Designing Software for Certification

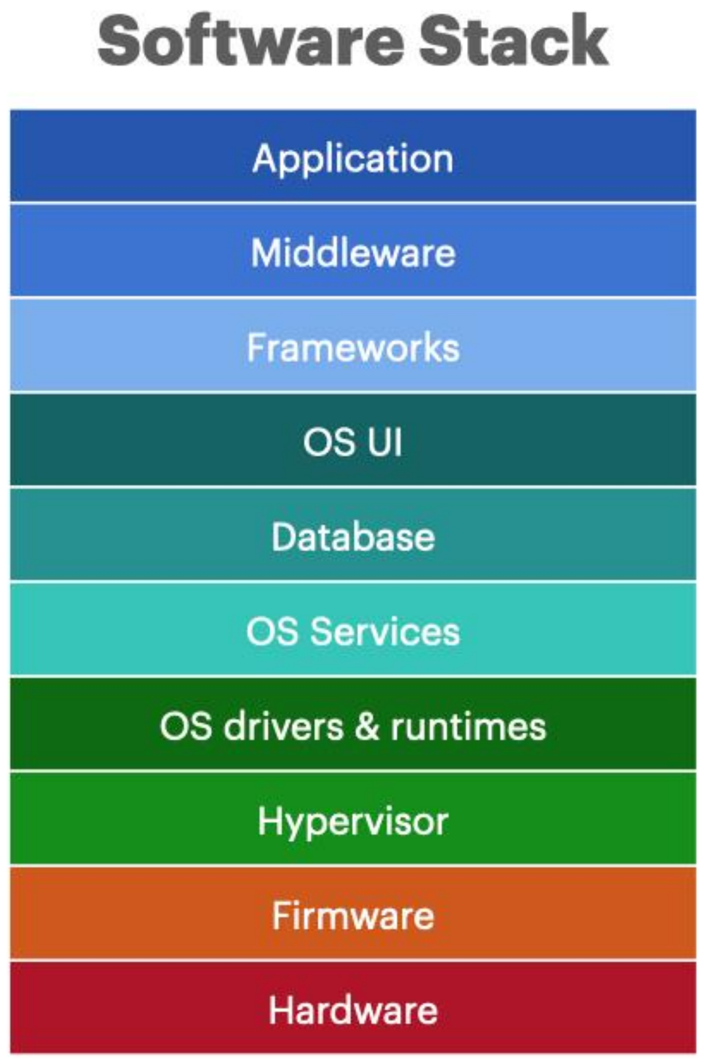
Natarajan Shankar

SRI International Computer Science Laboratory

Joint with the Project DesCert Team
(SRI, Honeywell Research, U. Washington)

The Software Stack

- The modern software stack is one of mankind's greatest engineering achievements
- With a few keystrokes, we can send email, make video calls, edit images, operate factories, control air traffic, and manage sensitive data.
- But this power comes with a price: a large attack surface where bugs can have serious consequences.
- Estimated engineering cost of software errors for the US is around 2.41T $/year.
- Cybercrime is seen as a 6T$/year problem, and growing



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

https://appvance.com/wp-content/uploads/Software-Stack.001.jpeg

What Makes Software Weird?

- Unlike other engineering artifacts, software supports greater flexibility, resiliency, and versatility in the design and maintenance of a system
- However, software can be a significant source of system failure due to bugs and security vulnerabilities - even a small design, coding error, or malicious modification can have big consequences
- Software applications tend to be *sui generis* - we lack a mature engineering discipline of principled software construction
- Attackers can relentlessly probe software for vulnerabilities and compromise security and reliability
- The resulting attacks can wreak havoc on a global scale
- To secure the software supply chain, we need to invest in design and composable assurance, and not band-aids.

A Few Celebrity Bugs

- AT&T Cascading Failure
- Intel FDIV bug
- Ariane-5 launch
- Patriot Missile bug
- Northeast blackout
- Obamacare web site
- OpenSSL RNG
- OpenSSL Heartbleed
- Therac-25
- Boeing 737 MAX-8
- Mars Climate Orbiter
- Apple Maps
- Windows Genuine Advantage

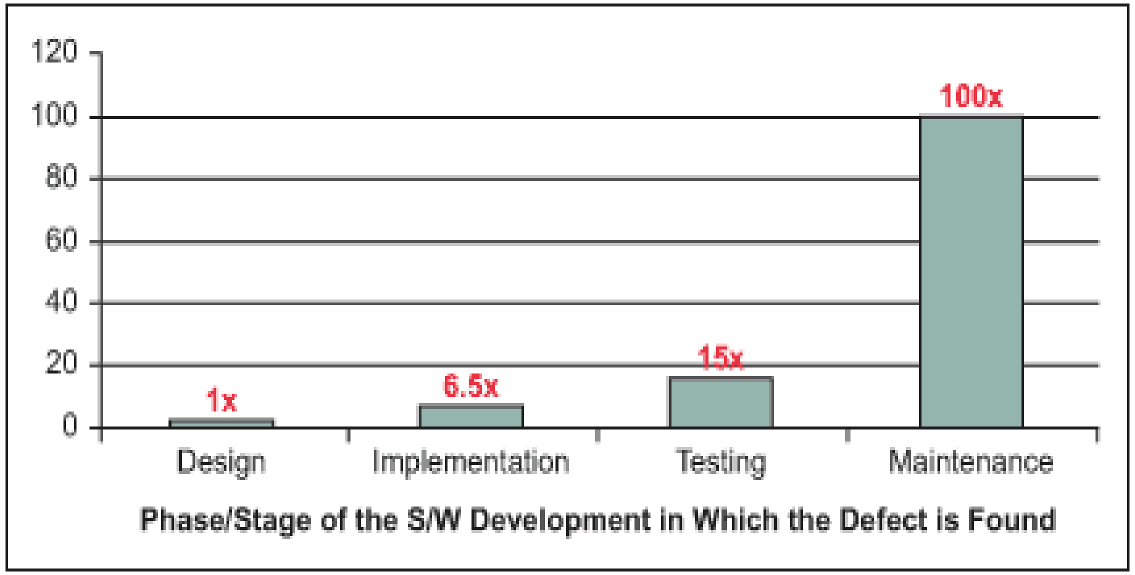
Cost Overruns

- FAA Advanced Automation System (>$3B)
- HealthCare.gov ($1.5B vs 93M)

What can go wrong?

- Software-intensive systems must possess a stringent suite of *virtues* spanning functionality, performance, reliability, robustness, resilience, persistence, security, and maintainability.
- For safety, the design must mitigate all possible hazards, conditions for potentially dangerous events (fires, crashes, societal collapse) caused by failure(s).
- A failure is a deviation from the *intended behavior* caused by errors in the functioning of one or more components, due to faults such as a bad or missing check in the software.
- Failures can arise from a combination of many sources: poor regulation, inept management, incomplete/ambiguous requirements, bad design, defective engineering, inadequate maintenance, and improper operation.

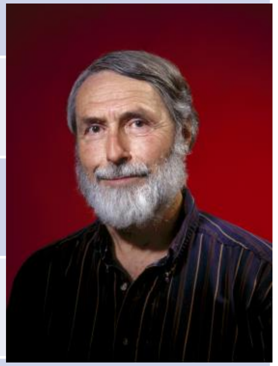
The cost of finding/fixing faults rises dramatically through the software development lifecycle.



https://www.isixsigma.com/industries/software-it/defect-prevention-reducing-costs-and-enhancing-quality/

Software-Related Risks

Channel	Instances
Hardware	Intel FDIV, Spectre/Meltdown,
Side Channel	Power, timing, 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	SQL/Format string, X.509 certificates, Heartbleed
Race/Reset condition	Therac-25, North American Blackout, AT&T crash of 1990, Mars Pathfinder
Code injection/reuse	Shell injection, Return-oriented Programming, Jump-oriented programming
Provenance/Backdoor	Athens Affair, Solar Winds
Social Engineering	Phishing, Spear Phishing, phone/in-person exploits


Peter Neumann

Software-Related Risks: The Enemy is Us

Channel	Instances
Hardware	Intel FDIV, Spectre/Meltdown,
Side Channel	Power, timing, radiation, wear-and-
Calculation	NASA Mariner, Mars Polar Lander, N
Memory/Type	Buffer Overflow, null dereference, u
Crypto	SHA-1, MD5, TLS Freak/Logjam, Ne
Input Validation	SQL/Format string, X.509 certificate
Race/Reset condition	Therac-25, North American Blackou
Code injection/reuse	Shell injection, Return-oriented Pro
Provenance/Backdoor	Athens Affair, Solar Winds
Social Engineering	Phishing, Spear Phishing, phone/in

AH, POGO, THE BEAUTY OF THE FOREST PRIMEVAL GETS ME IN THE HEART.
IT GETS ME IN THE FEET, PORKYPINE.
EARTH DAY 1971
IT IS HARD WALKIN' ON THIS STUFF.
YEP, SON, WE HAVE MET THE ENEMY AND HE IS US.

