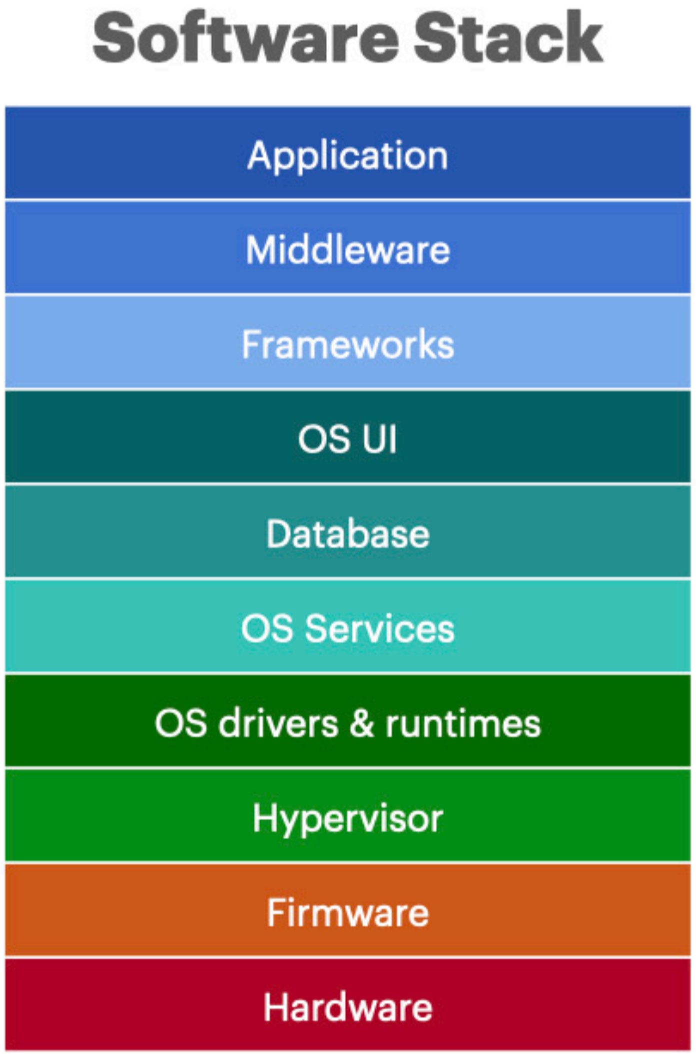
Composing High-Assurance Software with the Evidential Tool Bus

Natarajan Shankar

SRI International Computer Science Laboratory

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 2.1T $/year.
- Cybercrime is seen as a 6T$/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

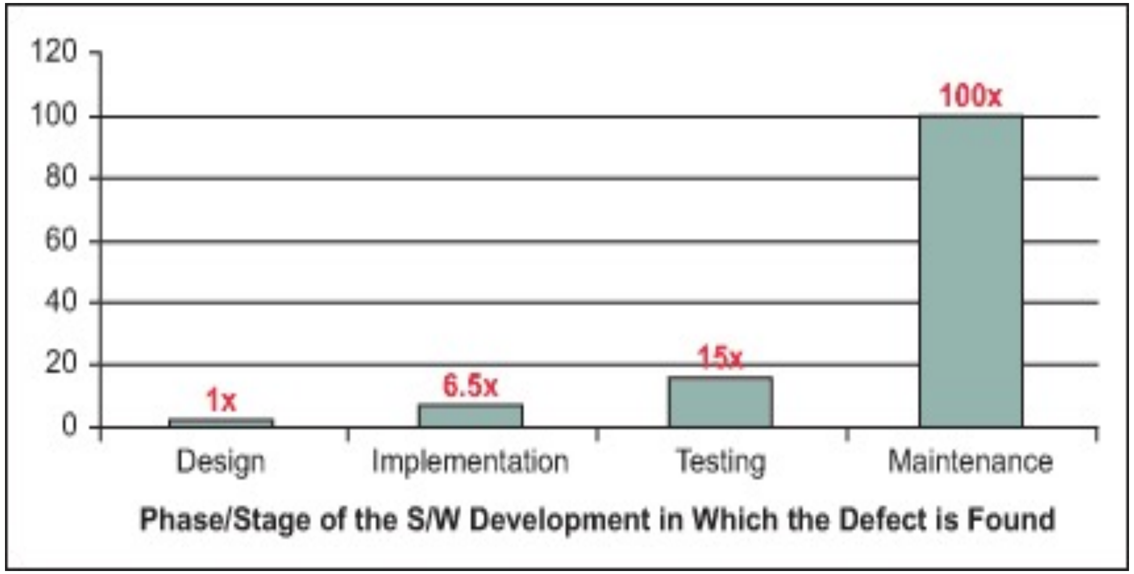
What Makes Software Weird?

