Advanced model checking for verification and safety assessment

Alessandro Cimatti

Fondazione Bruno Kessler (FBK)

Invited Lectures on Advanced Verification

Part 2

Lecture prepared in collaboration with

Stefano Tonetta and Marco Gario

Some slides borrowed from Cristian Mattarei, Marco Bozzano, Anthony Pires

Lecture 2

- Safety Assessment
  - Fault Extension
  - Fault Tree Computation
- Requirements Analysis
- Contract Based Design
- Contract-Based Safety Assessment
- Case-Studies
  - WBS
  - NASA
- Wrap-up

Safety Assessment

Safety Assessment

The **safety assessment process** provides a **methodology** to evaluate the design of systems, and to determine that the **associated hazards** have been properly addressed…

…and it should be planned to provide the **necessary assurance** that all relevant failure conditions have been **identified and considered.**

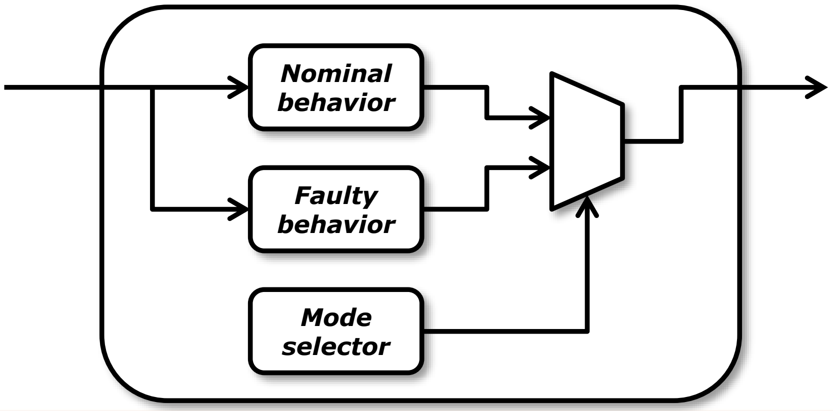
Aerospace Recommended Practice 4761
SAE International

Model-Based Safety Assessment (MBSA)

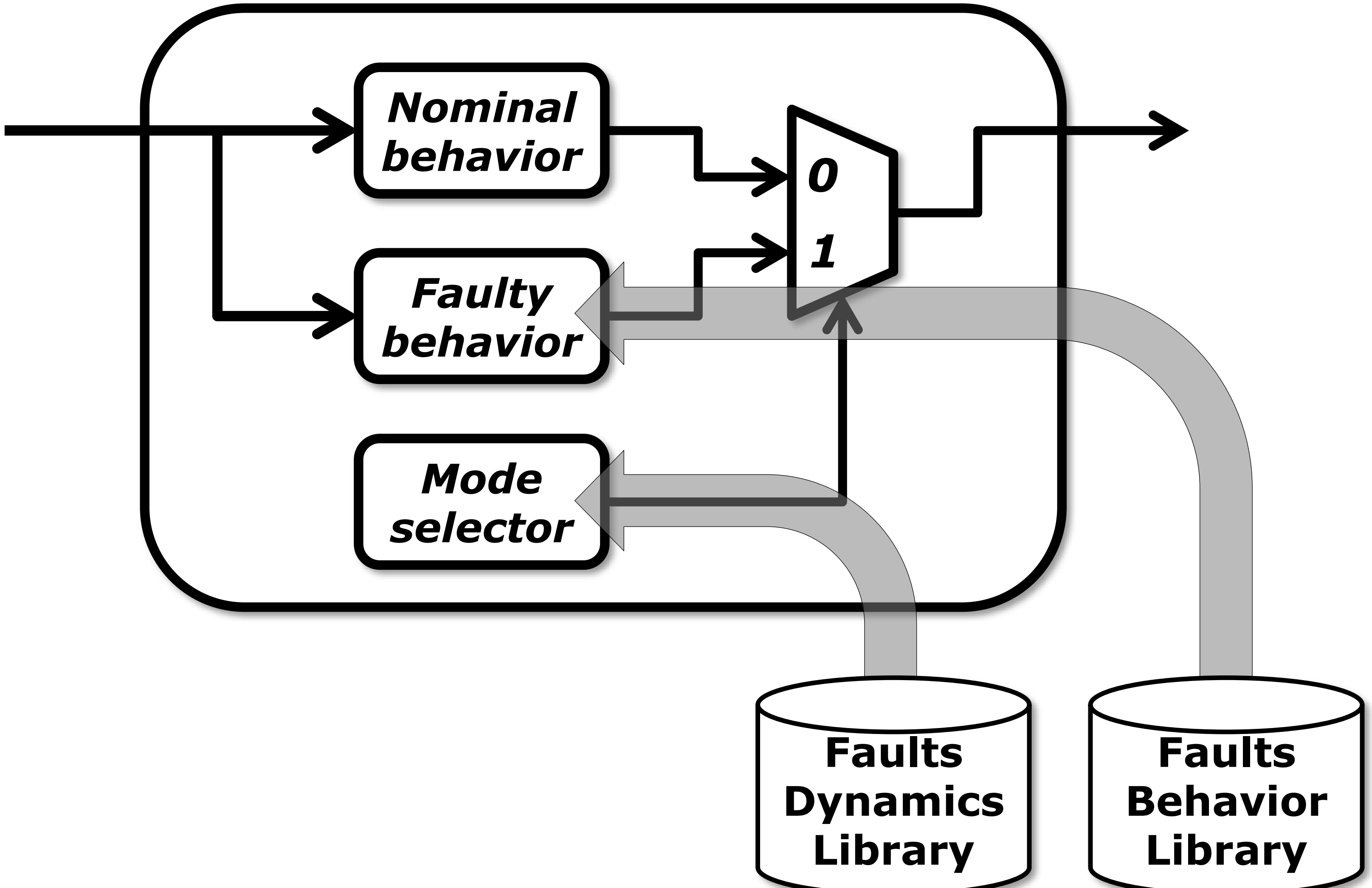
- Used for the evaluation of safety critical systems e.g., redundancy / fault tolerance
- The nominal system description is extended by allowing faulty behaviors (fault injection)
- Find all possible fault configurations that may cause the reachability of an unwanted condition (a.k.a. Top Level Event - TLE)
  - Assume $M \models \phi$
  - TLE $:= \neg\phi$
    - Bad states in case of invariant property
    - Generalized also to LTL

Model Extension

- From **nominal** $M \coloneqq \langle V, I, T \rangle$ to **extended** $M^X \coloneqq \langle V^X, I^X, T^X \rangle$ model, where $V \cup F \subseteq V^X$
- Extended model with disabled fault variables (i.e. set to FALSE) should have the same behavior as the nominal one
- **Symbolic Fault Injection,** additional behavior in parallel to the nominal one, selected via a mode selector:



Model-Based Fault Injection



Fault Tree Analysis

Fault Injection:

- $\mathcal{M} \Longrightarrow \mathcal{M}_{[\mathcal{F}]}$

Cutsets computation:

- $CS := \{cs \in 2^{\mathcal{F}} \mid \mathcal{M}^{X} \wedge cs \not\models \varphi\}$

Minimal cutsets computation:

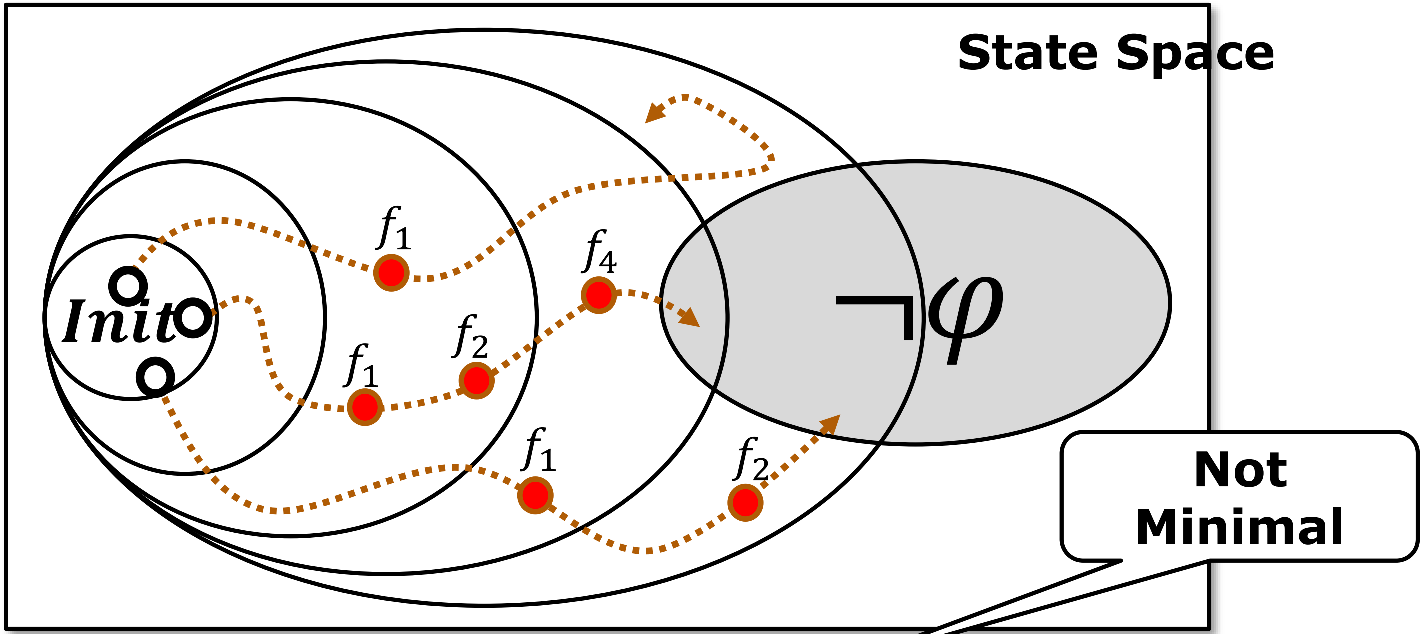
- $MCS := \{cs \in CS \mid \nexists cs' \in CS.cs' \subset cs\}$

Formula representing the minimal cutsets:

- $MCS^{\top} := \bigwedge_{cs \in MCS} (\bigvee_{f \in cs} (f = \top))$

Minimal Cutsets Computation

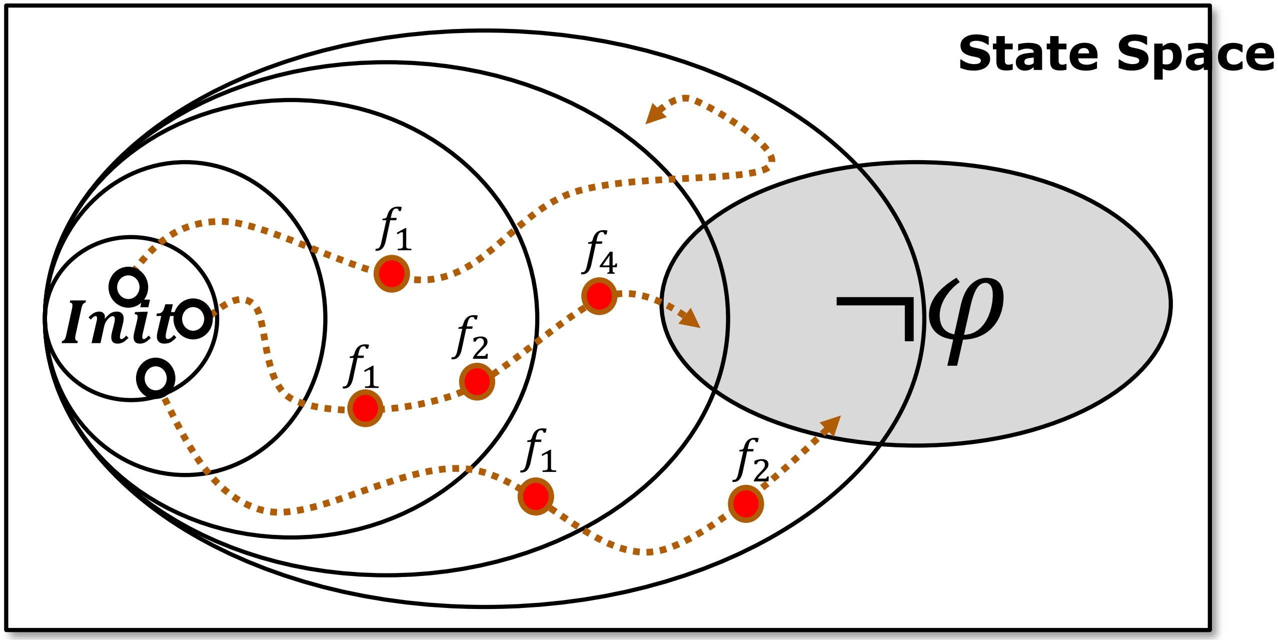
- Given an extended model $\mathrm{M}^{\mathrm{X}} \coloneqq \langle V^X, I^X, T^X \rangle$, find all **minimal** Faults Configurations FC (**Cutsets**) s.t. $\exists$ trace π triggering FC and witnessing $\mathrm{M}^{\mathrm{X}} \nvDash \varphi$



Example: $\mathrm{CS} = \{\{f_1, f_2, f_4\}, \{f_1, f_2\}\}$

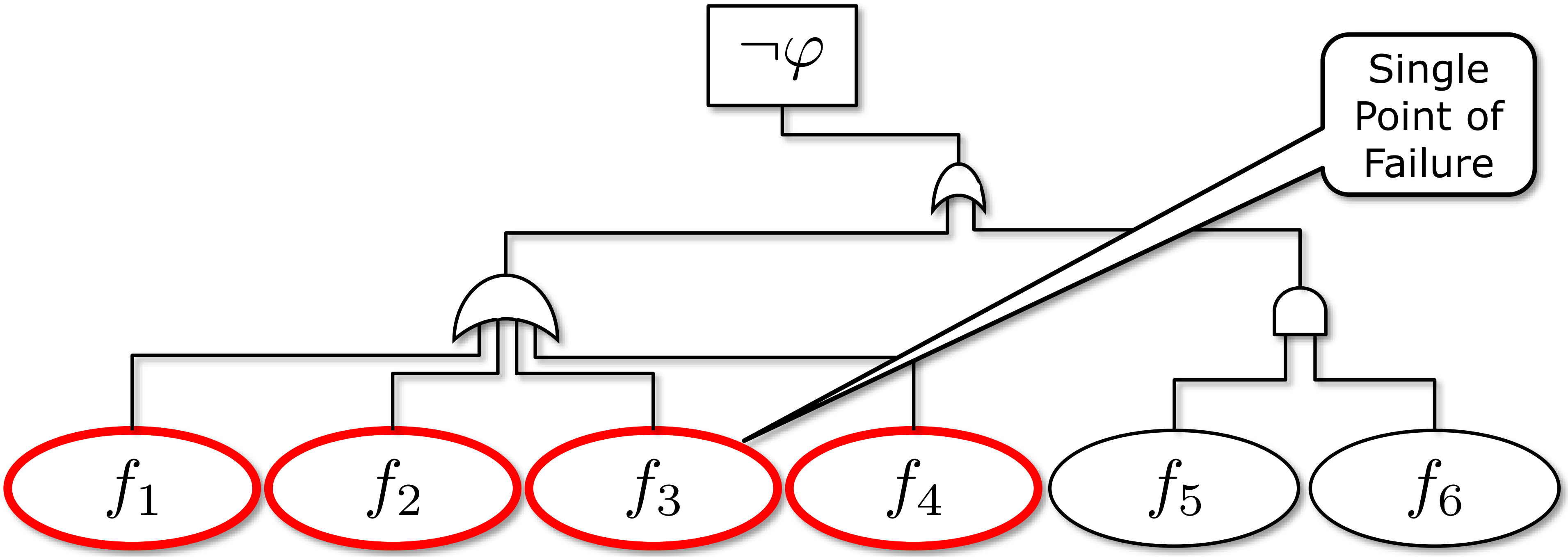
Minimal Cutsets Computation

- Given an extended model $\mathrm{M}^{\mathrm{X}} \coloneqq \langle V^X, I^X, T^X \rangle$, find all **minimal** Faults Configurations FC (**Cutsets**) s.t. $\exists$ trace π triggering FC and witnessing $\mathrm{M}^{\mathrm{X}} \not\models \varphi$

