An Introduction to Formal Modeling in Requirements Engineering

Prof Steve Easterbrook
Dept of Computer Science, University of Toronto, Canada
sme@cs.toronto.edu
http://www.cs.toronto.edu/~sme

Outline

Why do we need Formal Methods in RE? you are here!

What do formal methods have to offer?
A survey of existing techniques
Example modeling language: SCR
  ¦ The language
  ¦ Case study
  ¦ Advantages and disadvantages
Example analysis technique: Model Checking
  ¦ How it works
  ¦ Case study
  ¦ Advantages and disadvantages
Where next?

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)
  ¦ 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.

Example formal system: Propositional Logic

First Order Propositional Logic provides:
  ¦ a set of primitives for building expressions:
  ¦ variables, numeric constants, brackets
  ¦ a set of logical connectives:
  ¦ and (\&), or (\|), not (\neg), implies (\rightarrow), logical equality (=)
  ¦ the quantifiers:
  ¦ "for all" (\forall)
  ¦ "there exists" (\exists)
  ¦ a set of deduction rules

Expressions in FOPL
  ¦ expressions can be true or false
  ¦ x>y \& y>z
  ¦ x=y
  ¦ x\rightarrow y
  ¦ x=y\rightarrow y=x
  ¦ x=y\equiv y=x
  ¦ x>3
  ¦ x<-6

Open vs. Closed Expressions
  ¦ a variable that is quantified is bound (otherwise it is free)
  ¦ if all the variables are bound, the formula is closed
  ¦ a closed formula is either true or false

What does correctness mean?

Some distinctions:
  ¦ Domain Properties are things in the application domain that are true whether or not we ever build the proposed system
  ¦ Requirements are things in the application domain that we wish to be made true by delivering the proposed system
  ¦ A specification is a description of the behaviour the program must have in order to meet the requirements

Two correctness (verification) criteria:
  ¦ The Program running on a particular Computer satisfies the Specification
  ¦ The Specification, in the context of the given domain properties, satisfies the requirements

Two completeness (validation) criteria:
  ¦ We discovered all the important requirements
  ¦ We discovered all the relevant domain properties
Understanding the distinctions...

Requirement R:
- Reverse thrust shall only be enabled when the aircraft is moving on the runway.

Domain Properties D:
- Wheel pulses on if and only if wheels turning
- Wheels turning if and only if moving on runway

Specification S:
- Reverse thrust enabled if and only if wheel pulses on

S + D imply R
- But what if the domain assumptions are wrong?

Where do things go wrong?

Application Domain
- Specification is wrong (common)
  Causes:
  - Misunderstood requirements
  - Ambiguity, inconsistent or incomplete specification
  Caught by:
  - Inspection, formal verification, end-to-end testing

Machine Domain
- Requirements are wrong (common)
  Causes:
  - Insufficient communication with customers/users
  - Lack of analysis
  - Failure to handle change
  Caught by:
  - Inspections, customer reviews, modeling, formal validation, prototyping

Another Example

Requirement R:
- The database shall only be accessible by authorized personnel

Domain Properties D:
- Authorized personnel have passwords
- Passwords are never shared with non-authorized personnel

Specification S:
- Access to the database shall only be granted after the user types an authorized password

S + D imply R
- But what if the domain assumptions are wrong?

Where do things go wrong?

Application Domain
- Program is wrong (rare?)
  Causes:
  - Latent bugs (programmer error)
  - Misunderstood specification
  - Poor configuration management
  - Poor change control
  Caught by:
  - Test to specification; Inspection and Walkthroughs

Machine Domain
- Specification is wrong (common)
  Causes:
  - Misunderstood requirements
  - Ambiguous, inconsistent or incomplete specification
  Caught by:
  - Inspections, formal verification, end-to-end testing

Another Example

Requirement R:
- The database shall only be accessible by authorized personnel

Domain Properties D:
- Authorized personnel have passwords
- Passwords are never shared with non-authorized personnel

Specification S:
- Access to the database shall only be granted after the user types an authorized password

S + D imply R
- But what if the domain assumptions are wrong?
Where do things go wrong?

