

## CDM

### Model Checking

Klaus Sutner  
Carnegie Mellon University  
[www.cs.cmu.edu/~sutner](http://www.cs.cmu.edu/~sutner)

### Battleplan

- The Verification Problem
- Kripke Structures
- Computation Tree Logic
- Linear Temporal Logic

### Verification

### Verification

What kind of logic is required to verify the correctness of a "system" (which could be hardware, software, a protocol, . . .)?

We need three components:

- A framework for modeling the system in question.
- A specification language that describes the desired properties.
- A verification algorithm that checks the specification against the description.

One might think that classical first order logic could be used for this purpose, but unfortunately there are several issues that make FOL less than suitable for our task.

### Expressiveness versus Hardness

The main problem is that the more expressive a language we chose, the harder the associated difficult decision problems become: even for propositional logic we already have to contend with NP-hardness, so one should expect to encounter worse hardness or even undecidability when moving to a stronger system.

However, again in analogy to the propositional case, there is hope to develop good algorithms for "practical problems".

So the question is: is there a nice class of problems that is

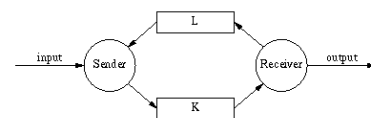
- close enough to real problems so that a solution is of interest, and
- simple enough so that one can find good algorithms.

We can get some guidance by looking at a concrete example of a system that we might want to deal with.

### Alternating Bit Protocol

There are four players in ABP:

- Sender  $A$  tries to transmit messages.
- Receiver  $B$  that would like to receive these messages.
- A message channel  $K$  that is responsible for the actual transmission of the messages.
- An acknowledgment channel  $L$  that is used for confirmation.



## Faulty Channels

If the channels are perfect there is no issue: just send the messages.

But what if the channels are faulty? For simplicity assume the only possible errors are that a channel may delete or duplicate messages (they cannot create messages or change the order).

How should sender and receiver communicate so that the messages are correctly transmitted?

Note that we must assume that the channels are not completely broken (if no message gets through ever no protocol will help).

The trick: The sender maintains a single protocol bit, initialized to 0.

## A Stop-and-Wait Protocol

- Each message sent by  $A$  contains a protocol bit, 0 or 1.
- $A$  repeatedly sends a message with its protocol bit, until receiving an acknowledgment (ACK) from  $B$  that has the same protocol bit.
- $B$  receives a message and sends an ACK to  $A$  with the same protocol bit. Only the first time the message is received it is delivered for further processing.
- $A$  receives an acknowledgment with the same bit as the current message, it stops transmitting, flips the protocol bit, and repeats the protocol for the next message.

Why does this protocol work? How do we prove it works? Automatically?

It's not even clear how we would express this formally (in a way suitable as input for a checker).

## Model Checking

A version of the Entscheidungsproblem, lot of work in computer science since the 1980's.

Basic idea: fix some suitable logic and a collection of structures for the logic.

- The structures describe the systems one is interested in (could be programs, hardware, protocols, ...)
- The logic is used to express properties of these systems (specifications, correctness assertions).

**Goal:** Automatically verify that a given structure conforms to the specification.

Find efficient decision algorithms to check if

$$\mathfrak{A} \models \varphi$$

## Kripke Structures

## Transition Systems

Let's focus on systems that have a clear notion of state transition: the system evolves in a sequence of steps

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n \rightarrow \dots$$

This type of behavior is more typical of hardware and protocols than of software.

We can think of the possible transitions as a binary relation on the space of all possible states:

$$\rightarrow \subseteq S \times S$$

or, if you prefer, as a digraph on  $S$ .

## Describing State

We have a collection  $P$  of atomic properties  $p_1, p_2, \dots, p_k$ .

Each property may or may not hold at any specific state  $s \in S$ . The exact nature of these properties depends on the system in question, think about

- message has been sent
- acknowledgment has been received
- printer is busy
- register  $R_0$  is initialized to 0
- process 2 is in its critical state

The details don't matter, but we have to be able to check easily whether property  $p$  holds in state  $s$ .

## Property Maps

Formally we use a *property map*  $L : S \rightarrow \mathfrak{P}(P)$  that describes which properties hold at which states.

A state  $s$  satisfies property  $p$  if  $p \in L(s)$ , in symbols

$$s \models p \iff p \in L(s)$$

The property map can be represented as a table of bit-vectors, one for each state.

