Property-Based and Contract-Based Design of System Architectures

Stefano Tonetta
tonettas@fbk.eu
Tutorial of ASE'13

FONDAZIONE BRUNO KESSLER

ES EMBEDDED SYSTEMS

Credits

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

Outline

1. Introduction and motivations
2. Infinite-state model checking
3. Properties specification languages
4. Contract-based design with temporal logics
5. OCRA tool support

First Part:
Introduction and motivations

A tutorial on property-based and contract-based design of system architectures

Model-based system engineering

- Models used for system requirements, architectural design, analysis, validation and verification.
- Different system-level analysis (safety, security, performance, …).
- Top-down refinement process.
- Software/hardware co-engineering.
- Definition of the platform and deployment.
- Applied to embedded systems:
  - Interaction with physical world (continuous time).
  - Real-time constraints.
  - Complex interaction of many components:
    - Sensors, actuators, monitors, communication links.

Formal methods as back-end

- Formal methods
  - Formal specification languages
    - Assign models a mathematical meaning
    - Different property languages for different model semantics
  - Formal verification to prove the properties on the models.
- Verification flow:
  - Design models translated into input for verification engine:
    - Typically a (meaningful) subset is considered
    - Automatic translation preserving semantics of properties of interest
  - Requirements formalized into properties
    - This is typically a manual process.
  - Results mapped back to the design flow.
- This tutorial will focus on:
  - Model checking [CGP99] techniques for a wide spectrum:
    - Finite states vs. infinite states
    - Discrete time vs. hybrid/continuous-time.
  - Properties languages in the different cases.

Component-based design

- A component is a unit of composition with contractually specified interfaces [Szy02].
- Components are the constituent parts of a system architecture.
- Sub-components interact through connections.
- They are seen as black box for proper
  - Compositional verification.
  - Reuse.
  - Structural/independent refinement.

Compositional verification techniques

- Compositional verification [RBH+01]:
  1. Prove properties of the components (for example, with model checking).
  2. Combine components' properties to prove system's property without looking into the internals of the components (sometimes reduced to validity/satisfiability check for composition of properties).
- Formally:

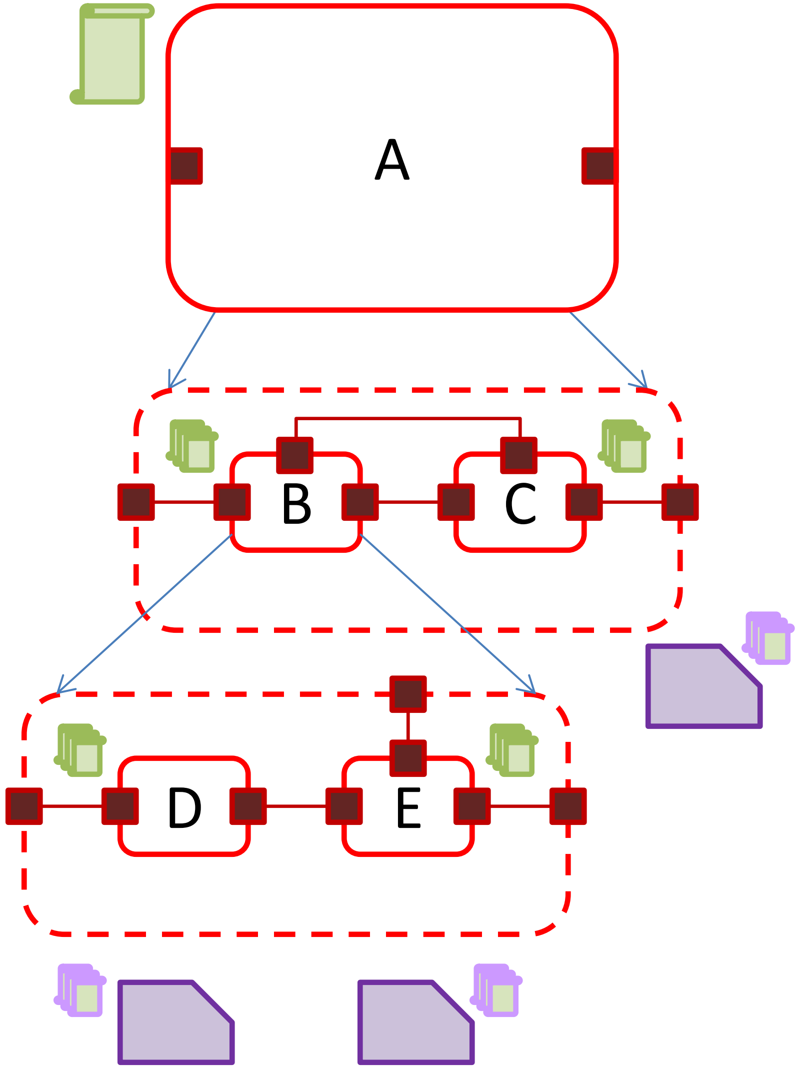
$$\frac{\dfrac{S_1 \vDash P_1, \quad S_2 \vDash P_2, \dots, S_n \vDash P_n}{\gamma_S(S_1, S_2, \dots, S_n) \vDash \gamma_P(P_1, P_2, \dots, P_n)} \qquad \gamma_P(P_1, P_2, \dots, P_n) \vDash P}{\gamma_S(S_1, S_2, \dots, S_n) \vDash P}$$

- γ_P combines the properties depending on the connections used in γ_S
- E.g. synchronous case:

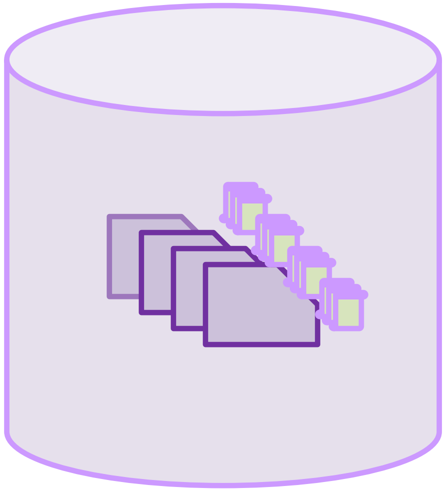
$$\gamma_P(P_1, P_2, \dots, P_n) = \rho_{\gamma_S}(P_1 \wedge P_2 \wedge \cdots \wedge P_n)$$

where ρ_{γ_S} is the renaming of symbols defined by the connections in γ_S.

