OVERTURE
AN INTRODUCTION CONSERTS

Andreas Schmidt

@ Dependable Systems and Software Chair – Jan. 18th 2022

Fraunhofer
IESE

public

Motivation

- Safety Assurance in Cyber-Physical Systems is impeded by
  - Increasing complexity of systems of systems
  - Increasing variability in terms of operating modes and collaboration groups

- Runtime Monitoring allows to assure certain properties during operation, based on pre-assured components

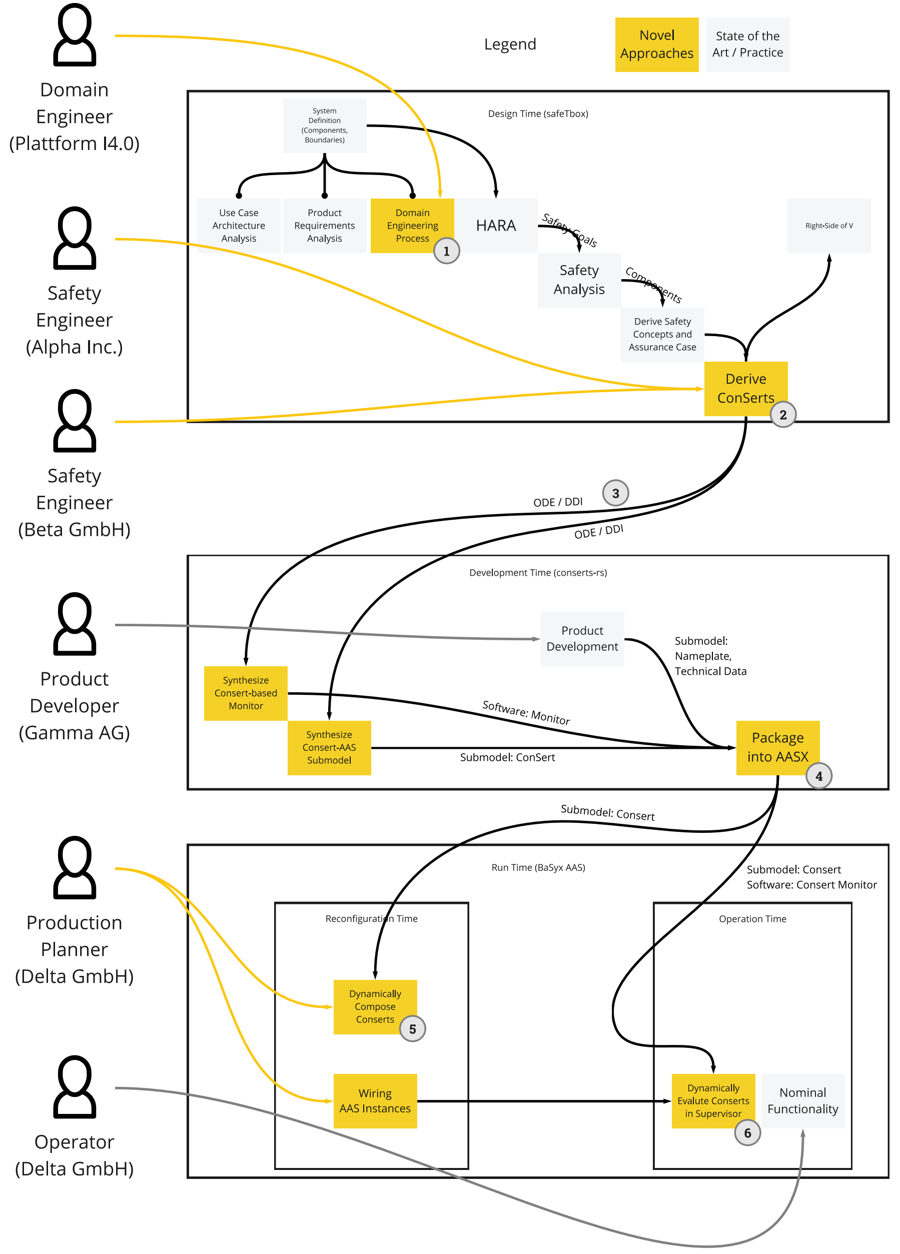
- Creation of Runtime Monitors should be as automated as possible – avoiding the introduction of development faults

AGENDA

- Conditional Safety Certificates (ConSerts)
- Implementation
  - Services & Dimensions
  - Safety Evaluation
  - Deployment Targets and Framework
- Evaluation & Discussion

