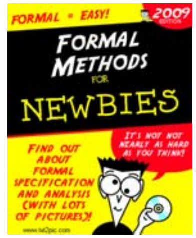
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification



Topic 2: Modeling, or
How to Describe Behaviour of Software Systems?

Juergen Dingel
Jan, 2009

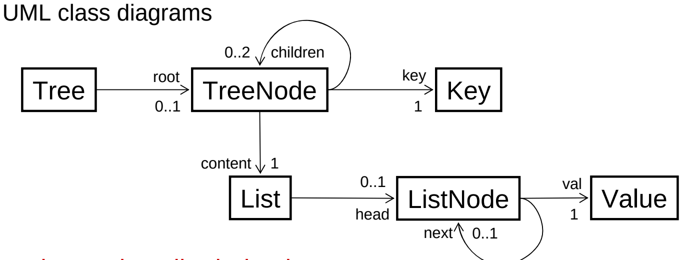
Spin Book:
- Appendix A (pages: 553 – 560)
- Chapter 6 (pages: 127 – 133)

CISC853: Contents

1. A few words about concurrency
2. Modeling: How to describe behaviour of a software system?
   - finite automata
3. Intro to 2 software model checkers
   - Bogor (Santos group at Kansas State University)
   - Spin (G. Holzmann at JPL)
4. Model checking I
   - algorithms for basic exploration
5. Specifying: How to express properties of a software system?
   - assertions, invariants, safety and liveness properties
   - Linear temporal logic (LTL) and Buechi automata
6. Model checking II
   - algorithms for checking properties
7. Overview of Software Model Checking tools

Two Views On Software

- Static
  - Describe the structure of a single state (snap shot)
    - Which objects exist?
    - How are they related?
  - Example:
    - UML class diagrams



  - They do not describe behaviour

Two Views On Software (Cont'd)

- Dynamic
  - Describe how the system evolves, that is, which executions it can exhibit
  - Could use
    - activity diagrams, sequence diagrams, collaboration diagrams, but they don't contain enough information for our purposes
    - Turing machines, but they contain too much information
  - Will use *finite state automata*

Finite State Automaton

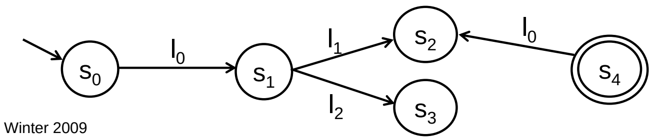
A *finite state automaton (machine)* is a tuple

$$(S, S_0, L, \delta, F)$$

where

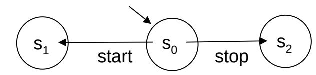
- S is a finite set of states
- S_0 is a set of distinguished initial states with $S_0 \subseteq S$
- L is a finite set of labels
- δ is a set of transitions with $\delta \subseteq (S \times L \times S)$
- F is a set of final states with $F \subseteq S$

Example:



Transitions

- System makes one step from one state to another
- Transitions can be enabled ...
  - transition (s_i, l, s_{i+1}) is enabled in state s_i iff $(s_i, l, s_{i+1}) \in \delta$
- ... or disabled
  - transition (s_i, l, s_{i+1}) is disabled in state s_i iff $(s_i, l, s_{i+1}) \notin \delta$
- Transition labels can contain information about, e.g.,
  - which process is carrying out the transition
  - how much time the transition is taking (Timed automata)
  - how likely it is that the transition is taken (probabilistic automata, Markov processes)
  - an instruction (e.g., guard, assignment, input, output)



Non-determinism

An FSA (S, S_0, L, δ, F) is *deterministic* iff

$\forall s, s_1, s_2 \in S.$

$\forall l \in L.$

$$(s, l, s_1) \in \delta \land (s, l, s_2) \in \delta \Rightarrow s_1 = s_2$$

An FSA is *non-deterministic* iff it's not deterministic.

