DEVELOPING HIGH ASSURANCE SYSTEMS:
ON THE ROLE OF SOFTWARE TOOLS

CONNIE HEITMEYER
CENTER FOR HIGH ASSURANCE COMPUTER SYSTEMS
NAVAL RESEARCH LABORATORY
WASHINGTON, DC

22nd International Conference on
Computer Safety, Reliability, and Security
SAFECOMP 2003
OUTLINE

• Introduction
• Background
  – Overview of SCR requirements method
  – SCR Tools
• Applying tools in the development of high assurance systems
  – A-7 Operational Flight Program (U.S. Navy)
  – Rockwell’s Flight Guidance System
  – U.S. Navy’s Weapon Control Panel
  – NASA’s Flight Protection Engine
  – U.S. Navy Family of Cryptographic Devices
• Problems tools cannot solve
• Summary and Conclusions
WHAT ARE HIGH ASSURANCE SYSTEMS?

HIGH ASSURANCE COMPUTER SYSTEM

computer system where **compelling evidence is required** that the system delivers its services in a manner that satisfies certain **critical properties**

CLASSES OF HIGH ASSURANCE SYSTEMS

- **SECURE**: Prevents unauthorized disclosure, modification, and withholding of sensitive information
- **REAL-TIME**: Delivers results within specified time intervals
- **SURVIVABLE**: Continues to fulfill its mission in the presence of attacks, accidents or failures
- **FAULT-TOLERANT**: Guarantees a certain quality of service despite faults, such as hardware, workload, or environmental anomalies
- **SAFE**: Prevents unintended events that result in death, injury, illness, or damage to property

MATHEMATICS VS. ENGINEERING

MATHEMATICAL RESOURCES
(e.g., theories, models, and algorithms)

Logics (predicate, 1st order, higher order, etc.)
Automata models
Theories underlying decision procedures

...
HOW CAN TOOLS HELP IN DEVELOPING HIGH ASSURANCE SYSTEMS?

• Three major problems in software development
  – High cost of developing software
  – Lengthy software development times
  – Software errors

• Tools can help reduce all three
  – Can reduce software development costs
    • Automating a task can dramatically reduce the cost of the task
  – In many cases, can perform analysis much faster than humans
    • Often, a tool can do a task in fractions of a seconds
    • Doing the task manually can require orders of magnitude more time
  – Can find errors humans miss
    • Typically, human inspections overlook many errors
    • For certain classes of errors, tools can find ALL of the errors
HISTORY OF
SCR APPROACH

- **1978**: Heninger, Parnas+ publish A-7/SCR requirements document
  - Tabular notation
  - Events and conditions
  - Mode classes and terms

- **1980s-early 1990s**: SCR applied to a wide range of systems
  - Telephone networks (AT&T Bell Labs)
  - Submarine communications (NRL)
  - Control software for nuclear plants (Ontario Hydro)
  - Avionics software (Grumman)

- **Early 1990s**: Development of Four Variable Model and CoRE
  - Parnas+ introduce and apply Four Variable Model
  - Softw. Productivity Consortium develops CoRE method (based on SCR)
  - Lockheed applies CoRE and SCR tables to C-130J flight program

- **1992-present**: NRL develops formal SCR model and tools

**SCR → Software Cost Reduction**
**SCR GOAL: MAKE ‘FORMAL METHODS’ PRACTICAL**

**SPECIFY THE SYSTEM PRECISELY**

Use a **TABULAR notation** with an **explicit formal semantics** to specify the required behavior.

**APPLY “CONSISTENCY CHECKING”**

Automatically check spec for syntax/type errors, missing cases, nondeterminism, circular defs, etc.

**SIMULATE THE SYSTEM BEHAVIOR**

Symbolically execute the system based on the (executable) req. specs.

**VERIFY SPECS USING MODEL CHECKING**

Check critical application properties.

**VERIFY SPECS USING THEOREM PROVING**

- Usable, scalable **tabular notation**
- Integrated set of robust, powerful **software tools**
  - **light-weight tools** whose use does not require math. sophistication/thm proving
  - **heavy-duty tools** (e.g., theorem prover)

As we move down the chain, we increase assurance in the spec.
SCR TOOLS FOR DEVELOPING SOFTWARE REQUIREMENTS*

- Consistency and completeness – Is the spec well-formed?
- Validation – Is this the right spec?
  - I.e., does the spec capture the intended behavior?
- Verification – Is the spec right?
  - I.e., does the spec satisfy critical properties (e.g., safety, security)?

