

Software Assurance: Ontology, Evidence, and Workflows

Natarajan Shankar

SRI International Computer Science Laboratory

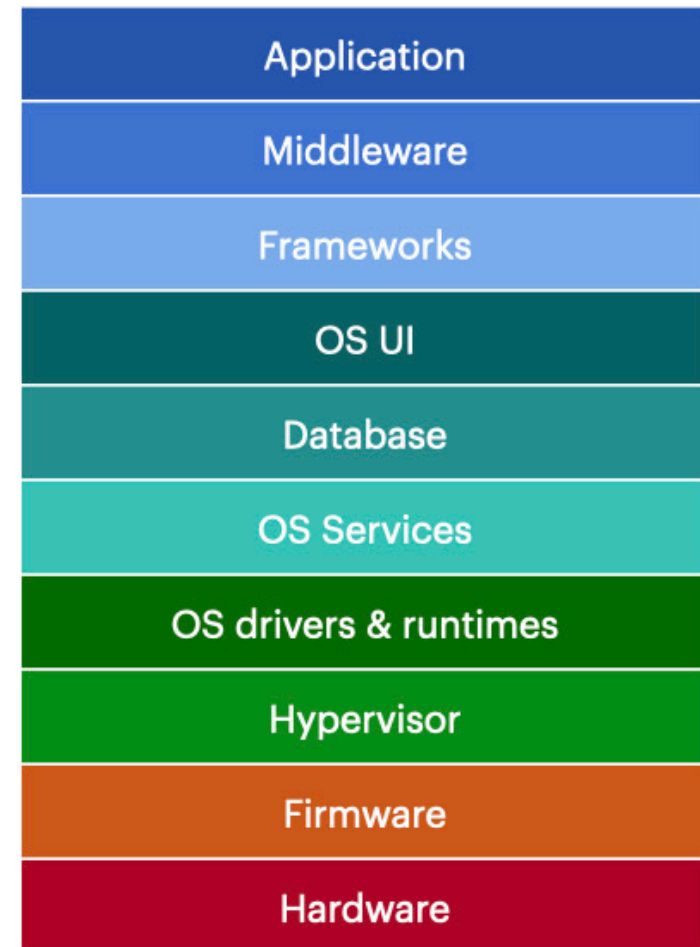
Joint with Devesh Bhatt and the Project DesCert Team
(SRI, Honeywell Research, U. Washington)

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 2.1T \$/year.
- Cybercrime is seen as a 6T\$/year problem, and growing

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

Software Stack



<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.**

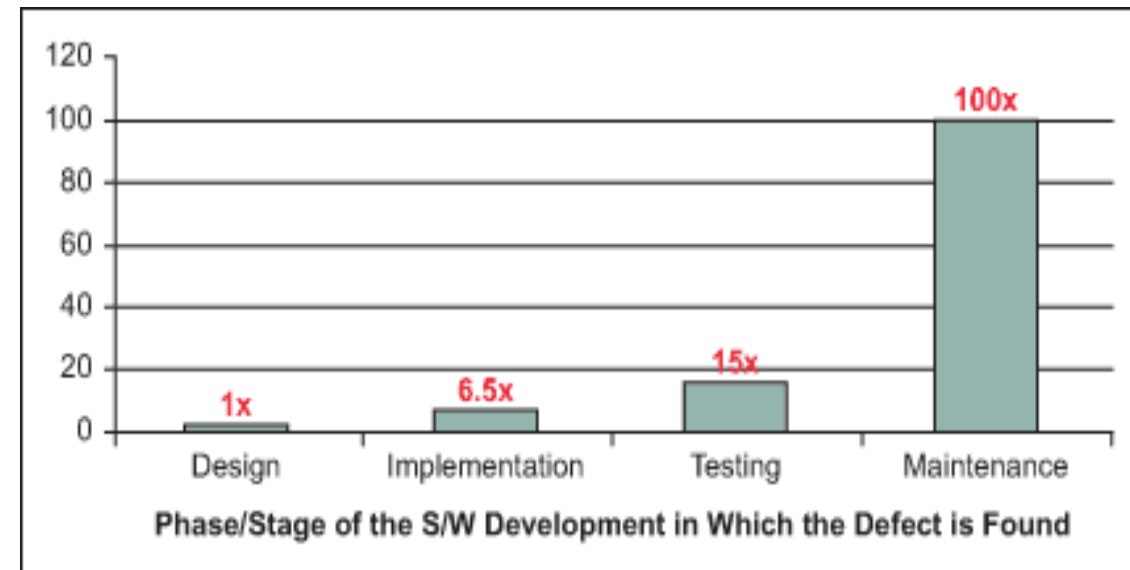
A Few Celebrity Bugs

- 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

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, bad design, defective engineering, inadequate maintenance, and improper operation**.

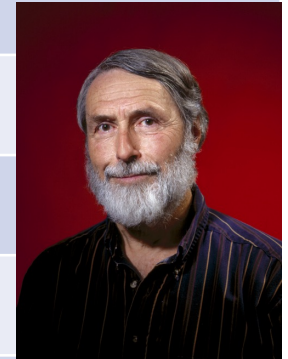
The cost of finding/fixing faults rises dramatically through the software development lifecycle.



<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, 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, M
Memory/Type	Buffer Overflow, null dereference, v
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



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 (**a tough challenge**).
- Even if perfection were only partially attainable, the strategic deployment of lightweight and heavyweight analysis techniques can yield huge dividends.

