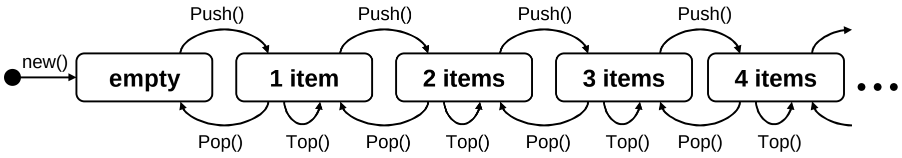
Lecture 8, Part 1: Modelling “State”

- **What is State?**
  - statespace for an object
  - concrete vs. abstract states
- **Finite State Machines**
  - states and transitions
  - events and actions
- **Modularized State machine models: Statecharts**
  - superstates and substates
  - Guidelines for drawing statecharts

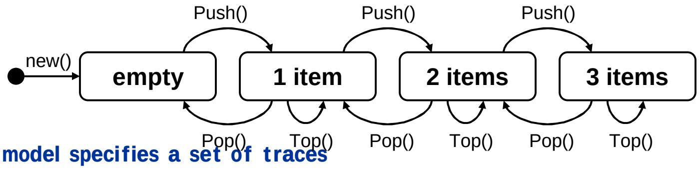
Getting objects to behave

- **All objects have “state”**
  - The object either exists or it doesn’t
  - If it exists, then it has a value for each of its attributes
  - Each possible assignment of values to attributes is a “state”
    - (and non-existence is a state, although we normally ignore it)
- **E.g. For a stack object**



What does the model mean?

- Finite State Machines
  - There are a finite number of states (all attributes have finite ranges)
    - E.g. imagine a stack with max length = 3



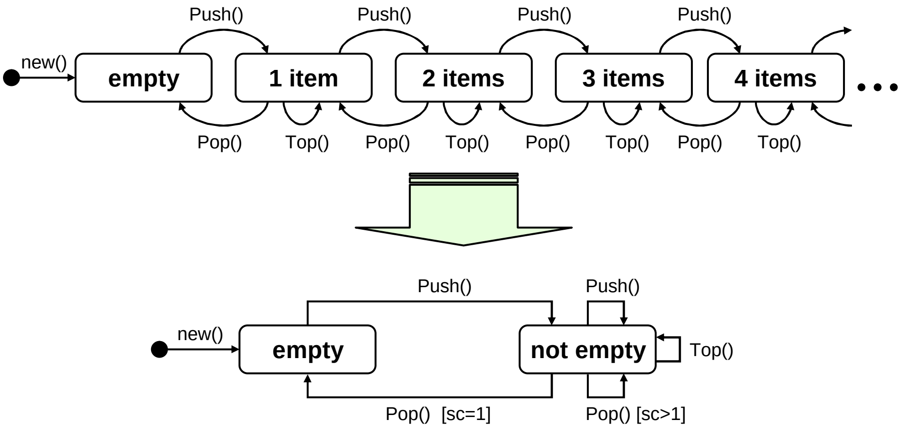
  - The model specifies a set of traces
    - E.g. new();Push();Push();Top();Pop();Push()…
    - E.g. new();Push();Pop();Push();Pop()…
    - There may be an infinite number of traces (and traces may be of infinite length)
  - The model excludes some behaviours
    - E.g. no trace can start with a Pop()
    - E.g. no trace may have more Pops than Pushes
    - E.g. no trace may have more than 3 Pushes without a Pop in between

---

Abstraction

- The state space of most objects is enormous
  - State space size is the product of the range of each attribute
    - E.g. object with five boolean attributes: 2^5+1 states
    - E.g. object with five integer attributes: (maxint)^5+1 states
    - E.g. object with five real-valued attributes: …?
  - If we ignore computer representation limits, the state space is infinite
- Only part of that state space is "interesting"
  - Some states are not reachable
  - Integer and real values usually only vary within some relevant range
  - We're usually not interested in the actual values, just certain ranges:
    - E.g. for Age, we may be interested in age<18; 18=age=65; and age>65
    - E.g. for Cost, we may only be interested in cost=budget, cost=0, cost>budget, and cost>(budget+10%)

