CS 745 (Fall 2004): Computer Aided Verification (Introduction to Formal Methods)

Lecture 11: Modelling Notations, Explicit CTL Model Checking

Nancy Day

DC 2335, nday@cs.uwaterloo.ca

Office Hours: Mon 11:30-12:30, Thurs 3-4pm

`http://www.student.cs.uwaterloo.ca/~cs745`

uw.cs.cs745

Today's Agenda

- SMV: concurrency and fairness (leftover from last class)
- Modelling notations
- Explicit CTL model checking

Poetry on Formality

Excerpts from David Gries' Banquet Speech at ZUM'95, Limerick, Ireland, 7 Sept 1995.

`http://www.cs.cornell.edu/Info/People/gries/Papers/limerick.html`

The World is turning to C,
Though at best it is taught awkwardly.
But we don't have to mope,
There's a glimmer of hope,
In the methods of formality.
. . .

Poetry on Formality (con'd)

Specifying is certainly great,
For it helps to disambiguate.
It can make something clean
And precise like a dream.
If all specified that would be great!

But specs are but part of the game!
It's reas'ning that's most of the gain.
With reas'ning we prove,
Analyze, and review,
The programs we have to explain.

Modelling

The following is a discussion of styles of formal specification. In your projects, the verification tool you choose to use will have a certain modelling language that it uses.

Notes:

- This discussion is not meant to be exhaustive but rather to give you a bit of an overview.
- There are also a variety of semi-formal or informal specification styles, which aren't covered here.
- This discussion concentrate on notations that are good for describing the control aspects of the behaviour of a system.

A good source of more information on software specification is CS 445/645.

Behavioural Descriptions

operational: describe a high-level model of how the system works. Description is "complete" in the sense that if the description doesn't include a particular action, that system doesn't do that. Often just called models or "abstract models".

e.g., an algorithm for sorting an array, Kripke structure, SMV model

descriptive: state the desired properties of the system in a declarative manner. Also called "implicit definition".

e.g., a temporal logic property, the result of sorting an array is a permutation of the original array that is sorted.

Src: Ghezzi [GJM91]

States vs Streams

Two paradigms:

state-based: Describe the states of the system and use a next-state function or relation to describe how the system moves between states. Variety of ways to compose systems. Also called "model-oriented" notations.

stream-based: Think of the variables of the system as functions of time (modelled as either a function or a sequence) and describe how components modify these signals. Composition is just function composition.

Note: these can each be converted to the other.

Classification

basic modelling notations Verification algorithms are usually described in terms of these notations. These are often used to describe the semantics of other modelling notations.

user-level modelling notations A notation you would actually use to write a specification.

By analogy, a basic modelling notation is like assembler, and a user-level modelling notation is more like C or Java.

Basic Modelling Notations

- state-based:
  - Kripke structures
  - ω-automata
  - Büchi automata
  - labelled transition systems (LTS)

These are all variations of state transition graphs and are sometimes just called transition systems.

Logic is also a basic modelling notation and can be used to describe any of the above (can be state- or stream-style).

Kripke Structures

A Kripke structure $\mathcal{M}$ over a set of atomic propositions, AP, is a four tuple $\mathcal{M} = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is $\forall s \in S. \exists s'. R(s, s')$.
4. $L : S \rightarrow 2^{\text{AP}}$ is a function that labels each state with the set of atomic propositions true in that state.

DFAs, NFAs

Recall: A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states
- Σ is a finite input alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function ; could be partial or total transition function.
- q_0 is a start state ($q_0 \in Q$)
- F is a set of final states ($F \subseteq Q$)

In a non-deterministic finite automaton (NFA), δ is a transition relation: $\delta : Q \times \Sigma \times \Sigma$.

We talk about the language accepted by the automata.

Language Acceptance

The words accepted by an automata ($v \in \Sigma^*$) can describe sequences of execution events (also sequence of states) of a system. We often call them traces.

We often compare the words accepted by one automata with the words accepted by another automata.

ω-Automata

Recall that we are usually modelling reactive system whose execution does not terminate. So we aren't dealing with finite words, rather we want to describe infinite words to describe an infinite sequence of execution events or states.

So we need finite automata over infinite words. These are called ω-automata. They are still finite automata because they have a finite set of states.