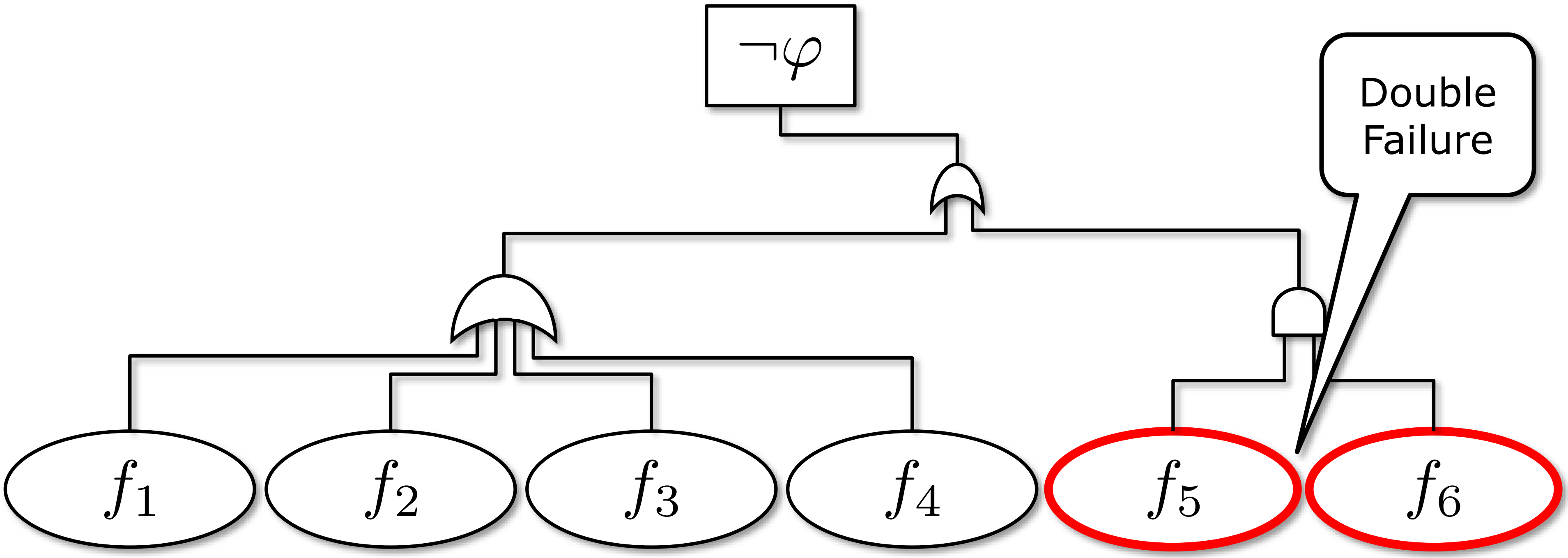


Example: $\mathrm{MCS} = \{\{f_1, f_2\}\}$

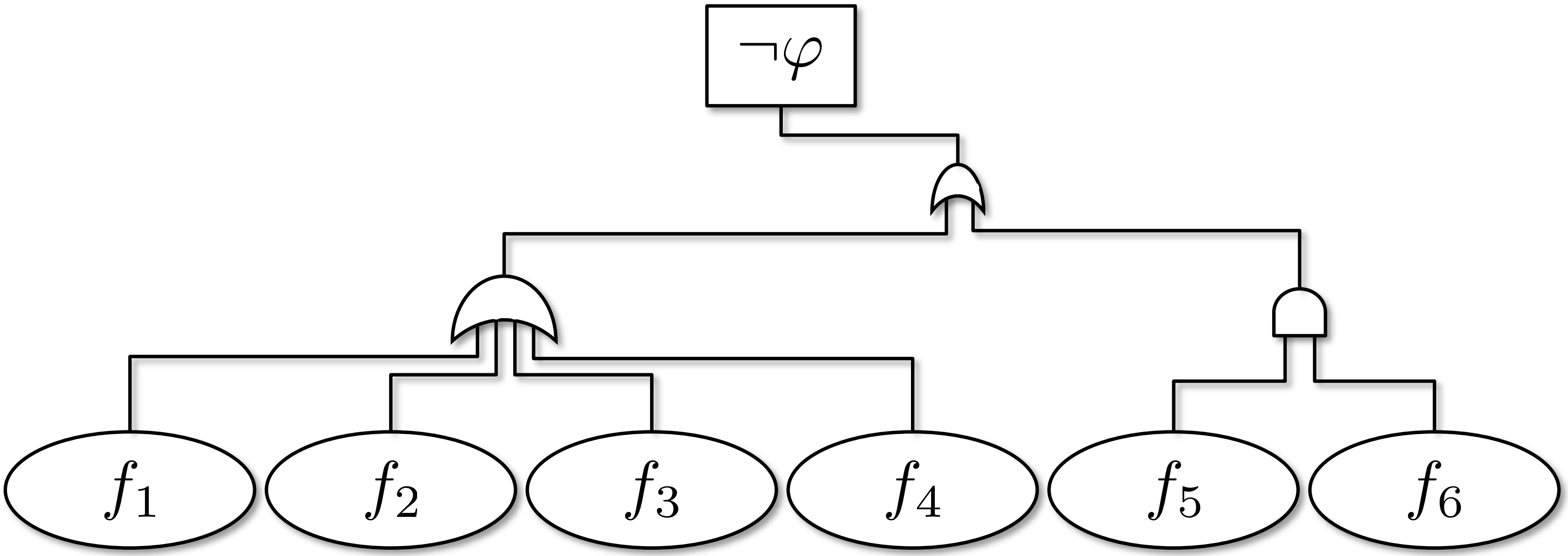
Fault Tree Analysis



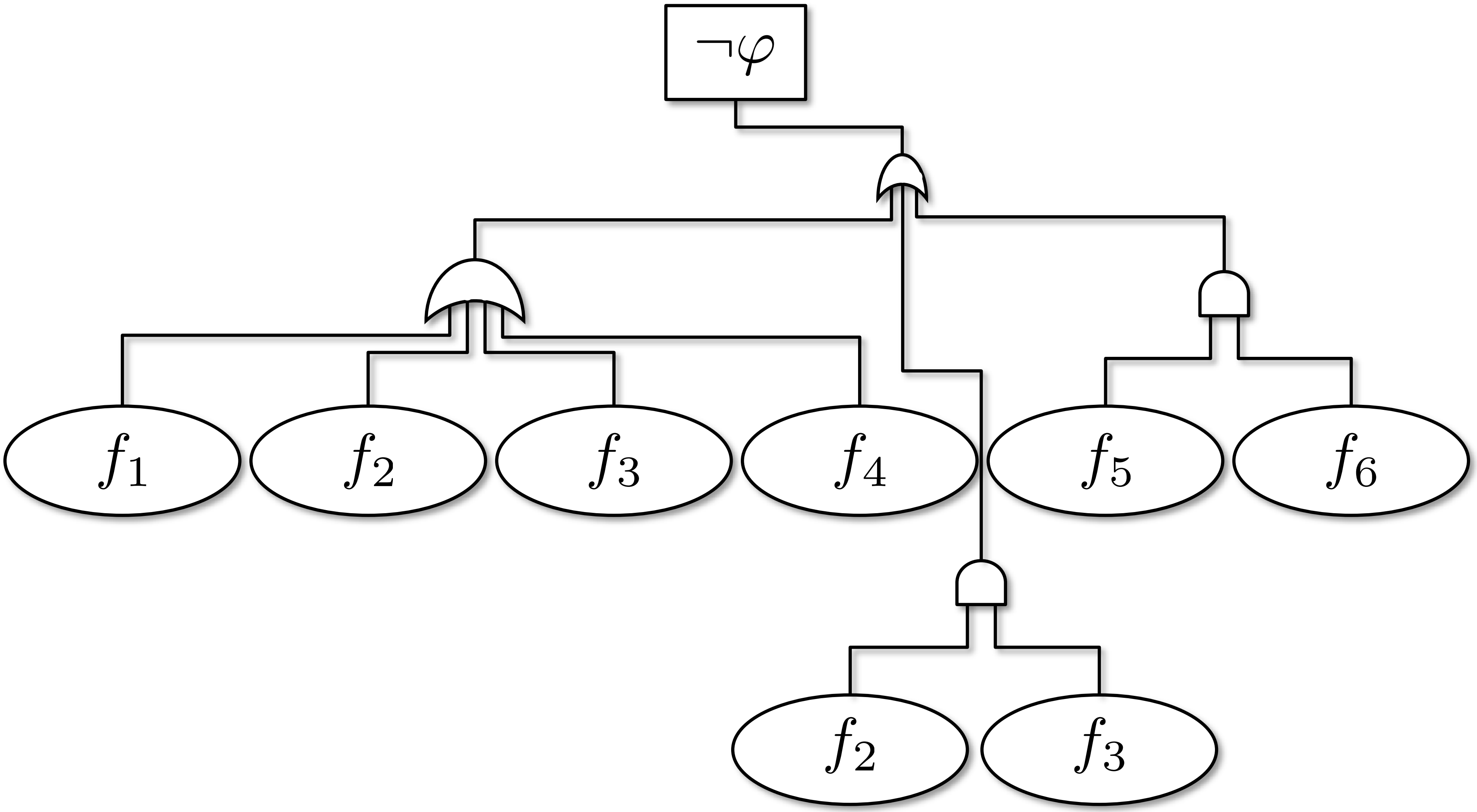
Fault Tree Analysis



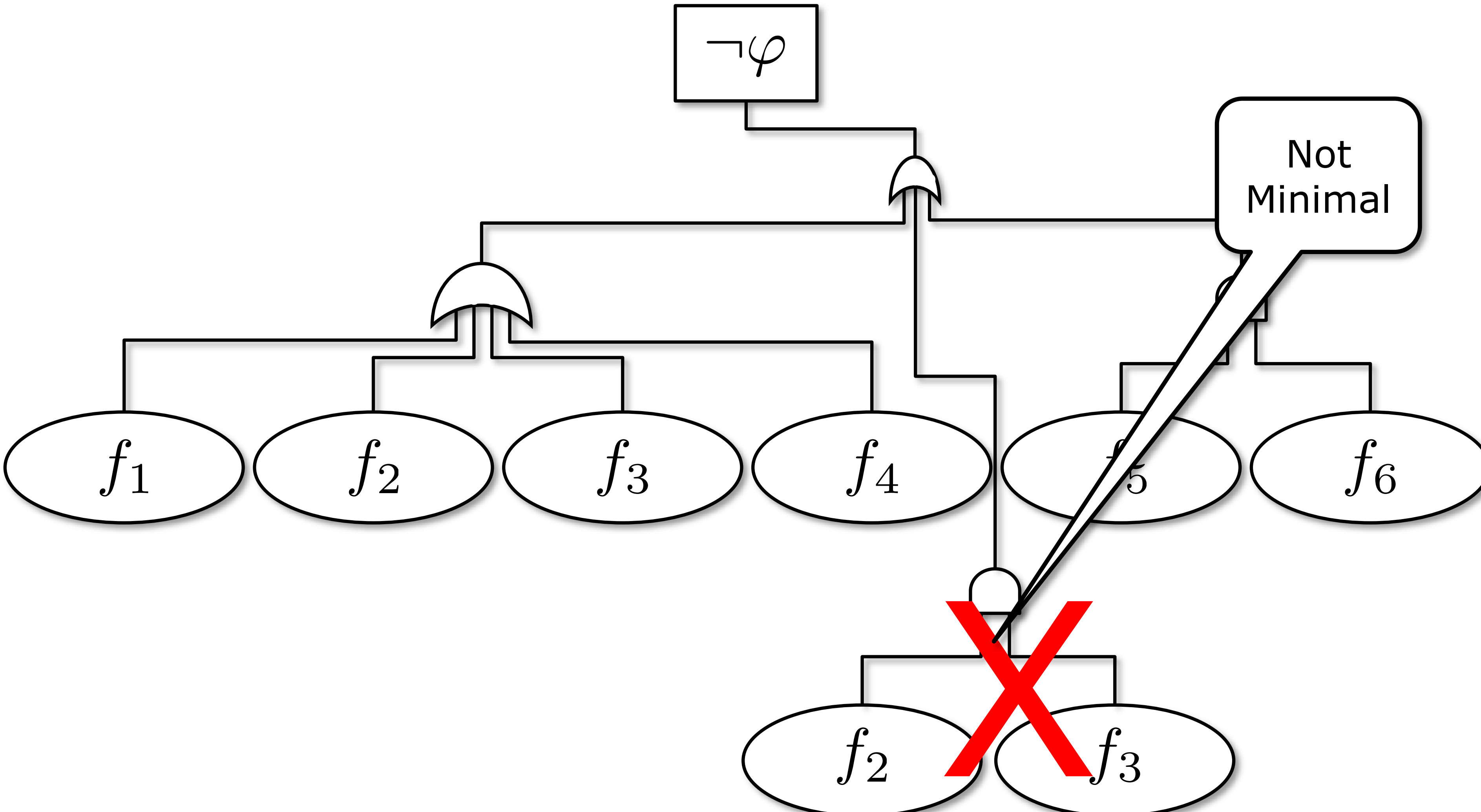
Fault Tree Analysis



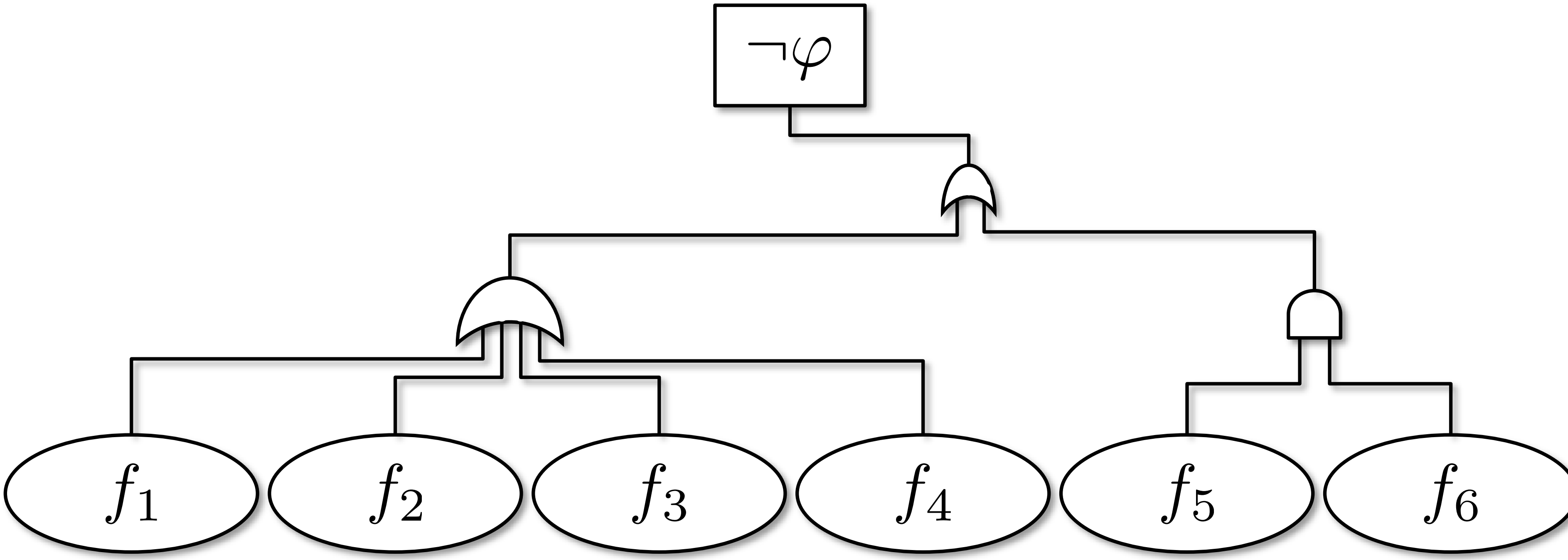
Fault Tree Analysis



Fault Tree Analysis

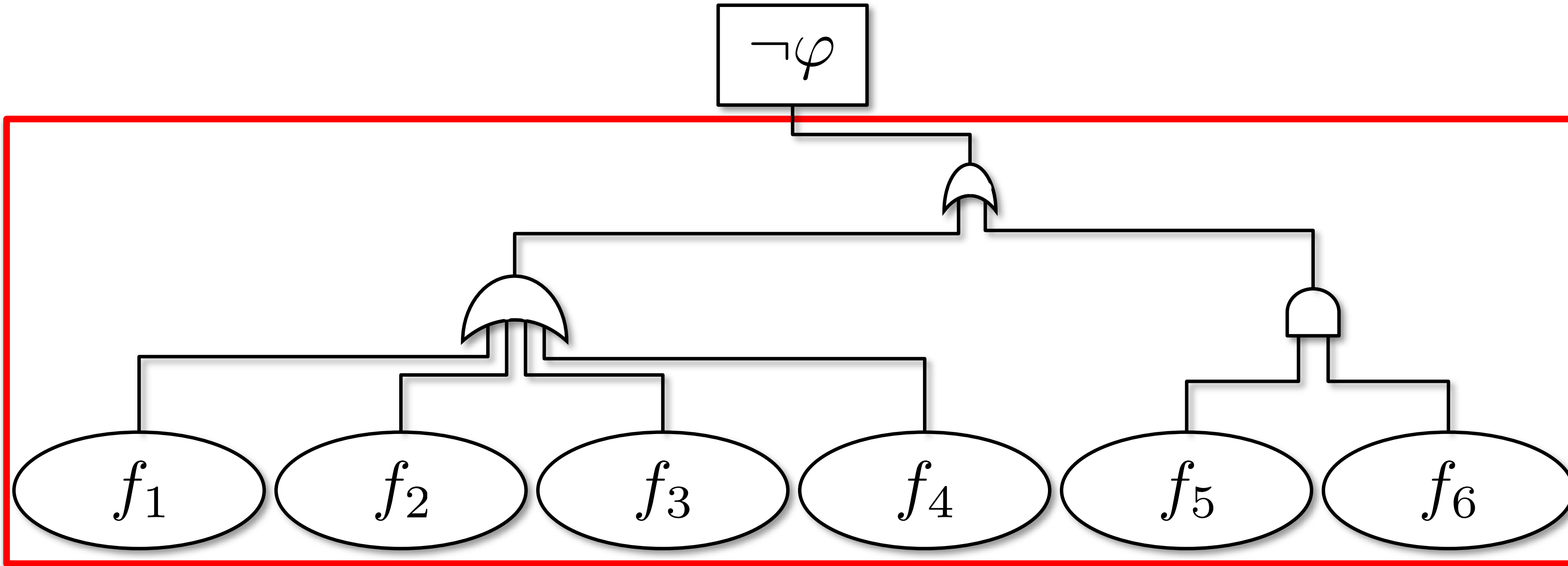


Fault Tree Analysis



- $\mathcal{F} := \{f_1, ..., f_{20}\}$
- $CS := \{\{f_1\}, .., \{f_4\}, \{f_5, f_6\}, \{f_1, f_8\}, \{f_2, f_3\}\}$
- $MCS := \{\{f_1\}, .., \{f_4\}, \{f_5, f_6\}\}$
- $MCS^{\top} := f_1 \vee f_2 \vee f_3 \vee f_4 \vee (f_5 \wedge f_6)$

Fault Tree Analysis

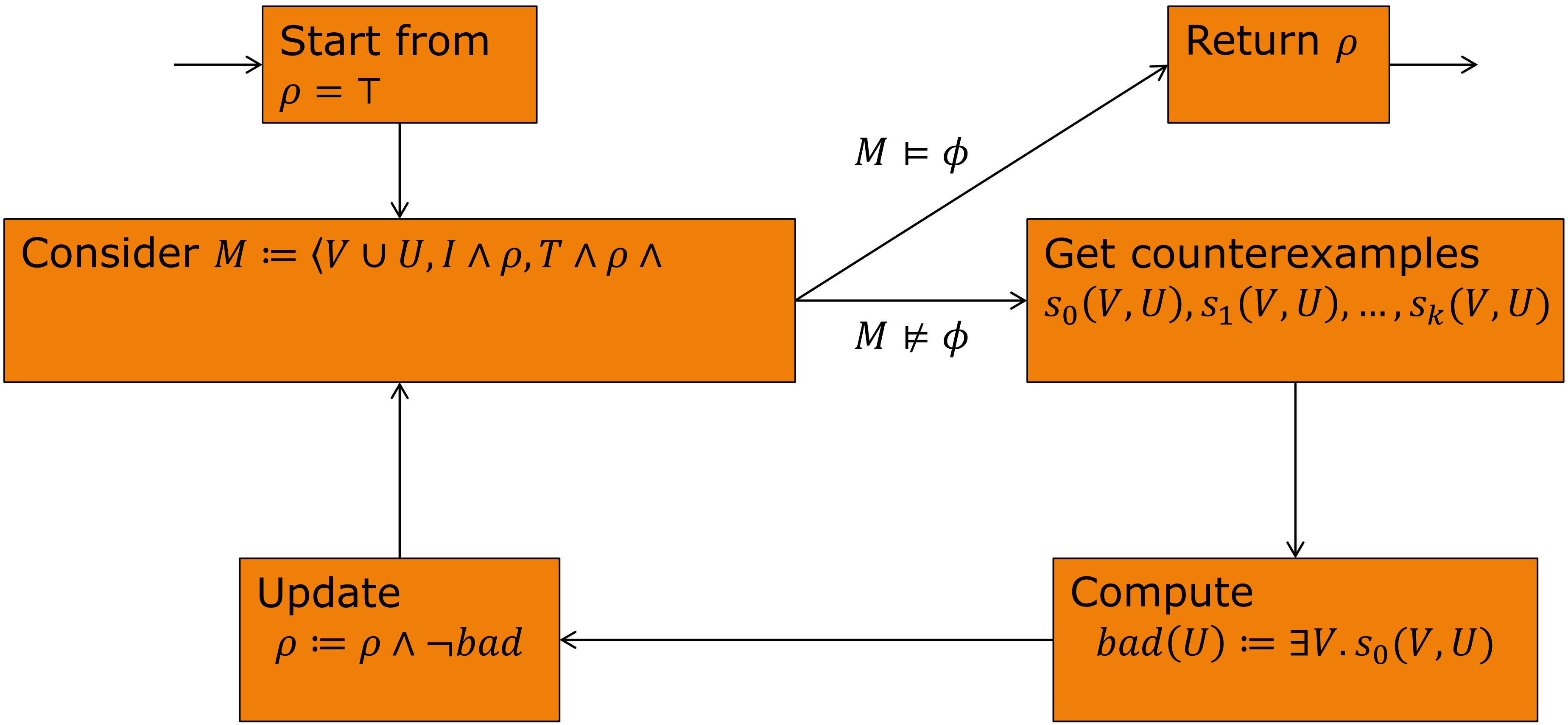


- $\mathcal{F} := \{f_1, ..., f_{20}\}$
- $CS := \{\{f_1\}, .., \{f_4\}, \{f_5$ … $\{f_1, f_8\}, \{f_2, f_3\}\}$
- $MCS := \{\{f_1\}, .., \{f_4\}$ … $\}$
- $MCS^\top := f_1 \vee f_2 \vee f_3 \vee f_4 \vee (f_5 \wedge f_6)$

CS computation as parameter synthesis

- Parameter synthesis problem:
  - Transition system extended with parameters X: $\langle V, I, T, X\rangle$ such that
    - I is a formula over $V \cup X$
    - T is a formula over $V \cup X \cup V'$
  - Valuation γ of X induces a transition system $\mathrm{M}_{\gamma} \coloneqq \langle V, \gamma(I), \gamma(T)\rangle$
  - Problem: find all γ such that $M_{\gamma} \models \phi$
    - Or dually find all γ such that $M_{\gamma} \not\models \phi$
- CS computation as parameter synthesis:
  - Faults $\mathcal{F}$ as parameters
  - M^X as parametric transition system
  - Find all assignments to $\mathcal{F}$ such that $M_{\gamma}^X \not\models \phi$

Parameter synthesis



Exploiting IC3 incrementality

- At each iteration:
  - $I := I \wedge \neg bad$
  - $T := T \wedge \neg bad$
- No need to restart from scratch
- IC3 can keep previous frames F_i
- Similarly, exploit incrementality in the underlying SAT/SMT solver

