots, \psi_n\} \subseteq \Psi$ such that $\Phi \vdash (\neg\psi_1 \vee \ldots \vee \neg\psi_n)$. To prove completeness, we have to show

(*) Every formula consistent with Φ is satisfiable in a model validating Φ.

For, if $\Phi \mid\!\!\sim \varphi$, then no model validating Φ satisfies $\{\neg\varphi\}$; therefore with **(*)** it follows that $\{\neg\varphi\}$ is inconsistent with Φ, hence $\Phi \vdash \varphi$. (Without loss of generality, we can assume here Φ to be consistent with itself, or else $\Phi \mid\!\!\sim \varphi$ holds).

Thus, the task is to construct a model for a given consistent set of formulas. *Lindenbaum's extension lemma* states that for any formula φ which is consistent with Φ there exists a maximal consistent set w_0 such that $\varphi \in w_0$ and $\Phi \subseteq w_0$: Start with $\Phi \cup \{\varphi\}$; for every formula ψ according to a fixed enumeration add either ψ or $\neg\psi$ to w, whichever is consistent with the set constructed so far.

The *canonical model* for Φ is $(U, \mathcal{I}, w)$, where

- U is the set of maximal consistent sets which include Φ,
- $\mathcal{I}(a) \triangleq \{(w_0, w_1) \mid q \in w_1 \rightarrow \langle a\rangle q \in w_0\}$, and
- $\mathcal{I}(\mathtt{p}) \triangleq \{w_0 \mid \mathtt{p} \in w_0\}$, and
- w is any element from U such that $\varphi \in w$.

For every $w_0 \in U$ of our canonical model and every $a \in \mathcal{R}$ there is at most one w_1 with $(w_0, w_1) \in \mathcal{I}(a)$. For, assume $(w_0, w_1) \in \mathcal{I}(a)$ and $(w_0, w_1') \in \mathcal{I}(a)$. Then, there must be a formula ψ such that $\psi \in w_1$ and $\psi \notin w_1'$, or else $w_1 = w_1'$. Since w_1' is maximal, $\neg\psi \in w_1'$. Therefore $\langle a\rangle\psi \in w_0$ and $\langle a\rangle\neg\psi \in w_0$. But, this is a contradiction to the consistency of w_0: axiom **U** requires that if $\langle a\rangle\psi \in w_0$, then $\neg\langle a\rangle\neg\psi \in w_0$.

Similarly, we can show that for every $w_0 \in U$ there is at most one a such that $(w_0, w_1) \in \mathcal{I}(a)$. The opposite assumption would lead to a contradiction with axiom **L**.

The fundamental 'truth' or 'killing' lemma states that for any formula φ and maximal consistent set w it holds that $\varphi \in w$ iff $(U, \mathcal{I}, w) \models \varphi$.

In the inductive step for this lemma, we have to show that $\langle a\rangle\varphi \in w_0$ iff $(U, \mathcal{I}, w_0) \models \langle a\rangle\varphi$. The 'if' direction being a direct consequence of definition and induction hypothesis, assume that $\langle a\rangle\varphi \in w_0$. We have to find a maximal consistent set w_1 such that $(w_0, w_1) \in \mathcal{I}(a)$ and $\varphi \in w_1$. Since $\vdash (\langle a\rangle\varphi \wedge [a]\psi) \rightarrow \langle a\rangle(\varphi \wedge \psi)$, the set $\{\varphi\} \cup \{\psi_j \mid [a]\psi_j \in w\}$ is consistent. Let w_1 be any maximal consistent extension of this set. Then for all $\psi \in w_1$ the formula $\langle a\rangle\psi$ must be in w_0 (otherwise, a contradiction could be derived). Therefore $(w_0, w_1) \in \mathcal{I}(a)$. Since $\varphi \in w_1$, the induction hypothesis gives $(U, \mathcal{I}, w_1) \models \varphi$. Together with $(w_0, w_1) \in \mathcal{I}(a)$ we have $(U, \mathcal{I}, w_0) \models \langle a\rangle\varphi$.

Since for the canonical model $(U, \mathcal{I}, w)$ it holds that $\Phi \subseteq w$ and $\varphi \in w$, we proved that $(U, \mathcal{I}, w) \models \Phi$ and $(U, \mathcal{I}, w) \models \varphi$. Thus we have achieved our goal of constructing a natural model for $\Phi \cup \{\varphi\}$.

Completeness of Temporal Logic

How can this completeness proof be lifted to more expressive logics like **LTL**?

Let us for the moment focus on temporal logic on natural models with the operators $\circ$ for the union of all accessibility relations and $\Diamond$ for the transitive closure; the extensions for until- and since operators being almost straightforward extensions of the basic ideas.

The relation between $\circ$ and $\langle a\rangle$ is fixed by the following axiom:

(nex) $\vdash \langle a\rangle q \rightarrow \circ q$ for all $a \in \mathcal{R}$

A close inspection of the semantics of $\Diamond$ reveals a fundamental problem:

Consider the set $\Phi \triangleq \{\circ\,\mathtt{p},\ \circ\circ\,\mathtt{p},\ \circ\circ\circ\,\mathtt{p},\ \ldots\}$. Then clearly $\Phi \mid\!\!\sim \Box\varphi$. However, $\Phi \not\vdash \Box\varphi$, since every proof of $\Box\varphi$ from Φ can use only a limited number of premisses (proofs are *finite* sequences). But, for no finite subset $\Phi_0 \subset \Phi$ the statement $\Phi_0 \vdash \Box\varphi$ holds.

Where does the above completeness proof fail? It is not possible to find a maximal consistent extension, since we can not apply an axiom to show the consistency of an infinite set of premisses.

When dealing with second order concepts like transitive closure we have to limit ourselves to a weaker form of completeness. Call a logic *weakly complete*, if for all *finite* Φ it holds that $\Phi \mid\!\!\sim \varphi$ implies $\Phi \vdash \varphi$.

In first order logic, the *deduction theorem* allows to discard any finite set of assumptions: $\psi \mid\!\!\sim \varphi$ iff $\mid\!\!\sim \forall\psi \rightarrow \varphi$,

where $\forall\psi$ is the universal closure of ψ. In temporal logic, similar deduction theorem holds:

$$\psi \mid\!\!\sim \varphi \text{ iff } \mid\!\!\sim \psi \wedge \Box\psi \rightarrow \varphi$$

Therefore, to prove weak completeness it suffices to prove that $\mid\!\!\sim \varphi$ implies $\vdash \varphi$.

We use the following axiom set (in addition to the modal axioms above):

(Rec) $\vdash \circ(q \vee \Diamond q) \rightarrow \Diamond q$
(Ind) $\circ(p \vee q) \rightarrow q \vdash \Diamond p \rightarrow q$

Dually, this can be written as

(Rec) $\vdash \Box q \rightarrow \neg \circ \neg(q \wedge \Box q)$
(Ind) $q \rightarrow \neg \circ \neg(p \wedge q) \vdash q \rightarrow \Box p$

These are the so-called *Segerberg axioms* reflecting the definition of the transitive closure as the minimal transitive relation which includes all $a \in \mathcal{R}$. **(Rec)** is the *recursion* axiom which can be used to unfold a Box-operator:

$$\Box\varphi \rightarrow \neg \circ \neg(\varphi \wedge \neg \circ \neg(\varphi \wedge \neg \circ \neg(\varphi \wedge \ldots))).$$

(Ind) is the *induction* axiom which can be used to derive a property $\Box\varphi$ from an *invariant* ψ, i.e. from a formula ψ for which $\psi \rightarrow [a]\psi$ and $\psi \rightarrow [a]\varphi$ are derivable (for all $a \in \mathcal{R}$).

How do these axioms prove completeness of the transitive closure relation? Up to the truth lemma, the proof is almost the same as for modal logic. But, we only use *finite* maximal consistent sets: we start with a single (finite) consistent formula φ for which we have to construct a model. Define the notion of *extended sub-formula* of φ (sometimes also called *Fisher-Ladner closure*) as follows:

- φ is an extended sub-formula of φ,
- $\neg\varphi$ is an extended sub-formula of φ, if φ is not of form $\neg\varphi'$,
- φ_1 and φ_2 are extended sub-formulas of $(\varphi_1 \rightarrow \varphi_2)$, (thus φ is an extended sub-formula of $\neg\varphi$)
- φ is an extended sub-formula of $\circ\varphi$,
- $\circ\varphi$ is an extended sub-formula of $\Diamond\varphi$, and
- $\circ\Diamond\varphi$ is an extended sub-formula of $\Diamond\varphi$

For any given formula, there are finitely many different extended sub-formulas. Now, a consistent set of formulas is called *finitely maximal*, if it is maximal with respect to extended sub-formulas; that is, for every extended sub-formula ψ of φ, either ψ or $\neg\psi$ is in the finitely maximal consistent set.

In the proof of the truth lemma we additionally have to show

$$\Diamond\varphi \in w_0 \text{ iff } (U, \mathcal{I}, w_0) \models \Diamond\varphi.$$

One direction again is easy: If $\Box\varphi \in w_0$, then $\Box\varphi$ must be in any finitely maximal consistent set reachable from w_0 by any number of steps, because the recursion axiom forces $\neg \circ \neg\Box\varphi$ to be in w_0, and hence $\Box\varphi$ is in every w_1 with $(w_0, w_1) \in \mathcal{I}(a)$.

The other direction follows from the induction axiom: Assume that $\Diamond\varphi \in w_0$, but no finitely maximal consistent set reachable from w_0 has φ in it. Let $\Psi_1, \ldots, \Psi_n$ be all different finitely maximal consistent sets in the same strongly connected component as w_0, and $\Psi \triangleq \Psi_1 \vee \ldots \vee \Psi_n$, where $\Psi_i \triangleq \bigwedge\{\psi \mid \psi \in \Psi_i\}$ (remember that the Ψ_i are finite). Then

- $\vdash w_0 \rightarrow \Psi$, since w_0 is one of the Ψ_i of which Ψ is composed. Furthermore,
- $\vdash \Psi \rightarrow \neg\varphi$, since $\neg\varphi$ was assumed to hold in the whole component. Finally,
- $\vdash \Psi \rightarrow \neg\circ\neg\Psi$, since Ψ consists of *all* finitely maximal consistent sets in this component.

Putting these parts together, we have a contradiction, since the induction axiom gives $\vdash w_0 \rightarrow \Box\neg\varphi$, but $\Diamond\varphi \in w_0$.

A detailed exposition of this proof can be found in [7].

Completeness of the μ-calculus

We just briefly indicate how the above axioms can be extended for μ**TL**.

(Recν) $\vdash \nu q\ \varphi(q) \rightarrow \varphi(\nu q\ \varphi(q))$
(Indν) $q \rightarrow \varphi(q) \vdash q \rightarrow \nu p\ \varphi(p)$

The recursion and induction axiom can be obtained as special cases of these very general axioms by defining $\Box p \triangleq \nu q \bigwedge_i [a_i](p \wedge q)$.

The above completeness proof can be adapted to show completeness for a certain subclass of monotonic μ**TL** formulas, the *aconjunctive* ones.

The problem of completeness of these axioms for *all* μ**TL** formulas had been open for more than a decade until it was solved in [13]. It can be shown that for any formula there exists an equivalent aconjunctive formula. Thereby it suffices to derive this aconjunctive formula from the axioms in order to prove any given formula.

This proof also applies to tree models. In general, however, for each class of models under consideration the completeness question has to be solved independently.