Safety Engineering with ConSerts

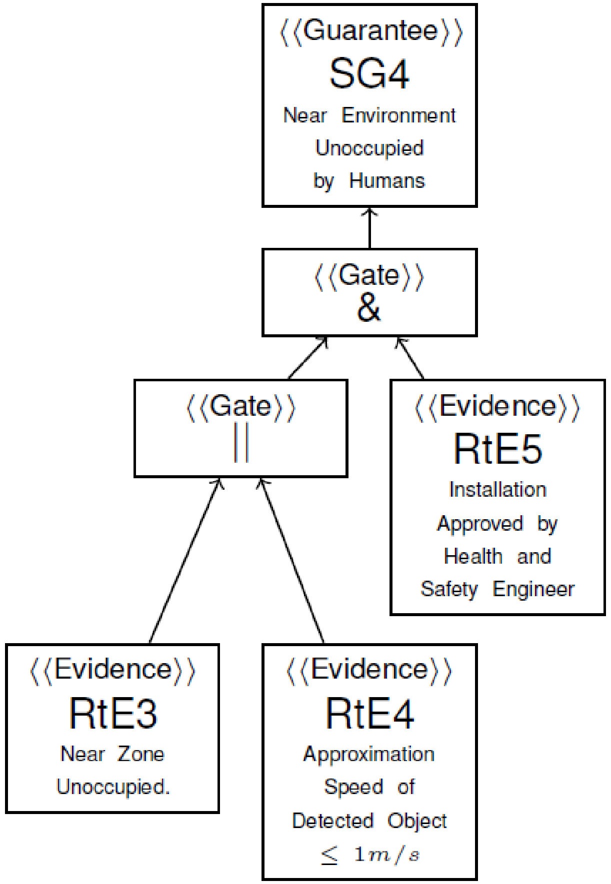
- Domain Engineers define Ontology / Type System
- Safety Engineers derive ConSerts from Assurance Process
- ConSerts are used for
  - Composition- (aka Compile-) Time Assurance
    Can systems conceptually collaborate?
  - Run-Time Assurance
    what can systems do safely right now



public

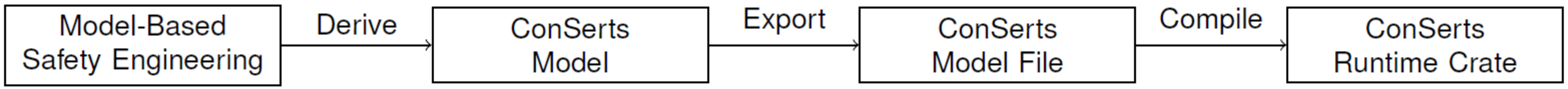
Fraunhofer IESE

Conditional Safety Certificates (ConSerts)



- Methodology-wise derived from Assurance Cases
- Model-Based Approach to specify
  - Success Trees for System-Local Behaviour and Provided Guarantees
  - Demand-Guarantee Relations between Collaborating Systems
- Collaborations and Adaptations assured conditionally at design-time
- Conditions monitored and evaluated at run-time
- Approach technically suited to also model quality properties that are not safety

End-to-End Safety Engineering with ConSerts



- Execute Model-Based Safety Engineering to identify:
  - Safety Properties that can only be assured collaboratively
    One system guarantees another system's demand

- Exported ConSerts Model File serves as input to `conserts-rs`
  - Rust-based Command-Line Tool
  - `compile` turns ConSert models into executable Rust-based monitoring code
  - `compose` takes several ConSerts and checks if they can be composed safely

- Automating this process integrates well with *Continuous Delivery for Safety-Critical Software*[1]