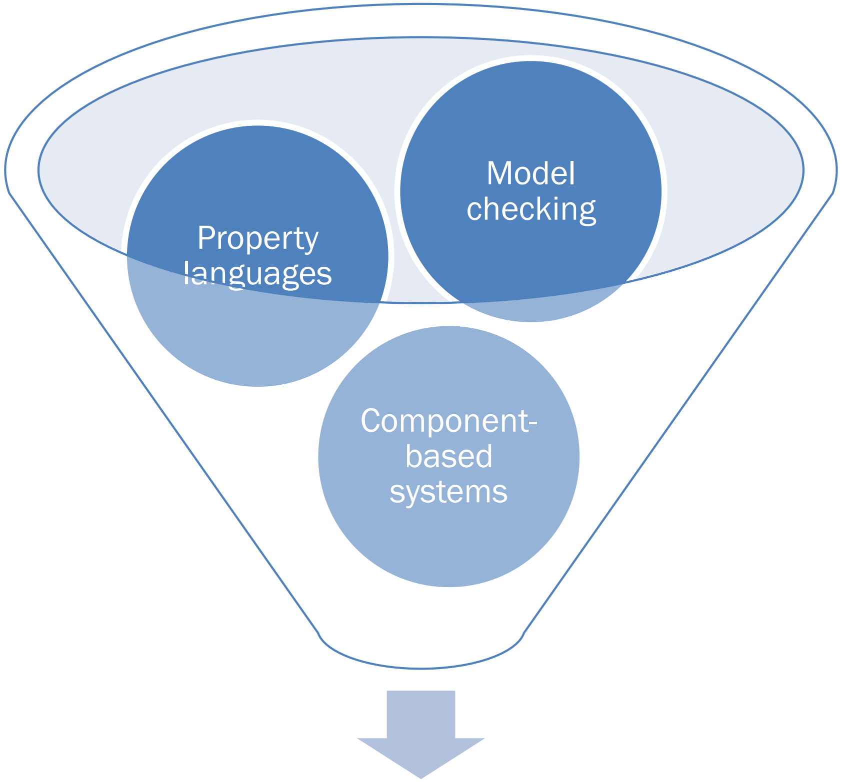
Contract-based approach



1. Step-wise refinement of components.
2. Compositional verification.
3. Proper reuse of components.



Main ingredients



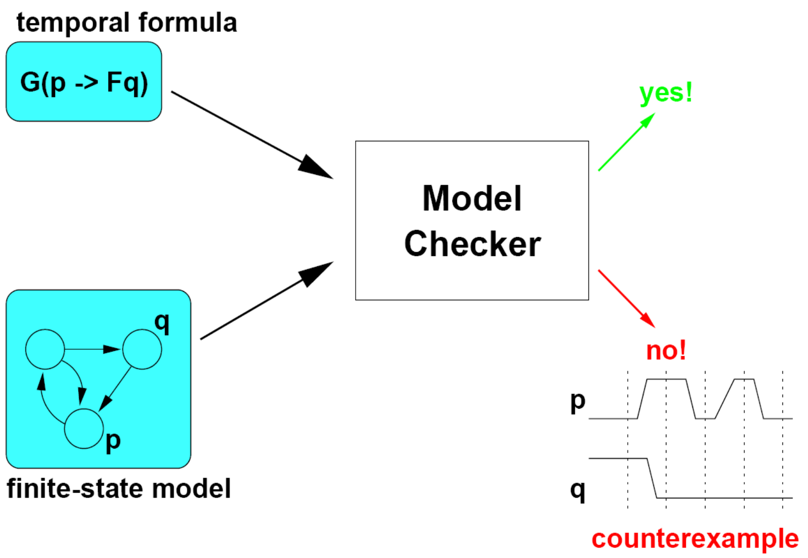
Support to contracts: a temporal logic approach.

Second Part:
Infinite-state model checking

A tutorial on property-based and contract-based design of system architectures

Model checking

- Problem of checking if a system satisfies a property [CGP99].
- Algorithmic procedure to analyze Reactive Systems
  - systems with infinite behaviors
  - hardware, communication protocols, operating systems, controllers
- 30 years old
- Turing Award 2007 (Clarke, Emerson, and Sifakis).
- Tremendous Impact:
  - Routinely applied in hardware design.
  - Increasing use in the design of embedded systems.
  - Ideal for model-based system engineering.

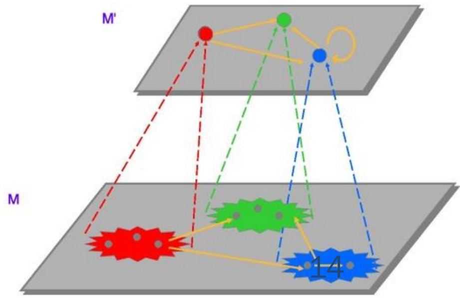


Symbolic representation

- Symbolic variables $V = \{v_1, \dots, v_n\}$ to represent the state space.
- Symbolic formulas used to represent:
  - Set of states: $\phi(V) \equiv \{ s \mid s \vDash \phi \}$
  - Set of transitions: $T(V, V') \equiv \{ \langle s, s' \rangle \mid \langle s, s' \rangle \vDash \phi \}$
    - Where the variables $V' = \{v'_1, \dots, v'_n\}$ represent next state variables.
- A valuation s:V→D used to build a formula true for exactly that valuation.
  - $\langle x \leftarrow 1, y \leftarrow 1, z \leftarrow 5 \rangle$ we derive the formula $x=1 \wedge y=1 \wedge z=5$
- Each complete assignment can be considered a state
- A transition system is represented by:
  - The set of initial states represented by the formula $I(V)$
  - The transition relation represented by the formula $R(V, V')$

Symbolic algorithms

- Symbolic algorithms search the state space manipulating formulas.
- Main types of algorithms:
  - Based on fix-point:
    - Compute the pre/post-image of a set of states with quantifier elimination, e.g., $pre(\phi) \coloneqq \exists V'(\phi \wedge T)$
    - Accumulate until at fix-point you get all reachable states.
  - Based on satisfiability:
    - Prove properties with a series of satisfiability checks ($sat(\phi)$ iff there exists s such that $s \vDash \phi$).
  - Based on abstraction:
    - E.g. predicate abstraction (partition states according to predicates).
    - Properties proved on abstract system hold also on the original system.



SAT-based algorithms

- Bounded Model Checking (BMC) [BCC+99]
  - Check $sat(\phi_k)$ where ϕ_k is sat iff there exists a path of M of length up to k violating the property P.
  - Focused on finding errors.
- Induction
  - Base case: check if the initial state satisfies P (invariant)
  - Inductive case: check if the transitions preserve the invariant.
- K-induction [SSS00]
  - Base case: check if all initial path satisfies P (invariant) up to k steps.
  - Inductive case: check if every path of $k + 1$ steps preserve the invariant.
- IC3 [Bra11]
  - Keeps sequence of relative inductive invariants (frames).
  - Use counterexamples to strengthen the frames.
- Also combined with abstraction:
  - Interpolation-based abstraction [McM03]
    - Unsat BMC used to over-approximate reachable states.
  - Implicit abstraction [Ton09]
    - SAT-based algorithms on abstract state space (without computing explicitly it).

