



Lecture 14: 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



What are we modelling?

Application Domain

Machine Domain

D - domain properties

R - requirements

S - specification



C - computers

P - programs

→ Starting point:

↪ **States** of the environment

↪ (Application domain) **events** 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"

➤ I.e. "In state X, event Y shall be prevented"

→ To get to a specification:

↪ For each relevant application domain event, find a corresponding **input event**

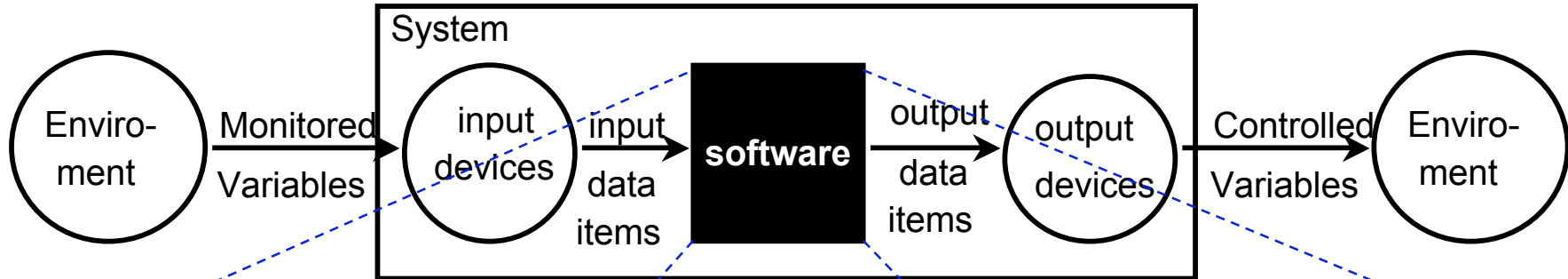
↪ For each relevant state, ensure there is a way for the machine to detect it

↪ For each required action, find a corresponding **output event**



Tabular Specifications: SCR

Four Variable Model:



Dictionaries:

Monitored/Controlled Variables

Variable	Type	Initial Value	Units	Warning Flag	boolean	false	Other Flag	boolean	initial

Types

Type	Base Type	Values	Units	Warning Level	enumerated	low	mid	high	Temperature

Constants

Constant	Type	Value	Units	Low Temp	integer	5degrees	CHigh	Temp	integer	23deg

Tables:

Mode Transition Tables

Current Mode	Powered	Too Cold	Temp OK	Too Hot	Mode Off	T-Err

Event Tables

Modes	Events	No Failure	@T	(INMODE)	never	Sensor	Fail	@T	(re

Condition Tables

Modes	Events	No Failure	re	re	false	AC	Failure	temp	>	temp0

also: Assertions, Scenarios, ...

SCR Specification



SCR basics

Source: Adapted from Heitmeyer et. al. 1996.

→ 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
- ↪ Overall system state is:
 - the system is in exactly one mode from each mode class...
 - ...and each variable has a unique value

→ Events

- ↪ An event occurs when any system entity changes value
 - An input event occurs when an input variable changes value
- ↪ **Single input assumption** - only one input event can occur at once
- ↪ Notation:
 - We may need to refer to both the old and new value of a variable:
 - 'Primes' denote values after the event:
 $@T(c) \equiv \neg c \wedge c'$ e.g. $@T(y=1) \equiv y \neq 1 \wedge y' = 1$
 $@F(c) \equiv c \wedge \neg c'$
- ↪ A conditioned event is an event with a predicate
 $@T(c) \text{ WHEN } d \equiv \neg c \wedge c' \wedge d$



Defining Mode Classes

Source: Adapted from Heitmeyer et. al. 1996.

→ Mode Class Tables

- ↪ Define a (disjoint) set of *modes* (states) that the software can be in.
- ↪ Each mode class has a **mode table** showing which events cause mode changes
 - 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	-	Inactive
	@T	t	-	-	Heat
	@T	-	-	t	AC
Inactive	@F	-	-	-	Off
	-	@T	-	-	Heat
	-	-	-	@T	AC
Heat	@F	-	-	-	Off
	-	-	@T	-	Inactive
AC	@F	-	-	-	Off
	-	-	@T	-	Inactive



Defining Controlled Variables

Source: Adapted from Heitmeyer et. al. 1996.

