SafetyADD:
A Tool for Safety-Contract Based Design

Fredrik Warg
Benjamin Vedder
Martin Skoglund
Andreas Söderberg
SP Technical Research Institute of Sweden

What is SafetyADD?

- A tool for working with safety contracts for software components
  - Create safety contracts
  - Verify that contracts match when components are integrated
- Developed within the nSafeCer project (http://safecer.eu)

Research questions

- Investigate validity of safety contracts when changing context for components
- Explore how safety contracts can be expressed and used efficiently in the software design/certification process

Our goal with Safety-contract based design

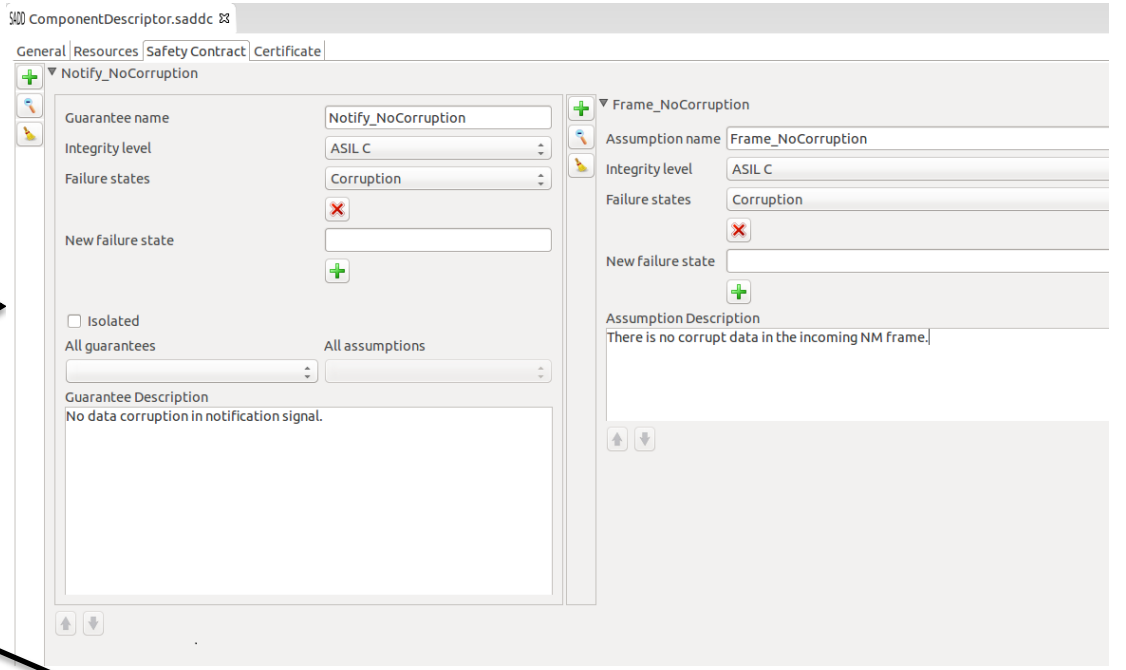
- A foundation for realizing pre-certified software components
  - Simplify certification of complex systems
- Improve efficiency and reduce errors when integrating safety-critical software components