From SAT to SMT

- Previous algorithms assume to have a solver for the satisfiability of formulas.
- First developed for finite-state systems with the support of SAT solvers.
- Satisfiability Modulo Theory (SMT):
  - Satisfiability for decidable fragments of first-order logic.
  - SAT solver used to enumerate Boolean models.
  - Integrated with decision procedure for specific theories, e.g., theory of real linear arithmetic.
- SAT solvers substituted by SMT solvers.
- Search algorithms applied to infinite-state systems (although in general undecidable).

SMT-based hybrid systems

- Hybrid systems encoded into symbolic transition systems with SMT constraints [CMT11,CMT13].
- Reals used to represent time and continuous variables.
- Transitions are either
  - Discrete: time does not change, state variables change according to transition relation $\phi(V, V')$
  - Timed: time elapses, discrete variables do not change, continuous variables evolve according to the flow law
    - E.g., the flow condition $\dot{x} < a$ is encoded into $x' - x < a(t' - t)$ where t is the time variable.

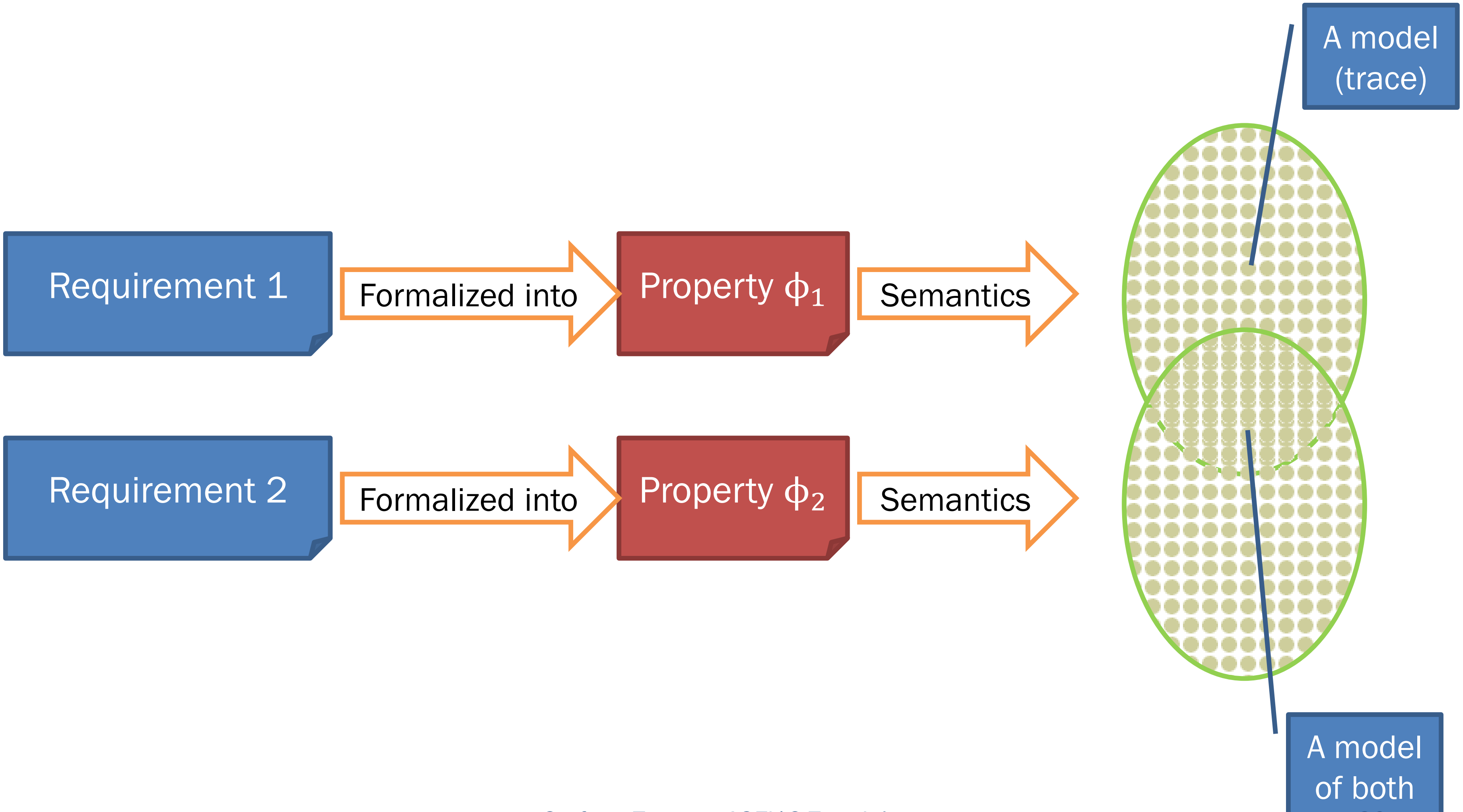
Third Part: Property specification languages

A tutorial on property-based and contract-based design of system architectures

Properties

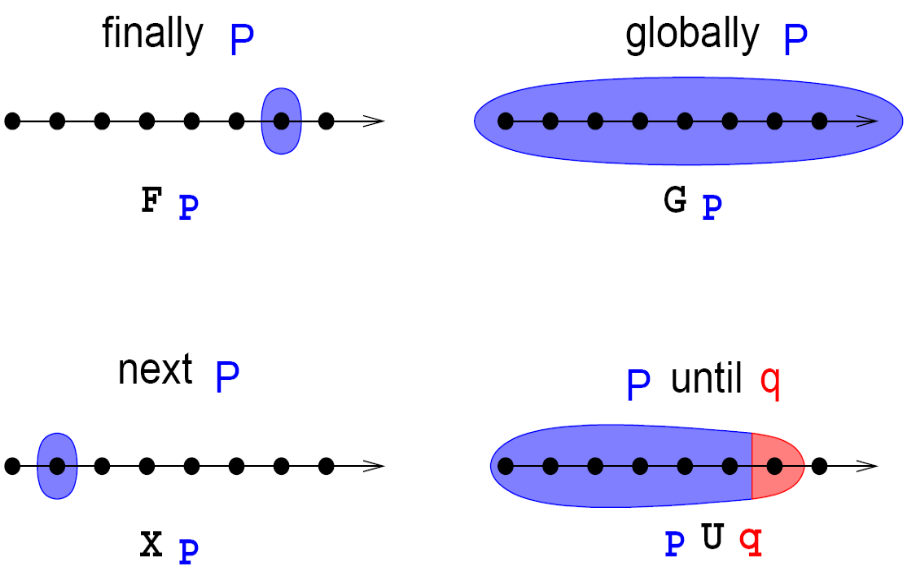
- Properties are expressions in a mathematical logic using symbols of the system description.
- Used to formalize requirements.
- Also defined as assertions on the system’s behavior.
- Problems:
  - Analysis: find the properties of a system.
  - Verification: check if the system satisfies the properties.
  - Validation: check if we are considering the right properties.
  - Synthesis: construct a system that satisfies the properties.

