Contract-based design: a temporal logics approach

Alessandro Cimatti

cimatti@fbk.eu

Layered Assurance Workshop, 2013

FONDAZIONE BRUNO KESSLER

ES EMBEDDED SYSTEMS

Credits

- Joint work with
  - Stefano Tonetta
  - Michele Dorigatti
  - Pietro Braghieri
- Supported by the European ARTEMIS SafeCer project.

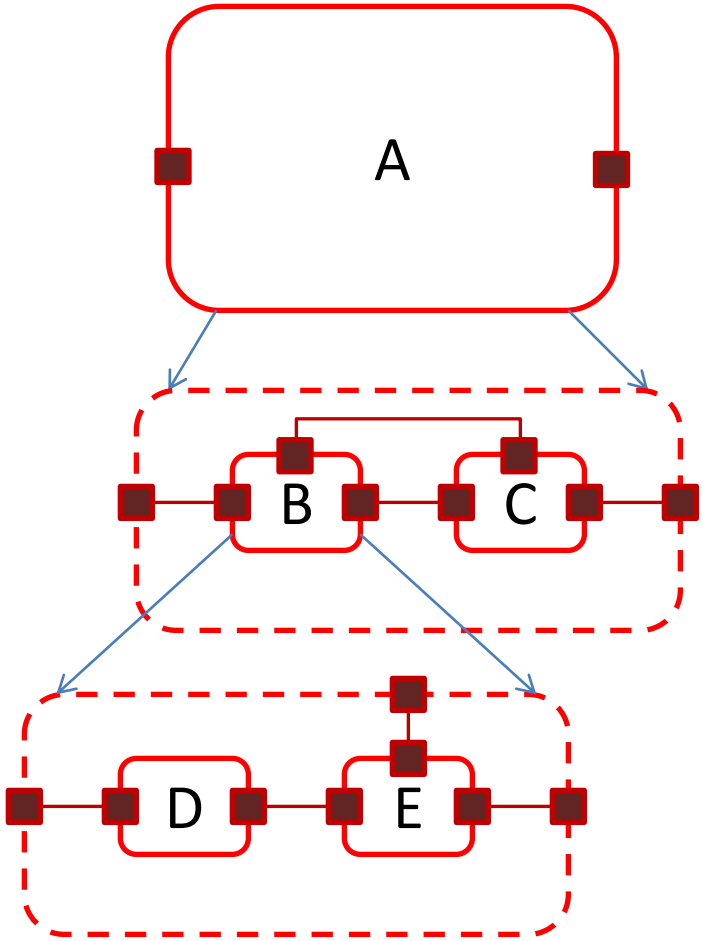
Outline

1. Contract-based design
2. Temporal Logics
3. Contract-based design with temporal logics
4. OCRA: tool support and applications
5. Conclusions and future directions

CONTRACT-BASED DESIGN

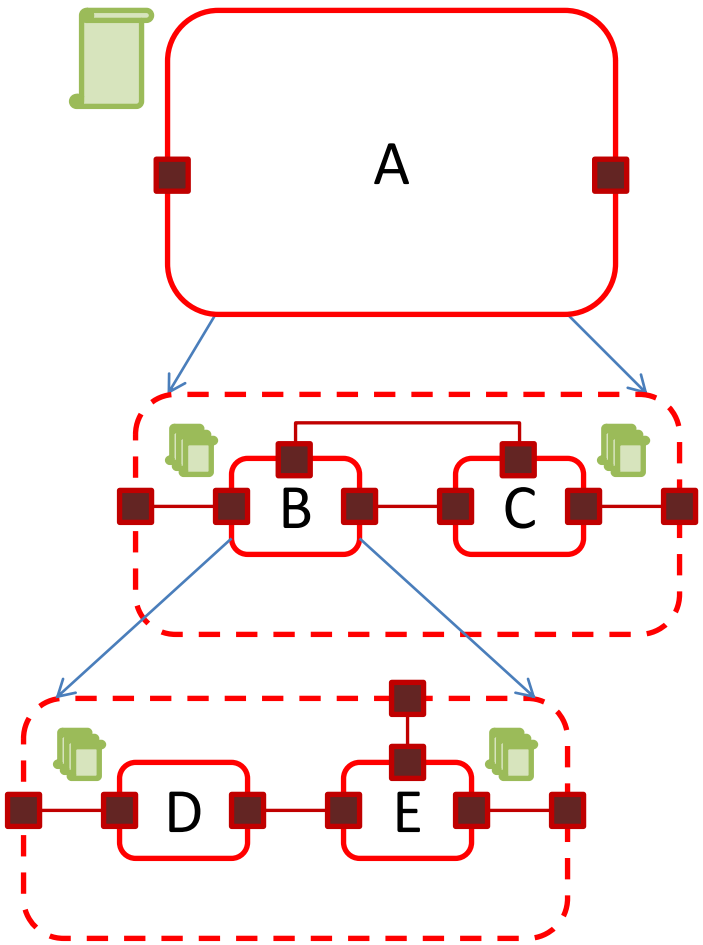
Hierarchical decomposition

- System A
- System A decomposed into subsystems B and C
- Subsystem B decomposed into equipments D and E
- Hierarchical decomposition preserves ports



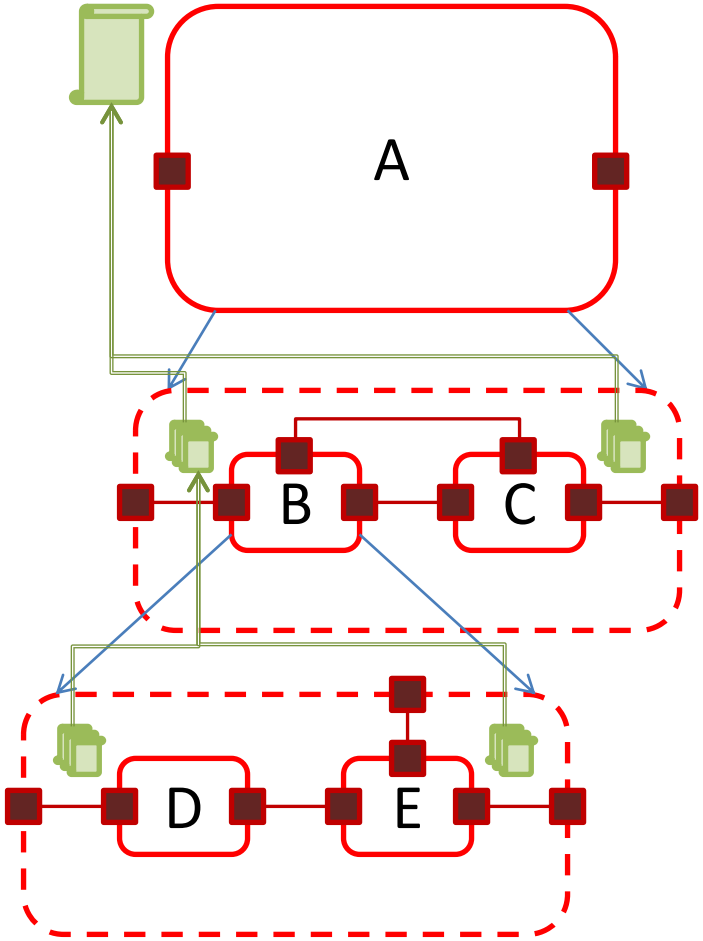
Specifying components with contracts

- A component is immersed in an environment
- Its behaviour is specified by contracts
- Contract: assumptions + guarantees
- Assumptions: what the environment of the component is supposed to do
- Guarantee: what the component shall do



