University of Toronto — Department of Computer Science

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

University of Toronto — Department of Computer Science

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?

University of Toronto — Department of Computer Science

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.

University of Toronto — Department of Computer Science

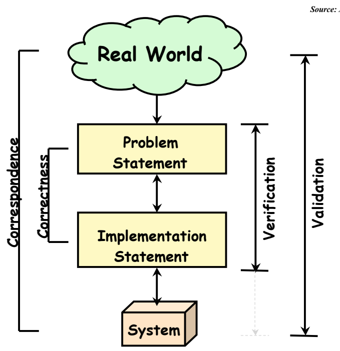
Example formal system: Propositional Logic

- First Order Propositional Logic provides:
  - a set of *primitives* for building expressions:
    - variables, numeric constants, brackets
  - a set of logical *connectives*:
    - and ($\wedge$), or ($\vee$), not ($\neg$), implies ($\Rightarrow$), logical equality (=)
  - the *quantifiers*:
    - $\forall$ - "for all"
    - $\exists$ - "there exists"
  - a set of *deduction rules*
- Expressions in FOPL
  - expressions can be *true* or *false*
    - $(x>y \wedge y>z) \Rightarrow x>z$
    - $x=y \equiv y=x$
    - $\forall x,y,z\ ((x>y \wedge y>z) \Rightarrow x>z)$
    - $x+1 < x-1$
    - $\forall x\ (\exists y\ (y=x+z))$
    - $x>3 \quad x<-6$
- 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

University of Toronto — Department of Computer Science

The Essential Software Process?

Source: Adapted from Blum, 1992, p32



University of Toronto — Department of Computer Science

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 behaviours 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

Source: Adapted from Jackson, 1995, p170-171

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?

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?

- **Computer goes wrong (very rare!)**
- **Causes:**
  - Power failure
  - Circuit or Chip failure
  - Operating System Failure
  - Unforeseen Device or Network failure
- **Caught by:**
  - Design analysis, testing, certification through use, …

Application Domain Machine Domain

Where do things go wrong?

Application Domain Machine 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

Where do things go wrong?

- **Specification is wrong (common)**
- **Causes:**
  - Misunderstood requirements
  - Poor choice of specification language
  - Ambiguous, inconsistent or incomplete specification
- **Caught by:**
  - Inspection, formal verification, end-to-end testing

Application Domain Machine Domain

Where do things go wrong?

Application Domain 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

Where do things go wrong?

- Domain Properties are wrong (very common)
- Causes:
  - Lack of domain expertise
  - Unquestioned assumptions
  - Insufficient domain analysis

"I unquestioningly assume that all assumptions must be questioned" - Parnas

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

Outline

- Why do we need Formal Methods in RE?
- What do formal methods have to offer? *you are here!*
- 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?

Why formalize in RE?

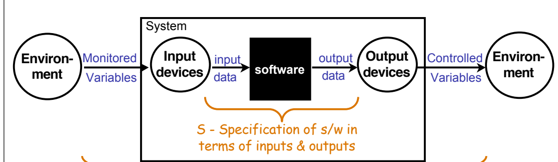
- To remove ambiguity and improve precision
- Provides a basis for verification:
  - That a program meets its specification
  - That a that specification captures the requirements adequately
- Allows us to reason about the requirements
  - Properties of formal requirements models can be checked automatically
  - 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 advocates get too attached to one tool!
  - false expectations about scaleability 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…

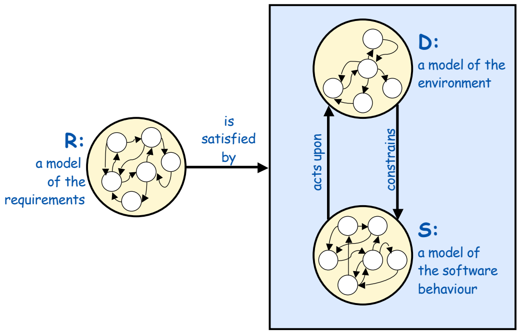
Setting the Boundaries

- Formal methods assume the boundaries are known
- E.g. the four variable model:
  - Fixes the input/output devices
  - Uses I/O data items as proxies for the monitored and controlled variables



R - Requirements: what control actions the system must take in which circumstances.
D - Domain Properties that constrain how the environment can behave

Three different models??