ω-automata recognize words in Σ^ω, where ω indicates an infinite number of repetitions.

See: Clarke [CGP99], Ch. 9

Büchi Automata

The simplest kind of ω-automata is a Büchi automata.

A Büchi automaton has the same components as a regular automaton, but F is a set of accepting states rather than final states.

Words accepted by a Büchi automata are infinite in length and are accepted if they infinitely often enter a state in F.

Büchi Automata

Büchi automata are used to describe the language containment approach to model checking, where both the specification and the implementation are represented as automata and compared.

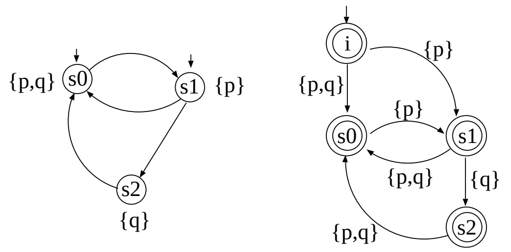
Notes:

- Non-deterministic Büchi automata are strictly more powerful than deterministic ones.
- An LTL formula can be converted to a Büchi automata by a tableau construction.

See Clarke [CGP99]

Kripke Structures vs Büchi Automata

A Kripke structure can be converted to a Büchi automata. All the states in the automata are accepting states, i.e., $F = Q$.



The words of the Büchi automata are infinite sequences of sets of propositions.

Src: Clarke [CGP99], p. 123

Labelled Transition Systems (LTS)

A labelled transition system is an automata on infinite words. Usually, descriptions of LTSs leave off the acceptance criteria and assume that all states are accepting.

$$(Q, \Sigma, \delta, q_0)$$

LTSs often come with a notion of composing two or more LTSs to run concurrently.

There are two common models of concurrency:

interleaving: the two models take turns

parallel: the two models can execute at the same time

User-level Modelling Notations

Characteristics
- states, state hierarchy
- composition, concurrency operators, definition of a step
- means of communication (shared variables, directed, broadcast)
- visual format: text, pictures, tables
- domain-specific constructs (e.g., VHDL δ-cycle, methods of synchronization between processes for communication)
- structuring mechanisms
- ability to extend with new definitions, etc.
- variety of tool support available (parsing, typechecking, visualisation, simulation, model checking, theorem proving)

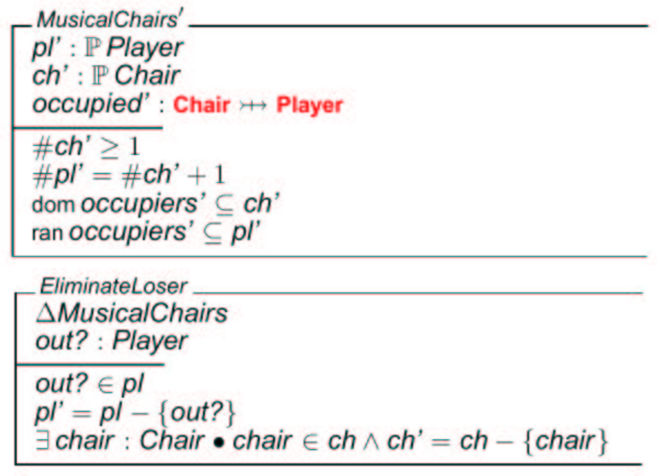
The following slides are a random sample of user-level modelling notations.

SMV

- no built-in notion of state (you can make your own)
- parallel or interleaving concurrency, shared memory
- step = set of parallel assignments
- textual format
- no domain specific constructs
- modules for structuring
- can extend using modules to make parameterized components; enumerated types, records using modules; limited in types
- describes a Kripke structure

Related notations: Promela for the model checker Spin (asynchronous composition)

Example Z Spec



MusicalChairs'

$pl' : \mathbb{P}\ Player$
$ch' : \mathbb{P}\ Chair$
$occupied' :$ **Chair ⤔ Player**

$\#ch' \geq 1$
$\#pl' = \#ch' + 1$
$\mathrm{dom}\ occupiers' \subseteq ch'$
$\mathrm{ran}\ occupiers' \subseteq pl'$