Collapsing the state space

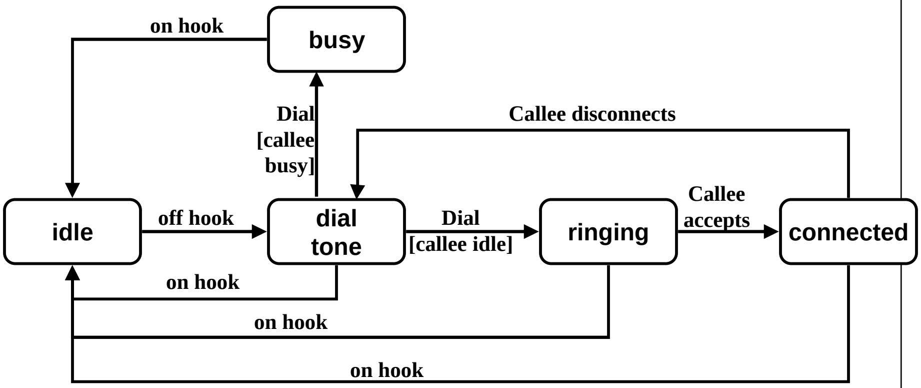


- The abstraction usually permits more traces
    - E.g. this model does not prevent traces with more pops than pushes
    - But it still says something useful

What are we modelling?

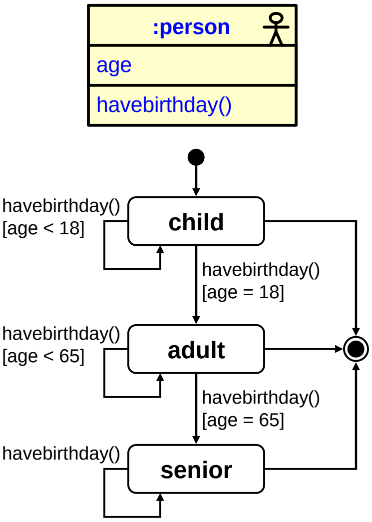
- Observed states of an application domain entity?
    - E.g. a phone can be idle, ringing, connected, …
    - Model shows the states an entity can be in, and how events can change its state
    - This is an indicative model
- Required behaviour of an application domain entity?
    - E.g. a telephone switch shall connect the phones only when the callee accepts the call
    - Model distinguishes between traces that are desired and those that are not
    - This is an optative model
- Specified behaviour of a machine domain entity?
    - E.g. when the user presses the 'connect' button the incoming call shall be connected
    - Model specifies how the machine should respond to input events
    - This is an optative model, in which all events are shared phenomena

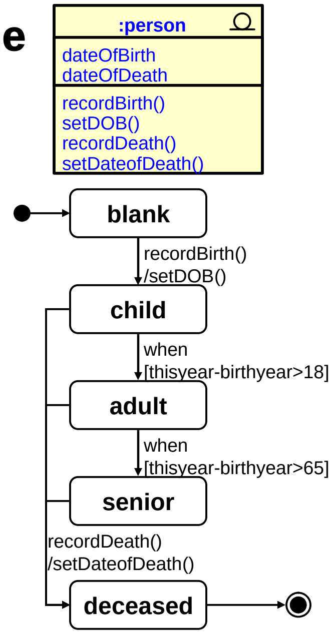
Is this model indicative or optative?



