Runtime verification

István Majzik

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

Main topics of the course

- Overview (1)
  - V&V techniques, Critical systems
- Static techniques (2)
  - Verifying specifications
  - Verifying source code
- Dynamic techniques: Testing (7)
  - Developer testing, Test design techniques
  - Testing process and levels, Test generation, Automation
- **System-level verification (3)**
  - Verifying architecture, Dependability analysis
  - **Runtime verification**

Learning outcomes

- Explain the role of runtime verification and the related main challenges (K2)
- Explain the monitoring technique that uses reference automata (K2)
- Explain the monitoring technique that uses temporal logic expressions (K2)
- Construct an observer automaton on the basis of a sequence chart specification (K3)
- Identify how context-dependent behavior can be monitored (K1)

Table of contents

- Goals and challenges
  - Use cases
- Runtime verification techniques
  - Verification based on reference automata
  - Verification based on temporal logic properties
  - Verification based on sequence diagrams
  - Verification based on scenario and context description
- Implementation experience

Goals and challenges

What is runtime verification?

- Definition:

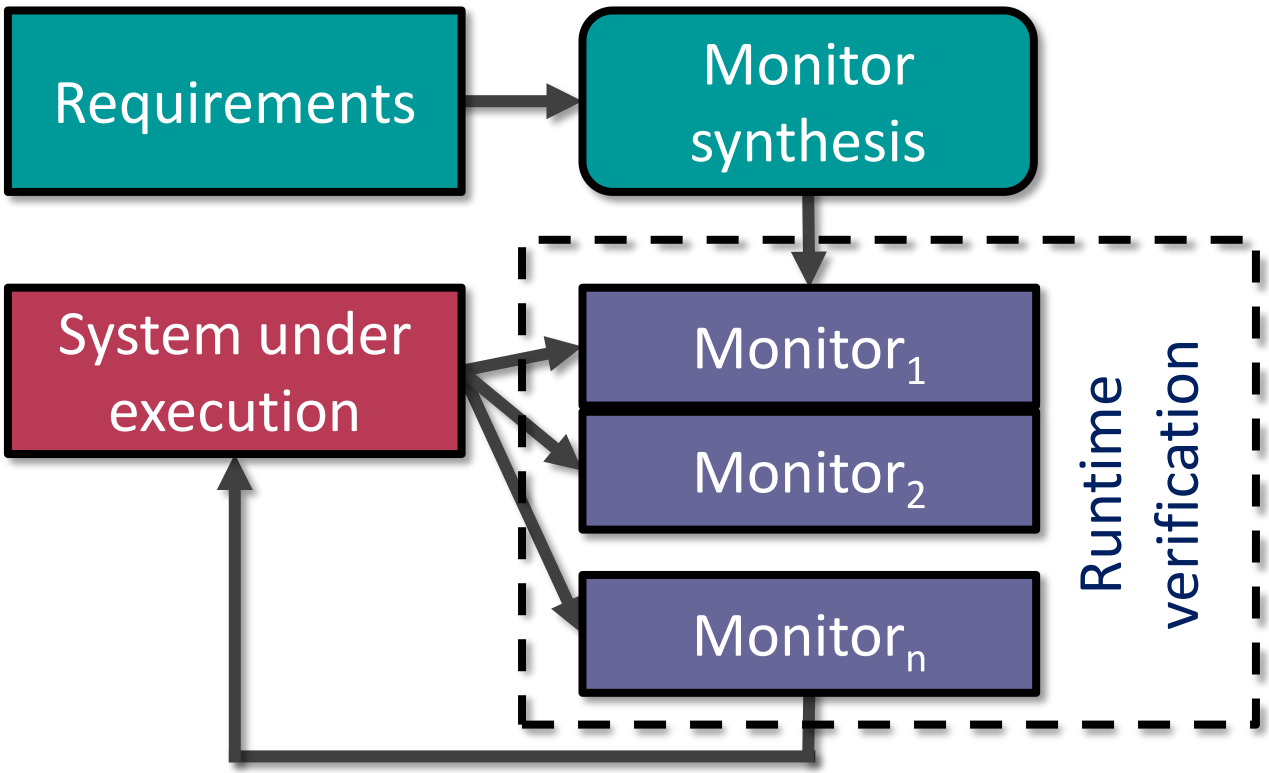
  Checking the behavior of systems
  - in runtime (on-line),
  - based on formally specified properties
- Motivation
  - Dependability and safety requirements
    - Safety-critical system: Safety (THR), fault tolerance
    - IT services: Service correctness (SLA), …
  - Runtime faults are inevitable
    - Random faults in hardware components
    - Software design, implementation, configuration faults

Goal: Runtime detection of faults

- Runtime fault detection is the basis of fault handling
  - Detection of hardware faults based on source code
    - E.g., checking the CFG (watchdog processors)
    - Only for operational faults, based on implementation
  - Checking on the basis of requirements
    - For systematic (design, coding, configuration) faults as well
  - Verification on the basis of formalized properties
    - Precise representation of requirements
    - Automated synthesis of checker (monitor) components
- Example: Reactive fault handling
  - Fault detection followed by reaction (e.g., recovery, reconfiguration, setting of safe state, …)

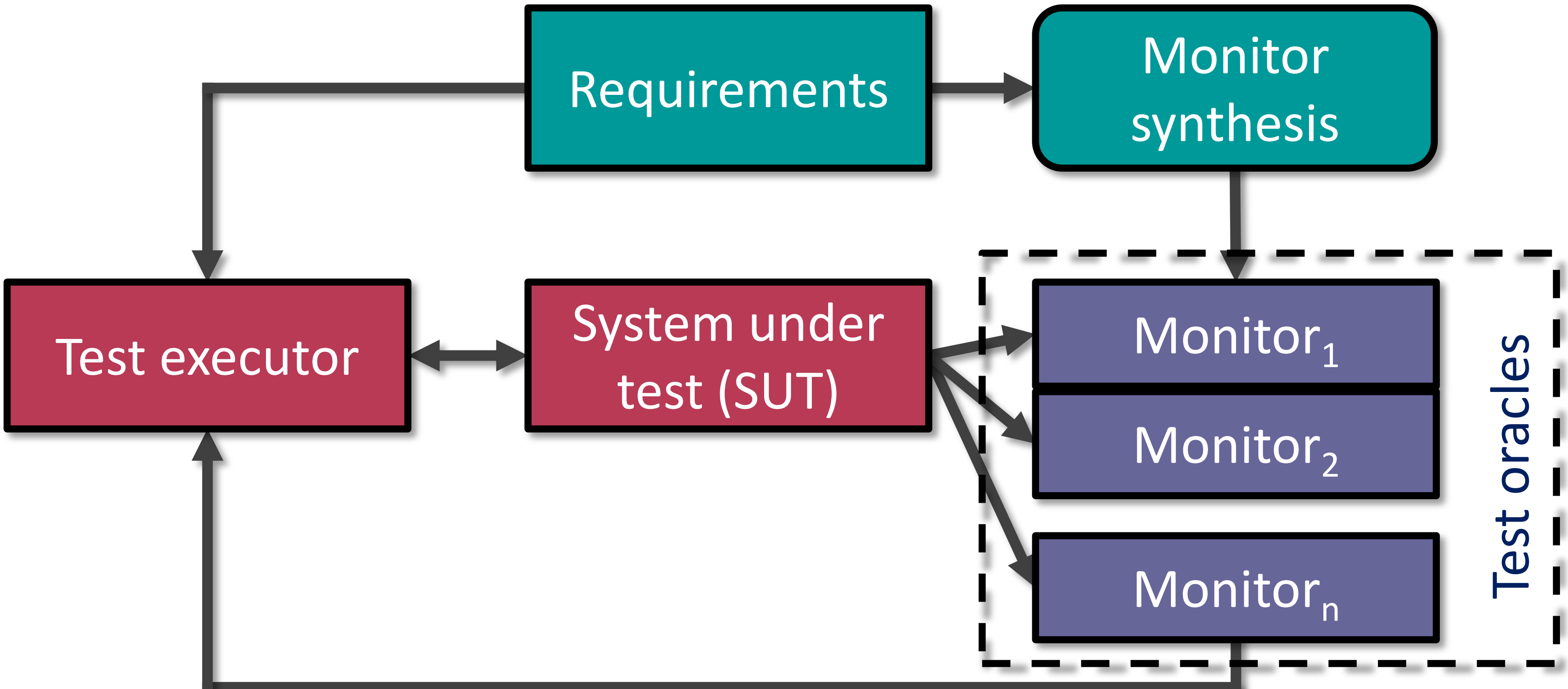
Use case 1: Runtime verification

- Monitors used for runtime verification
  - Evaluating formalized requirements
  - Detecting errors resulting from operational faults, configuration errors, unexpected environmental conditions



Use case 2: Evaluation of test output

- Monitors can be test oracles in testing frameworks
  - Evaluating the satisfaction of selected requirements
  - Detecting design or implementation errors



Challenges

- Verification techniques
  - Formalization of checked properties
  - Efficient algorithms for verification
- Instrumentation
  - Observation of the information needed for verification
  - Minimizing overhead
- Practical aspects of theoretical results
  - Monitor synthesis
  - Low resource needs, scalable implementation

    → Application in safety relevant embedded systems

Challenges

- Verification techniques

  - Execution trace based checking of temporal properties
    - Temporal logics
    - Reference automata
    - Regular expressions
  - Design-by-contract based monitoring
    - Executable assertions
  - Specification-less monitoring
    - Checking the generic correctness requirements of concurrent execution (e.g., deadlock, race, livelock, serialization conflicts)

Challenges

- Verification techniques
  - Formalization of checked properties
  - Efficient algorithms for verification
- Instrumentation
  - Observation of the information needed for verification
  - Minimizing overhead
- Practical aspects of theoretical results
  - Monitor synthesis
  - Low resource needs, scalable implementation

    → Application in safety relevant embedded systems

Challenges

- Verification techniques
  - Formalization of checked properties
  - Efficient algorithms for verification
- Instrumentation

- Active and passive instrumentation
  - Active: inserting source code snippets into observed code
  - Passive: observation without interference
- Techniques for active instrumentation
  - Aspect-Oriented Programming (AOP)
  - Tracematch: AspectJ extension for trace patterns
- Synchronous and asynchronous monitoring

Challenges

- Verification techniques
  - Formalization of checked properties
  - Efficient algorithms for verification
- Instrumentation
  - Observation of the information needed for verification
  - Minimizing overhead
- Practical aspects of theoretical results
  - Monitor synthesis
  - Reducing resource needs, scalable implementation

  → Application in critical embedded systems

Example: Framework for monitor synthesis

- MOP: Monitoring-Oriented Programming

Logic Plugins

Languages

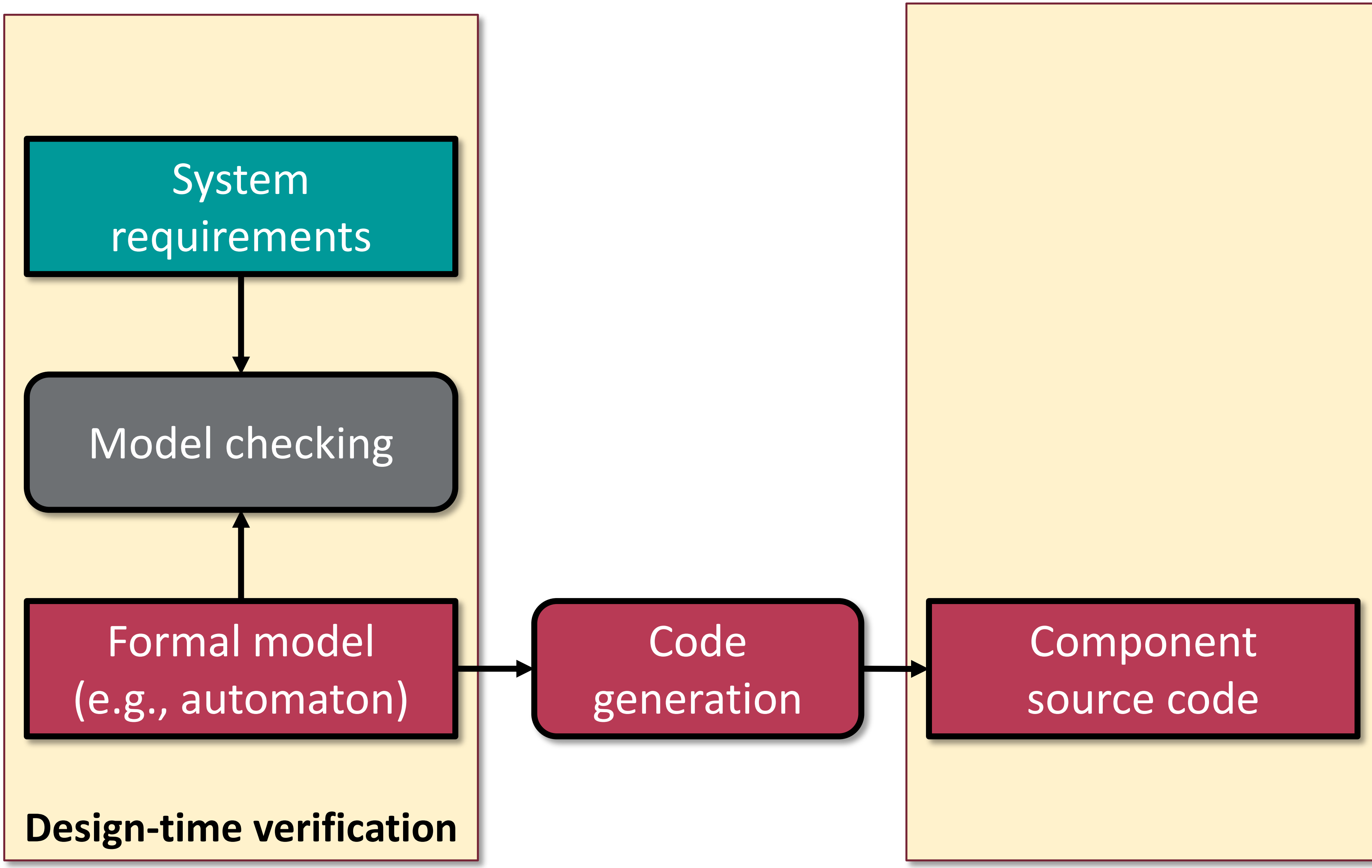
MOP	FSM	ERE	CFG	PTLTL	LTL	PTCaRet	SRS	...
JavaMOP	JavaFSM	JavaERE	JavaCFG	JavaPTLTL	JavaLTL	JavaPTCaRet	JavaSRS	...
BusMOP	BusFSM	BusERE	...	BusPTLTL	...	...	...	...
ROSMOP	ROSFSM	...	ROSCFG	...	...	...	...	...
...	...	...	...	...	...	...	...	...

  - FSM: Finite State Machines
  - ERE: Extended Regular Expressions
  - CFG: Context Free Grammars
  - PTLTL: Past Time Linear Temporal Logic
  - LTL: Linear Temporal Logic
  - PTCaRet: Past Time LTL with Calls and Returns
  - SRS: String Rewriting Systems

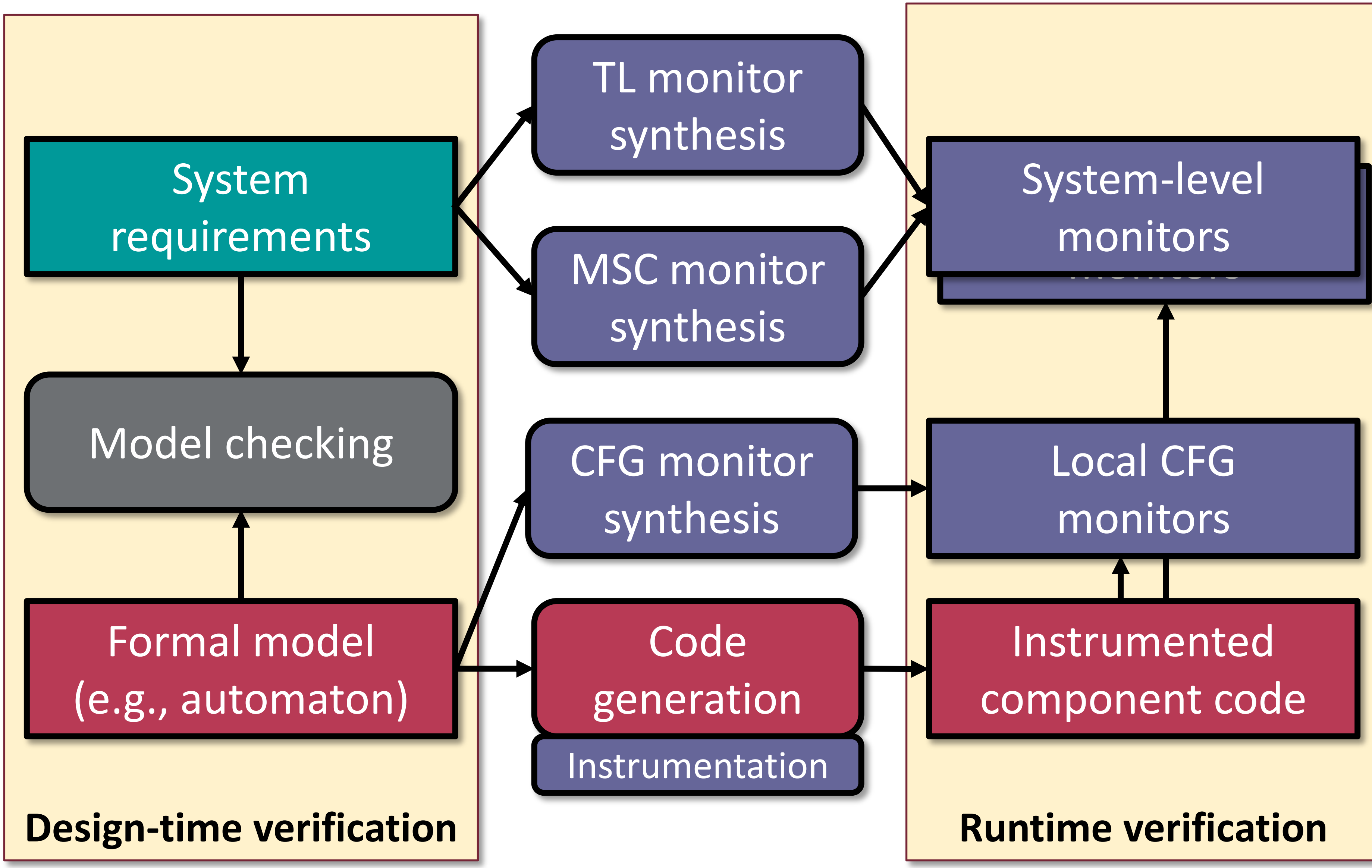
The discussed solutions

- To be used in: Control-oriented applications
  - State based, event- and message-driven behavior
  - E.g., safety functions, protocols, …
- Hierarchical (scalable) runtime verification
  - Local: Behavior of single components (controller, ECU)
    - Reference automaton: control and simple data faults
    - Local temporal properties of states
  - System-level: Interaction of components
    - Temporal properties of interactions
    - Scenario (MSC) based properties
- Relation to model based design
  - Model based code generation with instrumentation