- 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.

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, potentially dangerous events caused by a failure.
- 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, bad design, defective engineering, inadequate maintenance, and improper operation.

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



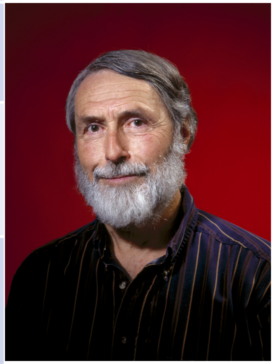
https://www.isixsigma.com/industries/software-it/defect-prevention-reducing-costs-and-enhancing-quality/

Software-Related Risks: The Enemy is Us

Channel	Instances
Hardware	Intel FDIV, Spectre/Meltdown,
Side Channel	Power, timing, radiation, wear-and-tear (Row Hammer)
Calculation	NASA Mariner, Mars Polar Lander, 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

Software-Related Risks: The Enemy is Us

Channel	Instances
Hardware	Intel FDIV, Spectre/Meltdown,
Side Channel	Power, timing, radiation, wear-and-tear (Row Hammer)
Calculation	NASA Mariner, Mars Polar Lander, 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



What then shall we do?

- Many vulnerabilities are consequences of original sins:
  - conflating call and parameter/variable stacks: data and control should only interact through code
  - stack abuse: allocating non-scalar data (arrays, structs) on the variable stack
  - broken abstractions: program access to privileged data
  - weakened protections, and many more.
- 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 cost
- Software designs ought to be centered around software architectures (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 Possibility of Perfection

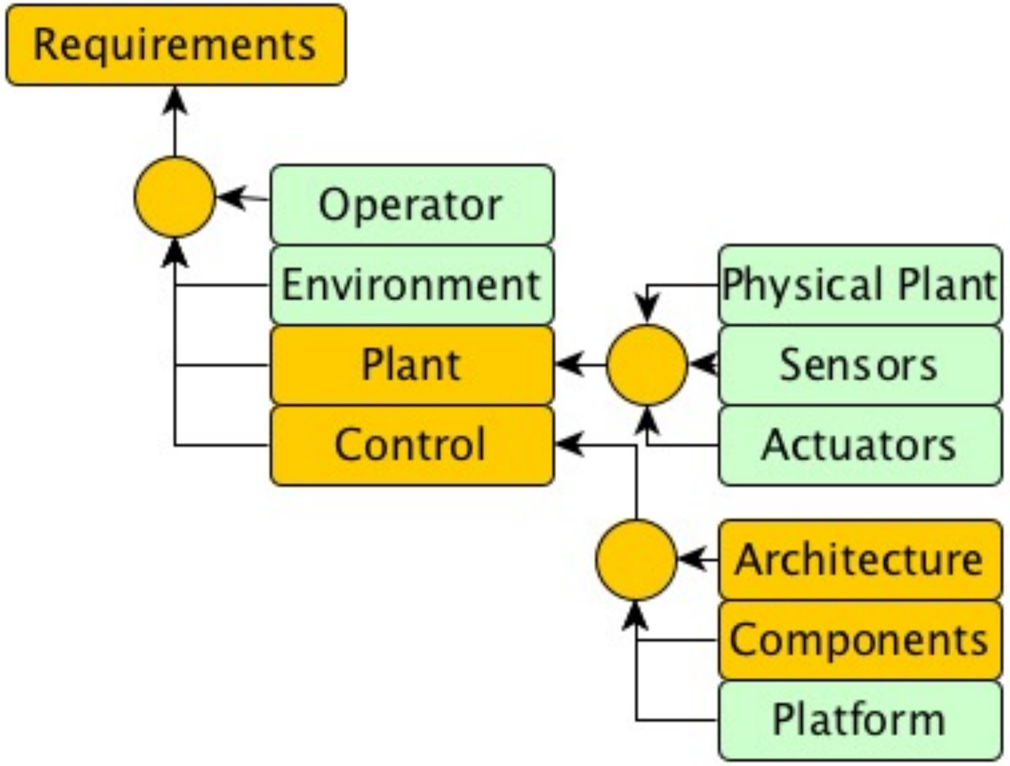
- Software and hardware behavior can be modeled with mathematical precision.
- Software can, in principle, be engineered to perfection (modulo messy reality) given accurate specifications (which is easier said than done).
- Even if perfection were only partially attainable, the strategic deployment of lightweight and heavyweight analysis techniques can yield huge dividends.