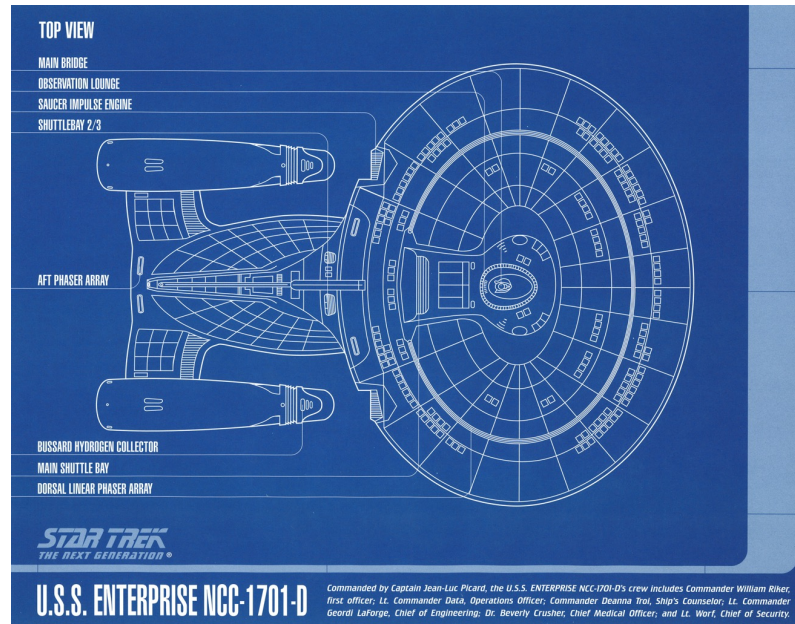
Formal Verification Milestones

- 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)

What then shall we do?

- Formal modeling and analysis is practical and even necessary, but not a panacea
- Many vulnerabilities are consequences of **original sins**: conflating call and variable stacks, stack abuse, broken abstractions, weakened protections, etc.
- 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**).

On Design



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

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, has been flying since 1969 but the AAR had been added by BAE first in 1982 and upgraded in 1989, and certified on the basis of a safety case developed by BAE in consultation with QinetiQ 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.

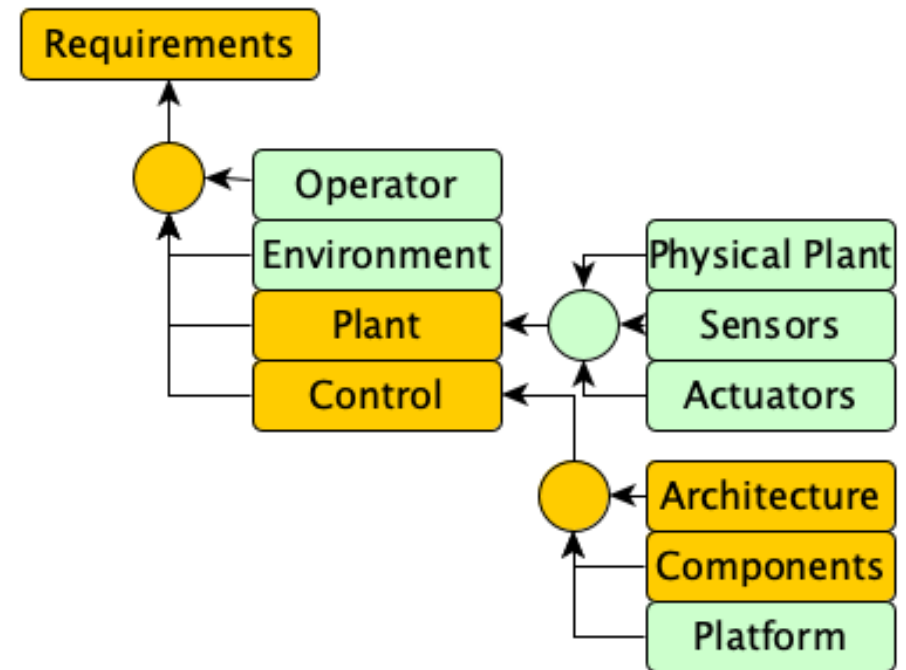
Haddon-Cave on the Nimrod Safety Case

- *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.”

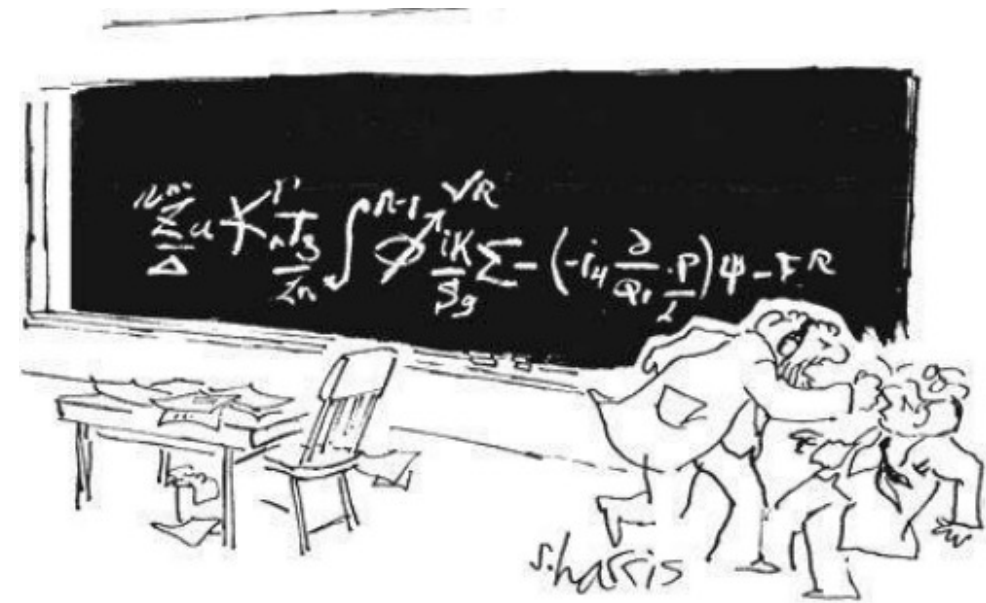
FDA Draft Guidance document Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions: ... *an assurance case is a formal method for demonstrating the validity of a claim by providing a convincing argument together with supporting evidence. It is a way to structure arguments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

Making Arguments Efficient (for the skeptic)