*Heitmeyer et al., Proc. CAV ‘98.

SCR TOOLSET

- most mature tools
- installed at 100+ org’ns in industry, govt., and academia

New ANALYSIS TOOLS

- THEOREM PROVER
- PROPERTY CHECKER (Salsa)
- INVARIANT GENERATOR

system spec

- SPECIFICATION EDITOR
- SIMULATOR
- MODEL CHECKER
- DEPENDENCY GRAPH BROWSER
- CONSISTENCY CHECKER

*Heitmeyer et al., Proc. CAV ‘98.
TOOLS FOR TESTING & CODE SYNTHESIS ARE BEING DEVELOPED

**SCR TOOLSET**
- most mature tools
- installed at 100+ org’ns in industry, govt., and academia

**ANALYSIS TOOLS**
- TAME is an interface to PVS designed to prove properties of state machine models

---

Next step: Optimized, provably correct source code
USE OF SCR TOOLS
BY LOCKHEED-MARTIN (LM)

• LM using SCR in U.S. rocket programs -- Atlas 5, J2, IUS for satellite launch
• LM in Denver used SCR to detect critical error in software controlling landing procedures in the Mars Polar Lander
  – "most likely cause of $165M failure of Mars Polar Lander in Dec. 99"*
• SCR is a key component of RETTA, the software approach described in LM's winning proposal for the Joint Strike Fighter**
  – Goal of RETTA (Requirements Testability and Test Automation) is "early defect prevention"
  – "such formalized techniques [i.e., SCR] have not been used previously because requirements have been expressed using pseudo-formal models and textual documents written in English prose"

**Lockheed Martin report, August, 2000 (Proprietary Information).
A-7 requirements document contains a complete spec of the required externally visible behavior of the A-7 flight program.

Checked manually for errors by two independent review teams.

Results of analyzing the specs with our consistency checker:
- Check of 36 condition tables, a total of 98 rows
  - Results: 17 rows in 11 tables violated the Coverage Property (i.e., 17 missing cases detected)
- Checked all 3 mode transition tables, a total of 700 rows (4319 logical expressions)
  - Results: 57 violations of the Disjointness Property were detected (i.e., 57 instances of non-determinism detected)
- All checks performed in a few minutes

Consistency checking finds MANY errors that human inspections miss and usually does so in a very short time (seconds to minutes).
For each error detected, the **consistency checker** displays

1. the table containing the error with erroneous entry highlighted
2. a state pair demonstrating the error (counterexample)

**Event that could trigger either transition**

@T(Doppler_up) WHEN [NOT CA_stage_complete AND latitude > 70 deg. AND NOT present_position_entered AND NOT latitude > 80 deg. AND IMSMODE=Gndal]
APPLYING THE SCR TOOLS TO ROCKWELL’S FLIGHT GUIDANCE SYSTEM

- SPECIFICATION EDITOR
- CONSISTENCY CHECKER
- SYSTEM SPEC
  - terms
  - cont vars
  - mon vars
  - modes
  - conditions
  - events
- SIMULATOR
Experimental application of SCR tools by Rockwell

- Despite extensive reviews by Rockwell engineers, the tools found many errors in the spec
  - 28 errors detected, “many of them significant”
  - one third each: constructing the specification, applying the completeness and consistency checks, and simulating the system behavior based on the specification

Example: Disjointness error leading to two possible flight modes

Example: Missing cases (Lateral Armed Annunciation field undefined in certain cases)

“...preliminary execution of the specification and completeness and consistency checking [with the SCR tools] has found several errors in a specification that represented our best effort at producing a correct specification manually.”

Steve Miller
Rockwell-Collins Aviation
APPLYING THE SIMULATOR AND MODEL CHECKING TO A WEAPONS CONTROL PANEL

SPECIFICATION EDITOR

CONSISTENCY CHECKER

MODEL CHECKER

DEPENDENCY GRAPH BROWSER

SIMULATOR

system spec

modes

terms

cont vars

conditions

events

mon vars
ANALYZING A CONTRACTOR REQ. SPEC OF A WEAPONS CONTROL PANEL

WCP OVERVIEW

- WCP used to prepare & launch weapons
- Sizable, complex program (~15KLOC)
- Monitored quantities
  - switches and dials
  - numeric quantities (read by sensors)
- Controlled quantities
  - lights
  - doors and valves (set by actuators)

PRODUCING THE SCR SPEC

- Used scanner and OCR to read in contractor spec of the WCP (250+ vars)
- Used text editor to convert to SCR spec

USER-FRIENDLY SIMULATION

- Scanned in diagrams of operator interface
- Used interface builder to develop realistic simulator front-end
- Operators unfamiliar with SCR can run scenarios to validate requirements spec
Opening the Torpedo Tube Vent Valve shall be prevented unless the Missile-to-Torpedo-Tube differential pressure is within safe limits.