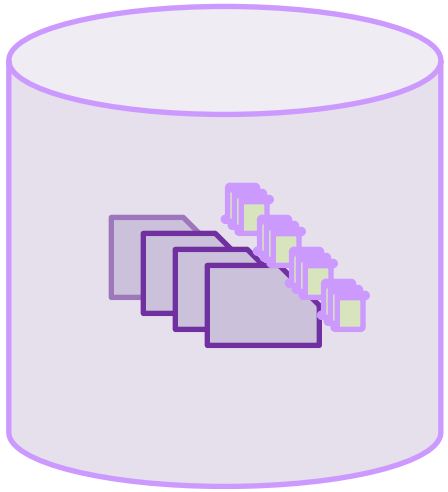
W_{ish}1: early check of requirements

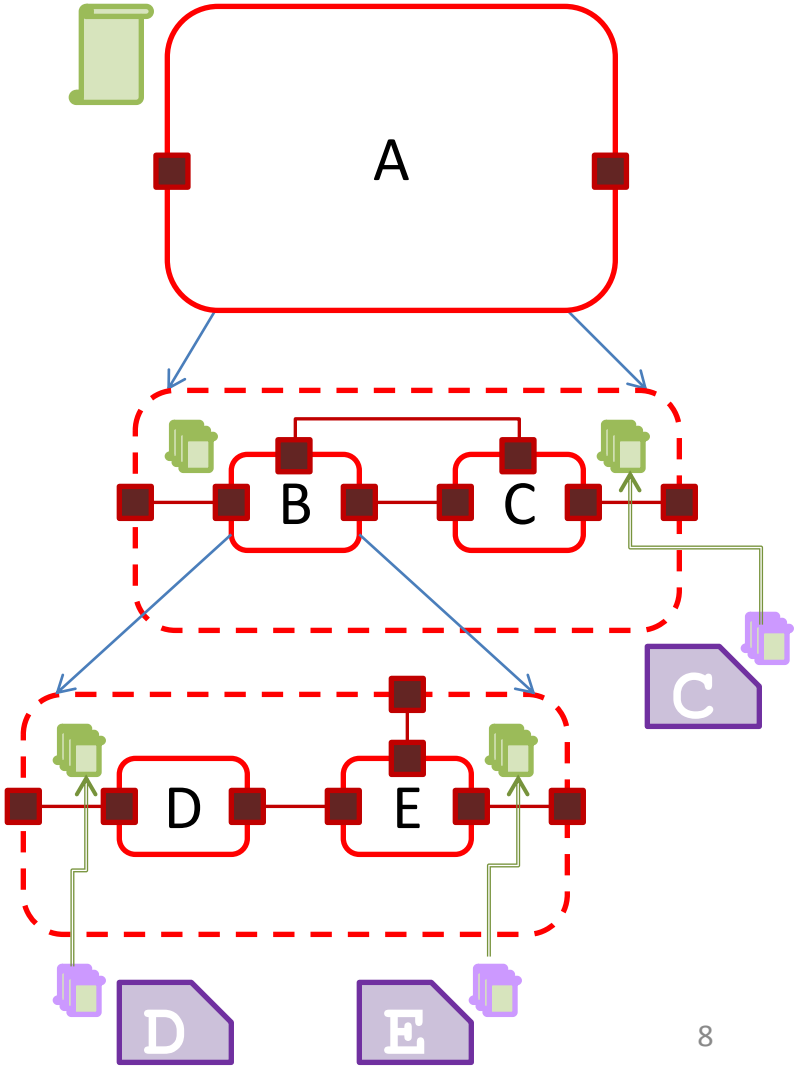
- Specify components while designing
  - decomposing the specification based on the decomposition of the architecture
- Ensure the correctness of the decomposition
  - Does the contract of A follow from the contracts of B and C?



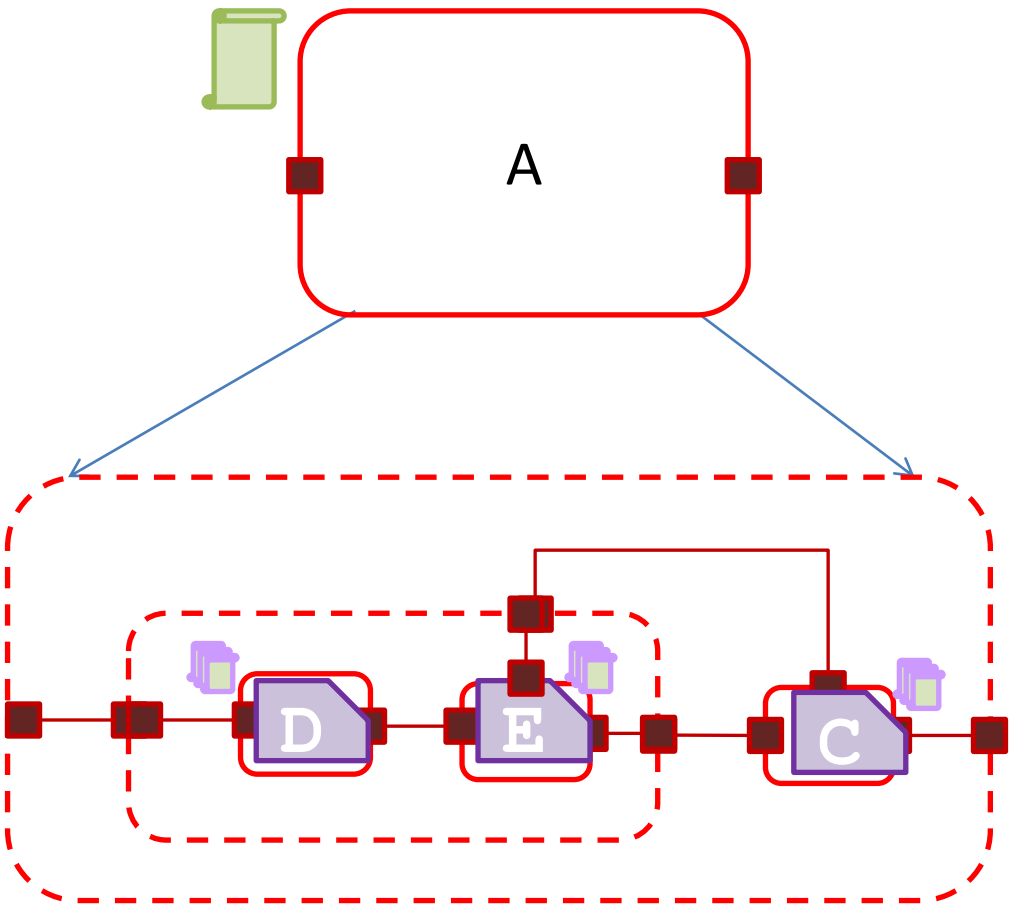
W$_{ish}$2: component reuse

- Library of trusted components
  - e.g. the SRA of the ESA
- Implementation + contracts
- Pluggable?
  - compare contracts!





Correctness by construction!



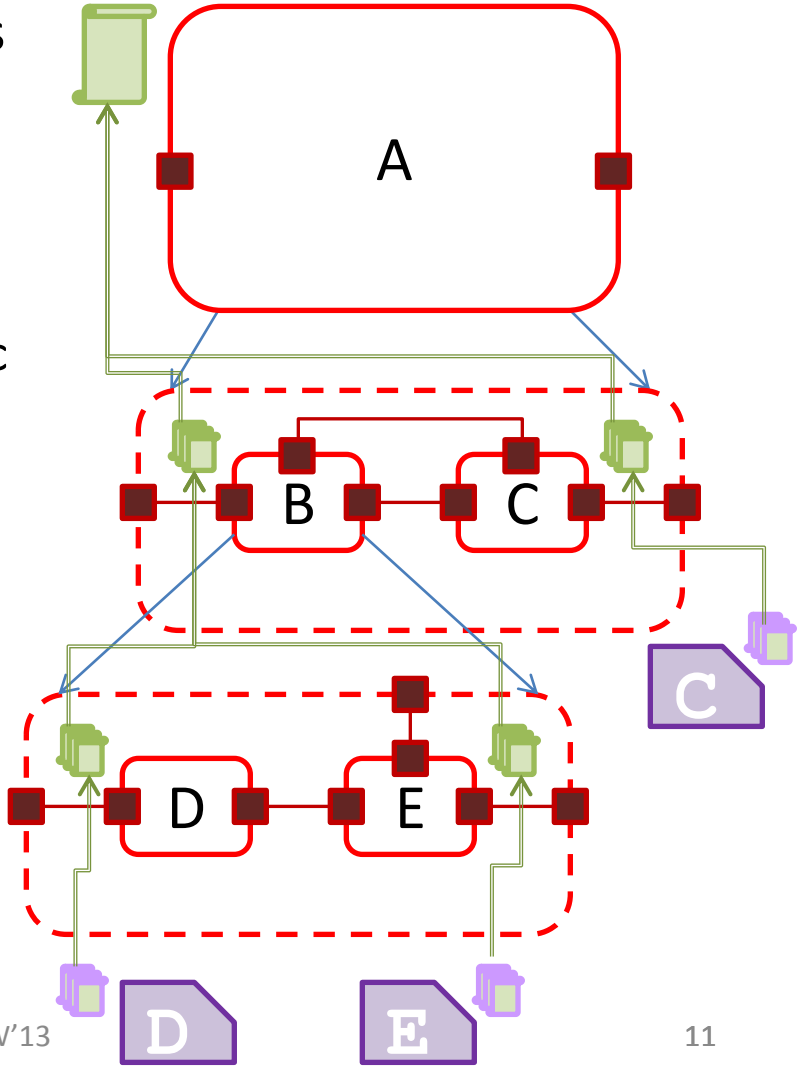
Additional advantages

- Follows structured development process
  - E.g. aerospace domain
- Provides formal support
  - Informal to formal (“semantic gap”)
- Early automated support
  - No detailed behavioural models required
- Leverages decomposition for scalability
  - No monolithic analysis
- Inherently incremental
  - Local changes do not affect whole decomposition

Ingredients for the formal view

- A formal language to specify contracts
  - Temporal logics
- A framework for correct contract refinement
  - Proof obligations
  - Logical consequence of temporal logic formulae

- A formal language to specify implementation
  - Finite state machines
- Checking implementation
  - Model checking