6 Decision Procedures

In this section we derive *decision procedures* for some of the logics introduced above. We already indicated that the decision procedures will be extracted from the completeness proofs of the previous section. In the next section, this line of thought is continued to derive *model checking algorithms* from the decision procedures.

Given a set Φ of assumptions, and a formula φ. We want to decide whether $\Phi \mid\!\!\sim \varphi$, which by completeness is the same as $\Phi \vdash \varphi$. Now $\Phi \mid\!\!\sim \varphi$ iff $\Phi \cup \{\neg\varphi\}$ is (universally) unsatisfiable. Hence we need an algorithm which, given a set of formulas, decides whether this set has a model or not.

Modal Decision Algorithms

We considered completeness of modal logic (with $\langle a\rangle$-operators) and of temporal logic (with operators $\circ$ and $\Diamond$). The completeness proof of temporal logic depended on the fact that we could use *finite* maximal consistent sets to construct our model. For multimodal logic, we allowed infinite sets of assumptions because we wanted to show strong completeness. If we restrict ourselves to the weak notion

of completeness ($\Phi \mid\!\!\sim \varphi \Rightarrow \Phi \vdash \varphi$ for *finite* Φ), then also here it is not necessary that maximal consistent sets are maximal in the space of all formulas. It is sufficient to consider maximality with respect to all *extended subformulas* of the given consistent set.

Let us quickly recall the definition of *extended subformula* for multimodal formulas:

- $SF(\bot) \triangleq \{\bot\}$
- $SF((\varphi \to \psi)) \triangleq \{(\varphi \to \psi)\} \cup SF(\varphi) \cup SF(\psi)$
- $SF(\mathtt{p}) \triangleq \{\mathtt{p}\}$
- $SF(\langle a\rangle\varphi) \triangleq \{\langle a\rangle\varphi\} \cup SF(\varphi)$
- $NSF(\varphi) \triangleq \{\neg\psi \mid \psi \in SF(\varphi)\}$
- $ESF(\varphi) \triangleq SF(\varphi) \cup NSF(\varphi)$
- $ESF(\Phi) \triangleq \bigcup\{ESF(\varphi) \mid \varphi \in \Phi\}$

Expanding the definition, we see that for any formula $\varphi = (\psi_1 \wedge \psi_2)$ or $\varphi = (\psi_1 \vee \psi_2)$, all $\{\psi_1, \psi_2, \neg\psi_1, \neg\psi_2\} \subseteq ESF(\varphi)$.

For any finite Φ, there are only finitely many different extended subformulas, and hence only finitely many sets of extended subformulas. Any such set is called *maximal* with respect to Φ, if for any $\psi \in ESF(\Phi)$, either $\psi \in w$ or $\neg\psi \in w$.

Call such a set w of subformulas *propositionally consistent*, if

$\bot \notin w$, and
if $(\psi_1 \to \psi_2) \in ESF(\varphi)$ then
$(\psi_1 \to \psi_2) \in w$ iff $\psi_1 \in w$ implies $\psi_2 \in w$.

That is, if $(\psi_1 \to \psi_2) \in w$ then $\neg\psi_1 \in w$ or $\psi_2 \in w$, and if one of $\neg\psi_1$, $\psi_2 \in w$ then $(\psi_1 \to \psi_2) \in w$. Again, expanding the definitions we see that

- for any $(\psi_1 \wedge \psi_2) \in ESF(\varphi)$,
$(\psi_1 \wedge \psi_2) \in w$ iff both $\psi_1 \in w$ and $\psi_2 \in w$,
- for any $(\psi_1 \vee \psi_2) \in ESF(\varphi)$,
$(\psi_1 \vee \psi_2) \in w$ iff $\psi_1 \in w$ or $\psi_2 \in w$.

Any propositionally maximal consistent sets is "consistent for propositional logic": if we consistently replace any modal formula in w by a new proposition, then the resulting set of formulas is satisfiable in propositional logic. A satisfying propositional interpretation is given by $\mathcal{I}(\mathtt{p}) \triangleq \mathbf{true}$ iff $\mathtt{p} \in w$.

The modal formulas in w determine the structure of the accessibility relation(s) in any model for Φ, if such a model exists. There are two approaches to construct these accessibility relations.

The first, 'local' algorithm, is tableaux-based. Start with the set $w_0, \ldots, w_n$ of all propositionally maximal consistent sets which include Φ and try to systematically extend one of these to a model. Given a propositionally maximal consistent set w_i, if it does not contain any formula $\langle a\rangle\psi$, we are finished. If it contains both some $\langle a\rangle\psi_1$ and some $\langle b\rangle\psi_2$, then this set is unsatisfiable (because we are considering models in which the labelling of any arc from some point is unique); thus we backtrack. Otherwise, there is exactly one a such that some formulas $\langle a\rangle\psi \in w_i$. Construct the sets $w_i' \triangleq \{\psi \mid \langle a\rangle\psi \in w_i\} \cup \{\neg\psi \mid \neg\langle a\rangle\psi \in w_i\}$ and $w_i'' \triangleq w_i' \cup \Phi$ (We are considering *linear* accessibility relations and *universal* consequence!). There are finitely many propositionally maximal consistent extensions $w_{i.1}, \ldots, w_{i.n}$ of w_i''.

If w_i'' is not propositionally consistent, then there is no such extension ($n = 0$): backtrack. Otherwise, recurse with all propositionally maximal consistent extensions $w_{i.j}$ of w_i and continue *ad infinitum*. No wait; that might take too long. Since there are only finitely many propositionally consistent sets, we will hit onto a cycle sooner or later. In that case, we are also finished: we have constructed a model consisting of an infinite loop.

Modal Tableau Rules

Before giving a example, we give a set of tableau rules which can be seen as another formulation of this idea. A large number of similar tableau rules for all sorts of modal logics can be found in [5].

Let Φ be the set of formulas whose satisfiability we have to check. Γ is any set of formulas;

$$(\to)\ \frac{\Gamma, (\psi_1 \to \psi_2)}{\Gamma, \neg\psi_1 \quad\quad \Gamma, \psi_2} \qquad (\neg\to)\ \frac{\Gamma, \neg(\psi_1 \to \psi_2)}{\Gamma, \psi_1, \neg\psi_2}$$

$$(\bot_1)\ \frac{\Gamma, \psi, \neg\psi}{*} \qquad (\bot_2)\ \frac{\Gamma, \bot}{*} \qquad (\mathrm{L})\ \frac{\Gamma, \langle a\rangle\psi_1, \langle b\rangle\psi_2}{*}$$

$$(\neg\neg)\ \frac{\Gamma, \neg\neg\psi}{\Gamma, \psi}$$

$$(\langle\rangle)\ \frac{\Gamma, \langle a\rangle\varphi_1, \ldots, \langle a\rangle\varphi_n, \neg\langle a\rangle\psi_1, \ldots, \neg\langle a\rangle\psi_m}{\Phi, \varphi_1, \ldots, \varphi_n, \neg\psi_1, \ldots, \neg\psi_m} \qquad ([]) \frac{\Gamma}{\Gamma}$$

Derived rules:

$$(\vee)\ \frac{\Gamma, (\psi_1 \vee \psi_2)}{\Gamma, \psi_1 \quad\quad \Gamma, \psi_2} \qquad (\neg\vee)\ \frac{\Gamma, \neg(\psi_1 \vee \psi_2)}{\Gamma, \neg\psi_1, \neg\psi_2}$$

$$(\wedge)\ \frac{\Gamma, (\psi_1 \wedge \psi_2)}{\Gamma, \psi_1, \psi_2} \qquad (\neg\wedge)\ \frac{\Gamma, \neg(\psi_1 \wedge \psi_2)}{\Gamma, \neg\psi_1 \quad\quad \Gamma, \neg\psi_2}$$

$$(\neg\langle a\rangle)\ \frac{\Gamma, \neg\langle a\rangle\psi}{\Gamma, [a]\neg\psi} \qquad (\neg[a])\ \frac{\Gamma, \neg[a]\psi}{\Gamma, \langle a\rangle\neg\psi}$$

$$(\mathbf{L'})\ \frac{\Gamma, \langle a\rangle\psi_1, [b]\psi_2}{\Gamma, \langle a\rangle\psi_1} \qquad (\mathbf{U})\ \frac{\Gamma, \langle a\rangle\psi_1, [a]\psi_2}{\Gamma, \langle a\rangle\psi_1, \langle a\rangle\psi_2}$$

The tableau rules allow to derive a set of sets of formulas from any set of formulas. Additional regulations are:

- Rule ($\to$) can only be applied if $\psi_2 \neq \bot$
- Rules ($\langle\rangle$) and ([]) can only applied if no other rule is applicable.
- Rule ($\langle\rangle$) can only be applied if no other $\langle a\rangle\varphi$ or $\neg\langle a\rangle\psi$ is in Γ
- Rule ([]) can only be applied if no $\langle a\rangle\varphi$ is in Γ

A *tableau* is a finite tree of sets of formulas such that

- The root of the tableau is Φ, and
- The successors of each node are constructed according to some tableau rule.

A leaf is called *closed*, if it consists of the symbol $*$. It is called *open*, if it consists of a subset of formulas of some other node on the path from the root to this leaf. (In particular, if rule ($\langle\rangle$) regenerates the root Φ, the new leaf is open. Also, any empty node constructed by rule ([]) is open). A tableau is *completed*, if any leaf is closed or open. A completed tableau is *successful*, if it contains an open leaf.

The tableau rules are formulated in a nondeterministic way, since we did not specify any order in which the rules have to be applied. Nevertheless all tableaus for a given

formula are equivalent: If Φ has any successful tableau, then every completed tableau for Φ is successful.

Adequacy of the Modal Tableau Procedure

Φ is satisfiable iff Φ has a successful tableau.

The proof of this statement is more or less straightforward: Assume Φ is satisfiable in a natural model $\mathcal{M} \triangleq ((w_0, w_1, w_2, ...), \mathcal{I}, w_0)$, and show that there is a tableau for Φ with an open leaf. Equivalently, assume that any tableau for Φ is given, and show that it contains an open leaf. We construct a sequence of tableau nodes n_i, and associate a point $w(n_i)$ in the model with any n_i. As an invariant of this construction, we show that for all formulas $\psi \in n_i$ it holds that $w(n_i) \models \psi$. Initially n_0 is the root of the tableau, with $w(n_0) \triangleq w_0$. Since $w_0 \models \Phi$, the invariant is satisfied. Given any tableau node n_i with $w(n_i) = w_j$, no closing rules can be applicable, because this would contradict the invariant. Assume the successor of w_i is constructed by rule $(\neg \rightarrow)$ or $(\neg\neg)$. Then $w(n_{i+1}) \triangleq w_j$, and the invariant is preserved. If two successors of w_i are constructed by rule $(\rightarrow)$, then any one of them is chosen which preserves the invariant, and again $w(n_{i+1}) \triangleq w_j$. If n_i has a successor obtained by rule $(\langle\rangle)$, then $w(n_{i+1}) \triangleq w_{j+1}$. The specific formulation of the rule guarantees that the invariant is preserved. Since the tableau is finite, and we can never apply one of the closing rules, we must hit onto an open leaf sooner or later.

For the other direction, we have to show that from any tableau with open leafs we can construct a model. The construction is similar to above. We consider the *unfolding* of the tableau, which is the tree arising from the repeated substitution of any open leaf with the subtableau rooted at the node subsuming this open leaf. If the tableu contains open leafs, then the unfolding contains infinite paths. In the unfolding, call any node whose successor is constructed by rule $(\langle\rangle)$ or $([])$ a *pre-state*. The set of pre-states of any infinite path from the root constitutes an infinite model.

