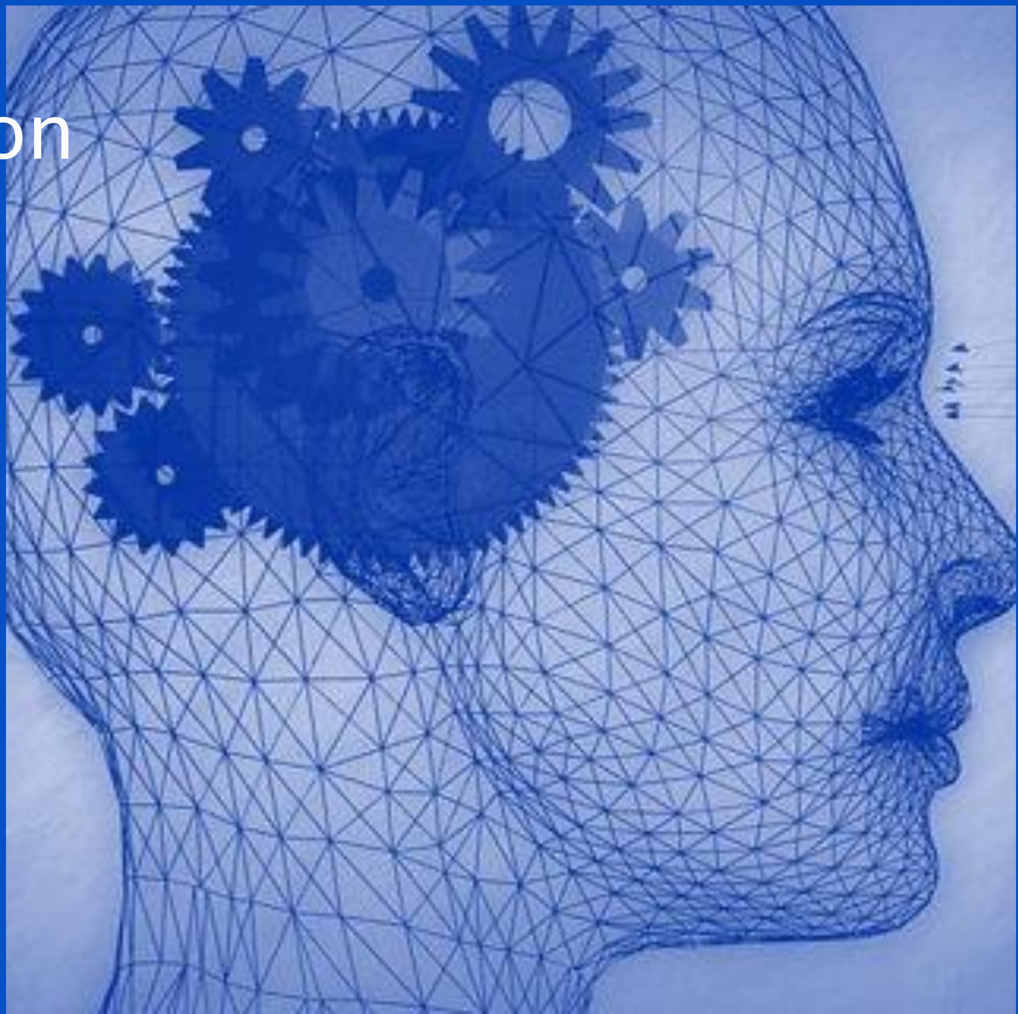
Shift left / Shift up:
Protecting Safety-Critical Software Intensive Systems from Malicious Action