---

the world vs. the machine





StateCharts

➲ Notation:

- **States**
  - “interesting” configurations of the values of an object’s attributes
  - may include a specification of action to be taken on entry or exit
  - States may be nested
  - States may be “**on**” or “**off**” at any given moment
- **Transitions**
  - Are enabled when the state is “on”; disabled otherwise
  - Every transition has an **event** that acts as a trigger
  - A transition may also have a condition (or **guard**)
  - A transitions may also cause some action to be taken
  - When a transition is enabled, it can **fire** if the trigger event occurs and it guard is true
  - Syntax: event [guard] / action
- **Events**
  - occurrence of stimuli that can trigger an object to change its state
  - determine when transitions can fire

Superstates

➲States can be nested, to make diagrams simpler

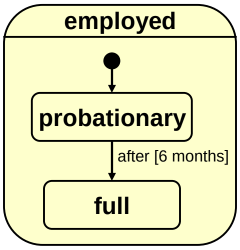
- A superstate consists of one or more states.
- Superstates make it possible to view a state diagram at different levels of abstraction.

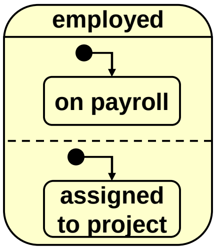
OR superstates

- when the superstate is “on”, only one of its substates is “on”

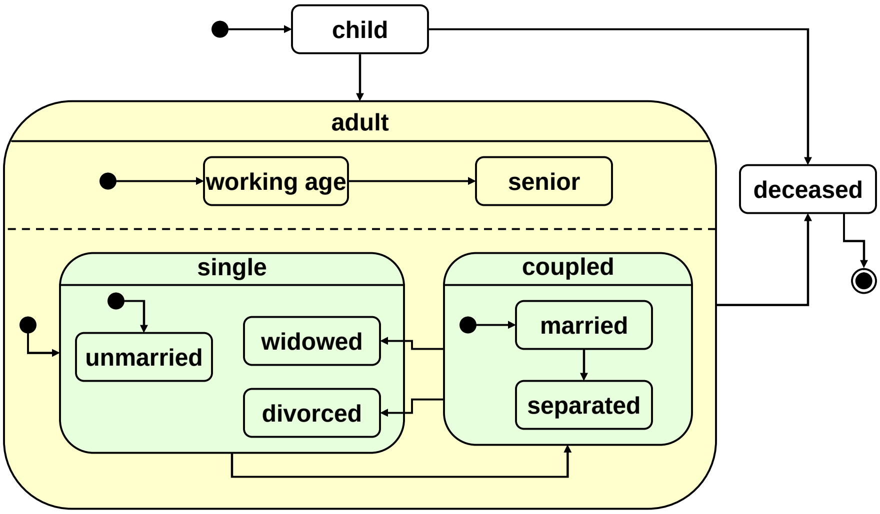
AND superstates (concurrent substates)

- When the superstate is “on”, all of its states are also “on”
- Usually, the AND substates will be nested further as OR superstates





A more detailed example



States in UML

- **A state represents a time period during which**
  - A predicate is true
    - e.g. (budget - expenses) > 0,
  - An action is being performed, or an event is awaited:
    - e.g. checking inventory for order items
    - e.g. waiting for arrival of a missing order item
- **States can have associated activities:**
  - do/activity
    - carries out some activity for as long as the state is “on”
  - entry/action and exit/action
    - carry out the action whenever the state is entered (exited)
  - include/stateDiagramName
    - “calls” another state diagram, allowing state diagrams to be nested

Events in UML

- Events are happenings the system needs to know about
  - Must be relevant to the system (or object) being modelled
  - Must be modellable as an instantaneous occurance (from the system's point of view)
    - E.g. completing an assignment, failing an exam, a system crash
  - Are implemented by message passing in an OO Design
- In UML, there are four types of events:
  - *Change events* occur when a condition becomes true
    - denoted by the keyword 'when'
    - e.g. when[balance < 0]
  - *Call events* occur when an object receives a call for one of its operations to be perfomed
  - *Signal events* occur when an object receives an explicit (real-time) signal
  - *Elapsed-time events* mark the passage of a designated period of time
    - e.g. after[10 seconds]