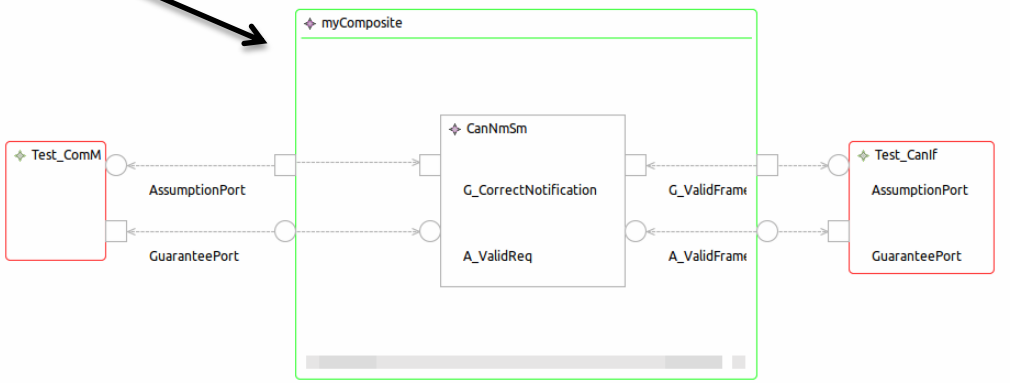
Outline

- SafetyADD overview
- Software components - primitive component
- Safety Contracts
  - Assumptions and guarantees
  - Demo
- Software integration - composite component
  - The composite component editor
  - Demo
- Contract checking
  - Example report
- Summary and future work

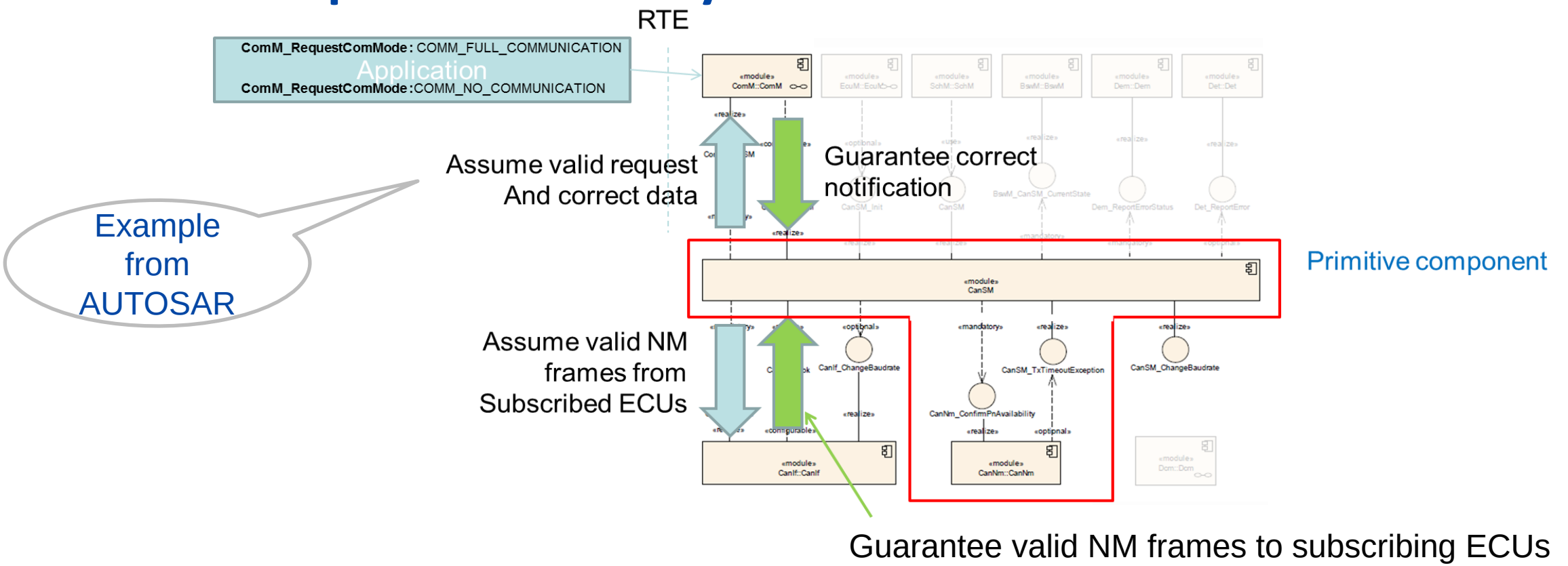
SafetyADD Overview

- Two main parts:
  - *Contract editor*
    - Create safety contract for a software component
  - *Composite component editor*
    - Connect existing components into composite components
    - Check that contracts match within the composite component
- Eclipse based (plugin)
  - Visual editors
  - Can be integrated in development workflow