The complete structure then consists of a collection of states, a binary relation meaning "next state" and a property map.

## Designing Models

**Definition 1.** A *Kripke structure* for  $P$  has the form

$$\mathcal{A} = \langle S, \rightarrow, L \rangle$$

where  $\rightarrow$  is a binary relation on the set of states  $S$  and  $L : S \rightarrow \mathfrak{P}(P)$  is a property map. To avoid special cases we require that  $\forall s \exists t s \rightarrow t$ .

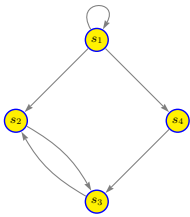
One can think of such a structure as a digraph on  $S$  whose nodes have out-degree at least 1 and are labeled by subset of  $P$ .

These structures will model the systems whose properties we are trying to verify.

**Exercise 1.** Explain how these Kripke structures can be construed as specialized first order structures. What is the appropriate first order language here?

## A Small Kripke Structure

Here is system with 4 states  $s_1, s_2, s_3, s_4$  and 4 atomic properties  $p, q, r, t$ . The atomic properties at each state are given in the table:



$s_1$	$r$	0	0	1	0
$s_2$	$q, r, t$	0	1	1	1
$s_3$	$q, r$	0	1	1	0
$s_4$	$p, q$	1	1	0	0

## Designing a Language

Next we need a specification language in which to describe the properties of a Kripke structure.

The easy part of this is to deal with atomic propositions and logical connectives. We use the symbols

$\perp, \top$	constants false, true
$p, q, r, \dots$	propositional variables
$\neg$	not
$\wedge$	and, conjunction
$\vee$	or, disjunction
$\Rightarrow$	conditional (implies)

So this is very similar to propositional logic, but an assertion  $p$  now means: the current state has property  $p$ .

We will give a more detailed definition of the semantics of this language below.

## Quantifiers

The propositional part is easy, but in order to make interesting assertions we need to add quantifiers. It is tempting to duplicate the approach of FOL:

$\exists s$  there exists a state  $s$   
 $\forall s$  for all states  $s$

Unfortunately, this is less helpful than one might think. For example, we would like to be able to say something like:

If the current state  $s$  has property  $p$  then there is a path to some state that has property  $q$ .

The problem is that in FOL we cannot express the concept of a path of arbitrary length (of course we can say things like "there is a path of length 5").

## Choosing Quantifiers

To obtain the right notion of quantifier recall that we are dealing with systems that evolve via state transitions:

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n \rightarrow \dots$$

Any such path can be considered as a computation of the system.

Since the relation  $\rightarrow$  is not required to be deterministic the evolution may be branching: for any  $s \in S$  there may be several "next" states.

So, we have a tree of possible computations (behaviors) rather than just a single path.

The key idea is select quantifiers that make it possible to make assertions about what happens during a computations, along these paths.

### Modal Quantifiers

This is really a *temporal logic* problem: think of the transitions as time, so we want to make assertions about the future. Here are some natural candidates.

- **X**: at the next state in the path
- **F**: at some point along the path
- **G**: always along the path
- **U**: until some point along the path
- **R**: release at some point along the path

All these quantifiers except the last two unary. The last two are a bit more complicated and combine two subformulae:

$\varphi \mathbf{U} \psi$  means that  $\varphi$  holds until  $\psi$  happens.

$\varphi \mathbf{R} \psi$  means that  $\psi$  holds up until (including) when  $\varphi$  holds for the first time (which may be never).

### Linear-Time Temporal Logic

### A Logic for Kripke Structures

Here is a proposal for a logic that can express specifications for Kripke structures.

**Definition 2.** *Linear-time Temporal Logic (LTL) formulae are defined by taking the Boolean closure of atomic predicates  $p$  and the modal quantifiers **X**, **F**, **G** and **U**.*

To define semantics we consider infinite paths in a Kripke structure starting at a particular source vertex.

Write  $\sigma$  for the shift operation (left-shift) on infinite paths.

Hence  $\sigma^i(\pi)$  is the path obtained from path  $\pi$  by dropping the first  $i$  states.

### LTL Semantics

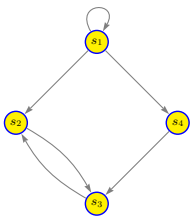
As usual, induction on the built-up of  $\varphi$ . No surprises here.

