

CIS 673: Fall 2006

Lecture 9/13: Reactive Modules

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 $\text{ctr}X'_U$

A set of atoms is consistent if

- No variable is controlled by two atoms
- Await dependencies are acyclic and induce a partial order on execution of atoms

Modules

- 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

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

Shared variables protocols

- Communication via shared variables
- Typically, no references to primed vars, no awaits dependencies
- Disjoint interface variables means single-writer (multi-reader) vars
- Atomicity: single guarded assignment, can read and write multiple vars simultaneously
- Typical example: mutual exclusion for conflict resolution

Design problem: Complete the following

module Q_1 **is**

$pc_1: \{outC, reqC, inC\}$

atom controls pc_1

init

$\parallel true \rightarrow pc'_1 := outC$

update

$\parallel pc_1 = outC \rightarrow$

$\parallel pc_1 = outC \rightarrow pc'_1 := reqC$

$\parallel pc_1 = inC \rightarrow$

$\parallel pc_1 = inC \rightarrow pc'_1 := outC$

module Q_2 **is**

$pc_2: \{outC, reqC, inC\}$

atom controls pc_2

init

$\parallel true \rightarrow pc'_2 := outC$

update

$\parallel pc_2 = outC \rightarrow$

$\parallel pc_2 = outC \rightarrow pc'_2 := reqC$

$\parallel pc_2 = inC \rightarrow$

$\parallel pc_2 = inC \rightarrow pc'_2 := outC$

Design problem

- Goal: both should not be in critical section at the same time (pc_1 and pc_2 should not be simultaneously equal to inC)
- Design issues:
 - should we add more variables
 - which vars should be interface/external
 - add guarded assignments to update pc from $reqC$ to inC
- Note the use of nondeterminism:
 - update from $outC$ to $reqC$ takes arbitrarily many rounds
 - update from inC to $outC$ takes arbitrarily many rounds

Synchronous Solution

module Q_1 **is**

interface $pc_1: \{outC, reqC, inC\}$

external $pc_2: \{outC, reqC, inC\}$

atom controls pc_1 **reads** pc_1, pc_2

update

| | |
|----------------------------------------------|-----------------------------|
| $\parallel pc_1 = outC$ | \rightarrow |
| $\parallel pc_1 = outC$ | $\rightarrow pc'_1 := reqC$ |
| $\parallel pc_1 = reqC \wedge pc_2 \neq inC$ | $\rightarrow pc'_1 := inC$ |
| $\parallel pc_1 = inC$ | \rightarrow |
| $\parallel pc_1 = inC$ | $\rightarrow pc'_1 := outC$ |

module Q_2 **is**

interface $pc_2: \{outC, reqC, inC\}$

external $pc_1: \{outC, reqC, inC\}$

atom controls pc_2 **reads** pc_1, pc_2

update

| | |
|--------------------------------------------|-----------------------------|
| $\parallel pc_2 = outC$ | \rightarrow |
| $\parallel pc_2 = outC$ | $\rightarrow pc'_2 := reqC$ |
| $\parallel pc_2 = reqC \wedge pc_1 = outC$ | $\rightarrow pc'_2 := inC$ |
| $\parallel pc_2 = inC$ | \rightarrow |
| $\parallel pc_2 = inC$ | $\rightarrow pc'_2 := outC$ |

Asynchrony

- Processes running at different and unknown speeds
- Distributed environment: processes running on different processors
- Multiprogramming: processes scheduled on the same processor
- We model asynchrony using nondeterminism

Lazy atoms

- Sleep assignment: all the controlled variables stay unchanged
- Lazy atom: one of the choices in the update command is the sleep assignment
- y is copied into x once in a while:

atom controls x reads x awaits y

update

$\parallel true \rightarrow x' := y'$

$\parallel true \rightarrow x' := x$

Possible trajectory xy : 00, 02, 33, 36, 38, 11, ...