Domain Properties are wrong (very common)

Causes:
- Lack of domain expertise
- Unquestioned assumptions
- Insufficient domain analysis

Caught by:
- Failure analysis, talking to the right experts, off-normal testing

Where do formal methods apply?

Formalize S - the specification
- as a precise baseline to verify the program against
  - specification languages: Z, VDM, Larch,
  - as an explicit model of program behavior to compare against requirements

Formalize D - the domain knowledge
- so we can reason about:
  - whether it is complete
  - how it affects the proposed system
  - Formalization helps us to be precise and explicit about the environment

Formalize R - the requirements
- so we can animate them
- so we can test logical coherence
- so we can check for completeness
  - against an underlying mathematical (semantic) model

Why formalize in RE?

- To remove ambiguity and improve precision
- Provides a basis for verification:
  - That a program meets its specification
  - That a that specification captures the requirements adequately

- 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

Why people don't formalize in RE

FM tend to be lower level than other techniques
- they force you to include too much detail
- you have to have decided the system boundaries already

FM tend to concentrate on consistent, correct models
- but most of the time your models are inconsistent, incorrect, incomplete...

Confusion about which tools are appropriate:
- Are we modeling program behavior or modeling the requirements?
- Formal methods advocate get too attached to one tool
- false expectations about scalability of research prototypes

FM require more effort
- "they slow you down"
- "they require lots of mathematical training"
- ...and the payoff is deferred

FM are not appropriate in many projects...

Why do we need Formal Methods in RE?

Outline

- What do formal methods have to offer? you are here!
  - A survey of existing techniques
    - The language
    - Case study
    - Advantages and disadvantages
  - Example analysis technique: Model Checking
    - How it works
    - Case Study
    - Advantages and disadvantages
  - Where next?

Setting the Boundaries

S - Specification of s/w in terms of inputs & outputs

D - Domain Properties that constrain how the environment can behave

Environment

- Monitored Variables
- Controlled Variables

System

Input devices

Software

Output devices

Output

Software

Input devices

Environment

Transformations

Transformations

- Requirements: what control actions the system must take in which circumstances.
Three different models??

D: a model of the environment

S: a model of the software behaviour

R: a model of the requirements

is satisfied by

Modeling...

Modeling can guide elicitation:
- Does the modeling process help you figure out what questions to ask?
- Does the modeling process help to surface hidden requirements?
  - i.e. does it help you ask the right question?

Modeling provide a measure of progress:
- Does completeness of the model imply completeness of the elicitation?
  - i.e. if we’ve filled in all the pieces of the model, are we done?

Modeling can help to uncover problems:
- Does inconsistency in the model reveal interesting things?
  - e.g. inconsistency could correspond to conflicting or infeasible requirements
  - e.g. inconsistency could mean confusion over terminology, scope, etc
  - e.g. inconsistency could reveal disagreements between stakeholders

Modeling can help us check our understanding:
- Can we test that the model has the properties we expect?
- Can we reason over the model to understand its consequences?
- Can we animate the model to help us visualize/validate the requirements?

Type of Model

Modeling is necessary. But we can choose how:

natural language
- extremely expressive and flexible
- very poor at capturing the semantics of the model
- good for elicitation, and to annotate models for communication

semi-formal notation
- captures structure and some semantics
- can perform some reasoning, consistency checking, animation, etc.
  - e.g. diagrams, tables, structured English, etc.

formal notation
- very precise semantics, extensive reasoning possible
- long way removed from the application domain
  - note: requirements formalisms are geared towards cognitive considerations, hence differ from most computer science formalisms

Desiderata for Modeling Languages

Implementation Independence
- does not model data representation, internal organisation, etc.

Abstraction
- extracts essential aspects
  - e.g. things not subject to frequent change
- supports “big picture” views

Formality
- unambiguous syntax
- rich semantic theory

Constructability
- can compose pieces of the model to handle complexity and size
- construction should facilitate communication

Ease of analysis
- ability to analyse for ambiguity, incompleteness, inconsistency, etc.

Traceability
- ability to cross-reference elements
- ability to link to design, implementation, etc.