EliminateLoser

$\Delta MusicalChairs$
$out? : Player$

$out? \in pl$
$pl' = pl - \{out?\}$
$\exists\, chair : Chair \bullet chair \in ch \land ch' = ch - \{chair\}$

Z

- no control states
- preconditions and postconditions describe a step
- schema for structuring
- mostly textual format
- predicate logic and set theory is the underlying theory
- can use uninterpreted types and functions; can define functions
- “lots of funny symbols”

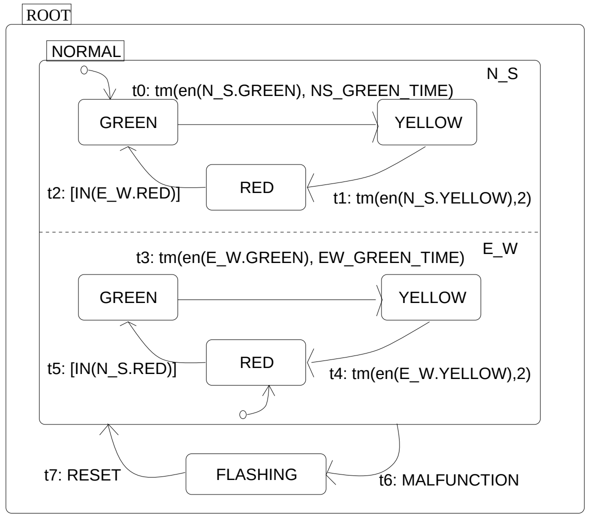
Related notations: VDM, B

Extended FSMs

We can extend finite state machines in a variety of ways:

- label transitions with triggering events, conditions, and actions to be taken upon following the transition
- allow these events and actions to reference variables of more complex (possibly non-finite) datatypes; the result may be a system with an infinite state space (even though a finite set of control states)
- hierarchy of states
- different operations for composing state machines such as parallelism and interleaving, etc.

Example Statecharts Spec



Src: STATEMATE manual

Statecharts

- address the problem of specifying a composition of state machines without writing out the product state machine
- state hierarchy: OR states, and AND states (for parallel composition), used for structuring specification; priority to transitions leaving states higher in the hierarchy
- broadcast communication
- since more than one component can be active, a step can be a cascading sequence of transitions
- visual representation of states
- a few domain-specific constructs: e.g., timeout events
- not easy to define parameterized states or new types or definitions

Related notations: RSML, UML, SDL

Example SCR Spec

Direction:

Current Mode	Dest $<$ Location	Dest $>$ Location	**New Mode**
Up	@T	–	Down
Down	–	@T	Up

Initial Mode: Up

Speed:

Current Mode	Dest = Location	Door=Closed	**New Mode**
Idle	@F	t	Moving
	f	@T	Moving
Moving	@T	–	Idle

Initial Mode: Idle

Src: Jo Atlee

SCR

- one level of state hierarchy
- flavours of tables as the structuring mechanism (e.g., mode transition table describes the next “mode” (state) depending on certain conditions)
- domain-specific constructs: @T, @F meaning a variable becomes true or becomes false (these are events)
- more interesting datatypes

Related notations: Parnas tables, AND/OR tables of RSML

Example CCS Spec [Mil89]

$$\begin{aligned}
Jobber &\equiv \mathsf{in}(job).Start(job)\\
Start(job) &\equiv \text{if } easy(job) \text{ then } Finish(job)\\
&\quad\ \text{else if } hard(job) \text{ then } Usehammer(job)\\
&\quad\ \text{else } Usetool(job)\\
Usetool(job) &\equiv Usehammer(job) + Usemallet(job)\\
Usehammer(job) &\equiv \overline{\mathsf{geth}}.\overline{\mathsf{puth}}.Finish(job)\\
Usemallet(job) &\equiv \overline{\mathsf{getm}}.\overline{\mathsf{putm}}.Finish(job)\\
Finish(job) &\equiv \overline{\mathsf{out}}(done(job)).Jobber
\end{aligned}$$

…descriptions of $Hammer$ and $Mallet$

$$Jobshop \equiv (Jobber|Jobber|Hammer|Mallet)\backslash\{\mathsf{geth}, \mathsf{puth}, \mathsf{getm}, \mathsf{putm}\}$$