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 questions?
- → **Modeling can 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:

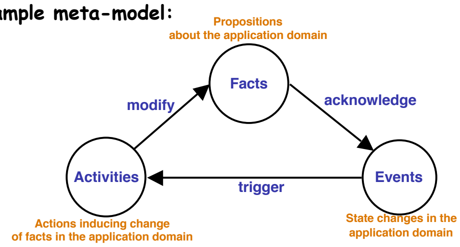
- → **natural language**
  - ✎ extremely expressive and flexible
  - ✎ very poor at capturing the semantics of the model
  - ✎ good for elicitation, and to annotate models for communication
- → **semi-formal notation**
  - ✎ captures structure and some semantics
  - ✎ can perform (some) reasoning, consistency checking, animation, etc.
    - ➢ E.g.s: 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

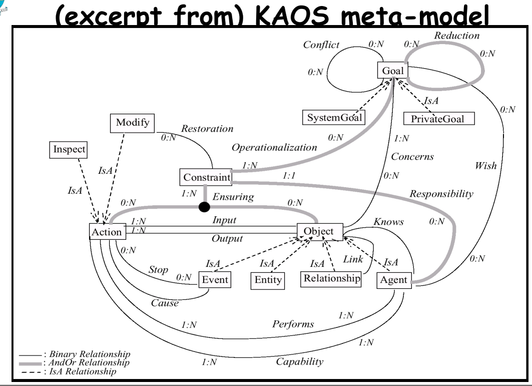
- → **Implementation Independence**
  - ✎ does not model data representation, internal organization, etc.
- → **Abstraction**
  - ✎ extracts essential aspects
    - ➢ e.g. things not subject to frequent change
  - ✎ supports "big picture" views
- → **Formality**
  - ✎ unambiguous syntax
  - ✎ rich semantic theory
- → **Constructability**
  - ✎ can compose pieces of the model to handle complexity and size
  - ✎ construction should facilitate communication
- → **Ease of analysis**
  - ✎ ability to analyze for ambiguity, incompleteness, inconsistency,…
- → **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
    - ➢ i.e. no extraneous 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:**



(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 reviews, prototyping, etc to demonstrate the model is "valid"

→ Popper's modification to logical positivism:
- "theories can't be proven correct, they can only be refuted by finding exceptions"
- In RE: "requirements models must be refutable"
  - Look for evidence that the model is wrong
  - E.g. collect scenarios and check the model supports them

→ post-modern view:
- "there is no privileged viewpoint; all observation is value-laden; scientific investigation is culturally embedded"
- E.g. Kuhnian paradigms; Toulmin's *weltanschauungen*
- In RE: "validation is always subjective and contextualised"
  - Use stakeholder involvement so that they 'own' the requirements models
  - Use ethnographic techniques to understand the weltanschauungen

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 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 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
  - 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

Outline

→ Why do we need Formal Methods in RE?

→ What do formal methods have to offer?

→ A survey of existing techniques *you are here!*

→ 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?

How do FMs differ?

→ Ontology
- fixed
  - 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 propositional logic - e.g. Albert II, SCR, KAOS
- Other
  - algebraic languages - e.g. Larch
  - set theory - e.g. Z

→ 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

Applicability to RE is poor
- No abstraction or structuring
- closely tied to program semantics

Examples: Larch, Z, VDM, …

Reactive System Modeling
- Aim to capture dynamic models of system behaviour
- Focus 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

Applicability to RE is good
- modeling languages were developed specifically for RE

Examples: Statecharts, RSML, Parnas-tables, SCR, …

Formal Conceptual Modeling
- Concerned with capturing real-world knowledge in RE
- Focus on modeling domain entities, activities, agents, assertions
  - provide a formal ontology for domain modeling
  - use first order predicate logic as the underlying formalism
- Key technologies: inference engines, default reasoning, KBS-shells

Applicability to RE is excellent
- modeling schemes capture key requirements concepts

Examples: Reqts Apprentice, RML, Telos, Albert II, …

(1) Formal *Specification* Languages

- **Three basic flavours:**
  - 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…
    - … using pre/post-conditions to specify the effect of procedures
    - 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 realise that requirements specification ≠ program specification

(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
    - 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

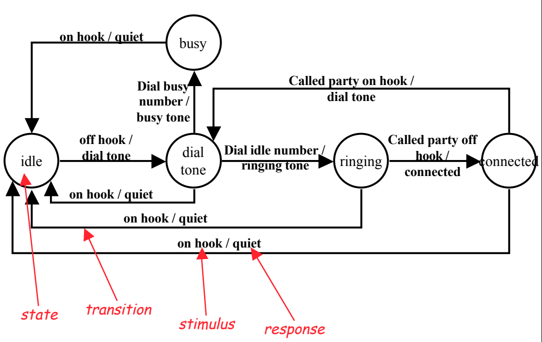
(3) Formal Conceptual *Modeling*

- **General approach**
  - model the world beyond functional specifications
    - a specification is prescriptive, concentrating on desired properties of the machine
    - but we also need to understand 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 object-oriented, 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