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)
- 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)

Evidence-Based Assurance

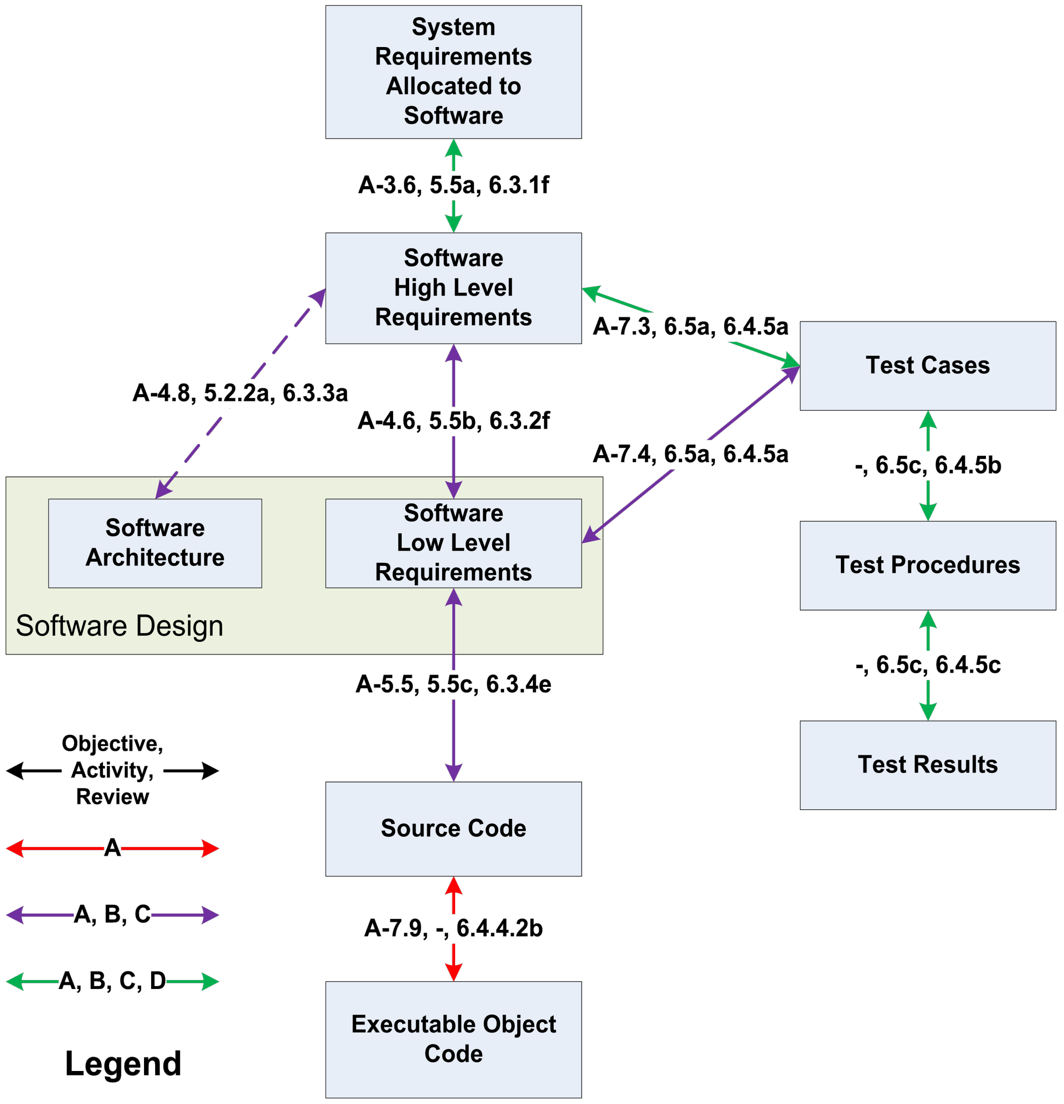
FDA Draft Guidance document Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions: … an assurance case is *a formal method for demonstrating the validity of a claim by providing a convincing argument together with supporting evidence. It is a way to structure arguments 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.