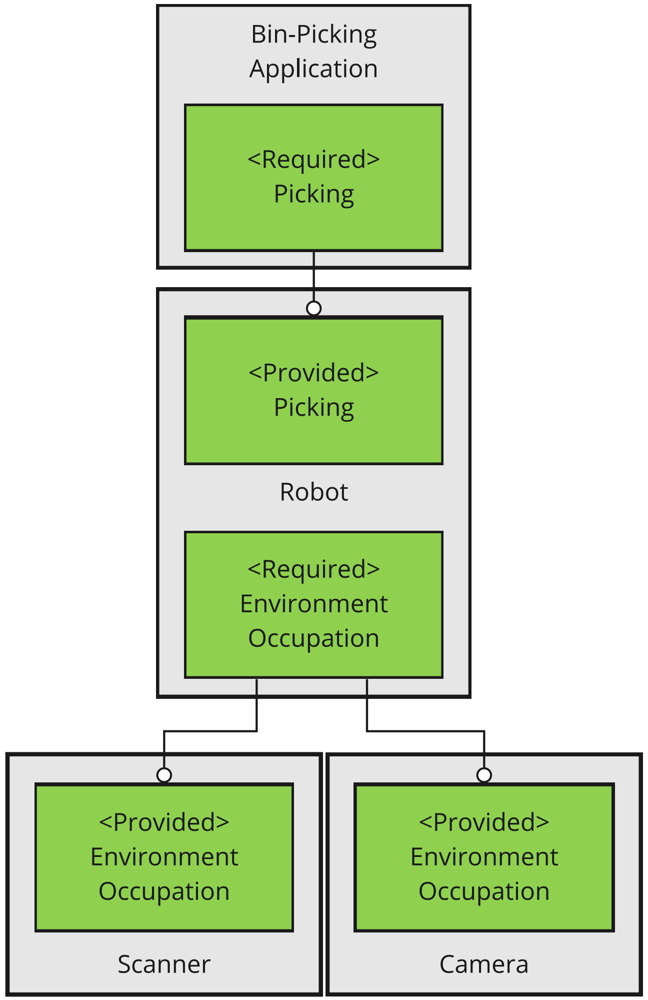
[1] Marc Zeller, Daniel Ratiu, Martin Rothfelder, and Frank Buschmann. An industrial roadmap for continuous delivery of software for safety-critical systems. In 39th International Conference on Computer Safety, Reliability and Security (SAFECOMP), Position Paper, 2020.

ConSert Services

- When formalizing, we consider Services as provided/required by individual systems
- Services have a type and multiple services can be composed if they have the same type

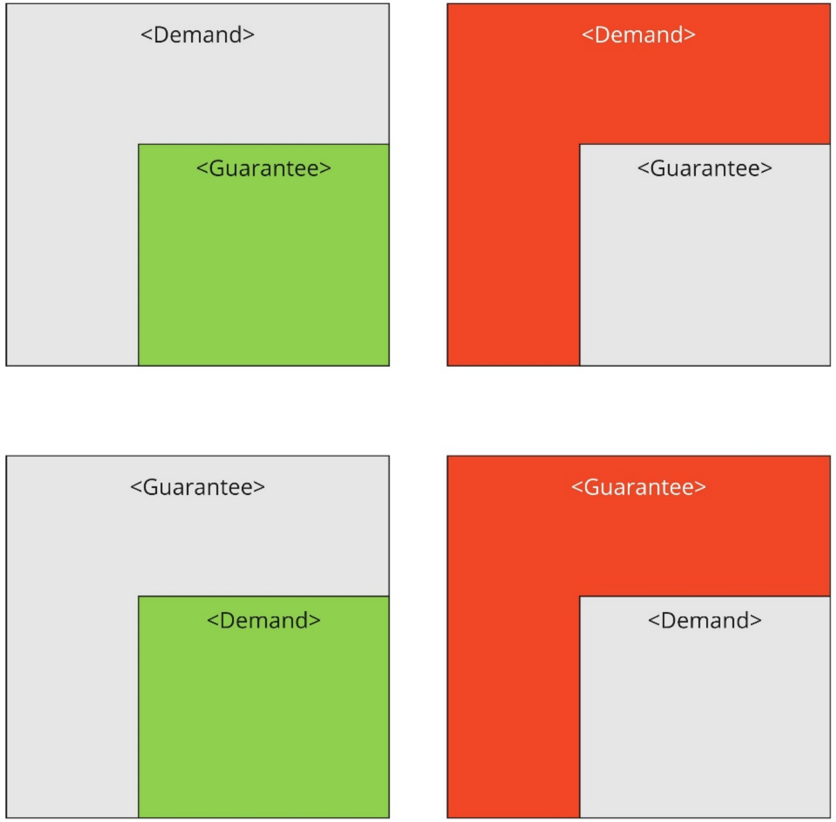
- Provided Service
  - Type
  - Guarantees