Executability
- can animate the model, to compare it to reality

Minimality
- No redundancy of concepts in the modeling scheme
  - e.g. no arbitrary choices of how to represent something

Meta-Modeling

Can compare modeling languages using meta-models:
- What phenomena does each language capture?
- What guidance is there for how to elaborate the models?
- What analysis can be performed on the models?

Example meta-model:

Actions inducing change of facts in the application domain

State changes in the application domain

(excerpt from) KAOS meta-model
Validating our models

**logical positivist view:**
- There is an objective world that can be modeled by building a consistent body of knowledge grounded in empirical observation.
- In RE: "there is an objective problem that exists in the world"
- Build a consistent model; make sufficient empirical observations to check validity
- Use tools that test consistency and completeness of the model
- Use reviewers, prototyping, etc. to demonstrate the model is "valid."

**Popper’s modification to logical positivism:**
- "In RE: "requirements models must not be refutable"
- Typically incorrect requirements (system does the wrong thing in response to an input)
- Use stakeholder involvement to see that they own the requirements models
- Use ethnographic techniques to understand the worldviews

**post-modern view:**
- "There is no privileged viewpoint; all observation is value-laden; scientific investigation is culturally embedded"
- e.g. collect scenarios and check the model supports them

- Other logic: extensible
- First order predicate logic: e.g. RML
- Meta language for defining new concepts: e.g. Telos
- Meta-level class to represent time - e.g. Telos
- States, events, actions - e.g. SCR, RSML, Statecharts, ...

### Formal Validation

- **Consistency analysis and typechecking**
  - "Is the formal model well-formed?"
  - Usually, well-formedness is needed before other validation can be done...
  - Well-formedness may also correspond to a useful real-world integrity property

- **Validation:**
  - Animation of the model in 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 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?"

### 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
  - Formal analysis tends finer, but more subtle problems
  - Typical errors found include:
    - Inconsistent interfaces
    - Incorrect requirements (system does the wrong thing in response to an input)
    - Clarity/reusability problems

<table>
<thead>
<tr>
<th>Issue Severity</th>
<th>With FM Existing</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Major</td>
<td>2 0</td>
</tr>
<tr>
<td>Low Major</td>
<td>5 1</td>
</tr>
<tr>
<td>High Minor</td>
<td>17 3</td>
</tr>
<tr>
<td>Low Minor</td>
<td>6 0</td>
</tr>
<tr>
<td>Totals</td>
<td>30 4</td>
</tr>
</tbody>
</table>

### How do FMs differ?

**Ontology**
- States, events, actions - e.g. SCR, RSML, Statecharts, ...
- Entities, activities, assertions - e.g. RML
- Extensible
- Meta language for defining new concepts - e.g. Telos

**Mathematical Foundation**
- Logic
  - First order predicate logic - e.g. RML
- Temporal prepositional logic - e.g. Albert II, SCR, KAOS

**Other**
- Algebraic languages - e.g. Larch
- Set theory - e.g. Z

**Treatment of Time**
- Discrete-event models
  - Time as a discrete sequence of events - e.g. SCR
  - Time as an abstracted interval - e.g. KAOS
- Continuous-time models
  - Meta-level class to represent time - e.g. Telos

### Outline

- Why do we need Formal Methods in RE?
  - What do formal methods have to offer?
    - A survey of existing techniques
    - Example modeling language: SCR
    - Example analysis technique: Model Checking

- Three traditions...
(1) Formal Specification Languages

Three basic flavours:

- Operational - specification is executable abstraction of the implementation
  - good for rapid prototyping
  - e.g., Lisp, Prolog, SealTalk
- 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...
  - operations are defined declaratively by giving a set of axioms
  - 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 very appropriate for RE
- But people fail to realize that requirements specification is a program specification

Examples:

- RML - Requirements Modeling Language
  - Developed by Greenspan & Mylopoulos in mid-1980s
  - First major attempt to use knowledge representation techniques in RE
  - Essentially object-oriented, with classes for activities, entities and assertions
  - Uses First Order Predicate Language as an underlying reasoning engine
- SCR
  - Heitmeyer et. al. “Software Cost Reduction”
  - Extends the A7e approach to include dictionaries & support tables

