What are Formal Methods?

- **Broad View (Leveson)**
  - Application of discrete mathematics to software engineering
  - Involves modeling and analysis
  - With an underlying mathematically-precise notation

- **Narrow View (Wing)**
  - Use of a formal language
    - A set of strings over some well-defined alphabet, with rules for distinguishing which strings belong to the language
  - Formal reasoning about formulae in the language
    - E.g., formal proofs: use axioms and proof rules to demonstrate that some formula is in the language

- **For requirements modeling**
  - A notation is formal if:
    - It comes with a formal set of rules which define its syntax and semantics.
    - The rules can be used to analyse expressions to determine if they are syntactically well-formed or to prove properties about them.

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
- People get confused about which tools are appropriate:
  - E.g., modeling program behavior vs. modeling the requirements
  - Formal methods advocates get too attached to one tool
- Formal methods require more effort
  - ...and the payoff is deferred

Why formalize in RE

- To remove ambiguity and improve precision
- Provides a basis for verification that the requirements have been met
- Allows us to reason about the requirements
  - Properties of formal requirements models can be checked automatically
  - Can test for consistency, explore the consequences, etc.
  - Allows us to 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

Varieties of formal analysis

- **Consistency analysis and typechecking**
  - “Is the formal model well-formed?”
    - [Assuming that we only use modeling languages 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?”
**Three different models??**

- **R**: a model of the environment
  - acts upon
  - satisfies

- **D**: a model of the requirements

- **S**: a model of the software behaviour

---

**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

<table>
<thead>
<tr>
<th>Issue Severity</th>
<th>With FM</th>
<th>Existing</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Major</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Low Major</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>High Minor</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>Low Minor</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

  **Totals**: 30 vs 4

---

**How do FMs differ?**

- **Mathematical Foundation**
  - Logic
    - first order predicate logic - e.g. RML
    - temporal logic - e.g. Albert II, SCR, KAOS
    - multi-valued logic - e.g. Xchek
  - Other
    - algebraic languages - e.g. Larch
    - set theory - e.g. Z

- **Ontology**
  - fixed
    - states, events, actions - e.g. SCR
    - entities, activities, assertions - e.g. RML
  - extensible
    - meta language for defining new concepts - e.g. Telos

- **Treatment of Time**
  - State/event models
    - time as a discrete sequence of events - e.g. SCR
    - time as quantified intervals - e.g. KAOS
  - Time as a first class object
    - meta-level class to represent time - e.g. Telos

---

**Three traditions...**

- **Formal Specification Languages**
  - Grew out of work on program verification
  - Spawned many general purpose specification languages
  - Suitable for specifying the behaviour of program units
  - Key technologies: Type checking, Theorem proving

- **Reactive System Modeling**
  - Grew out of a need to capture dynamic models of system behaviour
  - Focus is on reactive systems (e.g. real-time, embedded control systems)
  - Support reasoning about safety, liveness, performance(?)
  - Provide a precise requirements specification language
  - Key technologies: Consistency checking, Model checking

- **Formal Conceptual Modeling**
  - Grew out of a concern for capturing real-world knowledge in RE
  - Focus is on modeling domain entities, activities, agents, assertions
  - Use a formal ontology for domain modeling
  - Use first order predicate logic as the underlying formalism
  - Key technologies: inference engines, default reasoning, KBS-shells
(1) Formal Specification Languages

- **Operational** - specification is executable abstraction of the implementation
  - good for rapid prototyping
  - e.g., Lisp, Prolog, Smalltalk
- **State-based** - views a program as a (large) data structures whose state can be altered by procedure calls...
  - e.g., VDM, Z
- **Algebraic** - views a program as a set of abstract data structures with a set of operations...
  - e.g., Larch, CLEAR, OBJ

Developed for specifying programs

- Programs are formal, man-made objects
  - ... and can be modeled precisely in terms of input-output behaviour
- But in RE we’re more concerned with:
  - real-world concepts, stakeholders, goals, loosely define problems, environments
- So these languages are NOT appropriate for RE

(2) Reactive System Modeling

- **General approach**:
  - model the environment as a state machine
  - Model the system as a state machine
  - Model safety, liveness properties of the machine as temporal logic assertions
  - Check whether the properties hold of the system interacting with its environment

Examples:

- **Statecharts**
  - Harel’s notation for modeling large systems
  - Adds parallelism, decomposition and conditional transitions to STDs
- **RSML**
  - Heimdahl & Leveson’s Requirements State Machine Language
  - Adds tabular specification of complex conditions to Statecharts
- **A7e approach**
  - Major project led by Parnas to formalize A7e aircraft requirements spec
  - Uses tables to specify transition relations & outputs
- **SCR**
  - Heitmeyer et. al. “Software Cost Reduction”
  - Extends the A7e approach to include dictionaries & support tables

(3) Formal Conceptual Modeling

- **General approach**
  - model the world beyond functional specifications
    - a specification is prescriptive, concentrating on desired properties of the machine
    - but we also need to capture an understanding of the application domain
    - hence build models of humans’ knowledge/beliefs about the world
  - make use of abstraction & refinement as structuring primitives

Examples:

- **RML** - Requirements Modeling Language
  - Developed by Greenspan & Mylopoulos in mid-1980s
  - First major attempt to use knowledge representation techniques in RE
  - Essentially an object oriented language, with classes for activities, entities and assertions
  - Uses First Order Predicate Language as an underlying reasoning engine
- **Telos**
  - Extends RML by creating a fully extensible ontology
    - meta-level classes define the ontology (the basic set is built in)
- **Albert II**
  - developed by Dubois & du Bois in the mid-1990s
  - Models a set of interacting agents that perform actions that change their state
  - uses an object-oriented real-time temporal logic for reasoning
Adapted from Heitmeyer et. al. 1996.

Source: © 2000-2003, Steve Easterbrook

Event Tables
- An event table defines how a term or controlled variable changes in response to input events.
- Defines a partial function from modes and events to variables

Condition Tables
- A condition table defines the value of a term or controlled variable under every possible condition.
- Defines a total function from modes and conditions to variable values

SCR Tables

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 are described using a number of mode classes operating in parallel.

System State
- A (system) state is defined as:
  - 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.
  - A conditioned event is an event with a predicate.
  - \( \beta T(c) \) WHEN \( d \) means "\( c \) became true when \( c \) was false and \( d \) was true."

SCR Specification

Example: SCR

Four Variable Model:
- Monitored/Controlled Variables
- Mode Transition Tables
- Dictionaries:
  - Monitored/Controlled Variables
  - Mode Transition Tables
  - Conditions:
    - Type
    - Constants
    - Values
- Environments:
  - System
  - Events
  - Variables
  - Controllers
- Also:
  - Assertions, Scenarios, ...

Example: Statecharts

Diagrams:
- SCR Table
- Event Tables
- Condition Tables
- SCR Specification
Example: Temp Control System

Mode transition table:

<table>
<thead>
<tr>
<th>Current Mode</th>
<th>Powered on</th>
<th>Too Cold</th>
<th>Temp OK</th>
<th>Too Hot</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>@T</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>Inactive</td>
</tr>
<tr>
<td>~</td>
<td>@T</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>Heat</td>
</tr>
<tr>
<td>~</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>AC</td>
</tr>
<tr>
<td>Inactive</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td>~</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Heat</td>
</tr>
<tr>
<td>~</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>AC</td>
</tr>
<tr>
<td>Heat</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td>~</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>Inactive</td>
</tr>
<tr>
<td>AC</td>
<td>@F</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Off</td>
</tr>
<tr>
<td>~</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>@T</td>
<td>Inactive</td>
</tr>
</tbody>
</table>

Failure modes

Mode transition table:

<table>
<thead>
<tr>
<th>Current Mode</th>
<th>Powered on</th>
<th>Cold Heater</th>
<th>Too Cold</th>
<th>Warm AC</th>
<th>Too Hot</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoFailure</td>
<td>t</td>
<td>@T</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>HeatFailure</td>
</tr>
<tr>
<td>HeatFailure</td>
<td>t</td>
<td>@F</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>NoFailure</td>
</tr>
<tr>
<td>ACFailure</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>@F</td>
<td>t</td>
<td>NoFailure</td>
</tr>
</tbody>
</table>

Event table:

<table>
<thead>
<tr>
<th>Modes</th>
<th>Modes</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoFailure</td>
<td>@T(INMODE)</td>
<td>Off</td>
</tr>
<tr>
<td>ACFailure, HeatFailure</td>
<td>never</td>
<td>@T(INMODE)</td>
</tr>
<tr>
<td>Warning light =</td>
<td>Off</td>
<td>On</td>
</tr>
</tbody>
</table>

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?

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
- Lightweight Formal Methods
  - Have become popular as a means of getting the technology transferred
  - Two approaches
    - Lightweight use of FMs - selectively apply FMs for partial modeling
    - Lightweight FMs - new methods that allow unevaluated predicates