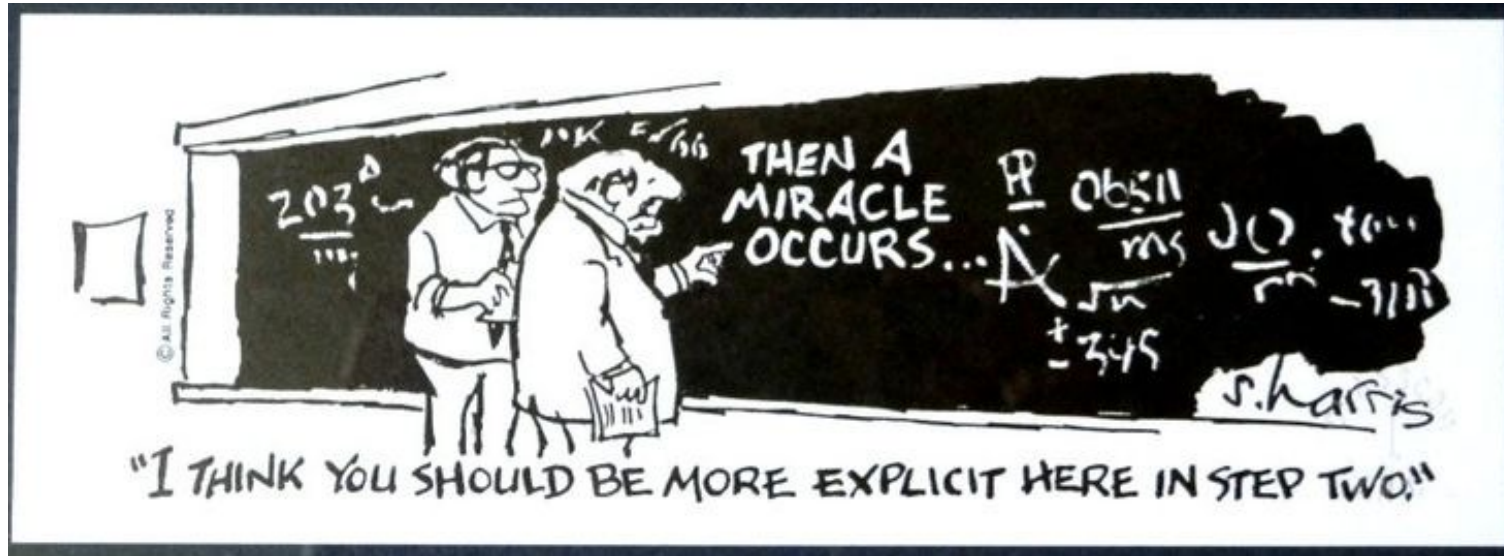
- An argument for a design is a tree of claims, subclaims, and assumptions.
- An assurance case is a theory-supported structured argument with claims, subclaims, and assumptions backed by artifacts and evidence that demonstrates that the software faithfully implements the intended behavior.
- The assumptions, e.g., on the environment or sensors, are supported by evidence
- The methods for the decomposition of claims into subclaims should be backed by a theory.
- A well-structured argument that can be effectively challenged by a skeptic: no leaps of faith.
- A good design should support an efficient argument that expands the falsification space for the skeptic.
- 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, 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



All models are wrong, but some are useful.

George E.P. Box

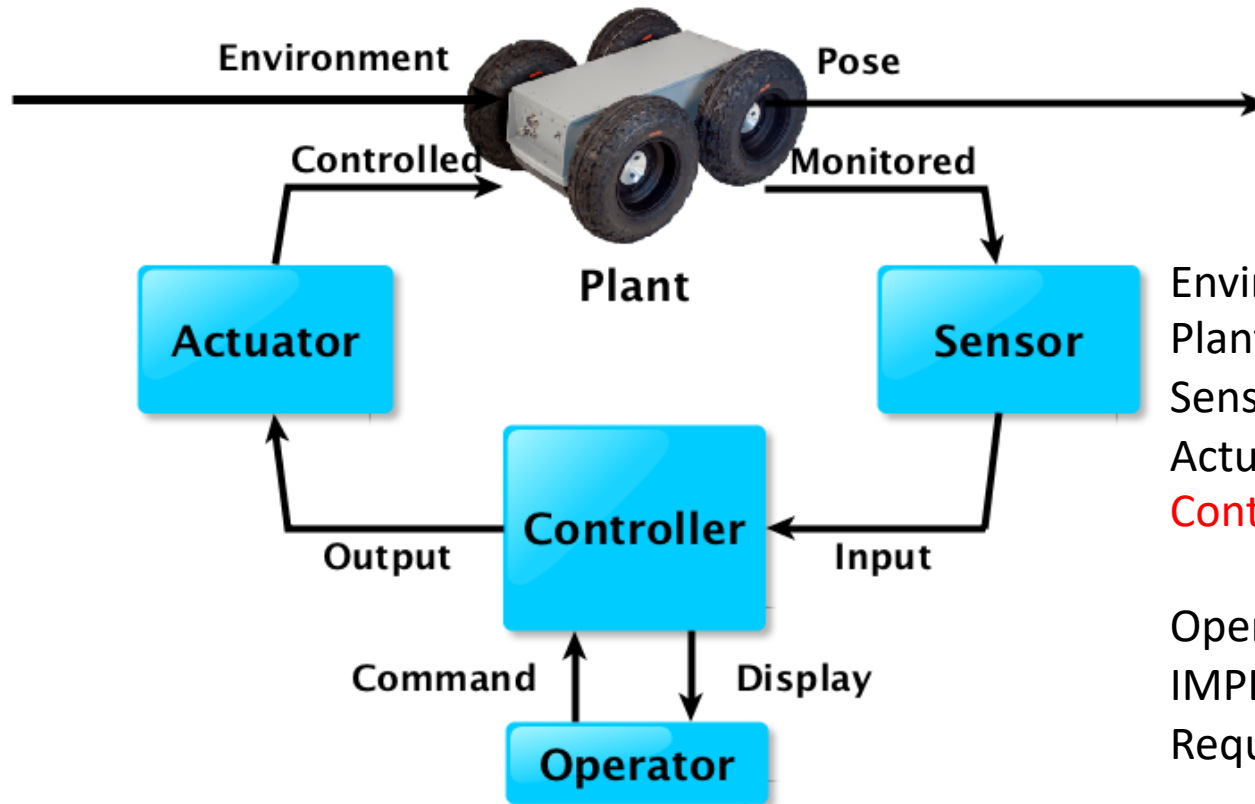
<https://i.pinimg.com/736x/d6/e7/54/d6e754d24aaef324c1595e68583ace7a.jpg>

