When Can Formal Methods Make a Real Difference?

- -

Peter G. Neumann
Computer Science Laboratory
SRI International
Menlo Park, CA 94025-3493
Neumann@CSL.sri.com
www.csl.sri.com/neumann
Tel 1-650-859-2375
ICFEM, 10 November 2004

Some Regions in the FM Spectrum

- Theorem proving
- Model checking
- Model-based design
- Formal specification
- Formal test-case generation
- Semi-formal approaches, heuristics
- Hand-waving ad-hoc-ery

These can be applied in the small up to entire systems in the large (with increasing effort). System-oriented analyses are needed, especially for critical systems.

Scope of Analyses

- Hardware/software life-cycle issues
- Requirements, architectures, algorithms, specs, protocols, components, subsystems, applications, total systems, enterprises
- Compositional soundness (for requirements, policies, specs, code, proofs)
- Dependency analyses (modules, specs, code)
- Code/spec & spec/requirement consistency
- Static analysis of software (buffer overflows, synchronization, race conditions, type safety, memory leaks, memory residues, etc.).

Properties of Compositions

- -

We seek to evaluate properties of systems developed as predictable compositions of subsystems– e.g., emergent higher-level properties such as system trustworthiness, survivability, end-to-end security with untrustworthy intermediates, avoidance of propagating failures, interoperability, ...
Decomposability: byproduct of composability.
(Existing research is generally either inadequate or ignored.)
See my CHATS report (URL follows).

System-Oriented Analyses

- Horizontal (modular) composition
- Vertical (hierarchical) composition, with state mappings and functional dependence
- Whole systems as subsystem compositions
- Temporal refinement consistency
- Other life-cycle considerations
- All of the above in coherently unified usage
- Focus here is on critical systems, but is also relevant to mass-market systems.

Emergent Trustworthiness Properties:

- -

- Human safety in critical systems
- Total-system reliability
- System survivability despite adversities
- System/network security/privacy
- Interface usability/robustness/perspicuity/ compatibility/interoperability

Application areas: critical infrastructures, aviation, space, DoD, health care, finance, law enforcement, national security, etc.

Static Analysis

- -

Buffer/stack overflows and other flaw types are ubiquitous. But we knew how to avoid many of them in the 1960s. Useful tools include Crispin Cowan's StackGuard, Dave Wagner's buffer overflow analyzer, Ashcraft-Engler, Brian Chess, RaceTrack, slint; spec#, slam, etc. Hao Chen's MOPS uses model checking, and found some hitherto undetected bugs. (See CHATS report App. A, Hao's thesis.)

Model-Based Design Might Have Avoided?

- Patriot system clock drift (req/spec/code)
- Yorktown divide by 0 outage (app/OS)
- Airbus A320 crashes (autopilot/pilot)
- Handley-Page Victor tailplane collapse
  (3 independent but all erroneous corroborating analyses)
- Therac 25 non-atomic mode change
- John Denver's gas tank interface:
  (Up for off, Right for L, Down for R)

Formal Test Generation Might Have Avoided?

- -

- **Intel FDIV hardware flaw**
- **First Columbia shuttle synchronization**
- **Mariner 1 Venus probe (R dot bar sub n)**
- **Patriot system clock drift (spec/code)**
- **Yorktown divide by 0 outage (app/OS)**
- **Bell V-22 Osprey (correct sensor outvoted)**
- **Airbus 320 crashes (autopilot/pilot)**
- **Lauda Air crash (thrust-reversers)**
- **Automated train collisions/crashes**
- **Therac 25 non-atomic mode change**

Anticipating Error Propagation Effects

- 1980 ARPANET collapse: router memory errors, weak status-message garbage collection, node memory exhaustion
- 1990 AT&T longlines half-day collapse: unvetted upgrade in recovery software
- Both thought to have been impossible!
- Power-grid propagation outages
- Identify enabling conditions; fault-tree analysis? model checking? exhaustive iterative closure? formal verification? dependency analysis? combination of these?

Election Systems

- Requirements include end-to-end integrity, tamper-proofing, reliability, accountability, nonsubvertible voter-verified auditing, without compromising voter privacy, etc.
- We need strength in depth, but have weakness in depth: faith-based elections. Today's standards are poor; evaluations are proprietary, paid by vendors; partisan oversight: Risks everywhere.
- FM in architectures, software engineering, finding vulnerabilities, weak links?

What Are Realistic Expectations?

- Historically, significant system promise in the 1970s was too far ahead of its time, e.g., SRI's SIFT and PSOS, TSSEC A1.
- The potential gains are much greater now, because of many other advances (CPU speed/memory, model checking, tools, increased scale of application).
- Intel, Microsoft, and others countries are exhibiting renewed interest.
  Many of you are the new generation!

General Lessons Learned

- **Computer development is an incremental process, driven by marketplace forces. But security research and high assurance are slow to be adopted commercially — despite many urgent critical needs.**
- **Good requirements, sound architectures, principled developments, whole-system analyses, experience, etc. are essential! Enormous benefits await, sooner and later.**

More Lessons Learned

- Hierarchical design with modular encapsulation is a powerful design aid. It need not undermine efficiency, especially with appropriate hardware.
- A formal methodology can be valuable for increasing critical-function assurance, especially security/reliability. Spec checks can detect/prevent many flaws. Use formal methods where most effective.
- Long-term global optimization is needed.