As an example for the tableau construction, we again prove $[P][P]\bot$ from the assumptions

$$\Phi = \{(\mathtt{s} \rightarrow \langle V\rangle\neg\mathtt{s}), (\neg\mathtt{s} \rightarrow \langle P\rangle\mathtt{s} \vee \langle V\rangle\neg\mathtt{s})\}.$$

A formula is valid, if its negation is unsatisfiable; hence we start the tableau with $(\mathtt{s} \rightarrow \langle V\rangle\neg\mathtt{s})$, $(\neg\mathtt{s} \rightarrow \langle P\rangle\mathtt{s} \vee \langle V\rangle\neg\mathtt{s})$, and $\langle P\rangle\langle P\rangle\top$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Phi,\ \langle P\rangle\langle P\rangle\top}{\neg\mathtt{s},\ \langle P\rangle\mathtt{s},\ \langle P\rangle\langle P\rangle\top}}{\Phi,\ \mathtt{s},\ \langle P\rangle\top}}{\mathtt{s}, \langle V\rangle\neg\mathtt{s},\ \langle P\rangle\top}}{*} \qquad \dfrac{\langle V\rangle\neg\mathtt{s}, \langle P\rangle\langle P\rangle\top}{*} \qquad \dfrac{\ldots}{*}$$

Here the dots indicate a number of other branches closed by rule (**L**). In the completed tableau, since every leaf is closed, the original formula $[P][P]\bot$ follows from the assumptions Φ.

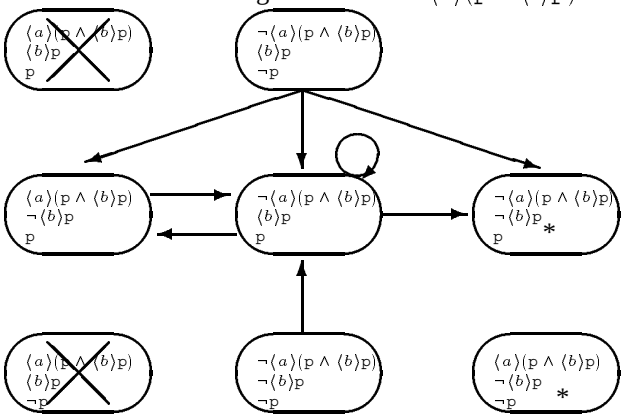
This example exhibits the connection between the tableau method and the local satisfiability algorithm sketched above: The propositional tableau rules systematically generate all necessary propositionally maximal consistent extensions of a given set of formulas, and the modal rules fix the structure of the accessibility relations in the generated model graph.

Global Modal Satisfiability

The second, 'global' algorithm for testing satisfiability of a set of formulas starts with the set W of all propositionally maximal consistent sets and the universal relation for any $\langle a\rangle$ operator. We first delete all nodes which contain both some $\langle a\rangle\psi_1$ and $\langle b\rangle\psi_2$. Then we iteratively delete 'bad arcs' and 'bad nodes' until stabilisation is reached. *Bad arcs* are pairs $(w_0, w_1) \in \mathcal{I}(a)$ such that w_0 contains $\langle b\rangle\psi'$ for some $b \neq a$, or $\langle a\rangle\psi$ or $[a]\psi$, but it is not the case that $\psi \in w_1$. *Bad nodes* w_0 contain a formula $\langle a\rangle\psi$, but there does not (or no longer) exist a tuple $(w_0, w_1) \in \mathcal{I}(a)$ with $\psi \in w_1$. The given formula set Φ is satisfiable iff upon termination there is a node n left in which it is included $(\Phi \subseteq n)$.

Since this algorithm iterates on all nodes and on all subformulas, we can implement it by a search on all nodes, with nested iteration on all subformulas of this node, or by a bottom up iteration on all diamond-subformulas, where we check all node whether they are 'bad' with respect to this subformula. In both cases, it is important to re-iterate after some deletions have taken place, until stabilisation is reached.

As a simple example for global satisfaction, we show the result of constructing all models for $\langle a\rangle(\mathtt{p} \wedge \langle b\rangle\mathtt{p})$.



(The starred ovals indicate points which are connected to all other points in the picture.)

Decidability for Branching Time

It is almost obvious how to extend both of the above approaches for branching time, i.e., for tree models in which every point can have arbitrarily many successors for each accessibility relation.

In the "local approach", we build a tree of trees: Again, we start with the set $w_0, ..., w_n$ of all propositionally maximal consistent sets which include Φ. By backtracking, we try to extend one of these to a tree model. Given a propositionally maximal consistent set w_i, if it does not contain any formula $\langle a\rangle\psi$, this branch of the tree is finite. In contrast to linear models, the set $\{\langle a\rangle\psi_1, \langle b\rangle\psi_2\}$ can be satisfied by constructing *two* successor nodes. In general, if w_i contains diamond-formulas $\psi_1, ..., \psi_n$, there will be n successor nodes of w_i in the constructed tree model. Let

$S(w_i)$ be the set of all n-tuples $(\Psi_1, ..., \Psi_n)$ of propositionally maximal consistent sets satisfying

If $\psi_j = \langle a\rangle\psi$, then Ψ_j is any maximal consistent extension of $\psi \cup \{\psi' \mid [a]\psi' \in w_i\}$.

Every n-tuple in $S(w_i)$ determines a set of possible tree-successors of w_i in the constructed tree; therefore we have to recurse and backtrack on *all* elements of *all* of these n-tuples. Formally, a leaf w_i is open if it is included in some node above, or if for *some* n-tuple $(\Psi_1, ..., \Psi_n) \in S(w_i)$, *all* Ψ_i are open.

To formulate this procedure with tableaus, we use *non-determined* tableau rules: For any given formula, there can be both successful and unsuccessful tableaus. The formula is satisfiable, if there is at least one successful tableau. (Thus, in the worst case, all possible tableaus have to be checked.) We only give the rules which are different from above:

$$(\rightarrow_1)\ \frac{\Gamma, (\psi_1 \rightarrow \psi_2)}{\Gamma, \neg\psi_1} \qquad (\rightarrow_2)\ \frac{\Gamma, (\psi_1 \rightarrow \psi_2)}{\Gamma, \psi_2}$$

$$(\langle\rangle)\ \frac{\Gamma, \Psi}{\Phi, \Psi_1 \quad \dots \quad \Phi, \Psi_n}$$

where $\Psi = \{\langle a_1\rangle\psi_1, ..., \langle a_n\rangle\psi_n\}$, no $\langle a\rangle\psi$ is in Γ, and $\Psi_i \triangleq \{\psi_i\} \cup \{\psi \mid [a_i]\psi \in \Gamma\}$.

It seems to be much easier to abridge the 'global' algorithm for satisfiability testing to the branching paradigm. Again, we start with the set W of all propositionally maximal consistent sets and the universal relation for any $\langle a\rangle$ operator, and delete bad arcs and bad nodes until stabilisation is reached. Here, bad arcs are pairs $(w_0, w_1) \in \mathcal{I}(a)$ such that w_0 contains $[a]\psi$, but it is not the case that $\psi \in w_1$. Bad nodes w_0 contain a formula $\langle a\rangle\psi$, but there does not (or no longer) exist a tuple $(w_0, w_1) \in \mathcal{I}(a)$ with $\psi \in w_1$.

Hence, on the multimodal level, it appears that for natural models the local algorithm is simpler, whereas for tree models the global algorithm is easier to implement. In fact, many automatic provers for linear time are tableau based, and many provers for branching time are global. However, the average complexity in both cases depends largely on the structure of the formulas which are to be proven.

Tableaus for LTL

Can we extend these methods for transitive closure operators?

Firstly, in the definition of extended subformula, all $\circ\varphi$ should be regarded as extended subformula of the formula $\Diamond\varphi$. Furthermore, we define also $\circ\Diamond\varphi$ to be an extended subformula of $\Diamond\varphi$. We have to be a bit careful to avoid a nonterminating recursion in the algorithmic reformulation of the recursive definition:

- $SF(\bot) \triangleq \bot$
- $SF((\varphi \rightarrow \psi)) \triangleq \{(\varphi \rightarrow \psi)\} \cup SF(\varphi) \cup SF(\psi)$
- $SF(\mathtt{p}) \triangleq \{\mathtt{p}\}$
- $SF(\circ\varphi) \triangleq \{\circ\varphi\} \cup SF(\varphi)$
- $SF(\Diamond\varphi) \triangleq \{\Diamond\varphi, \circ\varphi\} \cup SF(\varphi)$
- $TSF(\varphi) \triangleq SF(\varphi) \cup \{\circ\Diamond\psi, \circ\neg\Diamond\psi \mid \Diamond\psi \in SF(\varphi)\}$
- $NSF(\varphi) \triangleq \{\neg\psi \mid \psi \in TSF(\varphi)\}$
- $ESF(\varphi) \triangleq SF(\varphi) \cup NSF(\varphi)$
- $ESF(\Phi) \triangleq \bigcup\{ESF(\varphi) \mid \varphi \in \Phi\}$

Now, this definition of extended subformula guarantees that for any $\Diamond\psi$ in w_i, all possible successors will contain $\Diamond\psi$ or $\neg\Diamond\psi$ as well.

For **LTL** on natural models, we try to construct a linear path through the space of all propositionally maximal consistent sets by depth-first-search. If a node w_i contains some formula $\Diamond\psi$, but no formula $\circ\psi'$, we can discard it, because the eventuality $\Diamond\psi$ is not fulfilled. Also, if some candidate successor w_i' of node w_i with $\Diamond\psi \in w_i$ contains $\neg\Diamond\psi$ and $\neg\psi$, we can discard w_i'. But, if w_i' contains $\Diamond\psi$ again, this *unsatisfied eventuality* could be propagated, resulting in a cycle where the fulfilment of $\Diamond\varphi$ is infinitely delayed. The solution is to require that a backward loop only can be regarded as open, if for any $\Diamond\psi$ which occurs in any w_i in the loop, there must be a w_j in the loop such that $\psi \in w_j$.

For the tableau, we add the rules:

$$(\circ)\ \frac{\Gamma, \circ\psi}{\Gamma, \langle a_1\rangle\psi \quad \dots \quad \Gamma, \langle a_n\rangle\psi}$$

$$(\neg\circ)\ \frac{\Gamma, \neg \circ \psi}{\Gamma, \neg\langle a_1\rangle\psi, ..., \neg\langle a_n\rangle\psi}$$

$$(\Diamond)\ \frac{\Gamma, \Diamond\psi}{\Gamma, \circ(\psi \vee \Diamond\psi)} \qquad (\neg\Diamond)\ \frac{\Gamma, \neg\Diamond\psi}{\Gamma, \neg \circ (\psi \vee \Diamond\psi)}$$

These rules are based on the unfolding of the $\circ$- and $\Diamond$-operators:

- $\circ\psi \leftrightarrow \bigvee_a \langle a\rangle\psi$
- $\Diamond\psi \leftrightarrow \circ(\psi \vee \Diamond\psi)$

Additionally we have to require that a leaf is only called open, if for all formulas $\Diamond\psi$ occurring in it, the formula ψ is contained in some node between the leaf and its subsuming ancestor (*loop condition*).