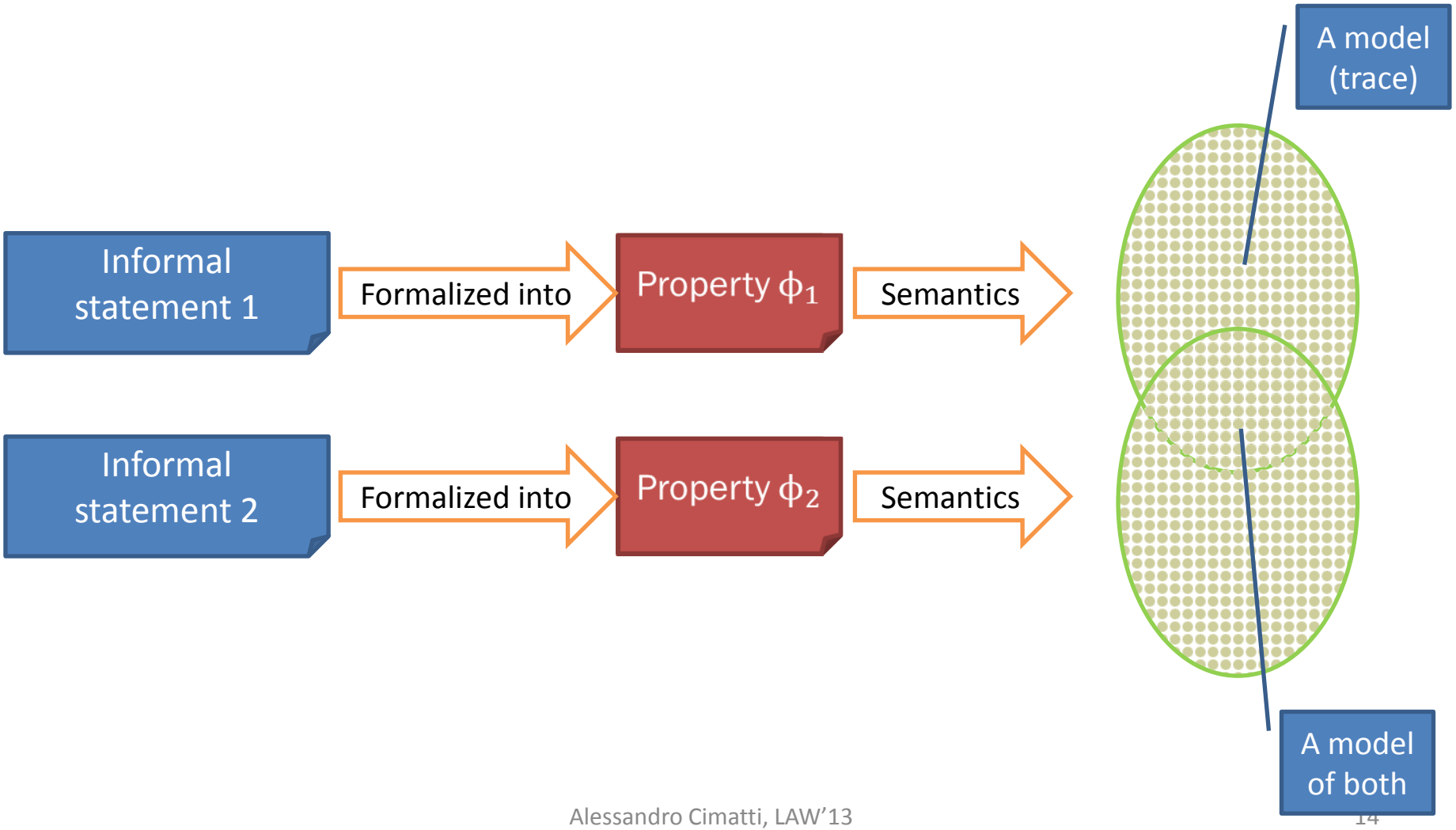
PROPERTY SPECIFICATION LANGUAGES

Properties

- Properties are expressions in a mathematical logic using symbols of the system description.
- Used to formalize requirements.
  - Often closer to informal than behavioural descriptions
- Each property associated with set of system's behavior.

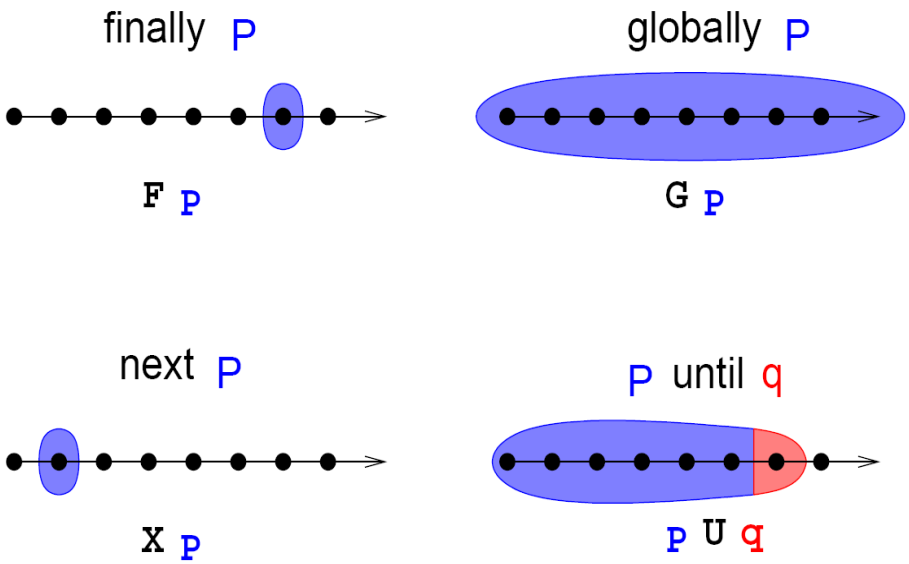
- Problems:
  - Specification: define the properties of a system.
  - Verification: check if the system satisfies the properties.
  - Validation: check if we are considering the right properties.
  - Synthesis: construct a system that satisfies the properties.

Properties, traces, and logic



Linear temporal logic [Pnu77]

- Linear models
  - Traces as sequences of states
- Built over atomic propositions
- Using Boolean connectives
- And temporal operators



LTL examples

- Gp
  - "always p" – invariant
- $G(p \rightarrow Fq)$
  - "p is always followed by q" – reaction
- $G(p \rightarrow Xq)$
  - "whenever p holds, q is set to true in the next cycle"
  - immediate reaction
- GFp
  - "infinitely many times p" – fairness
- FGp
  - "eventually permanently p"
- $G(p \rightarrow (q\ U\ r))$

Simple entailment example

- Given the following formulae
  - G(request $\rightarrow$ F(received))
  - G(received $\rightarrow$ F(processed))
  - G(processed $\rightarrow$ X(grant))
- From which we can entail
  - G(request $\rightarrow$ F(grant))

Past operators

- Y p:
  - “in the previous state p”
  - Dual of next operator X
- O p:
  - “in the past once p”
  - Dual of eventually operator F
- H p:
  - “in the past always p”
  - Dual of always operator G
- p S q (p since q)
  - “in a previous state p, and since then q”
  - Dual of until operator U

Regular expressions

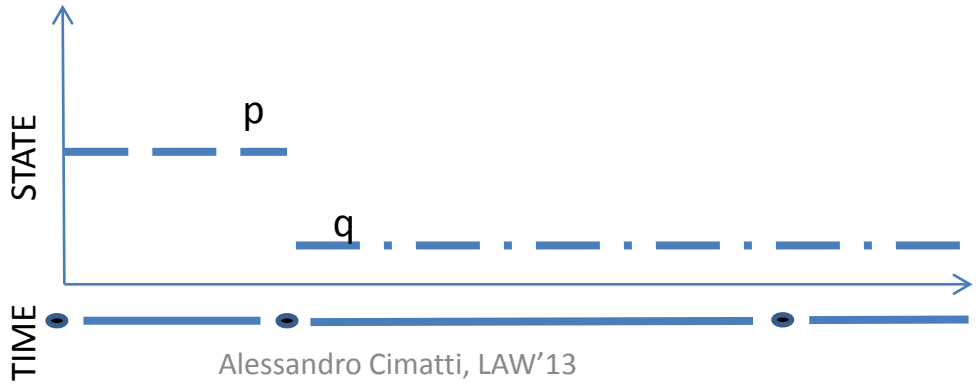
- RELTL enriches LTL with regular expressions:
  - Suffix implication {r} |→ p means that every finite sequence matching r is followed by a suffix matching p
  - Suffix conjunction {r} ◊→ p means that there exists one finite sequence matching r followed by a suffix matching p
- Examples
  - {{¬p}[*];p}[*3] |→ q
  - G({request; busy[*];grant} |→ response)
- Formal basis for PSL hw description language

From finite to infinite

- Use first-order predicates instead of propositions
  - $G(x \geq a \lor x \leq b)$
  - $GF(x = a) \land GF(x=b)$
- Predicates interpreted according to specific theory T (from here on, only reals used)
- Next operator to express changes/transitions:
  - $G(next(x) = x + 1)$
  - $G(next(a) - a \leq b)$

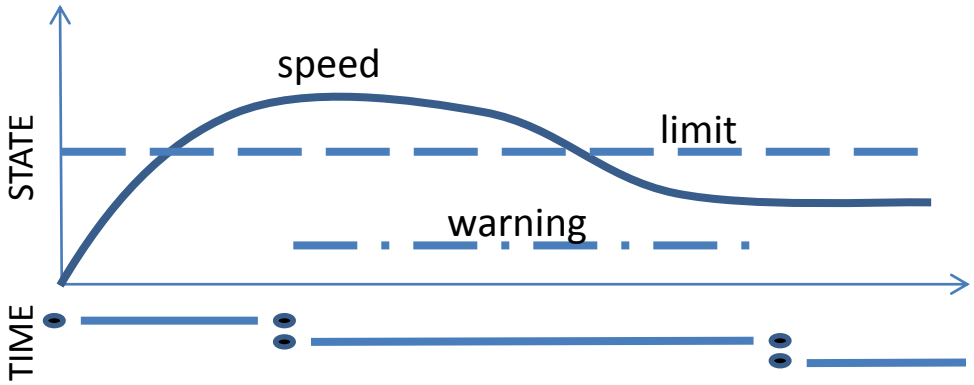
Metric Temporal Logic

- $G(p \longrightarrow F_{\leq 3}\, q)$
  - "p si followed by q within 3 time units"
- $G(p \longrightarrow G_{\leq 20}\, q)$
  - "whenever p holds, then q holds in the following 20 time units"
- $G(p \longrightarrow (\neg q\ U_{\geq 5}\ q))$
  - "p is followed by q but not earlier than 5 time units"