Overview: Design-time verification

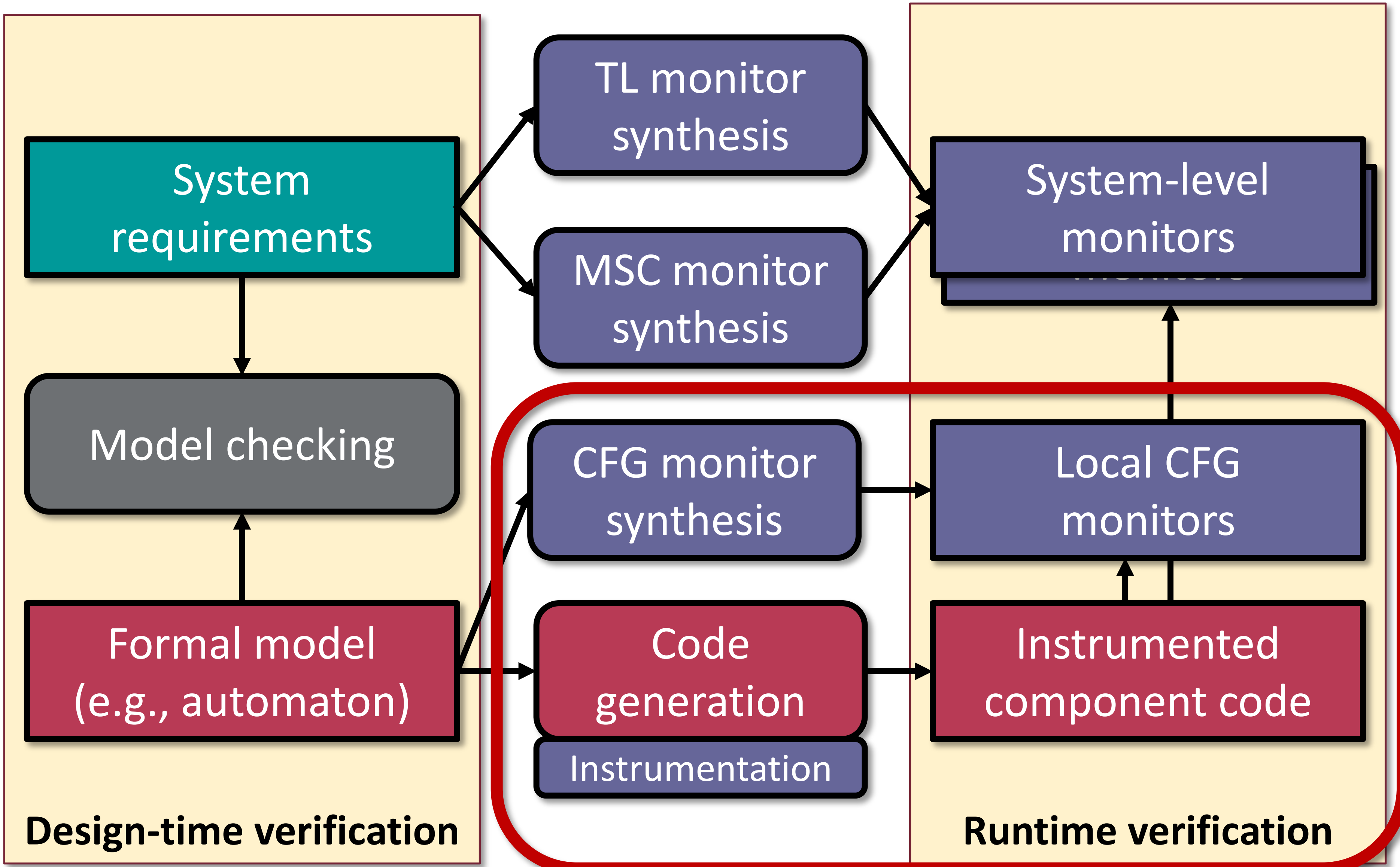


Overview: Runtime verification

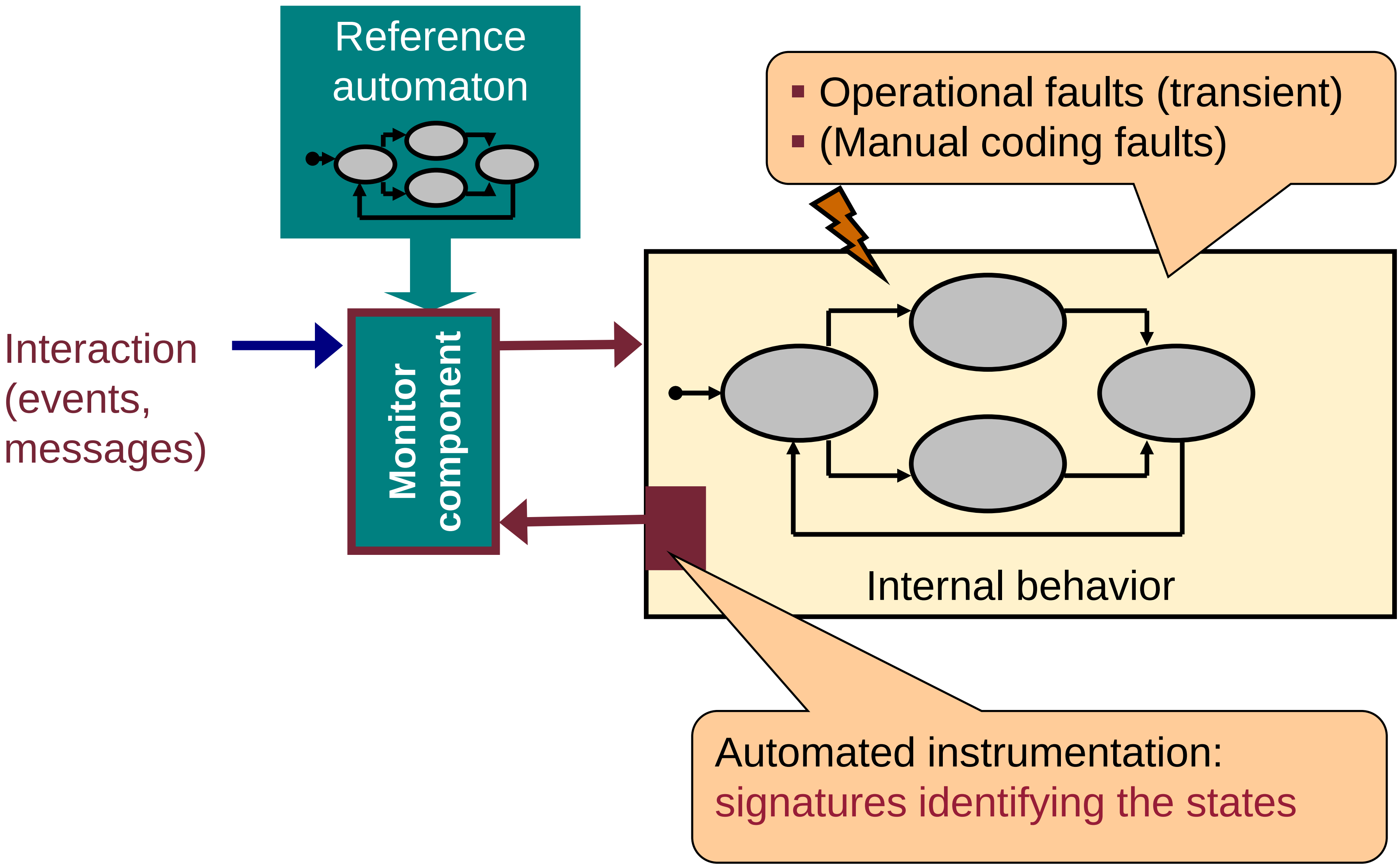


Runtime verification based on reference automata

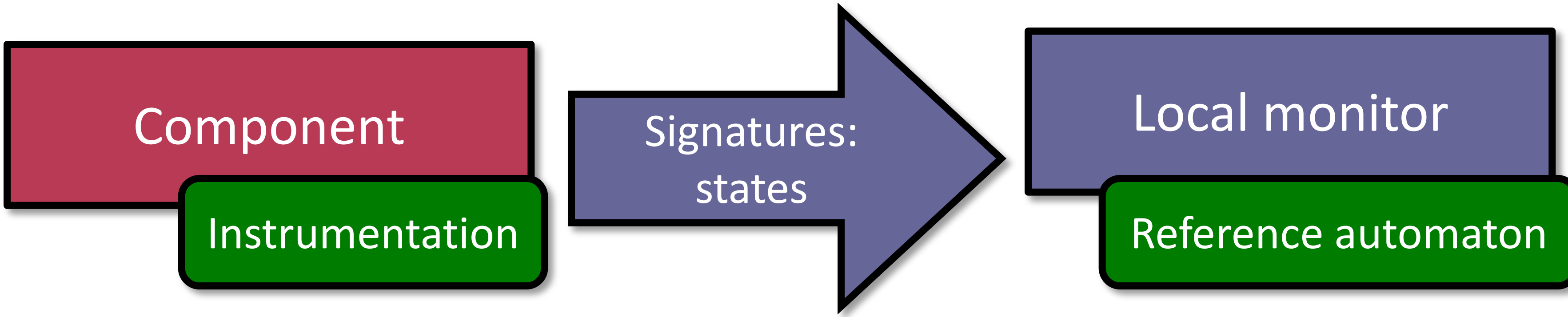
Overview: Runtime verification

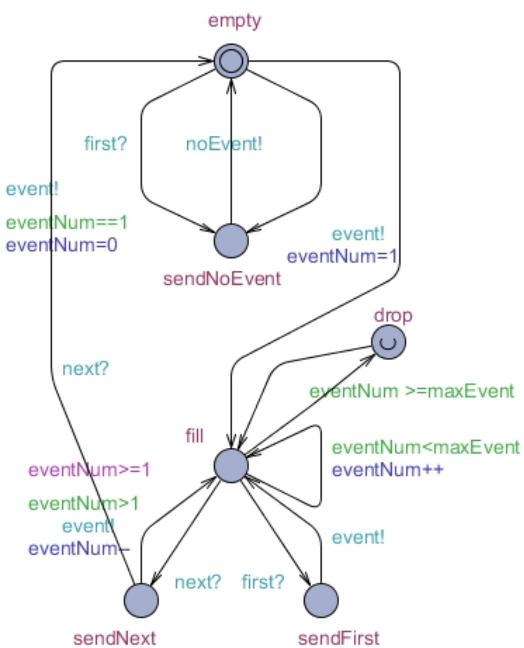


Monitoring on the basis of reference automaton



State-based monitoring of timed automata

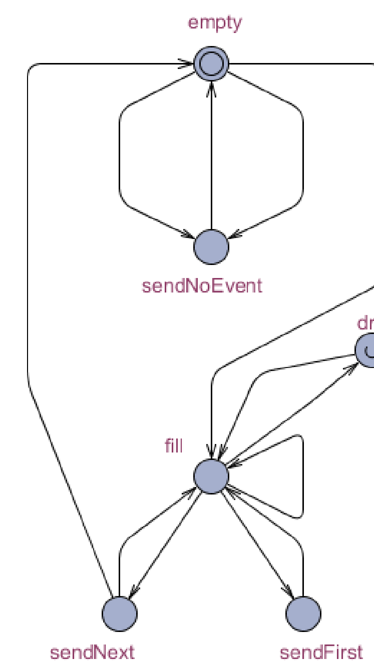




Basis for code generation

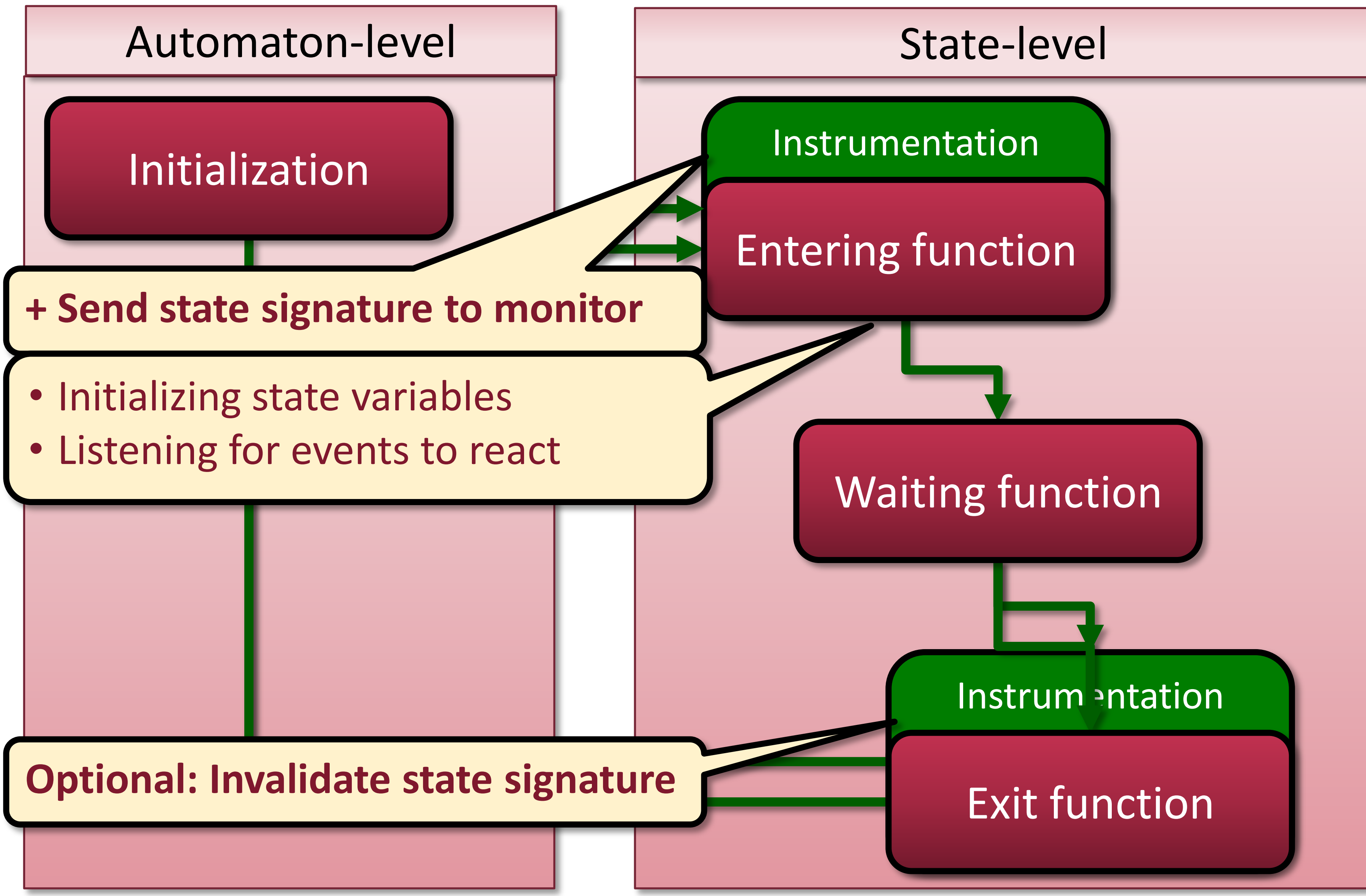
Detectable faults:

- Wrong state / state transition sequence
- Stuck in state (timeout)
- Violation of timing conditions (in case of timed automata reference)