Efficient arguments use

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

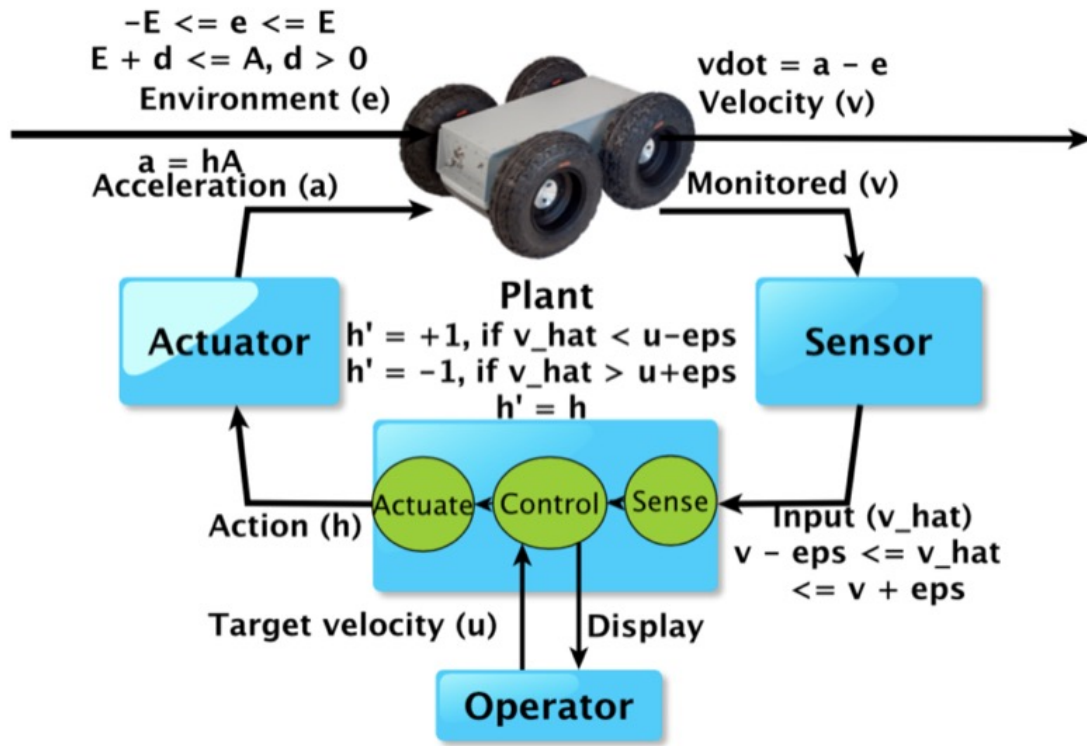
- **Models** (plant, environment, sensor, actuator, operator, platform, fault), **Architectures**, **Languages**, and **Tools** are the pillars of efficient arguments
- Efficient arguments lower the amortized falsification cost through big, reusable claims that expand the falsification space.

The Eight-Variables Model



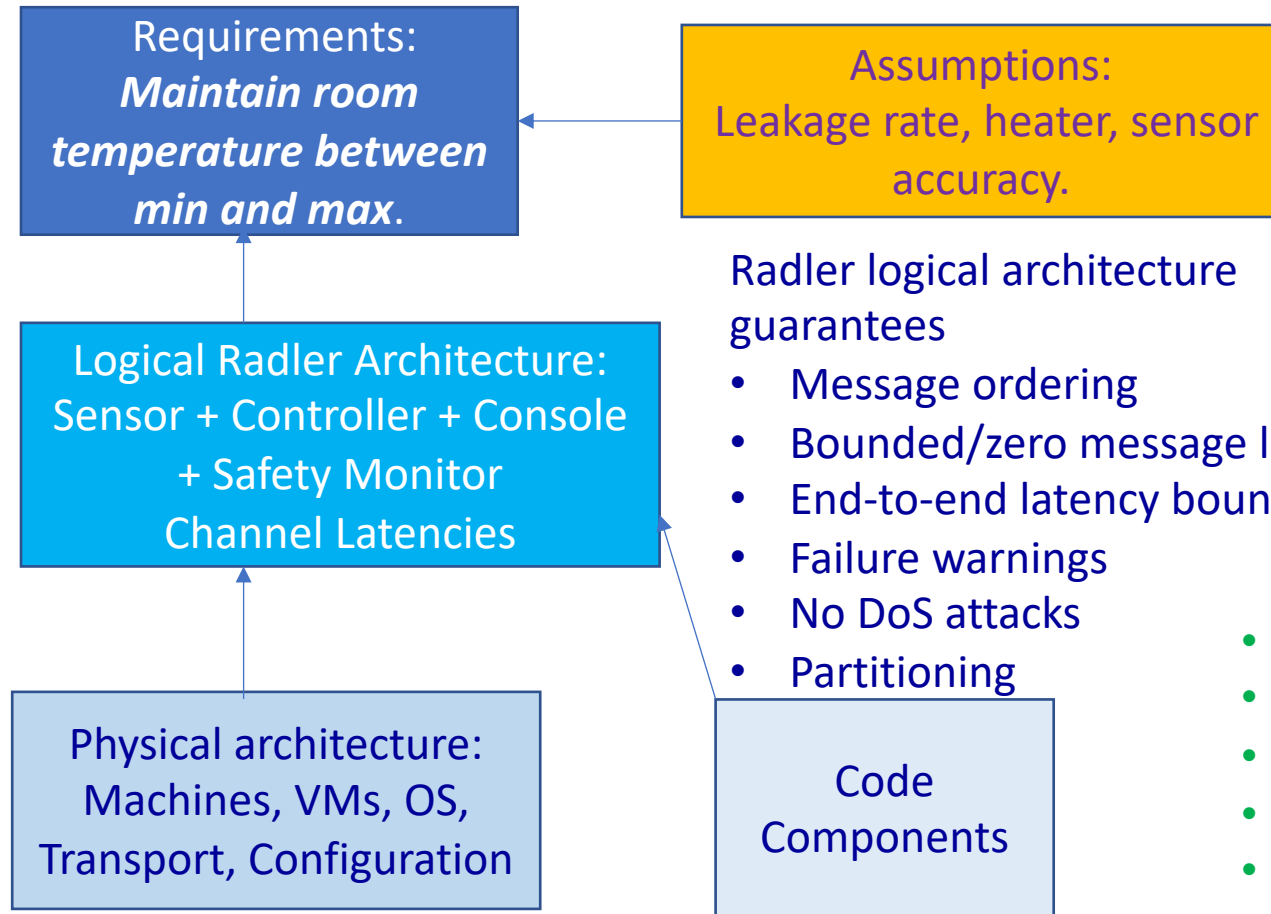
EnvironmentAssumption(environment) AND
 PlantModel(environment, control, pose, monitor) AND
 SensorAccuracy(monitor, input) AND
 ActuatorResponse(output, control) AND
ControllerSpecification(input, command,
output, display) AND
 OperatorModel(display, command)
 IMPLIES
 Requirement(command, environment,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