Properties, traces, and logic



Linear Temporal Logic

- Conceived by Pnueli in 1977 [Pnu77]
- Linear models
  - State sequences (traces).
- Built over set of atomic propositions AP.
- LTL formulas are the smallest set of formulas such that:
  - any atomic proposition p AP is an LTL formula;
  - if p and q are LTL formulas, then $\neg$p, p$\wedge$q, p$\vee$q are LTL formulas;
  - if p and q are LTL formulas, then X p, G p, F p, and [p U q] are LTL formulas.
- Semantics defined for every trace, for every $i \in \mathbb{N}$.
- $M \vDash \phi$ iff $M, \sigma, 0 \vDash \phi$ for every trace σ of M.



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” – immediate reaction
- GFp “infinitely many times p” – fairness
- FGp “eventually permanently p”
- $G(p \rightarrow (qUr))$

Simple entailment example

- $G(request \rightarrow F(received))$
- $G(received \rightarrow F(processed))$
- $G(processed \rightarrow X(grant))$

From which we can entail

- $G(request \rightarrow F(grant))$

Past operators

- Past operators
  - $Y\phi$, in the previous state ϕ, dual of X
  - $O\phi$, in the past once ϕ, dual of F
  - $H\phi$, in the past always ϕ, dual of G
  - $\phi_1 S \phi_2$, in the past ϕ_1 since ϕ_2, dual of U

Regular expressions

- RELTL enriches LTL with regular expressions:
  - Suffix implication: $\{r\} \mid\!\rightarrow \phi$ means that every finite sequence matching r is followed by a suffix satisfying ϕ.
  - Suffix conjunction: $\{r\} \diamond\!\rightarrow \phi$ means that there exists a finite sequence matching r and followed by a suffix satisfying ϕ.
- Example:
  - $\{\{\{\neg p\}[*]; p\}[*3]\} \rightarrow Fq$
  - $G(\{request; busy[*]; grant\} \rightarrow response)$

Property specification language

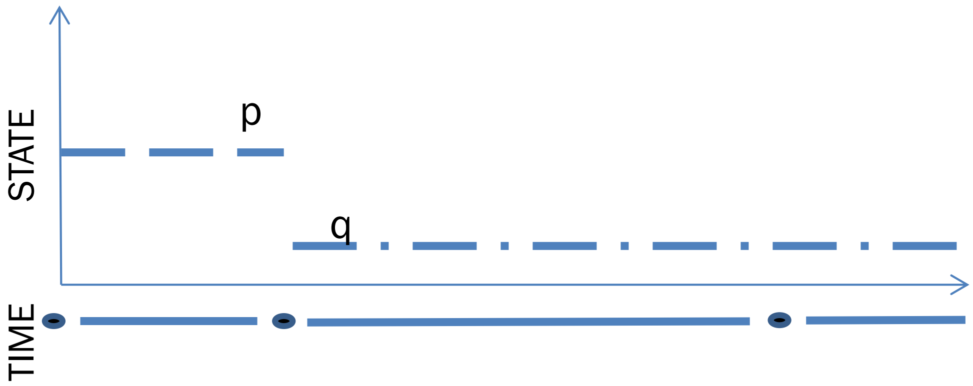
- Rich language to specify assertions on hardware design.
- Include RELTL.
- Increase usability with
  - Syntactic sugar
  - English words instead of math symbols:
    - "always" (G)
    - "never" ($G\neg$)
    - "eventually" (F)
    - "next" (X)

From finite to infinite

- Use first-order predicates instead of propositions:
  - $G(x \geq a \wedge x \leq b)$
  - $GF(x = a) \wedge GF(x = b)$
- Predicates interpreted according to specific theory T (henceforth, only used reals).
- “next” to express changes/transitions:
  - $G(next(x) = x + 1)$
  - $G(next(a) - a \leq b)$

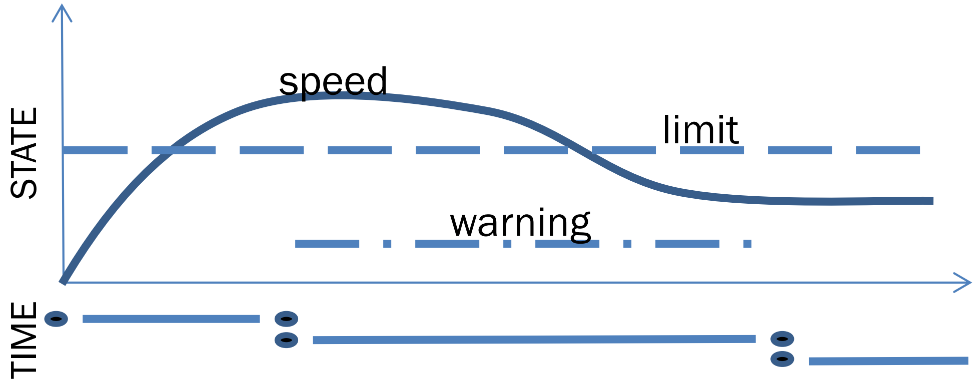
Metric Temporal Logic

- $G(p \rightarrow F_{\leq 3}q)$ "p is followed by q within 3 time units"
- $G(p \rightarrow G_{\leq 2}q)$ "Whenever p holds, q holds in the following two time units"
- $G(p \rightarrow (\neg q U_{\geq 1} q))$ "p is followed by q but only after 1 time unit"



Hybrid RELTL (HRELTL)

- $G(der(x) < 2)$ “The derivative of x is always less than 2”
- $G(a \rightarrow der(x) = 0)$ “Whenever a holds, the derivative of x is zero”
- $G(a \rightarrow (bUder(x) \leq 5))$ “Whenever a holds, b remain true until the derivative of x is less or equal to 5”.



$$G(speed > limit \rightarrow F(warning))$$

Othello

- Human-readable language for HRELTL.
- Controlled natural language expressions. Examples:
  - "always" (G)
  - "in the future" (F)
  - "and" ($\wedge$)
- Validated in the EuRailCheck project focus on the formalization and validation of ETCS requirements.
  - Example: "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 into: "always (train_trip implies (emergency_brake_command until (der(train_location)=0 and driver_acknowledges_trip)))"