Checking your Statecharts

- Consistency Checks
  - All events in a statechart should appear as:
    - operations of an appropriate class in the class diagram
  - All actions in a statechart should appear as:
    - operations of an appropriate class in the class diagram and
- Style Guidelines
  - Give each state a unique, meaningful name
  - Only use superstates when the state behaviour is genuinely complex
  - Do not show too much detail on a single statechart
  - Use guard conditions carefully to ensure statechart is unambiguous
    - Statecharts should be deterministic (unless there is a good reason)
- You probably shouldn't be using statecharts if:
  - you find that most transitions are fired "when the state completes"
  - many of the trigger events are sent from the object to itself
  - your states do not correspond to the attribute assignments of the class

Lecture 8, Part 2:
Modelling “events”

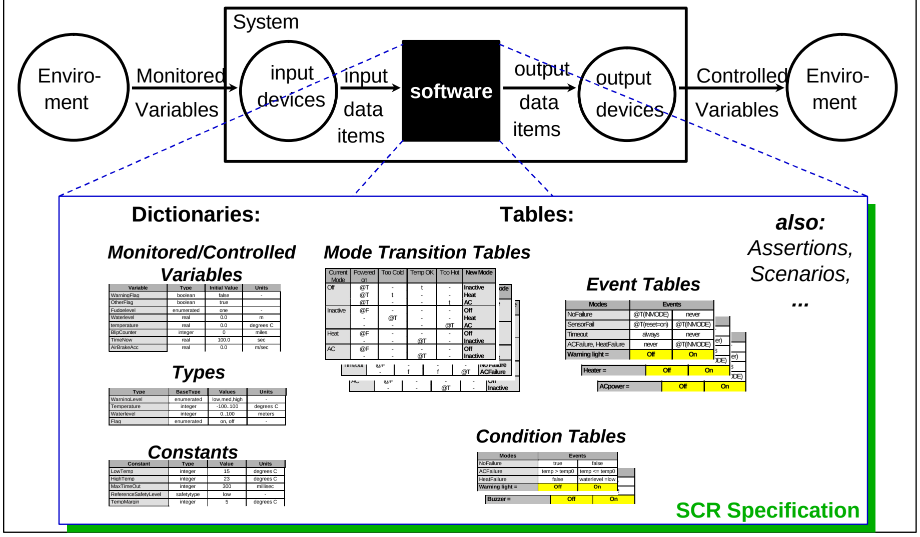
- **Focus on states or events?**
  - E.g. SCR table-based models
  - Explicit event semantics
- **Comparing notations for state transition models**
  - FSMs vs. Statecharts vs. SCR
- **Checking properties of state transition models**
  - Consistency Checking
  - Model Checking, using Temporal Logic
-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

What are we modelling?

- **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

Tabular Specifications: SCR

Four Variable Model:



Monitored/Controlled Variables

Variable	Type	Initial Value	Units
WarningFlag	boolean	false	-
OtherFlag	boolean	true	
Fudgelevel	enumerated	one	-
Waterlevel	real	0.0	m
temperature	real	0.0	degrees C
BlipCounter	integer	0	miles
TimeNow	real	100.0	sec
AirBrakeAcc	real	0.0	m/sec

Types

Type	BaseType	Values	Units
WarningLevel	enumerated	low,med,high	-
Temperature	integer	-100..100	degrees C
Waterlevel	integer	0..100	meters
Flag	enumerated	on, off	-

Constants

Constant	Type	Value	Units
LowTemp	integer	15	degrees C
HighTemp	integer	23	degrees C
MaxTimeOut	integer	300	millisec
ReferenceSafetyLevel	safetytype	low	-
TempMargin	integer	5	degrees C

Mode Transition Tables

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**

Event Tables

Modes	Events	
NoFailure	@T(INMODE)	never
SensorFail	@T(reset=on)	@T(INMODE)
Timeout	always	never
ACFailure, HeatFailure	never	@T(INMODE)
Warning light =	**Off**	**On**

