

CIS 673: Fall 2006
Lecture 9/11: Modeling

- Goal is to describe control and interaction
- Explicit construct for nondeterminism
- Simple set of operators to build complex models
- Precise mathematical semantics
- Reactive modules: Extended FSMs with a rich communication mechanism
- Modeling languages: Unity (Chandy/Misra), CCS (Milner), CSP (Hoare), TLA (Lamport), tool languages: Spin (Promela), SMV, Murphi...

Variables

- System state is described by a collection of discrete (typed) variables
- Round-based execution: initialization round followed by a sequence of update rounds
- Convention: x refers to value at the beginning of a round, x' refers to value at the end of the round
- Convention: For a set X of variables, Σ_X denotes the set of all possible values of X
- Both initialization and update specified by guarded commands
- Guarded commands: introduced by Dijkstra, and popular in many modeling languages such as Unity, Murphi, Promela

Example: Variable specification

- Sample Variable declaration

$$\begin{aligned} &proc: \{0, 1, 2\} \\ &task_1, task_2: \mathbf{nat} \end{aligned}$$

$task_i$ gives units of CPU required by i -th task, and $proc$ gives current assignment of CPU to task (0 means idle)

- Sample initialization of $proc$

$$\begin{aligned} &\mathbf{init} \\ &\parallel true \rightarrow proc' := 0 \end{aligned}$$

- Sample update of $proc$

$$\begin{aligned} &\mathbf{update} \\ &\parallel task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0 \\ &\parallel task_1 > 0 \quad \quad \quad \rightarrow proc' := 1 \\ &\parallel task_1 = 0 \wedge task_2 > 0 \rightarrow proc' := 2 \end{aligned}$$

Guarded Commands

- A guarded assignment from X to Y consists of
 - Guard: boolean expression over X
 - For each y in Y , expression assign_y over X
 - Formally, defines a partial function from Σ_X to Σ_Y
- Guarded command is a set of guarded assignments:
 - Nondeterministic: Guards of multiple assignments may be true simultaneously
 - Executable: In every state, at least one guard is true
 - Formally, a guarded command defines a total relation from Σ_X to Σ_Y
 - Note: ordering of guarded assignments is unimportant

Simultaneous updates

- Variables may be updated simultaneously
- Cluster of variables updated simultaneously is called an atom

$proc: \{0, 1, 2\}; prior: \{1, 2\}$
 $task_1, task_2: \mathbf{nat}$

atom

controls $proc, prior$ **reads** $task_1, task_2, prior$

init

$\parallel true \rightarrow proc' := 0; prior' := 1$
 $\parallel true \rightarrow proc' := 0; prior' := 2.$

update

$\parallel task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0$
 $\parallel prior = 1 \wedge task_1 > 0 \rightarrow proc' := 1; prior' := 2$
 $\parallel prior = 1 \wedge task_1 = 0 \wedge task_2 > 0 \rightarrow proc' := 2$
 $\parallel prior = 2 \wedge task_2 > 0 \rightarrow proc' := 2; prior' := 1$
 $\parallel prior = 2 \wedge task_2 = 0 \wedge task_1 > 0 \rightarrow proc' := 1$

Await Dependencies

- All variables need not be updated simultaneously within a round
- Initialization and update of x may depend on y' , that is, the new value of y : x awaits y , or y precedes x
- Effect: each round is partitioned into subrounds, later subrounds can use values computed in earlier subrounds
- Note: all subrounds can use values computed in the previous round, that is, unprimed values

$proc: \{0, 1, 2\}; prior: \{1, 2\}$
 $task_1, task_2, new_1, new_2: \mathbf{nat}$

atom $A1$

controls new_1

initupdate

$\parallel true \rightarrow new'_1 := 0$

$\parallel true \rightarrow new'_1 := 5$

atom $A3$

controls $task_1$ **reads** $task_1$ **awaits** $new_1, proc$

init

$\parallel true \rightarrow task'_1 := new'_1$

update

$\parallel proc' = 1 \rightarrow task'_1 := task_1 + new'_1 - 1$

$\parallel proc' \neq 1 \rightarrow task'_1 := task_1 + new'_1$

Atoms

An atom U over variables X consists of

- Nonempty set $\text{ctr}X_U \subseteq X$ of controlled vars
- Set $\text{read}X_U \subseteq X$ of read variables
- Set $\text{await}X_U \subseteq X \setminus \text{ctr}X_U$ of awaited variables
- Initial command from $\text{await}X'_U$ to $\text{ctr}X'_U$
- Update command from $\text{read}X_U \cup \text{await}X'_U$ to $\text{ctr}X'_U$