HRELTL: Hybrid RELTL

- G(der(x) < 2)
  - "the derivative of x is always less than 2"
- G(a ⟶ der(x) = 0)
  - "whenever a holds, the derivative of x is 0"
- G(a ⟶ (b U der(x) ≤ 5))
  - "Whenever a holds, b remains true until the der x is less or equal to 5"
- G(speed > limit ⟶ F(warning))



Othello

- Human-readable language for HRELTL
- Controlled natural language expressions
- Developed in the EuRailCheck project
  - Funding by the European Railways Agency
- Formalization and validation of ETCS requirements
  - Requirement “The train trip shall issue an emergency brake command, which shall not be revoked until the train has reached standstill and the driver has acknowledged the trip”
  - formalized as

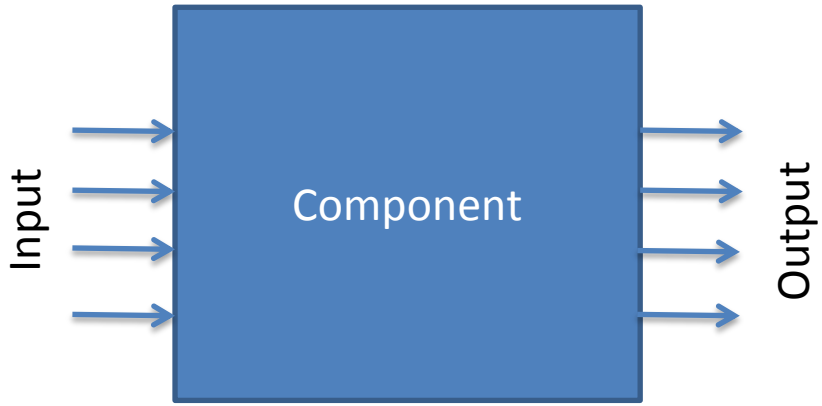
    “always (train_trip implies (emergency_break_command until (der(train_location)=0 and driver_acknowledges_trip)))”
- Validation based on modeling by specification engineers

COMPONENT-BASED DESIGN

Component

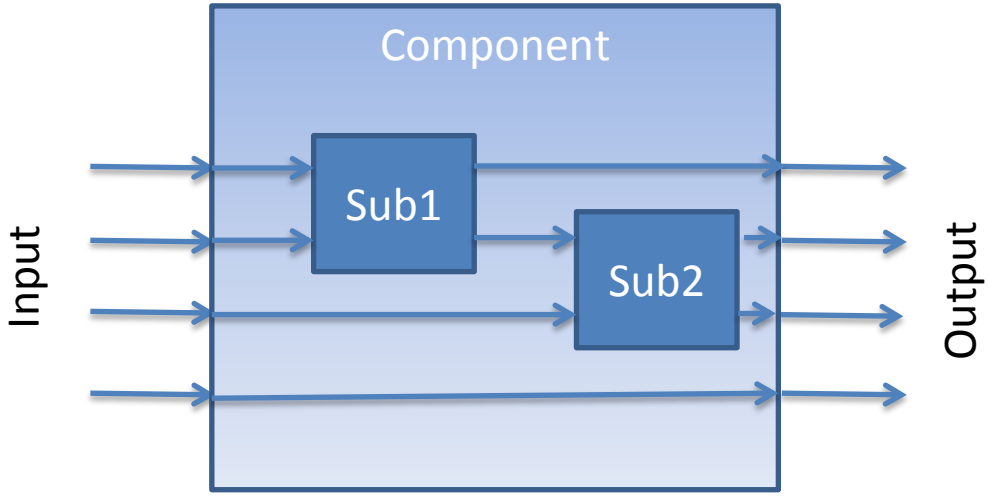
- A component has
  - A syntactic interface
  - Optionally, an internal structure
  - A behavior
  - An environment
  - Properties

Black-box component interface

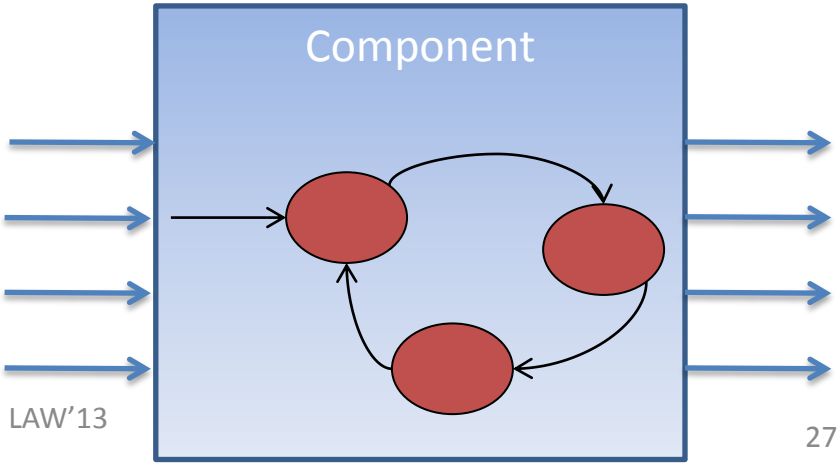


- A component interface defines boundary of the interaction between the component and its environment.
- Consists of:
  - Set of input and output ports (syntax)
    - Ports represent visible data and events exchanged with environment.
  - Set of traces (semantics)
    - Traces represent the behavior, history of events and values on data ports.

Glass-box component structure



- A component has an internal structure.
- Architecture view:
  - Subcomponents
  - Inter-connections
  - Delegations
- State-machine view:
  - Internal state
  - Internal transitions
  - Language over the ports



Component Implementation

- I_S: input ports of component S
- O_S: output ports of S
- $V_S = I_S \cup O_S$: all ports of S
- Tr(X) traces over $X \subseteq V_S$
  - Sequence of assignments to X
- State machine *Imp* implementation of S iff $L(Imp) \subseteq Tr(V_S)$
- M can be associated with $\mu_{Imp} : Tr(I_S) \rightarrow 2^{Tr(O_S)}$ such that $\mu_{Imp}(\sigma_i) = \{\sigma_0 | \sigma_i \times \sigma_0 \in L(Imp)\}$
  - Input trace mapped to a set of output traces
  - Set used to consider nondeterminism
  - Empty set corresponds to rejected input trace

Component Environment

- State machine *Env* environment of S iff L(*Env*) $\subseteq$ Tr(I_S)
- Compatibility of implementation with environment (e.g. for reuse)
  - Trace-based (black-box) view:
    - *Imp* must accept any trace of *Env*
    - L(*Env*) $\subseteq \{\sigma \mid \mu_{Imp}(\sigma) \neq \emptyset\}$
  - State-based (glass-box) view:
    - For any reachable state of *Imp* $\times$ *Env*, for any input transition of *Env*, there exists a matching transition of *Imp*
    - As in interface theory [AH01]
      - note that *Imp* $\times$ *Env* is a closed system

Composite components and connections

- Components are composed to create composite components.
- Different kind of compositions:
  - Synchronous,
  - Asynchronous,
  - Synchronizations:
    - Rendez-vous vs. buffered;
    - Pairwise, multicast, broadcast, multicast with a receiver
- Connections map (general rule of architecture languages):
  - Input ports of the composite component
  - Output ports of the subcomponents

  into
  - Input ports of the subcomponents.
  - Output ports of the composite component

System architecture

- A component is actually a component type.
- A system architecture is an instance of a composite component.
- It defines a tree of component instances.

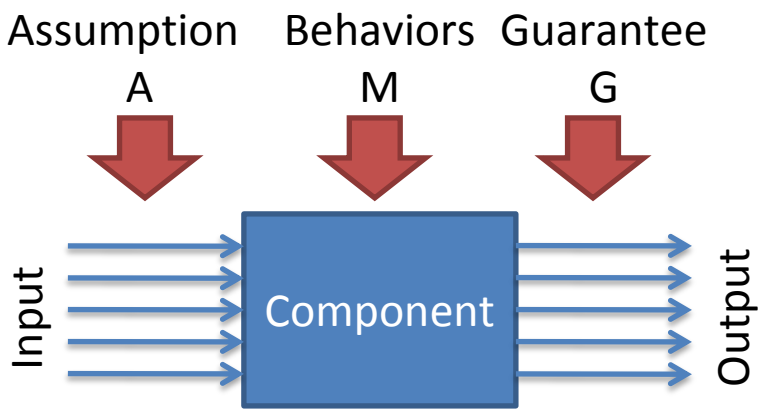
CONTRACT-BASED DESIGN WITH TEMPORAL LOGICS

Contracts

- Properties of the component and its environment.
  - Can be seen as assertions for component interfaces.
- Contracts used to characterize the correctness of component implementations and environments.
- Typically, properties for model checking have a “fully observable” view of the system internals.
- For components instead:
  - Limited to component interfaces.
  - Structure into assumptions and guarantees.
- Contracts for OO programming are pre-/post-conditions [Meyer, 82].
- For systems, assumptions correspond to pre-conditions, guarantees correspond to post-conditions.