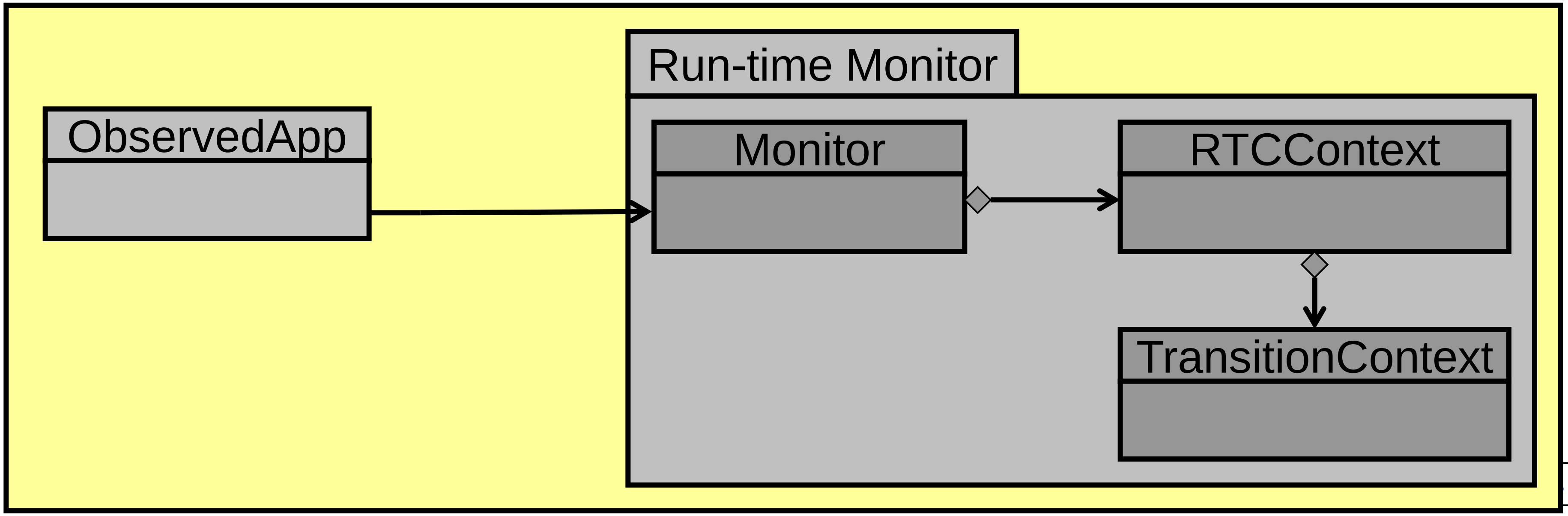
Reference automaton

Instrumentation in the generated source code



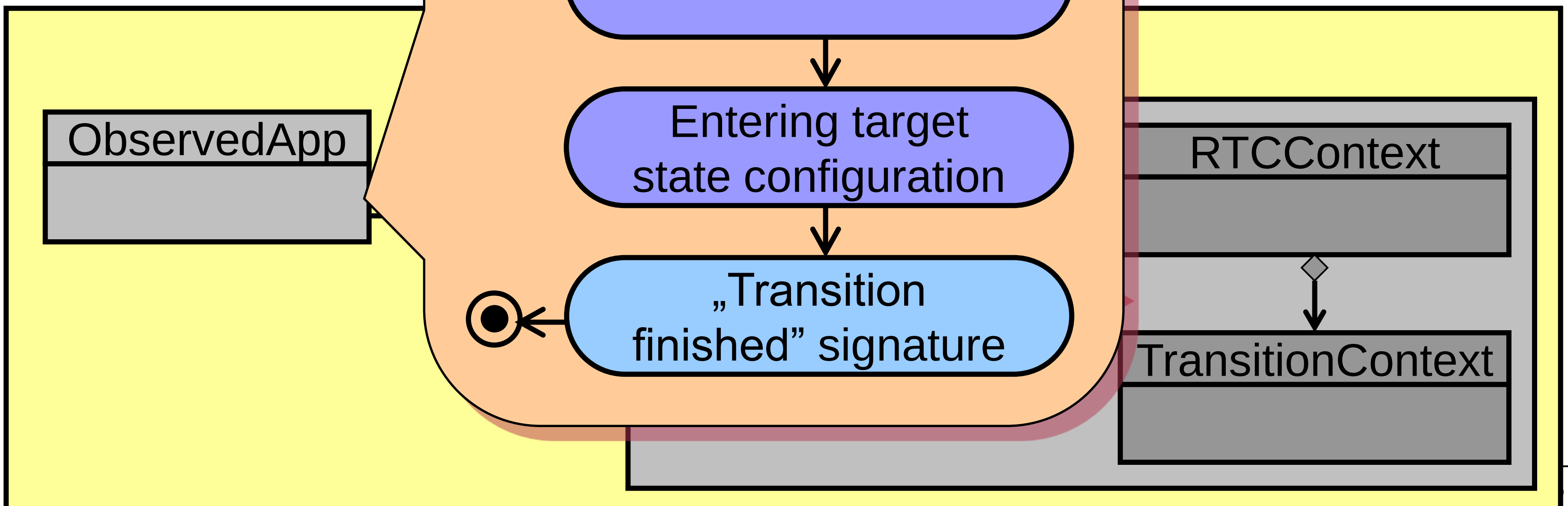
Case study: Monitoring on the basis of statecharts

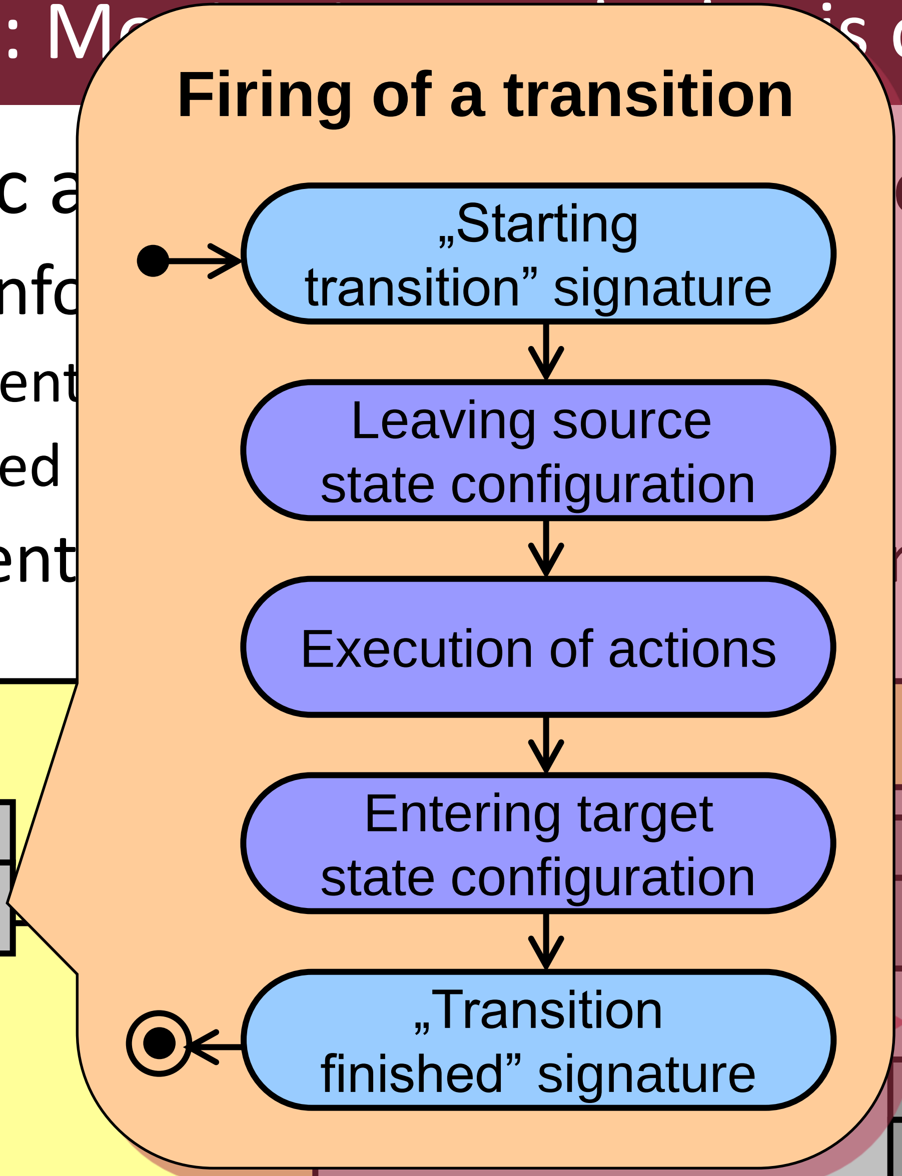
- Systematic and transparent instrumentation:
  - Explicit information for the monitor
    - States entered and left
    - Executed actions
  - Instrumentation: Aspect-oriented programming



Case study: M… …is of statecharts

- Systematic a… …entation:
  - Explicit info…
    - States ent…
    - Executed …
  - Instrument… …rogramming

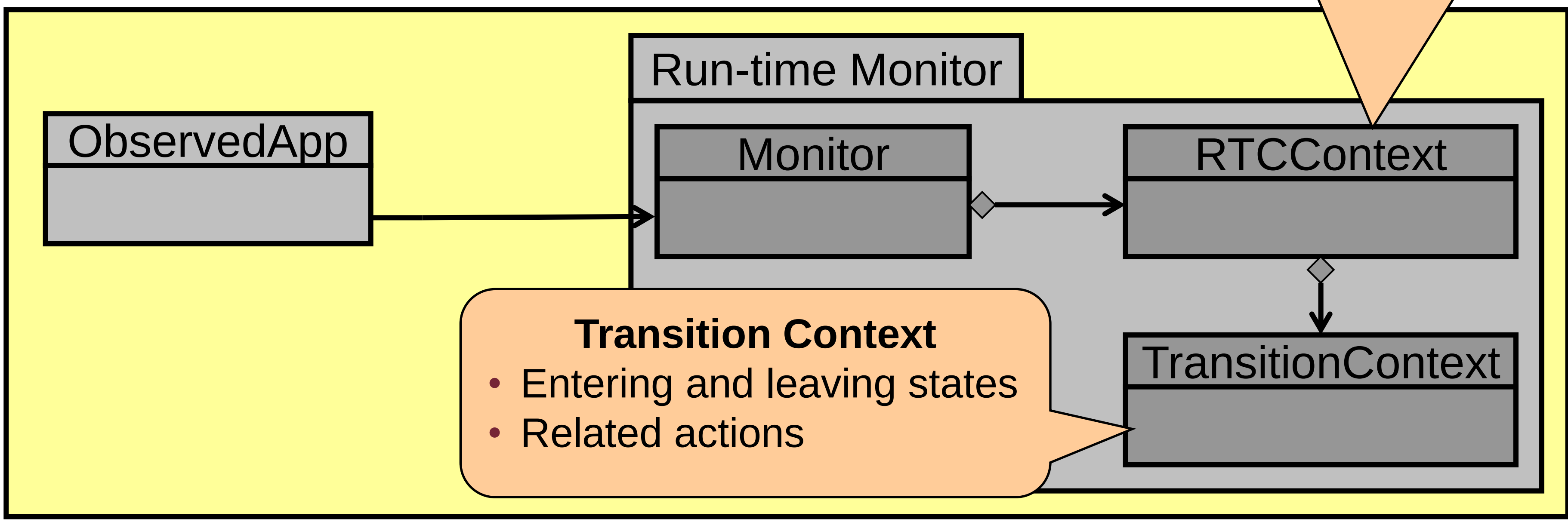




Case study: Monitoring on the basis of statecharts

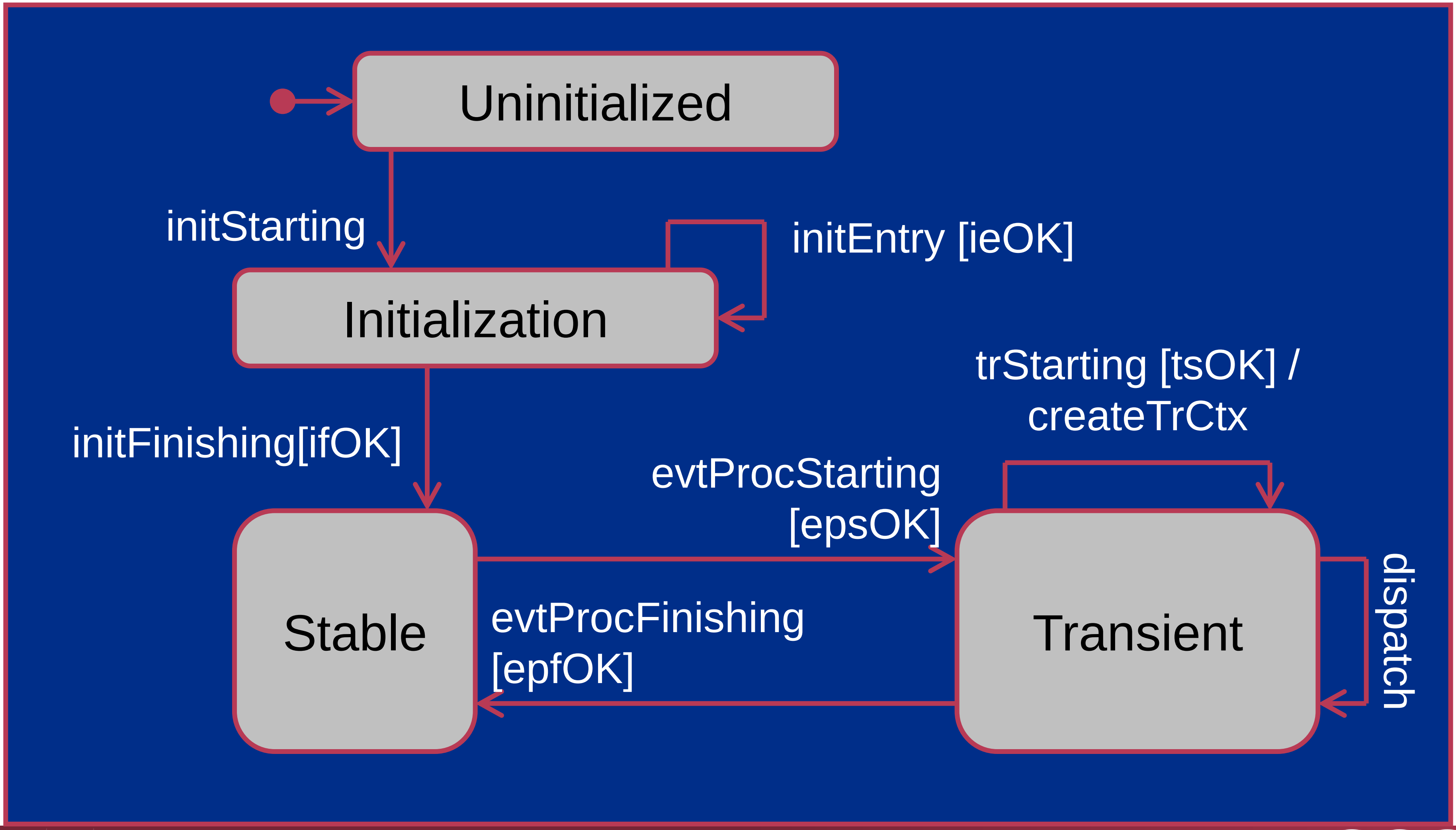
- Systematic and transparent instrumentation:
  - Explicit information for the mo
    - States entered and left
    - Executed actions
  - Instrumentation: Aspektus-orie

RTC context
- Initialization
- Starting and finishing event processing
- Signals for Transition Context



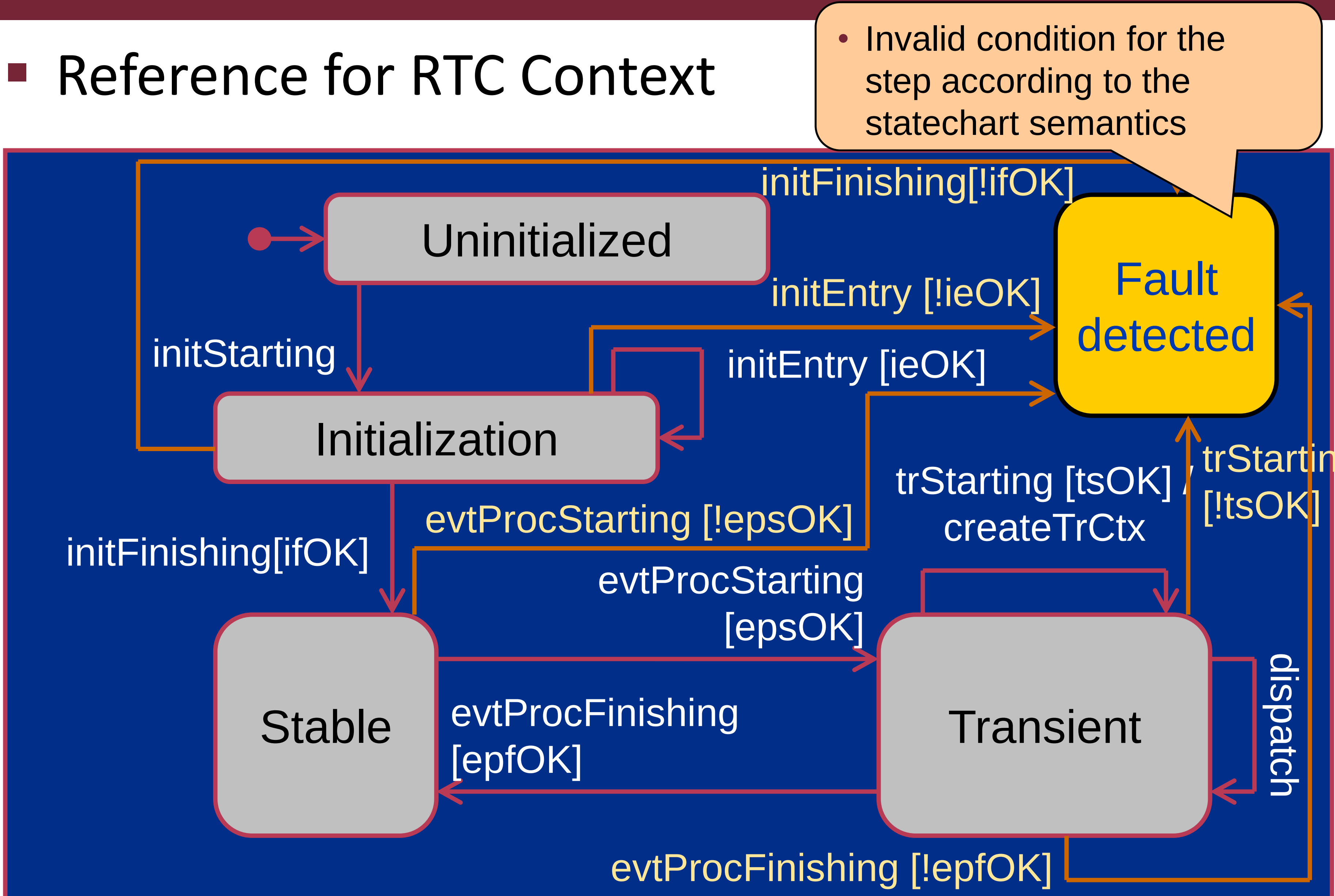
Case study: Monitoring on the basis of statecharts