\(@T(\text{cVENTSOLENOID}) \Rightarrow\)

\(k_{\text{MinTRANS_OK}} < \text{TRANS}_{A'} \land \text{TRANS}_{A'} < k_{\text{MaxTRANS_OK}} \lor\)
\(k_{\text{MinTRANS_OK}} < \text{TRANS}_{B'} \land \text{TRANS}_{B'} < k_{\text{MaxTRANS_OK}}\)

minimum allowable for launch

maximum allowable for launch
MODEL CHECKING THE WCP SPECIFICATION (1)

PROBLEM: Too many variables
SOLUTION: Remove variables irrelevant to the validity of the property

Technique used analogous to code "slicing"

Reduces spec from 250+ to 55 variables (~80% reduction)
MODEL CHECKING THE
WCP SPECIFICATION(2)

PROBLEM: Some variables are real-valued

SOLUTION: Apply data abstraction -- i.e.,
replace each real-valued variable with a
variable with a small, discrete value set

EXAMPLE

- Spec refers to real-valued variable tSEL_TRANS in two expressions:
  \[ tSEL_TRANS < 14.8 \text{ and } tSEL_TRANS < 9.2 \]
- The first expression partitions the interval \([l, u]\) into 2 subintervals
- The second expression partitions the interval \([l, 14.8)\) into 2 subintervals
- The new abstract variable has the type set \(\{0, 1, 2\}\).
- The function \(f\) mapping the concrete var to the abstract var is defined by

\[
f(tSEL_TRANS) = \begin{cases} 
0 & \text{if } l \leq tSEL_TRANS < 9.2 \\
1 & \text{if } 9.2 \leq tSEL_TRANS < 14.8 \\
2 & \text{if } 14.8 \leq tSEL_TRANS \leq u 
\end{cases}
\]
USING SIMULATION TO VALIDATE VIOLATION OF A SAFETY PROPERTY

**Corresponding system history** (each input and its results)

- Simulator notification of violation in *spec*
- Spin notification of violation in abstract model

**Input sequence (scenario)** that produces violation

9/24/03
## APPLYING SCR TO WCP:
### REQUIRED EFFORT

<table>
<thead>
<tr>
<th>TASK</th>
<th>PERSON-WEEKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translate contractor SRS into SCR</td>
<td>0.8</td>
</tr>
<tr>
<td>Use light-weight tools to detect errors</td>
<td>0.2</td>
</tr>
<tr>
<td>Correct errors</td>
<td>0.3</td>
</tr>
<tr>
<td>Abstraction/Detection of safety violation</td>
<td>0.7</td>
</tr>
<tr>
<td>Develop customized simulator front-end</td>
<td>3.0</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>5.1</strong></td>
</tr>
</tbody>
</table>

This small effort is quite surprising given that
- the contractor-produced SRS was large and complex
- the contractor had *no prior knowledge* of SCR
APPLYING THE SCR TOOLS, INCLUDING THE TEST CASE GENERATOR, TO NASA’S FAULT PROTECTION ENGINE (FPE)
PROBLEM

• NASA is using slightly different implementations of the FPE in various spacecraft
• NASA needs high reliance in the correctness of each version of the FPE code
• Our task
  – To develop a formal spec of the FPE beh.
  – From the spec, to construct a set of test cases satisfying some coverage criteria
  – The tests will be used to check the FPE code
SPECIFICATION-BASED
TEST CASE GENERATION

• Construct test predicates that “cover” the specification
  – Start with the set of (total) functions whose composition form the next state predicate
  – Given a function, define a predicate for each part of the function definition
  – Each predicate is called a test predicate and is the basis for defining a set of test cases

• Construct the test cases from the test predicates
  – Use the ability of a model checker to construct counterexamples
  – The set of test cases constructed is a test suite and can be used to automatically test the conformance of a program with a formal specification

For details, see Gargantini/Heitmeyer, *Proc., ESEC/FSE ‘99.*
PROGRESS TO DATE

• An SCR spec that is well-formed and relatively easy to understand
  – NASA personnel quickly learned to understand the SCR spec

• A simulator for use in validating the spec
  – Highly effective in helping to debug the spec
  – Summer intern found a serious error in the SCR spec by experimenting with the graphical simulator

• A complete set of test cases have been constructed from the spec using our testing tool and the model checker Cadence SMV
SCR LANGUAGE

• FPE algorithm involves many complex constructs that do not normally arise in embedded systems
  – e.g., feedback loops, queues, arrays simult. events, priorities, etc.