Assurance Guidelines

- Multiple standards: ISO 26262, MIL-STD-882E, SAE ARP4754/4761, DO-178C
- RTCA DO-178C guidance specifies four levels of assurance: A (catastrophic), B (hazardous), C (major), D (minor)
- *Traceability* establishes a bidirectional correspondence across levels
- Assurance case must (partially) comply with 71 objectives
- Overarching Properties (OAP) is outcome-based
    - Intent: What should the software do?
    - Correctness: Does the software satisfy the intent?
    - Innocuity: Does the extraneous functionality impact correctness?



https://en.wikipedia.org/wiki/DO-178C

Designing with Efficient Arguments

- 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 refueling (AAR).
- 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.*

- An efficient argument, one whose flaws, if any, are easily identified, would support the claim that *fuel and ignition should not interact outside the combustion chamber.*
- For assurance-driven development, a design must reflect the goal of an efficient assurance argument: verifiable requirements, operational testing theory, formal architecture, property-preserving model transformations, code generation, strong static analysis, precise/inclusive fault/threat models, and trusted automation.

A Simple Efficient Assurance Argument

Requirements:
***Maintain room temperature between min and max*.**

Assumptions:
Leakage rate, heater, sensor accuracy.

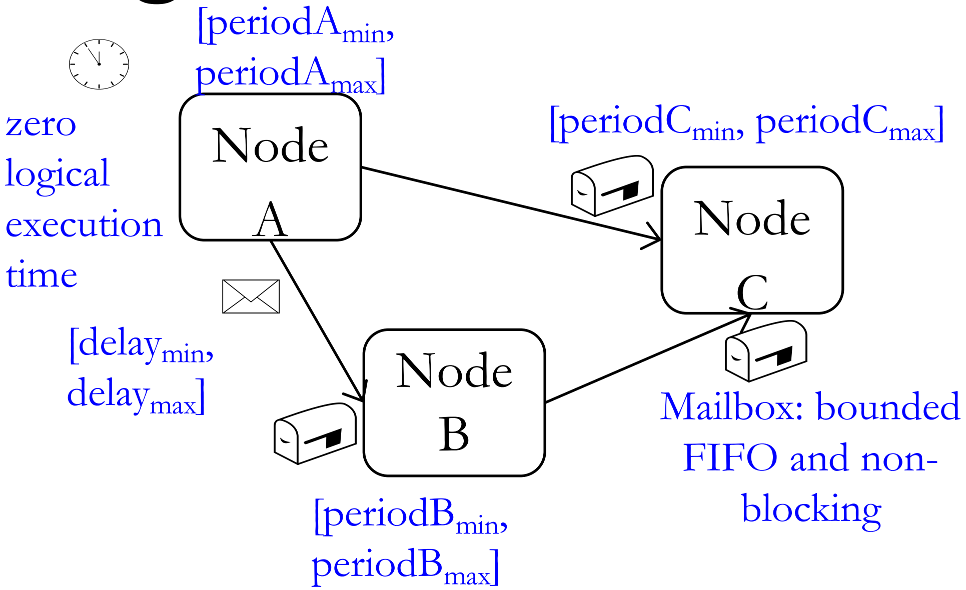
Logical Radler Architecture:
Sensor + Controller + Console + Safety Monitor
Channel Latencies