Requirements Analysis

Property correctness

- Standard problem: correctness of design against set of properties.
- Properties given as golden.
- Possible issues:
  - Properties wrongly formalized.
  - Properties may be abstract version of real requirements (to enable verification)
  - Set of properties incomplete.
- Same problems addressed by Requirements Engineering

Requirements engineering

- Old discipline (more than twenty years).
- Goal: precise and complete requirements.
- Many techniques on the different aspects:
  - management,
  - elicitation,
  - analysis,
  - validation.
- Why: errors in requirements take longer to find and correct than those inserted in later phases ⇒ higher cost
- More important in safety-critical application

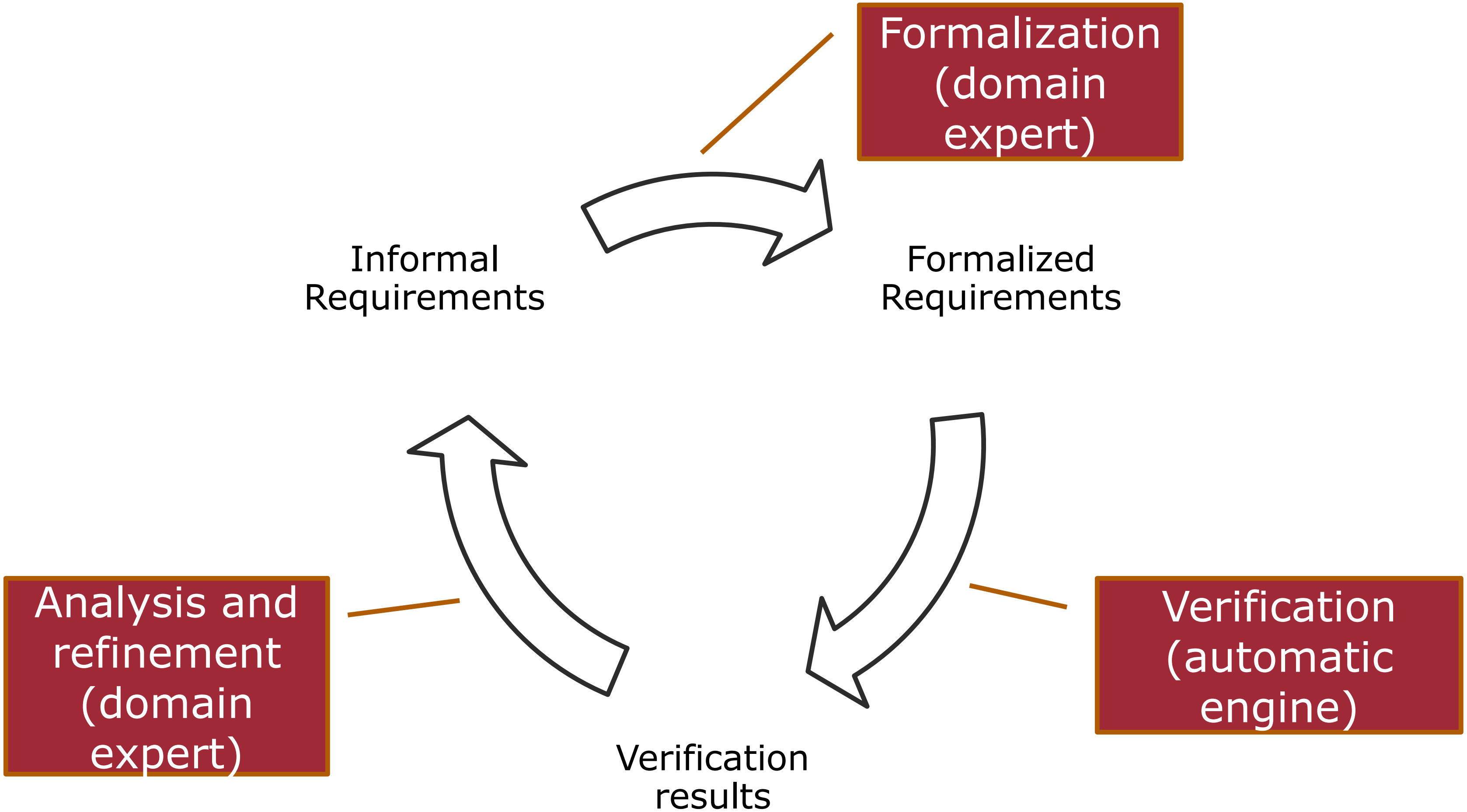
Vayager and Galileo examples

- Lutz in 1993 analyzed the Voyager and the Galileo software errors uncovered during integration and testing.
- Half errors were safety-related, half not.
- Most were functional faults: operating, conditional, or behavioral discrepancies with functional requirements.
- Primary cause (62% on Voyager, 79% on Galileo) is mis-understanding the requirements.

Standard Check List

- Analysis performed with a check list.
- Manual or automatic (based on linguistic techniques) to check if requirements are (IEEE Std 830-1993)
  - Complete: define all situations
  - Consistent: no contradictory statements
  - Correct: allow all and only desired behaviors
  - Modifiable: well structured, separation of concerns
  - Ranked: prioritized according to importance
  - Testable: specified tests
  - Traceable: identifier for each statement
  - Unambiguous: only one possible interpretation
  - Valid: all stakeholders must be able to understand, analyze and accept the requirement
  - Verifiable: ability to check design against the requirement.