Harald Ruess, N. Shankar

Formal Methods Program

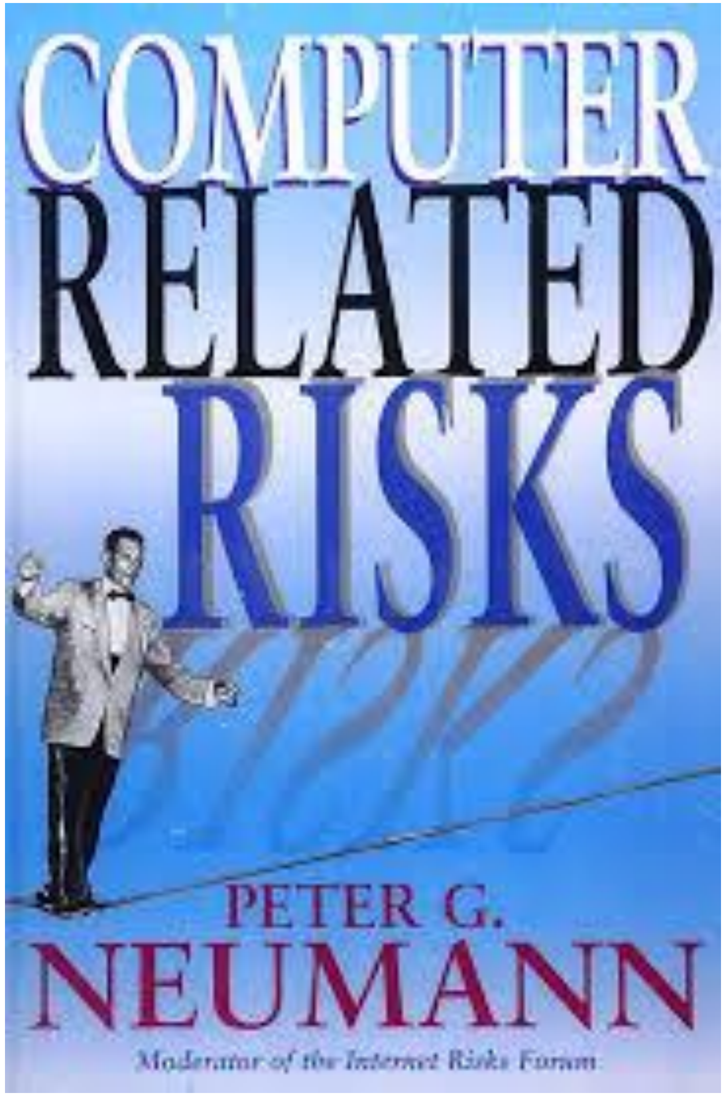
email: harald.ruess@sri.com

Annapolis, May 16, 2025



Disruption, Failures, and Adversity

Channel	Incidences
Hardware	Intel FDIV, Spectre/Meltdown
Side Channel	Electromagnetic, acoustic, power, timing, optical, radiation, wear-and-tear (Row Hammer)
Calculation	NASA Mariner, Mars Polar Lander, Mars Climate Orbiter, Ariane-5
Memory/Type	Buffer overflow, null dereference, use-after-free, bad cast
Crypto	SHA-1, MD5, TLS Freak/Logjam, Needham-Schroder, Kerberos
Input Validation	Buffer over-read (Heartbleed, Cloudbleed)
Race/Reset Condition	Therac-25, North American Blackout, AT&T crash of 1990, Mars Pathfinder
Code injection	SQL injection, cross/site scripting, malvertising, data poisoning
Provenance / backdoor	Athens Affair, Solar Winds
Social engineering	Pretexting, Honeytrap,Tailgating/Piggybacking



The Cost of Failure

The estimated engineering cost of fixing poor quality code exceeds $1 trillion annually in the U.S. alone

- with failure to patch known vulnerabilities being the largest contributor to these costs

Cybercrime thrives on code vulnerabilities, and is estimated to be another $8 trillion a year business and growing

- that is nearly $1 billion every hour

Sources:

www.synopsys.com/blogs/software-security/poor-software-quality-costs-us

cybersecurityventures.com/cybercrime-to-cost-the-world-8-trillion-annually-in-2023