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

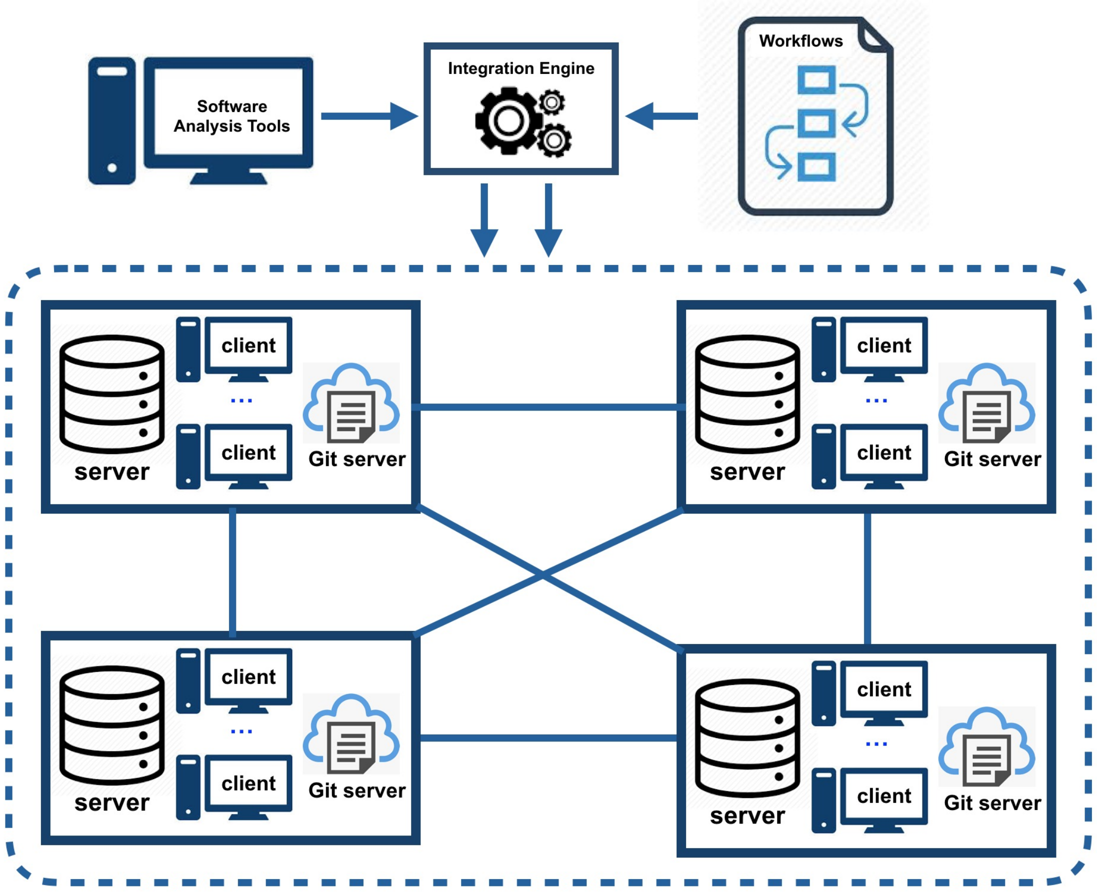


- Assumptions + Architecture => Requirements
- Architecture = Nodes + Channels + Timing
- Nodes = Step function contracts
- Physical Architecture => Architecture
- Code => Step function contracts + WCET bounds