(2) Reactive System Modeling

Modeling how a system should behave

- 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
  - Hans’s notation for modeling large systems
  - Adds parallelism, decomposition and conditional transitions to STDs
  - RSML
  - Heitmeyer & Larson’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 understand the application domain
  - hence build models of human knowledge/beliefs about the world
- make use of abstraction & refinement as structuring primitives

Examples:

- RAL - Requirements Modeling Language
  - Developed by Greenspan & Mylopoulos in mid-1980s
  - First major attempt to use knowledge representation techniques in RE
  - Essentially object-oriented, with classes for activities, entities and assertions
  - Uses First Order Predicate Language as an underlying reasoning engine
- Talks
  - 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 & de Bata 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

From notations to methods...

- A notation:
  - a representation scheme (or language) for expressing things;
    - e.g., Z, first order logic, state transition diagrams, UML.
- A technique:
  - prescribes how to perform a particular activity - and, if necessary, how to describe a product of that activity in a particular notation;
    - e.g., SCR, VDM.
- A method:
  - a technical prescription for how to perform a collection of activities, focusing on integration of techniques and guidance about their use;
    - e.g., SADT, DMT, ZSPD, KADS.
- A Process model:
  - an abstract description of how to conduct a collection of activities, focusing on resource usage and dependencies between activities.

State Transition Diagram

Statecharts

[Diagram showing state transitions and statechart notations]
**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></td>
<td>t</td>
<td></td>
<td>Heat</td>
</tr>
<tr>
<td></td>
<td>@T</td>
<td></td>
<td></td>
<td></td>
<td>AC</td>
</tr>
<tr>
<td>Inactive</td>
<td>@F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@T</td>
<td></td>
<td>@T</td>
<td></td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inactive</td>
</tr>
<tr>
<td>Heat</td>
<td>@F</td>
<td></td>
<td></td>
<td></td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>@T</td>
<td></td>
<td>@T</td>
<td></td>
<td>Inactive</td>
</tr>
<tr>
<td>AC</td>
<td>@F</td>
<td></td>
<td></td>
<td>@T</td>
<td>Inactive</td>
</tr>
</tbody>
</table>

**Failure modes:**

<table>
<thead>
<tr>
<th>Current Mode</th>
<th>Powered on</th>
<th>Cold</th>
<th>Too Cold</th>
<th>Warm</th>
<th>Too Hot</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Event table:**

<table>
<thead>
<tr>
<th>Modes</th>
<th>@T(INMODE)</th>
<th>never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never Failure</td>
<td>@T(INMODE)</td>
<td>never</td>
</tr>
<tr>
<td>Heat Failure</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>AC Failure</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>

**Definitions:**
- An event occurs when any system entity changes value.
- A condition table defines the value of a term or controlled variable under every possible condition.
- A complex system will have many different modes classes.
- A mode class is a finite state machine, with states called system modes.
- A conditioned event is an event with a predicate.
### Verification approach

**Modeling Approach**
- Two stages:
  - model the original requirements;
  - modify as per change request
- Use the SCRtool consistency checker to 'debug' the model
- SCR model status:
  - 13 functions
  - 265 constants
  - 165 variables
- SCR toolset restrictions:
  - Missing initial values
  - Needed deep understanding of SCR semantics

**Verification Properties**
- Most properties were invariants:
  - Checked for completeness; dependency cycles; functional decomposition
  - Needed deep understanding of SCR semantics
- Did not catch any of the errors in the Change Request!

### Results & Recommendations

**Results**
- Systematic ambiguity in natural language/pseudo-code
- Missing initial values
- 1 modified constant

**Comments**
- Difficult abstraction step
  - from imperative to state machine model
- Restrictions from the semantic model:
  - Needed deep understanding of SCR semantics
- SCR was not designed for analysis of change in legacy systems

**Caveats**
- SCR was not designed for analysis of change in legacy systems
- Some variables cannot be verified (need simulator):
  - A large peak immediately after MM603 may indicate an energy dump pull-up maneuver. There should be no difference between FSW and environment values.
- E.g. “The commanded roll angle will be zero during the alpha recovery and NZ hold phases.”
- Can do this on 3EO
- SCR was not designed for analysis of change in legacy systems!
# Conclusions