What then shall we do?

- Many vulnerabilities are consequences of original sins: conflating call and variable stacks, stack abuse, broken abstractions, weakened protections, etc. Expiating these sins must be a priority.
- Formal modeling and analysis is practical and even necessary, but not a panacea
- Software should be designed hand-in-hand with assurance artifacts that are verifiable by clients (or trusted third parties)
- Design for assurance must be based on efficient (fail-big, fail-easy) compositional arguments with low amortized certification cost
- Software designs ought to be centered around software architectures (formal models of computation & interaction) that deliver efficient arguments for isolation and composition
- Software development workflows must capture design refinements while maintaining the associated claims and evidence (the value proposition).

The RAF Nimrod XV230 Accident

- On 2 September 2006, RAF Nimrod XV230 “suffered a catastrophic mid-air fire" while flying in Helmand province, Afghanistan.
- All fourteen people aboard the plane died.
- The fire happened 90 seconds following air-to-air refuelling (AAR).
- The cause of the fire was a fuel leak around the AAR that was ignited by contact with an exposed (due to frayed/inadequate insulation) element of the cross-feed (CF) duct (1969-75) and Supplementary Conditioning Pack (SCP) duct (1979-84) that transported hot (470 deg. C) air.

What went wrong?

- The Nimrod, developed from the de Havilland Comet, had been flying since 1969 but the AAR was added by BAE first in 1982 and upgraded in 1989, and certified on the basis of a safety case developed by BAE (with QinetiQ as an independent reviewer) during 2001-2004.
- The Haddon-Cave report observed that *the cross-feed duct was placed dangerously close to a fuel tank:*

As a matter of good engineering practice, it would be extremely unusual (to put it no higher) to co-locate an exposed source of ignition with a potential source of fuel, unless it was designated a fire zone and provided with commensurate protection. Nevertheless, this is what occurred within the Nimrod.

Haddon-Cave on the Nimrod Safety Case

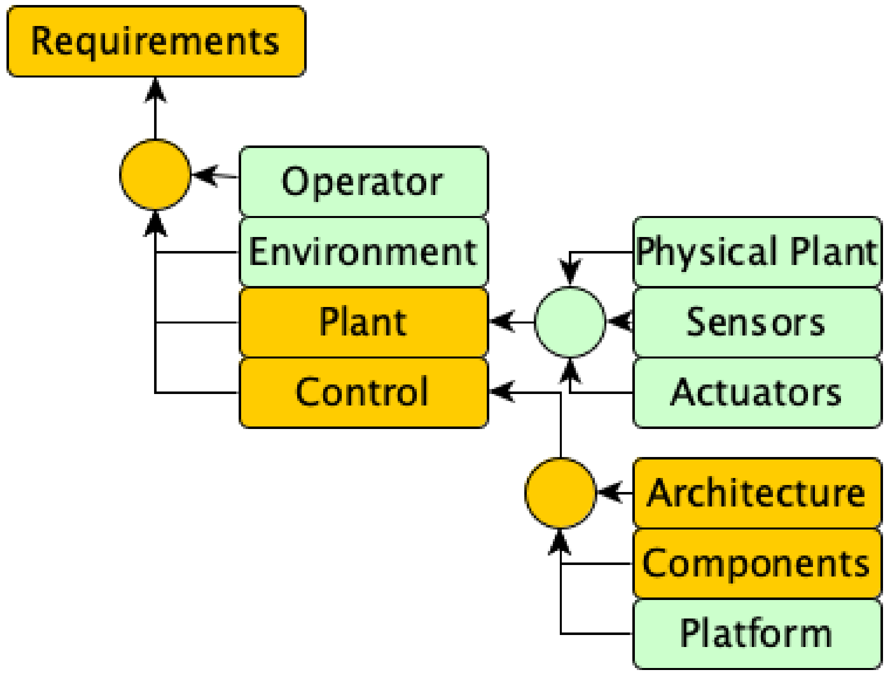
- *Unfortunately, the Nimrod Safety Case was a lamentable job from start to finish. It was riddled with errors. It missed the key dangers. Its production is a story of incompetence, complacency, and cynicism.*
- *The Nimrod Safety Case process was fatally undermined by a general malaise: a widespread assumption by those involved that the Nimrod was 'safe anyway' (because it had successfully flown for 30 years) and the task of drawing up the Safety Case became essentially a paperwork and 'tick-box' exercise.*

- *A Safety Case itself is defined as ``a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment''.*
- *The basic aims, purpose and underlying philosophy of Safety Cases were clearly defined, but there was limited practical guidance as to how, in fact, to go about constructing a Safety Case. … If the Nimrod Safety Case had been properly carried out, the loss of XV230 would have been avoided.*

Evidence-Based Assurance

Adelard describes an assurance case as ``a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a system's properties are adequately justified for a given application in a given environment.''

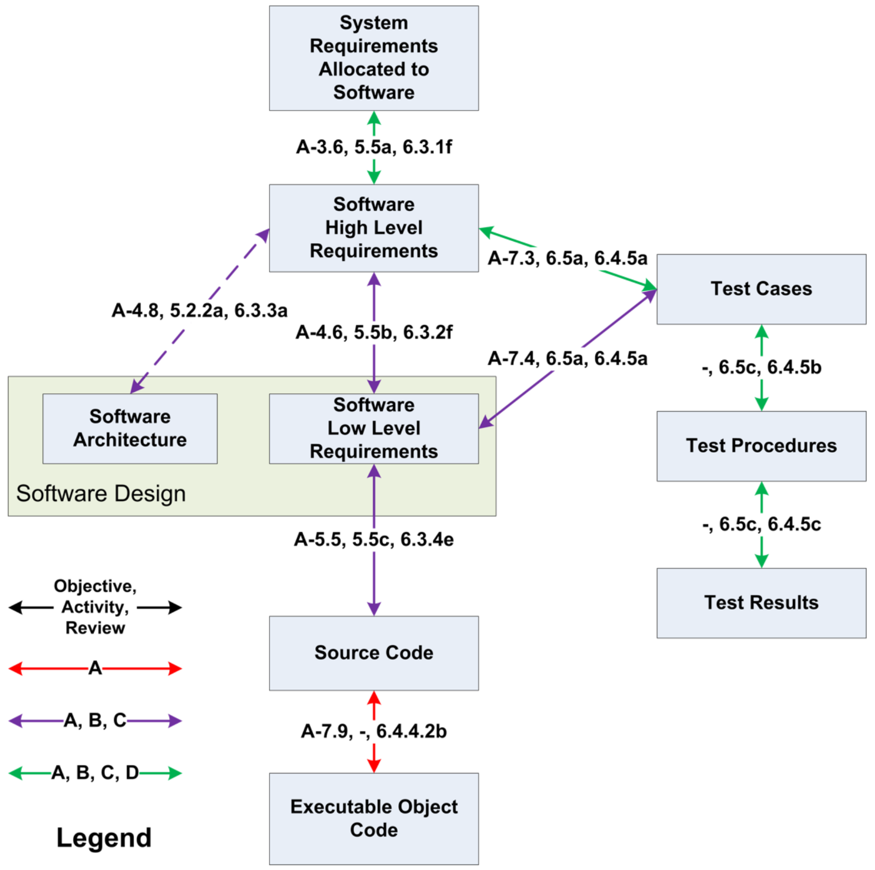
FDA Draft Guidance document Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions: … an assurance case is *a formal method for demonstrating the validity of a claim by providing a convincing argument together with supporting evidence. It is a way to structure arguments to help ensure that top-level claims are credible and supported. In an assurance case, many arguments, with their supporting evidence, may be grouped under one top-level claim. For a complex case, there may be a complex web of arguments and sub-claims.*



Gold components are verified; Green ones are assumptions/models supported by empirical evidence.