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

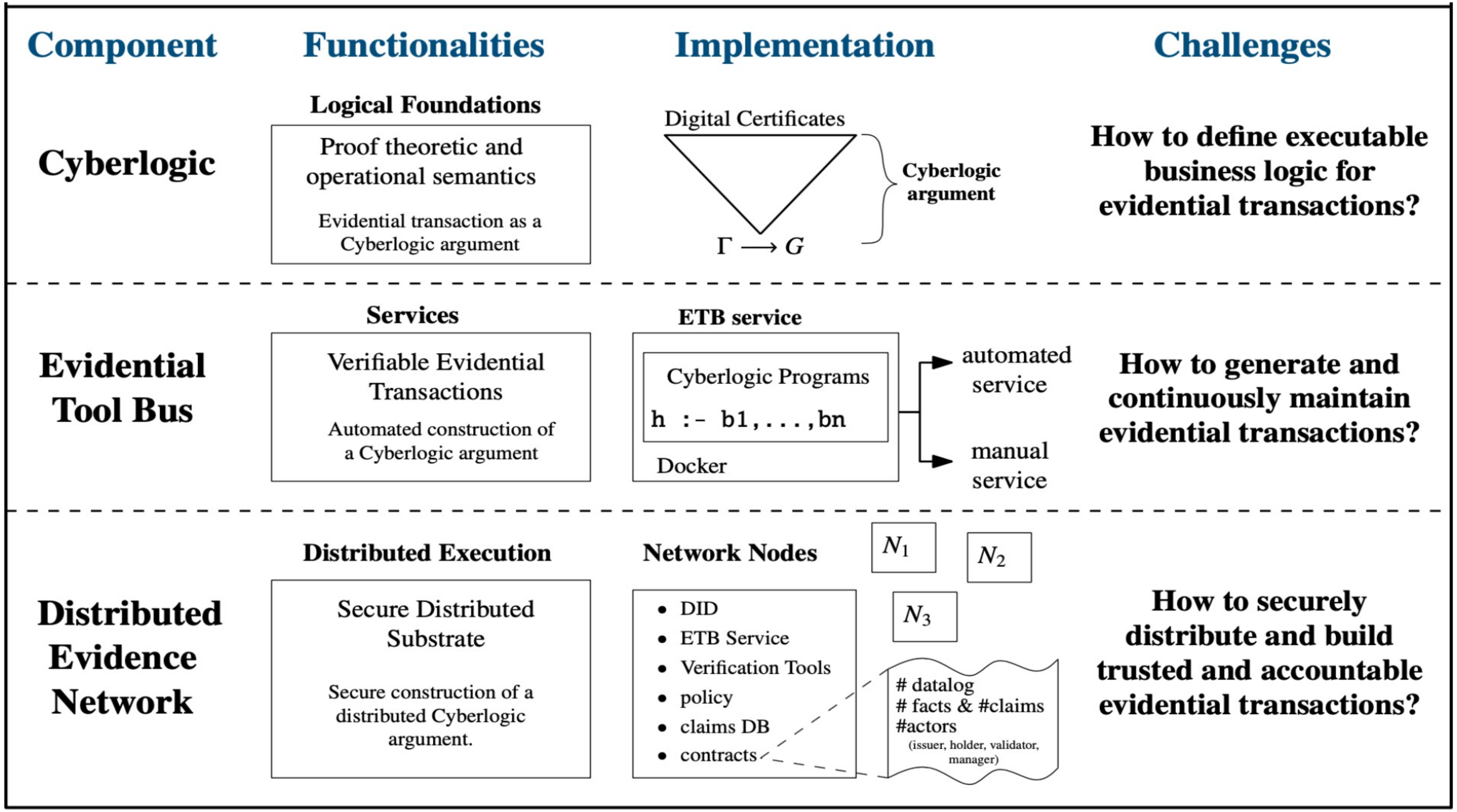
Evidential Tool Bus (ETB2)[SRI/fortiss]