- Non-determinism is useful to
  - model concurrent computations
    - to abstract from particular scheduling policies
  - model incompletely specified inputs or environments
    - to abstract from particular inputs or environments
  - write test harnesses

Runs and (Standard) Acceptance

A *run* (a.k.a., execution, trace) σ of an FSA (S, S_0, L, δ, F) is a possibly infinite sequence of transitions

$$(s_0, l_0, s_1)(s_1, l_1, s_2)(s_2, l_2, s_3)\ldots$$

such that $\forall 0 \leq i < |\sigma|.\ (s_i, l_i, s_{i+1}) \in \delta$.

An *ω-run* is an infinite run.

A *accepting run* of an FSA (S, S_0, L, δ, F) is a finite run

$$(s_0, l_0, s_1)(s_1, l_1, s_2)(s_2, l_2, s_3)\ldots(s_{n-1}, l_{n-1}, s_n)$$

such that $s_0 \in S_0$ and $s_n \in F$.

"An accepting run is a run that ends in a final state"

At this point, accepting runs are always finite!

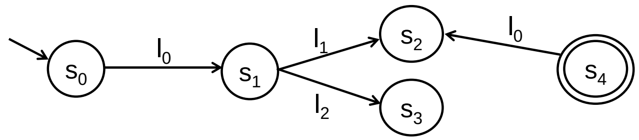
Reachable States

The *reachable states* (a.k.a., state space) of an FSA A is the set of all states along every run of A from an initial state.

"All states s to which there is a path from $s_0 \in S_0$ to s"

Example:

The FSA



has reachable states $\{s_0, s_1, s_2, s_3\}$

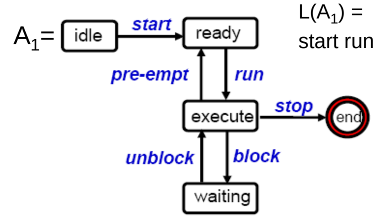
Words and Languages

A *word* w of an FSA A is the sequence of labels $l_0 l_1 l_2 \ldots l_n$ of an accepting run $(s_0, l_0, s_1)(s_1, l_1, s_2)(s_2, l_2, s_3)\ldots(s_{n-1}, l_{n-1}, s_n)$ of A.

The *language* L(A) of an FSA A is the set of words of A:

$L(A) = \{ w \mid w \text{ is word of } A\}$

Example:



$L(A_1)$ =
start run ((pre-empt run)* (block unblock)*)* stop

Theorem:
FSAs =
regular expressions =
regular languages

Asynchronous Composition

The *asynchronous composition* of 2 FSAs A and B is an FSA A||B such that A||B = (S, S_0, L, δ, F)

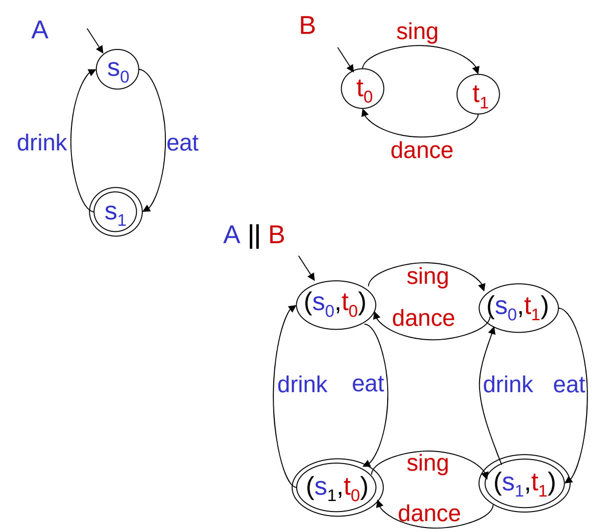
where

- S — is the Cartesian product $A.S \times B.S$
- S_0 — is $\{ (a_0, b_0) \in S \mid a_0 \in A.S_0 \wedge b_0 \in B.S_0\}$
- L — is the union $A.L \cup B.L$
- δ — is $\{((a_1, b), l, (a_2, b)) \in S \times L \times S \mid (a_1,l,a_2) \in A.\delta\} \cup$ $\{((a, b_1), l, (a, b_2)) \in S \times L \times S \mid (b_1,l,b_2) \in B.\delta\}$
- F — is $\{(s_1, s_2) \in S \mid s_1 \in A.F \vee s_2 \in B.F\}$