As an example for the loop condition, we show that the formula $\Diamond\bot$ is unsatisfiable:

$$\cfrac{\cfrac{\Diamond\bot}{\circ(\bot \vee \Diamond\bot)}}{\cfrac{\langle a_1\rangle(\bot \vee \Diamond\bot)}{\cfrac{\bot \vee \Diamond\bot}{\cfrac{\bot}{*} \quad \cfrac{\Diamond\bot}{*}}} \quad \dots \quad \cfrac{\langle a_n\rangle(\bot \vee \Diamond\bot)}{\cfrac{\bot \vee \Diamond\bot}{\cfrac{\bot}{*} \quad \cfrac{\Diamond\bot}{*}}}}$$

Each left branch closes because of rule ($\bot$), each right branch is closed because it forms a loop with unsatisfied eventuality $\Diamond\bot$.

Equally, for $\mathcal{U}(\psi_1, \psi_2)$ and $\neg\mathcal{U}(\psi_1, \psi_2)$, there are two tableau rules based on the fixed point unfolding of the $\mathcal{U}$-operator:

- $\mathcal{U}(\psi_1, \psi_2) \leftrightarrow \circ(\psi_1 \vee \psi_2 \wedge \mathcal{U}(\psi_1, \psi_2))$

There is a close connection between the tableau decision procedure and ω-automata: The pre-states in the tableau can be seen as states of a generalised Büchi-automaton. The set of open leafs forms the acceptance condition, and the recurrence condition is given as follows: For every subformula $\mathcal{U}(\psi_1, \psi_2)$, either it is infinitely often not contained in the accepting run, or ψ_1 is contained infinitely often. This can be formulated as generalised Büchi-acceptance condition.

Satisfiability for CTL

To conclude, we quickly sketch the necessary modifications for the **CTL**-transitive closure operators $A\mathcal{U}$ and $E\mathcal{U}$ in the global satisfiability procedure. In modal logic, a 'bad node' was defined to be one which contains $\langle a\rangle\psi$, but no a-successor has ψ. For **CTL**, a node is bad, if it *contains* $A\mathcal{U}(\psi_1, \psi_2)$ or $E\mathcal{U}(\psi_1, \psi_2)$, but does not *satisfy* this requirement. To check this, for all such subformulas we iteratively mark all nodes whether they satisfy the respective requirement:

A node satisfies $A\mathcal{U}(\psi_1, \psi_2)$, if all successors satisfy ψ_1 or ψ_2 and $A\mathcal{U}(\psi_1, \psi_2)$. A node satisfies $E\mathcal{U}(\psi_1, \psi_2)$, if some successor satisfies ψ_1 or ψ_2 and $E\mathcal{U}(\psi_1, \psi_2)$.

Thus, to delete all bad nodes for the subformula $A\mathcal{U}(\psi_1, \psi_2)$, we mark all good nodes which satisfy this formula: Initially, we mark all nodes for which all successors (if any) contain ψ_1. Then we continue to mark all nodes for which all successors contain ψ_1, or contain ψ_2 and are already marked. After stabilisation is reached, we can delete all unmarked nodes which contain $A\mathcal{U}(\psi_1, \psi_2)$.

Analogously, for the subformula $E\mathcal{U}(\psi_1, \psi_2)$ we initially mark all nodes which have a successor containing ψ_1. Then we repeat until stabilisation to mark all nodes which have a successor containing ψ_1, or containing ψ_2 and being already marked. Again, after stabilisation all unmarked nodes containing $E\mathcal{U}(\psi_1, \psi_2)$ are no good.

7 Basic Model Checking Algorithms

In this section, we will show how the most commonly used model checking procedures can be obtained from the satisfiability algorithms we considered above.

Decision procedures can be used to construct a proof or counter-model for any correctness statement φ about a model $\mathcal{M}$ by encoding $\mathcal{M}$ as a set of assumptions (or premisses, or program axioms) Φ, and deciding whether $\Phi \vdash \varphi$. However, some experiments will quickly convince you that a naïve approach of doing so is doomed to failure. Usually, the program axioms all have a very special form, such as

$$\texttt{state\_i} \rightarrow (\circ\texttt{succ\_i1} \vee \cdots \vee \circ\texttt{succ\_in})$$

in a linear time modelling, or

$$\texttt{state\_i} \rightarrow (\langle a_1\rangle\texttt{succ\_i1} \wedge \cdots \wedge \langle a_n\rangle\texttt{succ\_in})$$

in a branching time approach. The decision procedure in general can not take advantage of this special form of the assumptions, and will in every step break down all assumptions to its basic propositional components. This results in a very inefficient behaviour; usually only very small systems can be verified and debugged that way.

Therefore, model checking algorithms avoid the encoding of the models as a set of program axioms; they use the models directly instead. Model checking tries to determine whether a given specification formula is satisfied in a given Kripke model, i.e., whether a natural or tree model satisfying the formula can be generated from it.

There are two variants of this task, depending on whether the universal or initial definition of satisfaction of a formula in a model is used. In the initial definition, a Kripke model, consisting of universe U, transition relation(s) defined by $\mathcal{I}$, and initial point $w_0 \in U$, is given, and we have to check whether the formula φ is satisfied: $(U, \mathcal{I}, w_0) \models \varphi$. In the universal definition, we are given universe and transition relation, and want to know whether the formula is satisfied in *all* points w of the universe: $(U, \mathcal{I}) \models \varphi$ iff for all $w_0 \in U$ it holds that $(U, \mathcal{I}, w_0) \models \varphi$; Equivalently, we want to know whether the set of points satisfying the formula is the whole universe; therefore we need an algorithm which calculates the set of points $S(U, \mathcal{I}, \varphi) \stackrel{\triangle}{=} \{w \mid (U, \mathcal{I}, w) \models \varphi\}$ in which a given formula is satisfied.

Of course, any algorithm which calculates $S(U, \mathcal{I}, \varphi)$ can also be used to decide whether $(U, \mathcal{I}, w_0) \models \varphi$ holds: $(U, \mathcal{I}, w_0) \models \varphi$ iff $w_0 \in S(U, \mathcal{I}, \varphi)$. Vice versa, if we have an efficient algorithm for $(U, \mathcal{I}, w_0) \models \varphi$, we can calculate $S(U, \mathcal{I}, \varphi)$ by an iteration on all states. Thus, in theory there is not much difference between both variants; practically, the initial definition corresponds more or less to the local methods for satisfiability, whereas the universal definition fits more to the global methods.

However, in both cases we usually are given a set of *global assumptions* Φ together with the Kripke-model. These global assumptions are assumed to hold in the whole model; i.e., they restrict the model to those parts where they are valid. For example, an ω-automaton can be regarded as a Kripke model, together with global eventuality and fairness assumptions (acceptance and recurrence set). Global assumptions can be formulated in the same language in which the formula to be checked is specified; however, there have been suggested "mixed" approaches, where e.g. the global assumptions are described in **LTL** and the property is described in **CTL**.

Local Modal Model Checking

Given a Kripke model $\mathcal{M} = (U, \mathcal{I}, w_0)$, a finite set Φ of **ML**-formulas as global assumptions, and a property φ given as **ML**-formula. Model checking means to answer the question: "Does every natural model generated from $\mathcal{M}$ in w_0 which globally satisfies Φ also satisfy φ in w_0?" Or, equivalently, "Is there a sequence generated from $\mathcal{M}$ in w_0 which satisfies Φ in every point, and satisfies $\neg\varphi$ in w_0?"

There is a pitfall here which can easily be overlooked: Without additional assumptions, any set of formulas containing only formulas in which each outmost modal operator is a $[a]$-operator is immediately satisfiable! This is because we did *not* require sequences generated from the Kripke-model to be *maximal*; any one-point sequence is a legally generated natural model. To restrict the model checking procedure to infinite sequences, an assumption $\bigvee_a \langle a\rangle\top$ has to be added.

A local model checking procedure for multimodal logic can be obtained by building the product of the set of propositionally maximal consistent sets with the set of states of the model. Similar to our proceeding in the decision procedure, we restrict attention to those propositionally maximal consistent sets m which include Φ.

Formally, an *atom* α is any pair (w, m), where $w \in U$

is a point, and $m \subseteq ESF(\varphi)$ is a locally maximal consistent set including Φ, and w and m agree on the interpretation of propositions. That is, if $\mathtt{p} \in ESF(\varphi)$, then $\mathtt{p} \in m$ iff $w \in \mathcal{I}(\mathtt{p})$.

An *initial atom* is any atom $\alpha = (w_0, m_0)$, where w_0 is the initial state of $\mathcal{M}$, and $\neg\varphi \in m_0$. For each $a \in \mathcal{R}$, we define a relation X_a between atoms: $X_a((w,m),(w',m'))$ iff

1. $(w,w') \in \mathcal{I}(a)$,
2. if $\neg\langle a\rangle\psi \in m$, then $\neg\psi \in m'$,
3. if $\langle a\rangle\psi \in m$, then $\psi \in m'$,
4. for no $b \neq a$ is some $\langle b\rangle\psi \in m$.
5. some $\langle a\rangle\psi \in m$.

The first condition reflects the fact that the steps in the generated sequence are predetermined by the Kripke model (system to be verified). The second condition is imposed by the semantics of the $[a]$-operator; the third and fourth condition are reformulations of the axioms **(U)** and **(L)**, respectively, and the corresponding tableau rules $(\langle\rangle)$ and (L). The fifth condition allows the generated sequence to be finite when all diamonds are killed, even if the model has continuation states in it.

Now we can construct a forest of atoms as follows:

- initial nodes are all initial atoms
- any node α has as successors all α' such that $X_a(\alpha,\alpha')$

Since for any finite Kripke-model there are only finitely many atoms, each branch in this forest can be made finite by appropriate backward arcs. Similar to the tableau definition, a leaf is called *open*, if it has no $\langle a\rangle$ formulas in its m-component; otherwise, it is *closed*.

An *accepting path* through the resulting structure starts with any initial node and is either infinite or ends with an open leaf. Any accepting path is a sequence generated from the Kripke model which satisfies the given formula $\neg\varphi$, thereby forming a counterexample to the specification φ.

To implement the search for an accepting path, we perform a depth-first search with backtracking from the set of initial atoms to all of its X-successors. In order to be able to terminate loops in this search, we have to store all atoms which were encountered previously. Though there are several possibilities to represent such a set of atoms, the method of choice seems to be to employ a hash table. It is not necessary to use all components of m as hash indices, since boolean combinations of formulas can be recovered from their constituent parts.

In the depth-first search, we have to remove closed atoms from the list of possible loop points. A better way is to *mark* these nodes as closed while backtracking; then the search will not recurse again if such an atom reappears.

In general, since we are only looking for *some* counter-model, we can terminate the search if a counter-model is found. Although in the worst case (if no counter-model exists) the whole forest must be searched, there is a chance to find errors very quickly by an appropriate ordering of the depth-first search successors.

Extensions for LTL

We have seen that the local model checking algorithm for multimodal logic is almost the same algorithm as the local tableau decision procedure. Similarly, the local model checking for **LTL** is very close to its respective satisfiability algorithm. Again, we first give the version for a logic which contains $\circ$ and $\Diamond$-operators, and then show how to extend this for $\mathcal{U}$-operators.

Also, we do not consider additional assumptions in this subsection, since with $\Box$-operator we can dispose any finite set of **LTL**-assumptions: $\Phi \mid\!\!\vdash \varphi$ iff $\mid\!\!\vdash \Box\Phi \rightarrow \varphi$.