- Abbreviated to

lazy atom controls x reads x awaits y

update

$\parallel true \rightarrow x' := y'$

Asynchronous modules

- Stuttering: all interface variables stay unchanged (no observable update)
- Asynchronous module: All interface variables are controlled by lazy atoms
- Implies that the module may stutter in each update round
- The speed at which interface variables change is unrelated to the speed at which external variables change
- Private variables can be used so as to not to lose inputs
- Requiring asynchronous modules: harder to program, easier to implement

Asynchronous Mutual Exclusion

module P_1 **is**

interface $pc_1: \{outC, reqC, inC\}; x_1: \mathbf{bool}$

external $pc_2: \{outC, reqC, inC\}; x_2: \mathbf{bool}$

lazy atom controls pc_1, x_1 **reads** pc_1, pc_2, x_1, x_2

update

| | |
|---------------------------------------------|------------------------------------------|
| $\parallel pc_1 = outC$ | $\rightarrow pc'_1 := reqC; x'_1 := x_2$ |
| $\parallel pc_1 = reqC \wedge pc_2 = outC$ | $\rightarrow pc'_1 := inC$ |
| $\parallel pc_1 = reqC \wedge x_1 \neq x_2$ | $\rightarrow pc'_1 := inC$ |
| $\parallel pc_1 = inC$ | $\rightarrow pc'_1 := outC$ |

module P_2 **is**

interface $pc_2: \{outC, reqC, inC\}; x_2: \mathbf{bool}$

external $pc_1: \{outC, reqC, inC\}; x_1: \mathbf{bool}$

lazy atom controls pc_2, x_2 **reads** pc_1, pc_2, x_1, x_2

update

| | |
|--------------------------------------------|-----------------------------------------------|
| $\parallel pc_2 = outC$ | $\rightarrow pc'_2 := reqC; x'_2 := \neg x_1$ |
| $\parallel pc_2 = reqC \wedge pc_1 = outC$ | $\rightarrow pc'_2 := inC$ |
| $\parallel pc_2 = reqC \wedge x_1 = x_2$ | $\rightarrow pc'_2 := inC$ |
| $\parallel pc_2 = inC$ | $\rightarrow pc'_2 := outC$ |

$Pete = \mathbf{hide} \ x_1, x_2 \ \mathbf{in} \ P_1 \parallel P_2$

Asynchronous circuits

- Inputs and outputs of a gate do not change synchronously (not clocked)
- Advantage: clock signal need not be distributed throughout the chip
- Disadvantage:
 - Hard to design
 - Less tool support available
- Excellent candidate to apply formal verification
- Each gate can be
 - Stable
 - Unstable: input signal has not propagated
 - Hazard: if input changes before output

Asynchronous AND gate

module *AsyncAnd* **is**

private $pc: \{S, U, H\}$

interface $out: \mathbf{bool}$

external $in_1, in_2: \mathbf{bool}$

lazy atom controls out **reads** pc, out

update

$\parallel pc = U \rightarrow out' := \neg out$

$\parallel pc = H \rightarrow out' := \neg out$

atom controls pc **reads** pc, out **awaits** in_1, in_2, out

init

$\parallel \text{And}(in'_1, in'_2, out') \rightarrow pc' := S$

$\parallel \neg \text{And}(in'_1, in'_2, out') \rightarrow pc' := U$

update

$\parallel pc = S \wedge \neg \text{And}(in'_1, in'_2, out') \rightarrow pc' := U$

$\parallel pc = U \wedge \text{And}(in'_1, in'_2, out') \wedge out' \neq out \rightarrow pc' := S$

$\parallel pc = U \wedge \text{And}(in'_1, in'_2, out') \wedge out' = out \rightarrow pc' := H$

Events

- Events are like messages
- An event is modeled by toggling of a boolean variable:
 - $x!$ stands for $x' := \neg x$; issuing an event
 - $x?$ stands to $x' \neq x$; checking for presence
- $x : \mathbf{event}$ declares x to be of type event: value of x is unimportant, only changes are.
- In general, a channel is modeled by an event x and contents c :
 - Sending a message v : $x!; c' := v$
 - Receiving a message into y : $x? \rightarrow y' := c'$