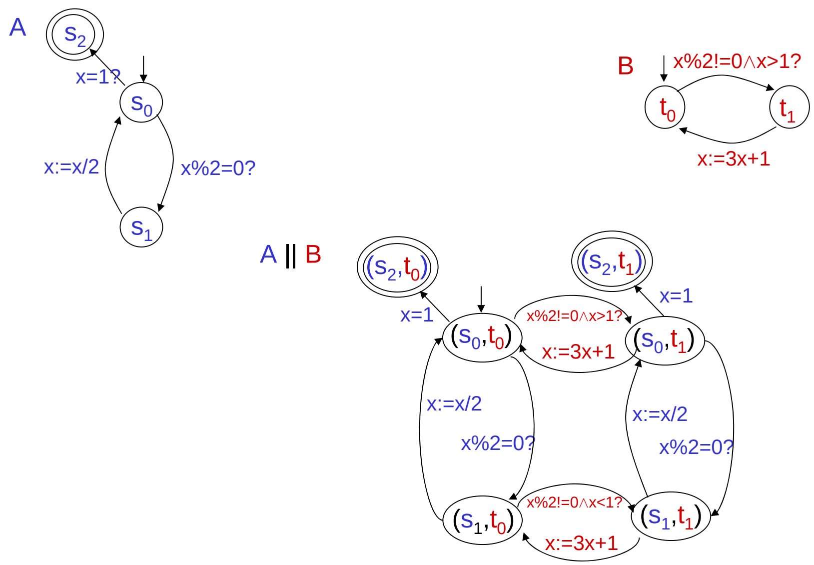
where A.S denotes the set of states of FSA A etc

$\wedge$ would result In a stronger acceptance condition

Example: Asynchronous Composition (1)



Example: Asynchronous Composition (2)



Asynchronous Composition (Cont'd)

- Form of parallel composition that allows each process to move completely independently of other processes
- Models our intuition about parallel or distributed processes executing at different speeds
- Introduces possibility of unfair executions, that is, executions in which, after some finite amount time, a process not executed anymore (e.g., P_1 P_2 P_1 P_2 P_1 P_1 P_1 …)
  - Only infinite executions can be unfair (more on fairness later)
- Related concepts:
  - asynchronous communication:
    - process can send w/o having to block until a matching receive is executed
      - E.g., communication channel is implemented as a buffer
    - Examples: Unix sockets, email
  - asynchronous circuits

Synchronous Composition

The *synchronous composition* of 2 FSAs A and B is an FSA $A \otimes B$ such that $A \otimes B = (S, S_0, L, \delta, F)$

where

S is $A.S \times B.S$

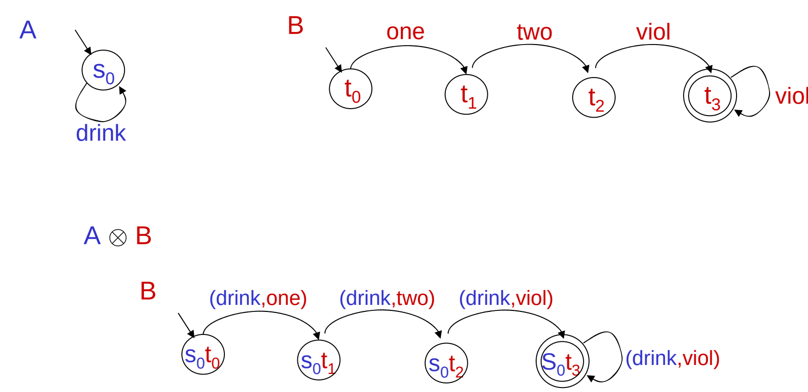
S_0 is $\{ (a_0, b_0) \in S \mid a_0 \in A.S_0 \wedge b_0 \in B.S_0 \}$

L is $A.L \times B.L$

δ is $\{((s,t), (l_1,l_2), (s',t')) \in S \times L \times S \mid (s, l_1, s') \in A.\delta \wedge (t, l_2, t') \in B.\delta\}$

F is $\{(s_1, s_2) \in S \mid s_1 \in A.F \vee s_2 \in B.F\}$

Example: Synchronous Composition



Synchronous Composition (Cont'd)