Software Assurance

- Since software is a source of catastrophic bugs and vulnerabilities, it must be certified as being fit for its intended purpose prior to deployment
- An assurance case captures the rationale for the claim that the software is fit for its intended purpose.
- There are numerous assurance case guidance/standards: IEC 62304, ISO 26262, MIL-STD-882E, SAE ARP4754/ARP4761, RTCA DO-178C, DO-326, DO-355/356.
- Counter to process-based standards, Overarching Properties (OP) takes a property-based approach focusing on intent, correctness, and innocuity.



https://en.wikipedia.org/wiki/DO-178C - /media/File:DO-178C_Traceability.png

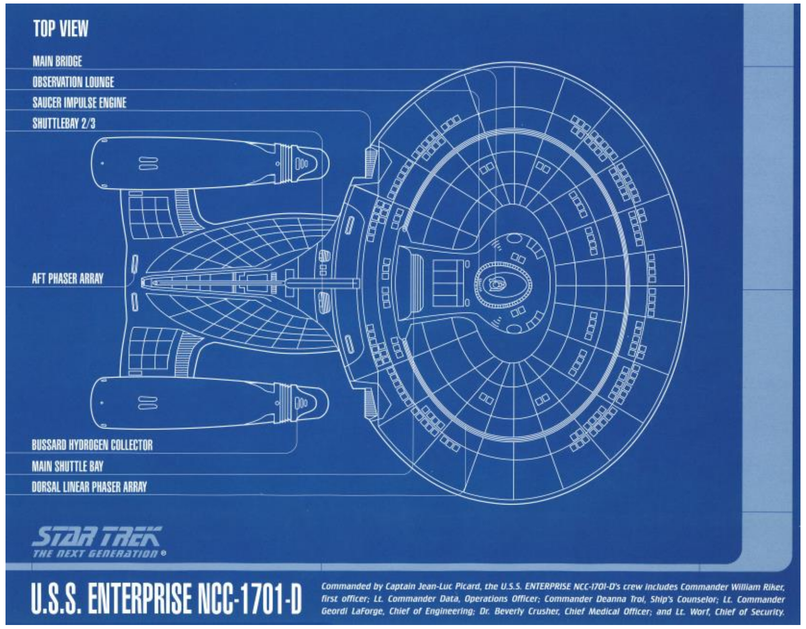
The Possibility of Perfection

- Software and hardware behavior can be modeled with mathematical precision.
- Software can, in principle, be provably engineered to perfection (modulo messy reality) given accurate specifications.
- Building accurate models, capturing requirements, and crafting specifications *are* critical challenges for software.
- Even so, the strategic deployment of lightweight and heavyweight analysis techniques can yield huge dividends.
- The main bottleneck is design, and even verified software can be poorly designed.

Formal Verification Milestones

- CLinc verified stack (1989)
- SPARK/Ada verification of avionics, medical device, air traffic control, crypto software
- NASA Langley verification of air traffic control algorithms/software (2004)
- NRL Separation Kernel (2007)
- CompCert verified compiler for subset of C (2008)
- Intel i7 processor verification (2009)
- seL4 microkernel verification (2010)
- Airbus 340 & 380 avionics software (2010)
- CakeML hardware/software stack (2014)
- Everest verified HTTPS, TLS code (2017)

On Design



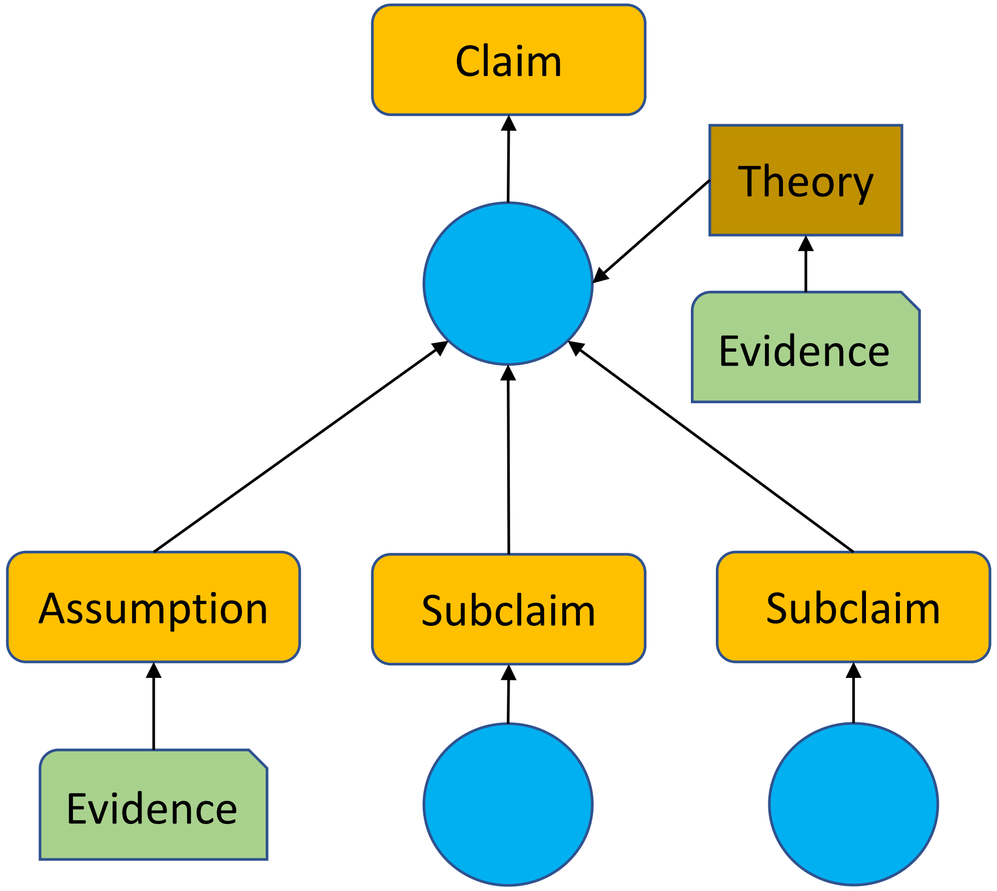
Commanded by Captain Jean-Luc Picard, the U.S.S. ENTERPRISE NCC-1701-D's crew includes Commander William Riker, first officer; Lt. Commander Data, Operations Officer; Commander Deanna Troi, Ship's Counselor; Lt. Commander Geordi LaForge, Chief of Engineering; Dr. Beverly Crusher, Chief Medical Officer; and Lt. Worf, Chief of Security.

- A design is a blueprint for the construction and operation of a system or artifact.
- The design can be decomposed into what is fixed: structure and semantics, and what is allowed to vary and how: dynamics.
  - Structure specifies the architecture (components, interfaces, and bindings) of a specific design.
  - Semantics specifies how the individual components act and interact.
  - Dynamics specify the (time-varying) variables in the systems.
- For critical systems, the end goal of a design process should be more than a blueprint
  - It should include an argument supported by evidence as to why the design meets its objectives.