• Problem: How to specify these

PROPERTIES/LIKELY CHANGES

• How to determine what these are
• None of this is captured in the current NASA documentation

TEST CASE GENERATION

• How to deal with the input data at a more abstract level
• How to reduce length of the test cases

Solution: apply symbolic model checking -- produces shortest counterexample

Trade-off: analysis more difficult
APPLYING THE SCR TOOLS TO CD I, A MEMBER OF A FAMILY OF CRYPTO SYSTEMS
CD FAMILY OF
CRYPTOGRAPHIC DEVICES

CD SERVICES

- Load (and zeroize) crypto algorithms and keys
- Configure channel (i.e., write alg and key into channel space)
- Encrypt and decrypt data using a crypto algorithm and a key
- Take emergency action when, e.g., device is tampered with
- Provide the above services for m channels

Each member is implemented in hardware and software

CD: Cryptographic Device
Objective
- Reduce human effort needed to verify properties with a theorem prover

Design Goals
- Easy to create specs
- Natural formulation of properties
- ‘Natural’ proof steps that match in size/kind steps used in hand proofs
- Proofs similar to hand proofs

Why build upon PVS?
- Avoid reinventing existing, well-known techniques
- Use PVS logic as a flexible means of further proof support for automata models
- State properties in the expressive but natural logic of PVS
<table>
<thead>
<tr>
<th>HUMAN-STYLE</th>
<th>PVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>In proving $A \Rightarrow B$ : “suppose $A$ ”</td>
<td>(FLATTEN)</td>
</tr>
<tr>
<td>In proving $\forall a. P(a)$ : “fix $a = a_0$”</td>
<td>(SKOLEM &lt;fnum&gt; “a0”)</td>
</tr>
<tr>
<td>“By the definition of &lt;function&gt;”</td>
<td>(EXPAND “&lt;function&gt;”)</td>
</tr>
<tr>
<td>To show “$\exists a. P(a)$ because $P(a_0)$”</td>
<td>(INST &lt;fnum&gt; “a0”)</td>
</tr>
<tr>
<td>??? (A miracle happens here -- maybe)</td>
<td>(GRIND)</td>
</tr>
<tr>
<td>Knowing “event $\pi$ precedes state $s$ and $P(\pi, s)$ holds” adduce “the last event $\pi_0$ before $s$ such that $P(\pi_0, s)$”</td>
<td>(let ((exists_case_body (format nil ...))...)...) (then (branch (case exists_case_body)) (then ... (branch (apply_lemma “last_event”(...)...)))) (then (branch (auto_cases inv)) (then(base_caseinv)(systimpl_simp_probe) (postpone)) (branch (induct_cases inv) (then (reduce_case_one_var_exp inv “t_1”) (match_univ_and_systimpl_simp_probe) (postpone)) ... (then (reduce_case_no_var_exp inv) (match_univ_and_systimpl_simp_probe) (postpone)) (let ((eps_lemma ...) (inst_pred ...)) (then (lemma eps_lemma) (inst -1 inst_pred) (branch (split -1) ((...)(postpone))))))))</td>
</tr>
</tbody>
</table>

In starting the proof of a state invariant: “Use induction.”

Introduce the constraints applying to a nondeterministic $\varepsilon$ value in the poststate

TAME Goal: Provide **natural** proof steps
SECURITY PROPERTIES

1. When the zeroize switch is activated, the keys are zeroized.
2. No key can be stored before an algorithm in the assoc. location is activated.
3. If undervoltage occurs in backup power while primary power is un-available, CD enters alarm or off mode.
4. If backup power is overvoltage, then CD is in initialization, standby, alarm, or off mode.
5. When an overvoltage occurs in primary power, then CD is in standby, alarm or off mode, or goes into initialization.
6. When an undervoltage occurs in primary power, then CD is in standby, alarm, or off mode, or goes into initialization mode.
7. If CD is tampered with, the keys are zeroized.