Source: Pieter Bruegel the Elder

State-of-the-practice

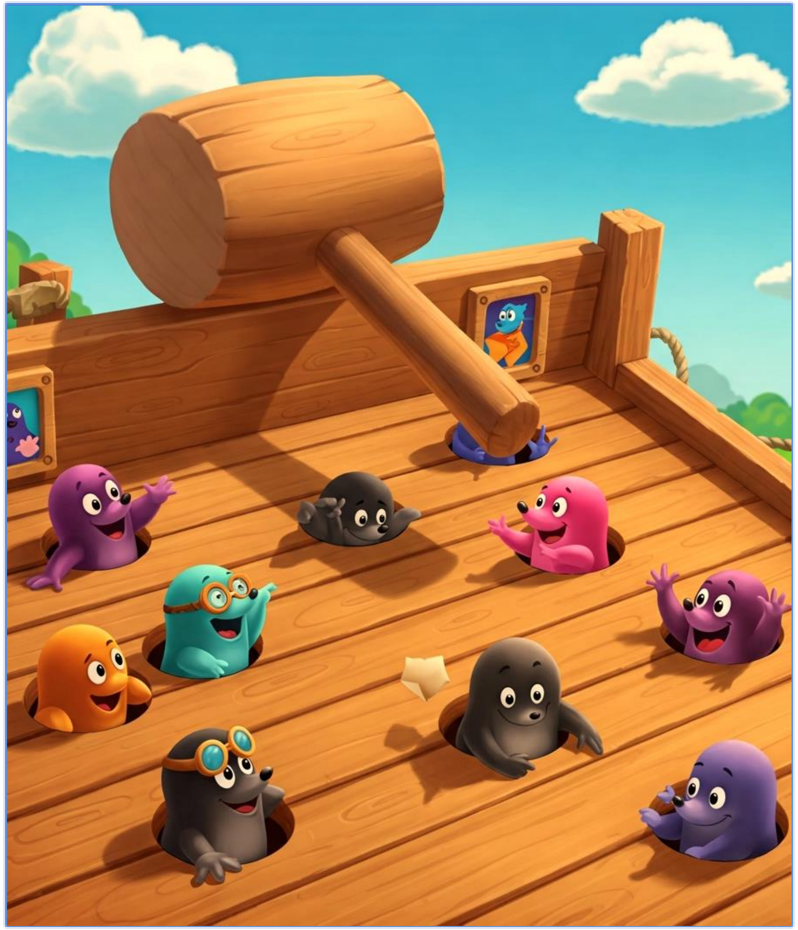
Traditional debugging is not only costly but also largely ineffective in dealing with the complexity of today's SW supply chain

- Manual inspection of largely informal specs and code
- Code testing in later development stages

Piling band-aids on top of poor-quality software only fuels an unwinnable race to the bottom

- >40,000 CVEs published in 2024, a 38% increase from 2023
- The reactive approach to cybersecurity is no longer sustainable

Code and attacks generated by current LLMs are likely to make matters worse



Generated by Gemini from unknown sources

Formal Methods to the Rescue?

FM touted to be a viable alternative to traditional bug hunting

- Integration with industrial development processes (e.g. Intel, Collins, AWS)
- Microprocessors, separation kernels, real-time operating systems, fault-tolerant algorithms, and crypto libraries nowadays formally verified almost routinely
- Billions of small theorems machine-proved every day

Satisfiability revolution (SMT, BMC, k-induction, IC3) is making Vannevar Bush's prophecy come true

But: despite all the progress, FM not widely used …

"We may someday click off arguments on a machine with the same assurance that we now enter sales on a cash register"

https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/

It's the Specification, Stupid!

Specification, formal or not, undoubtedly the Achilles' heel

- Imprecise, ambiguous, inconsistent, even outright wrong, and constantly changing
- Lack of agreed-upon specifications of basic software building blocks;
  - Messy realities of programming languages, emergent API behavior often undocumented, myriad poorly defined standards for data and exchange protocols

Verification, formal or not, rather late in the development cycle

- By then, the most egrigious design errors have already been made,
- Errors are to be found at least as often in specifications as in their implementation
- Design errors are the most costly ones to fix

Formal specification often developed as an afterthought

- Unclear relationship between generally accepted specifications (i.e. standards) and their formal counterparts

As a direct result it is not uncommon for even formally verified software to still contain alarming bugs, which is why some consider FM to be an academic *Glasperlenspiel*

Verify then Generate!