Formal validation loop



Formal checks and feedback

- Formal properties capture the semantics of requirements
  - No model to refine the semantics of propositions
  - Requires rich property specification language
    - E.g. first-order temporal logic
- Formal checks:
  - Consistency: free of contradictions
  - Scenario compatibility: desired behaviors are admitted
  - Property entailment: undesired behaviors are not admitted
  - Realizability: an implementation is possible
  - Inherent vacuity: free of redundant/vacuous subformulas
  - Completeness: every situation is constrained
- Formal feedback:
  - Traces: witnesses of consistency, compatibility, property violation
  - Cores: subset of inconsistent, incompatible, property-entailing formulas

Reduction to Satisfiability

- Check if requirements are:
  - consistent, i.e. if they do not contain some contradiction
  - not too strict, i.e. if they do allow some desired behavior ψ_d
  - not too weak, i.e. if they rule out some undesired behavior ψ_u
- All reduced to satisfiability:
  - Consistency: $\bigwedge_i \phi_i$
  - Admit desired behavior: $\bigwedge_i \phi_i \wedge \psi_u$
  - Does not forbid undesired behavior: $\bigwedge_i \phi_i \wedge \psi_u$

Satisfiability procedure

- Reduce the problem to model checking
- ϕ is satisfiable iff $M_U \not\models \neg\phi$
  - Where M_U is the universal model
- Use standard automata-theoretic approach to model checking
  - ϕ_A Boolean abstraction of ϕ replacing $p(V)$ with Boolean v_p
  - M_ϕ obtained from M_{ϕ_A} by adding $\bigwedge_p v_p \leftrightarrow p$

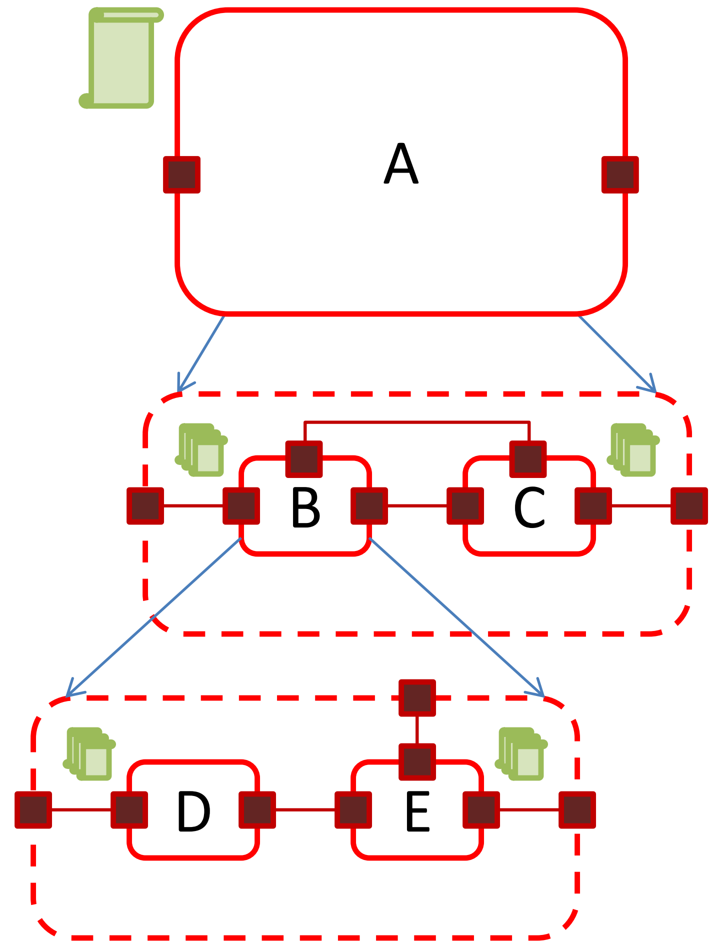
Contract Based Design

Component-based design

- So far, system seen as monolithic behavioral model
- A component can be defined as a unit of composition with contractually specified interfaces
  - Hides internal information
  - Defines interface to interact with the environment
- Component-based design ideal for
  - Separation of concerns
  - Independent development
  - Reuse of components
- First conceived for software, now popular also for system architectural design (SysML, AADL, AF3, Altarica, …)

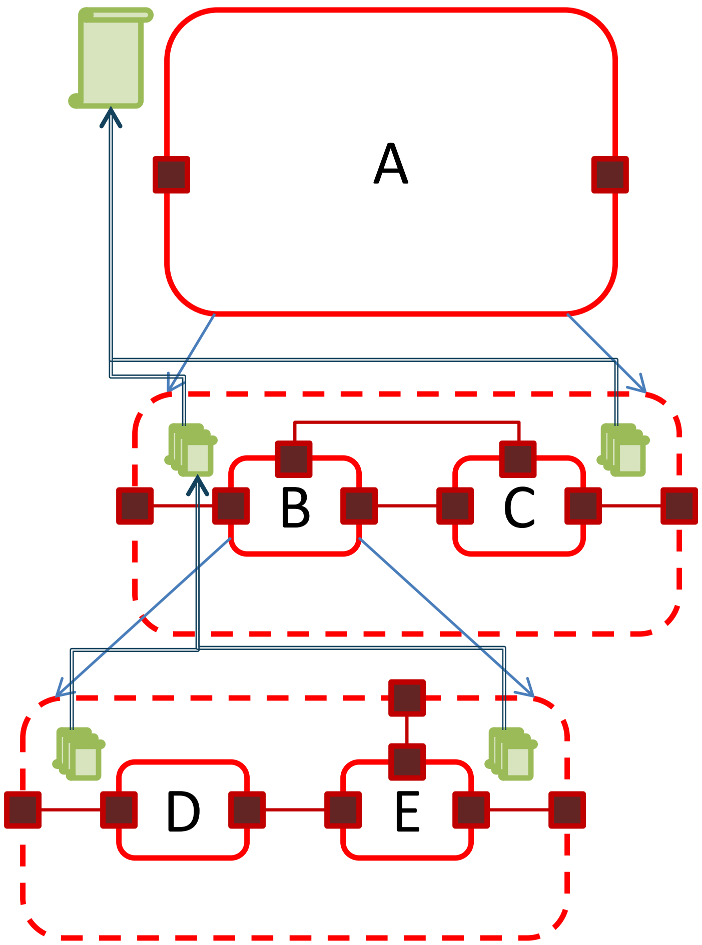
Specifying components with contracts

- Component hierarchically decomposed
- Requirements/properties specified at different levels of the hierarchy
- Contract: assumptions + guarantees
- Assumptions: properties expected to be satisfied by the environment
- Guarantees: properties expected to be satisfied by the component in response
- Correspond to pre/post conditions of standard SW contracts



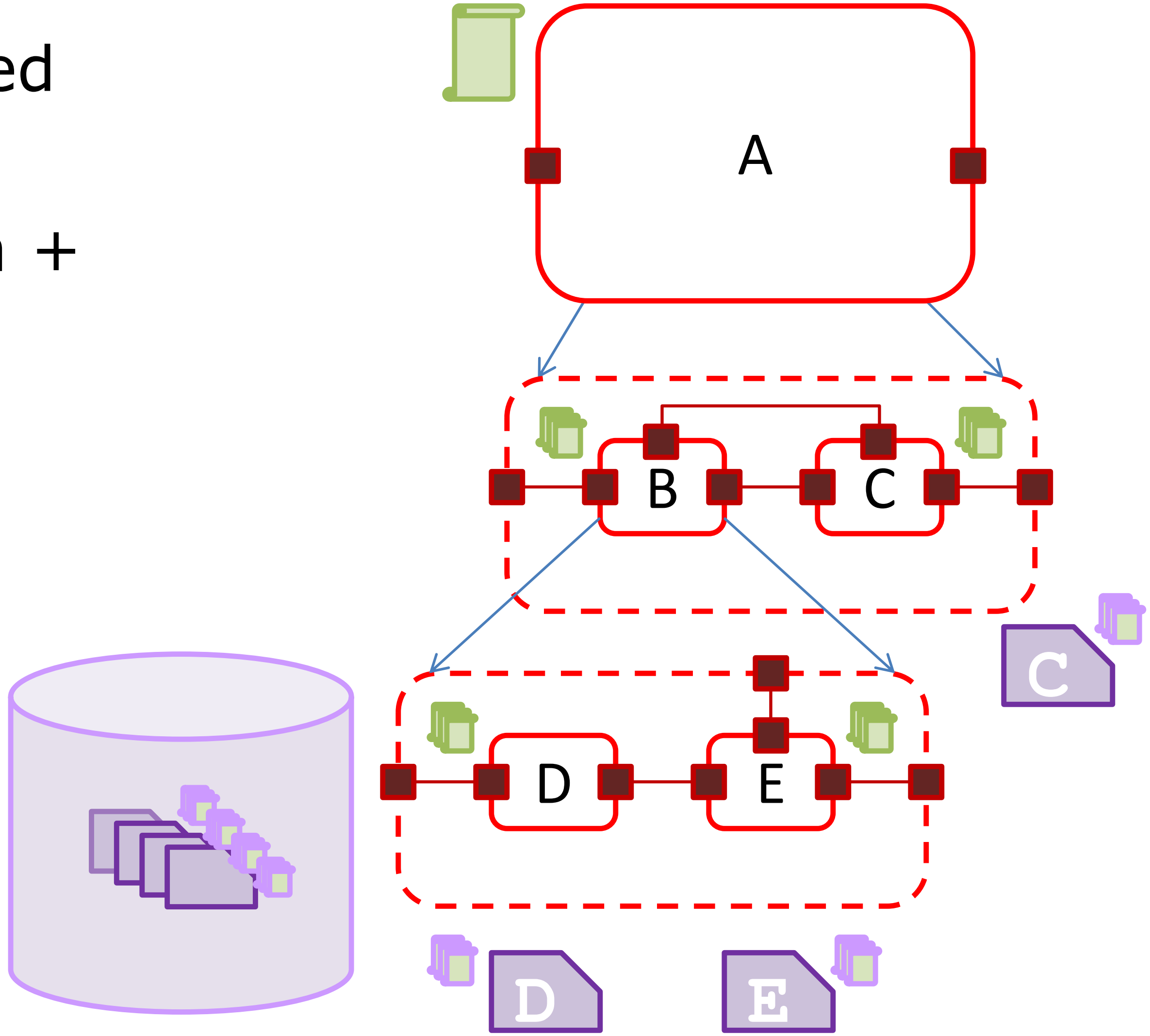
Stepwise refinement

- Specify components while designing
  - decomposing the specification based on the decomposition of the architecture
- Early check of requirements
  - Ensure the correctness of the decomposition
  - Does the contract of A follow from the contracts of B and C?
- Independent refinement:
  - Based on above check, B and C can be developed independently.

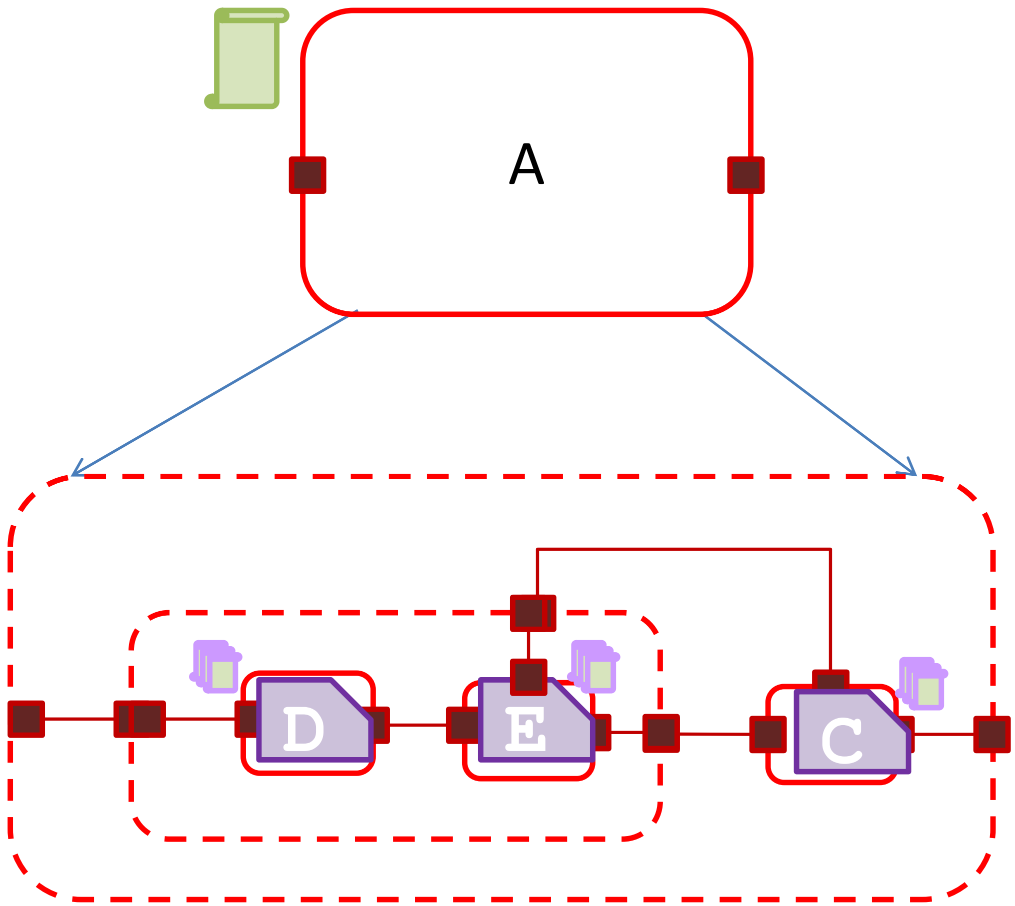


Component reuse

- Library of trusted components
- Implementation + contracts
- Pluggable?
  - compare contracts!