→ 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

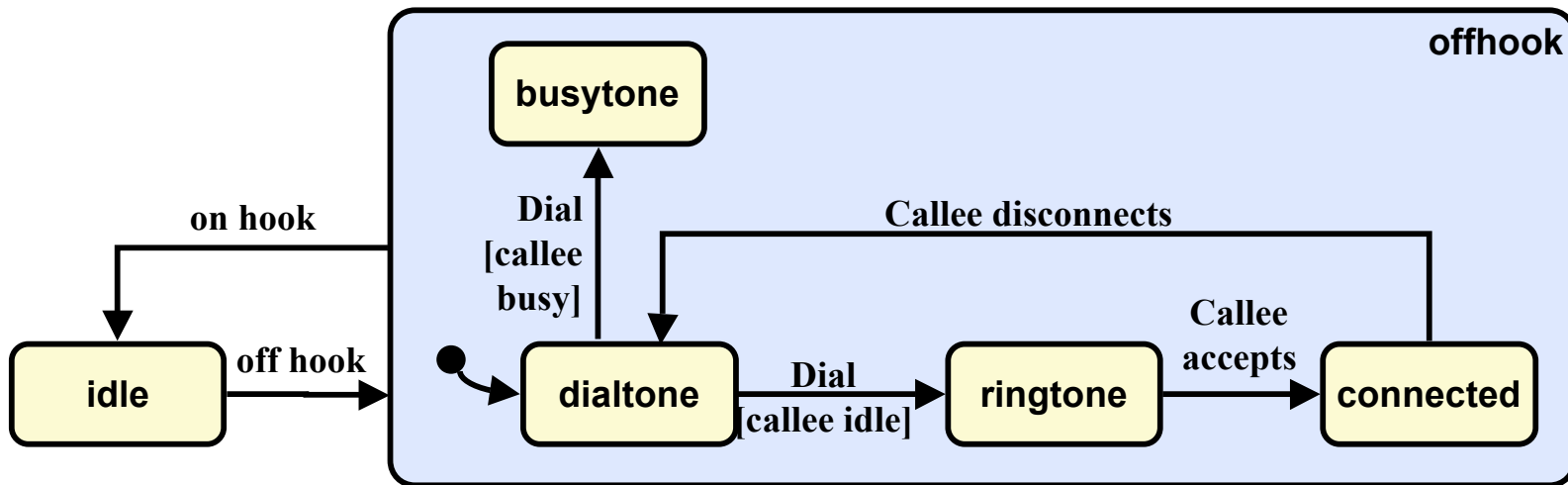
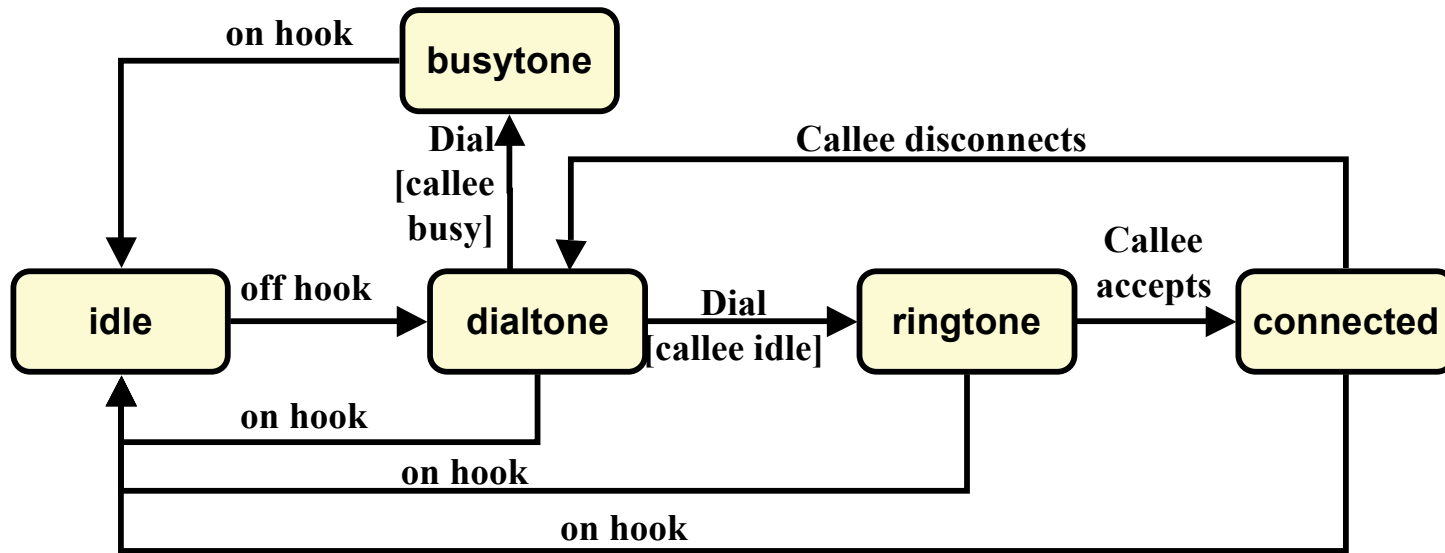
→ Condition Tables

- ↪ 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 \leq 5	target - temp $>$ 5
AC	temp - target \leq 5	temp - target $>$ 5
Inactive, Off	true	never
Warning light =	Off	On



Refresher: FSMs and Statecharts





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...



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 notation 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

↪ Reachability analysis

- E.g. use a model checker 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?"