A better paradigm is to

- Align informal and formal specs
- Formally verify specs
- Generate correct code from specs
- Use of integration architecture with well-defined properties

Specification → **Verification** → **Generation**

Specification

- Autoformalizing standards documents **(Arsenal)**
- Semantic browsing for generating consistent stakeholder views of evolving (standards) documents **(Effigy)**

Verification

- Type-checking for consistent data, representation, and use (**Predicate Subtypes**, **Ontic Types**)
- V&V of specifications and abstract programs (**PVS**)

Generation

- Correct-by-construction generation of efficient and memory-safe code (**PVS2C**)
- Correct-by-construction configuration generation for multi-rate computation and communication platform (**RADL)**

PVS2C: Autogenerating Correct and Efficient Code

PVS2C compiles executable subset of PVS specifications into stand-alone C code

- Functionally correct-by-construction
- Memory-safe
- No runtime environment needed
- “Matches” efficiency of hand-coded C

Benefits

- Program verification independent of peculiarities of target programming language and underlying execution environment
- Many program optimizations applied at the declarative level, where essential identities still hold

Example (Effigy): Auto(in)formalization of PDF standard documents and generating executable code with PVS2C: testing, analysis, reference implementation.

```
Rijndael_step(A, K): byv[16] =
  LET A1 = byteSubst(A),
       A2 = shiftRow(A1),
       A3 = mixColumn(A2)
   IN
      roundkeyXOR(A3, K)

Rijndael_rec(A, KK, (i : below(11))): RECURSIVE byv[16] =
  IF i >= 10 THEN
     roundkeyXOR(shiftRow(byteSubst(A)), KK(i))
  ELSIF i=0 THEN
     Rijndael_rec(roundkeyXOR(A, KK(i)), KK,+1)
  ELSE
     LET A4 = Rijndael_step(A, KK(i)) IN
       Rijndael_rec(A4, KK, i+1)
  ENDIF
 MEASURE 10 – I

Rijndael(A, K): byv[16] =
   Rijndael_rec(A,
      allKeys((LAMBDA (k: below(11)): K),1, 0), 0)
```


RADL: CPS Integration Architecture

Component integration a key pain point in industry

RADL multirate architectural design

- Quasi-periodical execution of nodes (at their own periods)
- Communication through non-blocking channels
- Bounded drift for local clocks and bounded communication latency

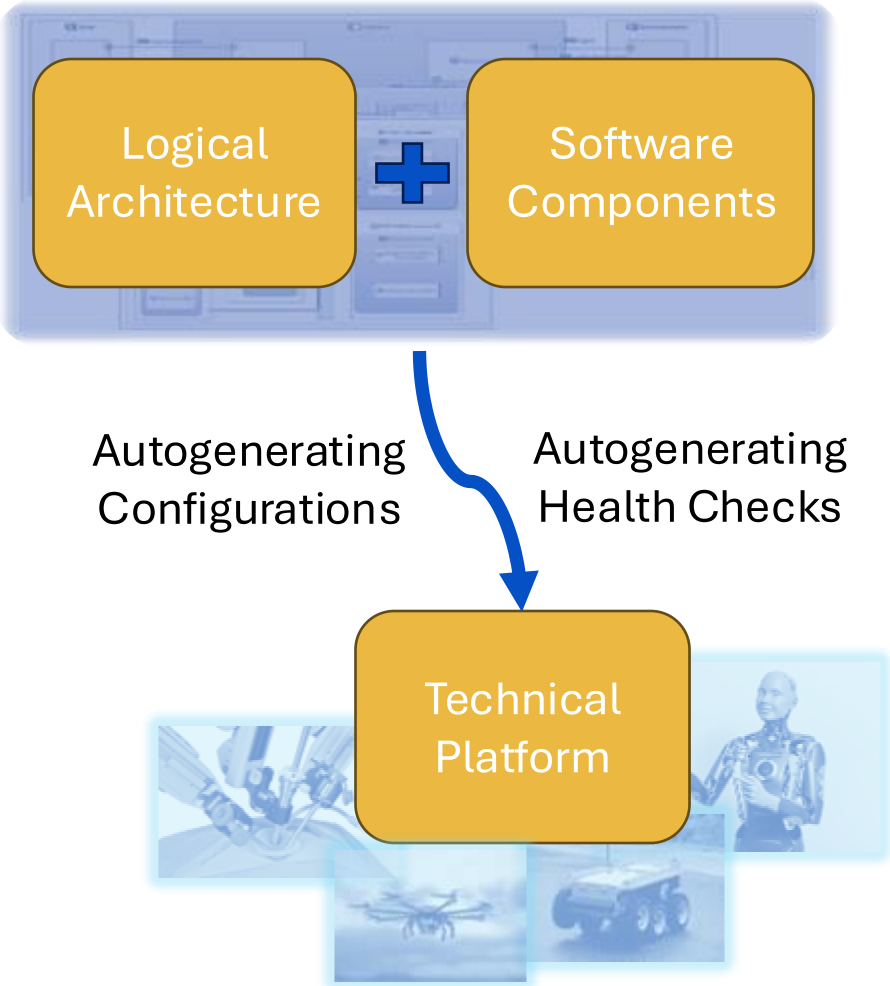
RADL covers the complete asynchronous (DDS/ROS2) to synchronous (TTP) design space