- Form of parallel composition in which all processes have to move in lockstep
- Models our intuition about the execution of processes being controlled by a global clock
- Related concepts:
  - synchronous communication:
    - process executing a send blocks until receiving process executes a matching receive
      - E.g., communication buffer is filled to capacity
    - Examples: telephone, rendezvous
  - synchronous circuits

Synchronous Composition (Cont'd)

- Useful for "monitoring", that is, the continuous observation (and checking) of one process by another
- Later, we will see how a property φ can be expressed with an automaton A_φ
- Then, A_φ is the monitor process
- For example, B (from before) is monitor process for *"# of 'drinks > 2"*

Observation:

For any process P, $P \otimes A\varphi$ has an accepting run iff P can satisfy φ

iff P can violate $\neg\varphi$

Interpreted FSAs (iFSAs)

- Previously,
  - states and labels could be anything
- Now,
  - states: uniquely describes particular "snapshot" during execution
    - values of all global variables in S, and
    - for all threads t,
      - value of program counter of t, and
      - values of local variables of t
  - labels: may describe how to get from one state to the next
    - statements (e.g., "y:=0;x:=x+y"), or
    - guards (e.g., "$x \geq 4$", "x even")
  - rest (i.e., initial and final states and transition relation): as before

State may have to contain more info, but for us, this suffices

Interpreted FSAs (iFSAs) (Cont'd)

- Formally, $A = (S, S_0, L, \delta, F)$ where

$S = \{(s^P,s^V) \mid s^P \in PC \rightarrow Loc \wedge s^V \in Vars \rightarrow D\}$ where

Vars = set of variables
D = set of values
PC = set of program counters
Loc = set of locations

} all finite

Notation: A.V = Vars // all variables used/defined in A

A.P = PC // all program counters used/defined in A

L **::= <stmt>** | **<guard>?** where

<stmt> ::= <var>:=<exp> | **<stmt>;<stmt>** | **...**

<var> $\in$ Var // variable used in labels is assigned value

// in states, i.e., varsUsedIn(A.L) $\subseteq$ A.V = Vars

<exp> is "expression over variables in A.V"

<guard> ::= <exp><relop><exp> | **<guard> <boolop> <guard>**

A different/ richer language for L is possible here

L sometimes also called **"Action Language"**

Interpreted FSAs (iFSAs) (Cont'd)

- But, now need to make sure that
  1. Labels are consistent with states:

     Definition of $(s,l,t)\in\delta$ can't ignore label l anymore

An *interpreted finite state automaton (machine)* is a tuple

$$(S, S_0, L, \delta, F)$$

where

S … // as on previous slide

$S_0 \subseteq S$ // as before

δ is $\{((s^P,s^V),l,(t^P,t^V)) \in S\times L\times S \mid (s^V,t^V)$ is consistent with $l\}$

where

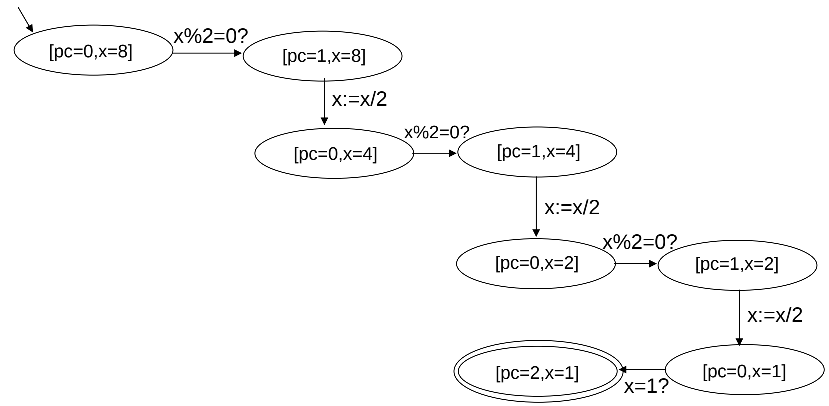
(s^V,t^V) consistent w/ stmt p iff *"execution of p from state s^V terminates in state t^V"*

(s^V,t^V) consistent w/ guard b iff *"b evaluates to 'true' in s^V"* and $s^V = t^V$