CCS

- a type of “process algebra”
- directed communication and synchronization; multiple composition operators
- no explicit states
- define processes
- few datatypes
- algebraic transformations to prove equivalences of systems

Related notations: CSP, LOTOS

Example Larch Spec

```
SetTrait (Set, E): trait
  introduces
   emptyset: -> Set
   insert: E, Set -> Set
   member: E, Set -> Bool
   subset: Set,Set -> Bool
   ...
  asserts
   !x, y : E, s,t : Set
   member(x,insert(y,s)) == (x=y) \/ member(x,s)
   subset(emptyset,S)
   ...
```

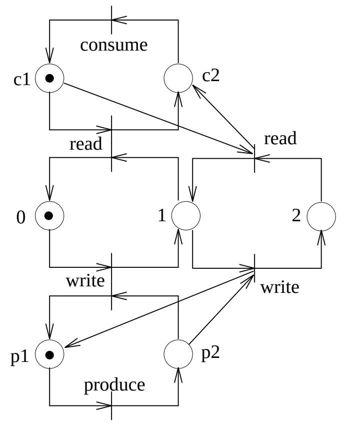
Src: Alagar and Periyasamy [AP98]

Larch

- algebraic (descriptive)
- describe the interfaces between components and observables
- set of equations describing relationships that must hold amongst operators
- textual
- hierarchy of “traits”
- based on predicate logic

Src: Alagar and Periyasamy [AP98]

Example Petri Nets Spec



Src: Ghezzi [GJM91]

Petri Nets

- “places” and transitions, markings of tokens to places
- asynchronous systems, transitions “fire” when they have their inputs
- lots of concurrency
- can get into deadlock
- limited ways to talk about data and data manipulation

Example Hawk Spec

```
refMachine flush = (outtrans) where
  (ifu_out,next_pc) = ifu pc flush
  pc = delay "pc" initpc next_pc
  rfouttrans = regFile ifu_out outtrans
  outtrans@alu_out = lift alu_trans rfouttrans
```

Hawk

- stream-based notation; set of equations on streams
- microarchitecture-specific components (alu, register file)
- textual
- built on top of the functional language Haskell

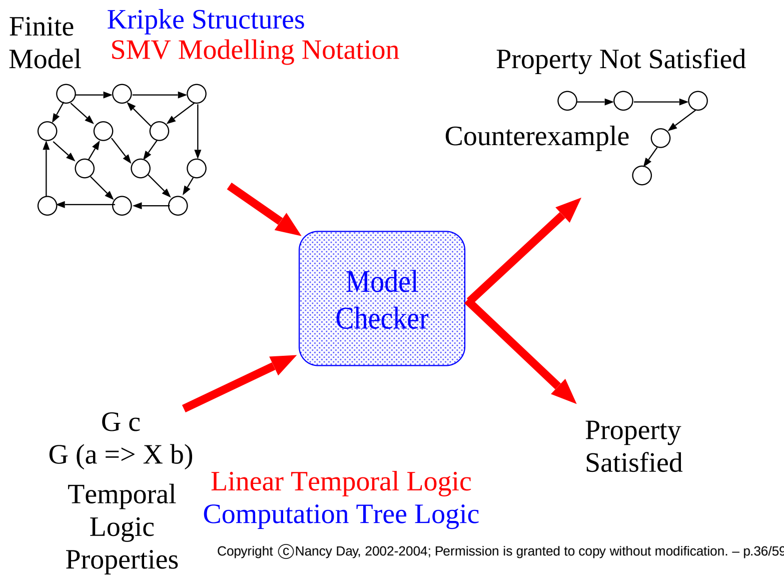
Related notations: Lava

Src: Matthews, Cook, and Launchbury [MCL98]

Today's Agenda

- SMV: concurrency and fairness (leftover from last class)
- Modelling notations
- Explict CTL model checking

Model Checking



Roadmap

Problem
Given a model $\mathcal{M}$ (usually a Kripke structure) that represents the behaviour of a system, and a temporal logic formula f that represents a desired property of the system, determine whether the model satisfies the formula:

$$\mathcal{M}, s \models f$$

Model Checking Algorithms that we'll study:

- Explicit CTL Model Checking: labelling a graph
- Symbolic CTL Model Checking: representing sets of states using propositional logic