Trace-based contracts

- Assertions used to represent sets of traces over the component ports:
  - Φ(V) assertion over variables V
  - $\langle\langle\Phi\rangle\rangle \subseteq$ Tr(V) semantics of Φ
- A contract of component S is a pair <A,G> of assertions over V_S
  - A is the assumption
  - G is the guarantee
- *Env* is a correct environment iff L(*Env*) $\subseteq \langle\langle\Phi\rangle\rangle$
- *Imp* is a correct implementation if L(*Imp*)$\cap\langle\langle$A$\rangle\rangle \subseteq \langle\langle$G$\rangle\rangle$



Example with Othello assertions:

```
assume:
  always (Pedal_Pos1 iff Pedal_Pos2)
guarantee:
  always ( (Pedal_Pos1 or Pedal_Pos2)
   implies (time_until(Brake_Line) <=10 ));
```

Refinement: proof obligations

- Given C=<A,G> contract for component
- Given C_1=<A_1,G_1>, …, <A_n,G_n> contracts for subcomponents
- Proof obligations for "{ C_i } refines C":
  - $\{(A_1 \rightarrow G_1), \ldots, (A_n \rightarrow G_n)\} \models A \rightarrow G$
  - $\{(A_2 \rightarrow G_2), \ldots, (A_n \rightarrow G_n)\} \models A \rightarrow A_1$
  - …
  - $\{(A_1 \rightarrow G_1), \ldots, (A_{i-1} \rightarrow G_{i-1}), (A_{i+1} \rightarrow G_{i+1}), \ldots, (A_n \rightarrow G_n)\} \models A \rightarrow A_i$
  - …
  - $\{(A_1 \rightarrow G_1), \ldots, (A_{n-1} \rightarrow G_{n-1})\} \models A \rightarrow A_n$

What does it mean?

- Focus on properties of father component
- $\{(A_1 \rightarrow G_1), \ldots, (A_n \rightarrow G_n)\} \models A \rightarrow G$
- The contract of the father component $A \rightarrow G$ must follow from the contracts of the subcomponents

- Alternative view:
  $\{(A_1 \rightarrow G_1), \ldots, (A_n \rightarrow G_n), A\} \models G$

What does it mean?

- Focus on i-th subcomponent
- $\{(A_1 \rightarrow G_1), \ldots, (A_{i-1} \rightarrow G_{i-1}),$
  $(A_{i+1} \rightarrow G_{i+1}), \ldots, (A_n \rightarrow G_n)\} \models A \rightarrow A_i$
- The assumptions of the i-th subcomponent must follow from the contracts of the other subcomponents plus the assumptions of the father component

Proof obligations

- PO's necessary and sufficient for correct contract refinement [CT12]
- Extension to deal with asynchronous composition

- Key issue: diagnostic information!
  - In case of violation, trace
  - Localization by means of proof-based methods
    - unsat core extraction

Weak vs. strong assumptions

- Weak vs. strong assumptions (both important):
  - Weak assumptions
    - Define the context in which the guarantee is ensured
    - As in assume-guarantee reasoning
    - Different assume-guarantee pairs may have inconsistent assumptions (if x>0 then …, if x<0 then …)
  - Strong assumptions
    - Define properties that must be satisfied by the environment.
    - Original idea of contract-based design.
    - If not satisfied, the environment can cause a failure (division by zero, out of power, collision).

Assume-guarantee reasoning

- Corresponds to one direction of the contract refinement
- Many works focus on finding the right assumption/guarantee
- E.g. how to break circularity

$$(G(a \rightarrow b) \wedge G(b \rightarrow a)) \Rightarrow G(a \wedge b)$$

- Induction-based mechanisms

$$(b \wedge G(a \rightarrow Xb) \wedge a \wedge G(b \rightarrow Xa)) \Rightarrow G(a \wedge b)$$

- Note they are structural ways to prove the property-based refinement

OCRA TOOL SUPPORT

OCRA tool support

- OCRA=Othello Contract Refinement Analysis [CDT13]
- Contracts' assertions specified in Othello.
- Textual representation of the architecture.
- Built on top of nuXmv for infinite-state model checking.
- Integrated with CASE tools:
  - AutoFocus3
    - Developed by Fortiss.
    - For synchronous system architectures.
  - CHESS
    - Developed by Intecs.
    - For SysML and UML modeling.
- One of the few tools supporting contract-based design for embedded systems.
- Publicly available (for non-commercial purposes) at
  https://es.fbk.eu/tools/ocra

OCRA main features

- Rich component interfaces to specify:
  - Input/output ports
  - Data/Event ports.
  - Including real-time and safety aspects.
- Contracts in temporal logics.
- Temporal formulas used to characterize set of traces over the ports of components.

OCRA language

```
COMPONENT system
…

COMPONENT A
…

COMPONENT B
…
```

Component interface

```
COMPONENT system
        INTERFACE
                INPUT PORT x: continuous;
                OUTPUT PORT a: boolean;
        …
        REFINEMENT
        …

COMPONENT A
…

COMPONENT B
…
```

Othello contracts

```
COMPONENT simple system
        INTERFACE
                INPUT PORT x: continuous;
                OUTPUT PORT v: boolean;

                CONTRACT v_correct
                        assume: always x>=0;
                        guarantee: always (x=0 implies v);

        REFINEMENT
        …

COMPONENT A
…

COMPONENT B
…
```

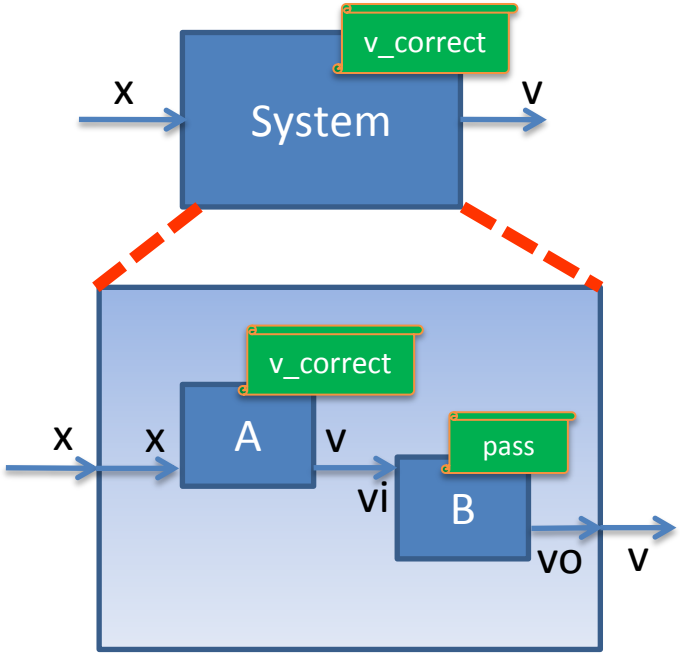
Component refinement

```
COMPONENT simple system
        INTERFACE
                INPUT PORT x: continuous;
                OUTPUT PORT v: boolean;

                CONTRACT v_correct
                        assume: always x>=0;
                        guarantee: always (x=0 implies v);

        REFINEMENT
                SUB a: A;
                SUB b: B;

                CONNECTION a.x := x;
                CONNECTION b.yi := a.v;
                CONNECTION v:= b.vo;
        …
```

Contract refinement

```
COMPONENT simple system
        INTERFACE
                INPUT PORT x: continuous;
                OUTPUT PORT v: boolean;

                CONTRACT v_correct
                        assume: always x>=0;
                        guarantee: always (x=0 implies v);

        REFINEMENT
                SUB a: A;
                SUB b: B;

                CONNECTION a.x := x;
                CONNECTION b.vi := a.v;
                CONNECTION v:= b.vo;

                CONTRACT v_correct REFINEDBY a.v_correct, b.pass;
```

Complete example



The complete example

```
COMPONENT simple system
 INTERFACE
  INPUT PORT x: continuous;
  OUTPUT PORT v: boolean;

  CONTRACT v_correct
   assume: always (x>=0);
   guarantee: always (x=0 implies v);

 REFINEMENT
  SUB a: A;
  SUB b: B;

  CONNECTION a.x := x;
  CONNECTION b.vi := a.v;
  CONNECTION v:= b.vo;

  CONTRACT v_correct REFINEDBY a.v_correct, b.pass;
```

```
COMPONENT A
 INTERFACE
  INPUT PORT x: continuous;
  OUTPUT PORT v: boolean;

  CONTRACT v_correct
   assume: always (x>=0);
   guarantee: always (x=0 implies v);
```

```
COMPONENT B
 INTERFACE
  INPUT PORT vi: boolean;
  OUTPUT PORT vo: boolean;

  CONTRACT pass
   assume: true;
   guarantee: always (vi implies vo);
```

OCRA: a parameterized view

- The flow implemented in OCRA is parametric
  - Multiple logics are supported
  - Traces can be discrete or hybrid

- Required functionalities
  - Logical entailment

OCRA hybrid aspects

- Port types are either
  - Finite types: “boolean”, enumeratives, ...
  - Infinite types: “real”, “integer”, …
  - “continuous”: real-value ports constrained to evolve continuously in time
  - “event”: boolean-value ports assigned only on discrete transitions
- Atomic formulas may be:
  - Boolean variables.
  - Equalities.
  - Arithmetic predicates over integer, real, and continuous terms.
- Temporal operators: as in LTL, and HRELTL

Functionalities

- Syntax checking
- Refinement checking
  - Generate all po's and run entailment checkers
- Consistency checking
  - Are there inconsistent assumptions/guarantees
- Checking implementation
  - Does component implementation satisfy contracts?
- Checking receptiveness
  - Does component implementation able to react to every input of the environment?