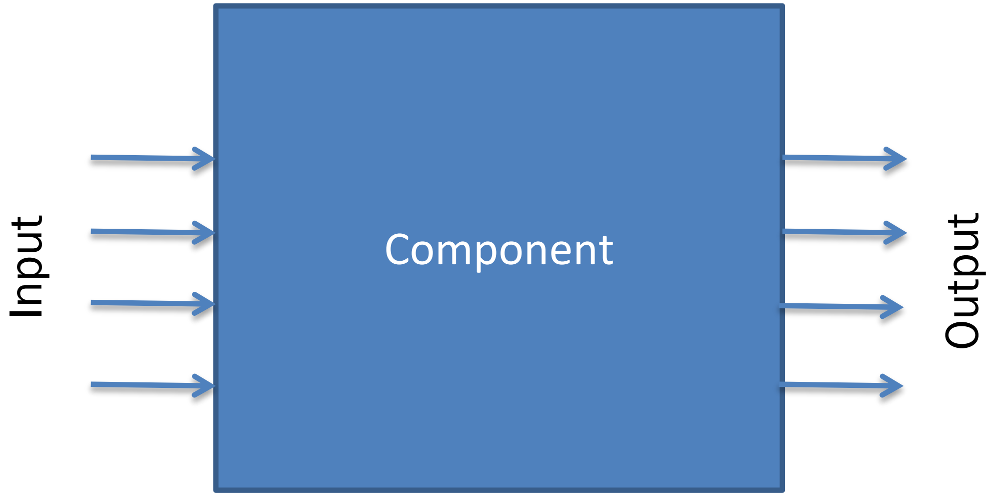
Fourth Part: Contract-based design with temporal logics

A tutorial on property-based and contract-based design of system architectures

Component

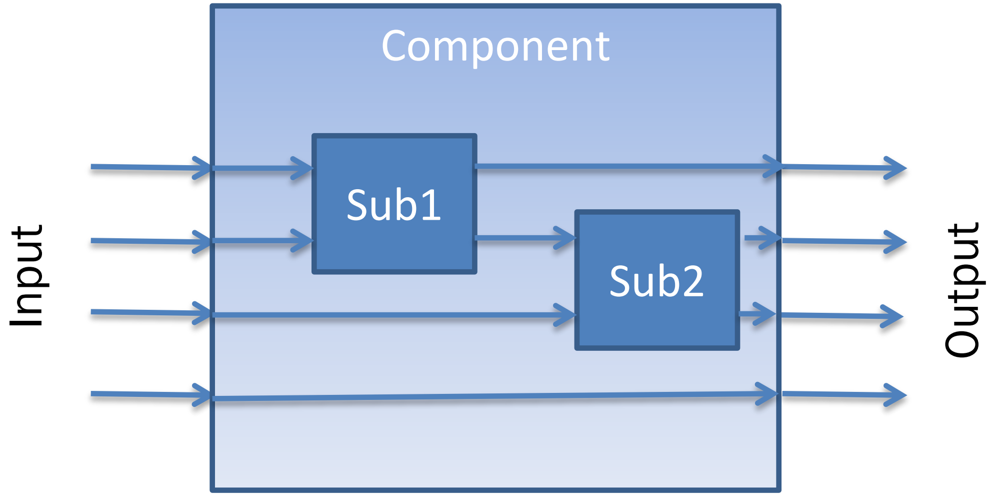
- A component has
  - A syntactic interface
  - Optionally, an internal structure.
  - A behavior.
  - An environment.
  - Properties.

Black-box component interface

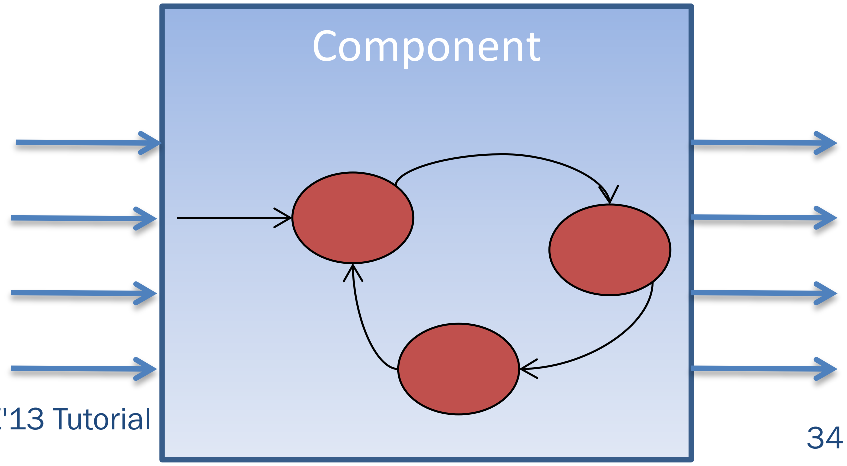


- A component interface defines boundary of the interaction between the component and its environment.
- Consists of:
  - Set of input and output ports (syntax)
    - Ports represent visible data and events exchanged with environment.
  - Set of traces (semantics)
    - Traces represent the behavior, 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 $\mathrm{L}(Imp) \subseteq Tr(V_S)$
- M can be associated with $\mu_{Imp}: Tr(I_S) \rightarrow 2^{Tr(O_S)}$ such that $\mu_{Imp}(\sigma_i) = \{\sigma_o \mid \sigma_i \times \sigma_o \in L(Imp)\}$
  - Input trace mapped to a set of output traces
  - “set” to consider non determinism
  - Empty set corresponds to rejected input trace

Component environment

- State machine Env environment of S iff $\mathrm{L}(Env) \subseteq Tr(I_S)$
- Compatibility of implementation with environment (e.g., for reuse):
  - Trace-based (black-box) view:
    - Imp must accept any trace of Env (i.e., $\mathrm{L}(Env) \subseteq \{ \sigma \mid \mu_{Imp}(\sigma) \neq \emptyset \}$)
  - 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
  - Output ports of the composite component
  - Input ports of the subcomponents.

System architecture

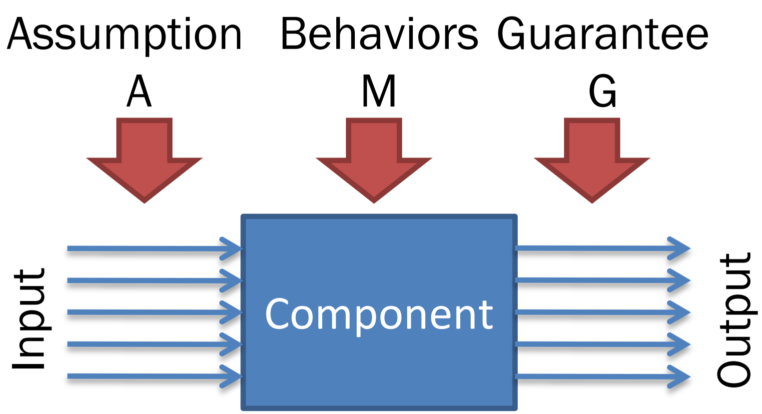
- A component is actually a component type.
- A system architecture is an instance of a composite component.
- It defines a tree of component instances.