8-Variables: An Example



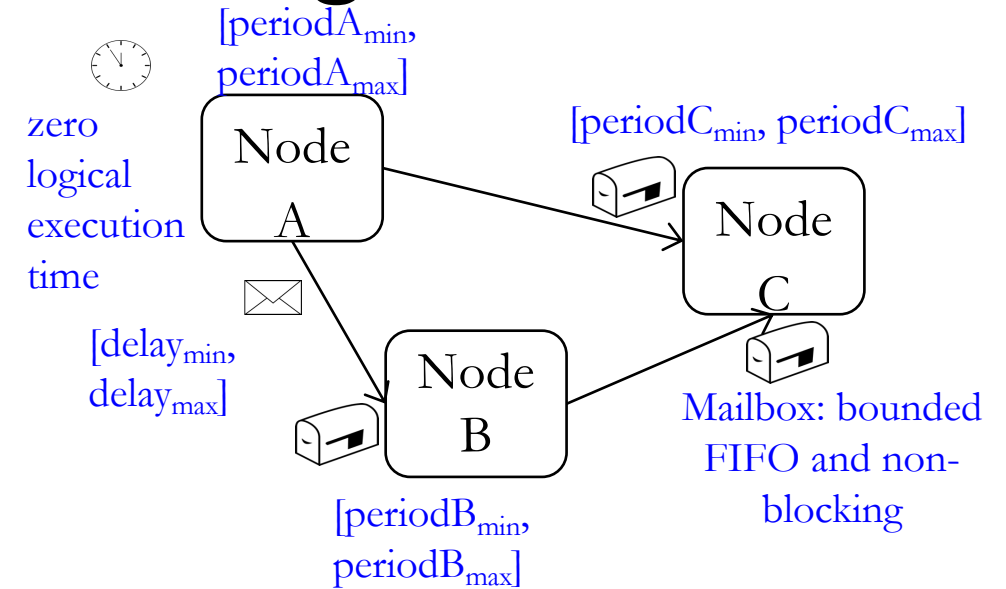
- The Plant consists of the vehicle that is trying to maintain a speed v and the Environment e is the grade of the road.
- The goal requirement is to maintain the vehicle velocity v within some bound of the target velocity u .

Radler Architecture for Efficient Arguments



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

Security Assurance

Attacks on IoT/cyber-physical systems include sensor spoofing, jamming, malware, bad input, unprotected/unauthenticated communication, unauthorized access

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

1. Ontological categories for *modeling* of:

1. Threats¹: Weak access control, weak input validation, race conditions, timing attacks, phishing, privilege escalation
2. Vulnerabilities²: Null dereference, SQL injection, Buffer overflow
3. Controls³: Physical security, Access control, Monitoring, Reporting, Authentication
4. Risk/loss events⁴: Loss of Confidentiality, Integrity, Availability, Safety.
5. Architecture/Touch (entry) Points: Sensors, Actuators, Communication channels, Files, Hardware

2. State and *prove* safety/security *properties* of entities modeled in the ontology.

DesCert Approach: Ontology as the basis for Security Assurance

1. Ontological Formalisms in CLEAR for modeling of:

- Attacks, vulnerabilities, controls, security violations
Mapped to Architectural elements and Touch Points
- Properties that mitigate vulnerabilities

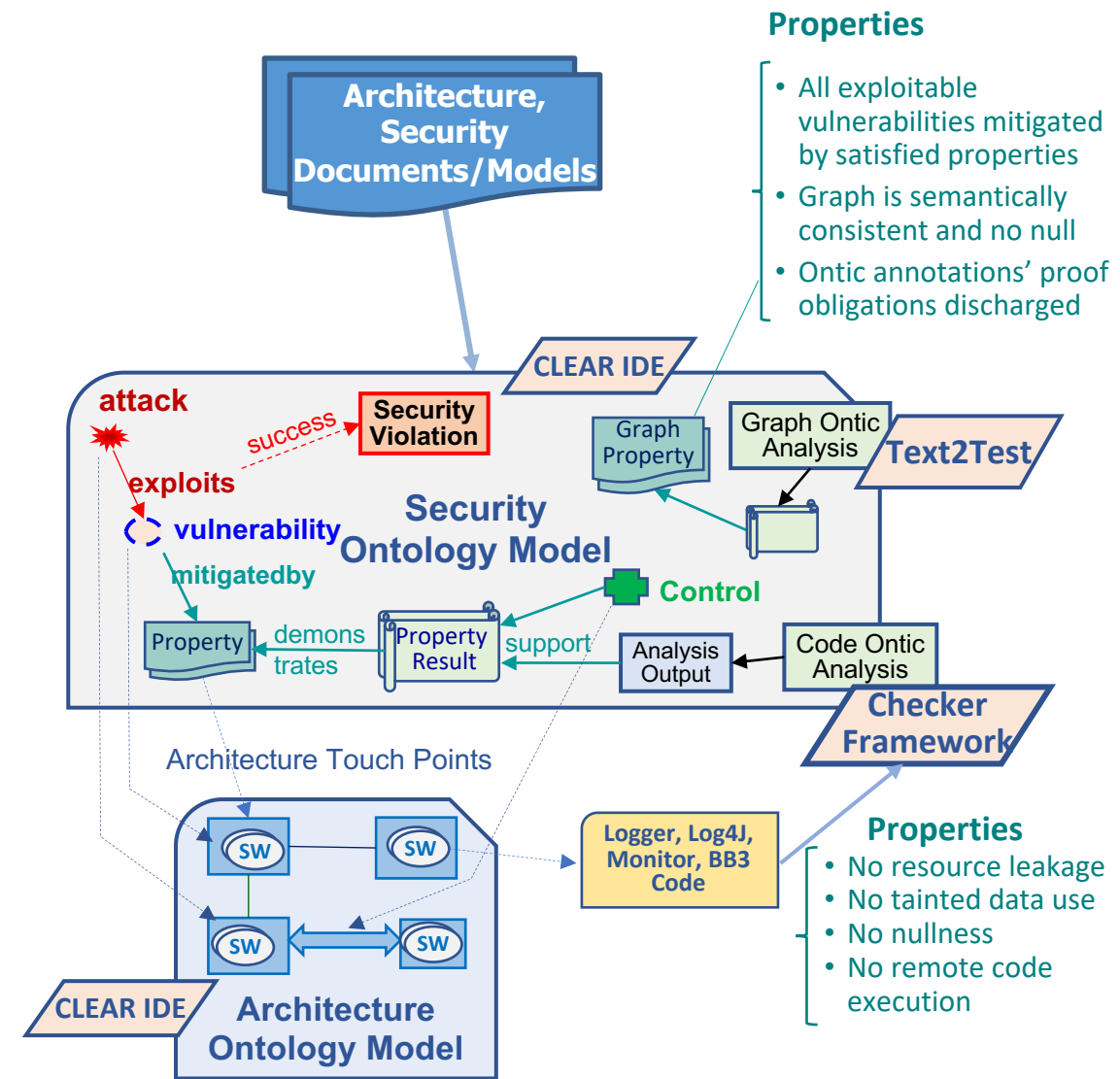
2. State and prove safety/security properties of modeled entities, rather than a solely process-compliance approach