$F \subseteq S$ // as before

Interpreted FSAs (iFSAs) (Cont'd)

- **Example:** iFSA for 3n+1 problem with x=8 initially



Translating FSAs into iFSAs

- Let
  - FSA A = (S, S_0, L, δ, F) // L is some standard action language
  - Vars = varsIn(A.L) // variables used in labels in A
  - D some finite domain // e.g., $D = \{i \in \mathbb{N} \mid i \leq 100\}$
- We can compute the corresponding iFSA

$int_{Vars,D}(A) = (S', S'_0, L, \delta', F')$ where

$S' = int_{Vars,D}(S)$

$S'_0 = int_{Vars,D}(S_0)$

$\delta' = \{((pc_A=s, s^V), l, (pc_A=t, t^V)) \mid (s, l, t) \in \delta \wedge s^V \in Vars \rightarrow D \wedge t^V \in Vars \rightarrow D \wedge (s^V, t^V)$ consistent with $l\}$

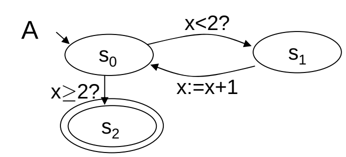
$F' = int_{Vars,D}(F)$

where

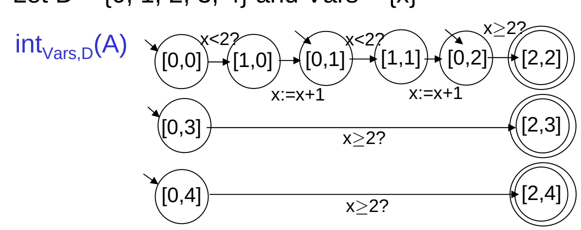
$int_{Vars,D}(S) = \{(pc_A=s, s^V) \mid s\in S \wedge s^V \in Vars\rightarrow D\}$

Translating FSAs into iFSAs (Cont'd)

- **Example 1:**



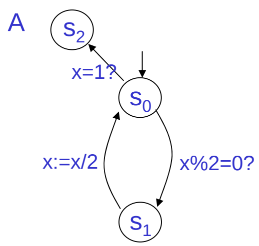
Let D = {0, 1, 2, 3, 4} and Vars = {x}



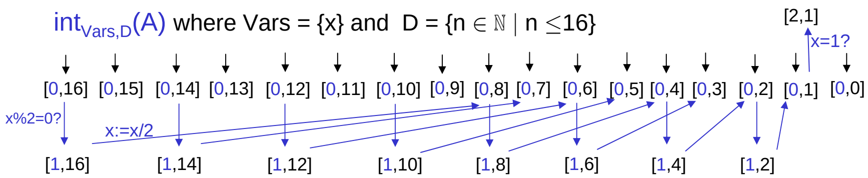
[i,j] abbreviates $[pc_A=s_i, x=j]$

Translating FSAs into iFSAs (Cont'd)

- **Example 2:**



$int_{Vars,D}(A)$ where Vars = {x} and D = {n ∈ ℕ | n ≤16}



[i,j] abbreviates [pc_A=s_i, x=j]

Interpreted FSAs (iFSAs) (Cont'd)

- Need to make sure that
  - 2) Composition operations result in consistent states:
    - "In state (s,t), variable assignment of s must not conflict with that of t"

The *asynchronous composition* of 2 FSAs A and B is an FSA A||B such that
A||B = (S, s_0, L, δ, F)
where
S is $\{((s^P,s^V), (t^P,t^V)) \in A.S \times B.S \mid s^V, t^V$ don't conflict}
… // unchanged

where s^V, t^V don't conflict iff

- $\forall x \in (A.V \cap B.V) . s^V(x) = t^V(x)$ and // A and B agree on shared vars
- $A.P \cap B.P = \emptyset$ // A and B use different program counters

Interpreted FSAs (iFSAs) (Cont'd)

- Need to make sure that
  - 2) Composition operations result in consistent states:
    - "In state (s,t), variable assignment of s must not conflict with that of t"