Primitive Component in SafetyADD

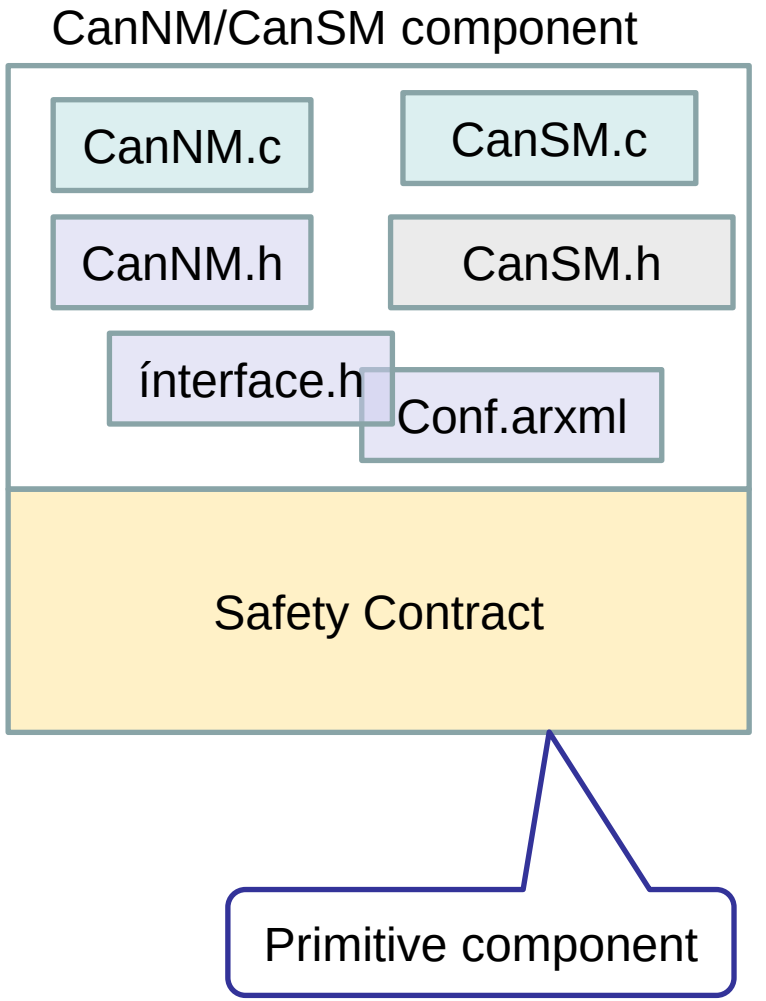


- A piece of software designed for reuse
- Well-defined: Known functionality, dependencies, side-effects, interfaces
- Not necessarily same boundaries as modules in the software architecture