- $\pi \models p: p \in \mathcal{L}(\pi_0)$ .
- $\pi \models \psi \wedge \varphi: \pi \models \psi$  and  $\pi \models \varphi$ .
- $\pi \models \psi \vee \varphi: \pi \models \psi$  or  $\pi \models \varphi$ .
- $\pi \models \neg\varphi: \text{not } \pi \models \varphi$ .
- $\pi \models \mathbf{X}\varphi: \sigma(\pi) \models \varphi$ .
- $\pi \models \mathbf{G}\varphi: \text{for all } i \geq 0, \sigma^i(\pi) \models \varphi$ .
- $\pi \models \mathbf{F}\varphi: \text{for some } i \geq 0, \sigma^i(\pi) \models \varphi$ .
- $\pi \models \varphi \mathbf{U} \psi: \text{for some } i \geq 0, \sigma^i(\pi) \models \psi$  and for all  $j < i: \sigma^j(\pi) \models \varphi$ .

We write  $\mathcal{A}, s \models \varphi$  if all paths starting at  $s$  in  $\mathcal{A}$  satisfy  $\varphi$ .

### Example: Properties of a Kripke Structure

Recall the 4 state Kripke structure from above for predicates  $p = \{p, q, r, t\}$ .



$s_1$	$r$	0	0	1	0
$s_2$	$q, r, t$	0	1	1	1
$s_3$	$q, r$	0	1	1	0
$s_4$	$p, q$	1	1	0	0

Some paths starting at  $s_1$  associated with this structure:

- $s_1, s_1, s_1, \dots, s_1, s_1, s_1, s_1, \dots$
- $s_1, s_1, s_1, \dots, s_1, s_2, s_3, s_2, s_3, s_2, s_3, \dots$
- $s_1, s_1, s_1, \dots, s_1, s_4, s_3, s_2, s_3, s_2, s_3, \dots$

### Some Assertions

The following assertions are all valid in the structure  $\mathcal{A}$  from the last slide.

- $\mathcal{A}, s_1 \models \mathbf{X}(q \wedge r)$
- $\mathcal{A}, s_1 \models \mathbf{X}(p \wedge a \wedge \neg r)$
- $\mathcal{A}, s_1 \models \mathbf{F}r$
- $\mathcal{A}, s_3 \models \mathbf{G}(q \wedge r)$
- $\mathcal{A}, s_1 \models r \vee (r \mathbf{U} p)$
- $\mathcal{A}, s_1 \models r \mathbf{U} (p \vee t)$

**Exercise 2.** *Verify that all these assertions are indeed all valid.*

**Exercise 3.** *Modify the structure and find valid assertions.*

### Model Checking

We have a framework but no algorithms: given a Kripke structure  $\mathcal{A}$ , a state  $s$  and an LTL formula  $\varphi$ , how do we check whether  $\mathcal{A}, s \models \varphi$ ?

More precisely, how do we solve the following decision problem:

**Problem: LTL Validity**

Instance: An LTL formula  $\varphi$ , a Kripke structure  $\mathcal{A}$  and a state  $s$  in  $\mathcal{A}$ .

Question: Does  $\mathcal{A}, s \models \varphi$  hold?

The example might suggest an approach where we associate formulae with the states. More precisely, try to label state  $s$  with all the subformulae of  $\varphi$  that hold at that state – using some kind of induction on the build-up of the formula, starting with atomic expressions.

This can be done, but there is an easier way.

### Büchi to the Rescue

Brilliant insight: a infinite path in  $\mathcal{A}$  can be interpreted as an  $\omega$ -word over  $\Sigma = \text{pow}(P)$ : the  $i$ th letter represents the atomic proposition that hold at the  $i$ th state in the path.

We can think of the Kripke structure as a Büchi automaton that accepts all such words. More precisely, we can construct a Büchi automaton  $\mathfrak{A}$  with this behavior for each source state  $s$ .

The states of  $\mathfrak{A}$  are  $S$  plus a special initial state  $q_0$  and there is a special transition  $(q_0, L(s), s)$ .

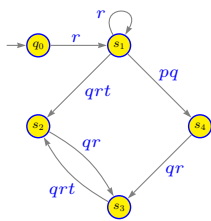
The other transitions are

$$(p, L(q), q) \text{ for } p \rightarrow q$$

All states are final.

### Example

The Büchi automaton for our 4-state Kripke structure with source state  $s = s_1$ .