Model Checking

→ A debugging tool for state machine models

↳ emphasis on *partial* verification of *partial* models

→ 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 state machine model, checks whether the model obeys:

➤ **safety** properties - a 'bad' state cannot be reached

➤ **liveness** properties - something good will eventually happen

→ How to apply it in RE:

↳ ... if the model is an (operational) Specification:

➤ Check whether particular requirements hold of the spec

↳ ... if the model is (an abstracted portion of) the Requirements

➤ Carry out basic validity tests as the model is developed

↳ ... if the model is a conjunction of the Requirements and the Domain

➤ Formalise assumptions and test whether the model respects them



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



Temporal Logic

→ LTL (Linear Temporal Logic)

↪ Expresses properties of infinite traces through a state machine model

↪ adds two temporal operators to propositional logic:

$\diamond p$ - p is true eventually (in some future state)

$\square p$ - 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 states

$AX p$ - 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;

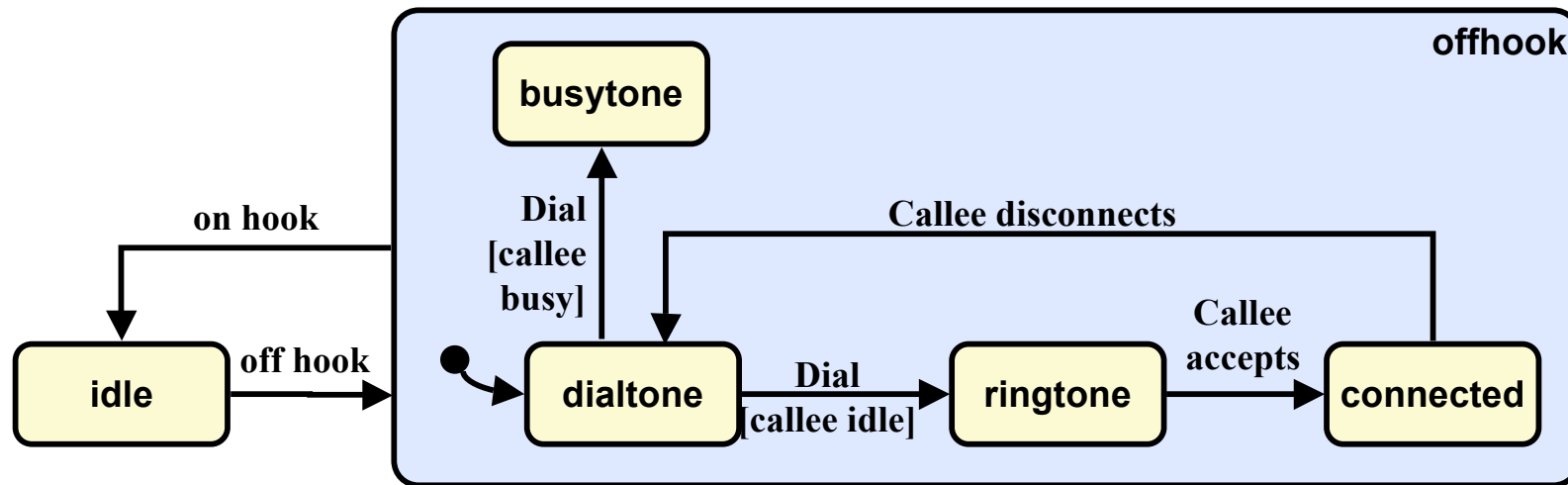
$A[p U q]$ - along all paths...

$EG p$ - along some path, p holds in every state;

$AG p$ - along all paths...



Example



→ Sample Properties

- ↪ If you are connected you can hang up:
 $AG(CONNECTED \rightarrow EX(\neg OFFHOOK))$
- ↪ If you are connected, hanging up always disconnects you:
 $AG(CONNECTED \rightarrow AX(\neg OFFHOOK \rightarrow \neg CONNECTED))$
- ↪ A connection doesn't start until you pick up the phone:
 $AG(\neg CONNECTED \rightarrow A[\neg CONNECTED \cup OFFHOOK])$
- ↪ If you make a call, the phone cannot ring without returning to idle first:
 $AG((RINGTONE \vee BUSYTONE) \rightarrow A[\neg RINGING \cup IDLE])$



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)



Summary

→ SCR vs UML Statecharts

- ↪ Tabular view allows more detail - e.g. complex conditions
- ↪ Graphical view shows hierarchical structure more clearly
- ↪ Event Semantics
 - SCR has a precisely defined meaning for "events"
 - UML Statecharts do not
- ↪ Uses:
 - UML statecharts good for sketches, design models
 - SCR good for writing precise specifications

→ Analysis:

- ↪ "Model checkers" are debugging tools for state machine models
- ↪ Write temporal logic properties and test whether they hold
- ↪ Very good at finding subtle errors in specifications