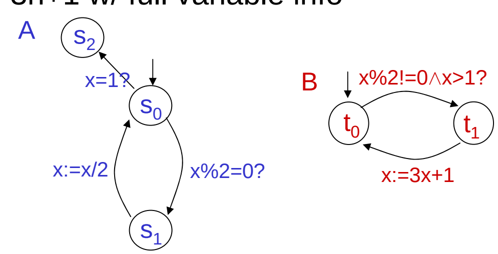
The *synchronous composition* of 2 FSAs A and B is an FSA A⊗B such that
A⊗B = (S, s_0, L, δ, F)
where
S is $\{((s^P,s^V), (t^P,t^V)) \in A.S \times B.S \mid s^V, t^V$ don't conflict}
… // unchanged

where s^V, t^V don't conflict iff

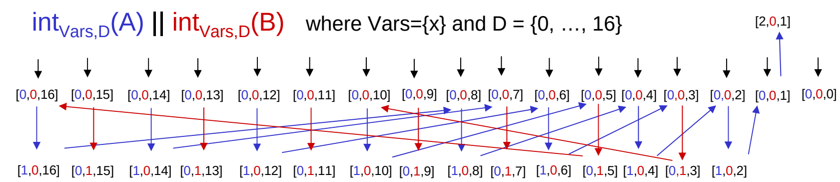
- $\forall x \in (A.V \cap B.V) . s^V(x) = t^V(x)$ and
- $A.P \cap B.P = \emptyset$

Interpreted FSAs (iFSAs) (Cont'd)

- **Example 1:** 3n+1 w/ full variable info



$int_{Vars,D}(A)$ || $int_{Vars,D}(B)$ where Vars={x} and D = {0, …, 16}

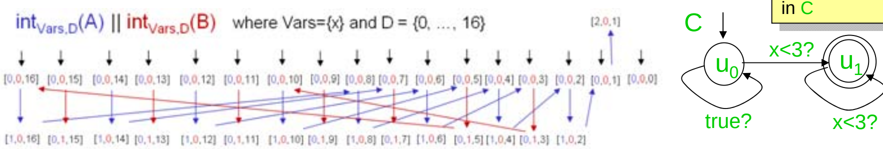


[i,j,k] abbreviates [pc_A=s_i, pc_B=t_j, x=k]
labels are elided

Interpreted FSAs (iFSAs) (Cont'd)

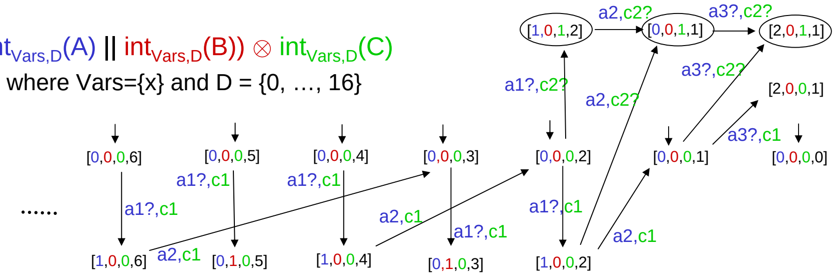
- **Example 2:** 3n+1 w/ full variable info w/ monitor C

Note non-determinism in C



$(int_{Vars,D}(A) \;||\; int_{Vars,D}(B)) \otimes int_{Vars,D}(C)$

where Vars={x} and D = {0, …, 16}



where a1 ≡ x%2=0, a2 ≡ x:=x/2, a3 ≡ x=1, c1 ≡ true, c2 ≡ x<3

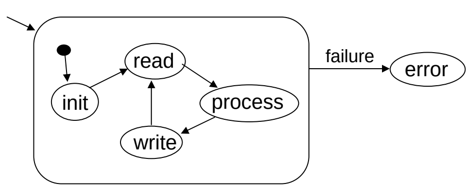
and [i,j,k,l] abbreviates $[pc_A=s_i, pc_B=t_j, pc_C=u_k, x=l]$