- Required Service
  - Type
  - Demands



ConSert Dimensions

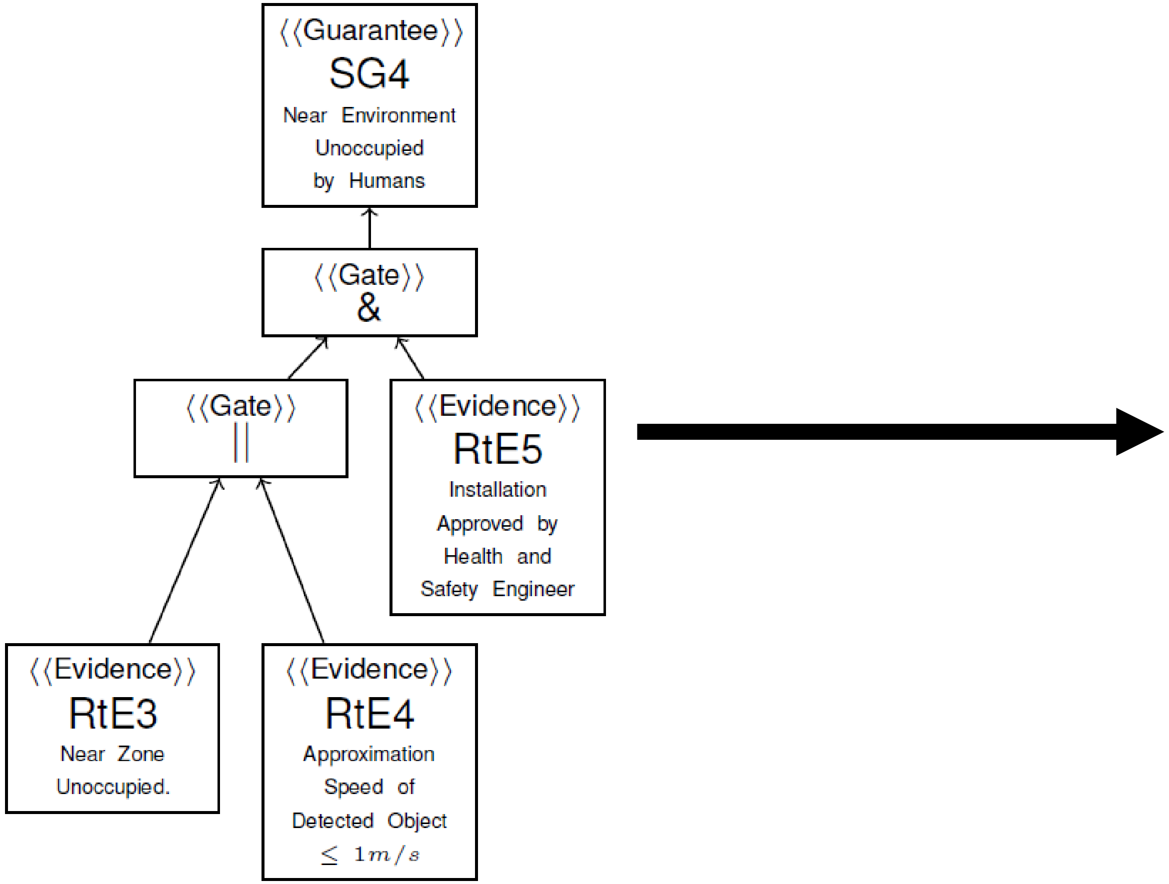
```
- id: ExampleGuarantee
  dimension:
    Numeric:
      type: UnoccupiedTime
      covered:
        - Inclusive:
            start: 0.0
            end: 1.48
      subset: Demand
      uom: second
```



ConSert Composition

- Start with an empty System-of-Systems
- While adding a ConSert:
  - Check for all required services if there are matching provided services
  - Check for each demand in a required service if at least one matching guarantee is present
- Matching is defined as dimensions musts match
  - Binary, Categorical, Numerical
  - If Categorical / Numerical, considered Subset relationship and covered set

Implementation | Safety Evaluation




```
use super::evidence::RuntimeEvidence;
#[doc = "Near Environment Unoccupied by Humans"]
pub struct SG4;
impl SG4 {
  pub fn evaluate(runtime_evidence: &RuntimeEvidence)
    -> bool {
    {
      let c0 = {
        #[doc = "Near Zone Unoccupied."]
        let c0 = runtime_evidence.RtE3;
        #[doc = "Approximation Speed of
        Detected Object <= 1m/s"]
        let c1 = runtime_evidence.RtE4;
        c0 || c1
      };
      #[doc = "Installation Approved by Health
      and Safety Engineer"]
      let c1 = runtime_evidence.RtE5;
      c0 && c1
    }
  }
}
```