Very High Fault Tolerance (History 1)

- -

SIFT: Software Implemented Fault Tolerance (SRI fly-by-wire system for NASA, 1973-80): redundant system (7 CPUs, 7 memories, 7 buses, 7 power supplies) with 2-out-of-3 voting, 10^{-5} processor failure probability leads to 10^{-10} system failure probability. Hierarchically layered abstractions, related functionally to one another, with higher-layer properties derived iteratively from the hardware up through the application SW.

SIFT Abstraction Hierarchy

- Markov model failure probability 10^{-10}/hr
- I/O Model: System Safe (all tasks correct)
- Task Replication Model: values voted upon on task completion
- Task Activity Model: startup, broadcast of values, voting, synchronization
- OS specs: scheduler, voter, dispatcher, buffer manager, etc.
- Pascal code for each routine
- BDX-930 Code

PSOS Design (1973-80, History 2)

- Pervasive capability addressing, tagged and nonforgeable in hardware, nonbypassable, high-end OS/application security
- Hierarachical abstraction with modular encapsulation, object oriented, strongly typed, formally specified in SRI's HDM: Hierarchical Development Methodology. Generic flaws avoided by spec language.
- Arbitrarily extensible

PSOS Capabilities

- Only two operations create capabilities: create new one, or create restricted copy.
- All objects typed, capability-addressed, nonbypassably.
- Capabilities directly accessible unless masked by some intervening layer.
- Capabilities can be tagged as propagation limited to avoid secondary use.

PSOS Implementability

- Many lower-layer ops * are executable from above, although some are hidden [ ]
- Multilevel security could be embedded in layer 0 or as a secure object type.
- Hardware could have been easily retrofitted in 1980s-style CPUs.
- PSOS-like typing is used in Honeywell/SCC secure systems.

Formal Methodology: HDM/PVS/SAL/ICS

- Use of HDM enabled formal specification: modules, interlayer state mappings, abstract implementations, system analyses.
- A complex design is conceptually simple through its hierarchy and composable modular abstraction with encapsulation.
- PVS “interpretations” now provide similar ability to consider systems in the large.

Historical Perspectives

- **Butler Lampson in the 1970s:**
  Capability systems are the way of the future, and always will be.

  (Also said of formal methods!)

- **Contrarian view from the 2000s:**
  Shortsightedness and local optimization are the way of the future, and always will be.

URLs

- -

- **PGN: www.CSL.sri.com/neumann/**
- **PGN: Principled Assuredly Trustworthy Composable Architectures: www.CSL.sri.com/neumann/ chats4.html, also .pdf, .ps**
- **Neumann-Feiertag PSOS paper, 2003: www.csl.sri.com/neumann/psos03.pdf**
- **ACM Risks Forum archives: www.risks.org**
- **SRI Formal Methods (PVS, SAL, ICS, ...): www.csl.sri.com/programs/formalmethods/**

PSOS/HDM References

- -

- **P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, L. Robinson, A Provably Secure Operating System: The System, Its Applications, and Proofs, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980.**
- **L. Robinson, K.N. Levitt, Proof Techniques for Hierarchically Structured Programs,** *CACM,* **20, 4, April 1977.**

layer	PSOS Abstraction or Function
17+	applications and user code (-)
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes*, visible input-output*
11	creation and deletion of user objects*
10	directories (*)[c11]
9	extended types (*)[c11]
8	segmentation (*)[c11]
7	paging [8]
6	system processes, input-output [12]
5	primitive input/output [6]
4	arithmetic, other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*), addressable memory [7]
0	capabilities *
Note:	
*	user-visible interface
(*)	partially visible interface
(-)	user-restrictable as desired
[c11]	creation/deletion hidden by layer 11
[i]	module hidden by layer i=6,7,8, or 12

layer	PSOS Abstraction or Function
17+	applications and user code (-)
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes*, visible input-output*
11	creation and deletion of user objects*
10	directories (*)[c11]
9	extended types (*)[c11]
8	segmentation (*)[c11]
7	paging [8]
6	system processes, input-output [12]
5	primitive input/output [6]
4	arithmetic, other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*), addressable memory [7]
0	capabilities *
Note:	
*	user-visible interface
(*)	partially visible interface
(-)	user-restrictable as desired
[c11]	creation/deletion hidden by layer 11
[i]	module hidden by layer i=6,7,8, or 12

Group	PSOS Abstraction	layers
G	user/application activities	17–...
F	user abstractions	14–16
E	community abstractions	10–13
D	abstract object manager	9
C	virtual resources	6–8
B	physical resources	1–5
A	capabilities	0

layer	PSOS Abstraction or Function
8	segmentation (*)[c11]
7	paging [8]
6	system processes, input-output [12]
5	primitive input/output [6]
4	arithmetic, other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*), addressable memory [7]
0	capabilities *
	Ideally all in hardware.

layer	PSOS Abstraction or Function
12	user processes*, visible I/O*
11	creation/deletion of user objects*
10	directories (*)[c11]
9	extended types (*)[c11]
8	segmentation (*)[c11]
4	arithmetic, other basic operations *
1	registers (*)
0	capabilities *

layer	PSOS Abstraction or Function
17+	applications and user code (-)
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes*, visible input-output*
11	creation and deletion of user objects*
10	directories (*)[c11]
9	extended types (*)[c11]
8	segmentation (*)[c11]
4	arithmetic, other basic operations *
1	registers (*)
0	capabilities *

layer	Properties for Analysis
17+	Application-relevant properties
16	Soundness of user types
15	Search-path flaw avoidance
12	Process isolation, no residues,
11	No lost objects
9	Generic type soundness
8	Segmentation integrity
6	Interrupts properly masked
4	Correctness of basic operations
0	Nonforgeable, nonbypassable, nonalterable capabilities