Contracts

- Properties of the component and its environment.
- Can be seen as assertion for component interfaces.
- Contracts used to characterize the correctness of component implementations and environments.
- Typically, properties for model checking have a "god" view of the system internals.
- For components instead:
  - Limited to component interfaces.
  - Structure into assumptions and guarantees.
- Contracts for OO programing 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:
  - $\phi(V)$ assertion over variables V
  - $\langle\langle\phi\rangle\rangle \subseteq Tr(V)$ semantics of ϕ
- A contract of component S is a pair $\langle A, G\rangle$ of assertions over V_S
  - A is the assumption,
  - G is the guarantee.
- Env is a correct environment iff $\mathrm{L}(Env) \subseteq \langle\langle A\rangle\rangle$
- Imp is a correct implementation iff $\mathrm{L}(Imp) \cap \langle\langle A\rangle\rangle \subseteq \langle\langle G\rangle\rangle$

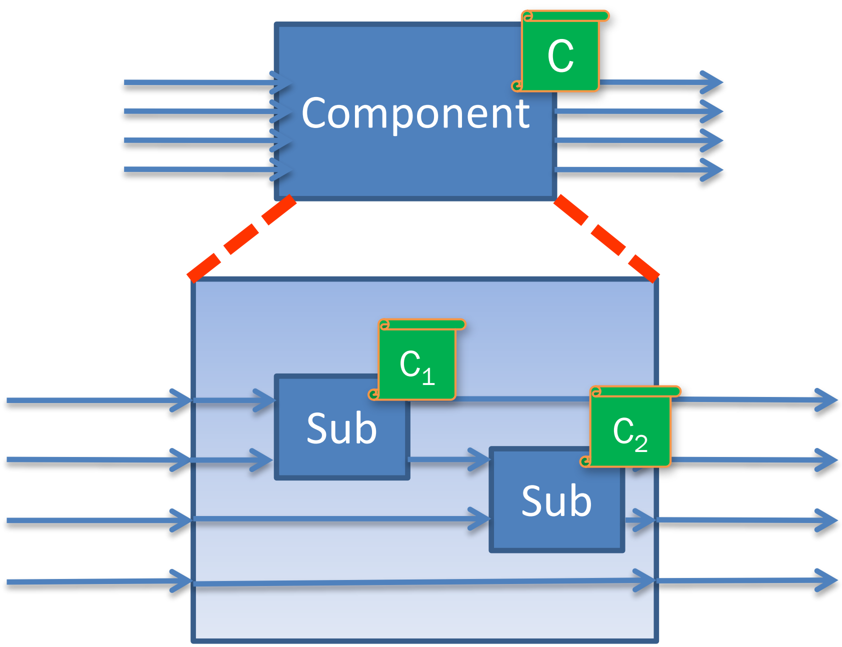


Example with Othello assertions:

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

Trace-based contract refinement

- The set of contracts $\{C_i\}$ refines C with the connection γ ($\{C_i\} \preccurlyeq_\gamma C$) iff for all correct implementations Imp_i of C_i and correct environment Env of C:
  1. The composition of $\{Imp_i\}$ is a correct implementation of C.
  2. For all k, the composition of Env and $\{Imp_i\}_{i \neq k}$ is a correct environment of C_k.
- Verification problem:
  - check if a given refinement is correct (independently from implementations).



Proof obligations for contract refinement

- Given C1=<α1,β1>, … , C1=<αn,βn>, C=<α,β>
- Proof obligations for $\{C_i\} \preccurlyeq C$:
  - $\gamma\left(\left(\bigwedge_{1\leq j\leq n}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \beta)\right)$
  - $\gamma\left(\left(\bigwedge_{2\leq j\leq n}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \alpha_1)\right)$
  - …
  - $\gamma\left(\left(\bigwedge_{1\leq j\leq n, j\neq i}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \alpha_i)\right)$
  - …
  - $\gamma\left(\left(\bigwedge_{1\leq j\leq n-1}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \alpha_n)\right)$

- Theorem: $\{C_i\} \preccurlyeq_\gamma C$ iff the proof obligations are valid. [CT12]

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

- Correspond to one direction of the contract refinement.
- Many works focused on finding the right assumption/guarantee.
- E.g. how to break circularity?
  - $\big(G(A \rightarrow B) \wedge G(B \rightarrow A)\big) \Rightarrow G(A \wedge B)$ is false
  - Induction-based mechanisms
  - $\big(B \wedge G(A \rightarrow XB) \wedge A \wedge G(B \rightarrow XA)\big) \Rightarrow G(A \wedge B)$ is true
- Note they are structural ways to prove the property-based refinement.

Fifth Part:
OCRA tool support

A tutorial on property-based and contract-based design of system architectures

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
...
```

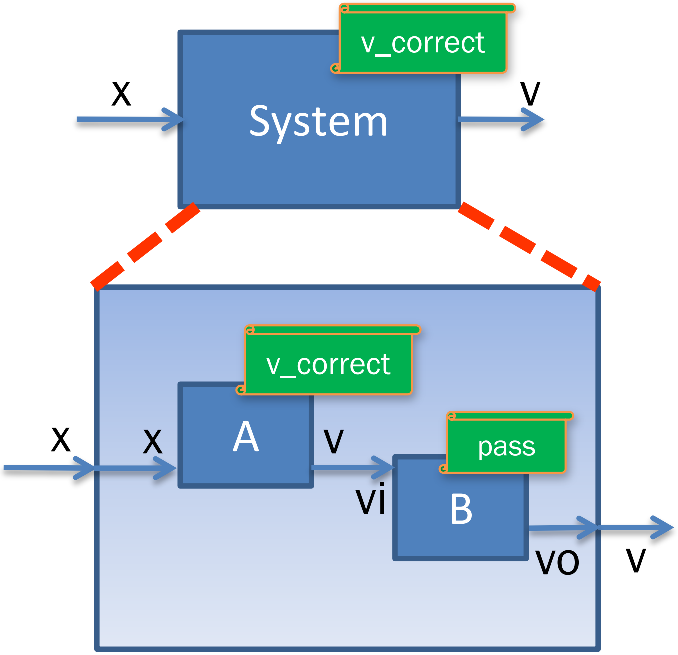
Component refinement