Explicit CTL Model Checking

- Explicit representation of the Kripke structure as a labeled, directed graph with arcs given by pointers.
- Nodes represent the states in S
- Arcs represent the transition relation R
- Labels associated with the nodes are given by the function $L : S \rightarrow 2^{\mathsf{AP}}$."
- Introduce another labelling function, *label*, to represent temporal formulae that are true of that state

Sources: Clarke, Emerson, Sistla [CES86], Clarke, Grumberg, Peled [CGP99], Huth and Ryan [HR04]

Explicit CTL M/C

Start with all states labeled with the atomic propositions that are true in those states

$$\textit{label}(s) := L(s)$$

Continue to label states with the temporal formulae that we find are true of that state.

The algorithm works recursively over the structure of the formula.

Explicit CTL M/C

The depth of a formula is the number of levels of subformulae. A depth of 0 means there are no subformulae. A depth of 1 means there is one level of subformulae.

1. Decompose formula f into subformulae
2. *label*$(s) := L(s)$ labels all states that satisfy subformulae of depth 0
3. Find all states that satisfy subformulae of depth 1 and add the subformula to the labels of the states where it is true.

   ⋮

Explicit CTL M/C

n + 1. Find all states that satisfy subformulae of depth n (which is formula f) and add the subformula to the labels of the states where it is true.

n + 2. Test whether *label*(s) contains formula f

$$\mathcal{M}, s \models f \text{ iff } f \in \textit{label}(s)$$

To check whether a formula holds of all reachable states of the model, check whether the initial states are all labelled with f.

$$\mathcal{M} \models f \quad \text{iff} \quad \forall s_0 \in S_0 \bullet s_0 \in \{s \mid \mathcal{M}, s \models f\}$$

Explicit CTL M/C

Any CTL formula can be expressed in terms of
$\neg, \vee,$ **EX**, **EU**, **EG**.

$f_1 \wedge f_2 = \neg(\neg f_1 \vee \neg f_2)$

EF $f =$ **E**$[$**true U** $f]$

AX $f = \neg($**EX** $\neg f)$
AF $f = \neg($**EG** $\neg f)$
AG $f = \neg($**EF** $\neg f) = \neg($**E**$[$**true U** $\neg f])$
A$[f$ **U** $g] = \neg$**E**$[\neg g$ **U** $(\neg f \wedge \neg g)] \wedge \neg$**EG**$(\neg g)$

Explicit CTL M/C

Check $\neg f_1$
Add $\neg f_1$ to all *label*(s) if $f_1 \notin$ *label*(s)

Complexity: $O(|S|)$ (walk over the set of states)

Check $f_1 \vee f_2$
Add $f_1 \vee f_2$ to *label*(s) if either f_1 or f_2 are in *label*(s)

Complexity: $O(|S|)$ (walk over the set of states)

Recall: S is the set of states, R is the transition relation. $|S|$ is the size of the state space.

Explicit CTL M/C: EX

Label every state that has some successor labelled by f_1.

```
CheckEX($f_1$)
    $K = \{s \mid f_1 \in label(s)\}$;
    while $K \neq \emptyset$ do
        choose $s \in K$;
        $K := K \setminus \{s\}$;
        for all $(t, s) \in R$ do
            if EX$f_1 \notin label(t)$ then
                $label(t) := label(t) \cup \{$EX$f_1\}$;
```

Complexity: $O(|S| + |R|)$

Explicit CTL M/C: EU

Find all states that are labelled with f_2. Work backwards using R to find all states that can be reached by some path in which each state is labelled with f_1. Label all these states with $\mathbf{E}[f_1 \mathbf{U} f_2]$.