https://www.universostartrek.com/USS-Enterprise-NCC-1701-D-Top-View.jpg

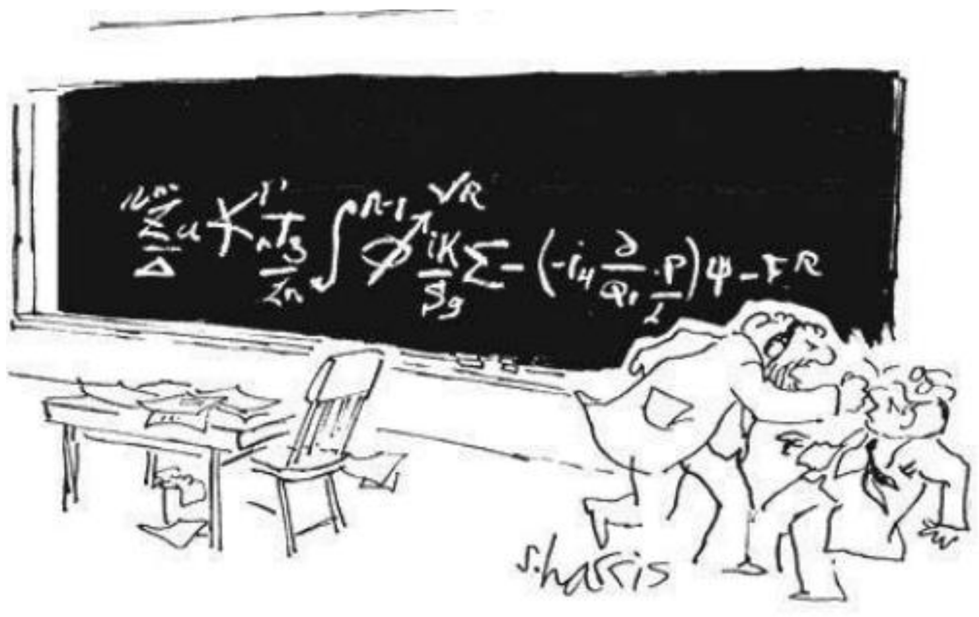
On Argument

- An argument for a system is a tree of claims, subclaims, and assumptions.
- An assurance case is a theory-supported structured argument with claims, subclaims, and assumptions backed by artifacts and evidence, that demonstrates that the software faithfully implements the intended behavior.
- The assumptions, e.g., on the environment or sensors, are supported by evidence.
- The refinement of claims into subclaims should be backed by a theory (with its own supporting evidence).



Making Arguments Efficient (for the skeptic)

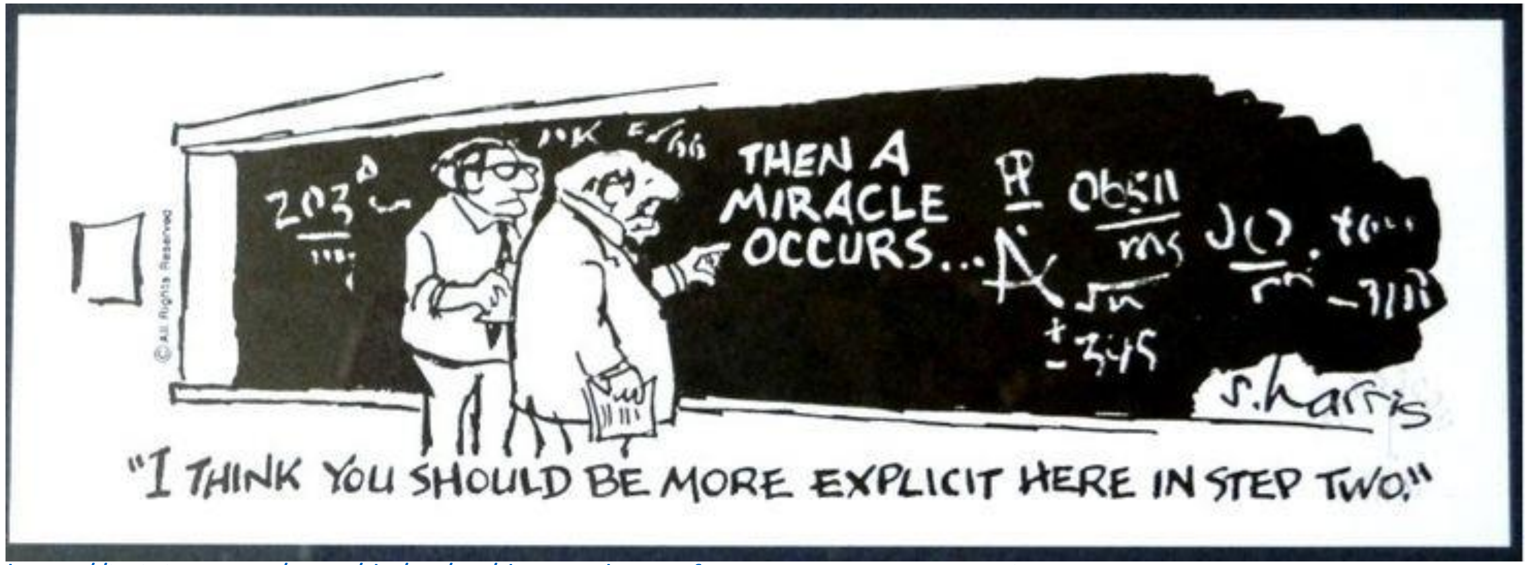
- An efficient argument is one whose flaws, if any, can be easily identified by a skeptic.
- A good design should support an efficient argument that expands the falsification space for the skeptic.
- Efficiency is measured by the amortized cost of falsification.
- Inefficient arguments are hard to falsify for a number of reasons: imprecise claims, unfalsifiable assumptions, complex technical arguments, flawed or irrelevant evidence, invalid chain of reasoning, leaps of faith, improper tracking of change.



"You want proof? I'll give you proof!"

https://pics.onsizzle.com/has-is-you-want-proof-ill-give-you-proof-6076357.png

Design for Efficient Arguments



https://i.pinimg.com/736x/d6/e7/54/d6e754d24aaef324c1595e68583ace7a.jpg

All models are wrong, but some are useful.
George E.P. Box

Efficient arguments use

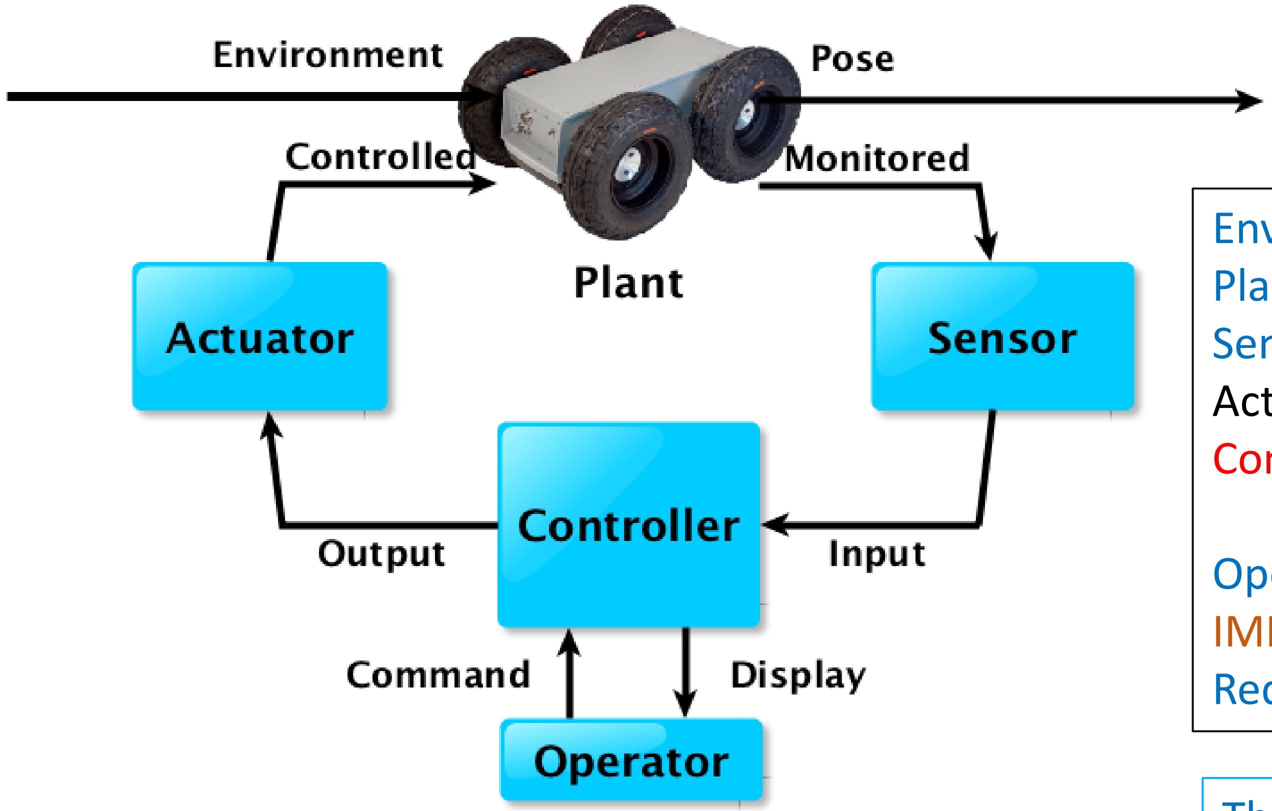
- Precise Claims
- Valid models and assumptions
- Reusable design tools/artifacts
- Architectural separation of concerns
- Rigorous chain of reasoning and evidence

- Models (plant, environment, sensor, actuator, operator, platform, fault), Architectures, Languages, and Tools are the pillars of efficient arguments
- Efficient arguments lower the amortized falsification cost through big, reusable claims that expand the falsification space.