In the definition of $X_a((w,m),(w',m'))$ we additionally require

6. if $\Diamond\psi \in m$, then $\psi \in m'$ or $\Diamond\psi \in m'$
7. if $\neg\Diamond\psi \in m$, then $\neg\psi \in m'$ and $\neg\Diamond\psi \in m'$

Similar as for modal logic, we try to thread an accepting path through the graph of atoms which arises from this definition. A leaf is *open*, if it neither contains any $\circ\psi$, nor any $\Diamond\psi$. In contrast to modal logic, however, we can only accept a path in which in addition to the 'killing' of all $\circ\psi$ by the next-step relation each occurrence of a $\Diamond\psi$ is eventually 'killed'. And, unlike in the tableau decision method, we can not guarantee that several diamonds are simultaneously satisfied in some single loop. For example, an atom α might contain $\Diamond\psi_1$, $\Diamond\psi_2$, some successor α_1 of α contains ψ_1 but not ψ_2, and some other successor α_2 of α contains ψ_2 but not ψ_1. Hence, a loop from α_1 to α is non-accepting, because it does not fulfil the eventuality ψ_2, and a loop from α_2 to α is non-accepting, because it does not fulfil the eventuality ψ_1. But, of course, any path which infinitely often passes through both α_1 and α_2 is accepting.

In general, we have to consider *maximal strongly connected components* of the transitive closure of all X_a. A maximal SCC is a set g of atoms, such that for each two α, $\alpha' \in g$ there exists a finite path $\alpha, \alpha_1, \ldots, \alpha_n, \alpha'$ connecting α and α', and all α which can be reached from g and lead into g are contained in g. Call such a maximal SCC g *self-fulfilling*, if for any $\Diamond\psi$ in some $\alpha \in g$ there exists some $\alpha' \in g$ with $\psi \in \alpha'$. Now an *accepting path* is one which starts in some initial atom and either ends in an open leaf or infinitely often passes through all atoms of a self-fulfilling maximal SCC.

Adequacy of this procedure: $\neg\varphi$ is satisfiable in the model iff there exists such an accepting path.

For $\mathcal{U}$-operators, each positive occurrence $\mathcal{U}(\psi_1,\psi_2)$ is an eventuality which has to be fulfilled at some point; thus the SCC g is defined to be self-fulfilling, if for any $\mathcal{U}(\psi_1,\psi_2)$ in some $\alpha \in g$ there exists some $\alpha' \in g$ with $\psi_1 \in \alpha'$.

How can we construct maximal SCCs, and decide whether they are self-fulfilling? There are two different algorithms known in the literature. For model checking, Tarjan's algorithm is particularly well-suited, since it enumerates the strong components of a graph during the backtrack from the depth-first search. Thus model checking can be performed "online" during the enumeration of the reachable state space of the model.

In this algorithm the function `successors` constructs for a given atom `alpha` the set of all possible successor atoms according to the transition relation of the Kripke model and to the fixed point definition

```
PROGRAM LTL_check(Model m, Formula phi) =
  Nat depth_first_count := 0;
  Atomset stack := {};
  Natarray table;
  Atomset init := {alpha |
    alpha is an initial atom of m and phi};
  FOR ALL (alpha IN init) DO
    depth_first_search (alpha)
  ENDDO;
  print (``phi is not satisfiable in m'');

PROCEDURE depth_first_search (Atom alpha) =
  IF (table [alpha] = UNDEFINED)
    Nat dfnumber := depth_first_count;
    depth_first_count:=depth_first_count+1;
    table [alpha] := dfnumber;
    push (stack, alpha);
    Atomset succ := successors (alpha);
    FOR ALL (beta IN succ) DO
      depth_first_search (beta);
      table [alpha] :=
        min(table [alpha], table [beta])
    ENDDO;
    IF (table [alpha] = dfnumber)
      Formulaset required := {},
                 fulfilled := {};
      REPEAT
        beta := pop (stack);
        table [beta] :=MAXNAT;
        required  := required  + {psi_1 |
              Until(psi_1, psi_2) in beta};
        fulfilled := fulfilled + beta
      UNTIL (alpha = beta);
      IF required <= fulfilled
        print (``phi satisfiable in m'');
        EXIT;
      ENDIF
    ENDIF
  ENDIF
```

Figure 1. Depth–first–search **LTL** *model checking algorithm*

8. $\mathcal{U}(\psi_1, \psi_2) \in w$ iff
$\psi_1 \in w'$ or $\psi_2 \in w'$ and $\mathcal{U}(\psi_1, \psi_2) \in w'$.

The procedure `depth_first_search` builds recursively all atoms reachable from a given atom `alpha`; upon backtrack `alpha` is found to be the root of a maximal SCC iff there are no atoms `beta` in the subtree below `alpha` such that `alpha` is also in the subtree of `beta`. In this case the maximal SCC containing `alpha` consists of all nodes in the subtree below `alpha`, and this maximal SCC can be checked for acceptance. `table` is implemented as a hash table from atoms to natural numbers. `table [alpha]` contains

- undefined, as long as atom `alpha` has not occurred,
- the depth-first-number of `alpha`, upon first encounter of `alpha`,
- the depth–first–number of the first encountered atom belonging to the same strongly connected component as `alpha`, after return from the recursive call, and
- `maxnat` (i.e. a value for which the test `n < maxnat` is always true), after the maximal strong component containing `alpha` has been analysed.

The main program calls `depth_first_search` for all initial atoms. If during the construction of the atom graph a maximal final SCC is found, the algorithm reports success; if the whole graph is searched without success we know that the formula is not satisfiable, and the program terminates with this result.

The complexity of this algorithm is exponential (PSPACE) in the number of $\mathcal{U}$-subformulas, because every set of such subformulas determines a locally maximal consistent set, and linear (NLOGSPACE) in the size of the Kripke model. The exponential complexity in the length of the formula is not very problematic, because specification formulas tend to be rather short. The linear complexity in the size of the model is a more serious limiting factor, since in the worst case (i.e., if the formula is unsatisfiable) all states have to be traversed. Current technology limits the applicability of such algorithms to models with approximately $10^5 - 10^6$ reachable states. In the last section we will discuss approaches which try to overcome this limit.

Global Branching Time Model Checking

We omit to detail the modification which are necessary to adapt the local modal model checking algorithm to branching time logics. Instead, we review the global model checking algorithm for **CTL**.

For a given Kripke model, branching time model checking answers the question: "Does the maximal tree model generated from the Kripke model $\mathcal{M}$ in any state w_0 satisfy the formula φ?" In some sense, this is an easier question than the one for linear time, because *the* maximal tree model generated from $\mathcal{M}$ is uniquely determined, whereas each Kripke model generates a *set* of natural models, and we want to check all of these.

Put differently, the value of each **CTL**-formula in each state of the generated tree model is uniquely determined. In the linear interpretation, from any given point in the Kripke model there is a *set* of generated sequences; therefore e.g. the formula $\Diamond\psi$ can both be true for one of them, and false for another one. Thus we used the product of the universe and the set of subformulas in the search for a counter-sequence. In branching time, any subformula $EF\psi$ or $AF\psi$ in any point of a Kripke model is either true or false, because there is only one maximal tree model generated from this point.

Consequently, the **CTL** model checking algorithm proceeds by marking each point with the set of subformulas valid for this point. I.e., to label the set of points satisfying $\varphi \triangleq E\mathcal{U}(\psi_1, \psi_2)$, suppose we have already labelled the set of points satisfying ψ_1 and those satisfying ψ_2. We use the fixpoint unfolding $E\mathcal{U}(\psi_1, \psi_2) \leftrightarrow EX(\psi_1 \vee \psi_2 \wedge E\mathcal{U}(\psi_1, \psi_2))$ and repeat until stabilisation to label all nodes with φ which have a successor which is labelled with ψ_1 or with ψ_2 and with φ.

Similarly for $\varphi \triangleq A\mathcal{U}(\psi_1, \psi_2)$ we repeat until stabilisation to label all nodes with φ for which all successors are labelled with ψ_1 or with ψ_2 and with φ. A recursive formulation of this algorithm is given below.

Since the Kripke model has a finite number of points, each `REPEAT` stabilises after at most $|U|$ passes. In the

```
PROGRAM CTL_check (Model m, Formula phi) =
  IF m.w_0 IN eval(phi)
  THEN print (''phi is satisfied in m'')
  ELSE print (''phi not satisfied in m'');

PROCEDURE eval (Formula phi): Nodeset =
  CASE phi OF
    p :  RETURN m.I(p);
    false :  RETURN {};
    (psi_1 -> psi_2) :
      RETURN m.U-eval(psi_1)+eval(psi_2);
    EUntil(psi_1, psi_2) :
      epsi_1 := eval(psi_1);
      epsi_2 := eval(psi_2);
      ephi := {};
      REPEAT UNTIL STABILIZATION
        ephi := ephi + {s |
          succ(s)*(epsi_1+epsi_2*ephi)#{}};
      RETURN ephi;
    AUntil(psi_1, psi_2) :
      epsi_1 := eval(psi_1);
      epsi_2 := eval(psi_2);
      ephi := {};
      REPEAT UNTIL STABILIZATION
        ephi := ephi + {s |
          succ(s)<=epsi_1+epsi_2*ephi};
      RETURN ephi;
```

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

worst case, each pass searches the whole model, hence the complexity is linear in the number of different subformulas, and cubic in $|U|$.

However, this bound can be improved if the search is organised better. Clarke, Emerson and Sistla gave an algorithm which is linear in the size of the model as well. For the EF-operator, the problem of marking all nodes for which $EF\varphi$ holds, given the set of node satisfying φ, is equivalent to the *inverse reachability problem*: Given a set of nodes, mark all nodes from which any finite path leads into the given set. Assuming that for any two nodes 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 nodes.

```
PROCEDURE reach (nodeset target): nodeset =
  source := {}; search := target;
  WHILE search # {} DO
    search := pred (search) - source;
    source := source + search
  ENDDO;
  RETURN source;
```

Every node 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 nodes; thus the overall complexity is quadratic in $|U|$, i.e., linear in the size of the Kripke model.

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

$$A\mathcal{U}(\psi_1, \psi_2) \leftrightarrow \neg(E\mathcal{U}(\neg(\psi_1 \wedge \psi_2), \neg\psi_1) \vee EG\neg\psi_1)$$

Thus, we only need a procedure marking all nodes for which $EG\varphi$ 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 nodes in the original model from which a non-trivial SCC or a node without successors can be reached by a path in the restricted model.

Fairness Constraints

Some automated model checkers for **CTL** allow to restrict the path-quantifiers A and E to *fair* paths, which satisfy certain fairness constraints specified in **LTL**. *Simple* fairness constraints are of form $\Diamond\psi$, where ψ is a boolean combination of propositions. E.g., the maximality condition $\Diamond\top$ of the preceding subsection was of this type. As another example for a simple fairness constraint, we might want to restrict our attention to execution sequences in which every component is always eventually scheduled. Streett fairness constraints are of form $(\Box\Diamond\psi_1 \rightarrow \Box\Diamond\psi_2)$ and e.g. are useful to restrict attention to *strongly fair* schedulers: if a component infinitely often requests a resource, it will be granted infinitely often. The above algorithm can be modified to deal with such fairness constraints by building the tableau of the **LTL**-assumption, and checking the **CTL**-formula on the product of Kripke model and tableau.