Explicit CTL M/C: EU

```
CheckEU($f_1, f_2$)
  $K := \{s \mid f_2 \in label(s)\}$;
  for all $s \in K$ do $label(s) := label(s) \cup \{\mathbf{E}[f_1 \mathbf{U} f_2]\}$;
  while $K \neq \emptyset$ do
    choose $s \in K$;
    $K := K \setminus \{s\}$;
    for all $(t, s) \in R$ do
      if $\mathbf{E}[f_1 \mathbf{U} f_2] \notin label(t)$ and $f_1 \in label(t)$ then
        $label(t) := label(t) \cup \{\mathbf{E}[f_1 \mathbf{U} f_2]\}$;
        $K := K \cup \{t\}$;
```

Complexity: $O(|S| + |R|)$

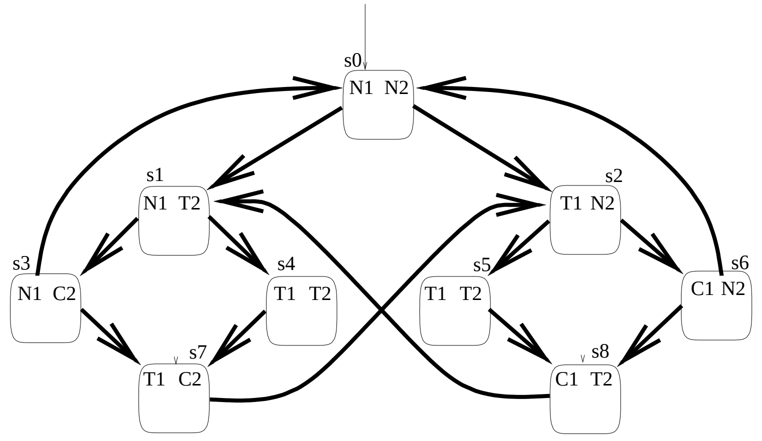
Example: Mutual Exclusion

Two processes:

- Each can be in one of three states, which they cycle through in the following order:
  1. non-critical (N)
  2. trying to enter the critical state (T)
  3. critical (C)
- interleave their steps with each other.

Both start in their non-critical sections of code.

Example: Mutual Exclusion



s_4 and s_5 are distinguished by which process was trying to get into its critical section first.

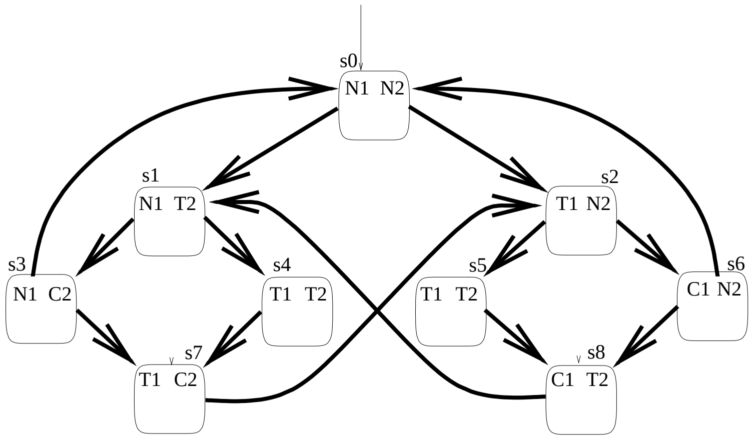
Properties of Mutual Exclusion

1. (safety) Only one process can be in its critical section at any time.

2. (liveness) Whenever a process wants to enter its critical section, it will eventually be permitted to do so.

Checking Safety

$\neg\mathbf{E}[\mathbf{true}\ \mathbf{U}\ \neg(\neg C1 \vee \neg C2)]$



Checking Liveness

Property: Whenever a process wants to enter its critical section, it will eventually be permitted to do so.

$$\mathbf{AG}(T1 \Rightarrow \mathbf{AF}\ C1)$$

alternatively:

$$\neg(\mathbf{E}[\mathbf{true}\ \mathbf{U}\ \neg(\neg T1 \vee \neg(\mathbf{EG}\ \neg C1))])$$

Explicit CTL M/C: EG (First Try)

EG f_1: Label all states labelled with f_1 with **EG** f_1 Repeat: delete the label **EG** f_1 from any state if none of its successors is labelled with **EG** f_1; until there is no change.

```
CheckEG($f_1$)
  $K = \{s \mid f_1 \in label(s)\}$;
  for all $s \in K$ do $label(s)$ := $label(s) \cup \{\mathbf{EG} f_1\}$;
  do
    $K$ := $\{s \mid \mathbf{EG} f_1 \in label(s)\}$;
    for all $t \in K$ do
      $label(t)$ := $label(t) - \{\mathbf{EG} f_1\}$;
      for all $(t, s) \in R$ do
        if $s \in K$ then $label(t)$ := $label(t) \cup \{\mathbf{EG} f_1\}$;
  until $K = \{s \mid \mathbf{EG} f_1 \in label(s)\}$;
```

Explicit CTL M/C: EG

We can do better than this algorithm by noting that for a state to satisfy **EG**f_1, it must have a path leaving it that eventually ends up in a loop in which f_1 is always true.