FSAs and iFSAs: Notes

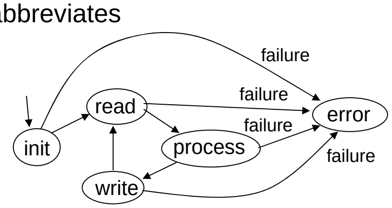
- Note
  - Typically, FSA used for representation, but
  - analysis always done on iFSA
- Given FSA A, corresponding iFSA int(A) computed either
  - before analysis
  - during analysis (on the fly)
    - This is what the "Semantics Engine" in the Spin Textbook does [Hol04, Chapter 7]

FSAs: Extensions

- Another notational abbreviation: **Composite (hierarchical) states**
  - With sequential substates: E.g.,



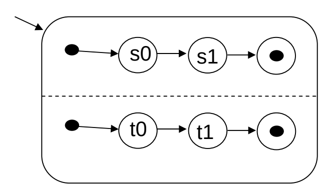
which abbreviates



These kinds of composite states are also known as "or-states" (b/c at most one of the immediate substates is active)

FSAs: Extensions (Cont'd)

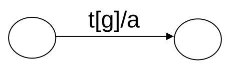
- Another notational abbreviation: **Composite (hierarchical) states**
  - With sequential substates
  - With concurrent substates (orthogonal regions)



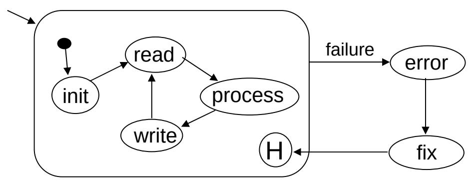
These kinds of composite states are also known as "and-states" (b/c, all immediate Substates are active)

FSAs: Extensions (Cont'd)

- Another notational abbreviation: **Composite (hierarchical) states w/ sequential & concurrent substates**
- Transition labeled with **trigger t, guard g and action** a



- **History states**



UML State Machines and StateCharts [Harel 1984] have all of these extensions

Example: StateCharts

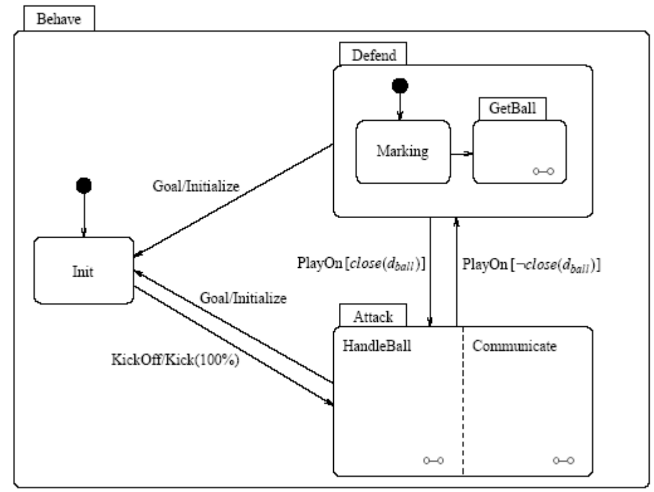


Figure 1: State machine for the overall behavior of soccer agents.

F. Stolzenburg. *From the Specification of Multiagent Systems by Statecharts to their Formal Analysis by Model Checking*. Fachberichte INFORMATIK. Universitaet Koblenz, Germany. June 2001.

Alternatives to FSAs

- Process algebras:
  - Calculus of Communicating Systems (CCS) [Milner, 1980]
  - Communicating Sequential Processes (CSP) [Hoare, 1985]
  - Lotos (Language of Temporal Ordering Specifications) [1989]
  - Estelle [1986]
- Petri nets [Petri, 1960]