Compositional verification



Compositional verification techniques

- Compositional verification:
  - Prove properties of the components (for example, with model checking).
  - 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:

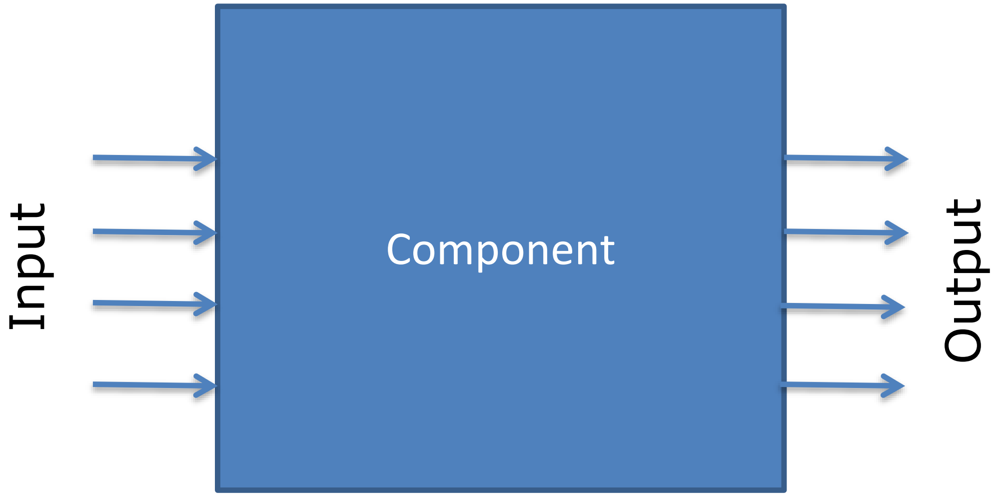
$$\gamma_P(P_1, P_2, \dots, P_n) = \rho_{\gamma_S}(P_1 \wedge P_2 \wedge \cdots \wedge P_n)$$

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

Contract-based compositional

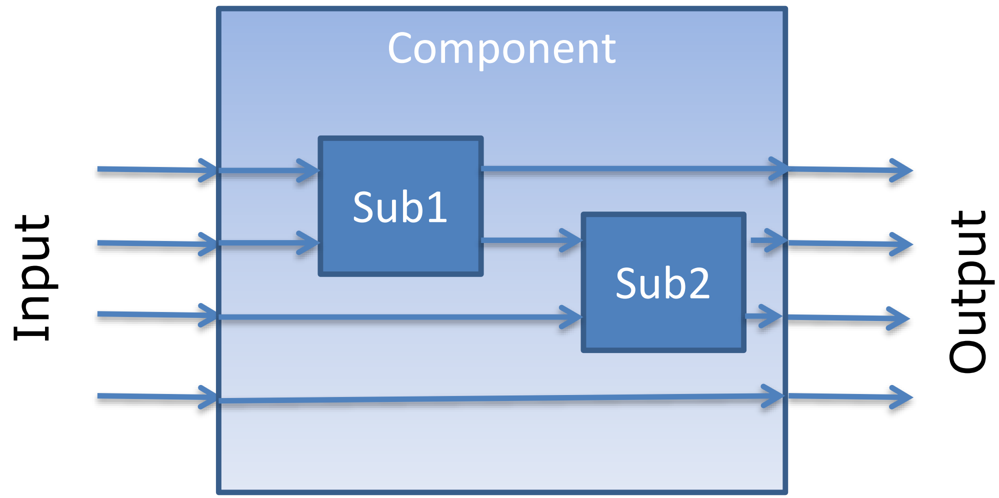
- Components interact with an environment.
  - Input/output data/events
  - Input controlled by environment, output controlled by component
- May be input enabled or possibly blocking.
- Blocking an input means constraining the environment.
  - The component can be used only in some environment (assumptions!)
- Compositional rule is not just an implication!
  - Guarantees of subcomponents must be stronger
  - Assumptions of subcomponents must be weaker
- Contract-based design requires a formal definition of components' syntax and semantics

Black-box component interface

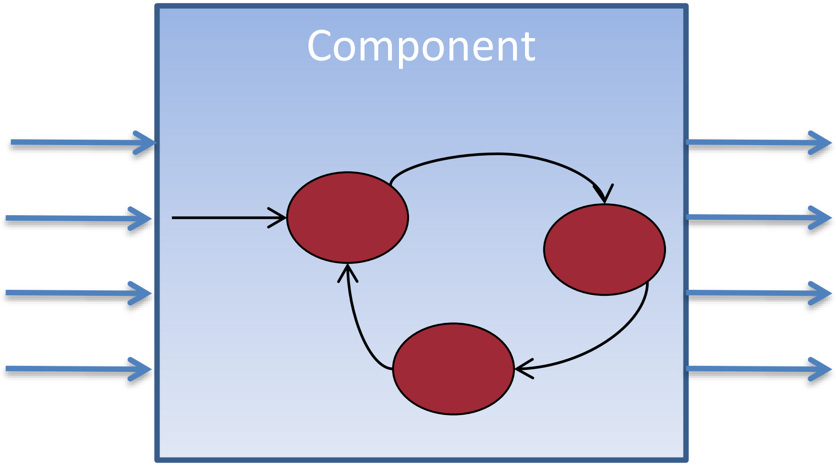


- A component interface defines boundary of the interaction between the component and its environment.
- Consists of:
  - Set of input and output ports (syntax)
    - Ports represent visible data and events exchanged with environment.
  - Set of traces (semantics)
    - Traces as sequences of events and assignments to data ports.

Glass-box component structure

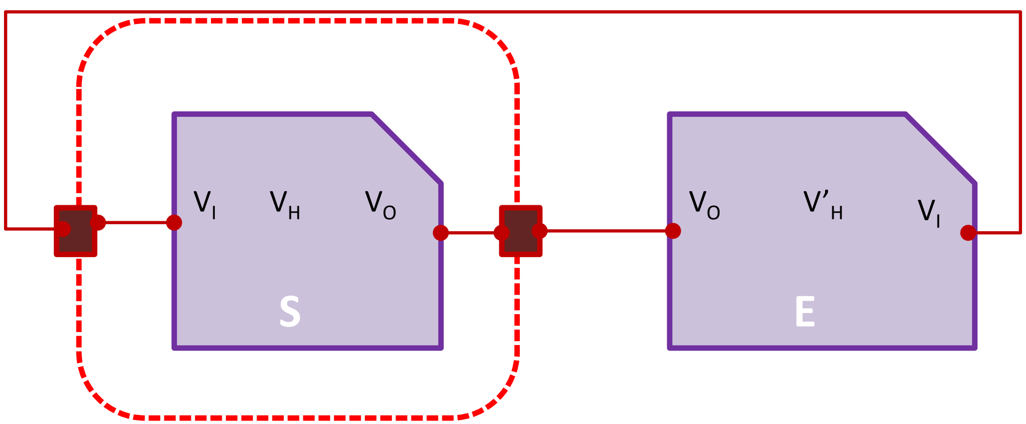


- A component has an internal structure.
- Architecture view:
  - Subcomponents
  - Inter-connections
  - Delegations
- State-machine view:
  - Internal state
  - Internal transitions
  - Language over the ports



Implementation and Environment

- I_S: input ports of component S
- O_S: output ports of S
- $V_S = I_S \cup O_S$: all ports of S
- Implementation/environment of S: transition system $\langle V, I, T \rangle$ with $V_S \subseteq V$



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

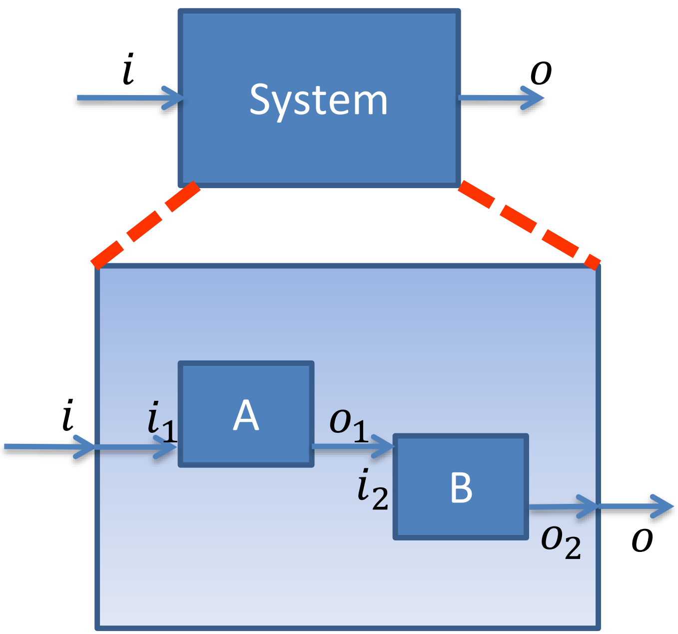
  Into
  - Output ports of the composite component
  - Input ports of the subcomponents.

Composite components and connections

- Sub_S: subcomponents of S
- Connection

$$\gamma: (O_S \cup \bigcup_{S' \in Sub_S} I_{S'}) \rightarrow (I_S \cup \bigcup_{S' \in Sub_S} O_{S'})$$

- Example:
  - $\gamma(o) = o_2$
  - $\gamma(i_2) = o_1$
  - $\gamma(i_1) = i$



Composite components and connections

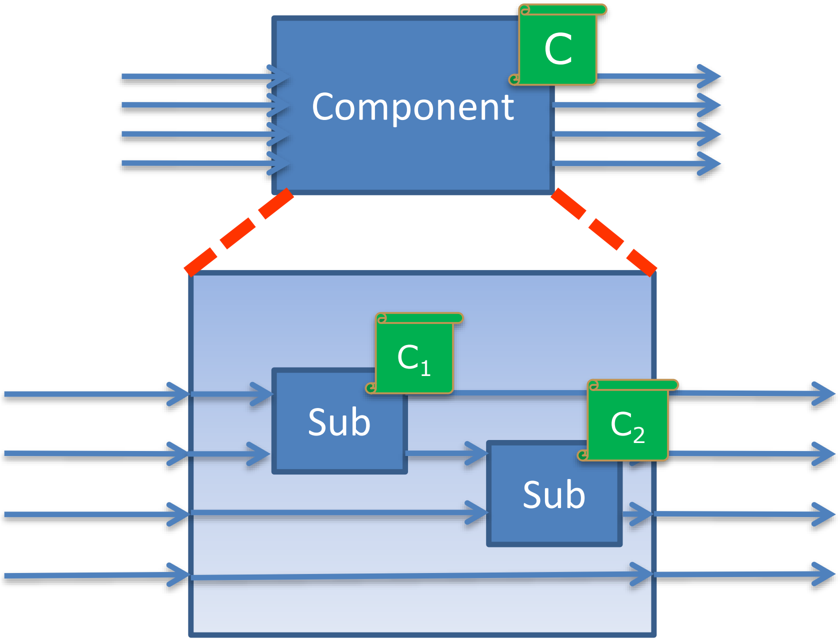
- Standard synchronous product:
  - $M_1 = \langle V_1, I_1, T_1 \rangle$ and $M_2 = \langle V_2, I_2, T_2 \rangle$
  - $M_1 \times M_2 := \langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$
- With connection γ:
  - $M_1 \times_\gamma M_2 := \langle \gamma(V_1 \cup V_2), \gamma(I_1 \wedge I_2), \gamma(T_1 \wedge T_2) \rangle$
  - Where
    - $\gamma(V) := \{v | v \in V \setminus dom(\gamma) \; or \; v = \gamma(w) \; for \; some \; w \in \mathrm{V}\}$
    - $\gamma(\phi) := \phi[v \mapsto \gamma(v)]$
- Given implementations $M_1, \ldots, M_n$ for $Sub_S = S_1, \ldots, S_n$, and environment E
  - Composite implementation of S:
    - $M_1 \times_\gamma \ldots \times_\gamma M_n$
  - Composite environment of S_i:
    - $M_1 \times_\gamma \ldots \times_\gamma M_{j \neq i} \times_\gamma \ldots \times_\gamma M_n \times_\gamma E$

LTL contracts

- A contract of component S is a pair $\langle A, G \rangle$ of LTL formulas over V_S
  - A is the assumption
  - G is the guarantee
- Env is a correct environment iff $Env \models A$
- Imp is a correct implementation iff $Imp \models A \rightarrow G$

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 $C_1 = \langle \alpha_1, \beta_1 \rangle, \dots, C_n = \langle \alpha_n, \beta_n \rangle, C = \langle \alpha, \beta \rangle$
- Proof obligations for $\{C_i\} \preccurlyeq C$:
  - $\gamma\left(\left(\bigwedge_{1 \le j \le n}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \beta)\right)$
  - $\gamma\left(\left(\bigwedge_{2 \le j \le n}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \alpha_1)\right)$
  - …
  - $\gamma\left(\left(\bigwedge_{1 \le j \le n, j \ne i}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \alpha_i)\right)$
  - …
  - $\gamma\left(\left(\bigwedge_{1 \le j \le n-1}(\alpha_j \rightarrow \beta_j)\right) \rightarrow (\alpha \rightarrow \alpha_n)\right)$

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

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

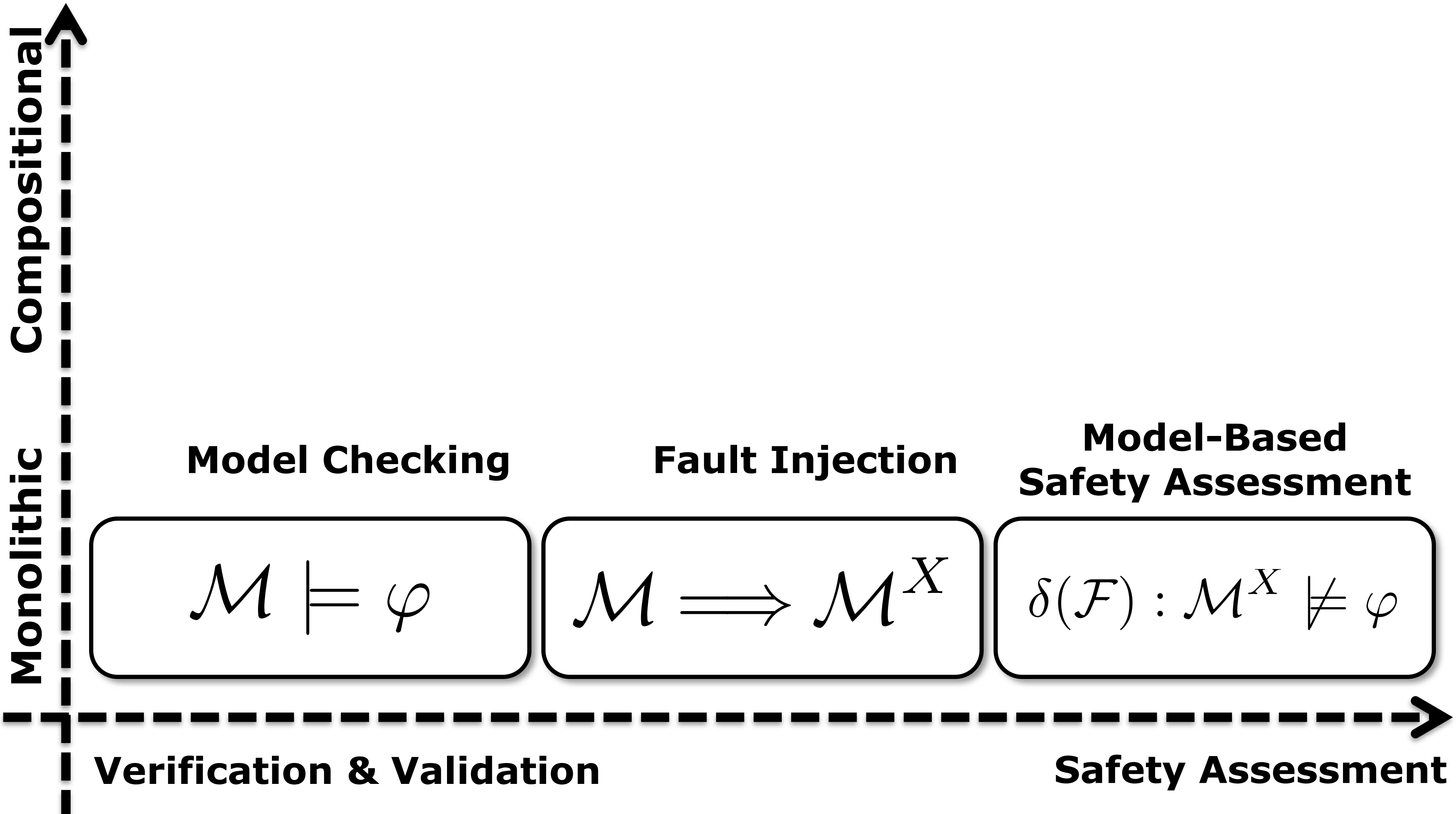
$\big(B \wedge G(A \rightarrow XB) \wedge A \wedge G(B \rightarrow XA)\big) \Rightarrow G(A \wedge B)$ is true