The labels really are over the 16-element alphabet  $\Sigma = \text{pow}(\{p, q, r, t\})$ .

### The System Automaton

It is clear from the construction that  $\mathcal{L}(\mathfrak{A})$  is the collection of all  $\omega$ -words corresponding to paths in the Kripke structure starting at the source state.

Now consider the specification formula  $\varphi$ . We only need to convert  $\varphi$  into another Büchi automaton  $\mathfrak{B}$  such that

$$\mathcal{A}, s \models \varphi \iff \mathcal{L}(\mathfrak{A}) \subseteq \mathcal{L}(\mathfrak{B})$$

The latter property can be checked by testing

$$\mathcal{L}(\mathfrak{A}) \cap (\Sigma^\omega - \mathcal{L}(\mathfrak{B})) = \emptyset$$

Note, though, that this involves potentially superexponential cost: we have to determinize the Büchi automaton  $\mathfrak{B}$  (Safra's algorithm).

### Counterexamples

Actually, in applications it is usually more interesting to construct  $\mathfrak{B}_{\neg\varphi}$ : if that automaton has empty intersection with  $\mathcal{L}(\mathfrak{A})$  we are fine.

If not, we can extract a **counterexample**: we can construct a word accepted by  $\mathfrak{B}_{\neg\varphi}$  that demonstrates exactly why the specification  $\varphi$  is violated by  $\mathfrak{A}$  and therefore by the Kripke structure  $\mathcal{A}$ .

The ability to construct counterexamples is very important in the real world: it is not enough to find out that the system fails the specification, we want a hint as to how one might fix the problem. Careful study of a counterexample or two can help a lot.

### Preprocessing

It is convenient to do a bit of preprocessing: formula  $\varphi$  is first converted into negation normal form, only operators  $\neg, \wedge, \mathbf{X}$  and  $\mathbf{U}$  are retained. To this end rewrite according to

$$\begin{aligned} \mathbf{F} \varphi &\Rightarrow \mathbf{T} \mathbf{U} \varphi \\ \mathbf{G} \varphi &\Rightarrow \varphi \mathbf{U} \perp \\ \neg \neg \varphi &\Rightarrow \varphi \\ \neg(\varphi \mathbf{U} \psi) &\Rightarrow \neg \varphi \mathbf{R} \neg \psi \\ \neg(\varphi \mathbf{R} \psi) &\Rightarrow \neg \varphi \mathbf{U} \neg \psi \end{aligned}$$

**Exercise 4.** Show that the formula produced in this way is equivalent to the original one.

### The Conversion Process

It is slightly more convenient to convert to a *generalized Büchi automaton*.

A GNBA has a family  $\mathcal{F} \subseteq \text{pow}(Q)$  of final state sets. It accepts its input if there is a run that touches all the final states sets infinitely often.

**Exercise 5.** Show how to convert a GNBA into an ordinary Büchi automaton.

The states of the automaton will be certain sets of subformulae of  $\varphi$ .

**Definition 3.** For any LTL formula  $\varphi$  define its **subformula closure**  $\text{subfml}(\varphi)$  to be the collection of all subformulae plus their negations.

Note that  $\text{subfml}(\varphi)$  can easily be generated by traversing the parse tree of  $\varphi$ .

### Elementary Formula Sets

Let  $A \subseteq \text{subfml}(\varphi)$ .

**Definition 4.**  $A$  is **logically consistent** if

- $\rho \wedge \psi \in A$  iff  $\varphi \in A$  and  $\psi \in A$ .
- $\rho \in A$  implies  $\neg\rho \notin A$ .
- $\top \in \text{subfml}(\varphi)$  implies  $\top \in A$ .
- $\rho \in A$  implies  $\psi \cup \rho \in A$ .
- $\psi \cup \rho \in A$  and  $\rho \notin A$  implies  $\psi \in A$ .

$A$  is **maximal** if  $\rho \notin A$  implies  $\neg\rho \in A$ .

$A$  is **elementary** if it is logically consistent, locally consistent and maximal.

### Quoi?

In an elementary set of subformulae every subformula of  $\varphi$  must appear either plain or negated (but not both).

For example, consider  $\varphi = p \cup (\neg p \wedge q)$ .

