

University of Toronto Department of Computer Science

Lecture 16: Modelling "events"

- Focus on states or events?
 - ↳ E.g. SCR table-based models
 - ↳ Explicit event semantics
- Comparing notations for state transition models
 - ↳ FSMs vs. Statecharts vs. SCR
- Checking properties of state transition models
 - ↳ Consistency Checking
 - ↳ Model Checking, using Temporal Logic
- When to use formal methods

© Easterbrook 2004 1

University of Toronto Department of Computer Science

What are we modelling?

Application Domain Machine Domain

D - domain properties R - requirements

C - computers P - programs

s - specification

- Starting point:
 - ↳ States of the environment
 - ↳ Events that occur in the application domain (that change the state of the environment)
- Requirements expressed as:
 - ↳ Constraints over states and events of the application domain
 - > E.g. "When the aircraft is in the air, the pilot should be prevented from accidentally engaging the reverse thrust"
- To get to a specification:
 - ↳ For each relevant application domain event, find a corresponding **input event**
 - ↳ For each relevant state, ensure there is a way for the machine to detect it
 - ↳ For each required action, find a corresponding **output event**

© Easterbrook 2004 2

University of Toronto Department of Computer Science

Tabular Specifications: SCR

Four Variable Model:

Environment Monitored Variables input devices software output devices Controlled Variables Environment

input data items output data items

also: Assertions, Scenarios, ...

Dictionaries: Monitored/Controlled Variables, Types, Constants
Tables: Mode Transition Tables, Event Tables, Condition Tables
SCR Specification

© Easterbrook 2004 3

University of Toronto Department of Computer Science

SCR basics

- Modes and Mode classes
 - ↳ A mode class is a finite state machine, with states called *system modes*
 - > Transitions in each mode class are triggered by *events*
 - ↳ Complex systems described using several mode classes operating in parallel
 - ↳ System State is defined as:
 - > the system is in exactly one mode from each mode class...
 - > ...and each variable has a unique value
- Events
 - ↳ Single input assumption - only one input event can occur at once
 - ↳ An event occurs when any system entity changes value
 - > An input event occurs when an input variable changes value
 - ↳ Notation:
 - > We may need to refer to both the old and new value of a variable:
 - > Used primed values to denote values after the event
 - > $@T(c) = \neg c \wedge c'$ e.g. $@T(y=1) = y \neq 1 \wedge y' = 1$
 - > $@F(c) = c \wedge \neg c$
 - ↳ A conditioned event is an event with a predicate
 - > $@T(c) \text{ WHEN } d = \neg c \wedge c' \wedge d$

© Easterbrook 2004 Source: Adapted from Heitmeyer et. al. 1996. 4

University of Toronto Department of Computer Science

Defining Mode Classes

→ **Mode Class Tables**

- Define a (disjoint) set of *modes* (states) that the software can be in.
- A complex system will have many different modes classes
 - Each mode class has a mode table showing the events that cause transitions between modes
- A mode table defines a *partial function* from modes and events to modes

→ **Example:**

Current Mode	Powered on	Too Cold	Temp OK	Too Hot	New Mode
Off	@T @T @T	- t -	t - -	- - t	Inactive Heat AC
Inactive	@F -	- @T	- -	- @T	Off Heat AC
Heat	@F -	- -	- @T	- -	Off Inactive
AC	@F -	- -	- @T	- -	Off Inactive

© Easterbrook 2004 Source: Adapted from Heimeyer et. al. 1996. 5

University of Toronto Department of Computer Science

Defining Controlled Variables

→ **Event Tables**

- defines how a controlled variable changes in response to input events
- Defines a *partial function* from modes and events to variable values
- Example:

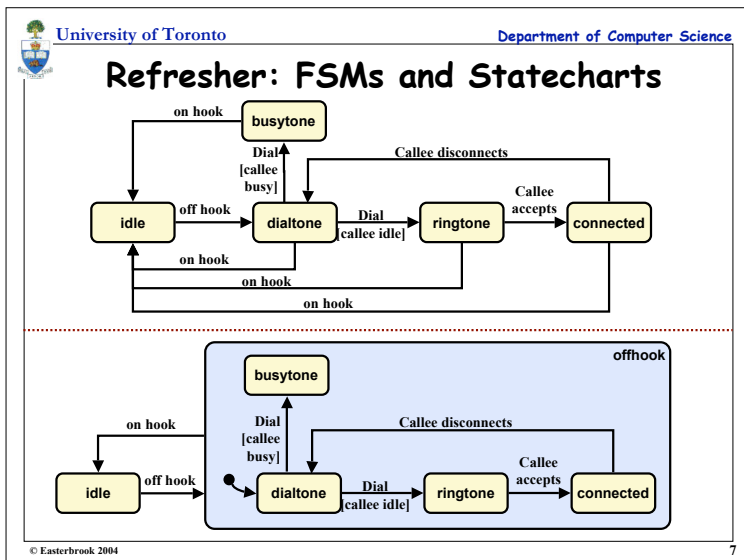
Modes		
Heat, AC	@C(target)	never
Inactive, Off	never	@C(target)
Ack_tone =	Beep	Clang