- 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



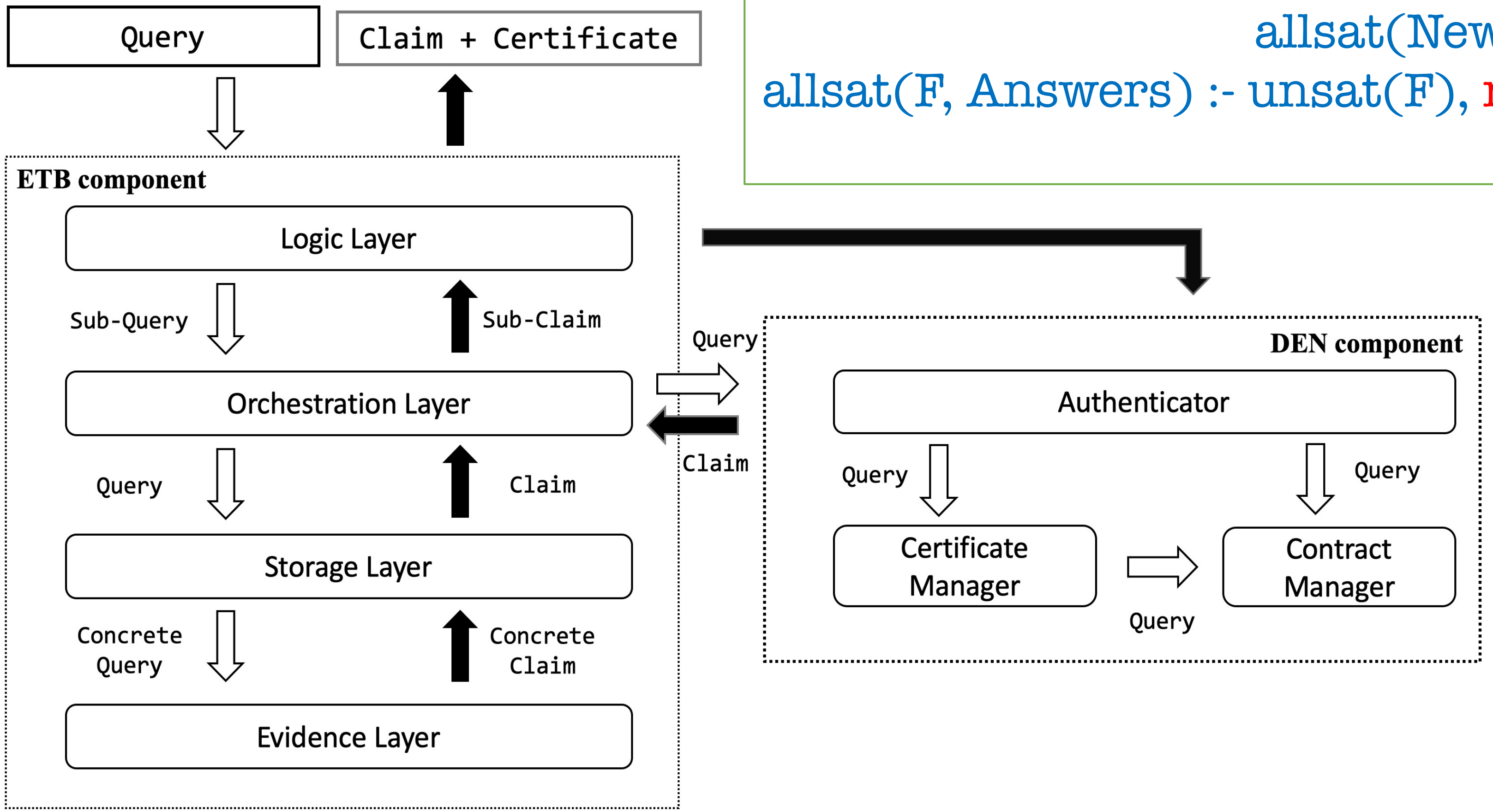
https://github.com/SRI-CSL/ETB2

Evidential Transactions on ETB



ETB Layers

```
sat(F, M) :- yices(F, S, M), equal(S, 'sat').
unsat(F) :- yices(F, S, M), equal(S, 'unsat').
allsat(F, Answers) :- sat(F, M), negateModel(F, M, NewF),
                      allsat(NewF, T), cons(M, T, Answers).
allsat(F, Answers) :- unsat(F), nil(Answers).
```



DO-178C compliance workflow can be captured through Datalog + Cyberlogic

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, provenance, and constraints derived from the domain ontology (e.g., coordinate systems).

Models to Code: HMAC in PVS

```
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
```