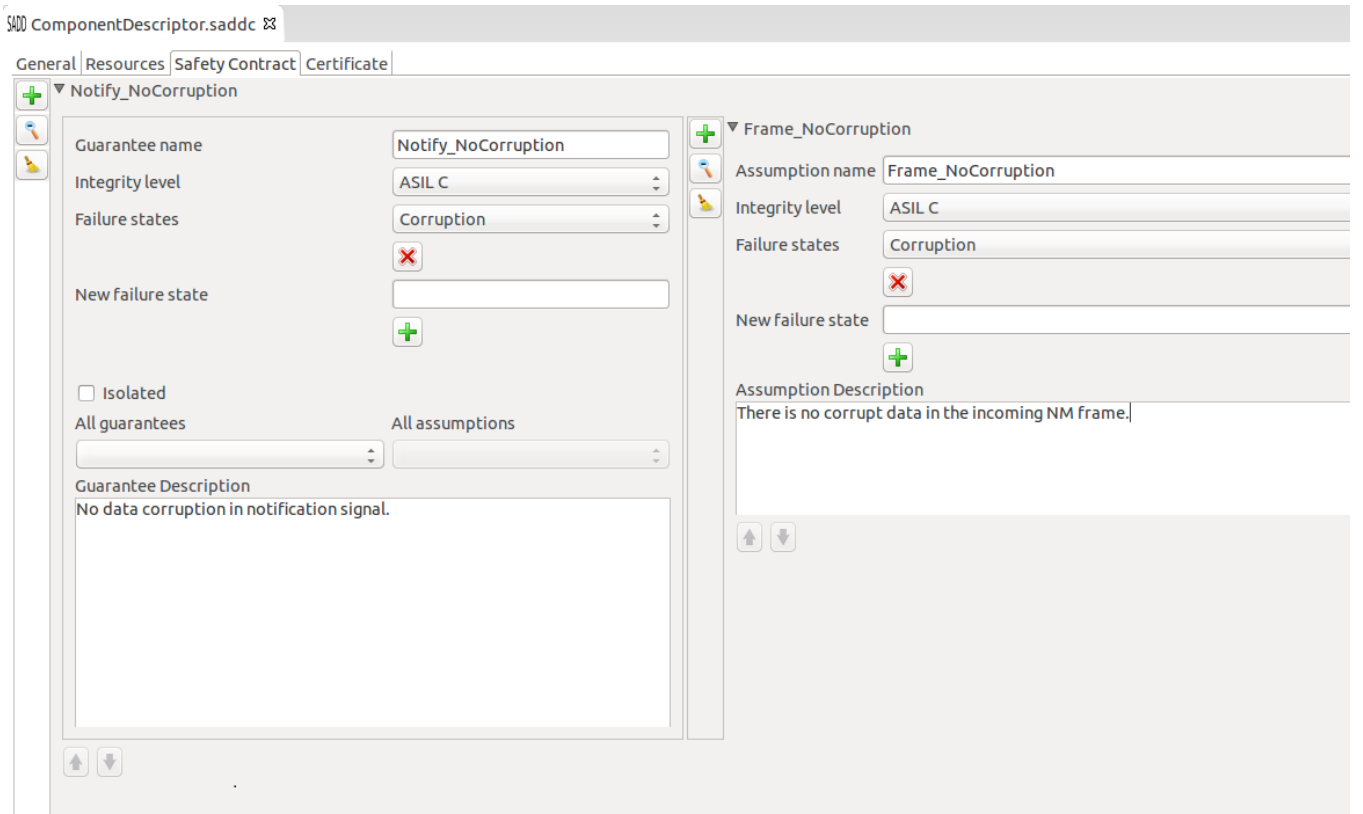
Safety Contract

- Captures safety related properties of the component
  - Safety Element Out of Context (SEooC)
- Not part of the code (stored in an XML format)
  - For future use in pre-certified components: Safety argumentation ties contract to specific version of the code

- Component information
  - Name, resources
- *Guarantees*
  - Safety properties the component is verified to hold given certain assumptions
- *Assumptions*
  - Properties the component expects from the environment in order to uphold its guarantees



Safety Contract example (demo)



Request assume:

- !Omission
- !Commission
- !Corruption

Notification guarantee:

- !Omission
- !Commission
- !Corruption

CanNM/CanSM

Assume received NM frames:

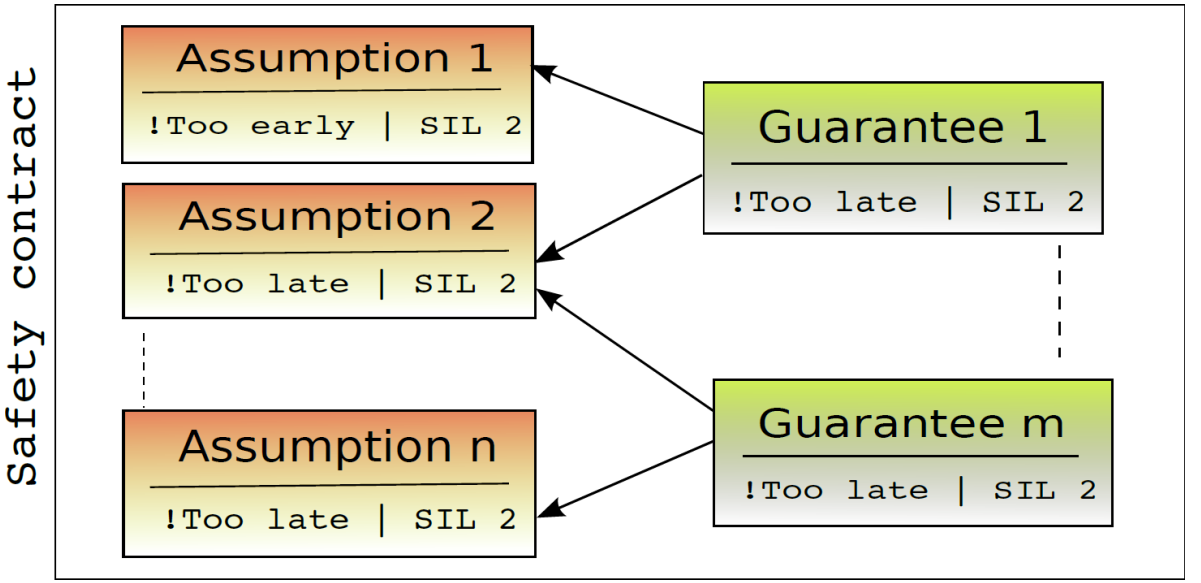
- !Omission
- !Commission
- !Corruption

Guarantee sent NM frames

- !Omission
- !Commission
- !Corruption

Guarantee-Assumption Mapping Within a Contract (demo)

- Guarantee
  - Safety integrity level
  - Failure state
  - Associated with any number of assumptions
- Assumptions
  - Safety integrity level
  - Failure state
- Current limitations
  - Failure state only string matching
  - Several guarantees can not map to same assumption (as in picture)
- Next step
  - Work on failure state expressions



"Guarantee 1 is valid iff assumption 1 and assumption 2 are fulfilled by the environment"