→ **Condition Tables**

- defines the value of a controlled variable under every possible condition
- Defines a *total function* from modes and conditions to variable values
- Example:

Modes		
Heat	target - temp ≤ 5	target - temp > 5
AC	temp - target ≤ 5	temp - target > 5
Inactive, Off	true	never
Warning light =	Off	On

© Easterbrook 2004 Source: Adapted from Heimeyer et. al. 1996. 6



University of Toronto Department of Computer Science

SCR Equivalent

Current Mode	offhook	dial	callee offhook	New Mode
Idle	@T	-	-	Dialtone
Dialtone	-	@T	F	Ringtone
	-	@T	T	Busytone
	@F	-	-	Idle
Busytone	@F	-	-	Idle
Ringtone	-	-	@T	Connected
	@F	-	-	Idle
Connected	-	-	@F	Dialtone
AC	@F	-	-	Idle

→ **Interpretation:**

- In Dialtone: @T(offhook) WHEN callee_offhook takes you to Ringing
- In Ringtone: @F(offhook) takes you to Idle
- Etc...

© Easterbrook 2004 8



State Machine Models vs. SCR

- All 3 models on previous slides are (approx) equivalent
- State machine models
 - ↳ Emphasis is on states & transitions
 - > No systematic treatment of events
 - > Different event semantics can be applied
 - ↳ Graphical notation easy to understand (?)
 - ↳ Composition achieved through statechart nesting
 - ↳ Hard to represent complex conditions on transitions
 - ↳ Hard to represent real-time constraints (e.g. elapsed time)
- SCR models
 - ↳ Emphasis is on events
 - > Clear event semantics based on changes to environmental variables
 - > Single input assumption simplifies modelling
 - ↳ Tabular notation easy to understand (?)
 - ↳ Composition achieved through parallel mode classes
 - ↳ Hard to represent real-time constraints (e.g. elapsed time)



formal analysis

- Consistency analysis and typechecking
 - ↳ "Is the formal model well-formed?"
 - > [assumes a modeling language where well-formedness is a useful thing to check]
- Validation:
 - ↳ Animation of the model on small examples
 - ↳ Formal challenges:
 - > "if the model is correct then the following property should hold..."
 - ↳ 'What if' questions:
 - > reasoning about the consequences of particular requirements;
 - > reasoning about the effect of possible changes
 - ↳ State exploration
 - > E.g. use a model checking to find traces that satisfy some property
 - ↳ Checking application properties:
 - > "will the system ever do the following..."
- Verifying design refinement
 - > "does the design meet the requirements?"



E.g. Consistency Checks in SCR

- Syntax
 - ↳ did we use the notation correctly?
- Type Checks
 - ↳ do we use each variable correctly?
- Disjointness
 - ↳ is there any overlap between rows of the mode tables?
 - > ensures we have a deterministic state machine
- Coverage
 - ↳ does each condition table define a value for all possible conditions?
- Mode Reachability
 - ↳ is there any mode that cannot ever happen?
- Cycle Detection
 - ↳ have we defined any variable in terms of itself?



Model Checking

- Has revolutionized formal verification:
 - ↳ emphasis on partial verification of partial models
 - > E.g. as a debugging tool for state machine models
 - ↳ fully automated
- What it does:
 - ↳ Mathematically - computes the "satisfies" relation:
 - > Given a temporal logic theory, checks whether a given finite state machine is a model for that theory.
 - ↳ Engineering view - checks whether properties hold:
 - > Given a model (e.g. a FSM), checks whether it obeys various safety and liveness properties
- How to apply it in RE:
 - ↳ The model is an (operational) Specification
 - > Check whether particular requirements hold of the spec
 - ↳ The model is (an abstracted portion of) the Requirements
 - > Carry out basic validity tests as the model is developed
 - ↳ The model is a conjunction of the Requirements and the Domain
 - > Formalise assumptions and test whether the model respects them

University of Toronto Department of Computer Science

Model Checking Basics

- Build a finite state machine model
 - ↳ E.g. PROMELA - processes and message channels
 - ↳ E.g. SCR - tables for state transitions and control actions
 - ↳ E.g. RSML - statecharts + truth tables for action preconditions
- Express validation property as a logic specification
 - ↳ Propositions in first order logic (for invariants)
 - ↳ Temporal Logic (for safety & liveness properties)
 - E.g. CTL, LTL, ...
- Run the model checker:
 - ↳ Computes the value of: $model \models property$
- Explore counter-examples
 - ↳ If the answer is 'no' find out why the property doesn't hold
 - ↳ Counter-example is a trace through the model