```
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.y := a.v;
                CONNECTION v:= b.v;
        …
```

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




simple.oss

OCRA temporal operator

- LTL operators with the following syntax:
  - “always” G
  - “in the future” F
  - “until” U
  - “then” X
  - “historically” H
  - “in the past” O
  - “since” S
  - “previously” Y

OCRA hybrid aspects

- Port types are either
  - NuSMV types: “boolean”, enumeratives, …
  - nuXmv additional types: “real”, “integer”, …
  - “continuous”, i.e. real-value ports evolving continuously in time.
  - “event”, i.e. boolean-value port that is assigned only on discrete transitions.
- Atomic formulas may be:
  - Boolean variables.
  - Equalities.
  - Arithmetic predicates over integer, real, and continuous terms.

OCRA hybrid aspects

- Special function symbols:
  - “der” denoting the derivative of a continuous variable (e.g., “der(x)=0”).
  - “next” denoting the next value after a discrete change (e.g. “next(x)=x+1”).
  - “time_until” used to express constraints on the time to the next occurrence of an event:
    - “time_until(e)<=2” means $(\neg e) U_{\leq 2} e$
- Syntactic sugar:
  - fall(x) means “x=true and next(x)=false”
  - rise(x) means “x=false and next(x)=true”
  - change(x) means “next(x)!=x”
- Important warning:
  - The time model is hybrid with continuous evolution.
  - What does “next” mean when time elapses?
  - In OCRA/Othello/HRELTL, “next” forces a discrete step:
    - “always ((der(timer)=1) and (timer=timeout implies next(timer)=0))”

Commands

- ocra_check_syntax