The 16 elementary sets of subformulae are obtained as follows:

$p$	$\neg p$	$q$	$\neg q$	$(\neg p \wedge q)$	$\neg(\neg p \wedge q)$	$\varphi$	$\neg\varphi$
0	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	1	1	0	0	1
			...				
1	0	1	0	1	0	1	0

### Constructing the Automaton

The set of states  $Q$  consists of all elementary sets  $A \subseteq \text{subfml}(\varphi)$ .

The initial states are all  $A$  such that  $\varphi \in A$ .

For each  $\rho \cup \psi \in \text{subfml}(\varphi)$  we get a collections of final states

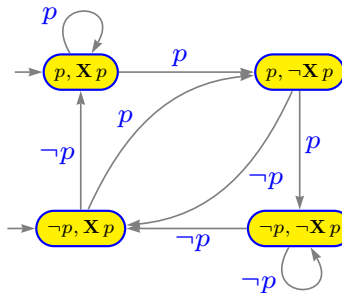
$$\{ A \in Q \mid \rho \cup \psi \notin A \text{ or } \psi \in A \}$$

There is a transition  $(A, a, B)$  if

- $a = A \cap P$ .
- For  $\tau = \mathbf{X} \tau_1 \in \text{subfml}(\varphi)$ :  $\tau \in A$  iff  $\tau_1 \in B$ .
- For  $\tau = \tau_1 \cup \tau_2 \in \text{subfml}(\varphi)$ :  $\tau \in A$  iff  $\tau_2 \in A \vee (\tau_1 \in A \wedge \tau \in B)$ .

### Example

Consider  $\varphi = \mathbf{X} p$ . Yes, yes, that's too cheap for words.



All states are final by default.

### Efficiency

Complexity of the conversion algorithm to GNBA is exponential in the size of  $\varphi$  in the worst case and the number of final state sets is  $O(|\varphi|)$ .

The ultimate Büchi can be built in time and space  $O(|\varphi| 2^{|\varphi|})$ .

However, for "practical" formulae the conversion process seems to work reasonably well.

The actual Emptiness test is linear in the size of the automaton.

## Computation Tree Logic

### More Quantifiers

The semantics for LTL has an implicit universal quantifier:

$$\mathcal{A}, s \models \varphi$$

means that for all paths starting at  $s$  property  $\varphi$  holds.

So it is very tempting to enhance our language a bit by adding explicit quantifiers over paths.

A reasonable notion of *path quantifiers* would seem to be

- **A**: for all paths (starting at some state)
- **E**: there exists a path (starting at some state)

### The CTL Language

**Definition 5.** In addition to the propositional part from above, the language for CTL also allows the constructs

$$\begin{array}{ll} \mathbf{E X} \varphi & \mathbf{A X} \varphi \\ \mathbf{E G} \varphi & \mathbf{A G} \varphi \\ \mathbf{E F} \varphi & \mathbf{A F} \varphi \\ \mathbf{E}(\psi \mathbf{U} \varphi) & \mathbf{A}(\psi \mathbf{U} \varphi) \\ \mathbf{E}(\psi \mathbf{R} \varphi) & \mathbf{A}(\psi \mathbf{R} \varphi) \end{array}$$

So there are 10 (in words: ten) quantifiers in this logic.

Each combines quantification over paths and some temporal assertion.

This may seem a whole lot more complicated than ordinary first-order logic (which has just a universal and an existential quantifier), but it isn't.

### Semantics

We need to pin down the semantics for CTL. As for LTL, we use Kripke structures. For any state  $s$  and we define

$$\mathcal{A}, s \models \varphi$$

meaning: the formula  $\varphi$  holds along the paths starting at  $s$ . And, of course, ultimately we want to solve the decision problem:

**Problem:** *CTL Validity*

**Instance:** A CTL formula  $\varphi$ , a CTL structure  $\mathcal{A}$  and a state  $s$  in  $\mathcal{A}$ .

**Question:** Does  $\mathcal{A}, s \models \varphi$  hold?

### Semantics

For the propositional part of CTL the semantics are no different from ordinary propositional logic. For the quantifiers we define