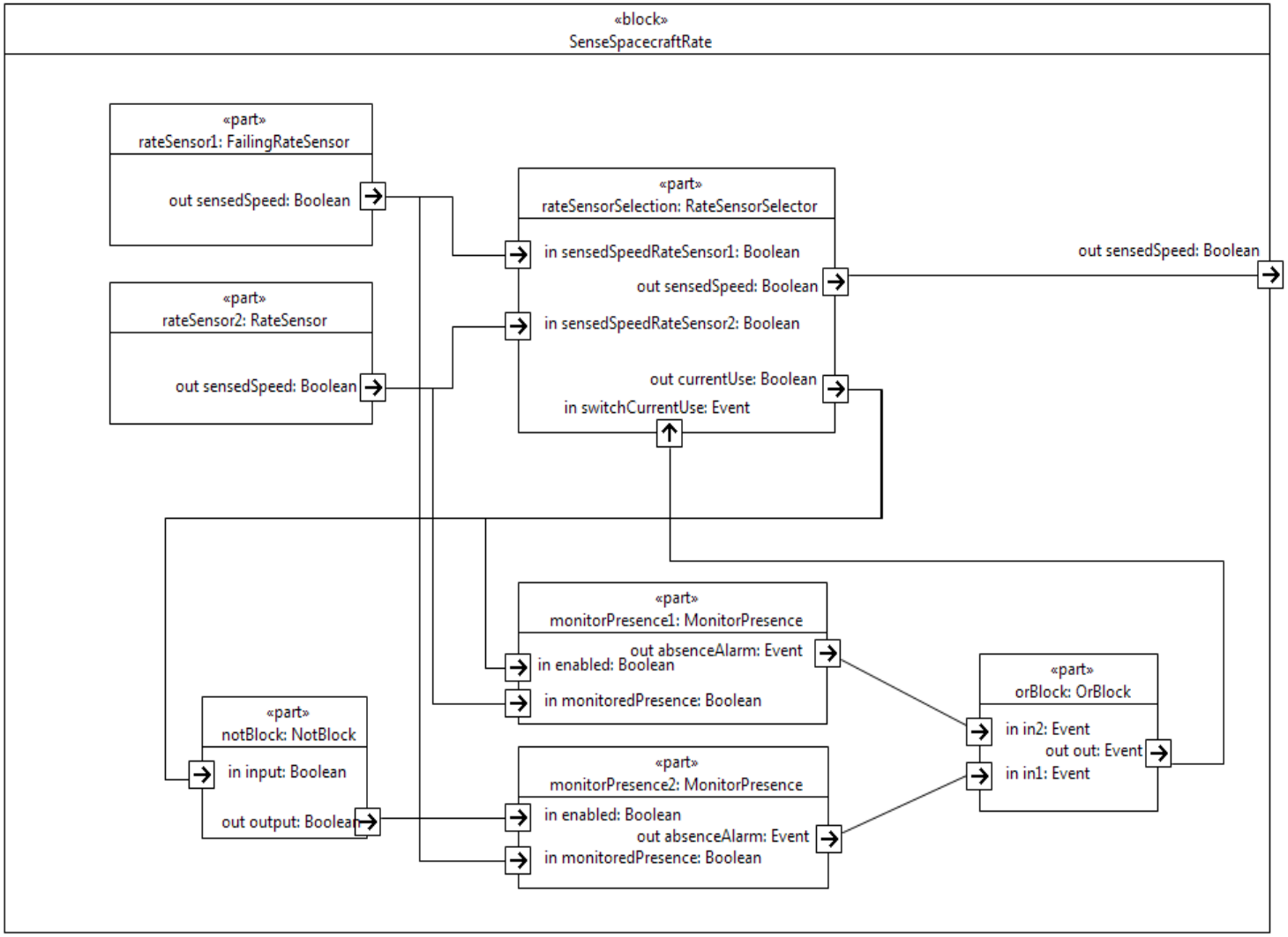
Contract refinement results

- For every component, for every refined contract, check refinement.
- For every proof obligation, check its validity:
  - [OK] if valid
  - [BOUND OK] if no counterexample found up to k
  - [FAIL] if found counterexample

Application in the Forever project

- OCRA integrated within CHESS design environment
- Various applications developed by industrial project partners
  - Thales-Alenia Space engineers