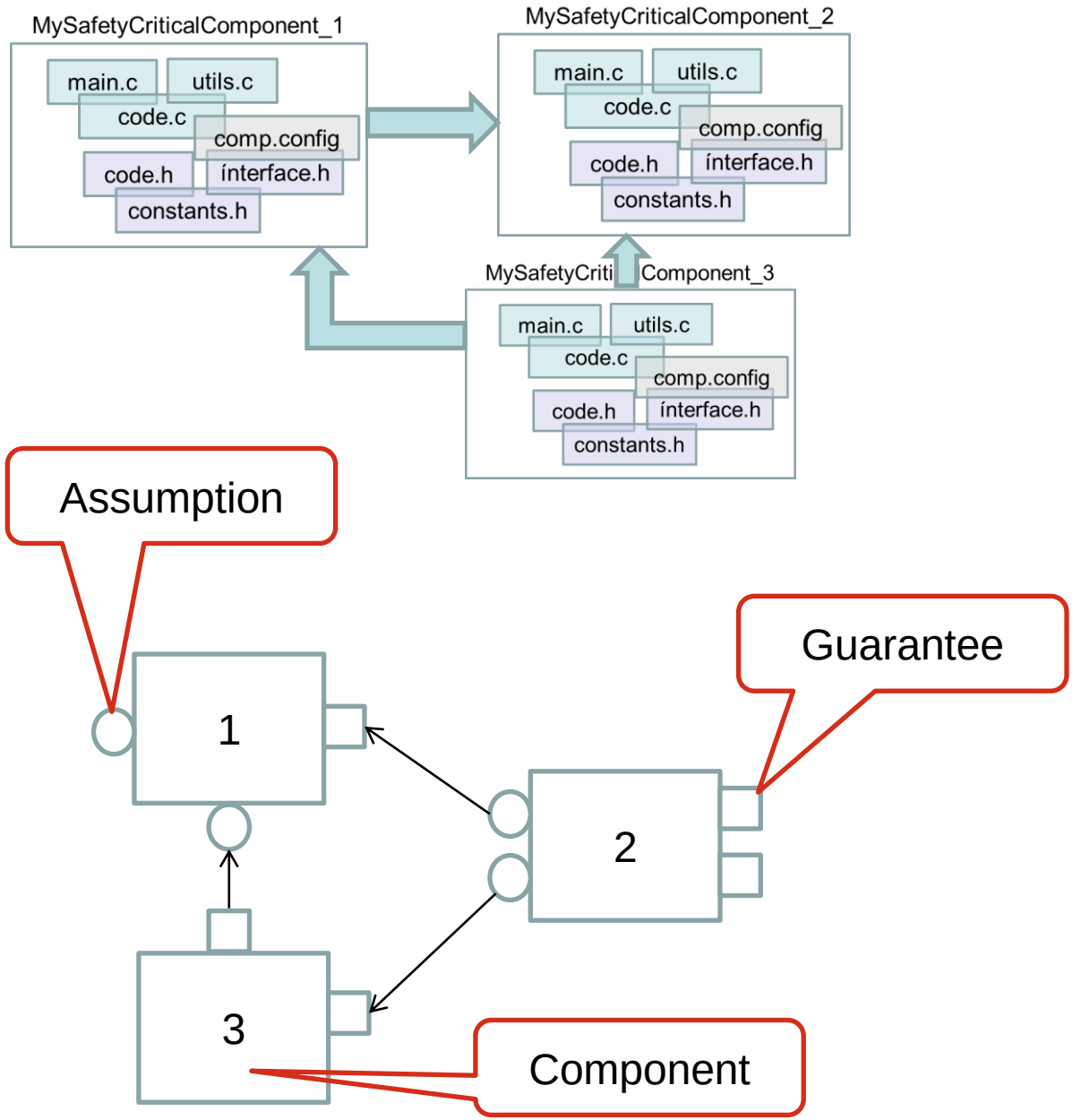
Composite Component

Software Integration

- Several software components are integrated to build a system or subsystem

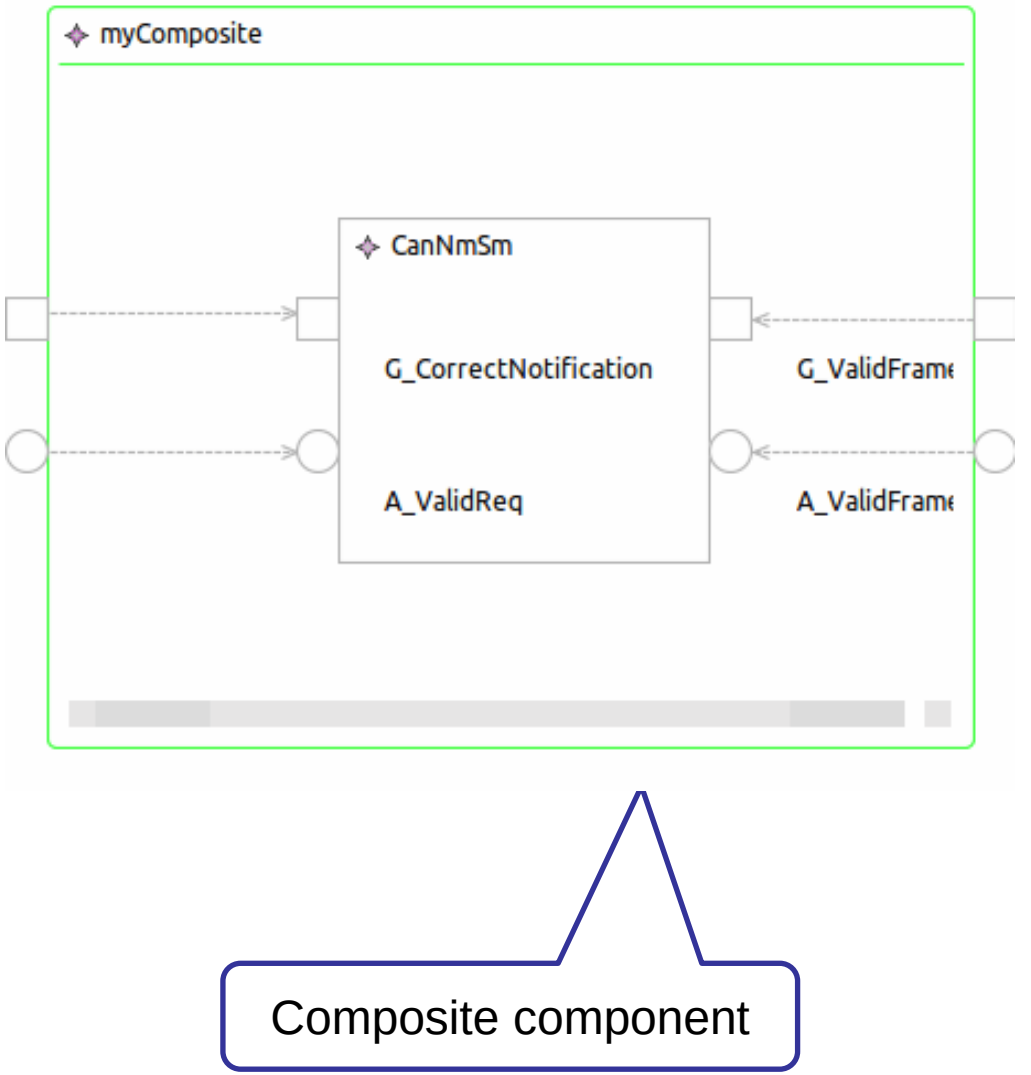
Composite components