- **E X**  $\varphi$ : for some  $s'$  such that  $s \rightarrow s'$ :  $\mathcal{A}, s' \models \varphi$ .
- **A X**  $\varphi$ : for all  $s'$  such that  $s \rightarrow s'$ :  $\mathcal{A}, s' \models \varphi$ .
- **E G**  $\varphi$ : there exists a path  $(s_i)$  starting at  $s$  such that  $\mathcal{A}, s_i \models \varphi$  for all  $i$ .
- **A G**  $\varphi$ : for all paths  $(s_i)$  starting at  $s$  we have  $\mathcal{A}, s_i \models \varphi$  for all  $i$ .
- **E F**  $\varphi$ : there exists a path  $(s_i)$  starting at  $s$  such that  $\mathcal{A}, s_i \models \varphi$  for some  $i$ .
- **A F**  $\varphi$ : for all paths  $(s_i)$  starting at  $s$  we have  $\mathcal{A}, s_i \models \varphi$  for some  $i$ .
- **E**  $(\psi \mathbf{U} \varphi)$ : there exists a path  $(s_i)$  starting at  $s$  and some  $i$  such that  $\mathcal{A}, s_i \models \varphi$  and for all  $j < i$   $\mathcal{A}, s_j \models \psi$ .
- **A**  $(\psi \mathbf{U} \varphi)$ : for all paths  $(s_i)$  starting at  $s$  there is some  $i$  such that  $\mathcal{A}, s_i \models \varphi$  and for all  $j < i$   $\mathcal{A}, s_j \models \psi$ .

That's it.

### Simple Statements

Since we are dealing with a digraph, one might wonder what kind of graph-theoretic statements one can make in CTL.

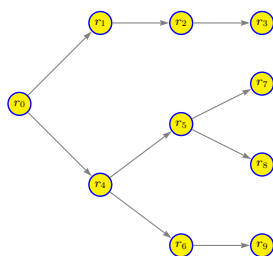
- $\mathcal{A}, s \models \mathbf{E X} p$  means there is a neighbor of  $s$  with property  $p$ .
- $\mathcal{A}, s \models \mathbf{A X} p$  means that all neighbors of  $s$  have property  $p$ .
- $\mathcal{A}, s \models \mathbf{E F} p$  means there is a path from  $s$  to  $t$  provided that  $t$  is the only state with property  $p$ .

As is clear from the last example, the labeling (the assignment of atomic properties) is crucial here.

Also note that we can only make statements about the weakly connected component of  $s$  in  $\mathcal{A}$ ,  $s \models \varphi$ : none of the other states play any role in the evaluation of the formula.

### Unfolding the Diagram

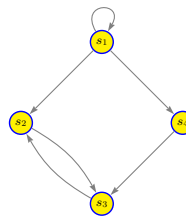
A good way to visualize the meaning of CTL formulae is to unfold the digraph into a tree, something like



One can then trace the paths in this tree.

### The Old Small Example

Here is system with 4 states  $s_1, s_2, s_3, s_4$  and 4 atomic properties  $p, q, r, t$ . The atomic properties at each state are given in the table:



$s_1$	$r$
$s_2$	$q, r, t$
$s_3$	$q, r$
$s_4$	$p, q$

### Some Assertions

The following assertions are all valid in the structure  $\mathcal{A}$  from the last slide.

- $\mathcal{A}, s_1 \models \text{EX}(q \wedge r)$
- $\mathcal{A}, s_1 \models \neg \text{AX}(q \wedge r)$
- $\mathcal{A}, s_1 \models \neg \text{EF}(p \wedge r)$
- $\mathcal{A}, s_3 \models \text{EG}r$
- $\mathcal{A}, s_3 \models \text{AG}r$
- $\mathcal{A}, s_1 \models \text{AF}r$
- $\mathcal{A}, s_1 \models \text{E}(p \wedge q \text{ U } r)$
- $\mathcal{A}, s_1 \models \text{A}(p \text{ U } r)$

**Exercise 6.** Verify that all these assertions are indeed valid in  $\mathcal{A}$ .

**Exercise 7.** Find more valid and invalid assertions for this structure.

### Some Typical Assertions

Here are some important assertions one can make in this language.

$p$  holds infinitely often, no matter what happens starting at  $s$  (think of  $p$ : some process is enabled):

$$\mathcal{A}, s \models \text{AGAF}p$$

We can make  $p$  hold again, no matter has happened so far (think of  $p$ : system is reset):

$$\mathcal{A}, s \models \text{AGEF}p$$

Ultimately,  $p$  will hold everywhere, no matter has happened so far (think of  $p$ : deadlock has occurred):

$$\mathcal{A}, s \models \text{AFAG}p$$

So the last property is not desirable and the system specification would presumably forbid it.

### Exploiting Logic

Recall from our discussion of propositional and predicate logic that equivalence is an important way to rewrite and simplify formulae.