Strongly Connected Components

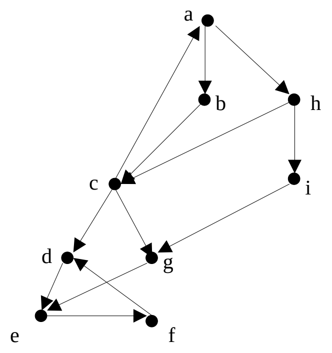
From Manber [Man89]:

A directed graph is strongly connected if, for every pair of vertices v and w, there is a path from v to w. In other words, it is possible to reach any vertex from any other vertex.

A strongly connected component is a maximal subset of the vertices such that its induced subgraph is strongly connected.

A strongly connected component is nontrivial "iff it either has more than one node or it contains one node with a self-loop" [CGP99], p. 36.

Strongly Connected Components



Tarjan's algorithm: $O(|S| + |R|)$

Explicit CTL M/C

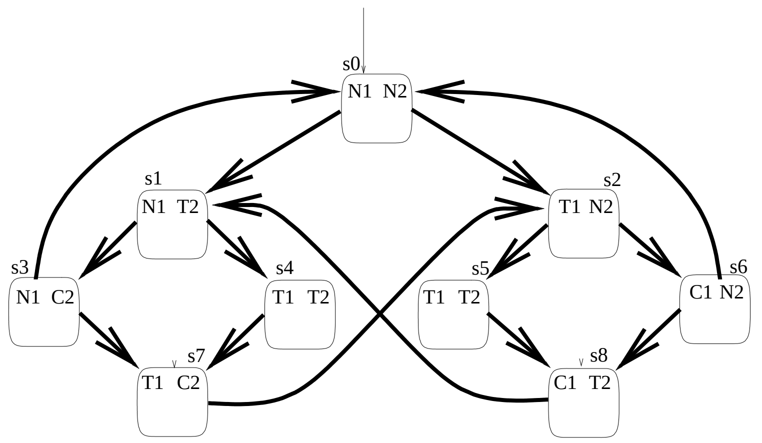
CheckEG(f_1), a more efficient algorithm:

```
  $S' := \{s \mid f_1 \in$ label$(s)\}$;
  $SCC := \{C \mid C$ is a nontrivial SCC of $S'\}$;
  $K := \bigcup_{C \in SCC} \{s \mid s \in C\}$;
  for all $s \in K$ do label$(s)$ := label$(s) \cup \{$EG$f_1\}$;
  while $K \neq \emptyset$ do
    choose $s \in K$;
    $K := K \setminus \{s\}$;
    for all $t$ such that $t \in S'$ and $(t, s) \in R$ do
      if EG$f_1 \notin$ label$(t)$ then
        label$(t)$ := label$(t)$ $nion$ $\{$EG$f_1\}$;
        $K := K \cup \{t\}$;
```

Complexity: $O(|S| + |R|)$

Checking Liveness

$\neg(\textbf{E}[\textbf{true U } \neg(\neg T1 \vee \neg(\textbf{EG } \neg C1))])$



Algorithm Complexity

- Worst case complexity of checking each subformula: $O(|S| + |R|)$
- For a formula of length n, there are n sub-formulae to check.
- Worst case complexity to check a formula of length n is $O(n \times (|S| + |R|))$

Thus, complexity is linear in the size of the formula and in the size of the state space. But the size of the state space is exponential in the number of parallel system components and in the number of variables!

State Space Explosion Problem

Summary

- SMV: concurrency and fairness (leftover from last class)
- Modelling notations
- Explict CTL model checking

Next class: Symbolic CTL model checking

References

[AP98] V. S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer, 1998.

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

[CGP99] Edmund Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[HR04] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science*. Cambridge University Press, Cambridge, 2004. Second Edition.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Publishing Company, Reading , Massachusetts, 1989.

[MCL98] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.