Build system autogenerates glue code for scheduling, communication, and health checks

Logical architecture guarantees (formally verified)

Message ordering, Bounded/zero message loss, End-to-end latency bounds, Failure warnings/recovery, No DoS attacks, No deadlocks

RADL properties form basis of CPS assurance cases



Evidential Toolbus (ETB)

Tool integration framework for constructing and maintaining claims supported by arguments based on evidence.

- Creating workflows (e.g. DO178C, ISO26262) and tool integration
- Decomposing claims into subclaims according to workflow
- Producing checkable evidence supporting these claims
- Maintaining the evidence against changes

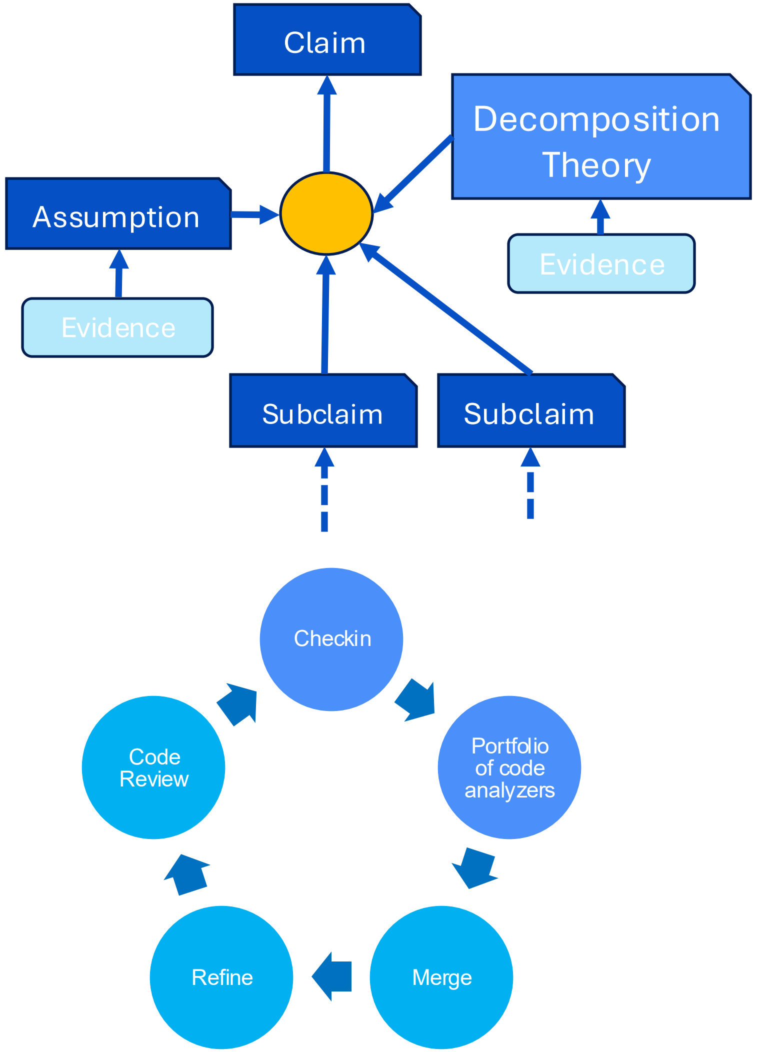
Decomposition encoded as logic programming rules

claim :- assumption, subclaim1, subclaim2.

assumption :- evidence.