Model Checking for μ-calculus

Both the local and the global model checking algorithms can be easily adapted to (monotonic) μ**TL**. For the global version, an algorithm is given in section 9. For the local version, there have been a number of algorithms proposed in the literature, which are all more or less similar to each other; see [10]. In general, the model checking problem for μ**TL** is exponential in the size of the formula (alternation depth); however, the proposed algorithms may vary in their average case performance.

8 Modelling of Reactive Systems

Up to now, we regarded a system as being given as a single Kripke model.

However, real-life systems are usually composed of a number of smaller subcomponents. Even if the target system is a single sequential machine, it is of advantage to model it as a set of processes running in parallel:

- usually the *functionality* suggests a certain decomposition into modules; sequentialization is not the primary issue in the design;
- certain subcomponents (e.g. hardware components) actually are independent of the rest of the system, and therefore conceptually parallel,
- the *environment* can be seen as a process running in parallel to the system
- software-reusability and object-oriented design requires modularity

Message Passing vs. Shared Variables

Hence, we have to consider systems of parallel processes, and the synchronisation between these processes. There are two main paradigms of parallel systems: *distributed* systems, where the subcomponents are seen as spatially apart from each other, and *concurrent* systems, where the subcomponents use common resources such as processor time or memory cells.

Consequently, there are two main paradigms for synchronisation between parallel processes: via *message passing* (for distributed systems), and via *shared variables* (for concurrent systems).

Of course, there is no clear distinction between distributed and concurrent programs. It is not possible to formalise the concept of being spatially apart, since this is dependent on one's own point of view: from South Africa, all computers in the local area network `informatik.tu-muenchen.de` can be regarded as a single system, whereas from the processor's viewpoint a hard disk controller can be regarded as a remote subsystem. On the other side, every component of a distributed system shares *some* resource with *some* other component; if it were totally unrelated it would not make sense to regard it as being part of one system.

Consequently, from a certain point of view, passing a message between process A and B can be seen as process A writing into a shared variable which is read by B. On the other side, writing a shared variable can be seen as sending to all other processes which might use this variable the message that its value has changed. In fact, this transition from the message passing paradigm to an implementation via shared variables occurs in every network controller; and the transition from the shared variables paradigm to an implementation via message passing occurs in every distributed cache.

However, different paradigms produce different techniques; many parallel programming languages and many verification systems support only one of these two paradigms.

Synchronous vs. Asynchronous Communication

Another paradigm, which applies mainly (but not solely) to the message passing approach is the question of synchronous versus asynchronous interaction between the parallel components. In the synchronous approach a partner wishing to communicate is blocked until a communication partner is willing to participate in the communication. In the asynchronous approach each process decides whether it wants to wait at a certain point or not; usually some kind of buffering mechanism is used for messages which are not needed immediately.

Synchronous communication can be seen as a special case of asynchronous communication where the length of each buffer queue is limited to one, and each process decides to wait after writing into or before reading from that queue until the queue is empty or full again, respectively.

Vice versa, a buffer can be seen as a separate process in a synchronous system which is always willing to communicate with other processes. If the size of the buffer is unbounded, the system is not finite state. Even if their size is bounded, the buffers can be the biggest part of the translation of an asynchronous system.

Examples of synchronous modelling formalisms are (parallel) transition systems, Petri nets, CCS, CSP, and its variants, semaphores and monitors, critical regions and so on. Examples of asynchronous formalisms are protocol specification languages such as SDL and Lotos.

Some Concrete Formalisms

We already mentioned that a *transition system* is basically a finite automaton without acceptance or recurrence condition. Formally, a transition system is a tuple (Σ, S, Δ, s_0), where

- Σ is a nonempty finite *alphabet*,
- S is a nonempty finite set of *states*,
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*, and
- $s_0 \in S$ is the *initial state*.

A *parallel transition system* is a tuple $T = (T_1, ..., T_n)$ of transition systems, such that $S_i \cap S_j = \emptyset$. The *global* transition system T *associated with* a parallel transition system $(T_1, ..., T_n)$ is defined by $T = (\Sigma, S, \Delta, s_0)$, where

- $\Sigma = \bigcup \Sigma_i$
- $S = S_1 \times \cdots \times S_n$
- $s_0 = (s_{10}, ..., s_{n0})$, and
- $((s_1, ..., s_n), a, (s'_1, ..., s'_n)) \in \Delta$ iff for all T_i
  — if $a \in \Sigma_i$, then $(s_i, a, s'_i) \in \Delta_i$, and
  — if $a \notin \Sigma_i$, then $s_i = s'_i$

Thus, we model parallelism by interleaving, and synchronisation by the common alphabet. A parallel transition system can be seen as a very restricted kind of CSP process, with a fixed number of parallel subprocesses. The size of the state space of the global transition system is the product of the sizes of all parallel components.

An *elementary Petri net* is a tuple $N = (P, T, F, s_0)$, where

- P is a finite set of *places*,
- T is a finite set of *transitions* ($P \cap T = \emptyset$),
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, and
- $m_0 \subseteq P$ is the *initial marking* of the net.

A *marking* m of the net is any subset of P. By $\bullet t \triangleq \{p \mid (p,t) \in F\}$ and $t\bullet \triangleq \{p \mid (t,p) \in F\}$ we denote the *preset* and the *postset* of transition t, respectively. A transition t is *enabled* at marking m if $\bullet t \subseteq m$ (all its input places are occupied at m) and $t\bullet \cap m \subseteq \bullet t$ (all its output places are empty at m, or they are also input places). Marking m' is the *result of firing* transition t from marking m, if t is enabled at m and $m' = (m \setminus \bullet t) \cup t\bullet$.

For every elementary Petri net there is an associated transition system: The alphabet is the set of transitions, the state set is the set of markings, the initial state is the initial marking, and $(m, t, m') \in \Delta$ iff m' is the result of firing t from m. However, the number of states in the transition system is exponential in the number of places of the net.

Vice versa, every parallel transition system can be formulated as an elementary Petri net: Exercise!

A *shared variables program* is a tuple (V, D, T, s_0), where

- $V = (v_1, ..., v_n)$ is a set of *variables*,
- $D = (D_1, ..., D_n)$ is a tuple of corresponding finite *domains* $D_i = \{d_{i1}, ..., d_{im_i}\}$
- $T \subseteq D \times D$ is a *transition relation*, and
- $s_0 = (d_{11}, ..., d_{n1})$ is the *initial state*.

A *state* of a shared variables program is a tuple $(d_1, ..., d_n)$, where each $d_i \in D_i$. Thus the number of states in a shared variables program is the product of the size of all domains. Usually the transition relation T is given as a propositional formula φ_T over $\mathcal{P} = \{(x = y) \mid x, y \in (V \cup V' \cup \bigcup D_i)\}$, where $V' = \{v'_1, ..., v'_n\}$. If $s = (d_1, ..., d_n)$ and $s' = (d'_1, ..., d'_n)$, then $(s, s') \in T$ iff $\mathcal{I} \models \varphi_T$, where $\mathcal{I}(v_i) = d_i$ and $\mathcal{I}(v'_i) = d'_i$.

For every elementary Petri net or parallel transition system there is an equivalent shared variables program of the same order of size. The translation in the other direction requires the evaluation of propositional formulas and thus can involve an exponential blowup. Using relational semantics, a shared variables program can be obtained for almost all other models for concurrency. Therefore, shared variable programs are widely used to model reactive systems.

9 Symbolic Model Checking

For any shared variables program, we can obtain an equivalent shared variables program which uses only binary domains: $D = \{0, 1\}^n$. To do so, we use an arbitrary binary encoding of domain D_i and introduce for any variable v_i over domain D_i new binary variables $v_{i1}, ..., v_{ik}$, where $k = \lceil \log_2(|D_i|) \rceil$. This encoding can be compared to the implementation of arbitrary data types on digital computers, where each bit can take only two values.

If all variables $V = \{v_1, ..., v_n\}$ of a shared variables program are over a binary domain, then any propositional formula φ over $\mathcal{P} = \{v_1, ..., v_n\}$ describes a set of states of the program, namely the set of all propositional models (interpretations) which validate the formula. Here we assume the substitution 0 for $\mathbf{false}$ and 1 for $\mathbf{true}$. Vice versa, for any set of states there is a propositional formula describing this set. However, this formula is not uniquely determined; the problem of finding a shortest formula describing a given set of states is NP-hard.

Also, the transition relation of a shared variables program with binary variables $V = \{v_1, ..., v_n\}$ can be represented as an ordinary propositional formula over $\mathcal{P} = \{v_1, ..., v_n, v'_1, ..., v'_n\}$. If the transition relation is given as a propositional formula with equalities, we replace 0 by $\bot$, and 1 by $\top$, and $(v = v')$ by $(v \leftrightarrow v')$. E.g., the formula

$$v_1 = 0 \rightarrow ((v'_1 = 1) \wedge (v'_2 = v_2) \wedge (v'_3 \neq v_3))$$

becomes in this notation

$$\neg v_1 \rightarrow (v'_1 \wedge (v'_2 \leftrightarrow v_2) \wedge \neg(v'_3 \leftrightarrow v_3))$$

For a shared variables with n variables over binary domains the size of the state space is 2^n. Therefore e.g. the state space of a buffer of length 10 with values between 1 and 1000 is $2^{100} \simeq 10^{30}$. The *reachable* state space is a subset of this state space, which can be of the same order of magnitude. The transition relation for this buffer consists of pairs of states and therefore has a size of approximately 10^{60}.

To perform global model checking on systems of this or bigger size, we need an efficient representation of large sets.

Binary Decision Diagrams

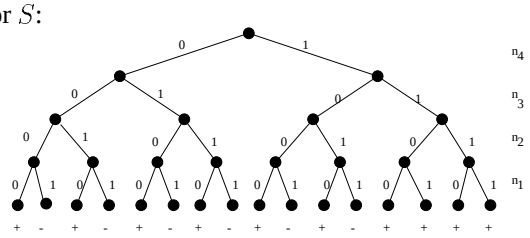
The representation of a set by explicit enumeration of its elements, e.g., as a list or array, can be rather wasteful, since it pays no respect to the internal structure of the set. E.g., given the domain $D = \{0, 1, ..., 15\}$, the explicit representation of the set "all numbers which are even or bigger than 11" is a list of the elements

$$S = \{0, 2, 4, 6, 8, 10, 12, 13, 14, 15\}$$

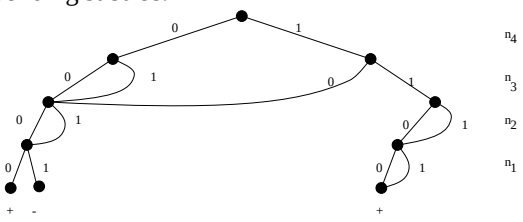
This representation takes $O(|D| \cdot \lceil \log_2(|D|) \rceil)$ memory bits. A much more succinct representation of the same set can be given by a propositional formula over the binary encoding $n = n_4 n_3 n_2 n_1$ of the domain:

$$S = \{n \mid n_1 = O \vee n_4 = L \wedge n_3 = L\}$$

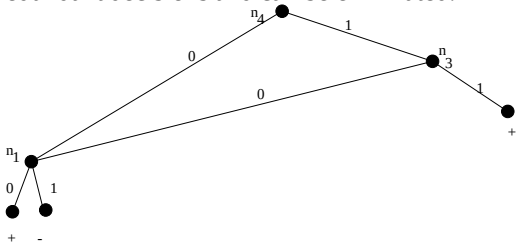
To put this idea into a picture, look at the binary decision tree for S:



This tree is just a transcription of the truth table of S's characteristic function. It has many isomorphic subtrees. For any two isomorphic subtrees it is sufficient to maintain only one copy. We can replace the other one by a link to the corresponding subtree.