**Feasibility study:**
- Are automated verification techniques mature enough?
  - FM in general, SCR in particular
  - large system, informal structured requirements
  - very safety-critical
  - analysis of change

**Good news:**
- SCR provides all the right modelling primitives
- Tool scalability does not appear to be a major issue

**Bad news:**
- Couldn't model existing structure
- Couldn't isolate change

---

# Example Analysis Technique: Model Checking

**How it works:**
- Explode counter-examples
- Run the model checker:
  - Build a finite state machine model
- Explore counter-examples:
  - Find out why the property doesn’t hold
- Counter-example is a trace through the model

**Computes the value of:**
- model |= property

**Types of propositions:**
- Propositions in first order logic (for invariants)
- 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:
  - \( \text{AG} p \): always (now and in the future)
  - \( \text{AU} p \): eventually (in some future state)
  - \( \text{AX} p \): always (now and in the future)
  - \( \text{AU} p \): eventually (in some future state)

**CTL (Computational Tree Logic):**
- Branching-time logic - can quantify over possible futures
  - Each operator has two parts:
    - \( \text{EF} p \): eventually (in some future state)
    - \( \text{AF} p \): always (now and in the future)
    - \( \text{EG} p \): eventually (in some future state)
    - \( \text{AG} p \): always (now and in the future)

---

# Example

**Sample Properties**

If you are connected you can hang up:
- \( \text{AD} \text{CONNECTED} \rightarrow \text{EOFFHOOK} \)

If you are connected, hanging up always disconnects you:
- \( \text{AD} \text{CONNECTED} \rightarrow \text{AOFFHOOK} \)

A connection doesn’t start until you pick up the phone:
- \( \text{AD} \text{CONNECTED} \rightarrow \text{ACONNECTED} \)

If you make a call, the phone cannot ring without returning to idle first:
- \( \text{AD(} \text{RINGTONE} \rightarrow \text{BUSYTONENOTRINGED} \) \)
### Complexity Issues

The problem:
- Model checking is exponential in the size of the model and the property
- Current MC engines can explore $10^{100}$ 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)

---

### Case Study I - Background

- Dual Redundant Spacecraft Controller
  - Two identical hardware/software platforms
  - Prime string controls the spacecraft
  - Online string acts as warm backup during critical sequences
  - Prime string generates a 32-word State Table Broadcast (STB) once per second
- Markpoint & Rollback scheme
  - Markpoints in the critical sequence represent completed subsequences
  - When a fault occurs:
    - Critical sequence is suspended
    - Fault recovery is invoked
    - If the fault is repaired, both strings resume from the last markpoint
    - 3 seconds delay to ‘age’ each markpoint, to allow mechanical operations to complete.

---

### Specified Behaviour (statecharts)

#### PRIME NOMINAL

- **Initial state:** Prime Init
- **Transitions:**
  - On Start: Prime Init $\rightarrow$ Power-Up Idle
  - On SFP Response: Resume Critical Sequence from aged markpoint

#### ONLINE

- **Initial state:** Online Init
- **Transitions:**
  - On Fault: Fault Idle $\rightarrow$ Resume Critical Sequence from aged markpoint

---

### State Space Size Estimation

#### Initial model:
- Assume each state in statechart needs 4 substates in the model
- Prime string: 16 major states, hence $4^{16}$
- Online string: 14 major states, hence $4^{14}$
- STB: 27 elements, each at least binary, hence $2^{27}$
- Total: $2^{37}$ states

#### Model reduction strategies:
- Projection - identified 5 fault classes, treated them separately:
  - Symmetry - doesn’t matter which processor runs which string
- Abstraction - removed all data from the STB other than that needed for rollback
- Abstraction - simplified possible input data: minimal length critical sequence
- Projection - verify each LTL property separately

- Initial model estimated at 196,608 states + never clause; actual was just over 100,000

---

### Verification against requirements

#### Requirements:
1. If a fault occurs when the last markpoint was at the start of the sequence, the prime string shall roll back to the start, regardless of how much time has expired since the program started running.
2. If a fault occurs when the time, $t$, since the last markpoint was less than 3 seconds, and the last markpoint was not at the start of the sequence, the prime string shall roll back to the previous markpoint.
3. If a fault occurs when the time $t$ following the last markpoint was greater than or equal to 3 seconds, the prime string shall roll back to the last markpoint.