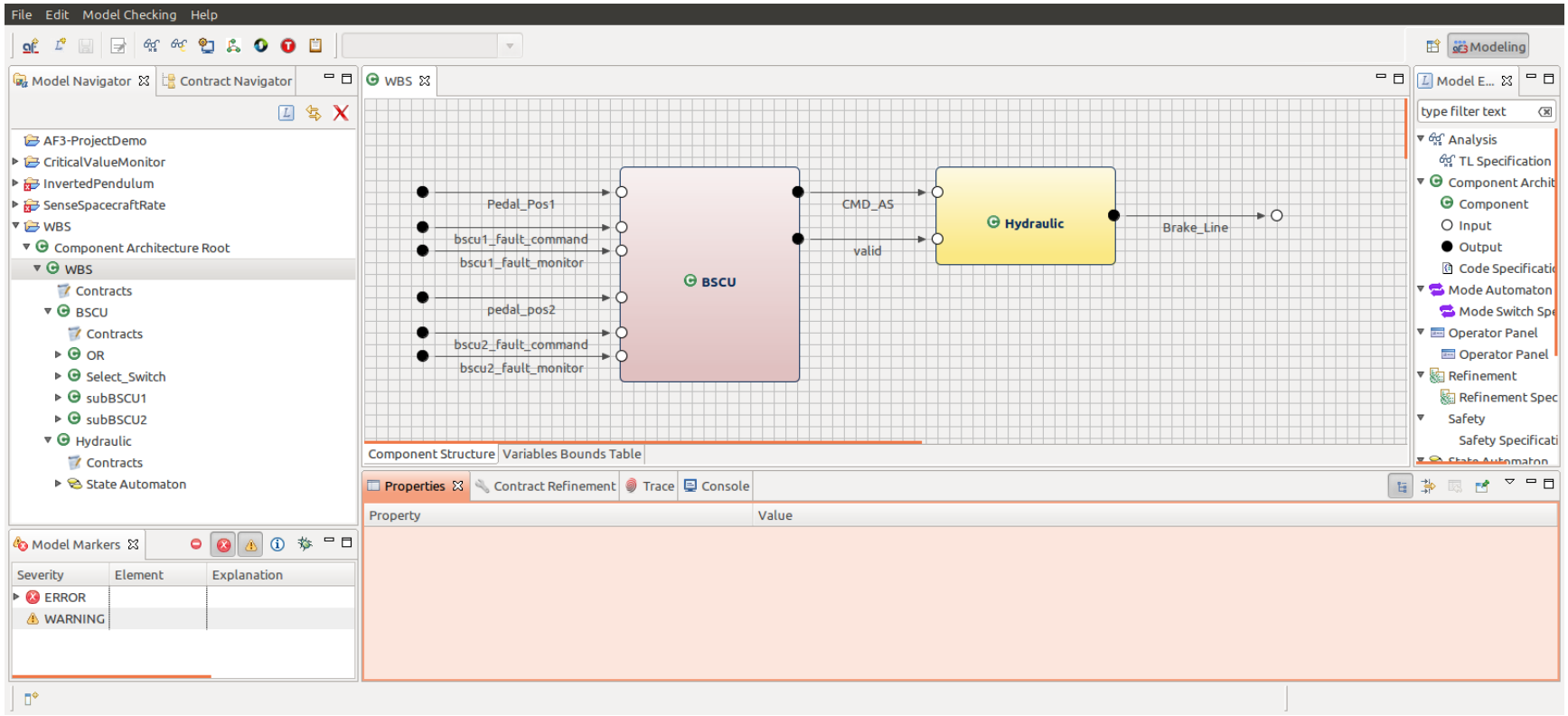
SenseSpacecraftRate Example



The complete example

```
COMPONENT SenseSpacecraftSpeed system
 INTERFACE
  INPUT PORT speed: continuous;
  OUTPUT PORT sensed_speed: real;
  OUTPUT PORT sensed_speed_is_present: boolean;

  CONTRACT sense
   assume: TRUE;
   guarantee:
    always ((sensed_speed - speed <= error) and
              (sensed_speed - speed >= - error) and
              sensed_speed_is_present);

 REFINEMENT
  SUB sensor1: SpeedSensor;
  SUB sensor2: SpeedSensor;
  SUB monitor1: MonitorPresence;
  SUB monitor2: MonitorPresence;
  SUB selector: Selector;

  CONNECTION sensor1.speed := speed;
  CONNECTION sensor2.speed := speed;
  CONNECTION monitor1.input_is_present := sensor1.sensed_speed_is_present;
  CONNECTION monitor2.input_is_present := sensor2.sensed_speed_is_present;
  CONNECTION monitor1.enabled := (selector.current_use=1);
  CONNECTION monitor2.enabled := (selector.current_use=2);
  CONNECTION selector.input1 := sensor1.sensed_speed;
  CONNECTION selector.input1_is_present := sensor1.sensed_speed_is_present;
  CONNECTION selector.input2 := sensor2.sensed_speed;
  CONNECTION selector.input2_is_present := sensor2.sensed_speed_is_present;
  CONNECTION selector.switch_current_use := monitor1.absence_alarm or  monitor2.absence_alarm;
  CONNECTION sensed_speed := selector.output;
  CONNECTION sensed_speed_is_present := selector.output_is_present;

  CONTRACT sense REFINEDBY sensor1.sense, sensor2.sense, monitor1.monitor, monitor2.monitor,
            selector.select, selector.switch;

COMPONENT SpeedSensor
 INTERFACE
  INPUT PORT speed: continuous;
  OUTPUT PORT sensed_speed: real;
  OUTPUT PORT sensed_speed_is_present: boolean;

CONTRACT sense
  assume: TRUE;
  guarantee: always ((sensed_speed - speed <= error) and
                              (sensed_speed - speed >= - error) and
                              sensed_speed_is_present);


COMPONENT MonitorPresence
 INTERFACE
  INPUT PORT input_is_present: boolean;
  INPUT PORT enabled: boolean;
  OUTPUT PORT absence_alarm: event;

  CONTRACT monitor
   assume: true;
   guarantee: always (absence_alarm iff (enabled and fall(input_is_present)));


COMPONENT Selector
 INTERFACE
  INPUT PORT input1: real;
  INPUT PORT input1_is_present: boolean;
  INPUT PORT input2: real;
  INPUT PORT input2_is_present: boolean;
  INPUT PORT switch_current_use: boolean;
  OUTPUT PORT current_use: {1,2};
  OUTPUT PORT output: real;
  OUTPUT PORT output_is_present: boolean;

  CONTRACT select
   assume: true;
   guarantee:
    always ((current_use=1 implies
                (output=input1 and output_is_present=input1_is_present)) and
               (current_use=2 implies
                (output=input2 and output_is_present=input2_is_present)));

  CONTRACT switch
   assume: true;
   guarantee: always (
    ((current_use=1 and switch_current_use) implies next(current_use)=2) and
    ((current_use=2 and switch_current_use) implies next(current_use)=1) and
    ((not switch_current_use) implies not change(current_use)));
```

Plugin for AutoFocus (fortiss)



Plugin for AutoFocus

File Edit Model Checking Help

Model Navigator | Contract Navigator

- AF3-ProjectDemo
- CriticalValueMonitor
- InvertedPendulum
- SenseSpacecraftRate
- WBS
  - Component Architecture Root
    - WBS
      - Contracts
      - BSCU
        - Contracts
        - OR
        - Select_Switch
        - subBSCU1
        - subBSCU2
      - Hydraulic
        - Contracts
        - State Automaton

WBS | WBS

```
CONTRACT brake_time
assume:
  always (Pedal_Pos1=Pedal_Pos2) and
  -- no double fault
  (always ( (not bscu1_fault_Monitor) and
                (not bscu1_fault_Command) and
                     (not bscu2_fault_Monitor) ) or
   always ( (not bscu1_fault_Monitor) and
                (not bscu1_fault_Command) and
                     (not bscu2_fault_Command) ) or
   always ( (not bscu1_fault_Monitor) and
                (not bscu2_fault_Command) and
                     (not bscu2_fault_Monitor) ) or
   always ( (not bscu1_fault_Command) and
                (not bscu2_fault_Command) and
                     (not bscu2_fault_Monitor) ) );

guarantee:
  always ( (change(Pedal_Pos1) or change(Pedal_Pos2)) implies
           (in the future change(Brake_Line) ));
```

Model Markers

Severity	Element	Explanation
ERROR		
WARNING		

Properties | Contract Refinement | Trace | Console

Property	Value

Writable | Insert | 1 : 1

Related work

- Basic concepts on contract-based design for embedded systems:
  - Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple Viewpoint Contract-Based Specification and Design. *FMCO 2007.*
  - Manfred Broy: Towards a Theory of Architectural Contracts: - Schemes and Patterns of Assumption/Promise Based System Specification. Software and Systems Safety - Specification and Verification 2011: 33-87
  - Alberto Sangiovanni-Vincentelli, Werner Damm and Roberto Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. European Journal of Control, 18(3):217-238, 2012.
  - Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. Contracts for Systems Design. Rapport de recherche RR-8147, INRIA, Nov. 2012.
  - S.S. Bauer, A. David, R. Hennicker, K.G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from Specifications to Contracts in Component-Based Design. In FASE, pages 43–58, 2012.

- META program and AGREE tool by D. Cofer and colleagues.
  - Also on system architecture with temporal logics
  - Focus on discrete-time, and assume-guarantee reasoning (weak assumptions).

Summary

- Contract-based design powerful
  - For stepwise refinement
  - Compositional reasoning
- Temporal logic is suitable for component contracts
- Contract framework parameterized by logic
- SMT-based model checking used to reason with expressive properties
- OCRA tool support

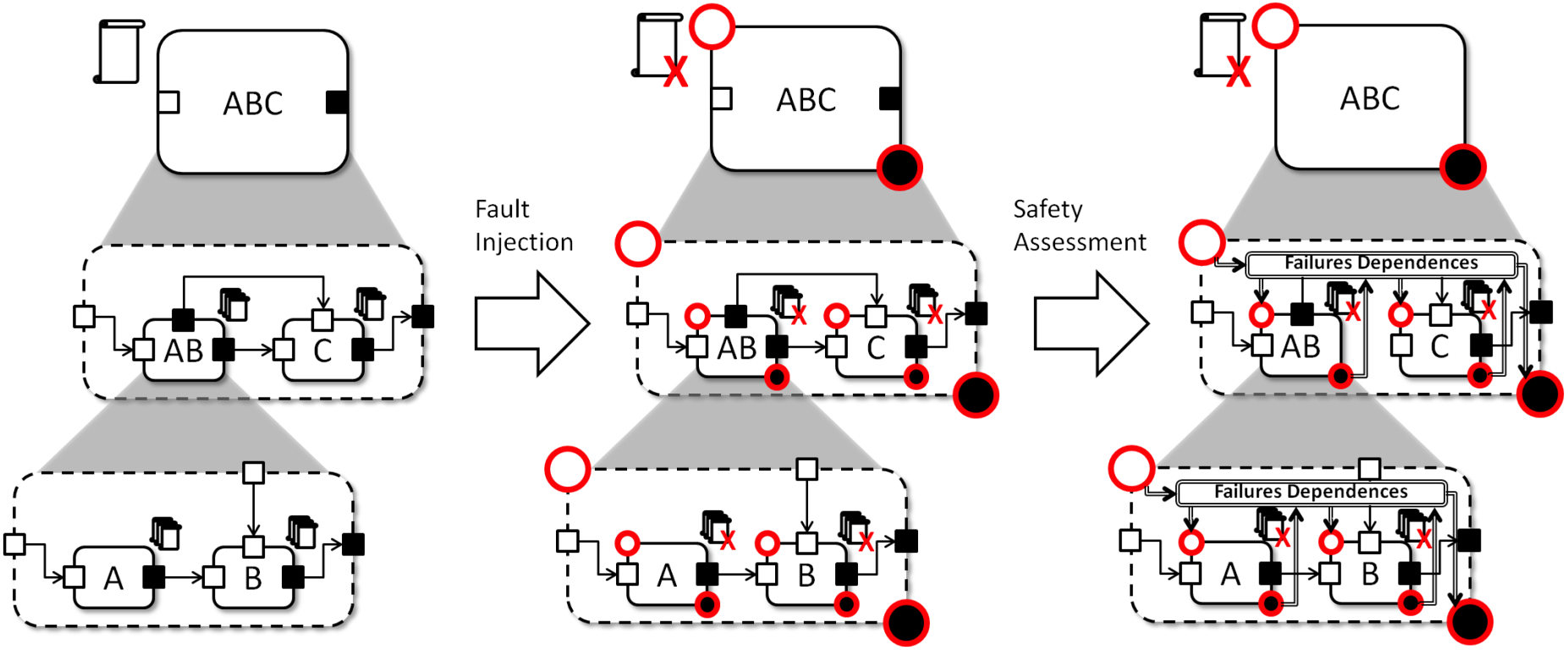
Model-based safety assessment

- Safety assessment
  - Analyze behaviour of system under faults
  - Artifacts: Fault Trees, FMEA tables
- Model-based Safety Assessment
  - Extend nominal model with faults
    - Symbolic fault injection
  - Analyze extended model
- Key issues
  - Requires behavioural models, comes too late
  - Requires model of faults, requires human intervention
  - No use of hierarchical decomposition, flat fault trees