In the resulting structure, there are nodes for which both alternatives lead to the same subtree. These nodes represent redundant decisions and can be eliminated.



The resulting structure is called an (ordered) *binary decision diagram* (BDD, [1]). As we see, the BDD can be much more concise than the original decision tree. In fact,

the size of the BDD depends on the *structure* of the represented set rather than on its *cardinality*. For example, the representation of the empty set and the full set are both of constant size 1. Because of this dependence on the structure of the represented object, the description by BDDs is sometimes called *symbolic*, and techniques using BDDs to represent objects are called *symbolic techniques*.

Given a tuple of binary domain variables $V = (v_1, ..., v_n)$, and a set of states given as a propositional formula over V, or the transition relation given as a propositional formula over $V \cup V'$. Can we directly construct a BDD for this formula (we want to avoid the full decision tree)? Call a leaf in the BDD *negative* if it marked by "-", positive otherwise. The BDD for $\bot$ is a negative leaf; BDDs for propositions v_i consist of a node with variable v_i and a negative and positive leaf.

To construct a BDD for $\neg\varphi$ from a BDD for φ is just exchanging of positive and negative leafs. A BDD for $(\varphi \rightarrow \psi)$ from BDDs for φ and ψ can be constructed by recursive descent. We have to consider the cases that either of φ, ψ is a negative or positive leaf, or a variable. If both BDDs for φ and ψ are not leafs, let v be the top variable in both BDDs, or, if the two top variables are different, the smaller one according to the order of variables. Split the problem into two subproblems for $v = \mathbf{false}$ and $v = \mathbf{true}$, respectively, and solve the two subproblems recursively. The result give the 0 and 1 branch of the result node. We do *not* create a new result if both branches are equal (return a common result), or if an equivalent node already exists in the result. To check this latter condition, we maintain a hash table of BDD nodes which is updated upon backtrack.

There are other boolean operations on BDDs which can be implemented with the algorithm sketched above. For example, substitution $\varphi(v/\bot)$ of a proposition v in a formula φ by $\bot$ can be done by assigning a pointer to the negative leaf to v during the backtrack. Boolean quantification $\exists v\ \varphi$ can be reduced to substitution by

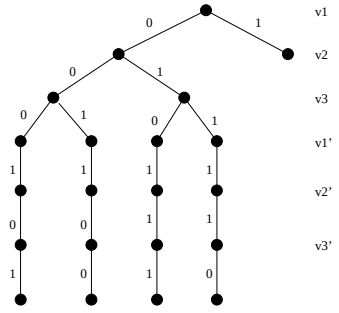
$$(\exists v\ \varphi) \leftrightarrow ((\varphi(v/\bot) \vee \varphi(v/\top)))$$

Since BDDs are a form of deterministic finite automata on finite strings over the alphabet $(0, 1)$, for any given variable ordering and formula there is a unique BDD representing the formula.

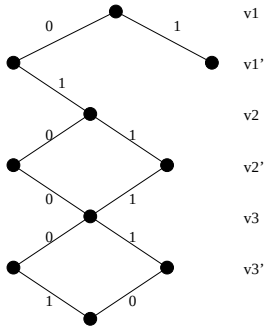
The size of this BDD depends critically on the ordering of the variables. For our above example formula

$$v_1 = 0 \rightarrow ((v_1' = 1) \wedge (v_2' = v_2) \wedge (v_3' \neq v_3))$$

and the variable ordering $(v_1, v_2, v_3, v_1', v_2', v_3')$, the above algorithm yields the following BDD. (We omit all branches leading to negative leafs.)



For the variable ordering $(v_1, v_1', v_2, v_2', v_3, v_3')$, however, we obtain the following much smaller BDD:



This is a common phenomenon when working with BDDs. In general, a good heuristics is to keep "dependent" variables as close together in the ordering as possible. For a more formal treatment in the context of sequential circuits, see [9]. Unfortunately, the problem of finding an *optimal* variable ordering is NP-hard. Basically, for every possible ordering one has to construct the BDD and compare their sizes, which is not feasable. Automatic reordering strategies usually proceed by steepest ascend heuristics.

Symbolic Model Checking for μTL

Recall from section 7 that global model checking for **CTL** unfolds the fixpoint definition of the $A\mathcal{U}$ and $E\mathcal{U}$ operators. If we restrict our attention to *monotonic* μ**TL**-formulas (to be defined below), then this idea can be used to obtain a global model checking algorithm for these formulas. Moreover, this algorithm can directly be implemented using BDDs as representation of transition relation and state sets([3]).

A formula φ is *positive* or *negative* in a propositional variable q, if every occurrence of q in φ is under an even or odd number of negations. Formally, this notion is defined recursively: Any μ**TL**-formula not containing the propositional variable q is as well positive as negative in q, and q is positive in itself. The formula $(\varphi \rightarrow \psi)$ is positive in q, if φ is negative in q and also ψ is positive in q; symmetrically, $(\varphi \rightarrow \psi)$ is negative in q, if φ is positive in q and also ψ is negative in q.

Given any set, the set of all of its subsets forms under the inclusion ordering a complete lattice. A function $f : 2^S \rightarrow 2^S$ is called *monotonic*, if $f(p) \subset f(q)$ whenever $p \subset q$. Any monotonic function has greatest and least fixed points:

$$\mathit{gfp}(f) = \mathit{sup}\{q \mid q = f(q)\} = \mathit{sup}\{q \mid q \subseteq f(q)\}$$
$$\mathit{lfp}(f) = \mathit{inf}\{q \mid q = f(q)\} = \mathit{inf}\{q \mid f(q) \subseteq q\}$$

Moreover, on a finite lattice every monotonic function is continuous, therefore in this case

$gfp(f(q)) = lim_{i\to\omega} f^i(\top)$ and
$lfp(f(q)) = lim_{i\to\omega} f^i(\bot)$

We have already seen, that any formula defines a set of points in the model, namely those points where it is satisfied. Likewise, a formula $\varphi(q)$ with a free propositional variable q can be seen as a function from sets of points to sets of points: it maps any set of states ψ to the set of states satisfying $\varphi(q/\psi)$, that is, φ where every occurrence of q is replaced by ψ.

If φ is positive in q, then it is also monotonic:

$$(p \to q) \vdash (\varphi(p) \to \varphi(q))$$

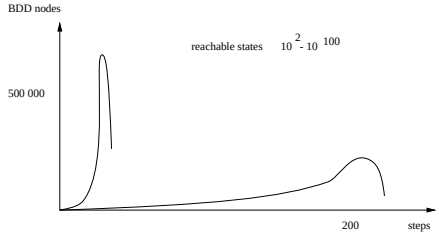
This can be proved by induction on the structure of φ. Therefore, for such φ we have $\nu q\varphi = gfp(\varphi)$ and $\mu q\varphi = lfp(\varphi)$. Consequently, for monotonic formulas model checking can be performed by extending the naïve global algorithm as follows:

eval(φ) =
case φ **of**
  p : **return** $\{s \mid M, s \models p\}$
  $\bot$: **return** $\emptyset$
  $(\psi_1 \to \psi_2)$: **return** $M -$ eval(ψ_1) + eval(ψ_2)
  $\langle a\rangle\psi$: **return** $\{s \mid \exists s' : (s, s') \in a \wedge s' \in$ eval(ψ)$\}$
  $\nu q(\psi) : H := M$
    repeat until stabilization
      $H := eval(\psi(q/H))$
    return H
  $\mu q(\psi) : H := \emptyset$
    repeat until stabilization
      $H := eval(\psi(q/H))$
    return H

In this algorithm, all set operations can be directly performed with BDDs. Calculation of the BDD for $\langle a\rangle\psi$ from the BDD for ψ amounts to calculation the *inverse image* of ψ under the relation a. This is done using propositional quantification: Recall that the BDD for ψ is using variables $(v_1, ..., v_n)$, and the BDD for a is defined over the variables $(v_1, ..., v_n, v'_1, ..., v'_n)$. To get the BDD for $\langle a\rangle\psi$, we first rename all variables v_i in the BDD for ψ by v'_i, then build the intersection of this BDD with the BDD for a to obtain a BDD over $(v_1, ..., v_n, v'_1, ..., v'_n)$, and then "throw away" all primed variables by an existential quantification. In fact, all these operations can be performed during a single BDD traversal, if v_i and v'_i are always kept together in the variable order.

A more sophisticated algorithm avoids the recalculation of certain common subexpressions during the traversal of the formula. For a detailed exposition, cf. [4].

The complexity of this algorithm is potentially exponential in the number of variables and exponential in the formula. Nevertheless, in practice the number of iteration steps required to reach a fixed point is often small ($\leq 10^3$). For hardware systems, that is, in the verification of sequential circuits, most states are reachable in very few steps, but the BDDs tend to grow exponentially in the first few steps. For software systems, especially if there is not much parallelism contained, the BDD often grows only linear with the number of steps, until the whole state space is traversed. The following picture shows the relation between the BDD size and number of steps in typical examples.

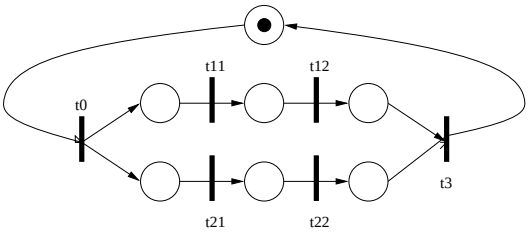


10 Partial Order Techniques

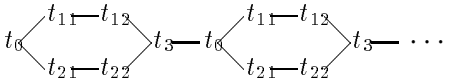
With symbolic methods we try to tackle the complexity problem which arises from the parallel composition of modules by using the BDD data structure which can handle very large sets. Partial order methods, on the other hand, try to avoid the generation of large sets: We only generate a minimal part of the state space which is necessary to evaluate the given formula. Thus symbolic methods correspond to the global model checking approach, whereas partial order methods are a refinement of local model checking.

The interleaving definition of parallel program semantics determines the state space of the global system to be the product of all state spaces of its parallel components. This can lead to wasteful algorithms. In general, each (nondeterministic) execution of a program generates a partial order, where points are ordered by causality. In interleaving semantics this partial order is represented by the set of all of its interleaving sequences.

For example, the following elementary Petri net represents a system with two processes synchronising via t_3 and t_0:



This system generates the following partial order:



Some of the interleaving sequences are

$t_0 t_{11} t_{12} t_{21} t_{22} t_3 \ldots$
$t_0 t_{11} t_{21} t_{12} t_{22} t_3 \ldots$
$t_0 t_{11} t_{21} t_{22} t_{12} t_3 \ldots$
$t_0 t_{21} t_{11} t_{22} t_{12} t_3 \ldots$
$t_0 t_{21} t_{11} t_{12} t_{22} t_3 \ldots$

However, it may not be necessary to consider *all* of these interleavings to determine, e.g., the truth value of the formula $\Box\Diamond t_3$. The main idea of partial order methods is to try to inspect only some "representative" interleaving sequences for the formula in question. Thus, we do *not* alter the semantics to deal with "real" concurrency (where independent transitions can occur at the same time), and we do *not* extend the logic to be able to express partial order

properties. On the contrary, we will limit the expressiveness of temporal logic and use the partial order to *improve efficiency* of model checking. The relevant keywords are *stubborn sets* ([12]), *sleep sets* ([14]) and *interleaving set temporal logic*.