Passive Modules

- Intuition: If environment does not change, then the module also may do nothing
- Different from asynchrony:
 - Interface variables can be synchronized with external variables
 - But no way to count rounds (i.e. no explicit notion of time)
- Suitable for discrete events systems
- Modeled using nondeterminism

Passive atoms

- Conditional sleep assignment for atom U : If awaited variables are unchanged then controlled variables stay unchanged
- An atom U is passive if it is lazy or combinational or contains the conditional sleep assignment, else is active
- Sample passive atom

**atom controls n reads n, x awaits x
update**

$\parallel x' \neq x \rightarrow n' := n + 1$

$\parallel x' = x \rightarrow n' := n$

Counts the number of times x changes

- Abbreviated as

**passive atom controls n reads n, x awaits x
update**

$\parallel x' \neq x \rightarrow n' := n + 1$

Closed versus Open Modules

- A module is closed if it has no external variables
- Behavior of a closed module is completely determined, and not influenced by the environment
- A module may be closed by taking parallel composition with another module that models the environment

Determinism

- Deterministic atom: Precisely one guard true at any time (initial and update commands define one-to-one mappings)
- Deterministic module: only deterministic atoms
- Private determinism: all private vars are controlled by deterministic atoms (state determined by observation)
- Nondeterministic: even private vars may change nondeterministically (hard to analyze)

Alternative Models

- Literature has many models for concurrency.
Important issues:
 - What constitutes a state?
 - What constitutes a round?
 - How do components interact in a round?
 - When are two systems equivalent?
- Examples:
 - Automata-style: Moore/Mealy machines, I/O automata
 - Process algebras: CSP, CCS, ACSR, pi-calculus...
 - Petri nets
 - Synchronous languages: ESTEREL, Lustre, Signal, Statecharts
 - Interleaving shared memory: Unity, Promela,

Significant Semantic Differences

- Blocking interaction (eg. CSP):
 - send ($x!$) is enabled only when there is a matching receiver, if not wait
 - Strong synchronization primitive
 - Influences theory of process equivalence
- Dynamic resolution of dependencies in Esterel:
 - there may be loops in awaits dependencies
 - but in every state, there is a consistent way to order atoms
- Interleaving composition:
 - Composing atoms corresponds to merging guarded commands
 - Less transitions
 - Write-shared variables allowed
 - Modular reasoning is difficult

Significant Syntactic Restrictions

- Instead of guarded commands, richer languages with sequencing, looping, and conditionals
- Many languages allow mixing sequencing and parallel composition eg. $a; b; (c \parallel d)$
- Synchronous languages allow interrupts
- Promela allows dynamic creation of processes
- Mobile calculi: instead of simple values, communication allows exchange of functions or processes

Interleaving mutual exclusion

module *Pete* **is**

interface $pc_1, pc_2: \{outC, reqC, inC\}$

private $x: \{1, 2\}$

init

$\parallel true \rightarrow pc'_1 := outC; pc'_2 := outC$

update

$\parallel pc_1 = outC \rightarrow pc'_1 := reqC; x' := 1$

$\parallel pc_1 = reqC \wedge pc_2 = outC \rightarrow pc'_1 := inC$

$\parallel pc_1 = reqC \wedge x = 2 \rightarrow pc'_1 := inC$

$\parallel pc_1 = inC \rightarrow pc'_1 := outC$

$\parallel pc_2 = outC \rightarrow pc'_2 := reqC; x' := 2$

$\parallel pc_2 = reqC \wedge pc_1 = outC \rightarrow pc'_2 := inC$

$\parallel pc_2 = reqC \wedge x = 1 \rightarrow pc'_2 := inC$

$\parallel pc_2 = inC \rightarrow pc'_2 := outC$