Designing for Efficiency

- Assurance-driven workflow for continuously/incrementally capturing evidence during design
  - No gaps in the chain of evidence.
- Property-directed assurance (in contrast to process-based assurance)
  - Software failure can be directly traced to property violation.
- Architecture based on a rigorous model of computation and interaction
  - Model offers high-level guarantees that are reusable over multiple designs.
- Separation of logical and physical architecture
  - Logical architecture is reusable, easy to check that physical architecture meets logical assumptions.
- Component contract specifications
  - Behavioral properties are established from architecture and component contracts.
- Abstract component models
  - Strong semantics + easier proofs.
- Static analysis for generic properties (e.g., absence of runtime errors)
  - Amortized cost of using type system or analyzer.
- Ontic type analysis for consistent data representation and use
  - Efficient check for ontic mismatches.
- Automatic code generation from models
  - Generator is reusable and easily verified.

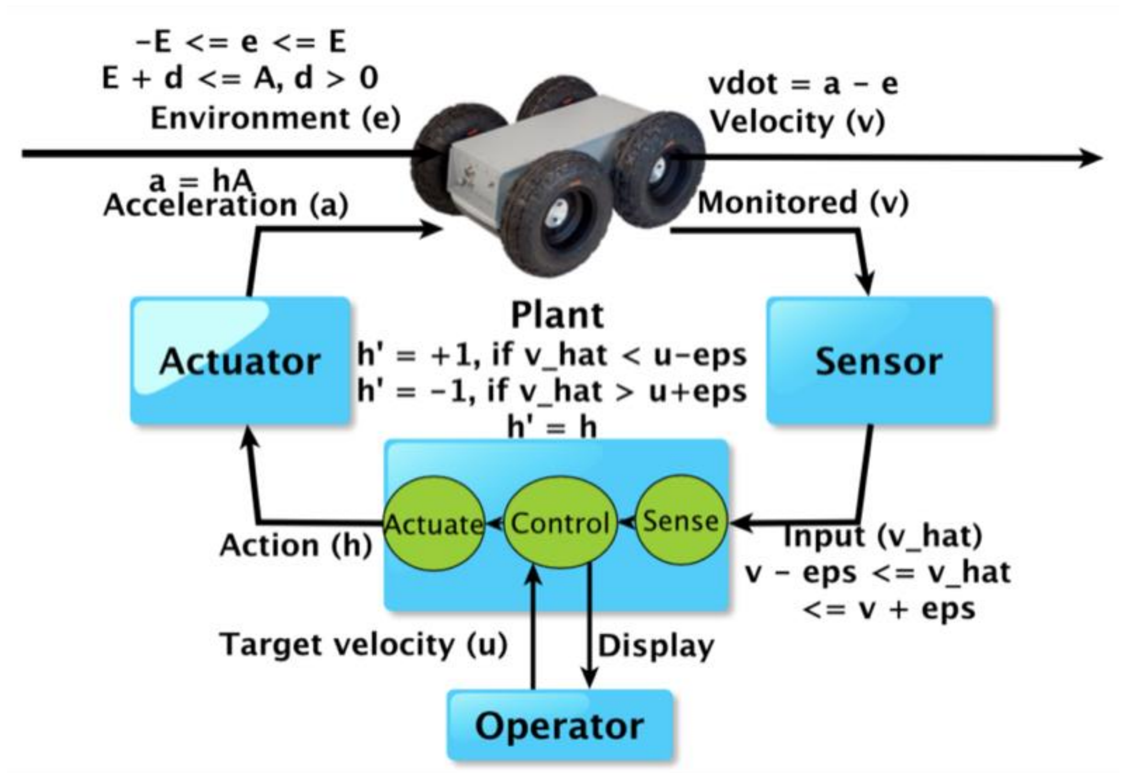
Property-Directed Assurance
The Eight-Variables Model



EnvironmentAssumption(environment) AND
PlantModel(environment, control, pose, monitor) AND
SensorAccuracy(monitor, input) AND
ActuatorResponse(output, control) AND
ControllerSpecification(input, command,
output, display) AND
OperatorModel(display, command)
IMPLIES
Requirement(command, environment, pose, display)

The Controller Specification can be seen as the High-Level Requirement (HLR) on the software.

8-Variables: An Example

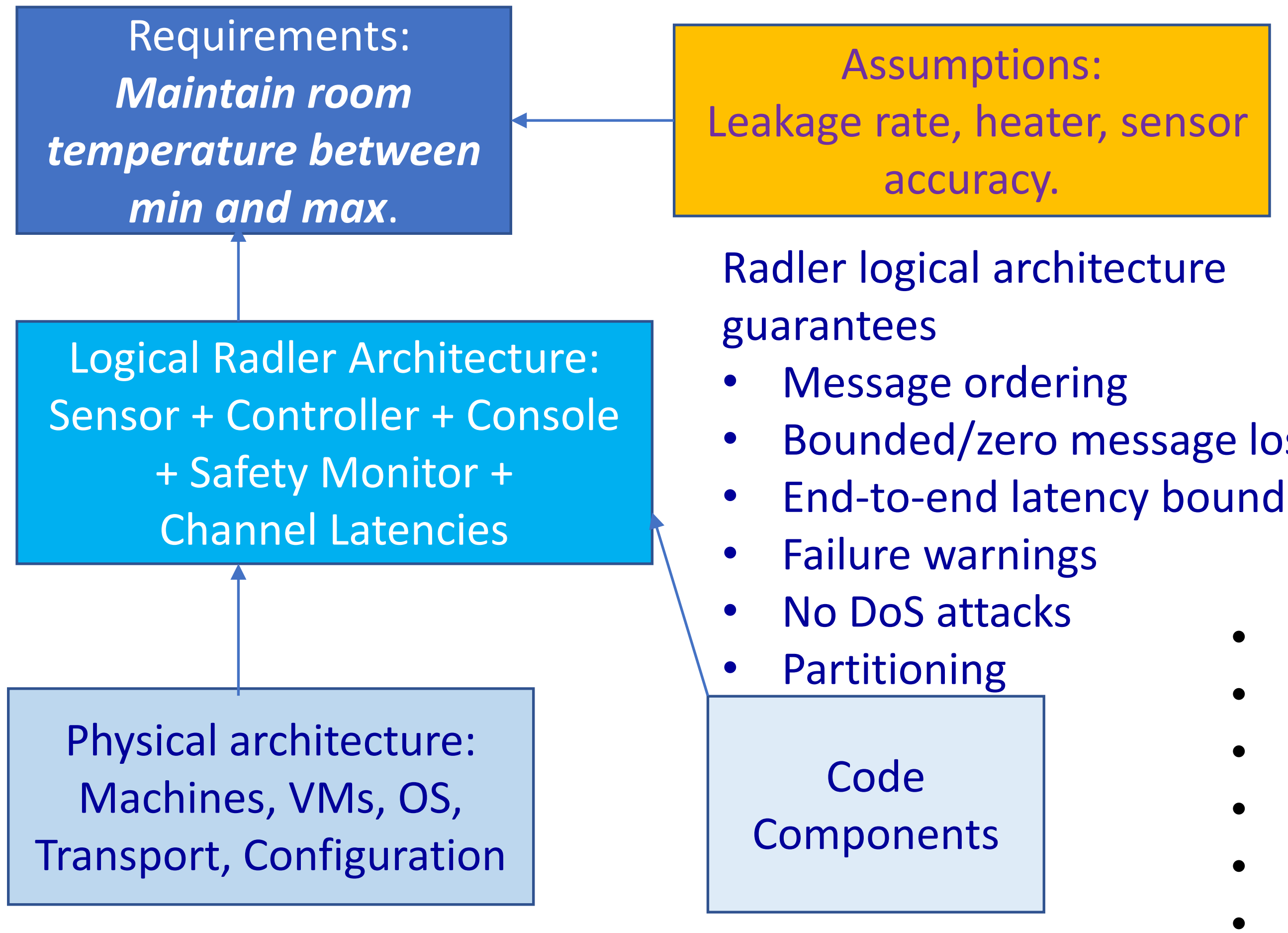


- The Plant consists of the vehicle that is trying to maintain a speed *v* and the Environment *e* is the grade of the road.