- Safety contracts for dependent components must match
- Assumptions connected to matching guarantees
  - Assumption seen as **assignment of verification effort** to another component



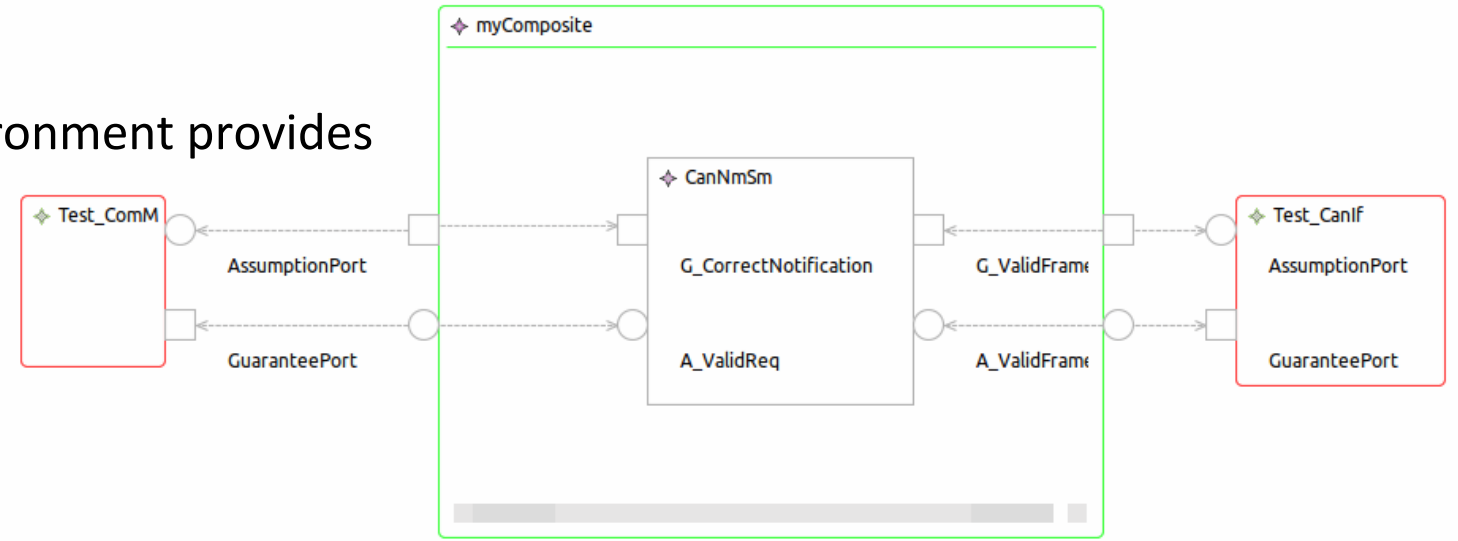
Using the Composite Component Editor (demo)