Properties' Results are established by:

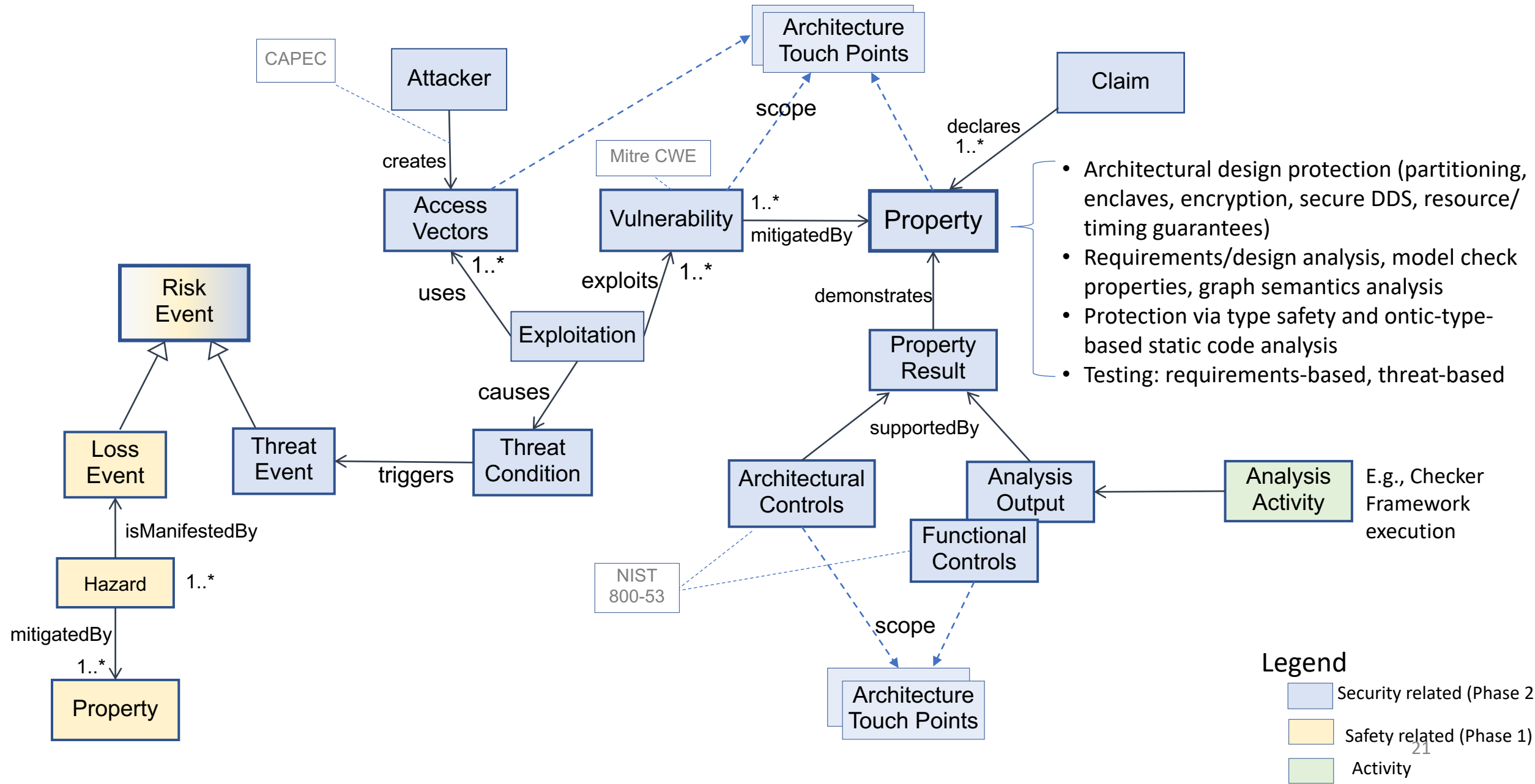
- Architectural design protection (partitioning, enclaves, encryption, secure DDS, resource and timing guarantees)
- Requirements/design analysis, model checking of properties, graph semantics analysis
- Protection via type safety (by construction) and ontic-type-based static code analysis
- Testing: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

The ontology elements and their relationships provide a way to create corresponding evidence sets and reasoning for direct construction of Assurance Claims.