How would we define equivalence here?

$$\psi \equiv \varphi \iff \forall \mathcal{A}, s (\mathcal{A}, s \models \psi \leftrightarrow \mathcal{A}, s \models \varphi)$$

Here are some easy equivalences for CTL.

- $\text{AX}\varphi \equiv \neg \text{EX}\neg\varphi$
- $\text{EF}\varphi \equiv \neg \text{AG}\neg\varphi$
- $\text{AF}\varphi \equiv \neg \text{EG}\neg\varphi$
- $\text{A F}\varphi \equiv \text{A}(\text{T U}\varphi)$
- $\text{E F}\varphi \equiv \text{E}(\text{T U}\varphi)$

### A Hard Equivalence

**Proposition 1.**

$$\text{A}(\psi \text{ U } \varphi) \equiv \neg(\text{E}(\neg\varphi \text{ U } (\neg\psi \wedge \neg\varphi))) \vee \text{EG}\neg\varphi$$

The reason these equivalences are important is that they allow us to eliminate some of the connectives and quantifiers from the language and thus simplify the decision algorithm.

In particular a system using only

$$\perp, \neg, \wedge, \text{EX}, \text{EG}, \text{EU}$$

is already adequate.

## Model Checking

The idea behind the algorithm is to, given  $\mathcal{A}$  and  $\varphi$ , compute

$$\{s \in S \mid \mathcal{A}, s \models \varphi\}.$$

To this end, label the states of  $\mathcal{A}$  with all the subformulae of  $\varphi$  that are satisfied at the state.

This is done by induction, starting with atomic formulae and gradually building up to  $\varphi$  itself.

The process starts with the trivial subformulae  $\perp$  and  $p$ : nothing is labeled  $\perp$ , and for  $p$  the labeling is determined by  $L(s)$ .

## The Labeling Algorithm

We assume that the formula is built from connectives and quantifiers

$$\neg, \wedge, \text{EX}, \text{AF}, \text{EU}.$$

- $\varphi = \neg\psi$ : label  $s$  with  $\varphi$  if  $s$  is not labeled with  $\psi$ .
- $\varphi = \psi_1 \wedge \psi_2$ : label  $s$  with  $\varphi$  if  $s$  is labeled by both  $\psi_1$  and  $\psi_2$ .
- $\varphi = \text{EX } \psi$ : Label any state with  $\varphi$  if at least one of their immediate successors is labeled  $\psi$ .
- $\varphi = \text{AF } \psi$ : First label all states labeled with  $\psi$  with  $\varphi$ . Then label all states all of whose immediate successors are labeled  $\varphi$  by  $\varphi$ , until a fixed point is reached.
- $\varphi = \text{E } (\psi_1 \text{ U } \psi_2)$ : First label all states labeled with  $\psi_2$  with  $\varphi$ . Then label all states with  $\varphi$  if they are labeled  $\psi_1$  and one of their immediate successors is labeled  $\varphi$ , until a fixed point is reached.

## Complexity

**Proposition 2.** *The running time of this algorithm is  $O(mn(n+e))$  where  $m$  is the number of logical operators in the formula,  $n$  the number of states in  $\mathcal{A}$  and  $e$  the number of transitions.*

*Proof.*

To see this, note that each logical operator is handled only once, and that processing each such operator except for the last two cases is linear in  $n+e$ , the size of the digraph.

The last two cases,  $\varphi = \text{AF } \psi$  and  $\varphi = \text{E } (\psi_1 \text{ U } \psi_2)$  require time  $O(n(n+e))$ : we have to repeat the basic operation until no changes occur (which might take  $n-1$  rounds).

□

This performance is sufficient for small structures but becomes a problem when  $\mathcal{A}$  is large.

## Speed-Up

The fixed point method from above ignores the structure of the digraph. We can exploit this structure if we switch to another system of quantifiers:

$$\perp, \neg, \wedge, \text{EX}, \text{EG}, \text{EU}.$$

**EX** and **EU** can be handled in linear time with a bit of care.

For **EG**  $\psi$  we first produce the subgraph of all states labeled  $\psi$ . Then compute the strongly connected components and the acyclic skeleton of the restricted graph. Then determine all nodes from which such SCC is reachable. All of this can be done in time linear in  $n+e$  using, say, Tarjan's algorithm.