- The goal requirement is to maintain the vehicle velocity *v* within some bound of the target velocity *u*.

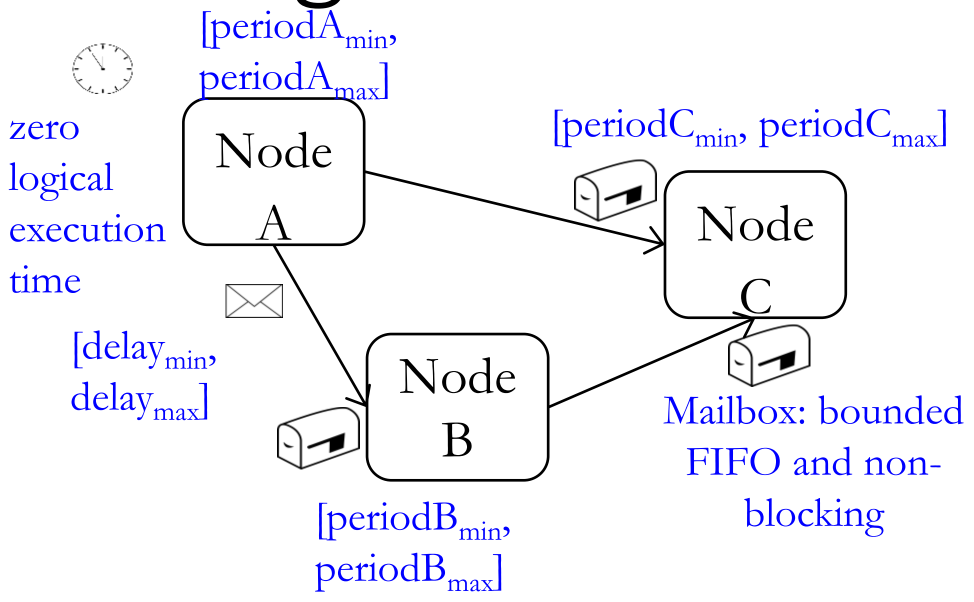
There are other ways to decompose the interaction:
- Operator as part of the World
- Actuator and Sensor as part of the Plant
- Actuator and Sensor as part of the Controller

Radler Architecture for Efficient Arguments


Requirements:
Maintain room temperature between min and max.
Assumptions:
Leakage rate, heater, sensor accuracy.
Logical Radler Architecture:
Sensor + Controller + Console + Safety Monitor + Channel Latencies
Physical architecture:
Machines, VMs, OS, Transport, Configuration
Code Components

Radler logical architecture guarantees

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


[periodA$_{min}$, periodA$_{max}$]
zero logical execution time
Node A
[periodC$_{min}$, periodC$_{max}$]
Node C
[delay$_{min}$, delay$_{max}$]
Node B
[periodB$_{min}$, periodB$_{max}$]
Mailbox: bounded FIFO and non-blocking

- Assumptions + Architecture => Requirements
- Architecture = Logical Arch. + Physical Arch.
- Logical Architecture = Nodes + Channels + Timing
- Node = Step function contract + Mailboxes + Period
- Physical Architecture => Logical Arch. Assumptions
- Code => Step function contract + WCET bounds

Radler build process ensures the architectural integrity of executables.

https://github.com/SRI-CSL/radler/

DesCert Approach: Ontology for Safety and Security Assurance

1. Without a rigorous ontology, the claims and chain of evidence in an assurance case can be ambiguous and misleading.
2. Honeywell's **CLEAR** language is used to capture both requirements and *ontology elements and their relationships to create corresponding evidence sets and reasoning for direct construction of* ***Assurance Claims.***

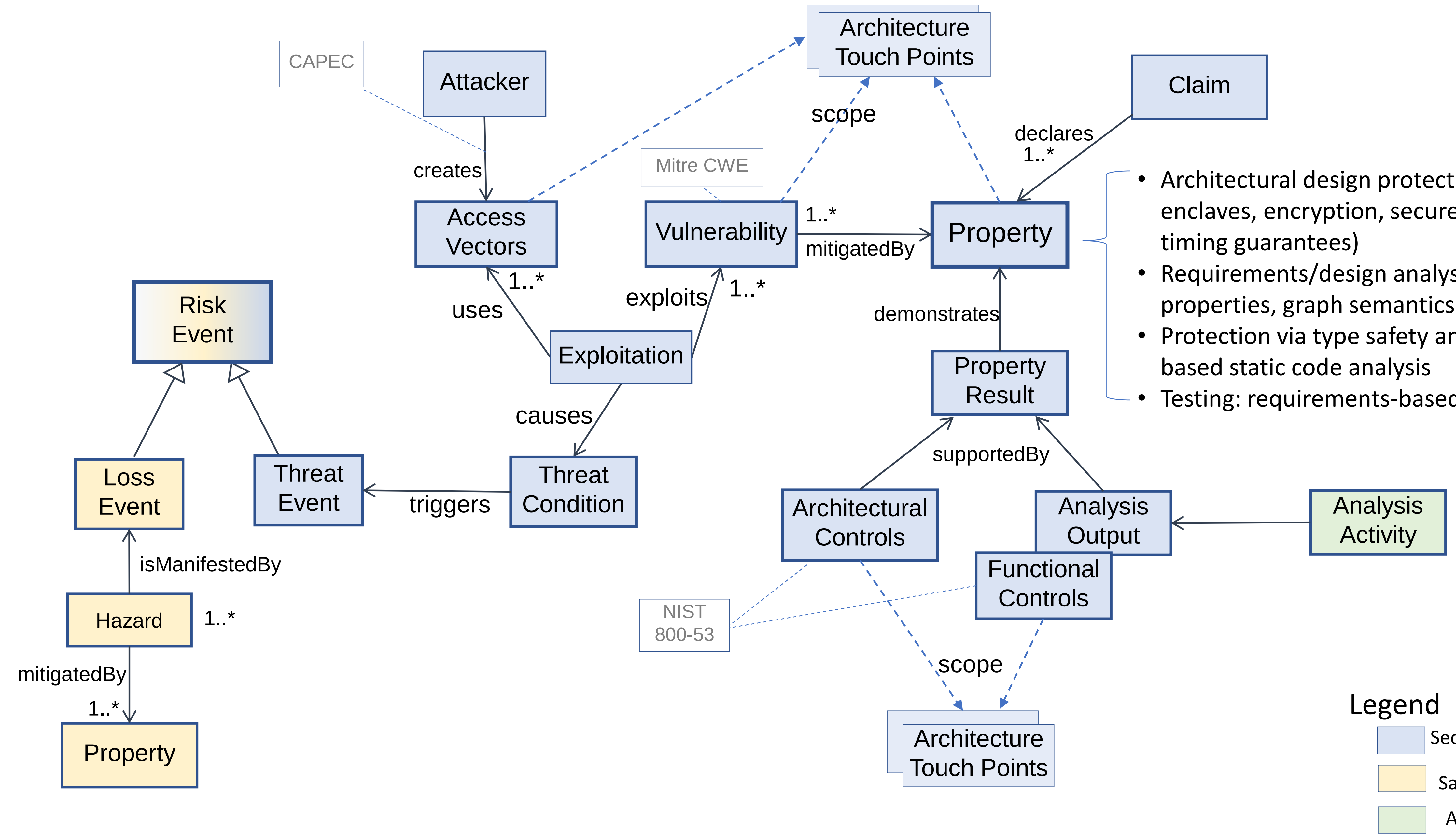
E.g.: *Threat A1's exploiting of vulnerabilities (v1, v2) is blocked due to controls (c1, c2, c3) present (with associated property results) in the architecture (radl1) in software components (s1, s2).*