- ocra_check_refinement
- ocra_check_consistency
- ocra_check_implementation
- ocra_check_receptiveness

- Typical script:
  - set verbose_level 1
  - set on_failure_script_quits 1
  - set pp_list cpp
  - ocra_check_syntax -i SenseSpacecraftRate.oss
  - ocra_check_refinement
  - quit

- Call: ocra –source SenseSpacecraftRate.cmd

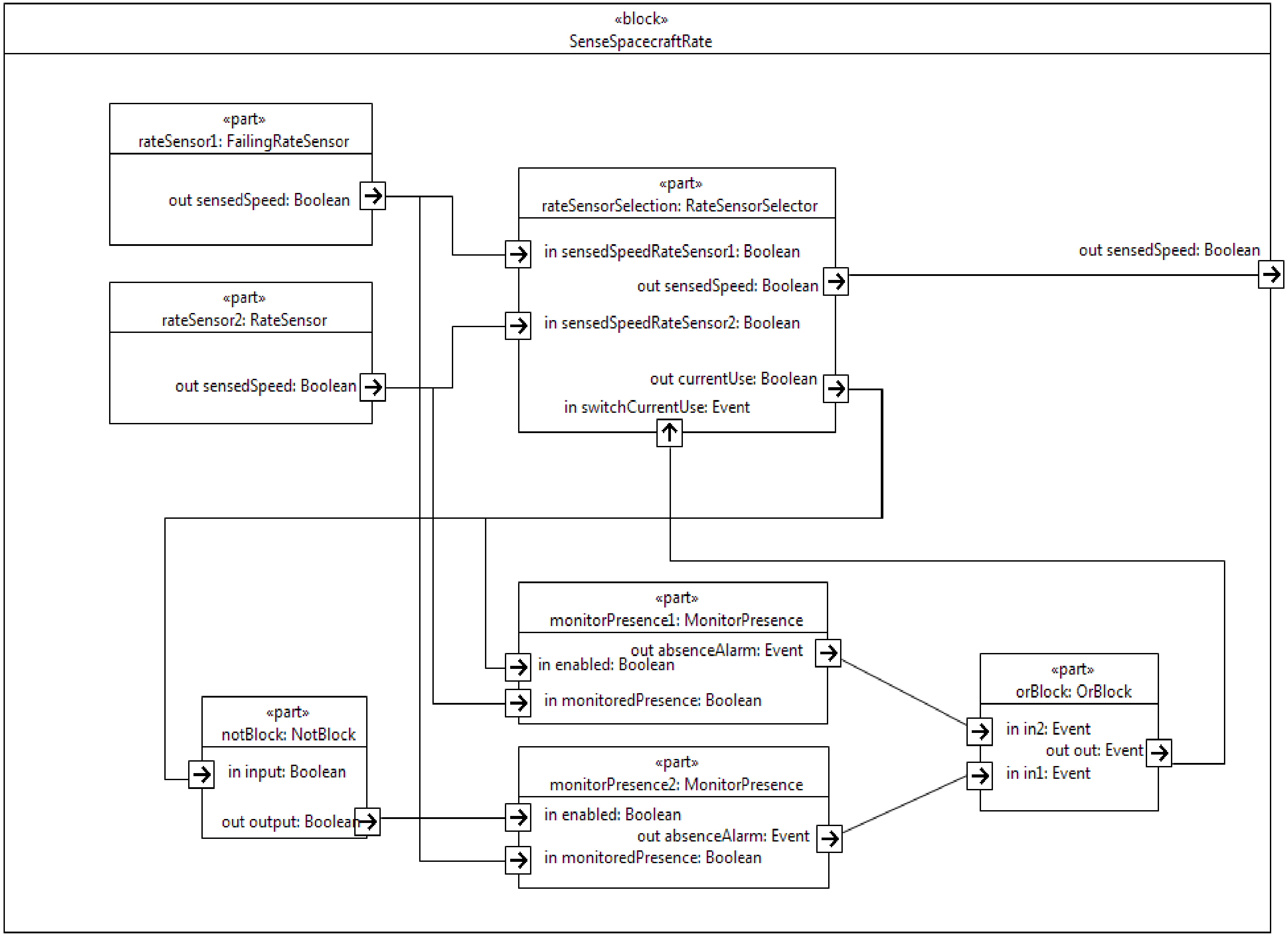
Discrete vs. hybrid

- OCRA is parametrized by the logic.
- The expressions can be restricted and interpreted as discrete-time LTL or hybrid LTL.
- Default is hybrid.
- Set discrete-time to switch to LTL.

Contract refinement results

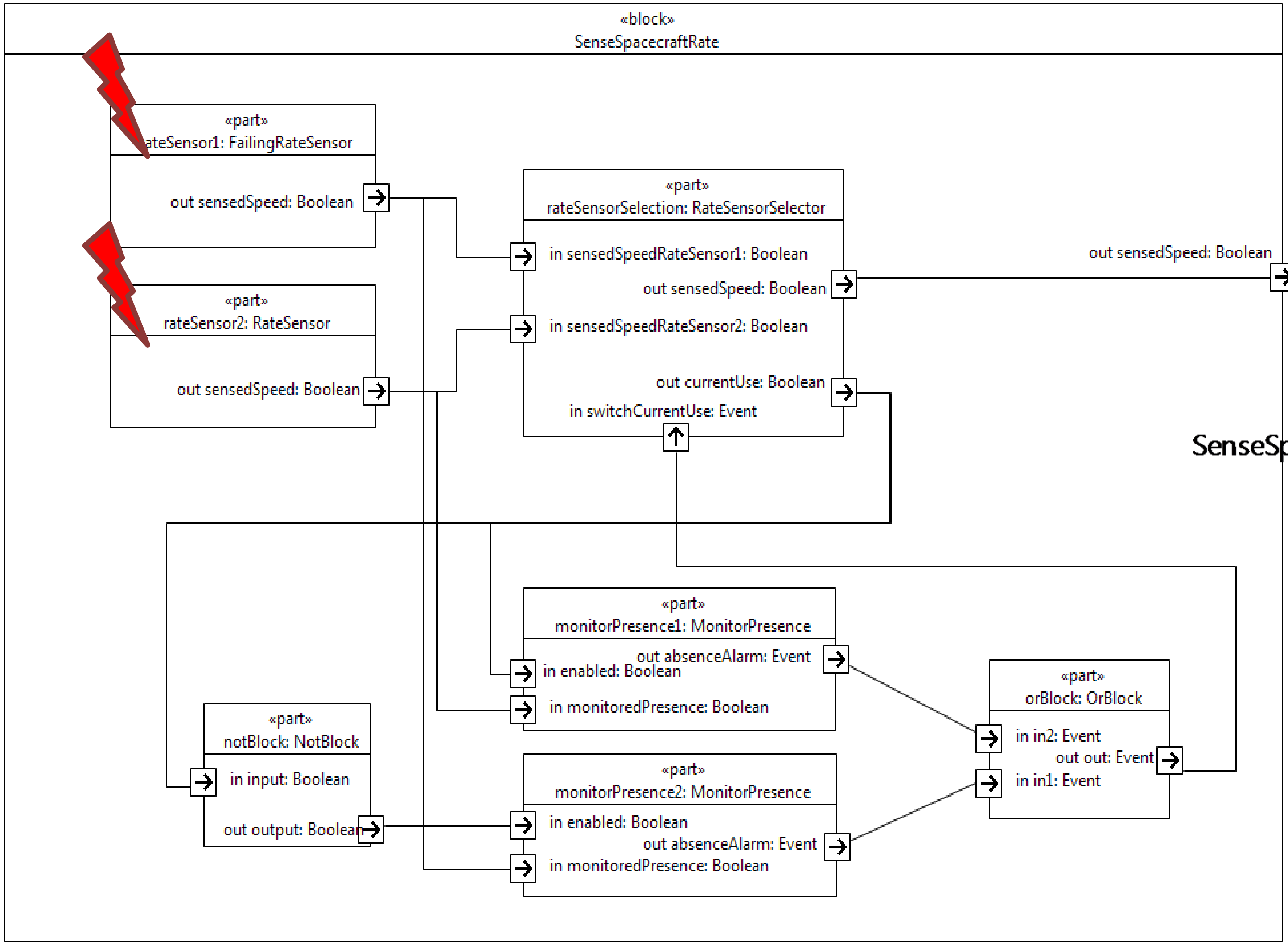
- For every component, for every refined contract, check refinement.
- For every proof obligation, check its validity:
  - [OK] if valid
  - [BOUND OK] if no counterexample found up to k
  - [FAIL] if found counterexample

SenseSpacecraftRate Example



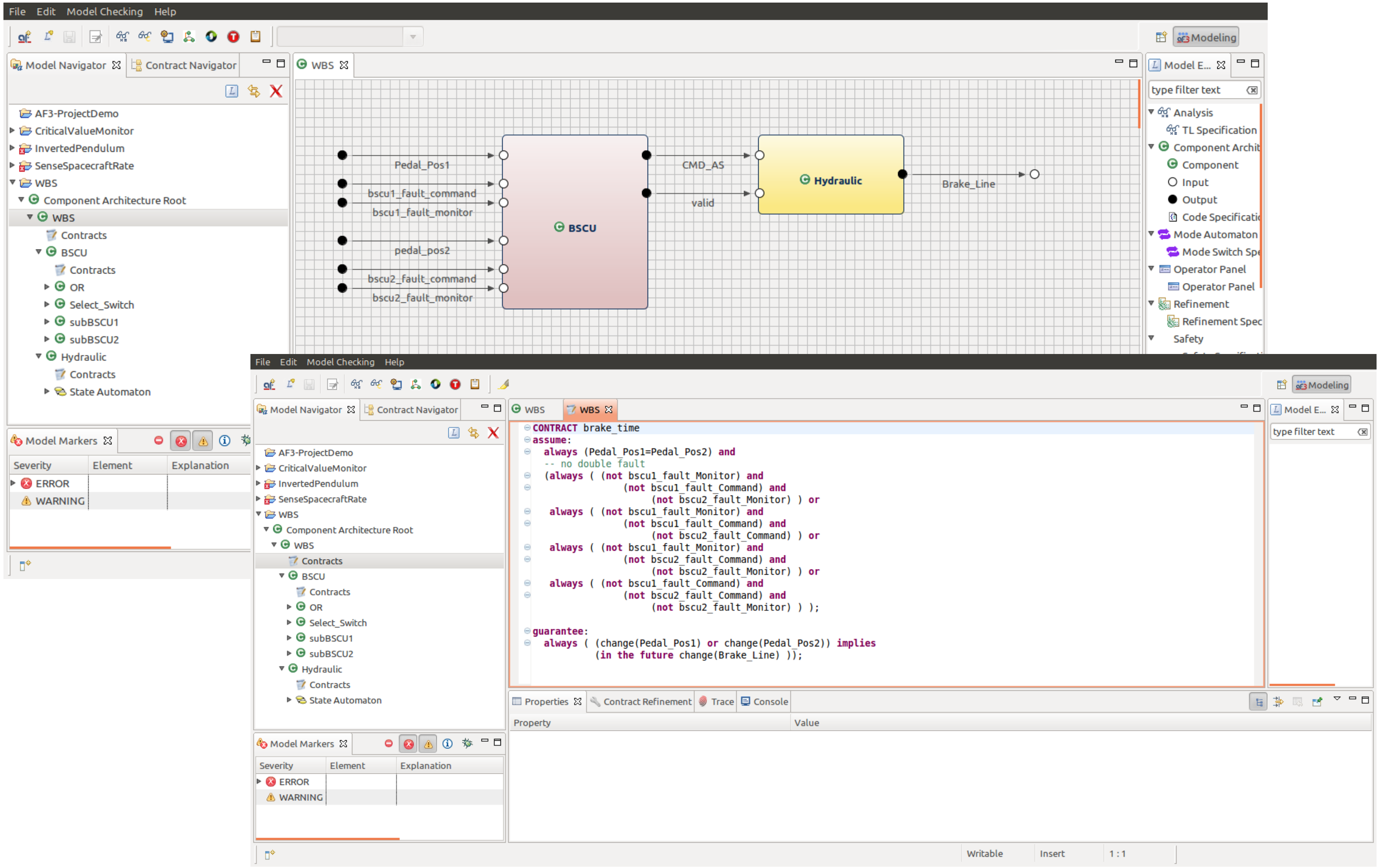
SenseSpacecraftRate.oss

Considering failures



SenseSpacecraftRate_singlefailure.oss

Plugin for AutoFocus



Summary

- Contract-based design powerful
  - For property refinement
  - Safety analysis
- Temporal logic is suitable for component contracts.
- Contract framework parametrized by the logic.
- SMT-based model checking used to reason with expressive properties.
- OCRA tool support.

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.
- META program and AGREE tool by Cofer and colleagues.
  - Also on system architecture with temporal logics for assume-guarantee reasoning.

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.*