**Proposition 3.** *The improved version of the algorithm has running time  $O(m(n+e))$ .*

## State Explosion

Even the fast model checking algorithm is often not good enough: in practical applications the size of the structure is often enormous.

Its size is often exponential in the number of variables, so the adding one boolean variable to the system doubles the size of the CTL structure.

There is no silver bullet for this problem, but a number of methods exist to deal with state explosion.

- Highly efficient data structures such as ordered binary decision diagrams (OBDDs).
- Abstraction: Shrinking the model by removing variables that are not relevant for the formula in question.
- Reduction: for asynchronous systems many different traces may be equivalent as far as the formula in question is concerned.
- Induction: if there is a large number of similar components some type of induction may be used to deal with them.
- Composition: try to break the problem into a number of smaller ones.

## Example: Mutual Exclusion

Suppose we have 2 processes that share a resource and we want to make sure that at any time only one of them can be in a *critical section* where the resource is used.

We think of each of the processes as being in one of three states:

- $n$ : non-critical,
- $t$ : waiting to enter its critical state, and
- $c$ : in its critical section.

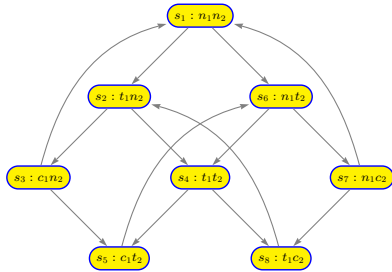
So each process is moving in the cycle  $n \rightarrow t \rightarrow c \rightarrow n \rightarrow \dots$

The problem is to devise a protocol that coordinates the two processes.

Needless to say, the protocol would have to guarantee certain properties. E.g., each process waiting to enter its critical section should ultimately get a chance to do so. We will write down these conditions later.

## A Protocol

Instead of spelling out the protocol in words we give the corresponding CTL structure  $c.A$ .



## Specification

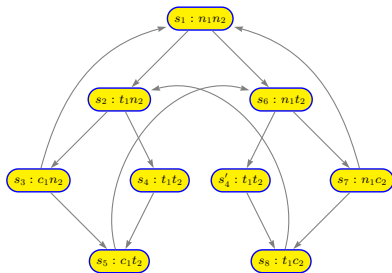
What properties should one demand from the protocol?

- Safety: only one process can be in its critical section at any time.  
 $\Psi_1 = \mathbf{A G} \neg(c_1 \wedge c_2)$ .
- Liveness: any process waiting to enter its critical section will ultimately do so.  
 $\Psi_2 = \mathbf{A G} ((t_1 \rightarrow \mathbf{A F} c_1) \wedge (t_2 \rightarrow \mathbf{A F} c_2))$ .
- Non-Blocking: Any process can always request to move into its critical section.  
 $\Psi_3 = \mathbf{A G} ((n_1 \rightarrow \mathbf{E X} t_1) \wedge (n_2 \rightarrow \mathbf{E X} t_2))$
- Sequencing: processes need not enter their critical sections in alternating fashion.  
 $\Psi_4 = \mathbf{E F} (c_1 \wedge \mathbf{E} (c_1 \mathbf{U} (\neg c_1 \wedge \mathbf{E} (\neg c_2 \mathbf{U} c_1)))) \wedge \dots$

**Claim 1.** Properties Safety, Non-Blocking and Sequencing are satisfied but Liveness is not.

## A Better Protocol

We can fix the Liveness problem by splitting state  $s_3$ .



## Success

**Claim 2.** The second protocol satisfies all four properties.

It is still not ideal, though.

For example, the protocol insists that at every step one of the state properties changes. But we cannot just let a process stay in its critical section, either: otherwise the other process never gets a chance.

This leads to the issue of *fairness*.

**Exercise 8.** Verify that the second model really satisfies  $\Psi_1$  through  $\Psi_4$ .

## Incomparable

One might wonder which logic is stronger: CTL or LTL.

The answer is: neither. There are properties that can be expressed in one but not the other.

CTL but not LTL

$\mathbf{A G E F} p$ : whatever happened, we can get back to  $p$ .

LTL but not CTL

$\mathbf{A} (\mathbf{G F} p \rightarrow \mathbf{F} q)$ : if there are infinitely many  $p$  (along a path) then there is a  $q$ .

Neither LTL nor CTL (this formula belongs to a larger logic called CTL\*)

$\mathbf{E G F} p$ : there is a path with infinitely many  $p$ .