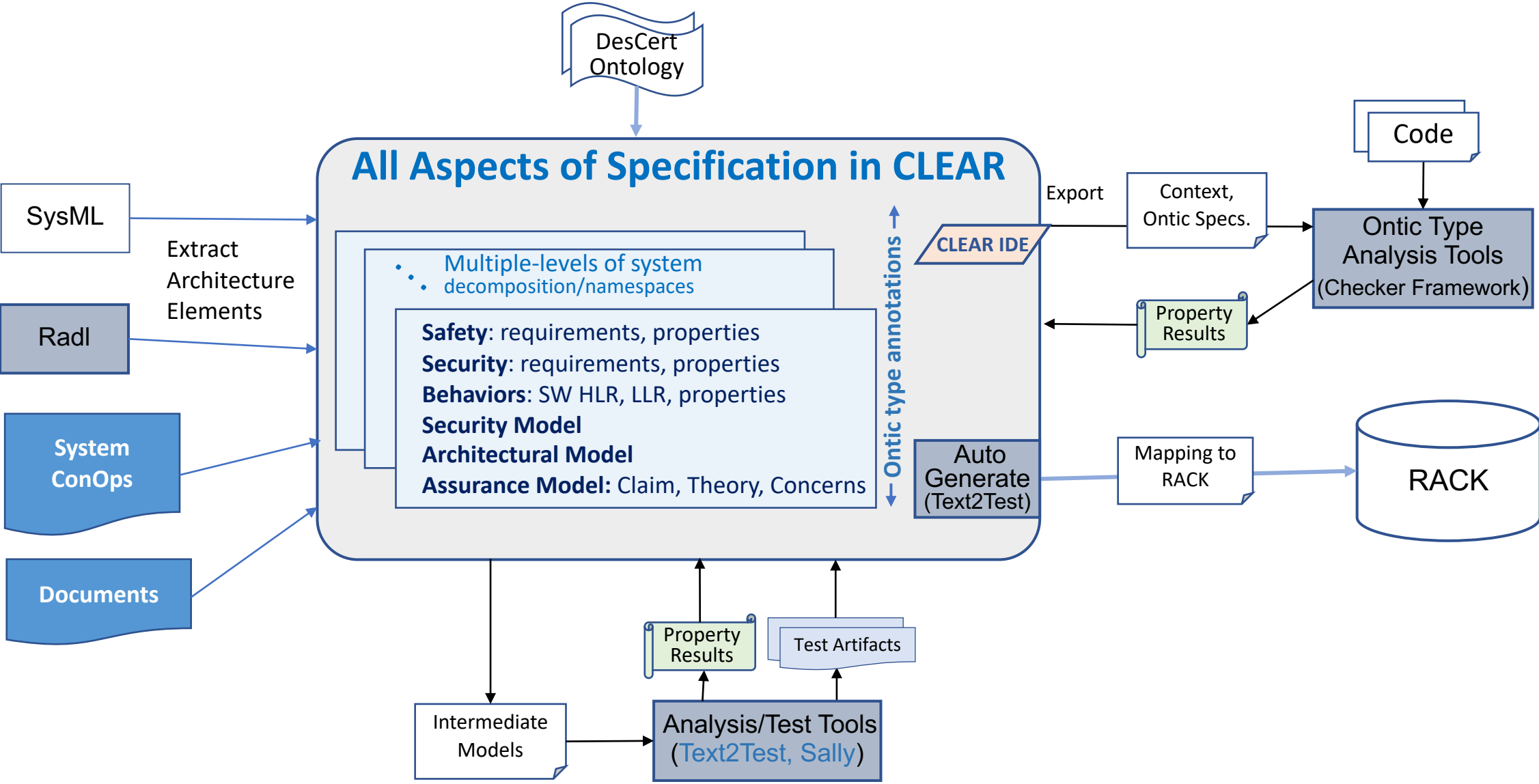
E.g.: Threat A1's exploiting of vulnerabilities (v1, v2) is blocked due to controls (c1, c2, c3) present (with associated property results) in the architecture (radl1) in software components (s1, s2).