Deployment | Embedded System with rtic.rs

```
use consert_edcc2021::{properties::*, *};

#[app(device = target, peripherals = true)]
const APP: () = {
  struct Resources {
    safe: bool,
    rtp: properties::RuntimeProperties,
    monitor: monitor::Monitor,
    // ...
  }
  // ...
  #[task(resources = [safe, monitor, rtp])]
  fn evaluate_safety(
    cx: evaluate_safety::Context) {

    let resources = cx.resources;
    // Move current sample to monitor
    resources.monitor.add_sample(
      *resources.rtp);
    *resources.rtp = RuntimeProperties::unknown();

    // Evaluate safety
    let rte = resources.monitor.get_sample();
      *resources.safe =
        guarantees::SG4::evaluate(&rte);
  }
}
```

```
#[task(binds = RTC0, resources = [rtc_0, rtp,
  velocity_sensor])]
fn rtc(cx: rtc::Context) {
  use uom::si::{velocity::*, f64::Velocity};

  let velocity =
    *cx.resources.velocity_sensor.sample();
  let rtp: &mut RuntimeProperties =
    &mut *cx.resources.rtp;
  rtp.approximation_speed_of_detected_object =
    ApproximationSpeedOfDetectedObject::Known(
      Velocity::new::<meter_per_second>(velocity));
}
```


public

Fraunhofer IESE

Deployment | ROS & C++

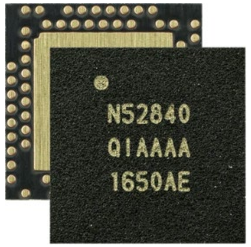
- ROS is popular in both academia and industry
- Foreign-Function Interfaces (FFI) to C++ allow for high flexibility in terms of application context
- Both deployment variants can be auto-generated

```rust
#[no_mangle]
pub unsafe extern "C" fn Monitor_get_sample(
  ptr: *mut Monitor) -> *mut RuntimeEvidence {

  let monitor = {
    assert!(!ptr.is_null());
    &mut *ptr
  };
  Box::into_raw(Box::new(monitor.get_sample()))
}
```

```rust
use crate::prelude::*;
use rosrust::{Publisher, Subscriber};
// more use statements omitted for clarity

pub struct RosMonitor {
  pub rtp: Arc<AtomicCell<RuntimeProperties>>,
  pub monitor: Monitor,
  pub subscriptions: Vec<Subscriber>,
  pub SG4: Publisher<msg::consert_edcc2021::SG4>,
  pub SG5: Publisher<msg::consert_edcc2021::SG5>,
}

impl RosMonitor {
  // ...
  pub fn run_standalone(mut self,
    frequency: Frequency) {
    let rate = rosrust::rate(
      frequency.get::<hertz>());
    while rosrust::is_ok() {
      let rte_sample = self.cycle();
      self.publish_all(&rte_sample);
      rate.sleep();
    }
  }
  pub fn cycle(&mut self) -> RuntimeEvidence {
    let rtp_sample = self.rtp.load();
    self.rtp.store(RuntimeProperties::unknown());

    self.monitor.add_sample(rtp_sample);
    self.monitor.get_sample()
  }
}
```


Fraunhofer IESE

Safety Evaluation Performance



https://www.nordicsemi.com/Products/nRF52840

- Measuring inference latency (WCET), aka „how many cycles are taken to compute guarantee value“.
- Nordic Semiconductor nRF52840, 64MHz Arm Cortex-M4 FPU, 1MB Flash, 256 KB RAM
  - No chip for safety-critical domain, but CPU & memory specification is comparable
- For realisitic ConSert size of 50 evidence (other values in paper confirm linear relationship):
  - Monitor Cycling Time: 92.92us (store current, apply majority vote over history)
  - Guarantee Evaluation Time: 3.22us (evaluate boolean logic)
- *In summary: additional monitoring meets real-time constraints and adds only little latency.*

Code Review Complexity

```rust
use super::evidence::RuntimeEvidence;
#[doc = "Near Environment Unoccupied by Humans"]
pub struct SG4;
impl SG4 {
  pub fn evaluate(runtime_evidence: &RuntimeEvidence)
    -> bool {
    {
      let c0 = {
        #[doc = "Near Zone Unoccupied."]
        let c0 = runtime_evidence.RtE3;
        #[doc = "Approximation Speed of
        Detected Object <= 1m/s"]
        let c1 = runtime_evidence.RtE4;
        c0 || c1
      };
      #[doc = "Installation Approved by Health
      and Safety Engineer"]
      let c1 = runtime_evidence.RtE5;
      c0 && c1
    }
  }
}
```