1. Create a composite component
2. Add primitive components
   - Assumptions/guarantees shown as ports
   - ToDo: Import of primitive components. Support for nested composite components
3. Connect internal assumptions and guarantees
4. Export assumptions/guarantees not consumed within the composite component
   - Ports on the composite component border
   - Delegation connectors
   - Only exported assumptions/guarantees externally visible



Contract Checking in Composite Component Editor

- Checking the validity of connections
  - ASIL and failure state match for all connections
  - No missing connections
- Test case of composite component
  - Test components define boundary for test
    - Represents HW, Application, or external component
- Test components indicate
  - Which guarantees to check
  - Which assumptions the environment provides



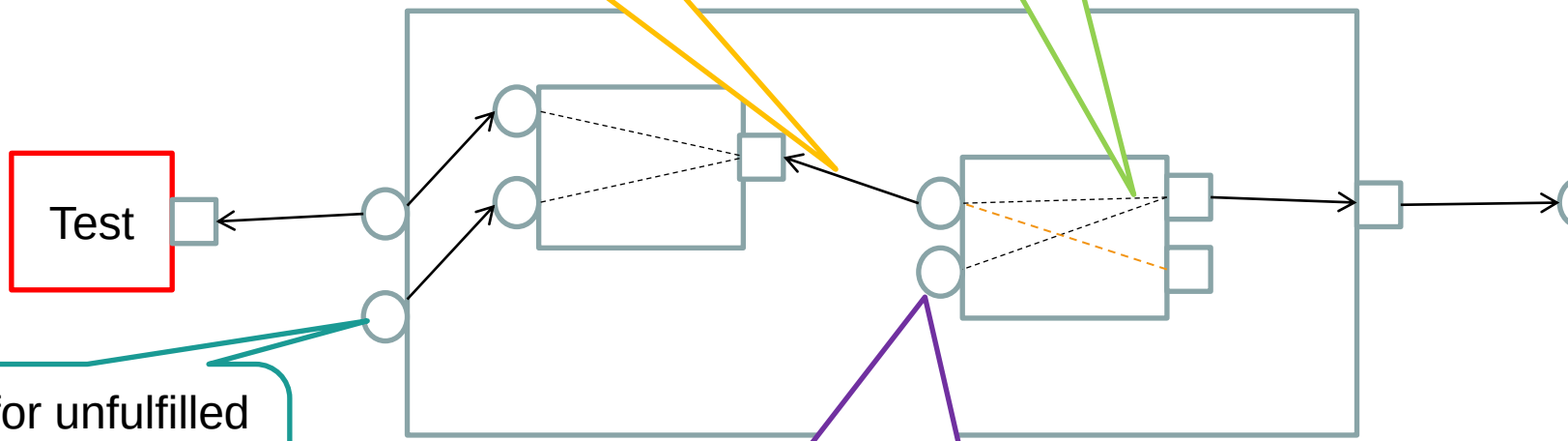
Contract Checking continued…

```
Pseudocode - path traversal algorithm
G = set of <test component guarantees>;
foreach g in G {
  test_guarantee(g);
}

test_guarantee(g) {
  A = set of <assumptions required by> g;
  foreach a in A {
    g_next = <guarantee connected to> a;
    assert(g_next != NULL);
    if(!is_test_component(g_next)) {
      assert(contract_match(a,g_next));
      test_guarantee(g_next);
    }
  }
}
```

Check matching:
- ASIL
- Failure state

Contract shows which assumptions are needed



Check for unfulfilled environment assumptions

Check for unconnected (unfulfilled) internal assumptions

Path traversal starts here!

Summary and Future Work

- Initial goals has been to investigate…
  - Contract validity when changing context for a component
  - How safety contract can be used and managed in the software design and certification process
- Long-term goal
  - Handle safety contracts for pre-certified software components
- Ongoing work and future work ideas…
  - Complete the implementation of current features ☺
  - Refine contract specification (better failure state expression, ASIL decomposition)
  - Include checklist for safety argumentation for ISO 26262
  - Handle nested components
  - Handle configurable components
  - Handling of verification-artefacts and certificate

Questions?