© Easterbrook 2004 13

University of Toronto Department of Computer Science

Temporal Logic

- LTL (Linear Temporal Logic)
 - ↳ Expresses properties of infinite traces through a state machine model
 - ↳ adds two temporal operators to propositional logic:
 - ◇p - p is true eventually (in some future state)
 - p - p is true always (now and in the future)
- CTL (Computational Tree Logic)
 - ↳ branching-time logic - can quantify over possible futures
 - ↳ Each operator has two parts:
 - EX p - p is true in some next state
 - AX p - p is true in all next states
 - EF p - along some path, p is true in some future state
 - AF p - along all paths...
 - E[p U q] - along some path, p holds until q holds;
 - A[p U q] - along all paths...
 - EG p - along some path, p holds in every state;
 - AG p - along all paths...

© Easterbrook 2004 14

University of Toronto Department of Computer Science

Example

```

stateDiagram-v2
    state idle
    state dialtone
    state ringtone
    state connected
    state busytone

    idle --> dialtone : off hook
    dialtone --> busytone : Dial [callee busy]
    dialtone --> ringtone : Dial [callee idle]
    busytone --> dialtone : Callee disconnects
    ringtone --> connected : Callee accepts
    connected --> busytone : off hook
  
```

- Sample Properties
 - ↳ If you are connected you can hang up:
 - AG(CONNECTED → EX(~OFFHOOK))
 - ↳ If you are connected, hanging up always disconnects you:
 - AG(CONNECTED → AX(~OFFHOOK → ~CONNECTED))
 - ↳ A connection doesn't start until you pick up the phone:
 - AG(~CONNECTED → A[~CONNECTED U OFFHOOK])
 - ↳ If you make a call, the phone cannot ring without returning to idle first:
 - AG((RINGTONE ∨ BUSYTONE) → A[~RINGING U IDLE])

© Easterbrook 2004 15

University of Toronto Department of Computer Science

Complexity Issues

- The problem:
 - ↳ Model Checking is exponential in the size of the model and the property
 - ↳ Current MC engines can explore 10^{120} states...
 - using highly optimized data structures (BDDs)
 - ...and state space reduction techniques
 - ↳ ...that's roughly 400 propositional variables
 - integer and real variables cause real problems
 - ↳ Realistic models are often too large to be model checked
- The solution:
 - ↳ Abstraction:
 - Replace related groups of states with a single superstate
 - Replace real & integer variables with propositional variables
 - ↳ Projection:
 - Slice the model to remove parts unrelated to the property
 - ↳ Compositional verification - break large model into smaller pieces
 - (But it's hard to verify that the composition preserves properties)

© Easterbrook 2004 16



Formal Methods in RE

→What to formalize in RE?

- models of requirements knowledge (so we can reason about them)
- specifications of requirements (so we can document them precisely)

Why formalize in RE?

- Remove ambiguity and improve precision
- Provides a basis for verification that the requirements have been met
- Can reason about the requirements
 - Properties of formal requirements models can be checked automatically
 - Can test for consistency, explore the consequences, etc.
- Can animate/execute the requirements
 - Helps with visualization and validation
- Will have to formalize eventually anyway
 - RE is all about bridging from the informal world to a formal machine domain

Why people don't formalize in RE

- Formal Methods tend to be lower level than other analysis techniques
 - They force you to include too much detail
- Formal Methods tend to concentrate on consistent, correct models
 - ...but most of the time your models are inconsistent, incorrect, incomplete...
- People get confused about which tools are appropriate:
 - E.g. modeling program behaviour vs. modeling the requirements
 - formal methods advocates get too attached to one tool!
- Formal methods require more effort
 - ...and the payoff is deferred



FM in practice

→ From Shuttle Study [Crow & DiVito 1996]

- More errors found in the process of formalizing the requirements than were found in the formal analysis
 - Formalization forces you to be precise and explicit, hence reveals problems
 - Formal analysis then finds fewer, but more subtle problems
- Typical errors found include:
 - inconsistent interfaces
 - incorrect requirements (system does the wrong thing in response to an input)
 - clarity/maintainability problems

Issue Severity	With FM	Existing
High Major	2	0
Low Major	5	1
High Minor	17	3
Low Minor	6	0
Totals	30	4



Using Formal Methods

→ Selective use of Formal Methods

- Amount of formality can vary
- Need not build complete formal models
 - Apply to the most critical pieces
 - Apply where existing analysis techniques are weak
- Need not formally analyze every system property
 - E.g. check safety properties only
- Need not apply FM in every phase of development
 - E.g. use for modeling requirements, but don't formalize the system design
- Can choose what level of abstraction (amount of detail) to model