Consistent atom sets

- No variable is controlled by more than one atom:
every variable is initialized/updated precisely once
- The transitive closure of the awaits dependencies is asymmetric
 - If x' depends on y' then y' can not depend on x'
 - It implies that there is a consistent way of ordering subrounds statically
 - No combinational loops
- Precedence relation $A \prec B$ over atoms is a partial order

Modules

- Reactive system interacting with the environment
- Module variables: private \cup interface \cup external;
pairwise disjoint
- Private: controlled by the module, not visible to
the environment
- Interface: controlled by the module, visible to
the environment (like outputs)
- External: controlled by the environment, visible
to the module (like inputs)
- Observables: interface and external
- Controlled variables: private and interface
- A consistent set of atoms over module variables
that control private and interface vars

Modules in Scheduler Example

module *Task1* **is**

interface *new₁*: **nat**

atom *A1* **controls** *new₁*

module *Task2* **is**

interface *new₂*: **nat**

atom *A2* **controls** *new₂*

module *Scheduler* **is**

private *task₁*, *task₂*: **nat**; *prior*: {1, 2}

interface *proc*: {0, 1, 2}

external *new₁*, *new₂*: **nat**

atom *A3* **controls** *task₁* **reads** *task₁* **awaits** *new₁*, *proc*

atom *A4* **controls** *task₂* **reads** *task₂* **awaits** *new₂*, *proc*

atom *A5* **controls** *proc*, *prior* **reads** *task₁*, *task₂*, *prior*

Module execution

- Order atoms U_1, \dots, U_n consistent with precedence relation (i.e. consider a linearization of \preceq)
- Initialization:
 - Initialize external vars to arbitrary values
 - For $i = 1 \dots n$, execute initial command of the atom U_i
- Each update round:
 - Choose arbitrary new values for external vars
 - For $i = 1 \dots n$, execute update command of the atom U_i
- Many orderings $U_1 \dots U_n$ are possible, but the particular choice has no influence
- Nonblocking interaction with the environment:
 - absolutely no constraints on how the external variables change
 - no deadlock (execution can always continue to next round)

Building complex modules

How to to define more complex modules?

- Renaming: Creating copies of a module
- Parallel Composition: Combining two modules
- Hiding: Scoping rule to make variables private

Compatibility for parallel composition

When can we compose two modules P and Q?

- No name-conflicts for private variables:
 - $\text{priv}X_P \cap X_Q$ is empty
 - $\text{priv}X_Q \cap X_P$ is empty
- Disjoint interface variables: $\text{intf}X_P \cap \text{intf}X_Q$ is empty. No write-shared variables
- Await dependencies of the two modules are not circular: the transitive closure of $(\prec_P \cup \prec_Q)$ is asymmetric: there is consistent way to order subrounds even after composition

Parallel Composition

If P and Q are compatible modules the $P \parallel Q$ is a module with

- Private variables: $\text{priv}X_P \cup \text{priv}X_Q$
- Interface variables: $\text{intf}X_P \cup \text{intf}X_Q$
- External variables: $(\text{extl}X_P \cup \text{extl}X_Q)$ minus interface variables.
- Set of atoms is the union of atoms of P and Q

Sample Composition: Scheduler System

$$\textit{SchedulerSystem} = \textit{Task1} \parallel \textit{Task2} \parallel \textit{Scheduler}$$

- Private vars: $task_1$, $task_2$, $prior$
- Interface vars: new_1 , new_2 , $proc$
- Atoms: A1, A2, A3, A4, A5