- Manual review of the auto-generated code is one option we consider
  - Review must be feasible, i.e. code should be comprehensible
- We consider lines of code (LoC) as most lines generated by conserts-rs are rather straightforward
- Code consists of (a) static infrastructure (monitor) and (b) dynamic ConSert-dependent code
  - Static code accounts for 150 LoC, most of it is infrastructure without logic
  - Dynamic code accounts for: 18 LoC per runtime property, 7 LoC per monitored evidence/demand, 1 LoC per gate and 1 LoC per evidence in evaluation tree, 4 LoC per guarantee
- *In summary, a 50 evidence ConSert yields approx. 1500 LoC that are still feasible to review.*

public

Fraunhofer IESE

Correctness & the Rust Programming Language

- Correctness of Code generation must be proved – several components must be qualified
  - 1. Code-Generation Logic (conserts-rs)
    - Mapping from Boolean trees to monitor code can be verified formally or via test kit
    - Small size of ConSert models allows for manual code review
  - 2. Rust Compiler
    - No certified compiler yet
    - Ferrocene[1] promises to solve that until end of 2022 for ISO 26262 ASIL B
- ConSert model validation (code-gen input) is executed via well-known safety assurance processes

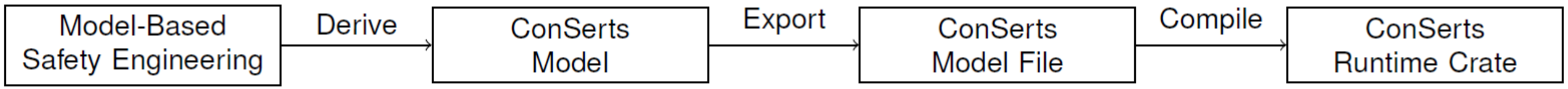
[1] https://ferrous-systems.com/ferrocene/

Fraunhofer IESE

Outlook

- Openly publish
  - Opus: The Book of ConSerts (handbook-style documentation for ConSerts)
  - `conserts-rs` (Rust-based library and CLI to work with ConSerts)

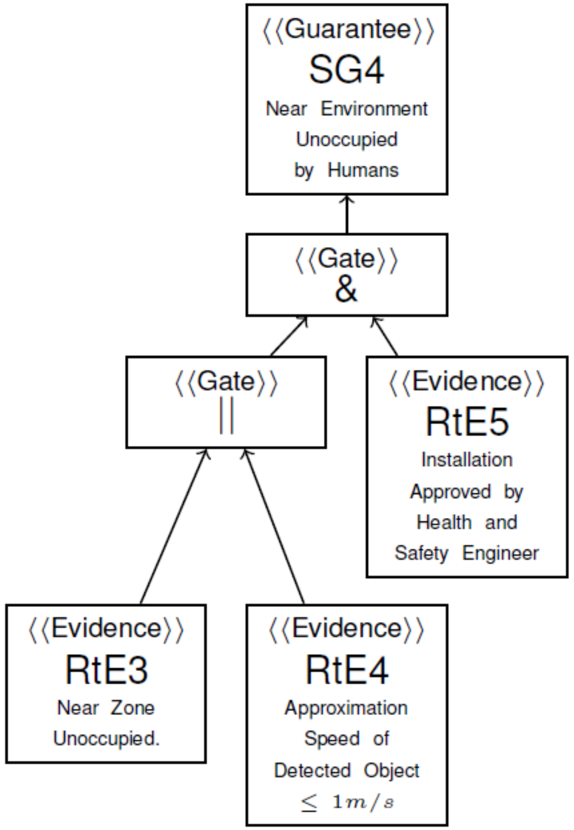
Summary



HARA &

FMEA &

Assurance Case




```
use consert_edcc2021::{properties::*, *};

#[app(device = target, peripherals = true)]
const APP: () = {
  struct Resources {
    safe: bool,
    rtp: properties::RuntimeProperties,
    monitor: monitor::Monitor,
    // ...
  }
  // ...
  #[task(resources = [safe, monitor, rtp])]
  fn evaluate_safety(
    cx: evaluate_safety::Context) {

    let resources = cx.resources;
    // Move current sample to monitor
    resources.monitor.add_sample(
      *resources.rtp);
    *resources.rtp = RuntimeProperties::unknown();

    // Evaluate safety
    let rte = resources.monitor.get_sample();
      *resources.safe =
        guarantees::SG4::evaluate(&rte);
  }
}
```