1. 3. Ontological categories for *modeling* of:
   1. Threats[1]: Weak access control, weak input validation, race conditions, timing attacks, phishing, privilege escalation
   2. Vulnerabilities[2]: Null dereference, SQL injection, Buffer overflow
   3. Controls[3]: Physical security, Access control, Monitoring, Reporting, Authentication
   4. Risk/loss events[4]: Loss of Confidentiality, Integrity, Availability, Safety.
   5. Architecture/Touch (entry) Points: Sensors, Actuators, Communication channels, Files, Hardware

Threat	Entry Point	Risk	Mitigation
Malicious Code	Build Process	Failure, Unauthorized Access	Radler Certified Build/Attestation
Malicious Inside Actor	Untrusted Code	DoS, Failure, exfiltration/infiltration	Radler Security Enclaves
Loss of Information Integrity	Tampering	Failure	Radler Security Enclaves
Loss of Comm. integrity	Communication layer	Infiltration, Exfiltration, Jamming	Radler/SROS2 protections
Access Control Violation	Architecture	Failure, Unauthorized Access	Radler config., Ontic analysis
Bad/Unexpected Input	Unchecked input ports	Failure/Remote Code Execution	Ontic Type Analysis

CLEAR – Constrained Language Enhanced Approach to Requirements

DesCert Evidence Ontology for Integrated Security/Safety Analysis



- Architectural design protection (partitioning, enclaves, encryption, secure DDS, resource/ timing guarantees)
- Requirements/design analysis, model check properties, graph semantics analysis
- Protection via type safety and ontic-type-based static code analysis
- Testing: requirements-based, threat-based

E.g., Checker Framework execution

Legend

🟦 Security related (Phase 2)

🟨 Safety related (Phase 1)

🟩 Activity

Ontic Type Analysis

- Basic types in programming language (such as `int, struct, array`) abstract from the representation of the data
- They are insensitive to the intended use of the data, e.g., an authenticated user ID, a private encryption key, the vertical acceleration of a vehicle in m/sec^2, an IP address, a URL, or an SQL query.

```
char input[30];
int response;
scanf("%s", input);
sqlstmt = "select * from employees where id = " + input + ";";
response = sqlite3_exec(db, sqlstmt, ...);
```

- Ontic type analysis (see Checker Framework from U.Washington) checks for the proper usage of data in terms of units/dimensions, freshness, nullity, mutability, taint, authentication, privacy, format validity, and provenance.
- Ontic typing can be viewed as information flow analysis on ghost data.

Generating Safe Code From Models

- SRI's Prototype Verification System (PVS) is an interactive proof assistant built and maintained over the last thirty years
- Almost all of the specification language is executable as a safe functional language
- Executable PVS specifications can be mapped to Common Lisp , C, Rust, and oCaml.
- The generated C is safe (free of uninitialized variables, null dereferences, out-of-bounds accesses, division-by-zero).
- It is also comparable in efficiency to hand-crafted code due to the use of reference counting and in-place updates.

```
hsummation: THEORY
 BEGIN

  i, m, n: VAR nat
  f: VAR [nat -> nat]

  hsum(f)(n): RECURSIVE nat =
    (IF n = 0 THEN 0 ELSE f(n - 1) + hsum(f)(n - 1)
ENDIF)
     MEASURE n

  id(n): nat = n

  hsum_id: LEMMA hsum(id)(n + 1) = (n * (n + 1)) / 2

  square(n): nat = n * n

  sum_of_squares: LEMMA
     6 * hsum(square)(n + 1) = n * (n + 1) * (2 * n +
1)

  cube(n): nat = n * n * n

  sum_of_cubes: LEMMA
     4 * hsum(cube)(n + 1) = n * n * (n + 1) * (n +
1)

  quart(n): nat = square(square(n))

  sum_of_quarts: LEMMA
    hsum(quart)(n + 1) =
     ((6 * (n ^ 5)) + (15 * (n ^ 4)) + (10 * (n ^ 3))
- n) / 30
```

Key Derivation in PVS: HMAC

```
function hmac is
  input:
    key:        Bytes    // Array of bytes
    message:    Bytes    // Array of bytes to be hashed
    hash:       Function // The hash function to use (e.g. SHA-1)
    blockSize:  Integer  // The block size of the hash function
                         //(e.g. 64 bytes for SHA-1)
    outputSize: Integer  // The output size of the hash function
                         //(e.g. 20 bytes for SHA-1)

  // Keys longer than blockSize are shortened by hashing them
  if (length(key) > blockSize) then
    key ← hash(key) // key is outputSize bytes long

  // Keys shorter than blockSize are padded to blockSize by padding
  //with zeros on the right
  if (length(key) < blockSize) then
    key ← Pad(key, blockSize) // Pad key with zeros to make it
                              // blockSize  bytes long
  o_key_pad ← key xor [0x5c * blockSize]   // Outer padded key
  i_key_pad ← key xor [0x36 * blockSize]   // Inner padded key
  return hash(o_key_pad ∥ hash(i_key_pad ∥ message))
```

From https://en.wikipedia.org/wiki/HMAC

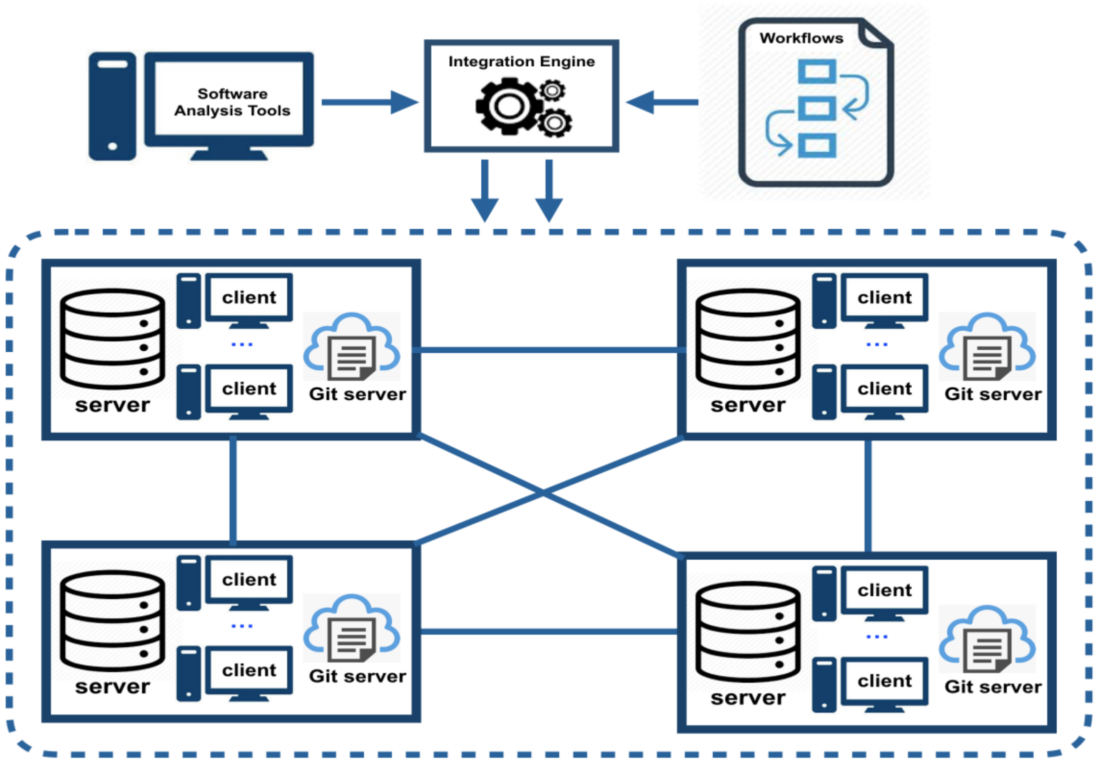
```
hmac(blockSize: uint8,
       key : bytestring,
       (message : bytestring | message`length + blockSize < byte
       outputSize: upto(blockSize),
       hash: [bytestring->lbytes(outputSize)]): lbytes(outputSiz
   = LET newkey = IF length(key) > blockSize THEN hash(key) ELSE key
         newerkey: lbytes(blockSize)
           = IF length(newkey) < blockSize
               THEN padright(blockSize)(newkey)
                       ELSE newkey
            ENDIF,
         oKeyPad = lbytesXOR(blockSize)(newerkey, nbytes(0x5c
         iKeyPad = lbytesXOR(blockSize)(newerkey, nbytes(0x36
     IN hash(oKeyPad ++ hash(iKeyPad ++ message))

  hmac256((blockSize: uint8 | 32 <= blockSize),
            key : bytestring,
     (message : bytestring |
               message`length + blockSize < bytestring_
      : lbytes(32)
  = hmac(blockSize, key, message, 32, sha256message)
```