Note: \parallel is associative and commutative operator

Variable Renaming

$P[x := y]$ means rename the variable x to y ,
provided y not already a module variable

```
module Task is  
  interface new: nat  
  atom  
  controls new  
  initupdate  
     $\parallel$  true  $\rightarrow$  new' := 0  
     $\parallel$  true  $\rightarrow$  new' := 5
```

```
module Task1 is Task[new := new1]  
module Task2 is Task[new := new2]
```

Variable hiding

- For a module P and an interface variable x of P , **hide x in P** is a module obtained by removing x from interface vars and adding it to private vars
- Effect of hiding x : x is invisible to other modules (they cannot read x)
- Typical use: Compose two modules P and Q and hide the variables used for communication

module *SchedSyst2* **is hide** new_1, new_2 **in** *SchedulerSystem*

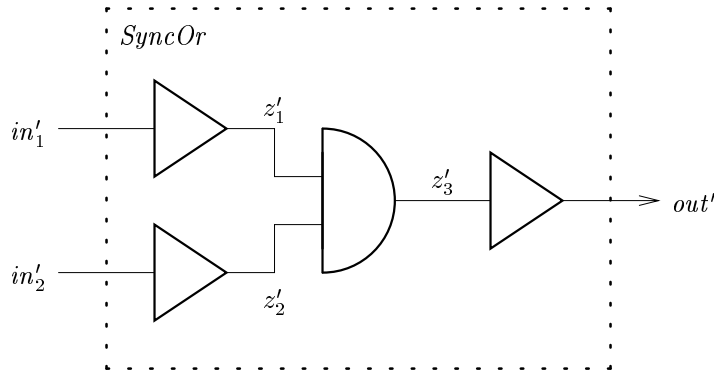
Combinational modules

- An atom U is combinational if
 - the set $\text{read}X_U$ of read variables is empty
 - initial and update commands are identical
- An atom is sequential if it is not combinational (such atoms behave differently in initial and update rounds)
- A module is combinational if all its atoms are combinational, otherwise sequential

Combinational Gates

```
module SyncNot is  
  interface out: bool  
  external in: bool  
  atom controls out awaits in  
  initupdate  
     $\parallel in' = 0 \rightarrow out' := 1$   
     $\parallel in' = 1 \rightarrow out' := 0$   
  
module SyncAnd is  
  interface out: bool  
  external in1, in2: bool  
  atom controls out awaits in1, in2  
  initupdate  
     $\parallel in'_1 = 0 \rightarrow out' := 0$   
     $\parallel in'_2 = 0 \rightarrow out' := 0$   
     $\parallel in'_1 = 1 \wedge in'_2 = 1 \rightarrow out' := 1$ 
```

Combining gates



```
module SyncOr is hide  $z_1, z_2, z_3$  in  
  —interface out  
  —external  $in_1, in_2$   
  || SyncAnd[ $in_1, in_2, out := z_1, z_2, z_3$ ]  
  || SyncNot[ $in, out := in_1, z_1$ ]  
  || SyncNot[ $in, out := in_2, z_2$ ]  
  || SyncNot[ $in := z_3$ ]
```

Latch

```
module SyncLatch is  
  private state : bool  
  interface out : bool  
  external set, reset : bool  
  atom ComputeOutput  
  controls out reads state  
  update  
     $\parallel$  true  $\rightarrow$  out' := state  
  atom ComputeNextState  
  controls state reads state awaits set, reset  
  initupdate  
     $\parallel$  set' = 1  $\rightarrow$  state' := 1  
     $\parallel$  reset' = 1  $\rightarrow$  state' := 0
```

Sequential Circuits

- Built from SyncAnd, SyncNot, and SyncLatch
- Wires correspond to primed variables, and latches to unprimed variables
- A subround typically corresponds to computation of a gate
- Example of 3-bit counter
- Partial order of awaits dependencies: every loop has to have at least one latch

Round-based execution

- Multiple subrounds allow complex forms of interaction: P writes to x , Q reads x , Q writes to y , P reads y
- Partial order ensures each round is finite
- Synchrony hypothesis (Berry):
 - Inputs do not change till a round is complete
 - Many synchronous languages such as ESTEREL