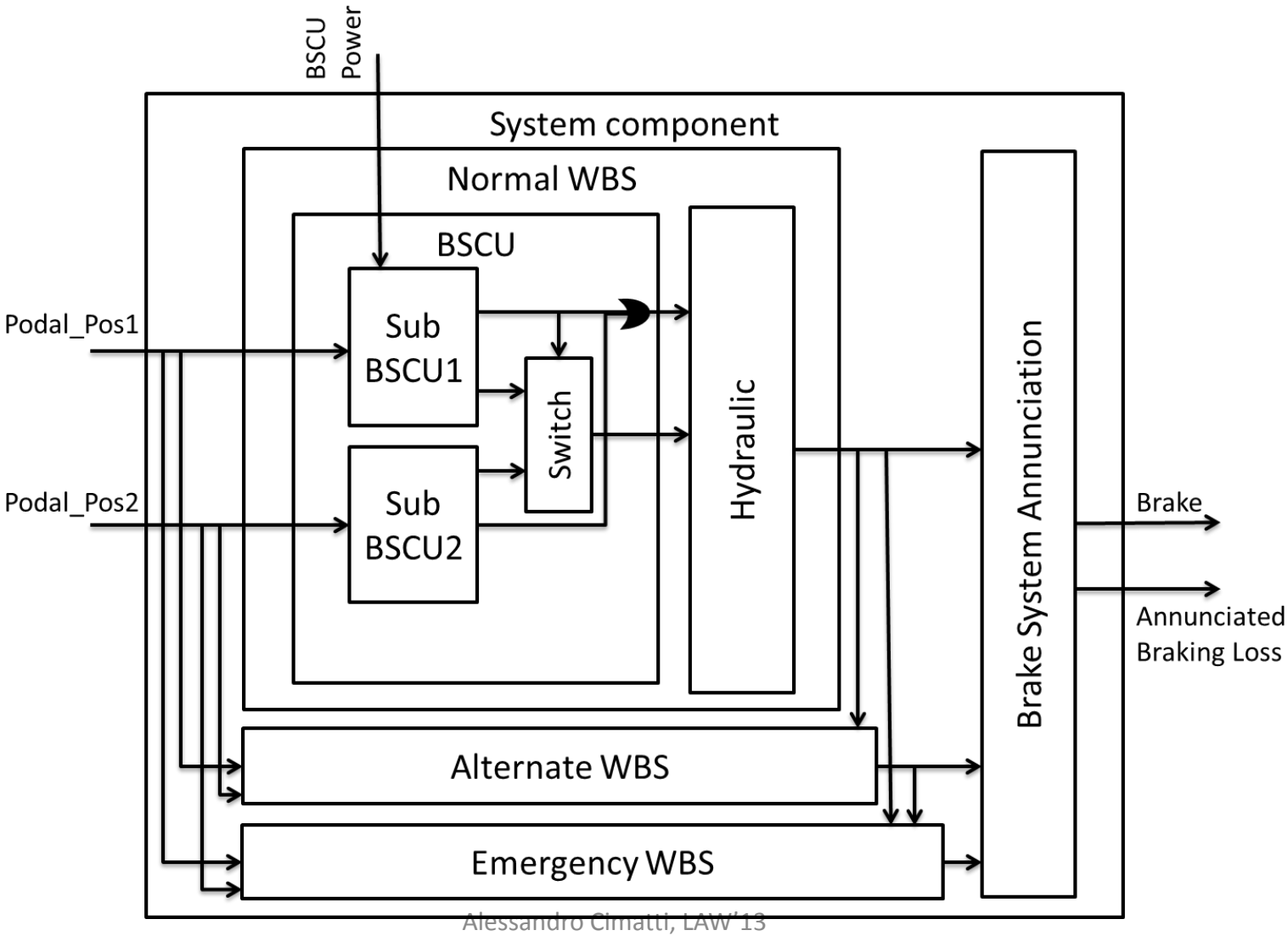
Hierarchical safety assessment

- Extend contract-based design for safety assessment
- Automatic fault injection based on failure to fulfill contract
  - Environment violates assumptions
    - Power always available
  - Component violates assumptions
    - Braking action delivered

Hierarchical Safety Assessment

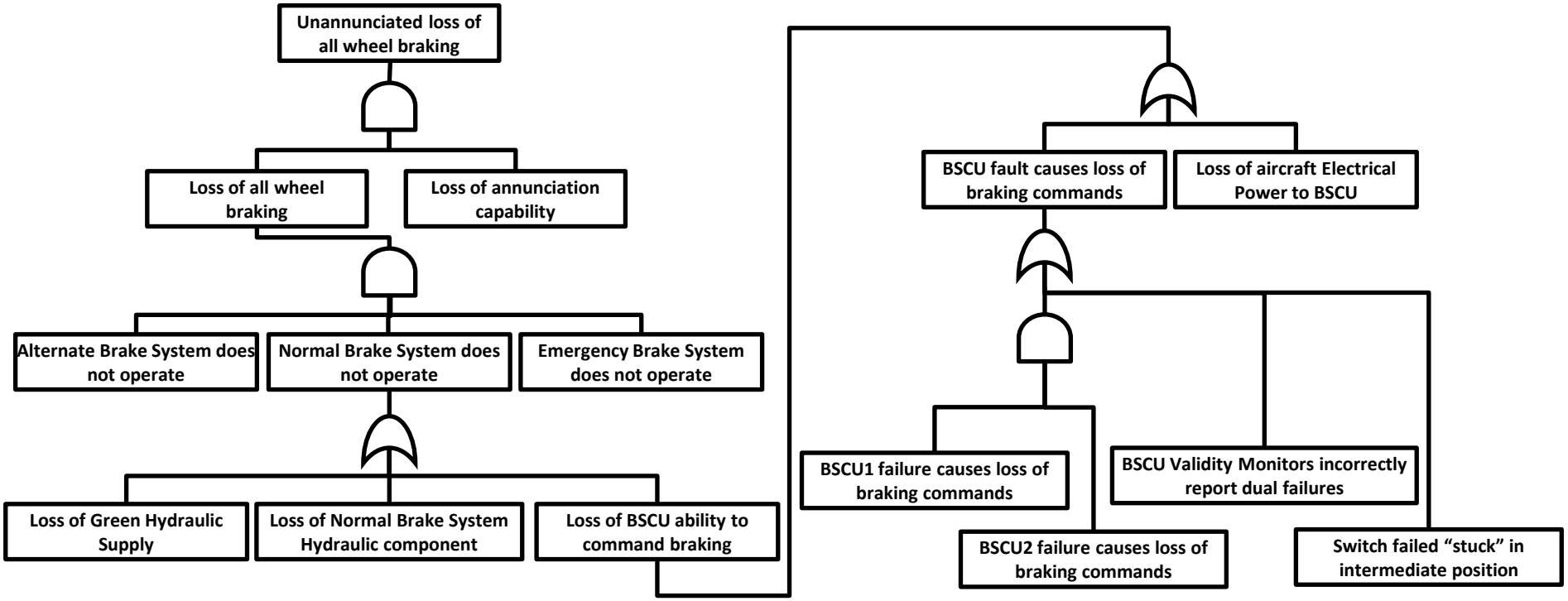


The Wheel Braking System [ARP10]



FT for the WBS

- Automatically generated (cfr [ARP10])



Future directions

- Application level
  - Usability
  - Advanced NLP techniques?
  - Diagnostic information
- Technological level
  - Improve back ends for hybrid cases
    - The nuXmv tool: NuSMV + MathSAT
  - Devise synthesis techniques
    - Strongest contract such that…
    - Region of assignment to parameters such that…

Bibliography

- [CGP99] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking.* MIT Press, 1999.
- [Szy02] C. Szyperski, *Component Software: Beyond Object-Oriented Programming, 2nd ed.*. Boston, MA: Addison-Wesley, 2002.
- [RBH+01] W.P. de Roever, F.S. de Boer, U. Hannemann, J.Hooman, Y. Lakhnech, M. Poel, J. Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press 2001.
- [BCC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, *Symbolic Model Checking Using SAT Procedures instead of BDDs*. DAC 1999: 317-320.
- [SSS00] M. Sheeran, S. Singh, G. Stålmarck, *Checking Safety Properties Using Induction and a SAT-Solver*. FMCAD 2000: 108-125.
- [Bra11] A.R. Bradley. *SAT-Based Model Checking without Unrolling*. VMCAI 2011: 70-87.
- [McM03] K.L. McMillan, *Interpolation and SAT-Based Model Checking*. CAV 2003: 1-13.
- [Ton09] S. Tonetta, *Abstract Model Checking without Computing the Abstraction*. FM 2009: 89-105.
- [CMT11] A. Cimatti, S. Mover, S. Tonetta, *HyDI: A Language for Symbolic Hybrid Systems with Discrete Interaction*. EUROMICRO-SEAA 2011: 275-278.
- [CMT13] A. Cimatti, S. Mover, S. Tonetta, *SMT-based scenario verification for hybrid systems*. Formal Methods in System Design 42(1): 46-66 (2013).
- [Pnu77] A. Pnueli, *The Temporal Logic of Programs*. FOCS 1977: 46-57.
- [AH01] L. de Alfaro, T.A. Henzinger, *Interface automata*. ESEC / SIGSOFT FSE 2001: 109-120.
- [CT12] A. Cimatti, S. Tonetta, *A Property-Based Proof System for Contract-Based Design*. EUROMICRO-SEAA 2012: 21-28.
- [CDT13] A. Cimatti, M. Dorigatti, S. Tonetta. *OCRA: A Tool for Checking the Refinement of Temporal Contracts . ASE 2013.*
- [ARP10] SAE ARP4754A Guidelines for Development of Civil Aircraft and Systems, 12/2010.