Contract Based Safety Assessment

Contract-Based Safety Assessment

- “Monolithic” safety assessment artifacts e.g., minimal cutsets, might be not easily understandable
- Need for more structured safety artifacts e.g., hierarchically organized fault trees
- Leverage the architectural decomposition of contract-based design
- Perform automated Safety Assessment on a Contract-Based system decomposition

Formal Verification, Validation, and Safety Assessment

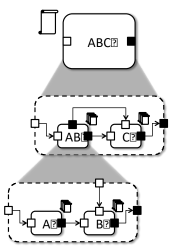


Formal Verification, Validation, and Safety Assessment

Compositional

Monolithic

Contract-Based Design



Model Checking

$\mathcal{M} \models \varphi$

Fault Injection

$\mathcal{M} \Longrightarrow \mathcal{M}^X$

Model-Based Safety Assessment

$\delta(\mathcal{F}) : \mathcal{M}^X \not\models \varphi$

Verification & Validation

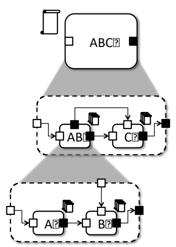
Safety Assessment

Formal Verification, Validation, and Safety Assessment

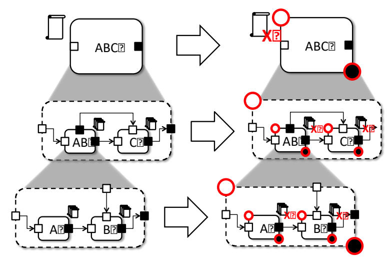
Compositional

Monolithic

Contract-Based Design



Contract-Based Fault Injection



Model Checking

$$\mathcal{M} \models \varphi$$

Fault Injection

$$\mathcal{M} \Longrightarrow \mathcal{M}^X$$

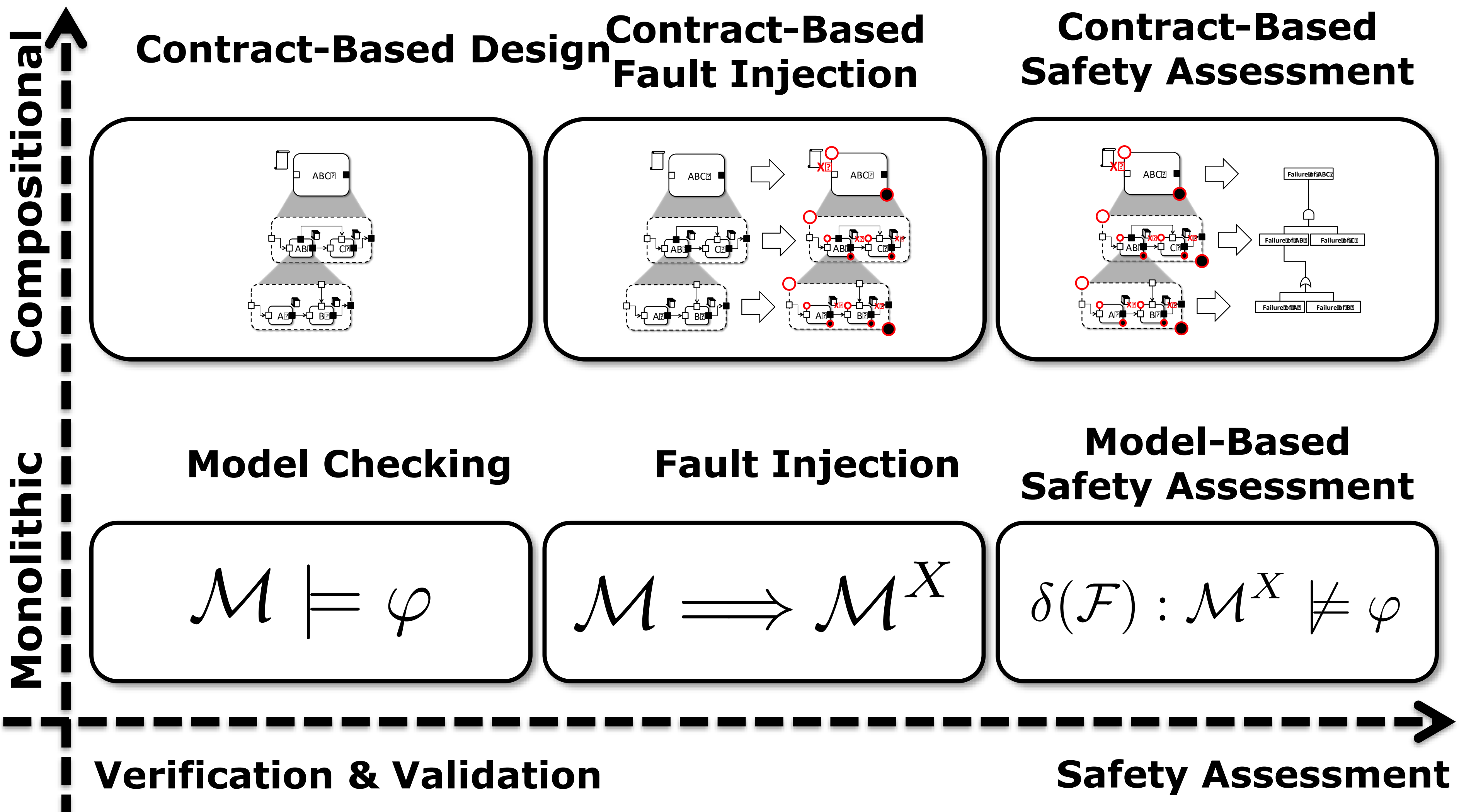
Model-Based Safety Assessment

$$\delta(\mathcal{F}) : \mathcal{M}^X \not\models \varphi$$

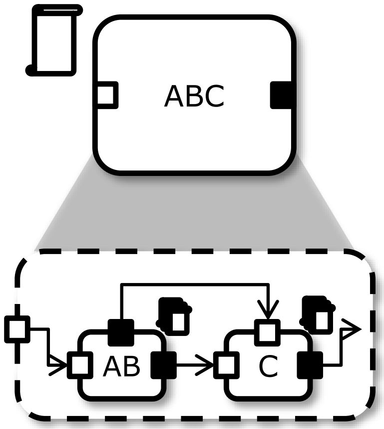
Verification & Validation

Safety Assessment

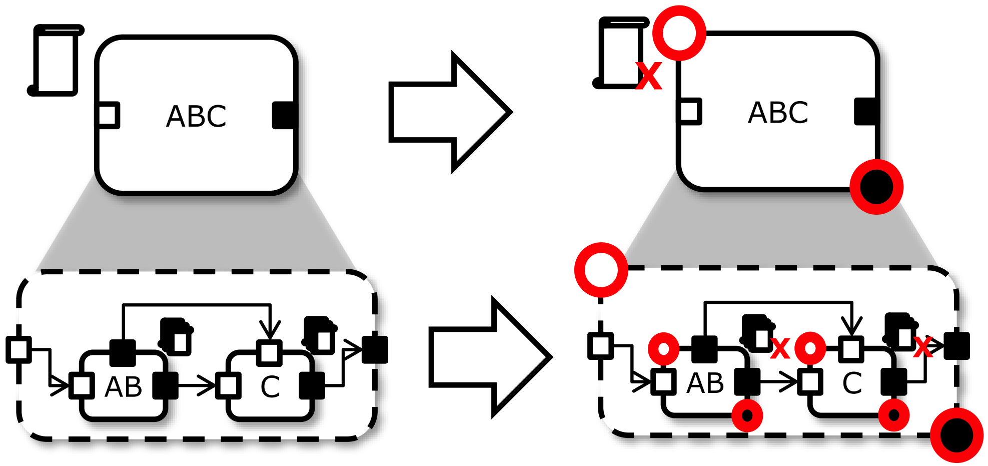
Formal Verification, Validation, and Safety Assessment



Contract-Based Safety Assessment

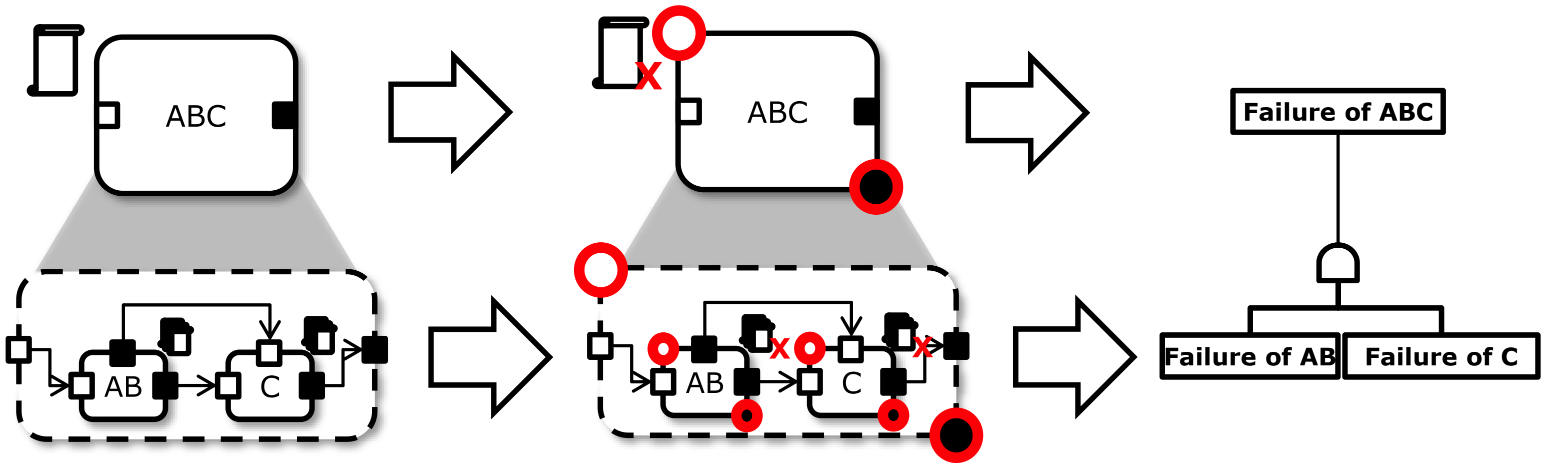


Contract-Based Safety Assessment



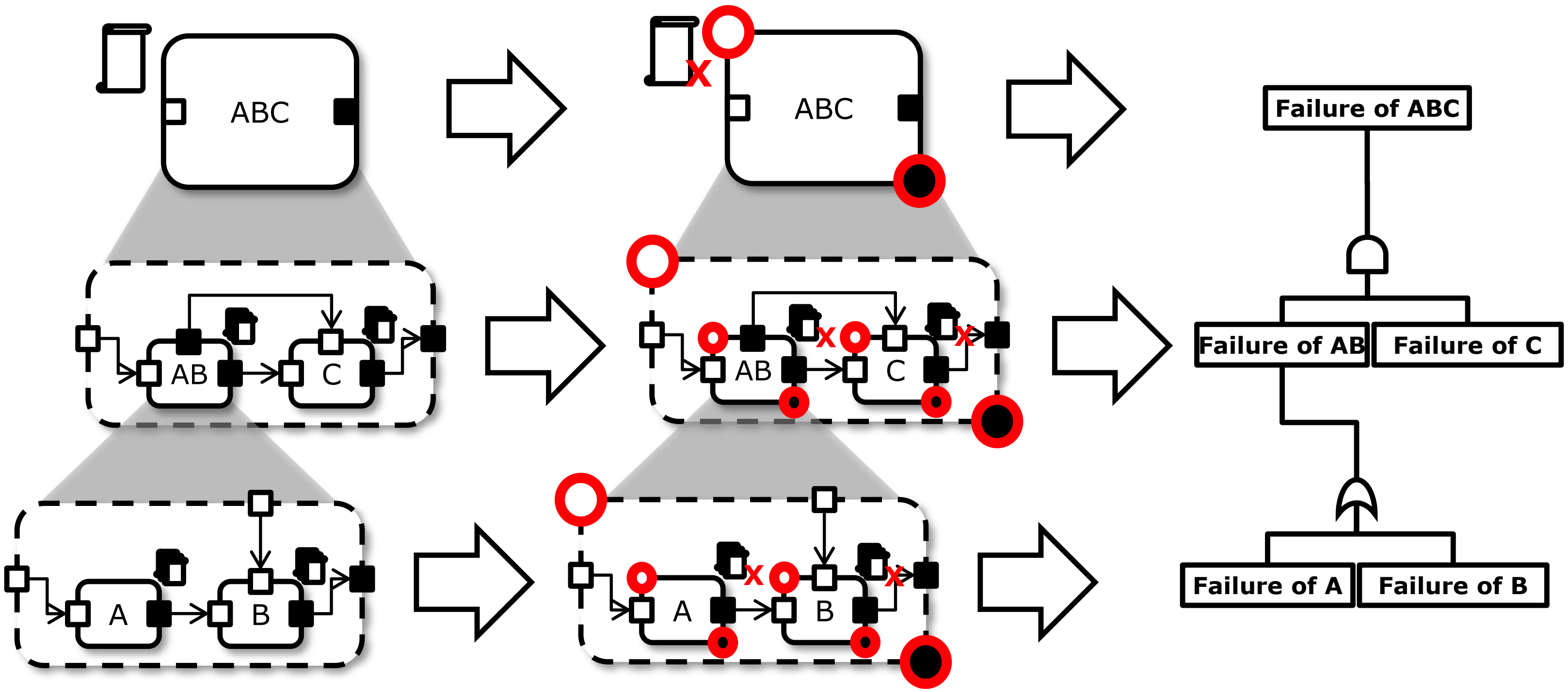
- Extension of contracts (fault injection) from a Contract-Based decomposition

Contract-Based Safety Assessment



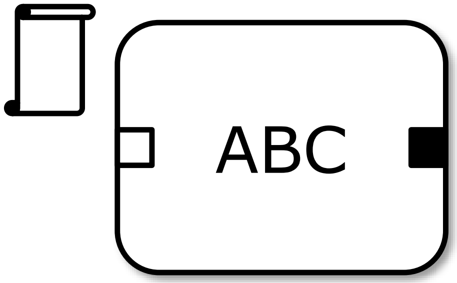
- Extension of contracts (fault injection) from a Contract-Based decomposition
- Automated Formal Safety Assessment i.e., Fault Tree Analysis