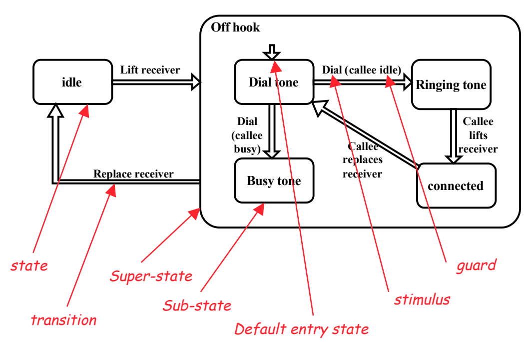
From notations to methods…

- **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, OMT, JSD, KAOS
- **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



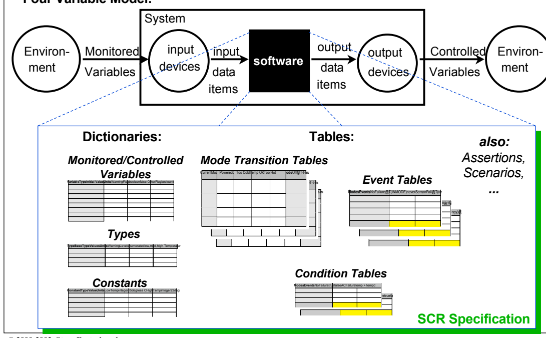
Statecharts



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 *you are here!*
  - The language
  - Case study
  - Advantages and disadvantages
- → Example analysis technique: Model Checking
  - How it works
  - Case Study
  - Advantages and disadvantages
- → Where next?

SCR basics



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
  - Notation: @T(c) means "c changed from false to true"
- A conditioned event is an event with a predicate
  - @T(c) WHEN d means: "c became true when c *was* false and d *was* true"

SCR Tables

→ Mode Class Tables

- Define the 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 conditions that cause transitions between modes
- A mode table defines a *partial function* from modes and events to modes

→ 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 variable values

→ 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

Example: Temp Control System

Mode transition table:

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**

Failure modes

Mode transition table:

Current Mode	Powered on	Cold Heater	Too Cold	Warm AC	Too Hot	**New Mode**
NoFailure	t	@T	t	-	-	**HeatFailure**
	t	-	-	@T	t	**ACFailure**
HeatFailure	t	@F	t	-	-	**NoFailure**
ACFailure	t	-	-	@F	t	**NoFailure**

Event table:

Modes		
NoFailure	@T(INMODE)	never
ACFailure, HeatFailure	never	@T(INMODE)
Warning light =	**Off**	**On**

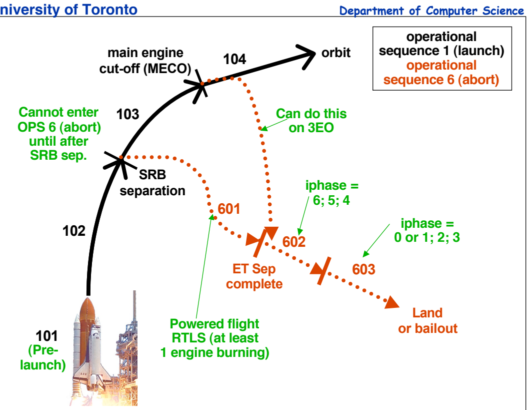
7

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?

Case Study Background

- **Space Shuttle Flight Software**
  - *Operational Increments (OI), approx every 12 - 18 months*
  - *Add new capabilities, correct anomalies, etc*
  - *Change Requests = specification pages + handwritten annotations*
  - *Manual review process*
- **East Coast Abort Landing Change Request**
  - *To automate entry guidance for emergency landing*
  - *To reduce crew training and increase survivability*
  - *Main functions:*
    - *Management of energy during descent*
    - *Guidance to align the shuttle with runway*
  - *104 pages from 7 specifications*
- **Source material**
  - Existing requirements use a functional decomposition:
    - 13 functions
  - Expressed informally:
    - natural language, equations, pseudo-code, tables of variables



If IPHASE=6, the constant alpha recovery angle-of-attack command, ALPCMD, as well as the altitude rate dependent incremental NZ command, DGRNZ, for the load relief phase (IPHASE = 5) are computed as shown in Equation Set 1.