- HMAC is a higher-order operation with complex type dependencies (not specified in the pseudocode)
- These dependencies are accurately captured in PVS
- C code generation is bit-accurate

Evidential Tool Bus (ETB2)[SRI/fortiss]

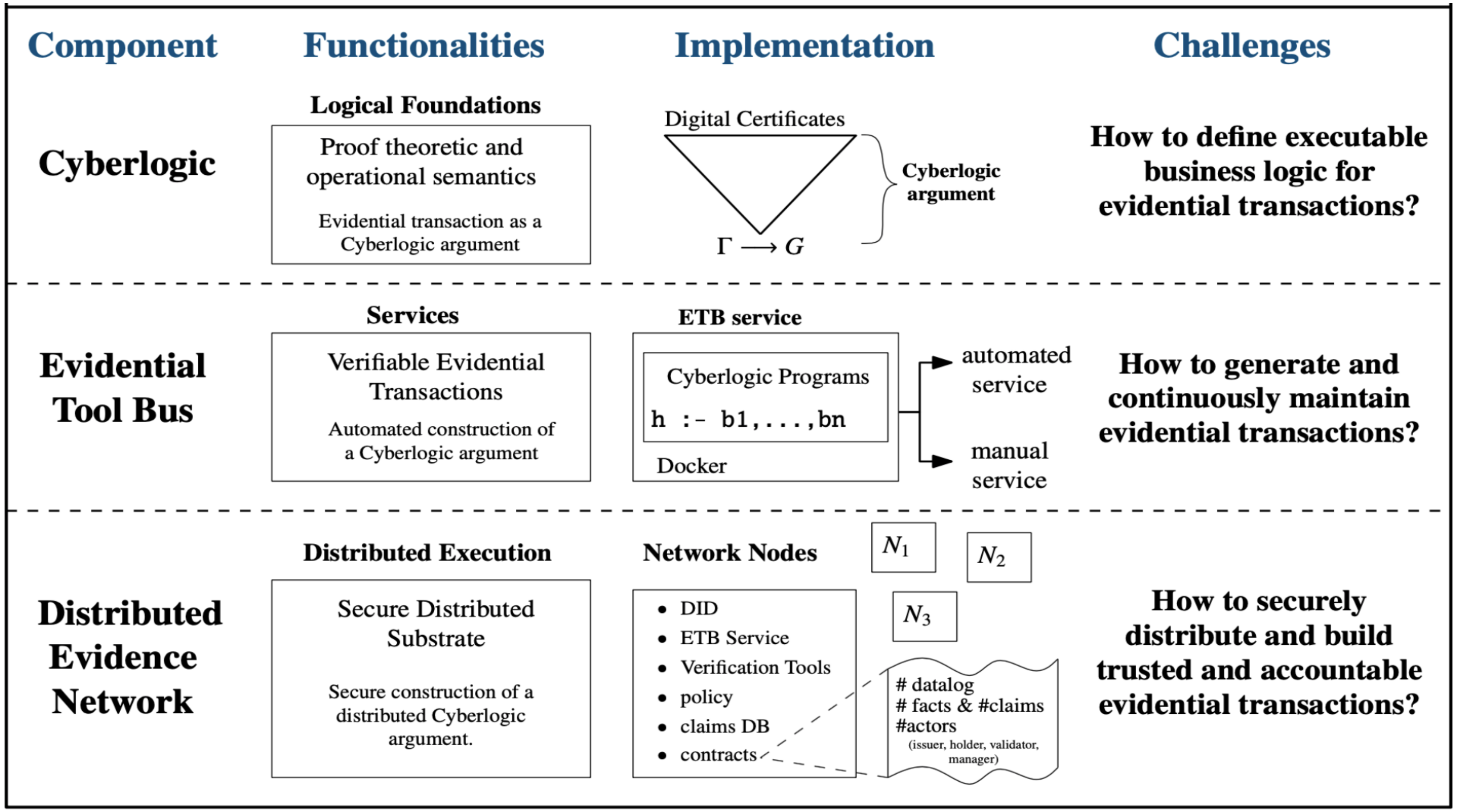
- The Evidential Tool Bus (ETB) is a distributed tool integration framework for constructing and maintaining claims supported by arguments based on evidence generated by static analyzers, dynamic analyzers, satisfiability solvers, model checkers, and theorem provers.
- Key ideas are:
  - Datalog as a metalanguage
  - Denotational and operational semantics
  - Interpreted predicates for tool invocation, and uninterpreted predicates for scripts
  - Datalog inference trees as proofs
  - Git as a medium for file identity and version control
  - Cyberlogic, a logic of attestations, to authenticate the claims and authorize the services

https://github.com/SRI-CSL/ETB2



```
allsat(F, Answers) :- sat(F, M), negateModel(F, M, NewF),
                      allsat(NewF, T), cons(M, T, Answers).
allsat(F, Answers) :- unsat(F), assign("nil",Answers).
sat(F, M) :- yices(F, S, M), equals(S, "sat").
unsat(F) :- yices(F, S, M), equals(S, "unsat").
```

Evidential Transactions on ETB



Component	Functionalities	Implementation	Challenges
Cyberlogic	**Logical Foundations** Proof theoretic and operational semantics Evidential transaction as a Cyberlogic argument	Digital Certificates $\Gamma \longrightarrow G$ **Cyberlogic argument**	**How to define executable business logic for evidential transactions?**
Evidential Tool Bus	**Services** Verifiable Evidential Transactions Automated construction of a Cyberlogic argument	**ETB service** Cyberlogic Programs `h :- b1,...,bn` Docker automated service manual service	**How to generate and continuously maintain evidential transactions?**
Distributed Evidence Network	**Distributed Execution** Secure Distributed Substrate Secure construction of a distributed Cyberlogic argument.	**Network Nodes** N_1 N_2 N_3 • DID • ETB Service • Verification Tools • policy • claims DB • contracts # datalog # facts & #claims #actors (issuer, holder, validator, manager)	**How to securely distribute and build trusted and accountable evidential transactions?**

Summing up

- Design is the key to building software that is demonstrably safe, reliable, and secure software.
  - Open Problem: Shift the locus of software design from code to models.
- Designs based on efficient arguments makes certification easier and more reusable.
  - Open Problem: Quantify efficiency by assessing relative certification cost, confidence, and amortization gain of different design choices
- Concrete ways of achieving efficiency include formal architectures, certified build systems, well-defined property/evidence ontologies, strong typing (including ontic types), code generators, and assurance-directed workflows.
  - Open Problem: Create a fully formal pipeline for continuous development/assurance.

Securing the Software Universe

- Software processes information: bank accounts, grades, medical records, books, videos, power grid controls, avionics, and medical devices
- Code is a poor representation of design: untrusted code should not be the input, trusted code should be the output
- Shotgun composition of code without a well-defined formal architecture will fail
- So,
  - Take information seriously: annotate the artifacts with ontic type information
  - Take requirements seriously: many major flaws are traceable to poor requirements
  - Take architecture seriously: the keystone of an efficient argument
  - Take assurance seriously: composable evidence should be the coin of the realm
  - Take the assurance ontology seriously: it binds the claims to the evidence
  - Take inline and independent runtime monitoring seriously: track integrity of assumptions
  - Re-engineer the platforms to root out the sins of our ancestors
  - Build workflows that create and maintain evidence as part of the design flow
  - Integrate attestation into the evidence as a foundation for trust

A Software Proof of Virtues (SPOV)

- Software is a core mediator of our perception of truth
- Software failures and cyber-attacks weaken trust
- The current strategy of applying larger and larger band-aids is only fueling an arms race
- We have the tools and insights to build the infrastructure of trust in software from the ground up:
  - Software development lifecycle workflows that continuously maintain both process and outcome-based assurance evidence
  - Tools and models that support designs annotated with traceable ontic information that are founded on efficient arguments
  - Verified platforms and services whose integrity is certified by audit logs and audits
  - Composable assurance cases validating intent, correctness, and innocuity