- Reference for RTC Context



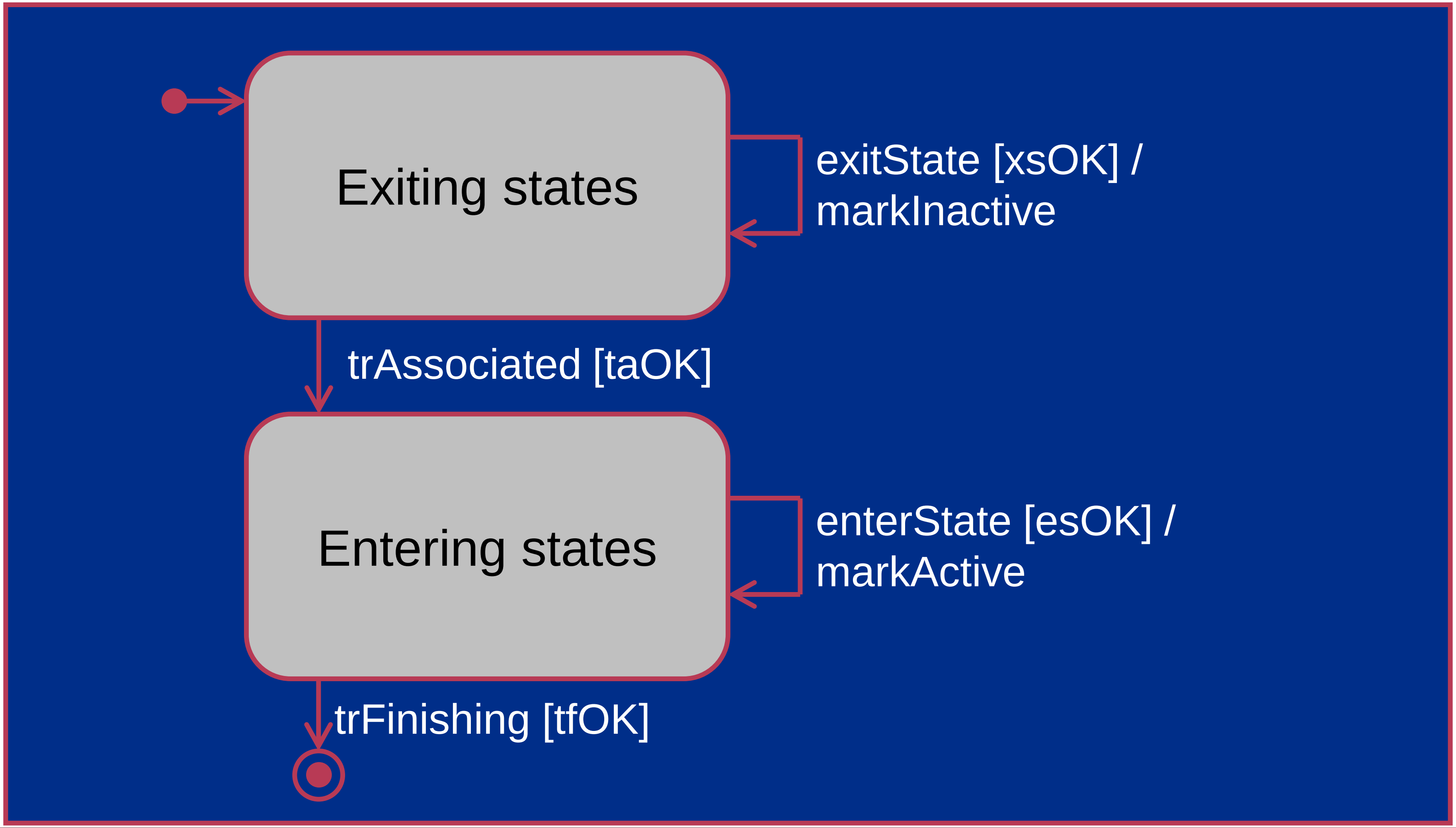
Case study: Monitoring on the basis of statecharts

- Reference for RTC Context



Case study: Monitoring on the basis of statecharts

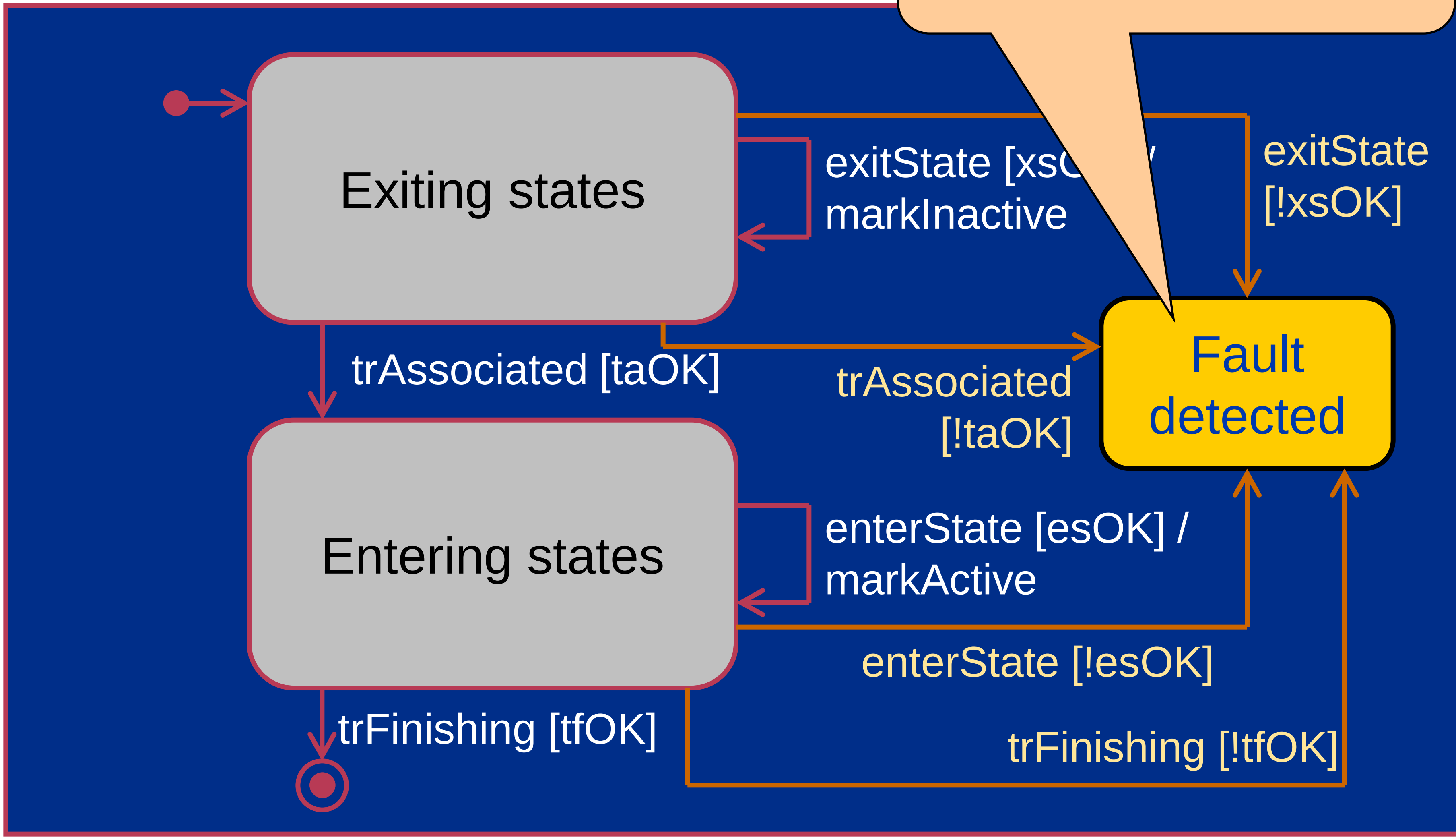
- Reference for Transition Context



Case study: Monitoring on the basis of statecharts

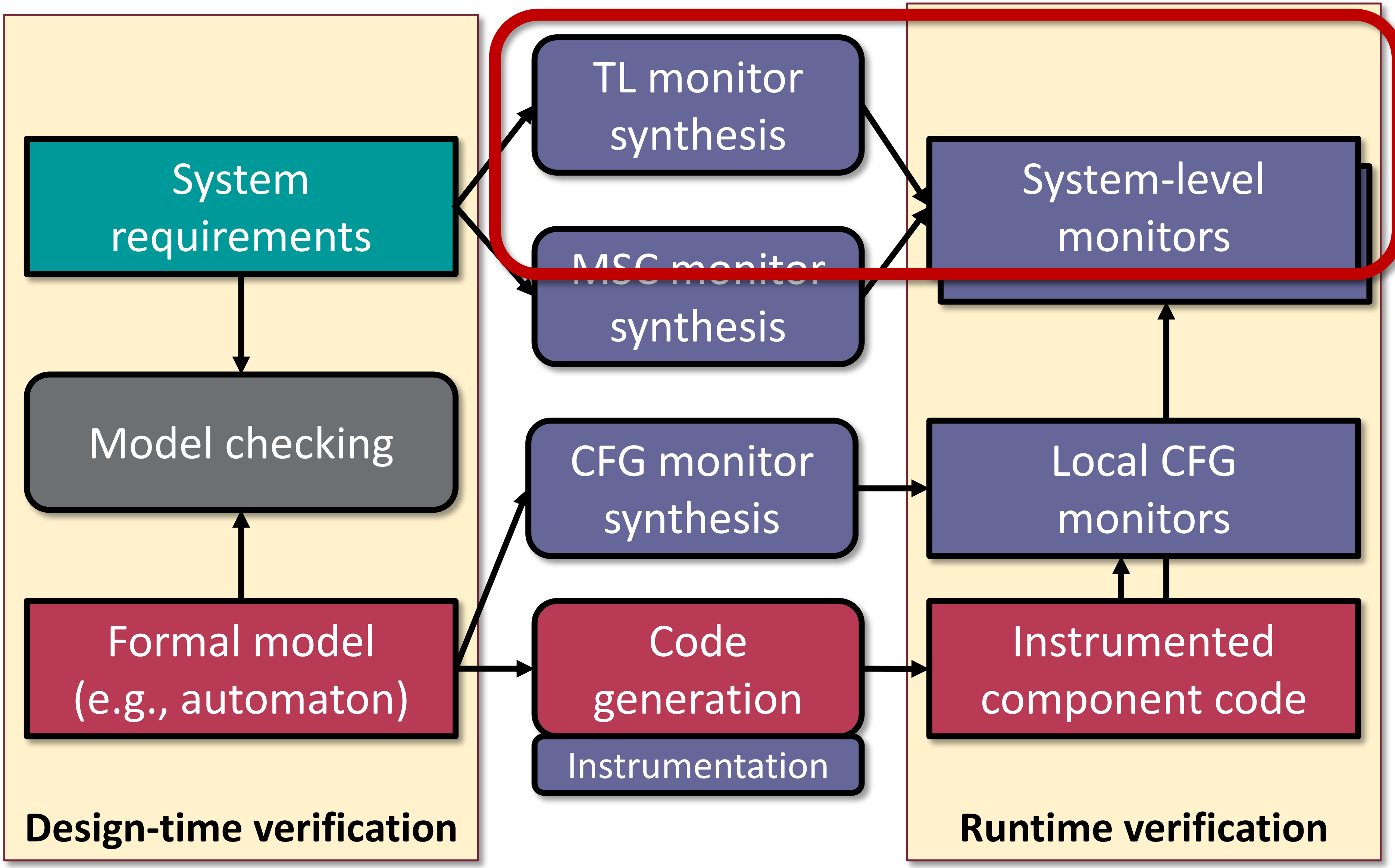
- Reference for Transition Conte

- Invalid condition for the step according to the statechart semantics



Runtime verification based on temporal logic properties

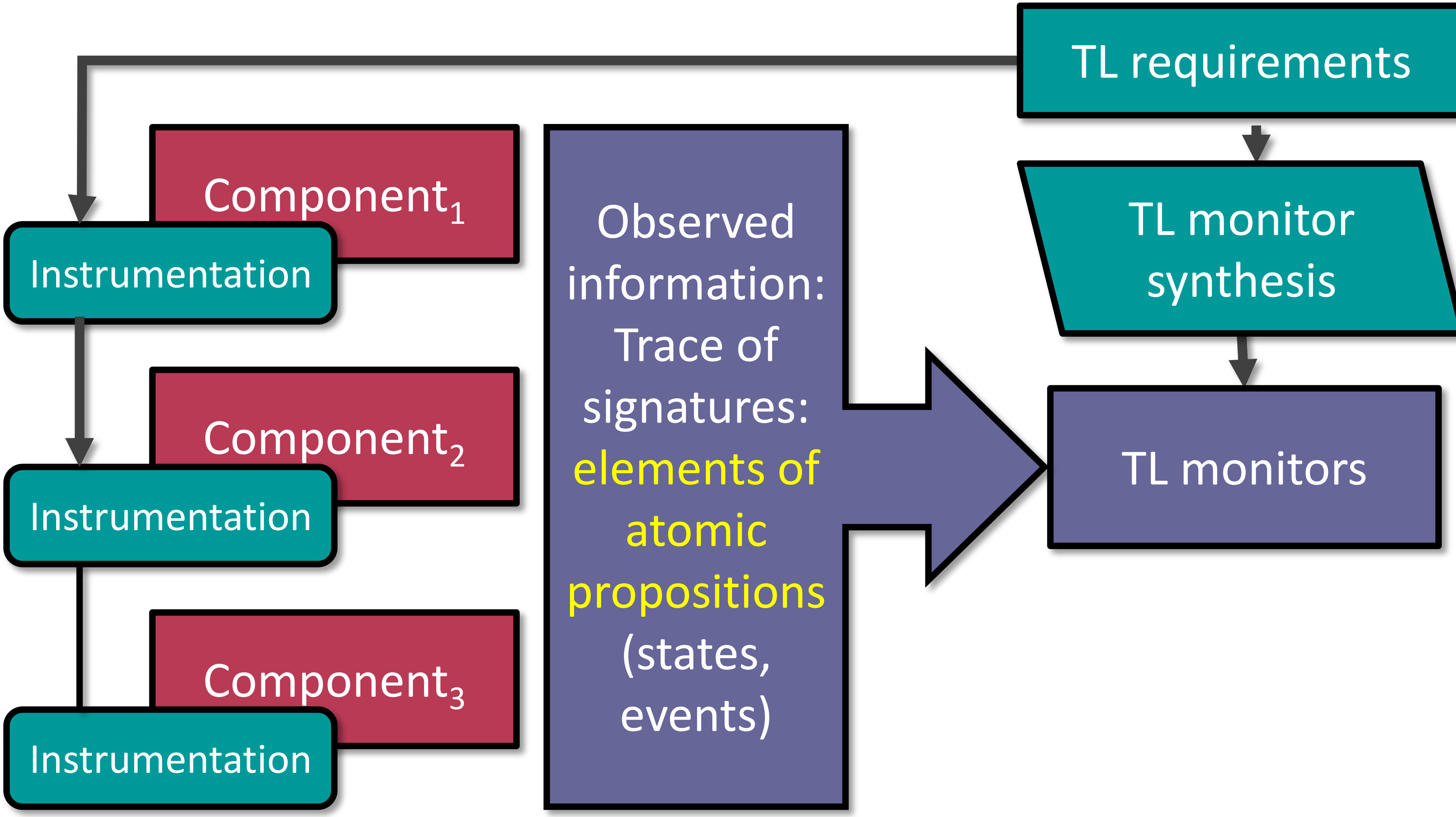
Overview: Runtime verification



Temporal logic based properties

- Properties: Ordering and reachability of states (events)
  - States: Characterized by atomic propositions
  - Safety properties: Invariants for all states
  - Liveness properties: Reachability of favorable states
- Formalization: Temporal logics (TL)
  - Linear Time TL: LTL; for a single path of execution (trace)
    - Temporal operators: X (next), U (until), G (globally), F (future)
    - Use case: Checking observed trace in runtime
  - Branching time TL: CTL; for all execution paths
    - LTL operators and path quantifiers: E (exists), A (forall)
    - Use case: Checking all paths (design-time or during testing)
- Runtime checking TL properties
  - Not on a model, but on an observed trace

Setup of TL based monitoring



Monitoring LTL expressions

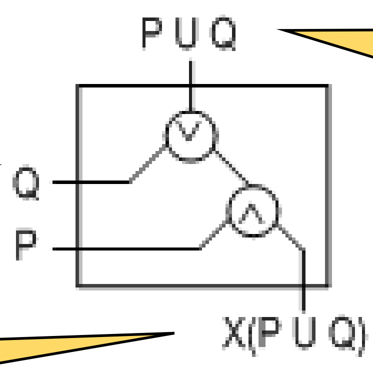
- Preprocessing: Normal form of expressions
  - Only $\wedge$, $\neg$, X and U operators can be included
  - All expressions can be mapped to this normal form
    - Using de Morgan's laws for Boolean expressions
    - Mapping LTL operators: $F\ p = true\ U\ p$, $G\ P = \neg\ (F\ \neg p)$
- Separating two parts of the expressions:
  - Present-time part: Boolean expressions of atomic propositions
  - Next-time part: Expression after an X operator
  - Basic rule: $P\ U\ Q = \underbrace{Q \vee (P \wedge}_{\text{present}} X \underbrace{(P\ U\ Q)}_{\text{next-time}})$