Contract-Based Safety Assessment



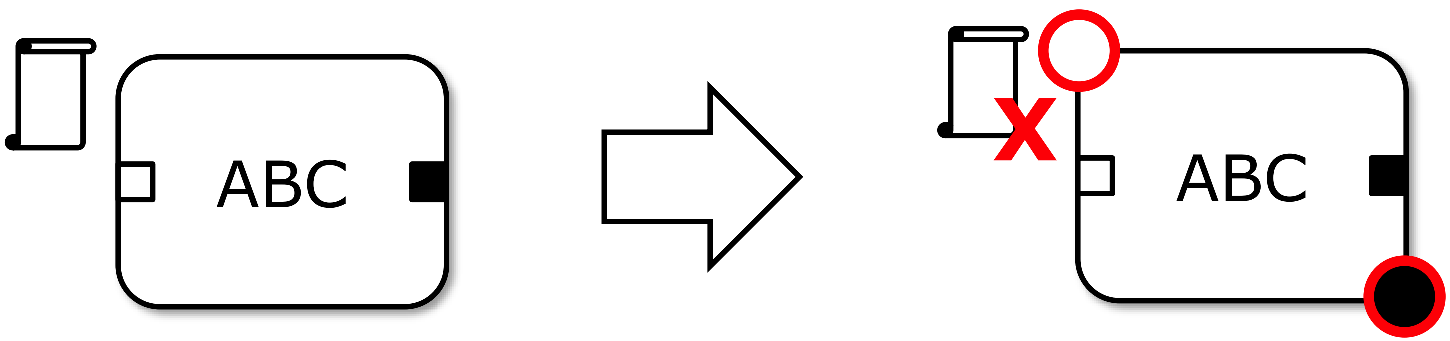
- Extension of contracts (fault injection) from a Contract-Based decomposition
- Automated Formal Safety Assessment i.e., Fault Tree Analysis
- Support for components refinement

Contract-Based Fault Injection



$$\langle \mathcal{A}, \mathcal{G} \rangle$$

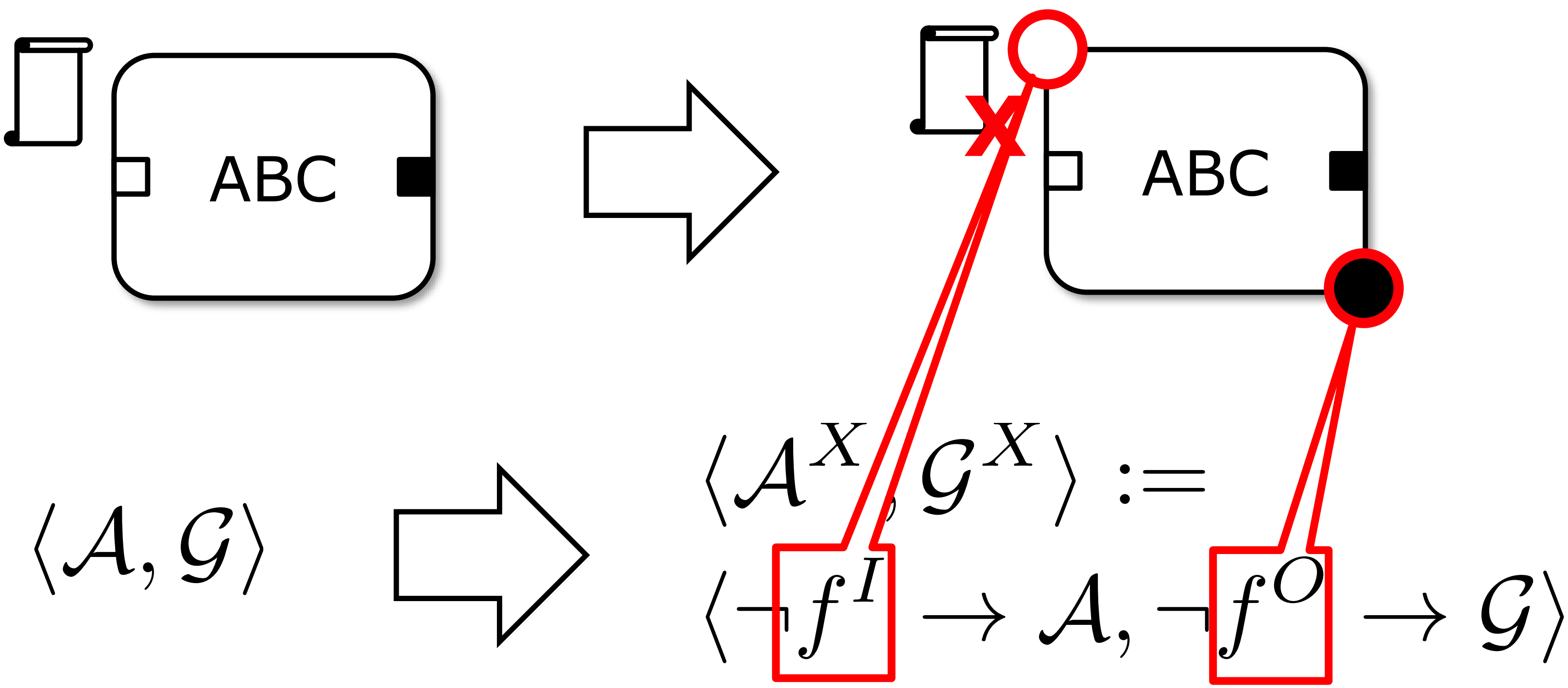
Contract-Based Fault Injection



$$\langle \mathcal{A}, \mathcal{G} \rangle \Rightarrow \langle \mathcal{A}^X, \mathcal{G}^X \rangle := \langle \neg f^I \rightarrow \mathcal{A}, \neg f^O \rightarrow \mathcal{G} \rangle$$

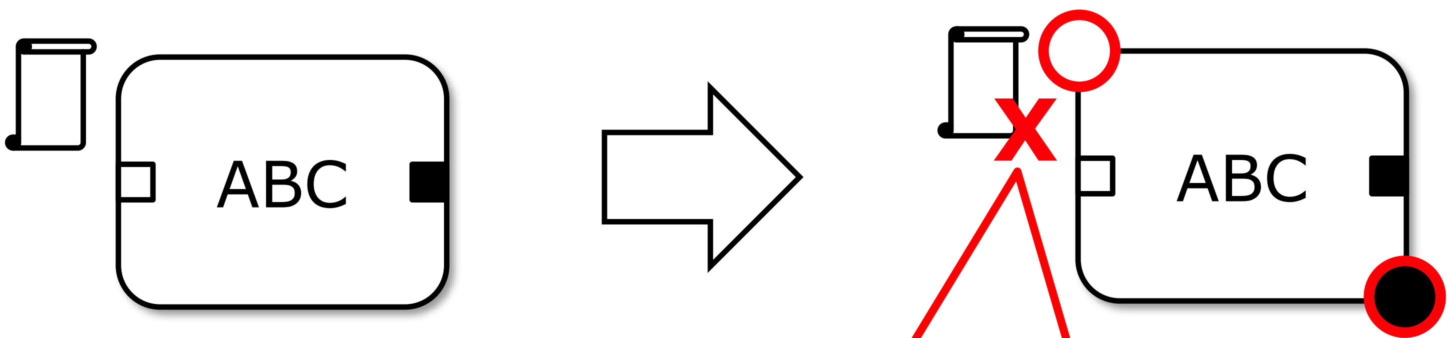
- Additional input and output failure ports
- Contract extension

Contract-Based Fault Injection



- **Additional input and output failure ports**
- Contract extension

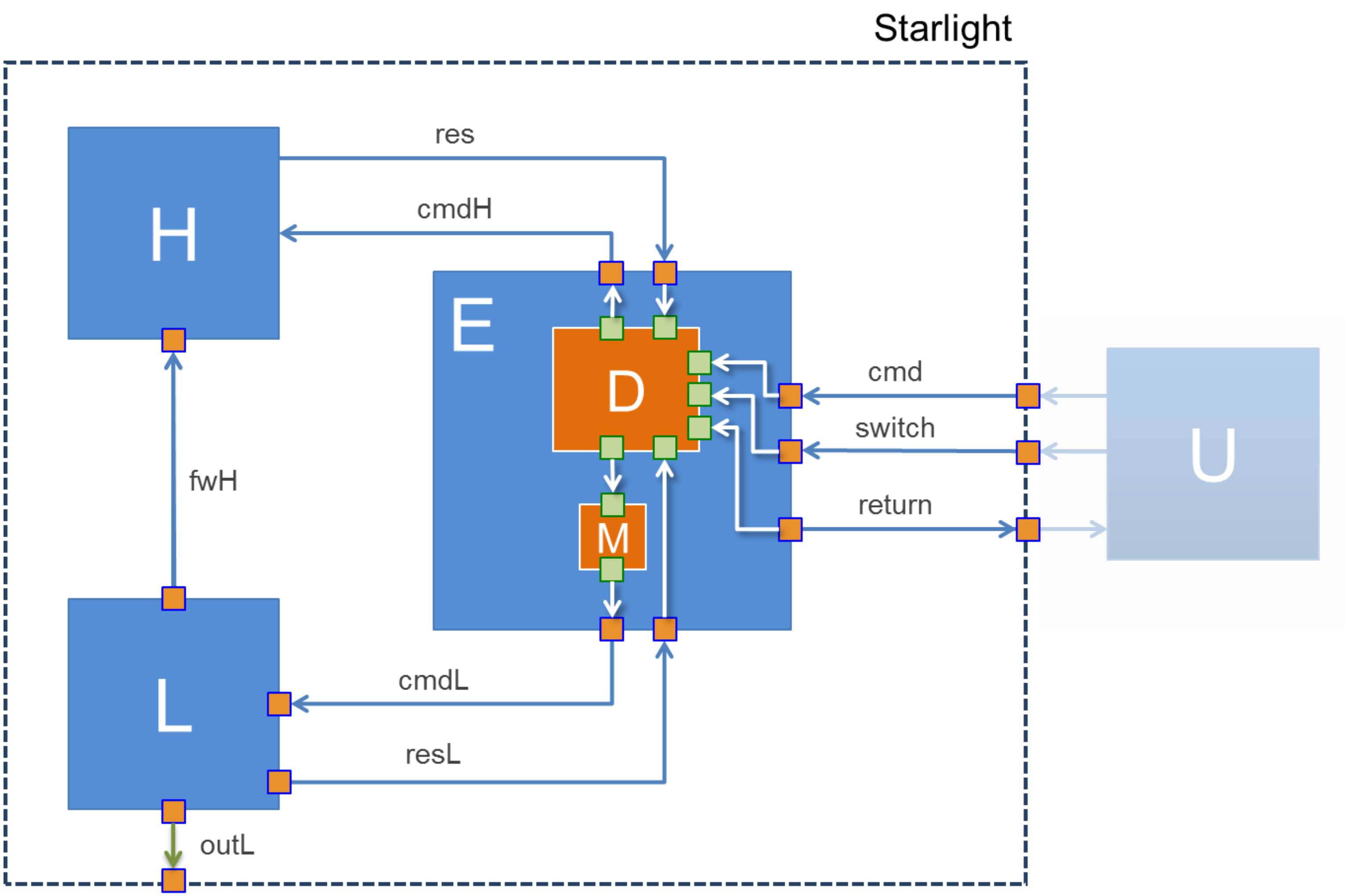
Contract-Based Fault Injection



$$\langle \mathcal{A}, \mathcal{G} \rangle \Rightarrow \langle \mathcal{A}^X, \mathcal{G}^X \rangle := \langle \neg f^I \rightarrow \mathcal{A}, \neg f^O \rightarrow \mathcal{G} \rangle$$

- Additional input and output failure ports
- **Contract extension**

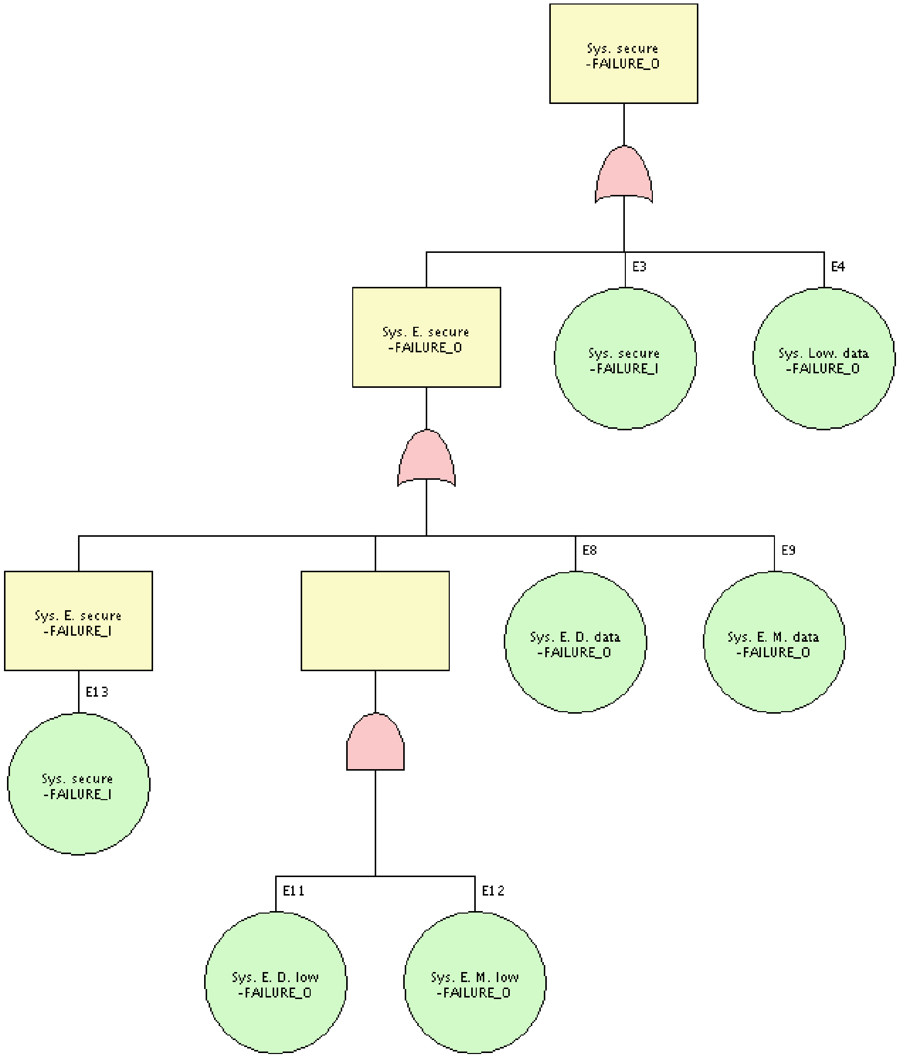
Starlight Example



Starlight reqs formalization

- Req-Sys-secure: No high-level data shall be sent by L to the external world.
  - Formal-Sys-secure: never is_high(last_data(outL))
- Req-User-secure: The user shall switch the dispatcher to high before entering high-level data.
  - Formal-User-secure: always ((is_high(last_data(cmd))) implies ((not switch_to_low) since switch_to_high))
- Proved system guarantess Formal-Sys-secure assuming Formal-User-secure.
- Req-Sys-safe: No single failure shall cause a loss of Req-Sys-secure.