Example: evidential and continuous integration of

- Portfolio of static code analyzers (a la JPL)
- Model checking to prune AI reports; counterexamples/witnesses
- Manual code review using refined reports with model checking inputs
- Integrated in Jenkins (**Disappearing formal methods…)**



Source: Beyene, R. Evidential and Continuous Integration of SW Verification Tools

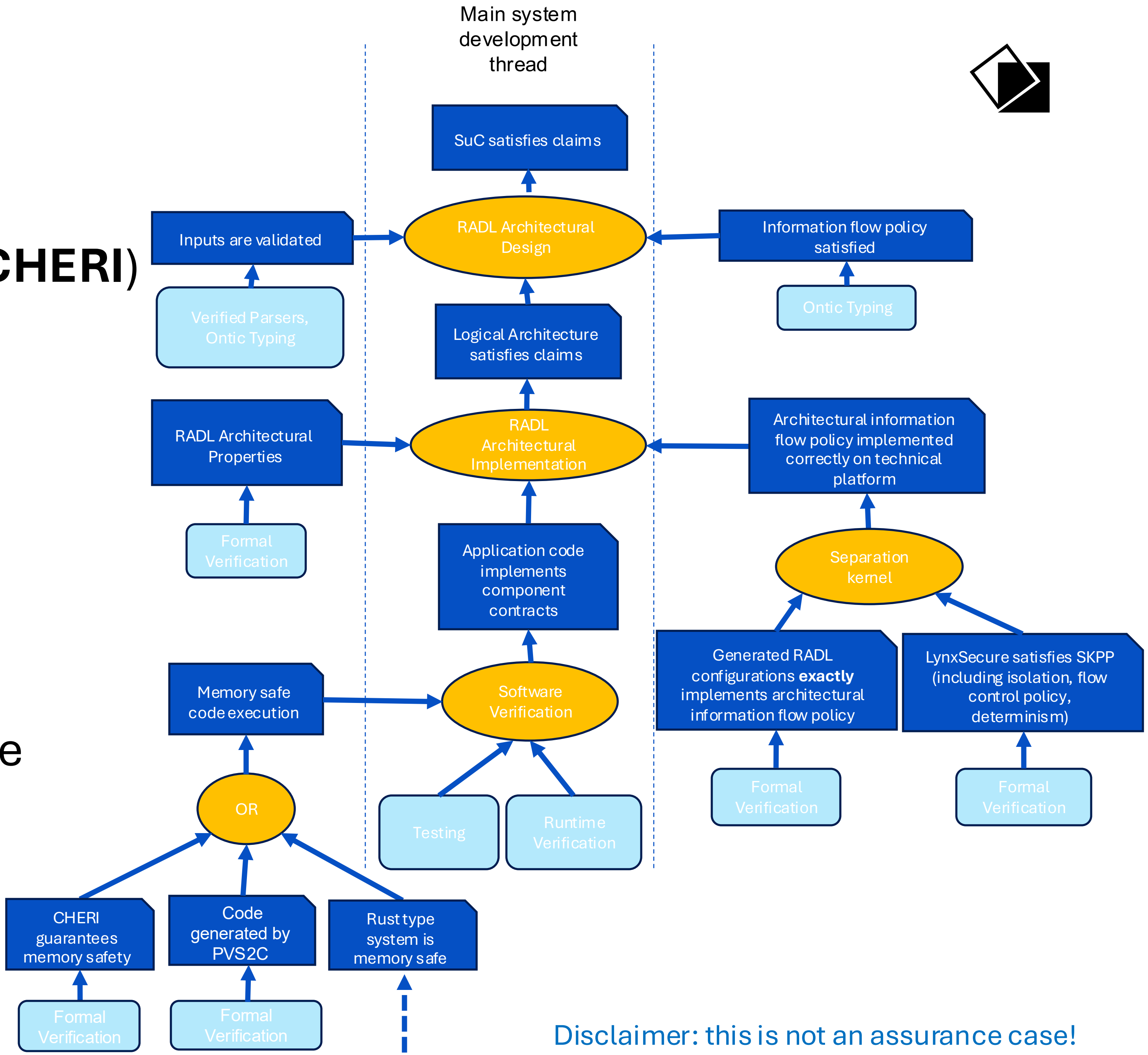
Compositional Assurance

Building blocks

1. Memory Safety (**PVS2C/Rust, CHERI**)
2. *Information flow (**Ontic Typing**)*
3. *Input Validation (**Parsley**)*
4. Isolation (**RADL, Hypervisors**)
5. ...

Efficiency

- Reduced cost of constructing the argument through reuse
- Reduced amortized cost of falsification and verification



Disclaimer: this is not an assurance case!

Conclusions

Protection of safety&security-relevant software needs to be increasingly **proactive**

- **Shift left** for early detection of design flaws
  - Identification of edge cases and failure scenarios
  - Impact of component failure on the overall system
- **Shift up** in rigor and abstraction
  - Support for compositionality, adaptability, and resilience
  - along with the curation of coherent evidence

Key enabling technologies of the **Shift left/Shift up** *paradigm* are generators such as PVS2C, RADL, and ETB

Hypothesis: with these ingredients it is **practical** and **economical** to design and build large-scale resilient software systems (within a theorem prover?)

***Impact** on continuous Authority to Operate (**cATO**) processes*

- Continuous curation of assurance cases in **CI/CD** pipelines
- Composition of assurance cases along software/hardware **supply chains**
- Independently checkable assurance cases as a central artefact of **cATO;** possibly automated

Thank you!

harald.ruess@sri.com

Acknowledgments: The SRI technology mentioned in this presentation has been supported through DARPA, IARPA, and ARPA-H grants, including HACMS, SafeDocs, DesCert, Effigy, ARCOS, CHALO, PARADIGM/POET

A Short History of Formal Methods

1970s:

SIFT: State-machine replication, modern fault tolerance

Byzantine Agreement: Tolerating faults with no assumptions on behavior; later, basis of blockchain

PSOS: Capability-based security, father of CHERI, ARM Morello

Early FM: JOVIAL Verifier, Boyer-Moore, SMT

Information Flow Analyzer: Pre-noninterference semantics

HDM: Hierarchical development of secure software