From https://en.wikipedia.org/wiki/SHA-2

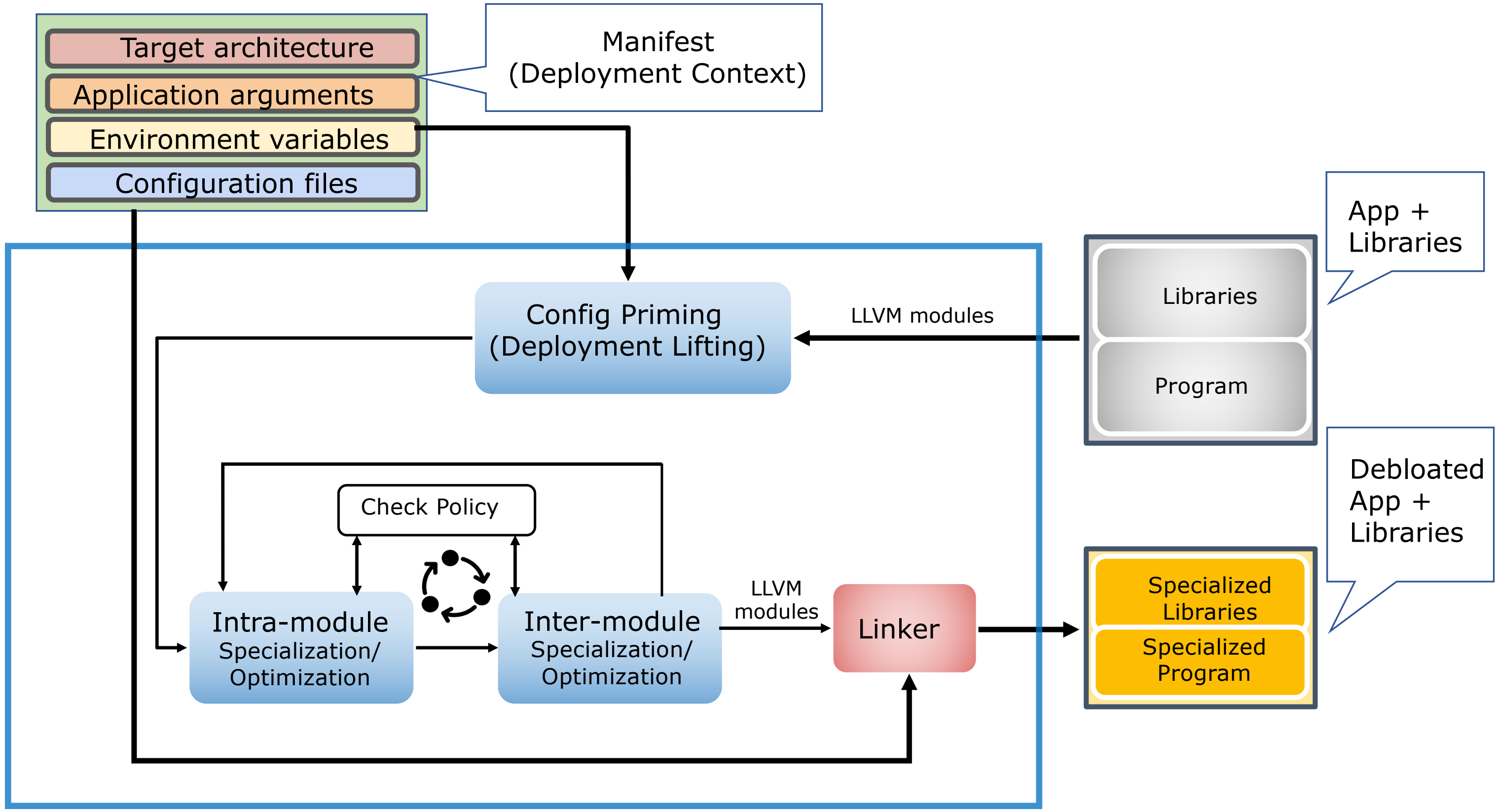
```
hmac(blockSize: uint8,
       key : bytestring,
       (message : bytestring | message`length + blockSize < bytestring_bound),
       outputSize: upto(blockSize),
       hash: [bytestring->lbytes(outputSize)]): lbytes(outputSize)
  = LET newkey = IF length(key) > blockSize THEN hash(key) ELSE key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

hmac256((blockSize: uint8 | 32 <= blockSize),
           key : bytestring,
                (message : bytestring |
             message`length + blockSize < bytestring_bound))
           : 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

OCCAM: Debloating and Sealing Software

- Application is developed on top of a large software stack, but uses only a fraction of it
- The rest of the code might contain exploitable vulnerabilities
- OCCAM is a whole-program LLVM partial evaluator that
    - Eliminates unreachable code
    - Specializes reachable code to the known parameters
    - Preserves legal executions
    - Seals the code with defenses
- Significant reduction in #functions, #instructions, code size



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

- Drew Dean & S, Transforming untrusted applications into trusted executables through static previrtualization. US Patent No. US20130111593A1, 2013.
- G. Malecha, A. Gehani, & S, Automated Software Winnowing, SAC 2015.

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 an architecture has no chance of being correct
- So,
  - Take information seriously and annotate the artifacts with ontic type information
  - Take requirements serious since many major flaws are traceable to poor requirements
  - Take architecture seriously since it is the keystone of an efficient argument
  - Take assurance seriously – composable evidence should be the coin of the realm
  - Take inline and independent runtime monitoring seriously to track integrity
  - 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 and incur a huge cost
- The current strategy of applying larger and larger band-aids is only fueling a futile and costly 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