#### In Linear Temporal Logic
- If $p$ is the occurrence of a fault, and $q$ is the correct response, then:
  \[ p \implies (\exists i < n)(\exists j < m)(p(i) \land q(j)) \]
- Six such properties (3 for prime rollback, 3 for online rollback)

---

### Results: 3 anomalies

- **Failure to Suspend**
  - Occurs when prime string detects and corrects a fault between two consecutive rendezvous broadcasts (i.e. within 1 second)
  - Online string never rolls back, hence gets out of synch.
  - Repeated occurrence may cause loss of redundancy
- **No rollback at end of critical sequence**
  - End of critical sequence is not designated a markpoint
  - When online string reaches the end of the sequence, it returns to ‘Power Up Idle’
  - Hence, if a fault occurs within 3 seconds of the end of the sequence, and the prime rolls back, there is no longer a backup
- **Fault occurs at $t+2$**
  - Prime string suspends its aging at $t+2$ seconds;
  - Online string does not receive the updated STB until the next second
  - Result is that online rolls back to the last markpoint, prime rolls back to the previous one
Outline

Why do we need Formal Methods in RE?
What do formal methods have to offer?
A survey of existing techniques
Example modeling language: SCR
\[ \text{The language} \]
\[ \text{Case study} \]
\[ \text{Advantages and disadvantages} \]

Example analysis technique: Model Checking
\[ \text{How it works} \]
\[ \text{Case Study} \]
\[ \text{Advantages and disadvantages} \]
Where next?

Lessons Learned

- Formal verification is effective for critical components
  - Initial models developed fairly quickly (e.g. one week)
  - Effort cost is similar to manual inspection
  - Always reveals ambiguities and minor documentation errors
  - Often reveals errors that cannot be detected through inspection and testing
  - But there are no studies of cost effectiveness vs. criticality tradeoff

- Emphasis on finding errors is appropriate
  - All existing methods used by verification/test teams have this focus
  - Hence, just another tool in the V&V toolbox

- Abstraction, partitioning & projection are important
  - This is the hard part - generating an analyzable model
  - Determine which verification properties you are interested in first:
    - Use trace to guide the modeling process

Technology Transfer Issues

Expertise needed
\[ \text{Knowing when and how to apply formal verification requires much expertise} \]
\[ \text{Building a state machine model is straightforward} \]
\[ \text{Building a checkable state machine model is much harder} \]
\[ \text{(Need training on how modeling ensures affect state space size)} \]
\[ \text{Identifying appropriate properties to verify requires domain expertise} \]
\[ \text{Expressing those properties formally is relatively straightforward} \]

Who performs the analysis?
\[ \text{Programmers don't like to program the same problem twice} \]
\[ \text{Can models be extracted from code??} \]
\[ \text{V&V teams do modeling and defect detection routinely} \]

More Tech Transfer Issues

- Cost effectiveness
  - No models yet for ROI for formal verification of specifications
  - Two expected benefits:
    - Some defects found earlier (before coding); reduces cost of testing
    - Some defects would not be found at all otherwise; increases quality of product

- Evolution of Models
  - Requirements and Design models must be evolvable
  - Evolution is difficult when the models have been carefully optimized for model checking
  - Regression testing is hard because properties are defined in terms of the model

- Link to testing
  - E.g. use traces from the requirements model as test cases
  - E.g. embed the never clause as a runtime monitor

Using Formal Methods

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

Lightweight Formal Methods
\[ \text{Have become popular as a means of getting the technology transferred} \]
\[ \text{Two approaches} \]
\[ \text{Lightweight use of PIs - selectively apply PIs for partial modeling} \]
\[ \text{Lightweight PIs - new methods that allow unverified predicates} \]

Further Reading I

- Introductions to Formal Methods:

- On modeling in RE:

- On formal methods and technology transfer:
Further Reading II

On the formal techniques described in this tutorial:

**SCR:**

**Model checking:**

**Case Studies:**