1.1 If CONT=OFF then ALPCMD=ALPREC
*1.2 If CONT=ON then ALPCMD=MIDVAL(ALPRECS * MACH + ALPRECI, ALPRECU, ALPRECL)*
1.3 If HDOT<HDMAX, then HDMAX=HDOT Otherwise (HDOT>=HDMAX) execute Equation Set 1.3 for intact aborts (CONT=OFF) or Equation Set 1.4 for contingency aborts (CONT=ON)
*1.3.1 DGRNZ= MIDVAL((HDNOM – HDMAX) * DHDNZ, DHDLL, DHDUL)*
1.3.2 DGRNZT= GRNZC1 + DGRNZ + 1.0
*1.4.1 DGRNZT= MIDVAL(DNZB – HDMAX * DHDNZ, DNZMIN, DNZMAX)*
*1.4.2 SMNZ1= ZDT1*DGRNZT*
1.4.3 DGRNZ= DGRNZT – GRNZC1 – 1.0
1.4.4 NZSW= GRNZC1 – SMNZ1 – SMNZ2 + 1.0

Otherwise (IPHASE /= 6), the angle-of-attack command for the alpha transition phase is computed. Equation Set 2 tests for the initial pass through the logic and initializes the command.

2.1 If IGRA=0, then ALPCMD=ALPHA and IGRA=1

Next, a test on IGRA is made. If IGRA = 1, Equation Set 3 is executed to compute the smoothed angle-of-attack command and the function is exited.

3.1 DGRALP=MIDVAL(GRALPR–ALPHA, GRALL, GRALU)
3.2 ALPCMD=ALPCMD+DGRALP
3.3 If (DGRALP < 0.0 and ALPCMD <= GRALPR) or (DGRALP > 0.0 and ALPCMD > GRALPR), then ALPCMD = GRALPR and IGRA = 2

Otherwise (IGRA /= 1), Equation Set 4 is used to set the angle-of-attack command equal to the reference angle of attack

4.1 ALPCMD = GRALPR

The function is exited.

Verification approach

- **Modeling Approach**
  - Two steps:
    - model the original requirements;
    - modify as per change request.
  - Use the SCRtool consistency checker to 'debug' the model
  - SCR model stats:
    - 30 types, 265 constants, 165 variables, 2 mode classes
- **Verification Properties**
  - Most properties were invariants.
    - E.g. "The commanded roll angle will be zero during the alpha recovery and NZ hold phases."
    - iphase = 6 or iphase =5 phic_at = 0
  - Some properties 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."

Results & Recommendations

- **Results**
  - Systematic ambiguity in natural language/pseudo-code
  - Missing initial values
  - 1 modified constant
  - *(We did not catch any of the errors in the Change Request!)*
- **Comments**
  - Difficult abstraction step
    - from imperative to state machine model
  - Restrictions from the semantic model:
    - completeness; dependency cycles; functional decomposition
    - Needed deep understanding of SCR semantics
    - (Sometimes had to fudge it)
  - Restrictions from the toolset:
    - trig functions; slow consistency checking
    - (got through several versions)
- **Caveat…**
  - …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 scaleability does *not* appear to be a major issue
- Bad news:
  - Couldn't model existing structure
  - Couldn't isolate change

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
  - The language
  - Case study
  - Advantages and disadvantages
- Example analysis technique: Model Checking
  - How it works
  - Case Study
  - Advantages and disadvantages

you are here!

- Where next?

Model Checking

- Has revolutionized formal verification:
  - emphasis on partial verification of partial models
  - fully automated
  - applies at any stage of sofware development
- What it does:
  - Mathematically – computes the "satisfies" relation:
    - For a given temporal logic theory, checks whether a given finite state machine is a model for that theory.
  - Engineering view – checks whether properties hold:
    - For a given model (expressed as a finite state machine), 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

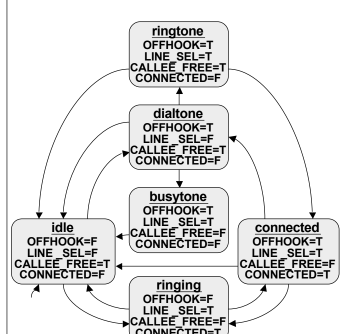
Model Checking Basics

- 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 *W* OFFHOOK])

If you make a call, the phone cannot ring without returning to idle first:
AG((ringtone busytone) □ A[□ringing *W* 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)