Heater =	**Off**	**On**

ACpower =	**Off**	**On**

Condition Tables

Modes	Events	
NoFailure	true	false
ACFailure	temp > temp0	temp <= temp0
HeatFailure	false	waterlevel =low
Warning light =	**Off**	**On**

Buzzer =	**Off**	**On**

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) ° Øc Ù c’ e.g. @T(y=1) ° y¹1 Ù y’=1**
    - **@F(c) ° c Ù Øc**
  - **A conditioned event is an event with a predicate**
    - **@T(c) WHEN d ° Øc Ù c’ Ù d**

Source: *Adapted from Heitmeyer et. al. 1996.*

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	-	**Inactive**
	@T	t	-	-	**Heat**
	@T	-	-	t	**AC**
Inactive	@F	-	-	-	**Off**
	-	@T	-	-	**Heat**
	-	-	-	@T	**AC**
Heat	@F	-	-	-	**Off**
	-	-	@T	-	**Inactive**
AC	@F	-	-	-	**Off**
	-	-	@T	-	**Inactive**

Source: *Adapted from Heitmeyer et. al. 1996.*

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:

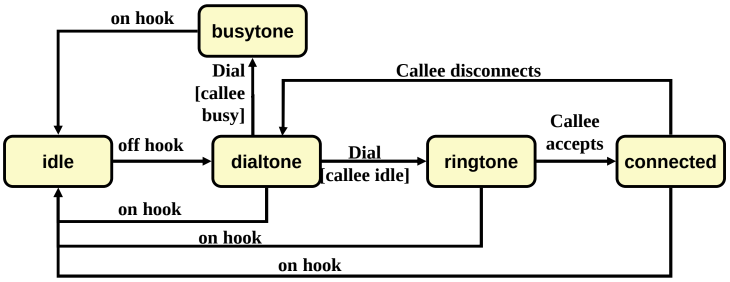
Modes		
Heat, AC	@C(target)	never
Inactive, Off	never	@C(target)
Ack_tone =	**Beep**	**Clang**

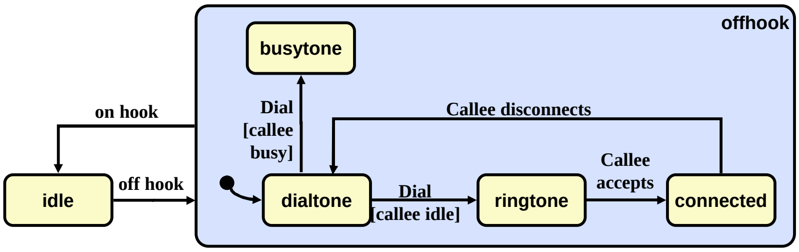
- **Condition Tables**
  - defines the value of a controlled variable under every possible condition
  - Defines a *total function* from modes and conditions to variable values
  - Example:

Modes		
Heat	target - temp ² 5	target - temp >5
AC	temp - target ² 5	temp - target >5
Inactive, Off	true	never
Warning light =	**Off**	**On**

Source: *Adapted from Heitmeyer et. al. 1996.*

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 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

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 |= 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

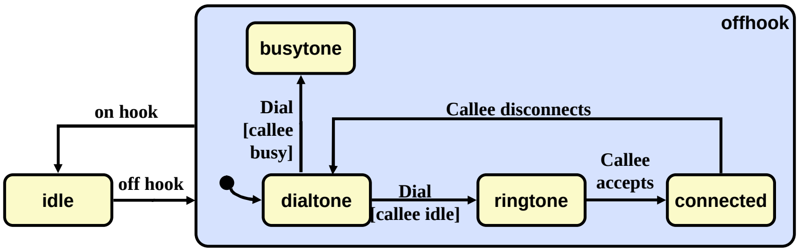
Temporal Logic

- LTL (Linear Temporal Logic)
  - Expresses properties of infinite traces through a state machine model
  - adds two temporal operators to propositional logic:

    ?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 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 ® 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 Ú BUSYTONE) ® A[¬RINGING U 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 to 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)

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