1980s:

Separation Kernel: Later evolved to MILS, also partitioning/safety

EHDM: Clock synchronization proofs

Noninterference and its intransitive form

Algebraic Semantics and rewriting

OBJ3: Modular and equational programming

IDES: Intrusion detection, evolved to network intrusion detection

Institutions: Abstract model theory for specification and programming

1990s:

Interrogator: Cryptographic protocol verification

State of the Art FM: PVS, RRM, Maude

PVS: Interactive specification and verification

Model-check the brain: human factors verification

Lazy compositional verification

RRM: performant rewrite rule machine

Maude: model-checking concurrent systems

Reflective Logic for meta-programming

Reconfiguration from first principles

AAMP5: Microprocessor verification

Bitvector decision procedures

Predicate abstraction

Floating-point verification

2000s:

ICS: First advanced SMT solver

Cyberlogic: Logic of evidential transactions

SAL: Combining finite and infinite-state model checking

CAPSL: Crypto-protocol analysis

Pathway logic: Analyzing biological pathways

Provenance: Theory and tools

State of the Art FM: PVS, Maude continue

Calendar automata: Verification of fault-tolerant distributed real-time algorithms

WMC: Witness-producing model checker

Parikh automata that count

Relational abstraction: generalizes qualitative physics

PCE: analyzing Markov logic networks

2010s:

DimSim/SimCheck: Simulink analyzer

OCCAM: Debloater

CHERI architecture

Yices2 SMT solver

State of the Art FM: PVS, Maude continue

ARSENAL: Semantic parsing

RADL: Resilient multirate architectures

Parsley: Verified parsing/unparsing

Sherlock: Neural net analyzer

Kernel of Truth: Verification of proof checkers against trusted kernel

PVS2C: : Autogenerating efficient code from specifications

Reverse engineering of hardware

ETB: Evidential toolbus, assurance workflow

OGIS: Oracle-guided inductive synthesis

SeaHorn: static analyzer

2020s:

Assurance 2.0: Rigorous assurance cases

State of the Art FM: PVS, Maude continue

Ontic types: Meaningful typing

CoProver: LLM-assisted interactive theorem proving

Stitches: Seamless system of systems integration

Zero-Knowledge: Verification of proof systems and code

DPRIVE: Hardware design for fully homomorphic encryption

CDSAT: Conflict-Driven Reasoning

Semantic Browser: Validated informal specifications

...