Case Study I - Background

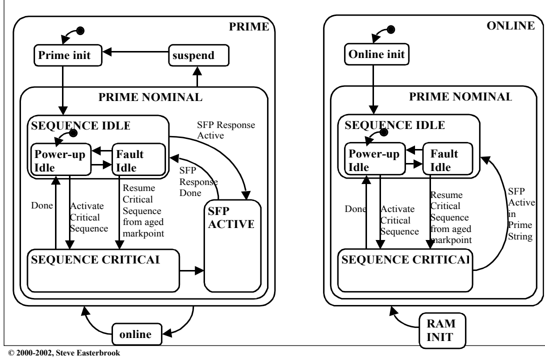
→ Dual Redundant Spacecraft Controller

- 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)



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^{87} states

→ Model reduction strategies:

- Projection - identified 5 fault classes, treated them separately:
- Symmetry - doesn't matter which processor runs which string
- Partitioning - ignore cases where critical sequence is not executing
- Abstraction - removed all data from the STB other than that needed for rollback
- Abstraction - simplest possible input data: minimal length critical sequence
- Projection - verify each LTL property separately

(Final 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 □ □(p □ ÷q)

  E.g. p = (t<3) □ (SFP=1) □ (mp_current ≠ start)

  q = (pc = mp_previous) □ (SFP = 0)
- 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
  - The language
  - Case study
  - Advantages and disadvantages
- Example analysis technique: Model Checking
  - How it works
  - Case Study
  - Advantages and disadvantages
- Where next? *you are here!*

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 - producing an analyzable model
    - Determine which verification properties you are interested in first;
    - use these to guide the modeling process

Technology Transfer Issues

- Expertise needed
  - Knowing when and how to apply formal verification requires much expertise
  - Building a state machine model is straightforward
  - Building a *checkable* state machine model is much harder
    (Need training on how modeling features affect state space size)
  - Identifying appropriate properties to verify requires domain expertise
  - Expressing these properties formally is relatively straightforward
- Who performs the analysis?
  - Programmers don't like to program the same problem twice
    (can models be extracted from code??)
  - 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
  - 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 specification
  - 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

Further Reading I

- Introductions to Formal Methods:
  - Huth M. & Ryan, M. "Logic for Computer Science: Modeling and Reasoning about Systems". Cambridge University Press, 1999
  - Heitmeyer, C. and Mandrioli, D. "Formal Methods for Real-time Computing" Wiley, 1996.
  - E.M. Clarke and J. Wing. "Formal Methods: State of the Art and Future Directions". ACM Computing Surveys, 28(4):626-643, December 1996.
  - Jackson, D. and Wing, J. "Lightweight Formal Methods". IEEE Computer, April 1996.
- On modeling in RE:
  - M. Jackson, "Software Requirements and Specifications: A lexicon of Practice, Principles and Prejudices", Addison Wesley, 1995.
- On formal methods and technology transfer:
  - D. Craigen and S. Gerhart and T. Ralston, "Formal Methods Reality Check: Industrial Usage", IEEE Transactions on Software Engineering, vol 21, no 2, pp90-98, Feb 1995.
  - J.A. Hall, Seven Myths of Formal Methods. IEEE Software, 7(5):11-19, September 1990.
  - J.P. Bowen and M.G. Hinchey, Seven More Myths of Formal Methods. IEEE Software, 12(4):34-41, July 1995

Further Reading II

→ **On the formal techniques described in this tutorial:**

SCR:

- C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications," IEEE Transactions on Software Engineering and Methodology, vol. 5, pp. 231-261, 1996.

Model checking:

- E. Clarke, O. Grumberg, and D. Peled. "Model Checking". MIT Press, 1999.
- M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In Proceedings of 21st International Conference on Software Engineering, May 1999.

→ **Case Studies:**

- T. Sreemani and J. M. Atlee. "Feasibility of Model Checking Software Requirements: A Case Study". In Proceedings of COMPASS'96, Gaithersburg, Maryland, June 1996.
- S. M. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling". IEEE Transactions on Software Engineering, vol. 24, (1), 1998.
- F. Schneider, S. M. Easterbrook, J. R. Callahan and G. J. Holzmann, "Validating Requirements for Fault Tolerant Systems using Model Checking", Third IEEE Conference on Requirements Engineering, Colorado Springs, CO, April 6 - 10, 1998.
- V. Wiels and S. M. Easterbrook, "Formal Modeling of Space Shuttle Software Change Requests using SCR", Proceedings, Fourth IEEE International Symposium on Requirements Engineering (RE'99), Limerick,Ireland, June 7-11, 1999.