Role of the separated expressions

- Present-time part
  - Can be checked in the actual state (step of the observed trace)
  - I.e., receiving a set of atomic propositions about current state and events of the observed system
- Next-time part
  - Can be checked from the next state (suffix of the trace)
  - I.e., receiving the future part of the observed trace of atomic propositions
- Example: $P \mathbin{U} Q = Q \lor (P \land X\,(P \mathbin{U} Q))$

Atomic propositions in the current state (represented as Boolean inputs)

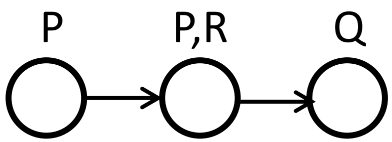
Next-time expression to be evaluated later



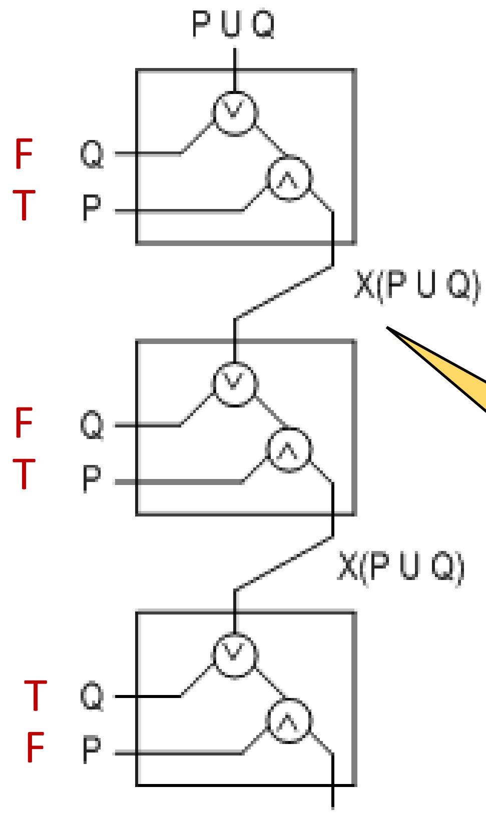
Output:
- True, if Q is true
- False, if Q is false and P is false
- Otherwise depends on next state

Evaluating a trace of atomic propositions

- Checking of P U Q = Q ∨ (P ∧ X (P U Q))
- Checking a trace:



Sequence of inputs (atomic propositions) according to the trace



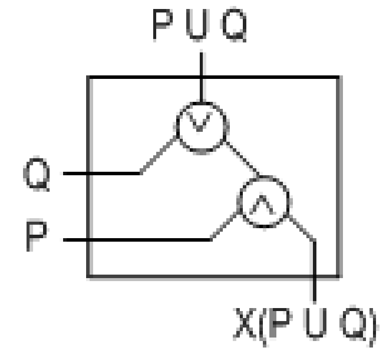
Blocks represent the iterative evaluation of next-time expressions

Construction of the observer automaton

- Basic idea of monitoring (summary):
  - Constructing an observer:
    Receives atomic propositions in each step of the trace
  - Evaluates present-time part in its actual state:
    Error detected if it is false independent of next-time part
  - Delegates next-time part of the expression to its next state:
    Error to be detected from the next state
- Iterative construction of the observer:
  - Separate present-time and next-time expressions
  - Assign monitor state (data structure) for the expressions
    - Evaluation of present-time expression
    - If the same expressions occur repeatedly:
      no new monitoring state shall be assigned
  - Continue with the next-time expression for the next state

Operations with ternary logic

- Evaluation of expressions
  - The result of evaluation of the next-time expression is "unknown"
  - The "unknown" is always resolved at the end of the trace



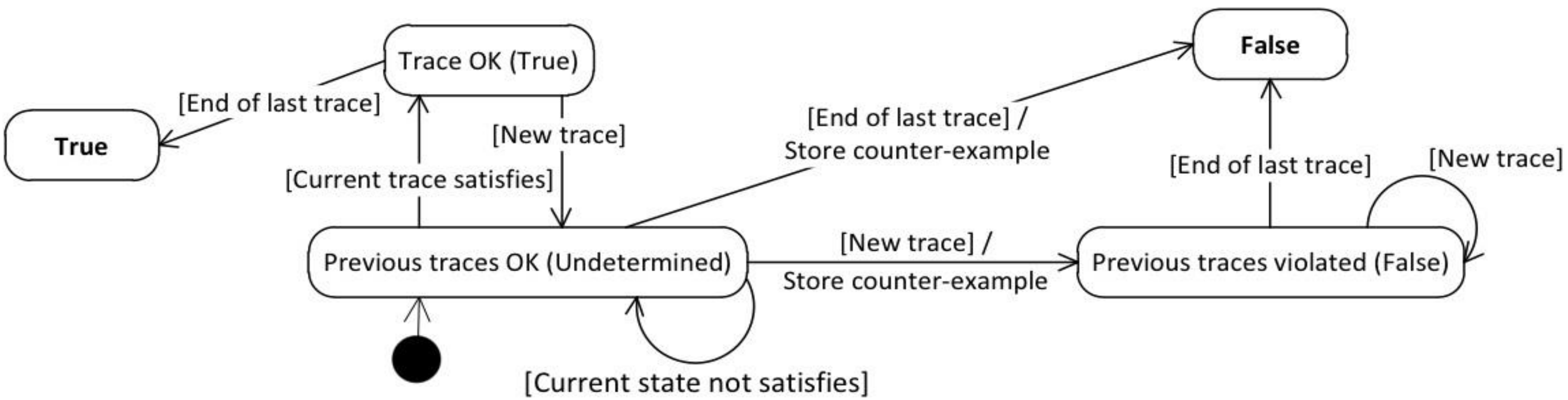
- Operations with ternary logic:

a	b	$a \wedge_3 b$
$\top$	$\top$	$\top$
$\bot$	$\{\top, \bot, ?\}$	$\bot$
$\{\top, \bot, ?\}$	$\bot$	$\bot$
$?$	$\{\top, ?\}$	$?$
$\{\top, ?\}$	$?$	$?$

a	$\neg_3 a$
$\top$	$\bot$
$\bot$	$\top$
$?$	$?$

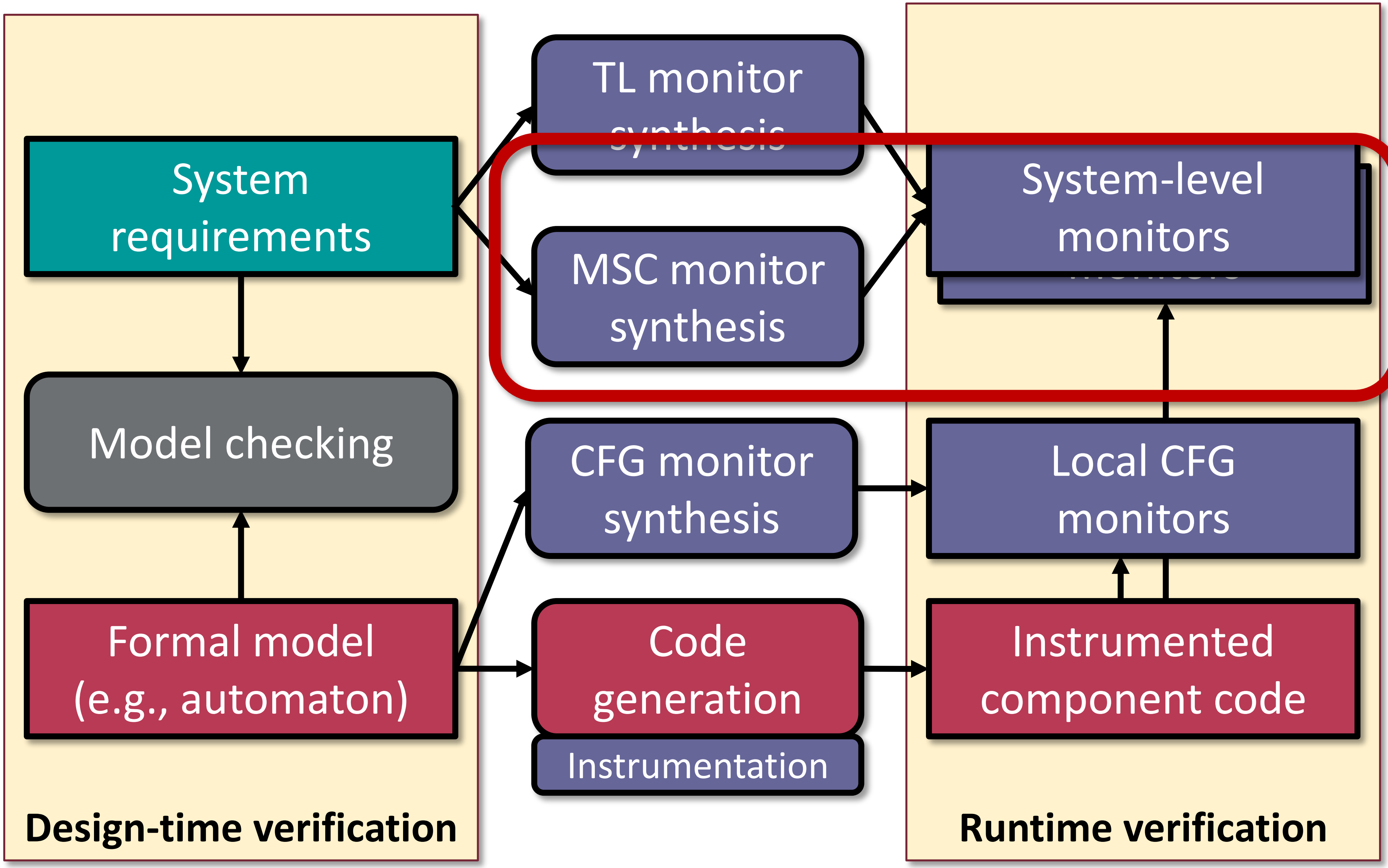
Extension: CTL based monitoring

- Suited to checking sets of execution traces
  - Quantification: “For all traces …”, “There shall exist a trace that …”
- E.g., monitors as test oracles check all traces of a test suite
  - Specific events are added: <New trace>, <End of last trace>
- Monitor synthesis
  - Checking a single trace: Similar to LTL checking
  - Checking a set of traces (test suite): Observer constructed
- Example: Observer for checking AF (for all traces eventually …)



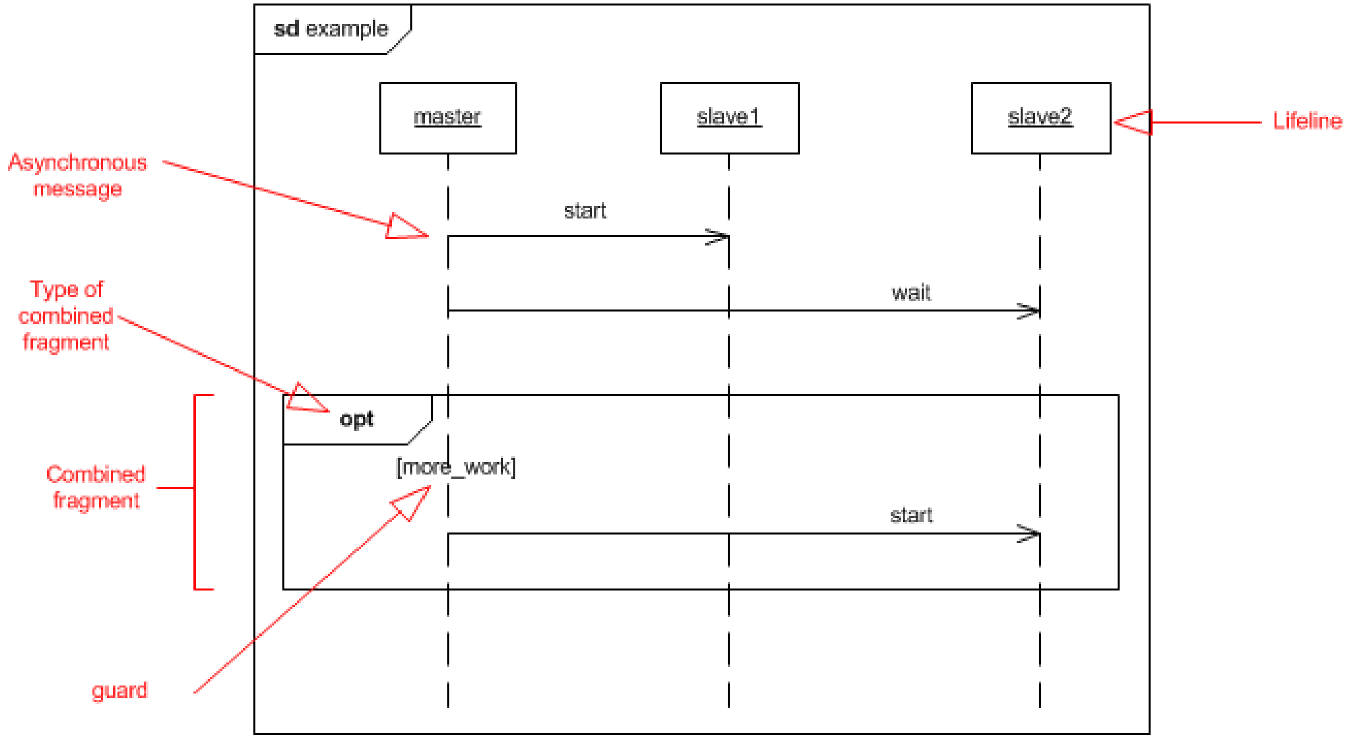
Runtime verification based on sequence diagrams

Overview: Runtime verification

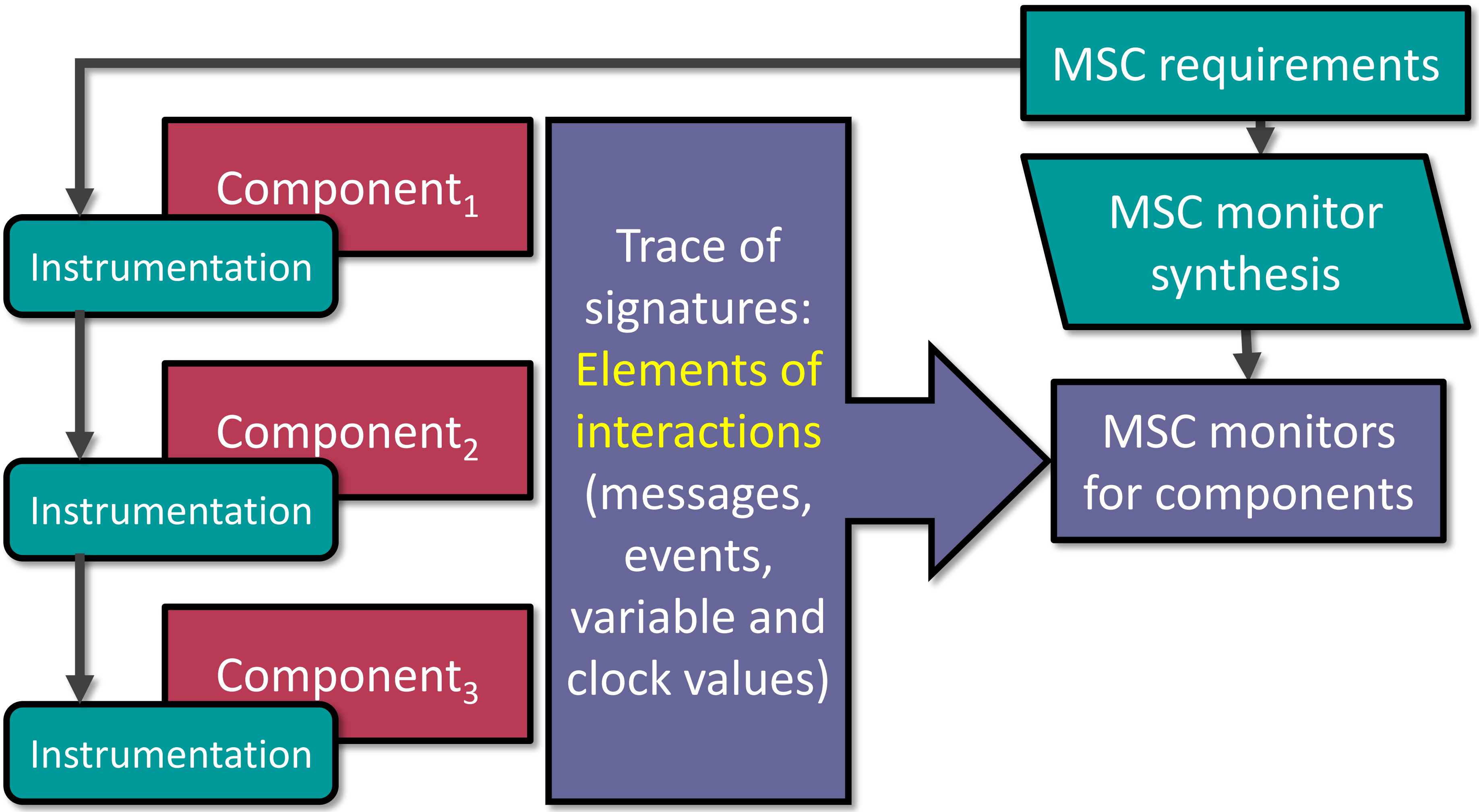


MSC based properties

- Goal: Checking interactions based on intuitive description
  - Synchronization, message passing, local conditions
- Formalism: Message Sequence Charts variant
  - Lifelines, messages, guard conditions, combined fragments