Starlight fault tree for secure req

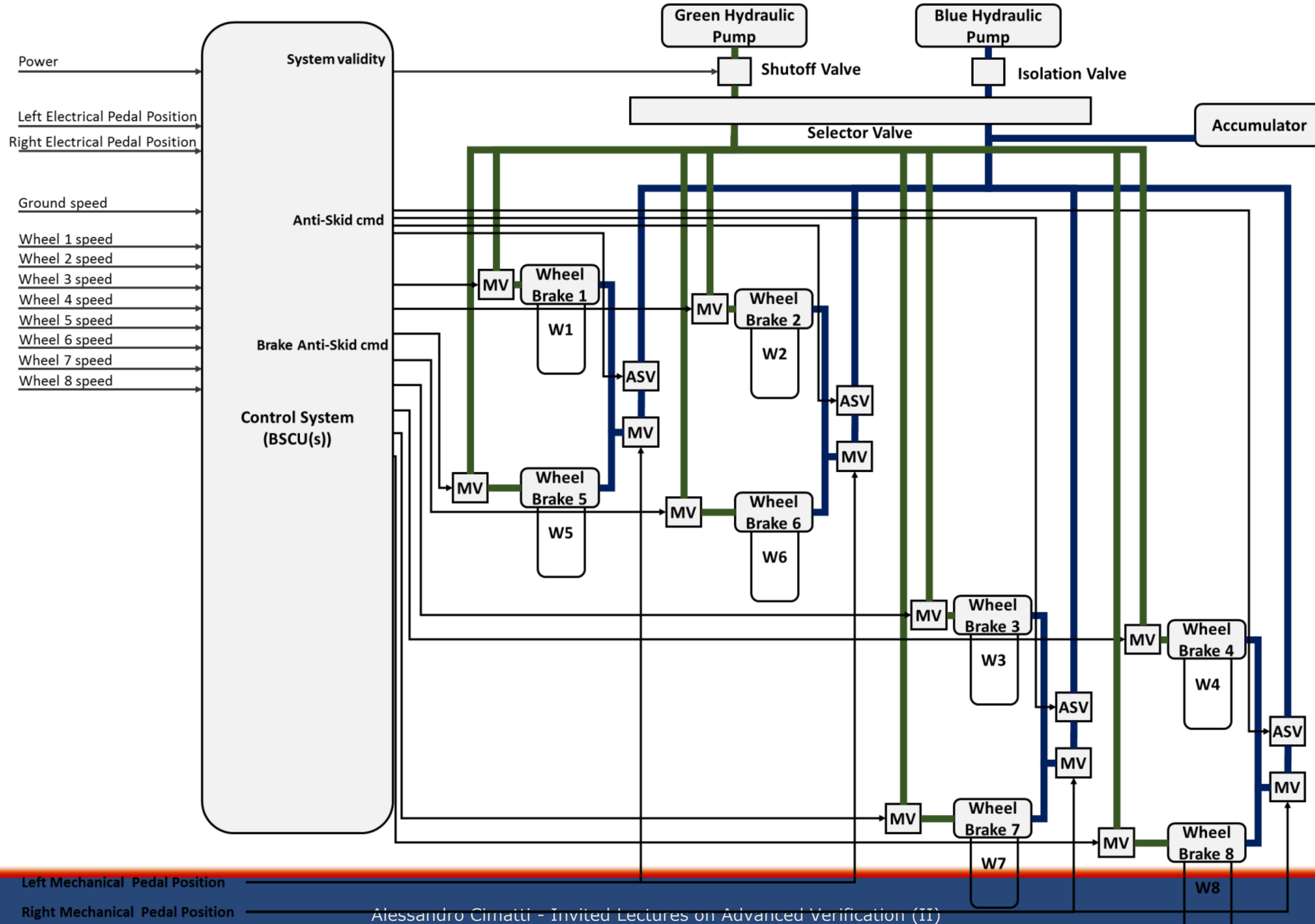


Case-Studies

AIR6110 Wheel Braking System

- Joint scientific study with Boeing
- Context
  - Aerospace systems become more complex and integrated
  - Safety assessment process is critical
    - Evaluate whether a selected design is sufficiently robust with respect to the criticality of the system and faults occurrence
- Objectives:
  - Analyze the system safety through mathematical models and techniques
  - Demonstrate the usefulness and suitability of these techniques for improving the overall traditional development and supporting aircraft certification
- Case study:
  - Aerospace Information Report 6110:
    - Traditional Contiguous Aircraft/System Development Process Example
  - Wheel Brake System of a fictional dual-engine aircraft
    - 300-350 passengers, 5h max of flight
    - 2 main landing gears (4 wheels each)

WBS: Overview



WBS: Adopted approach

ANALYSIS / MODELING	V & V	Safety Assessment: Fault extension	Safety Assessment: Fault trees computation
Architecture decomposition & Contracts *ocra language*	• Automatic contract refinement verification **OCRA**	• Automatic fault extension **OCRA**	• Automatic hierarchical fault tree generation • Over-approximation **OCRA**
Semi-automatic Generation ↓ **Behavioral Implementation** (Leaf components & System) *smv language*	• Automatic compositional verification **OCRA** • Automatic monolithic verification **nuXmv**	• Failure modes defined by the user • Generation of the extended system implementation **xSAP**	• Automatic flat fault tree generation **xSAP**
M	$M \models \varphi$	$M \rightsquigarrow M_{[F]}$	$\delta(F) : M_{[F]} \not\models \varphi$

WBS: Conclusion

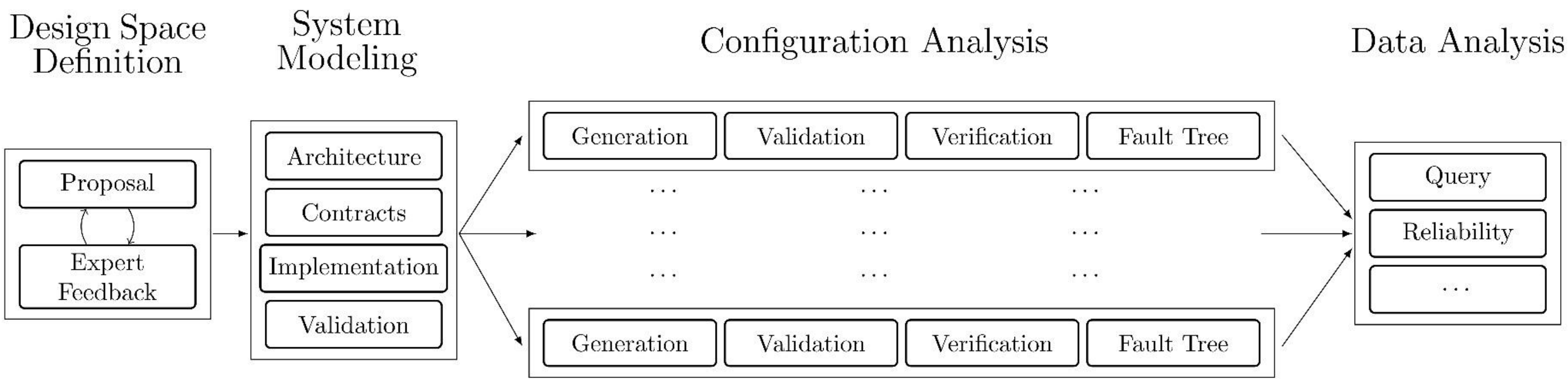
- Results:
  - Cover the process described in AIR6110 with formal methods
  - Production of modular descriptions of 5 architectures variants
    - Analysis of their characteristics in terms of a set of requirements expressed as properties
    - Production of more than 3000 fault trees
    - Production of reliability measures
  - Detection of an unexpected flaw in the process
    - Detection of the wrong position of the accumulator earlier in the process

- Lessons learned:
  - Going from informal to formal allows highlighting the missing information of the AIR6110 to reproduce the process
  - OCRA modular modeling allows a massive reuse of the design through architectures variant
  - Automated and efficient engines as IC3 is a key factor
  - MBSA is crucial in this context:
    - Automatic extension of the nominal model with faults
    - Automatic generation of artifacts eases the analysis and the architecture comparison in terms of safety

NASA NextGen Air Traffic Control

- Problem:
  - 4x airspace traffic in the next 20 years
  - Currently technology cannot scale
  - Need to increase automation, while preserving safety
- Apply Formal Methods to study the quality and Safety of many design proposals concerning the allocation of tasks between Air and Ground
- Objective:
  - Highlight Implicit assumptions
  - Model and Study a design space with more than **1600** proposals
  - Time-Frame: 12 Man-Month
- Joint project with NASA Ames and Langley

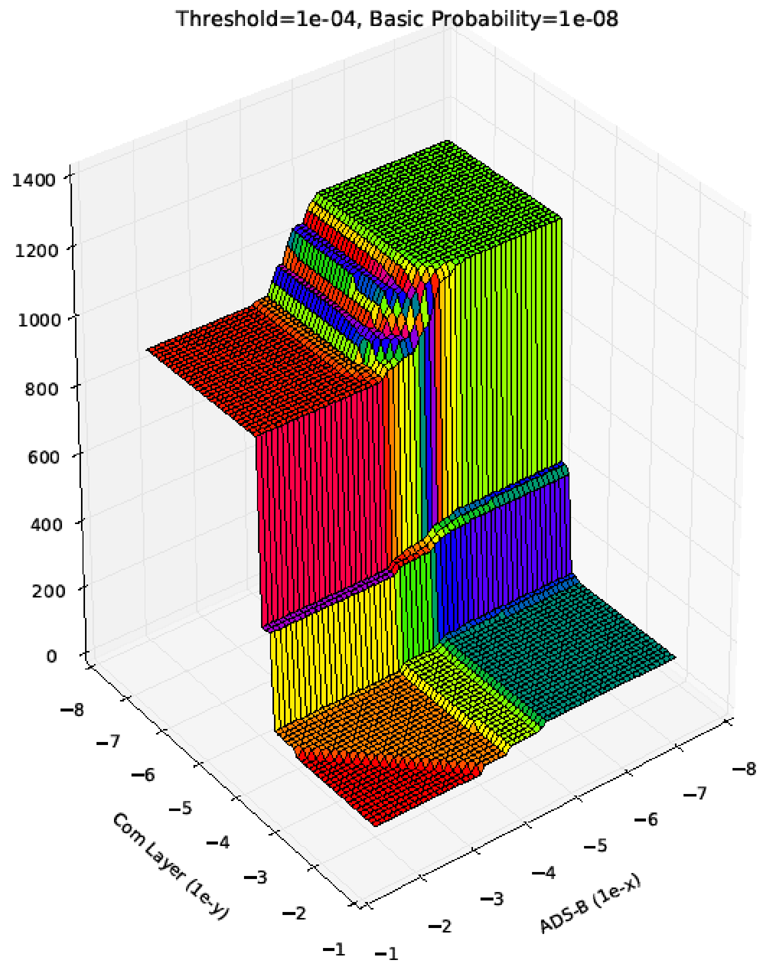
NextGen: Proposed Solution



- Identify dimensions of the design space
- Use a parametric model to encode all designs (symbolically)
- Unified design architecture makes it possible to push complexity into the leaf components
- Use contracts to validate components behavior
- Perform Model-Checking against interesting properties, and rank solutions based on their "quality"
- Perform Fault-Tree analysis to understand the resilience to faults

NextGen: Results

- Independently reproduced 2 known problems
- High-lighted a mismatch in requirements for one design proposal
- Results discussed and validated by NASA engineers
- Lessons Learned:
  - Model Validation is a key step
  - Technology is mature to tackle problems of realistic size
  - Lots of data: Need better ways to present complex results in an accessible way


Threshold=1e-04, Basic Probability=1e-08
1400
1200
1000
800
600
400
200
0
−8
−7
−6
−5
−4
−3
−2
−1
Com Layer (1e-y)
−1
−2
−3
−4
−5
−6
−7
−8
ADS-B (1e-x)

Wrap-up

Lecture Summary

- Importance of Safety Assessment
- Contract-Based Design
  - Specify & Validate Requirement
  - Decompose Requirements onto Architecture
  - Implement Leaf components
  - Functional correctness guaranteed by Contract-Decomposition
- CBSA: Leverage contracts to perform Safety Assessment

Readings

A list of suggested readings on the topics of the course. The list is not meant to be complete.

- Model-Based Safety Assessment:
  - Marco Bozzano, Adolfo Villafiorita: Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform. SAFECOMP 2003: 49-62
  - Marco Bozzano, Alessandro Cimatti, Francesco Tapparo: Symbolic Fault Tree Analysis for Reactive Systems. ATVA 2007: 162-176
  - Marco Bozzano, Alessandro Cimatti, Alberto Griggio, Cristian Mattarei: Efficient Anytime Techniques for Model-Based Safety Analysis. CAV (1) 2015: 603-621
- Parameter Synthesis:
  - Alessandro Cimatti, Alberto Griggio, Sergio Mover, Alessandro Cimatti: Parameter synthesis with IC3. FMCAD 2013: 165-168

Readings

- Requirements Formalization and Validation:
  - Alessandro Cimatti, Marco Roveri, Alessandro Cimatti: Requirements Validation for Hybrid Systems. CAV 2009: 188-203
  - Alessandro Cimatti, Marco Roveri, Angelo Susi, Alessandro Cimatti: Validation of requirements for hybrid systems: A formal approach. ACM Trans. Softw. Eng. Methodol. 21(4): 22 (2012)
- Compositional Verification:
  - Kenneth L. McMillan: Circular Compositional Reasoning about Liveness. CHARME 1999: 342-345
  - Anubhav Gupta, Kenneth L. McMillan, Zhaohui Fu: Automated assumption generation for compositional verification. Formal Methods in System Design 32(3): 285-301 (2008)
  - Anvesh Komuravelli, Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan: Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. FMCAD 2015: 89-96

Readings

- Contract-Based Design with Temporal Logics:
  - Alessandro Cimatti, Alessandro Cimatti: A Property-Based Proof System for Contract-Based Design. EUROMICRO-SEAA 2012: 21-28
  - Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, Andrzej Wasowski: Moving from Specifications to Contracts in Component-Based Design. FASE 2012: 43-58
  - Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, Lui Sha: Compositional Verification of Architectural Models. NASA Formal Methods 2012: 126-140
  - Alessandro Cimatti, Alessandro Cimatti: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Program. 97: 333-348 (2015)
  - Thi Thieu Hoa Le, Roberto Passerone, Ulrich Fahrenberg, Axel Legay: A tag contract framework for modeling heterogeneous systems. Sci. Comput. Program. 115-116: 225-246 (2016)
  - Alessandro Cimatti, Ramiro Demasi, Alessandro Cimatti: Tightening a Contract Refinement. SEFM 2016
  - Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, Cesare Tinelli: CoCoSpec: A mode aware contract language. SEFM 2016

Readings

- Contract-Based Safety Assessment:
  - Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, Alessandro Cimatti: Formal Safety Assessment via Contract-Based Design. ATVA 2014: 81-97
- Case Studies:
  - Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, Alessandro Cimatti: Formal Design and Safety Analysis of AIR6110 Wheel Brake System. CAV (1) 2015: 518-535
  - Cristian Mattarei, Alessandro Cimatti, Marco Gario, Alessandro Cimatti, Kristin Y. Rozier: Comparing Different Functional Allocations in Automated Air Traffic Control Design. FMCAD 2015: 112-119
- Tools used in the course:
  - Alessandro Cimatti, Michele Dorigatti, Alessandro Cimatti: OCRA: A tool for checking the refinement of temporal contracts. ASE 2013: 702-705
  - Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, Alessandro Cimatti: The nuXmv Symbolic Model Checker. CAV 2014: 334-342
  - Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, Gianni Zampedri: The xSAP Safety Analysis Platform. TACAS 2016: 533-539