Example: CCS

- Let

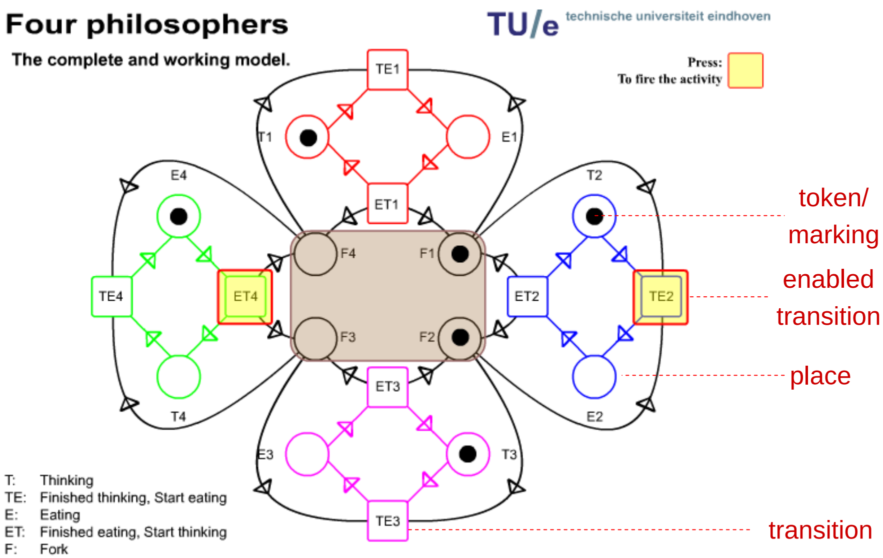
  $C \equiv coin.\overline{coffee}.C$ // coffee machine

  $P \equiv \overline{coin}.coffee.\overline{publish}.P$ // professor
- The (synchronous) parallel composition of C and P is

  $P \mid C \equiv \overline{coin}.coffee.\overline{publish}.P \mid coin.\overline{coffee}.C$
- Using the equational laws of CCS we can deduce that P | C is an infinite publishing machine:

  $P \mid C = \tau.\tau.\overline{publish}.(P \mid C) = \overline{publish}.(P \mid C)$
- CCS neatly captures basic notions of concurrency, e.g.,
  - communication, synchronization, input, output, observability

  and the rules that govern it, e.g.,
  - $P \mid Q = Q \mid P$
  - $a.P \mid \overline{a}.Q = \tau.(P \mid Q) = P \mid Q$
  - $a.P \mid b.Q = a.(P \mid b.Q) + b.(a.P \mid Q)$



http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/

Example: Petri Nets

Simple Petri Nets

A *Simple Petri Net* is a tuple

$N = (P, M_0, T, pre, post, M_F)$

where

- P is a finite set of places
- $M_0 \subseteq P$ is the initial marking
- T is a finite set of transitions
- pre: $T \rightarrow 2^P$ defines the pre-set of each transition
- post: $T \rightarrow 2^P$ defines the post-set of each transition
- $M_F \subseteq P$ is the final marking // a bit non-standard

Simple Petri Nets (Cont'd)

Let $N = (P, M_0, T, pre, post, M_F)$ and t be a transition in N (i.e., $t \in T$) and M be a marking in N (i.e., $M \subseteq P$)

- We say t is enabled in M iff

$$pre(t) \subseteq M$$

- If t enabled in M, then firing t in M creates new marking

$$M' = (M \setminus pre(t)) \cup post(t)$$

- Execution of N consists of repeated firings of enabled transitions from initial marking until final marking is reached

So, simple Petri nets seem similar to FSAs…

Simple Petri Nets as FSAs

Let $N = (P, M_0, T, pre, post, M_F)$.

Corresponding FSA_N is given by (S, S_0, L, δ, F) where

- $S = 2^P$
- $S_0 = \{M_0\} \subseteq S$
- $L = T$
- $\delta = \{(M, t, M') \in S \times L \times S \mid pre(t) \subseteq M \wedge M' = (M \setminus pre(t)) \cup post(t)\}$
- $F = \{M_F\} \subseteq S$

$\Rightarrow$ One-to-one correspondence between accepting runs in FSA_N and executions in simple Petri net N

Caveat: There is a whole lot more to Petri nets than what we've discussed here…

Modeling Behaviour of Systems

- Where are we?
  - We've decided to use FSAs to model the behaviour of software systems
  - Have seen:
    - Two types of parallel composition
    - Uninterpreted vs interpreted
    - Extensions
    - Some of the alternatives (e.g., Process algebra, Petri nets)
- What's next?
  - But, to be able to feed FSAs into a model checker, we need to be able to express FSAs textually in some language
  - Also, it would be nice if that language was as high-level (user-friendly) as possible.
  - 2 examples for modeling languages based on FSAs:
    - BIR (used by Bogor model checker)
    - Promela (used by Spin model checker)