Stuttering Invariance

Given an elementary Petri net N and a formula φ, we want to find whether there exists a run ρ of N satisfying φ. In general, there are infinitely many runs through the system; therefore we group these into a finite number of equivalence classes, such that the existence of a satisfying run ρ implies that every element of the equivalence class $[\rho]$ satisfies φ. Thus we only have to check a finite number of equivalence classes, and a coarser partition yields a better algorithm.

To do so, we need a *stuttering invariant* temporal logic, i.e., one in which no next-state operator is definable. The next-operator has always been a topic of emotional discussions in temporal specification. Most notions of refinement of systems will not preserve properties involving next-operators.

Consider a formula over atomic propositions $\mathcal{P} = \{\mathtt{p}_1, ..., \mathtt{p}_k\}$. Two natural models $\mathcal{M}$ and $\mathcal{M}'$ are *strongly equivalent* with respect to $\{\mathtt{p}_1, ..., \mathtt{p}_k\}$, if they are of the same cardinality, and for all $i \geq 0$ and all $\mathtt{p} \in \{\mathtt{p}_1, ..., \mathtt{p}_k\}$ we have $w_i \in \mathcal{I}(\mathtt{p})$ iff $w'_i \in \mathcal{I}'(\mathtt{p})$. A point w_i in $\mathcal{M}$ is *stuttering* w.r.t. $\{\mathtt{p}_1, ..., \mathtt{p}_k\}$, if for all $\mathtt{p} \in \{\mathtt{p}_1, ..., \mathtt{p}_k\}$ we have $w_i \in \mathcal{I}(\mathtt{p})$ iff $w_{i+1} \in \mathcal{I}(\mathtt{p})$. Two models $\mathcal{M}_1$ and $\mathcal{M}_2$ are *stuttering equivalent* w.r.t. $\{\mathtt{p}_1, ..., \mathtt{p}_k\}$, if the two models $\mathcal{M}'_1$ and $\mathcal{M}'_2$ obtained by eliminating all stuttering states from $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, are strongly equivalent w.r.t. $\{\mathtt{p}_1, ..., \mathtt{p}_k\}$. A formula φ is *stuttering invariant*, if for any two models $\mathcal{M}_1$ and $\mathcal{M}_2$ which are stuttering equivalent with respect to the set of atomic propositions of φ it holds that $\mathcal{M}_1 \models \varphi$ iff $\mathcal{M}_2 \models \varphi$.

Stuttering invariance allows to group all stuttering equivalent runs into the same equivalence class, thereby reducing the average complexity of the model checking. Of course, the reduction will be better if φ uses less propositions. Usually, a given formula mentions only a small subset of the system, allowing the equivalence classes to be rather large. In particular, all runs of an elementary Petri net which differ only in the interleaving of independent transitions (to be defined below) are stuttering equivalent with respect to all formulas not mentioning these transitions or adjacent places.

Analysis of Elementary Nets

How can we talk about elementary Petri nets? Recall that a state of the net is just a marking of its places. Thus, a reasonable choice seems to be to use the set of places as propositions, where $\mathtt{p}$ is true in a state iff place $\mathtt{p}$ is marked in that marking. As temporal operator, we define the *reflexive until*:

$\mathcal{M}, \mathcal{I}, w_i \models \mathcal{U}^o(\varphi, \psi)$ iff

$\exists j \; (i \leq j \;\wedge\; w_j \models \varphi \;\wedge\; \forall k \; (i \leq k < j \rightarrow w_k \models \psi))$

With this reflexive definition, it is no longer possible to define a next-state operator. (Of course, this spoils the expressive completeness of the logic...)

When are two transitions independent from one another? Firstly, independent transitions can neither disable nor enable each other; that is, if t_1 is enabled in s and s' is a successor of s with respect to the firing of t_1, then t_2 is enabled at s iff t_2 is enabled at s', and vice versa for t_2 firing. Secondly, if t_1 and t_2 are both enabled in s, then they can commute; that is, the state obtained by first firing t_1 and then t_2 is the same as first firing t_2 and then t_1.

However, it is not practical to check these two properties for all pairs of transitions in all global states of the system (remember, we wanted to avoid having to construct all global states). Therefore, we use a syntactic condition which ensures that some transition is independent from another one.

Call a set T of transitions *persistent* in s, if whatever one does from s while remaining outside of T does not affect T. Formally, T is persistent in s iff for all $t \in T$ and all firing sequences $t_0, t_1, ..., t_n, t$ such that all $t_i \notin T$ there exists a stuttering equivalent firing sequence starting with t.

If T is persistent, we do not have to consider the firing of transitions outside of T when constructing the successors of the given state in the depth-first-search; there will be a stuttering equivalent sequence constructed by the firing of some $t \in T$.

However, this definition still is not effective. There is no efficient way to compute a minimal persistent set of transitions for a given state. Therefore, we compute an approximation. There is a tradeoff between the amount of time spent in the calculation of minimal persistent sets, and the reduction of the state space obtained. As a general strategy, some simple heuristics will go a long way, and sophisticated methods don't add too much.

We start with a single enabled transition $T = \{t\}$ and repeat until stabilisation to add all transitions which can "interfere" with some transition in T. Here "interfere" means

- can enable or disable, or
- cannot commute with.

Given any marking m, firable transition t_f and disabled transition t, we have to find a set of firable transitions such that the firing of any transition in this set could make t fire before t_f fires. A set $NEC(t, m)$ of transitions is *necessary* for t in m, if $NEC(t) = \{t' \mid p \in t'\bullet\}$ for some $p \in \bullet t \setminus m$. $NEC^*(t, m)$ is any set of transitions containing t which is transitively closed under necessity; that is, for any $t' \in NEC^*(t, m)$ such that t is disabled in m there exists a set $NEC(t', m)$ of transitions necessary for t' such that $NEC(t', m) \subseteq NEC^*(t, m)$.

If t is in conflict with t_f, then all transitions in $NEC^*(t, m)$ have to be fired as alternatives to the firing of t_f. But, there is still another class of dependent transitions. We want to obtain stuttering equivalence with respect to the atomic propositions in φ. Therefore, we have to take into account that φ might fix an order onto the firing of independent transitions. Usually, φ contains only a few propositions. Call a transition *visible* for φ, if $\bullet t \cup t\bullet$ contains any place p appearing in φ. If t is visible, the firing order with other visible transitions is important. A visible

transition can be regarded to be in conflict with all other visible transitions. Define the *conflict* of t by

$$C(t) = \{t' \mid {\bullet}t' \cap {\bullet}t \neq \emptyset\} \cup \{t\}.$$

The *extended conflict* of t is just the conflict of t, if t is invisible; otherwise, it is the conflict of t plus all other visible transitions. Now a *dependent set* $DEP(t_f, m)$ of t_f is any set of transitions such for any t in the extended conflict of t_f there exists a set $NEC(t, m) \subseteq DEP(t_f, m)$.

Finally, the set of transitions which are fired should be transitively closed under dependency;
thus, let $READY(m)$ be any (smallest) nonempty set of firable transitions, such that

$$DEP(t_f, m) \subseteq READY(m) \text{ if } t_f \in READY(m).$$

Now, we can prove that for any firing sequence of the net there exists a firing sequence *generated only by firing ready transitions* which is stuttering equivalent with respect to all safety properties of the logic.

Therefore, during the construction of the set of successors of a state in the depth first search we can neglect all firable transitions which are not ready. This can result in a considerable average case reduction; in fact, for examples with many concurrent and "almost" independent processes it can logarithmically reduce the state space which has to be traversed. Though the worst case complexity of constructing a ready set is cubic in the size of the net, in average examples it is only linear in the number of transitions.

The above construction can be extended to deal also with liveness properties. To do so, we need to assure that whenever a state is reached for the second time, a different ready set is constructed, to make sure that no eventuality is delayed infinitely often. For a detailed exposition, see [15] (of which an extended version can be obtained via www from the author).

Further Topics

There are several extensions to each of the topics presented here, and in many areas there is a lot of active research.

For example, it is possible to reduce the size of the system by applying certain *symmetry* collapses and *refinement* operations.

To verify infinite state systems, some kind of *induction* on state spaces is necessary. What kind of language is of sufficiently low expressivity to be still decidable, but allows to specify all "interesting" properties?

Another "hot" item is the verification of real time and hybrid systems. In *real time systems* with each transition a certain time bound or interval is associated, and *hybrid systems* allow discrete and continuous transitions from one state to another.

Furthermore, there is the search for *compositional* proof systems, where a proof of the global system can automatically be deduced from the proofs for each parallel component against the interface specification.

References

1. R E Bryant. 'Symbolic boolean manipulation with ordered binary decision diagrams'. Technical Report CMU-CS-92-160, CMU School of Computer Science, Pittsburgh, (July 1992).
2. J P Burgess. 'Basic tense logic'. In D Gabbay and F Guenthner, eds., *Handbook of philosophical logic, Vol. II*, chapter 2, pp. 89–134. Reidel, Dordrecht, (1984).
3. E Clarke, O Grumberg, and D Long. 'Model checking'. In M Broy, ed., *Deductive Program Design*, Nato ASI Series F, pp. 305–350. Springer, Berlin, (1996).
4. E A Emerson. 'Temporal and modal logic'. In J van Leeuwen, ed., *Handbook of theoretical computer science, Vol. B*, chapter 16, pp. 997–1072. Elsevier, Amsterdam, (1990).
5. M Fitting. *Proof methods for modal and intuitionistic logics*. Reidel, Dordrecht, 1983.
6. D Gabbay. 'Temporal logic : mathematical foundations'. Technical Report MPI-I-92-213, Max-Planck-Institut f. Informatik, Saarbrücken, (March 1992).
7. F Kröger. *Temporal logic of programs*. Number 8 in EATCS Monograph. Springer, New York, 1987.
8. Z Manna and A Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, New York, 1992.
9. K McMillan. *Symbolic model checking*. Phd dissertation, CMU School of computer science, Pittsburgh, 1992.
10. C Stirling. 'Modal and temporal logics'. In S Abramsky, D Gabbay, and T Maibaum, eds., *Handbook of logic in computer science*. Oxford University Press, (1991).
11. W Thomas. 'Automata on infinite objects'. In J van Leeuwen, ed., *Handbook of theoretical computer science, Vol. B*, chapter 4, pp. 133–186. Elsevier, Amsterdam, (1990).
12. A Valmari. 'A stubborn attack on state explosion'. In E M Clarke, ed., *Proceedings of CAV 90*, pp. 25–41. Springer, (1991).
13. I Walukiewicz. 'Completeness of Kozen's axiomatization of the propositional μ-calculus'. In D Kozen, ed., *10th Annual IEEE Symposium on Logic in Computer Science*, pp. 14–24. IEEE Computer Society Press, (June 1995).
14. P Wolper. 'Partial-order methods for concurrent program verification'. Lecture notes, DIMACS tutorial on computer aided verification, September 1995.
15. T Yoneda and H Schlingloff. 'Efficient verification of parallel real-time systems'. In C Courcoubetis, ed., *Proceedings of CAV 93*, pp. 321–332. Springer, (1993).

Acknowledgements: I would like to thank Chris Brink for organizing WOFACS '96, and my colleagues M. Frey, M. Podolsky, A. Kurz, S. Merz and F. Regensburger at TUM, LMU and Siemens ZFE for proof-reading drafts of this manuscript. This work was supported by DAAD Procope Project 312/pro-bmbw-gg.