PROVED DIRECTLY BY INDUCTION USING TAME
SECURITY PROPERTIES

1. When the zeroize switch is activated, the keys are zeroized.
2. No key can be stored before an algorithm in the assoc. location is activated.
3. If undervoltage occurs in backup power while primary power is unavailable, \( CD \) enters alarm or off mode.
4. If backup power is overvoltage, then \( CD \) is in initialization, standby, alarm, or off mode.
5. When an overvoltage occurs in primary power, then \( CD \) is in standby, alarm or off mode, or goes into initialization.
6. When an undervoltage occurs in primary power, then \( CD \) is in standby, alarm, or off mode, or goes into initialization mode.
7. If \( CD \) is tampered with, the keys are zeroized.

AUTOMATICALLY GENERATED INVARIANTS*

- In Initialization mode, primary power is not unavailable.
- In Configuration mode, the system is healthy, backup power is not overvoltage, and primary power is not unavailable.
- In Idle mode, the system is healthy, backup power is not overvoltage, and primary power is not unavailable.
- In Traffic Processing mode, the system is healthy, backup power is not overvoltage, and primary power is not unavailable.
- In Off mode, KeyBank1Key1=0 and ...

ANOTHER SERIOUS PROBLEM THAT TOOLS & TECHNOLOGY CANNOT SOLVE

• A major barrier to using tools in developing high assurance systems: The lack of high quality specs

• Attributes of a high quality specification
  ▪ Precise
  ▪ Unambiguous
  ▪ Minimizes redundancy
  ▪ Minimizes implementation bias
  ▪ Readable
  ▪ Organized as a reference document -- info is easy to find

• Is UML the/a solution? IMHO, No…
  – Ambiguous: Lacks a formal semantics
  – Too much opportunity for implementation bias

• What is needed
  – Higher quality specs
  – Research in spec languages
  – Technology that makes it easier for practitioners to write good specs
ON THE ROLE OF TOOLS FOR STATIC ANALYSIS OF CODE

• Recently, a number of tools for static analysis of code have been developed (mostly for C and Java) that detect code that could lead to faults, e.g., buffer overflows, bad pointers, and arithmetic exceptions
  – Some are commercially available, e.g., Safe C, Codesurfer
  – Some are proprietary, e.g., SNAP (T. Ball at Microsoft Research)
  – Others have been developed at universities, e.g., ARCHER for C (D. Engler et al., ESEC/FME 2003, Helsinki), BOGOR for Java (M. Dwyer et al., ESEC/FSE 2003, Helsinki)

• “Integrity static analysis” (see Bishop, Bloomfield, et al., Proc., SAFECOMP 2003) using such tools should be highly effective in detecting code that could lead to a failure in a high assurance system

• Such an approach should be especially effective for developing high assurance for legacy, third-party, and COTS software

However, to achieve high confidence that a system satisfies critical safety (or security) properties, such analysis is not enough: it should be combined with other analyses that detect violations of application properties
Needed: A collection of well-founded software engineering disciplines, each customized for a particular class of software, e.g.,

- Automobile software
- Software for medical devices
- Web software
- Avionics software
- Software for security products
- …
SUMMARY

• **Tools can be extremely useful in developing/evaluating software**
  – Find missing cases and unwanted non-determinism
  – Help in validating a formal spec
  – Detect property violations
  – Support formal verification of properties
  – Reduce the time/effort required to construct and run test cases
  – Provide more confidence in testing by constructing a carefully constructed suite of test cases

• **Most effective: A combination of tools**
  – Different tools usually find different kinds of errors

• **A major contribution of tools: Liberate people to do the hard intellectual work required to build high quality specs and software**
  – Moreover, the “combination of human analysis and tool-based analysis is more powerful than either alone…” (paraphrasing John Rushby)

• **But, powerful tools are not enough**
  – Need **better methods** for developing high assurance software
  – Need **better specifications**
  – Need **better spec languages**
MY REACTION TO MARTYN’S TALK

• Where I agree
  – The emphasis in developing and certifying a high assurance system should be on the product (especially the system and the software) and its properties, not the process
    • Martyn’s case against the SILS was very convincing
  – Strong software engineering principles should be applied
  – A correct formal spec of a high assurance system is critical

• Where I disagree
  – In our experience, it costs significantly more “to do things properly”
    • Doing so requires much more thought AND more competent people
  – Students do not generally receive adequate training in software engineering in our universities
    • Certainly, this is the case in the U.S.
  – Both a formal proof AND testing can be usefully applied to a single artifact
    • A proof demonstrates that the artifact satisfies a single property of interest
    • Testing with good coverage evaluates a much wider range of behaviors