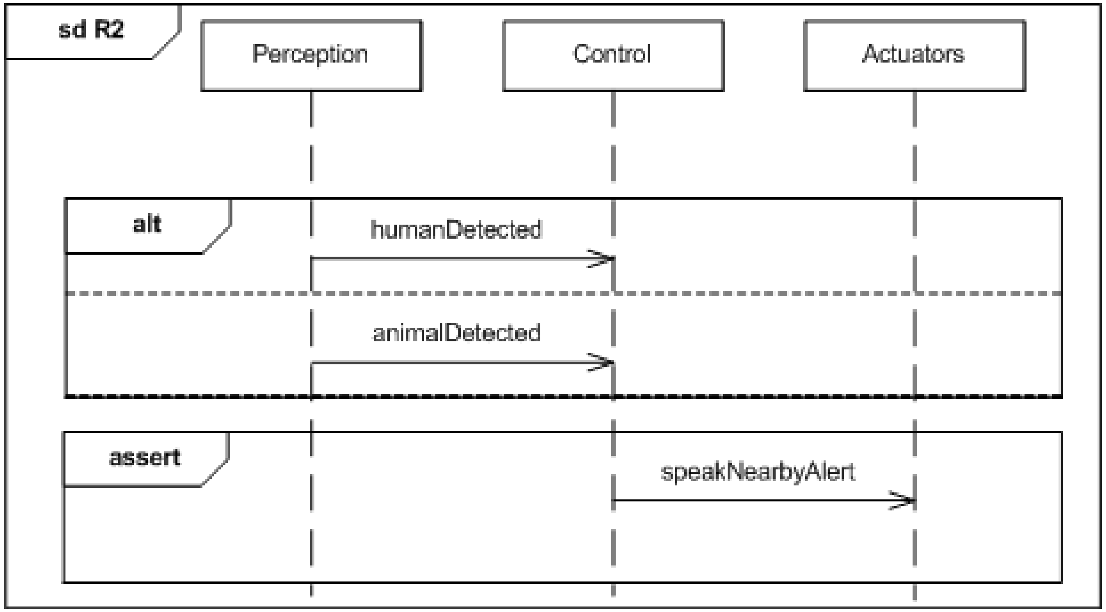


Setup of MSC based monitoring



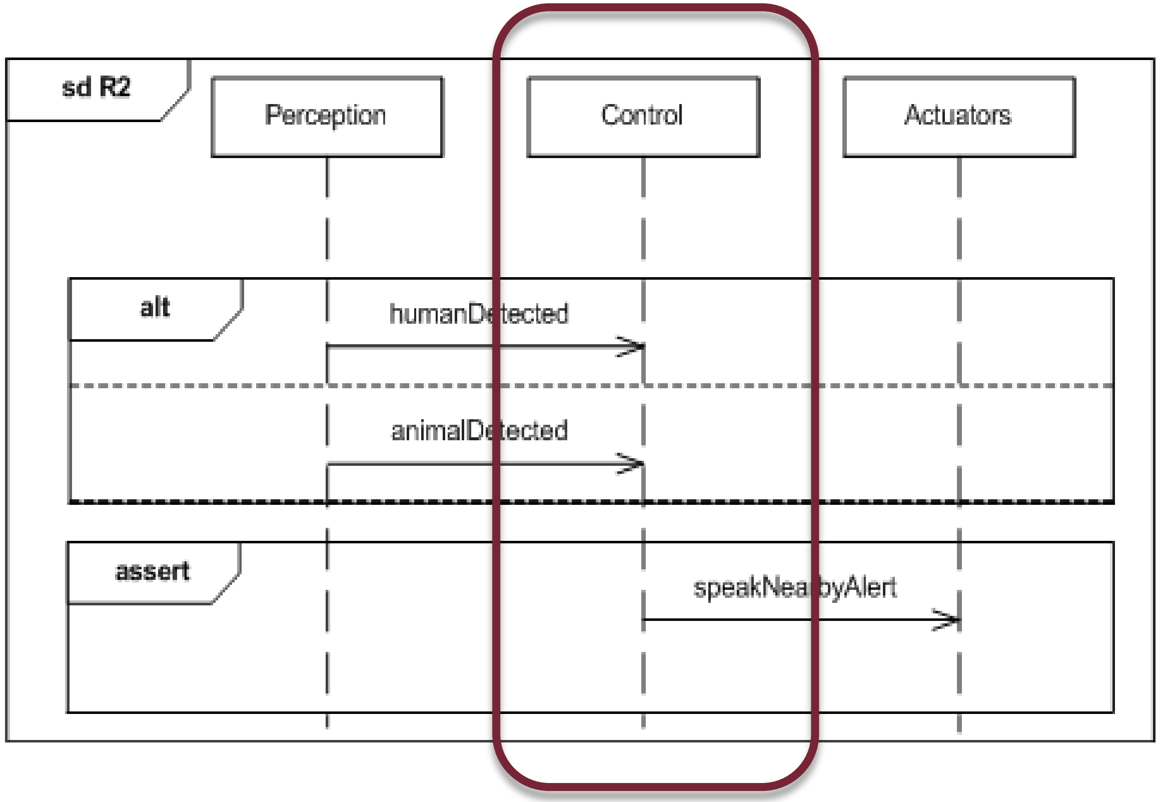
Restrictions and extensions

- Combined fragments relevant to monitoring:
  - Alternative (alt), optional (opt), parallel (par)
- Parts of the chart:
  - Condition part (pre-chart): behavior to be matched to check the property (otherwise not relevant)
  - Assert part (main chart): behavior to be matched to satisfy a property (otherwise violated)



Monitoring on the basis of an MSC

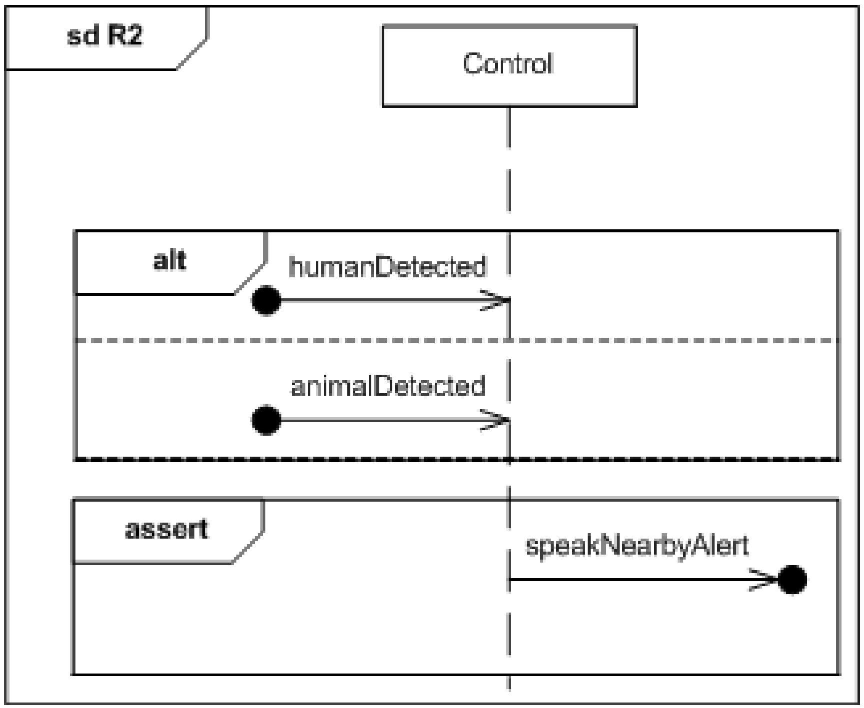
- Monitor constructed here: Observing a single lifeline (single component)

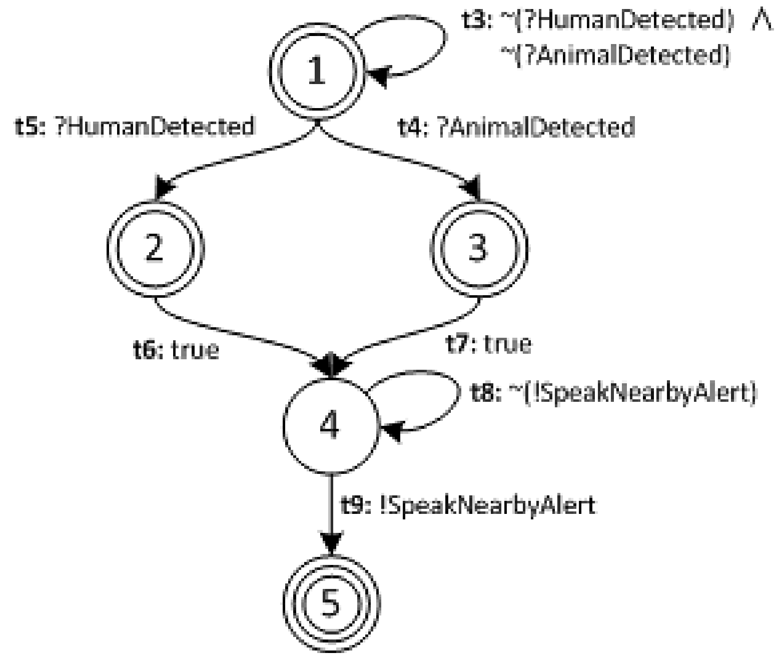


Monitoring on the basis of an MSC

- Observer automata constructed on the basis of the MSC lifeline

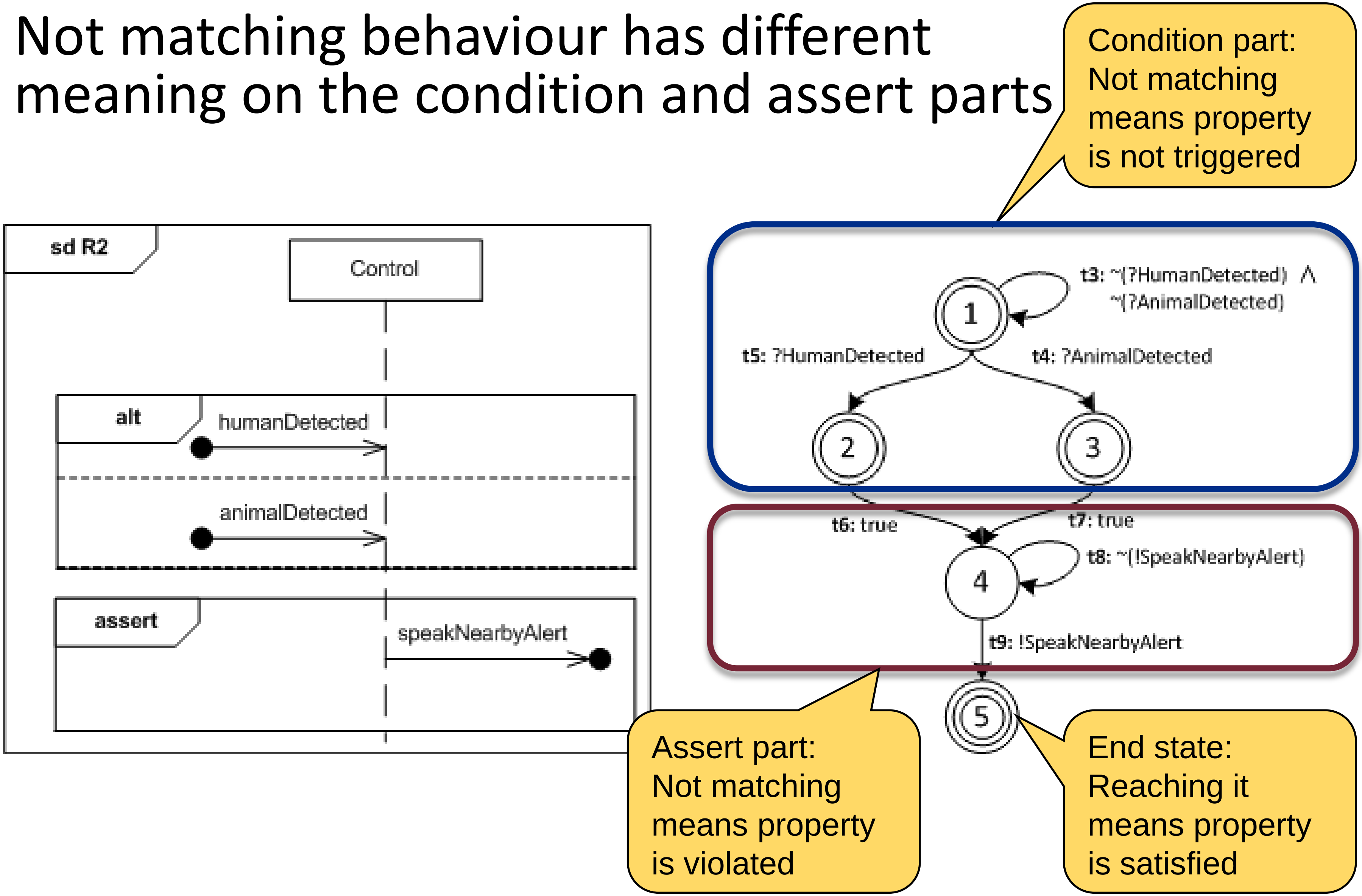
- Input events and messages, e.g., ?humanDetected
- Output actions and messages, e.g., !speakNearbyAlert





Role of condition and assert part

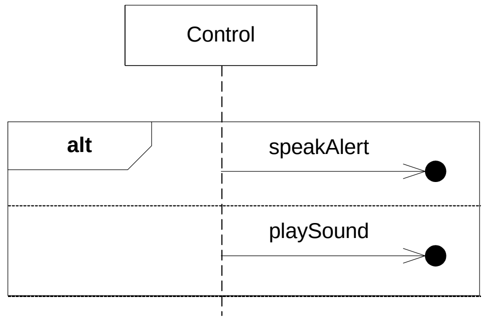
- Not matching behaviour has different meaning on the condition and assert parts

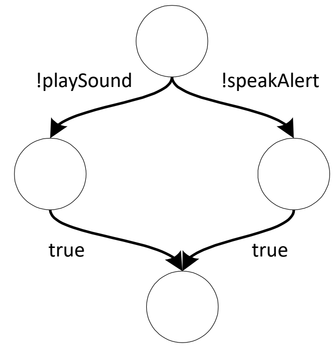


Basic patterns to construct the monitor

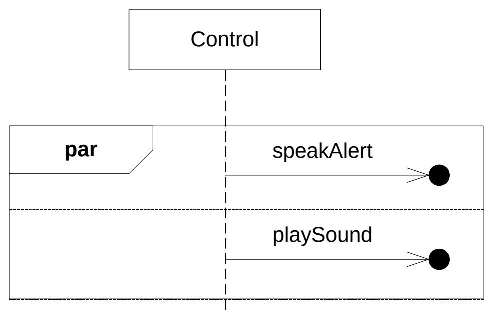
(Negative edges are omitted)

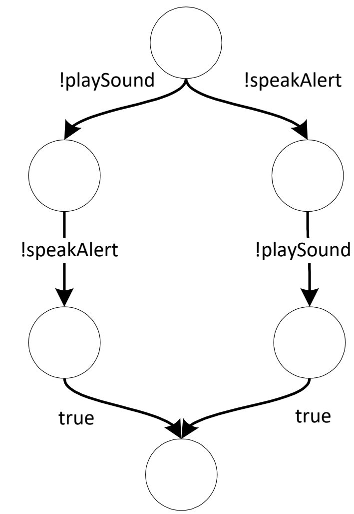
- Alternative:



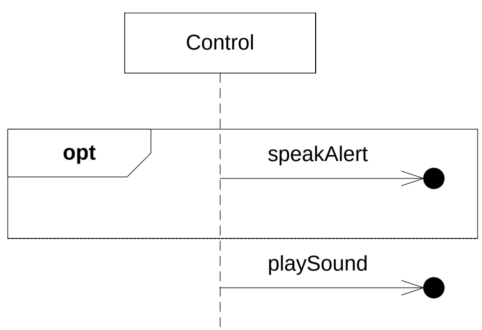


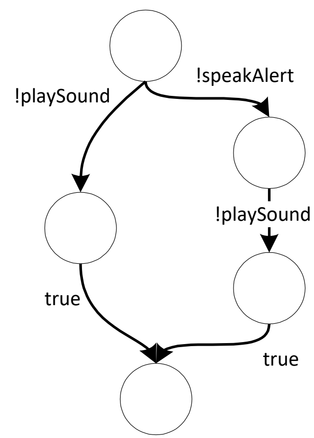
- Parallel:



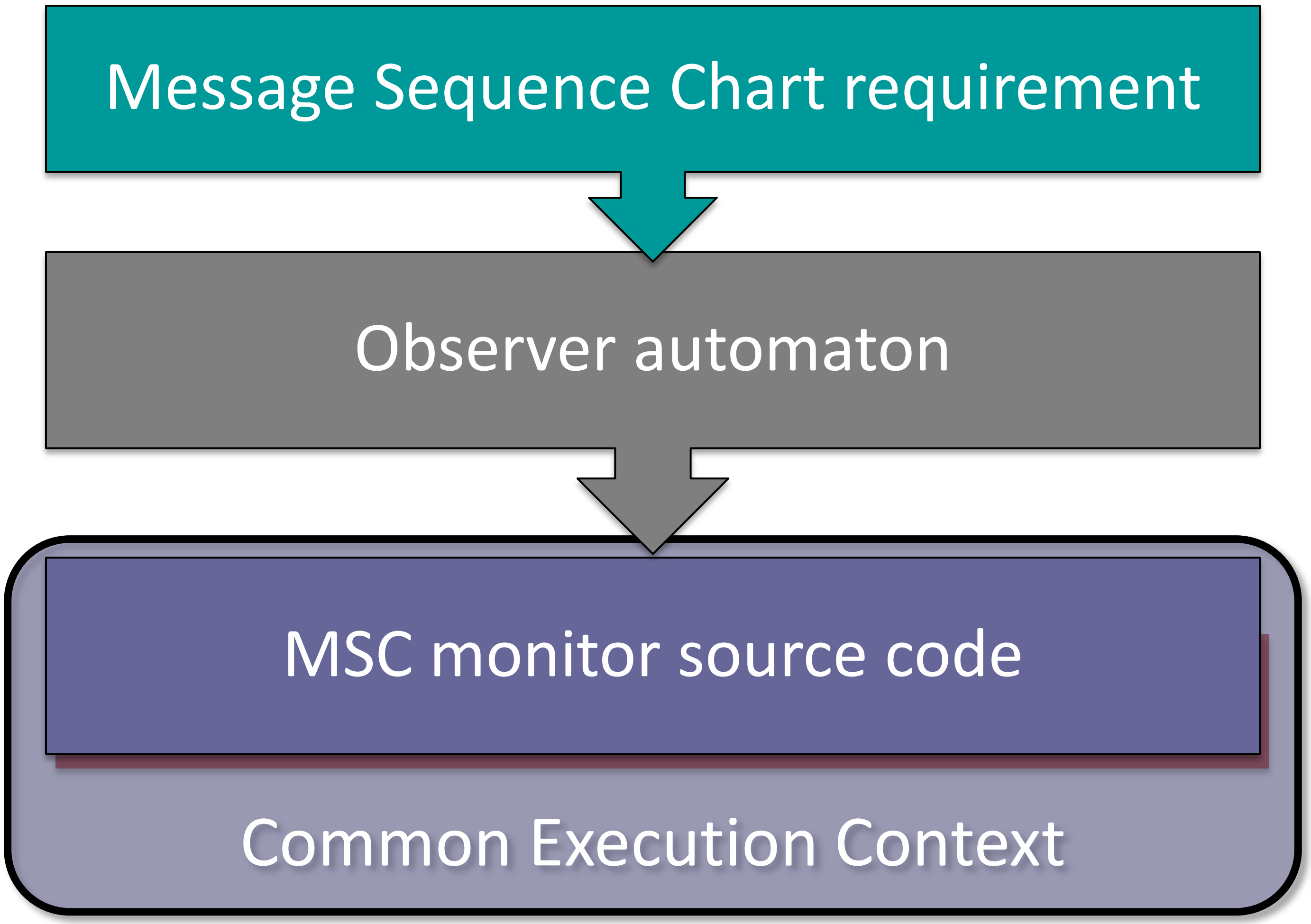


- Optional:



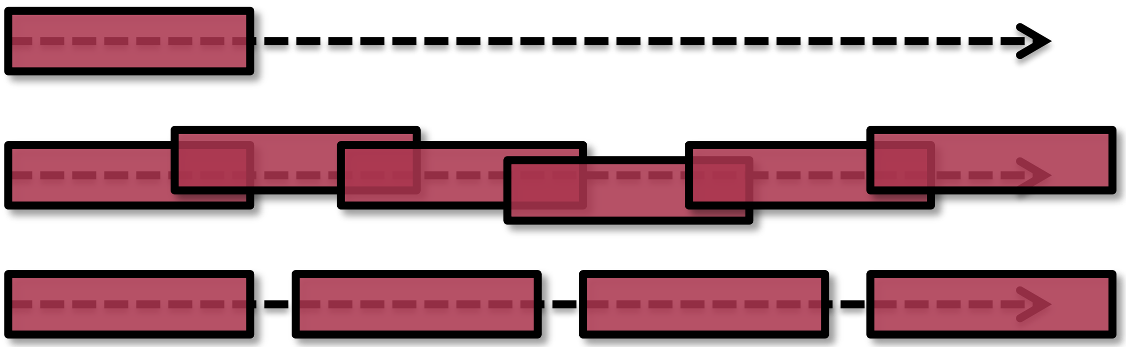


Steps of monitor synthesis



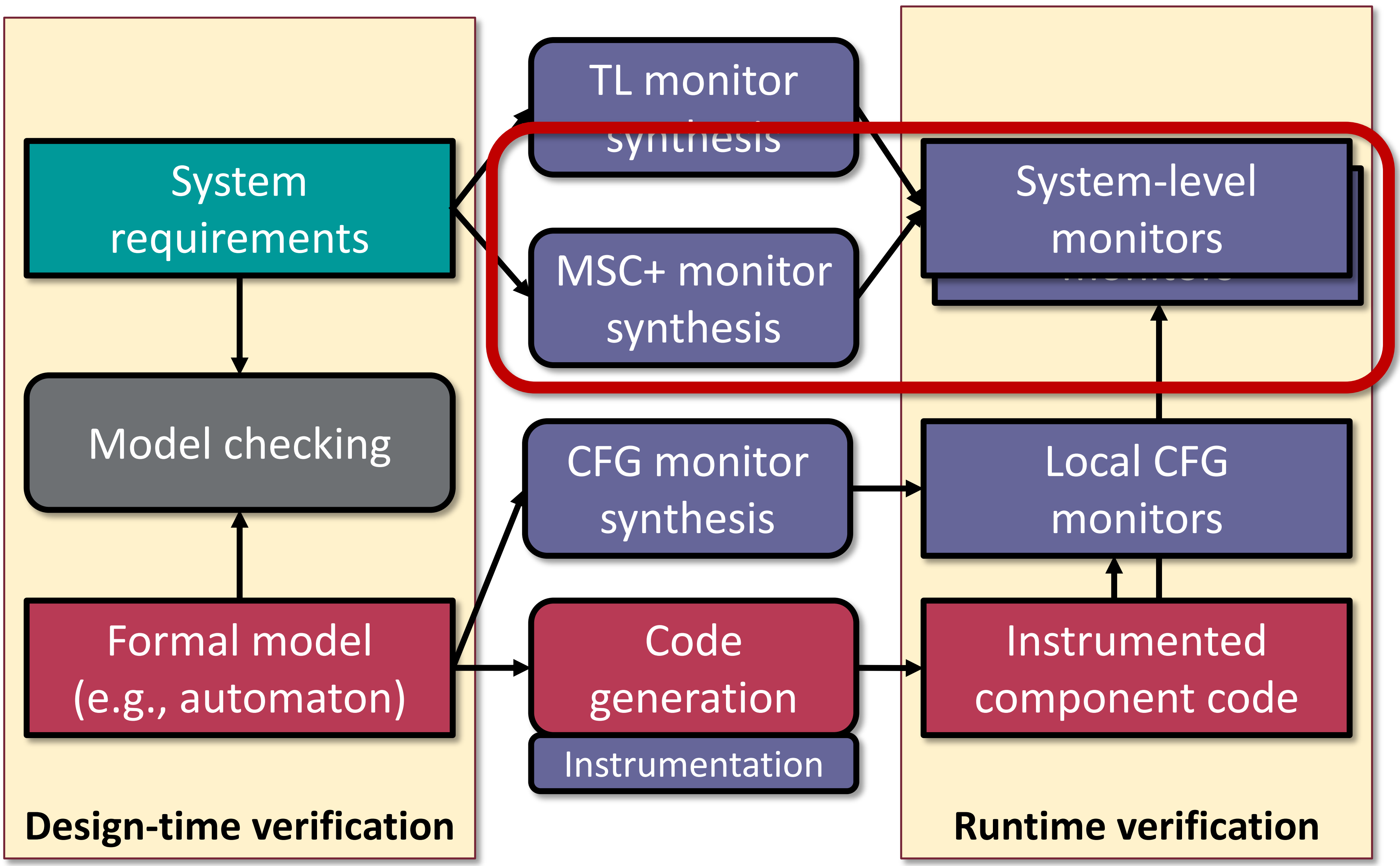
Execution context for the monitors

- Execution scheduler for monitor instances
  - Responsible for starting / stopping the monitors
  - Management of error notifications and status
- Activation modes of monitoring
  - Initial
  - Invariant
  - Iterative



Runtime verification based on scenario and context description

Overview: Runtime verification



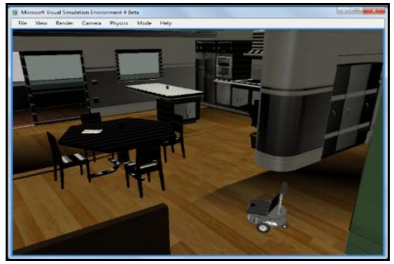
New challenges

- Monitoring autonomous systems
  - Context-aware behaviour (perceived environment)
  - Adaptation to changing context (decisions, strategy)
- Specification of requirements: Scenarios
  - Behaviour: Sequences of events / actions
    with condition (prechart) and assertion (main chart)
  - Including references to situations in the context
- Monitoring context-aware systems
  - Observing the changes in the context of the system
  - Checking the behaviour of the system itself

Monitoring setup

Real environment

OR



Simulator

Requirements

Runtime traces

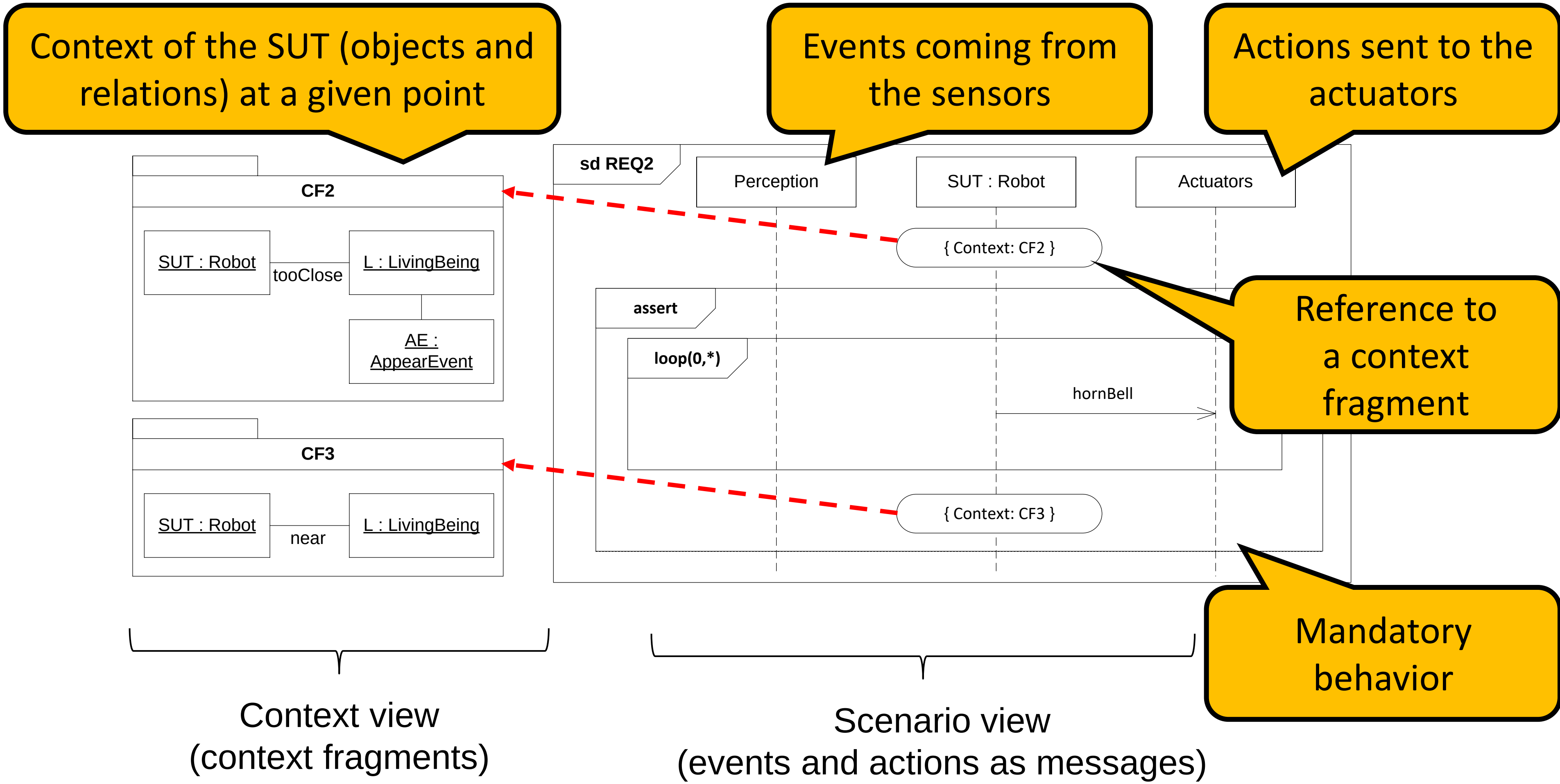
Observed events, actions, and **context changes**

Challenge:
Checking a runtime trace w.r.t. the scenario based requirements efficiently

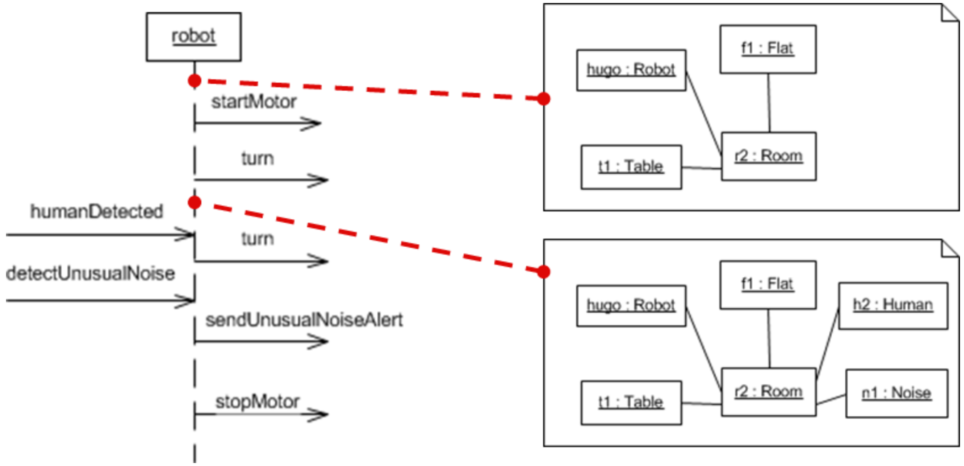
Trace evaluation by monitors

Formalization of requirements

- Scenarios of events/actions based on MSC
- Extensions for referencing contexts

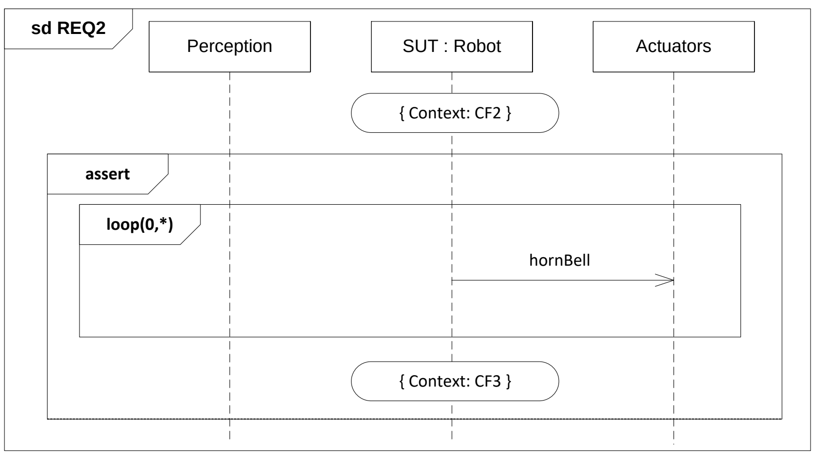


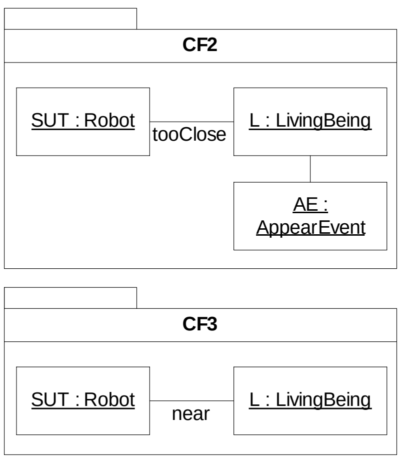
Tasks of the monitor



Observed trace:

- Events and actions of the SUT
- Concrete configurations of the context



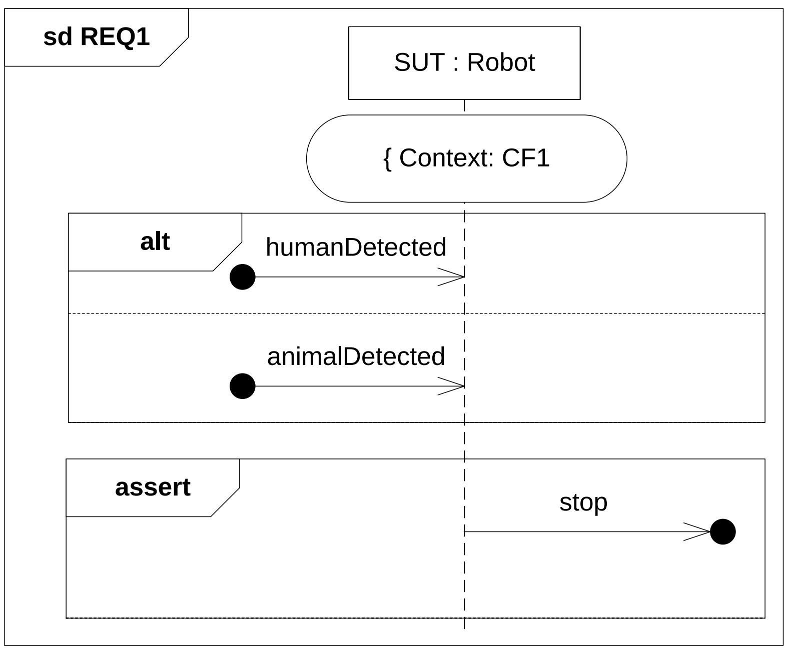


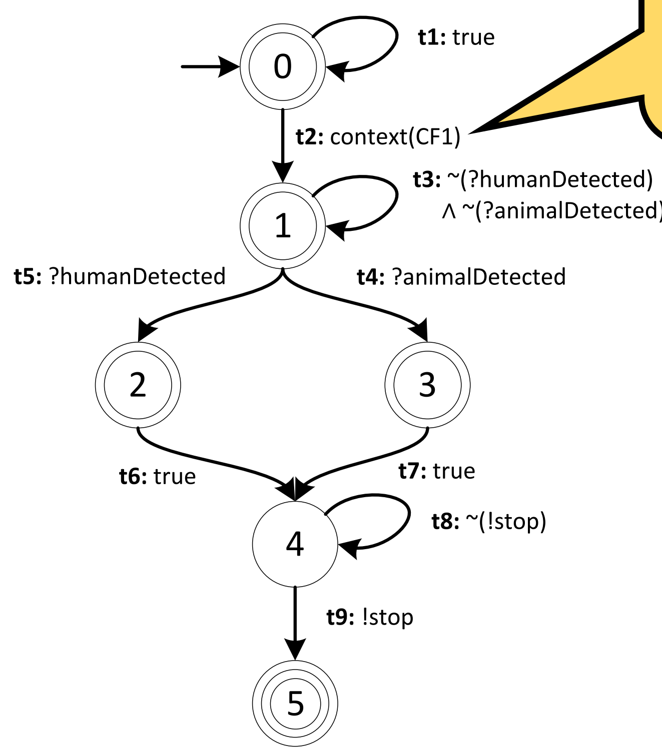
Matching messages:
Observer automaton

Matching context fragments:
Graph matching

Construction of the observer automaton

- One observer automaton for each req. scenario
  - Structure of the observer: like for MSC
  - Transitions: events, actions, or context changes
  - State types: not triggered / violated / satisfied