DesCert Evidence Ontology for Integrated Security/Safety Analysis



CLEAR as a Formal Notation for all Specifications in DesCert



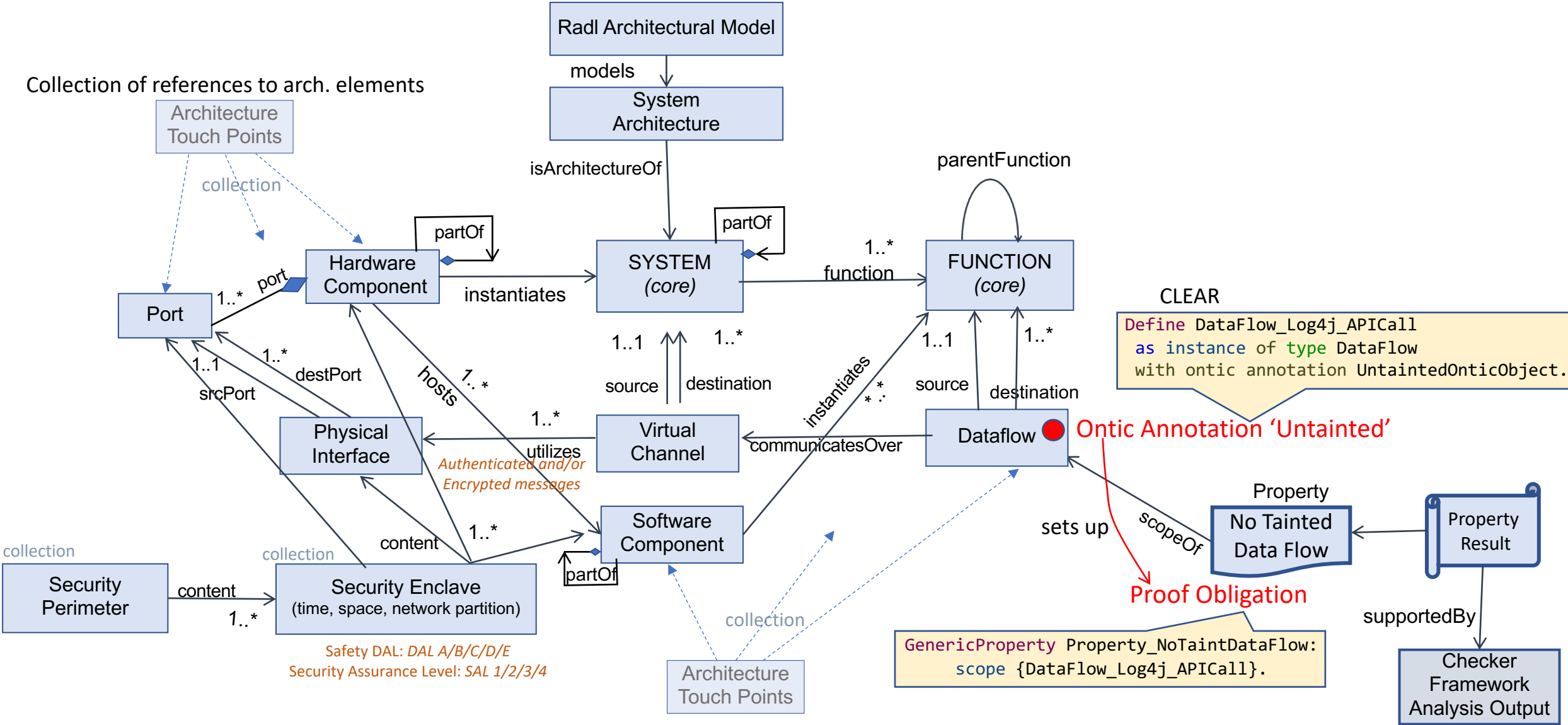
Ontic Type Analysis

- 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

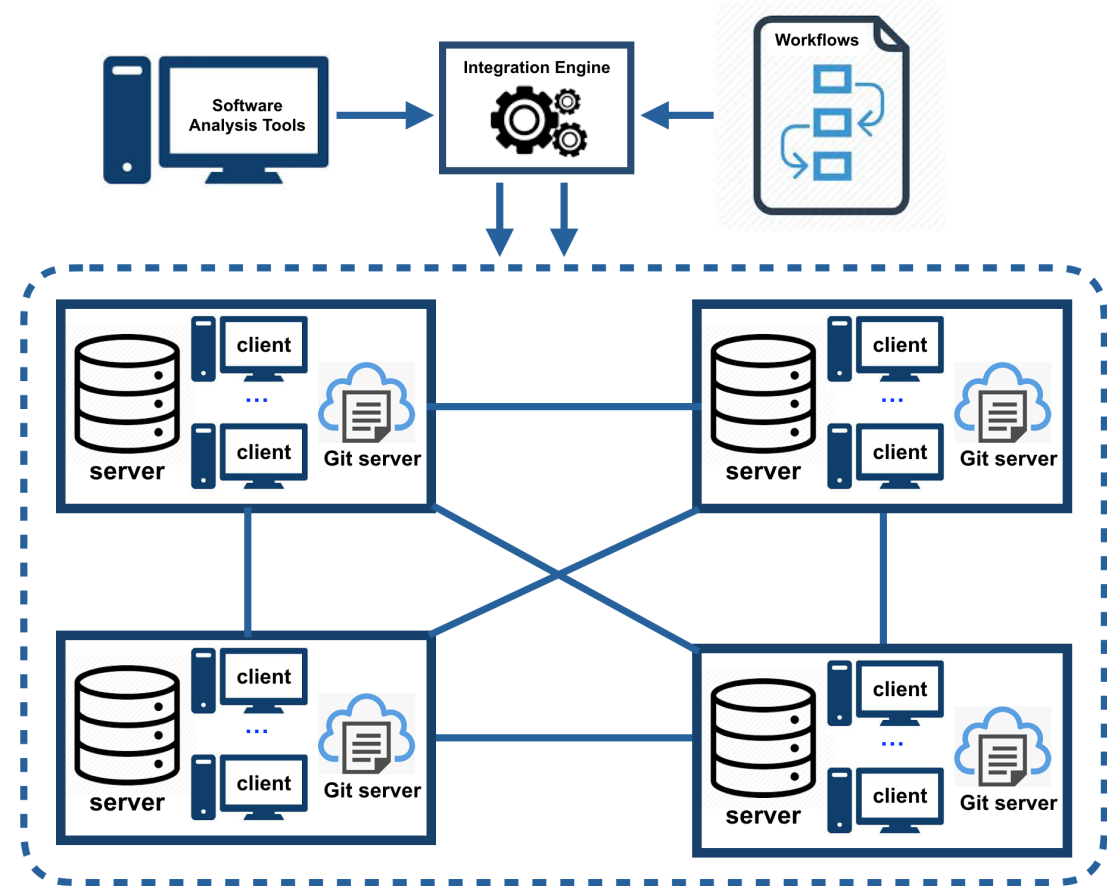
DesCert Architecture Ontology with example of CLEAR Ontic Type Annotations



Lifting ontic type specifications/annotations to higher levels of system model

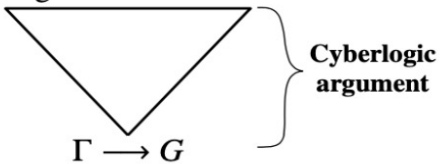
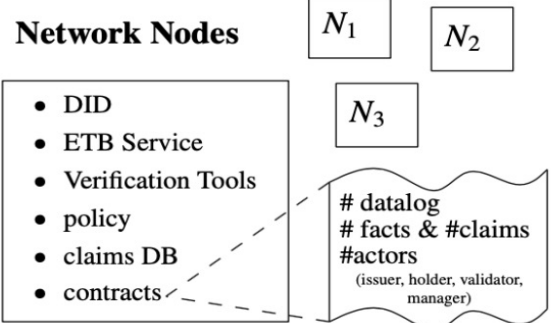
Evidential Tool Bus (ETB2)[SRI/fortiss]

- 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

Component	Functionalities	Implementation	Challenges
Cyberlogic	<p>Logical Foundations</p> <p>Proof theoretic and operational semantics</p> <p>Evidential transaction as a Cyberlogic argument</p>	<p>Digital Certificates</p> 	<p>How to define executable business logic for evidential transactions?</p>
Evidential Tool Bus	<p>Services</p> <p>Verifiable Evidential Transactions</p> <p>Automated construction of a Cyberlogic argument</p>	<p>ETB service</p> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> <p>Cyberlogic Programs</p> <p>$h \text{ :- } b_1, \dots, b_n$</p> <p>Docker</p> </div> <p>→ automated service</p> <p>→ manual service</p>	<p>How to generate and continuously maintain evidential transactions?</p>
Distributed Evidence Network	<p>Distributed Execution</p> <p>Secure Distributed Substrate</p> <p>Secure construction of a distributed Cyberlogic argument.</p>	<p>Network Nodes</p> <ul style="list-style-type: none"> • DID • ETB Service • Verification Tools • policy • claims DB • contracts 	<p>How to securely distribute and build trusted and accountable evidential transactions?</p>

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 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 the assurance ontology seriously – it binds the claims to the evidence
 - 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
- 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