SRI

Assurance 2.0: Rigorous Assurance Cases

Modern approach to systematically developing, presenting, and assessing rigorous assurance cases

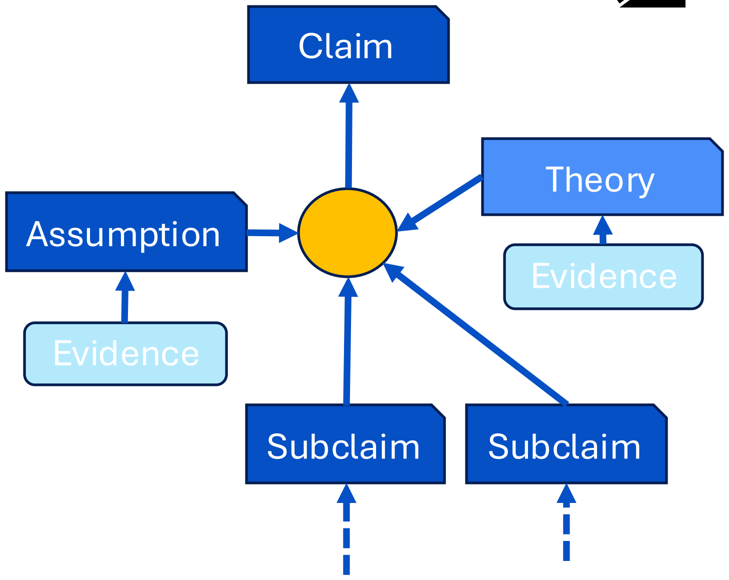
- Bloomfield, Rushby, *Confidence in Assurance 2.0 Cases,* 2024

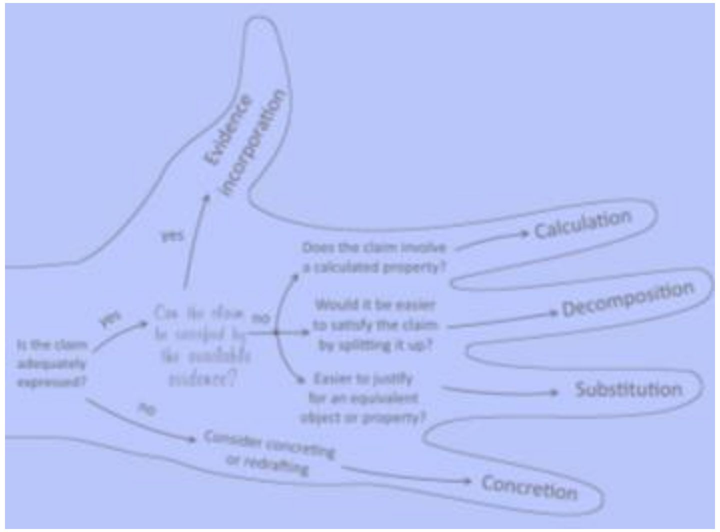
Indefeasible Confidence

- Emphasizes **deductive reasoning** in argument steps
- **Theories** that justify assurance methods
- Identification, analysis, and refutation of **potential defeaters** – arguments or evidence that could undermine the assurance case.
- **Defeaters for** the systematic identification and handling of potential doubt
- **Confirmation theory:** does evidence support a claim? how well does it discriminate against alternative claims?

Benefits

- Improved clarity and transparency, enhanced confidence as all reasonable doubts are addressed, systematic identification of weaknesses, support for innovation, automation for developing and assessing assurance cases





Source: Bloomfield, Rushby, Assurance 2.0