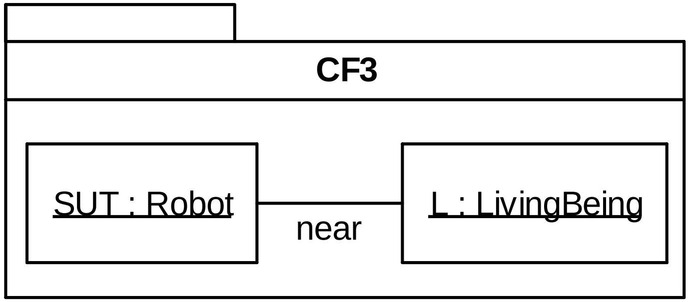


Context matching is included in checking

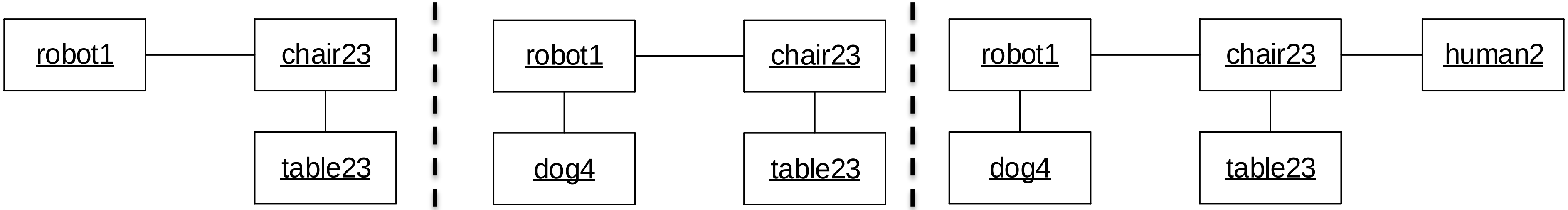
Context matching as graph matching

- Checking sequences of contexts observed in a trace
  - Graph based representation of the contexts
  - Matching of context graph fragments (requirements) to context graph sequences (observed trace)

Context fragment (requirement):

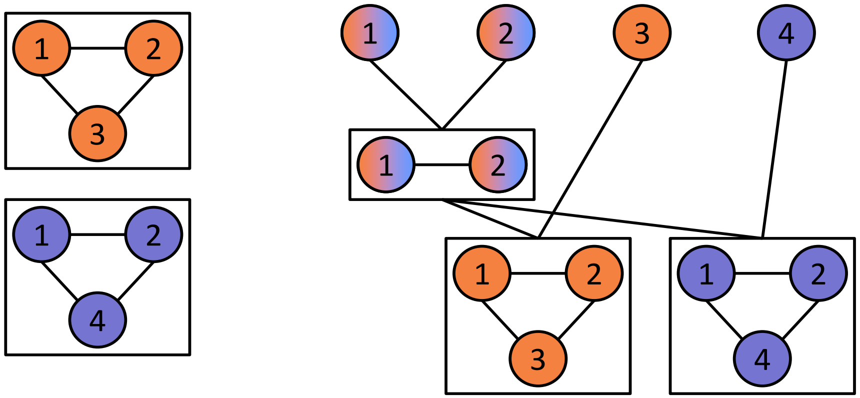


Observed trace:

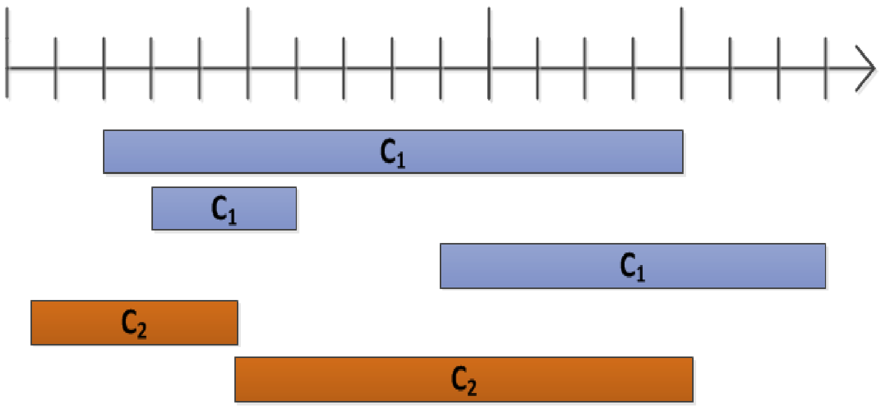


Specific problems of graph matching

- Matching all requirement scenarios to a trace
  - Decomposition of the context fragments to store and match common parts only once



- Matching context fragments of requirements at each step of the trace
  - Concurrent threads of monitors (evaluation) are started when matching is detected



Handling abstract relations

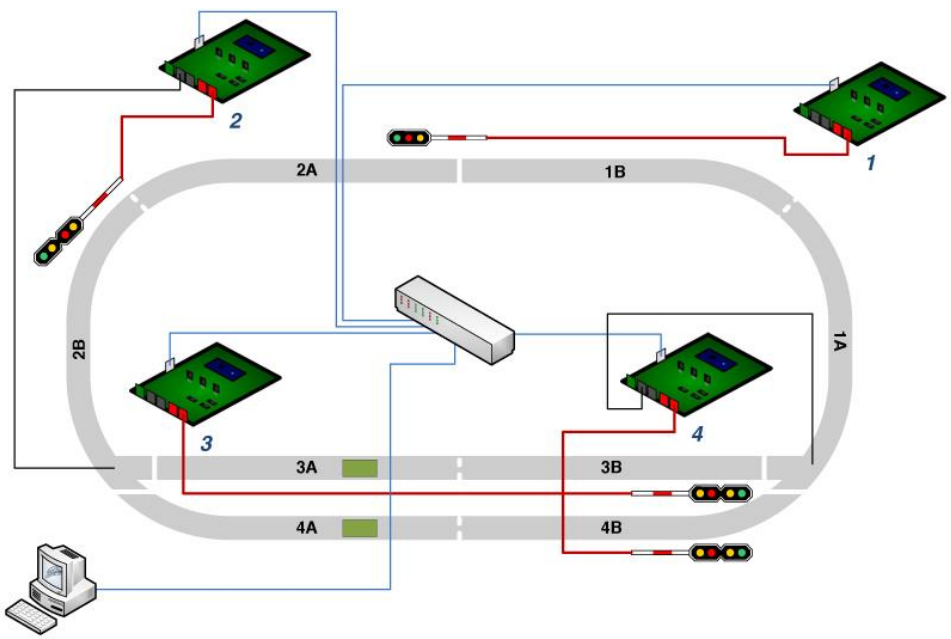
- Peculiarities in requirement properties
  - Abstract relations (e.g., “near”)
  - Hierarchy of objects (e.g., “desk” is a “furniture”)

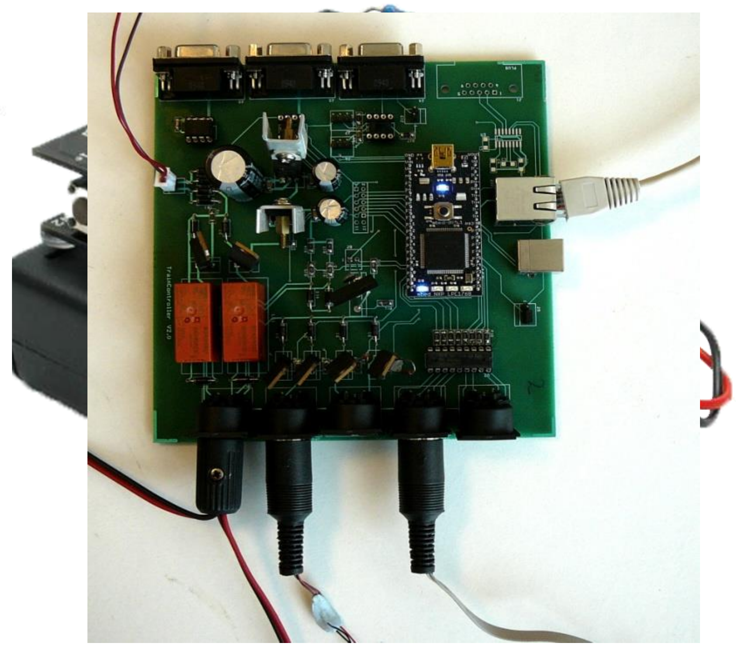
- Handling peculiarities in the monitor
  - Preprocessing the trace to derive abstract relations
  - Using compatibility relation when matching context elements

Implementation experience

Implementation of TL and LSC monitoring

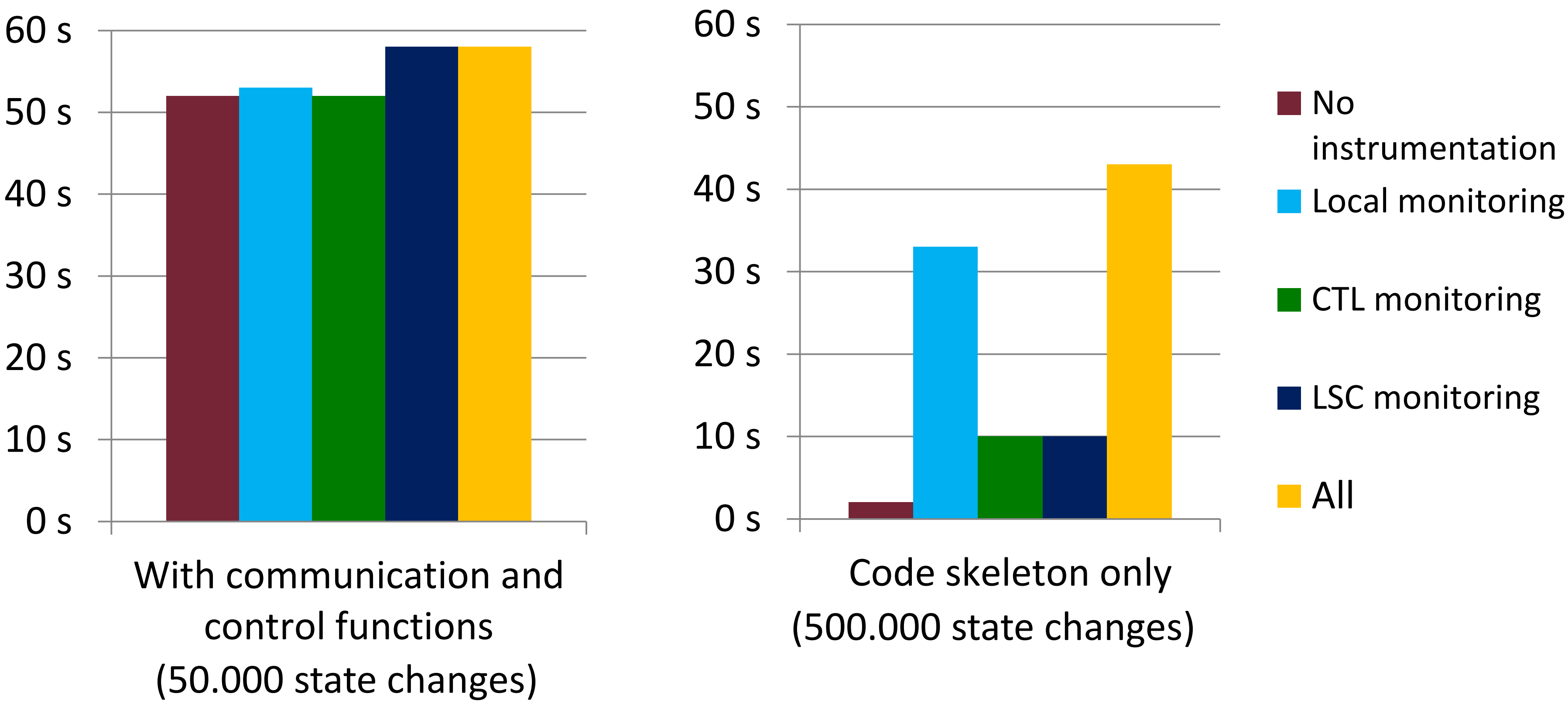
- Realized for two different embedded platforms
  - motes with wireless communication modules
    - Industrial case study: Bit synchronization protocol
  - mbed rapid prototyping microcontroller
    - Educational demonstrator: train controller system





Time overhead

- Execution time on the mbed platform

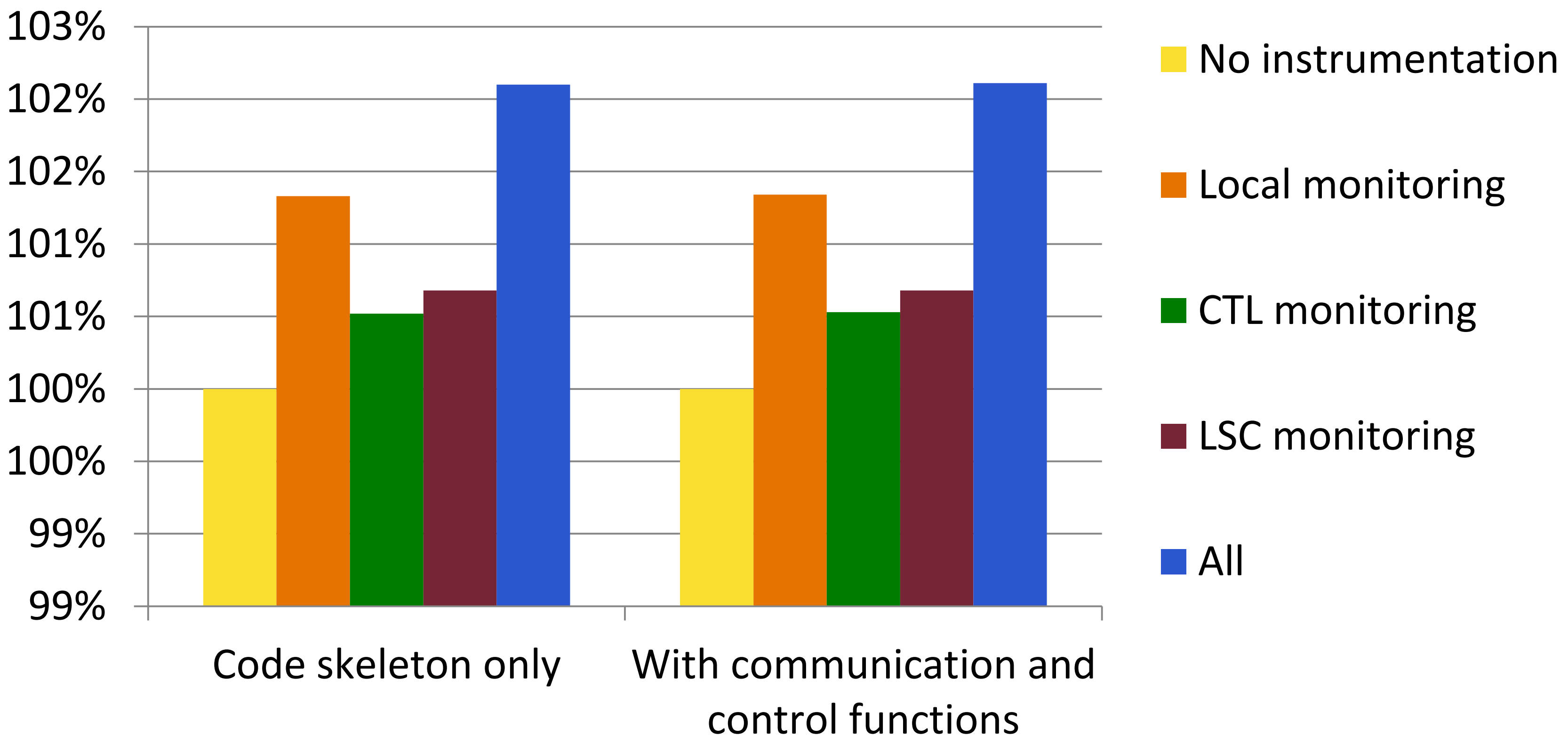


Complex control functions:
Less than 12% overhead

Simple control functions:
Larger overhead can be expected

Code (memory) overhead

- Code size on the mbed platform



Moderate overhead: Less than 5%

Implementation of scenario monitoring

- Prototype implementation
  - Scenario based requirements: In Eclipse UML2
  - Monitor: Java application
- Complexity is determined by the graph matching
  - Best case: $O(IM)$, worst case: $O(NI^{M}M^{2})$
    - N: number of requirement graph fragments to be matched
    - M: average size of requirement graph fragments
    - I: number of vertices in the context graph (in observed trace)
  - Requirement graphs (context fragments) are usually small (thus M is low)

Summary

Monitor synthesis for

- Runtime verification in critical systems
- Test oracles (test evaluation) in testing frameworks