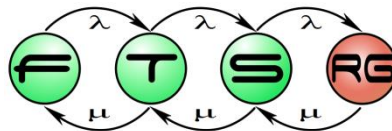


Runtime verification

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**

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

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)
- Understand how **context-dependent behavior** can be monitored (K1)

Goals and challenges

What is runtime verification?

■ Definition:

Checking the behavior of systems

- in runtime (online),
- based on formally specified properties

■ Motivation

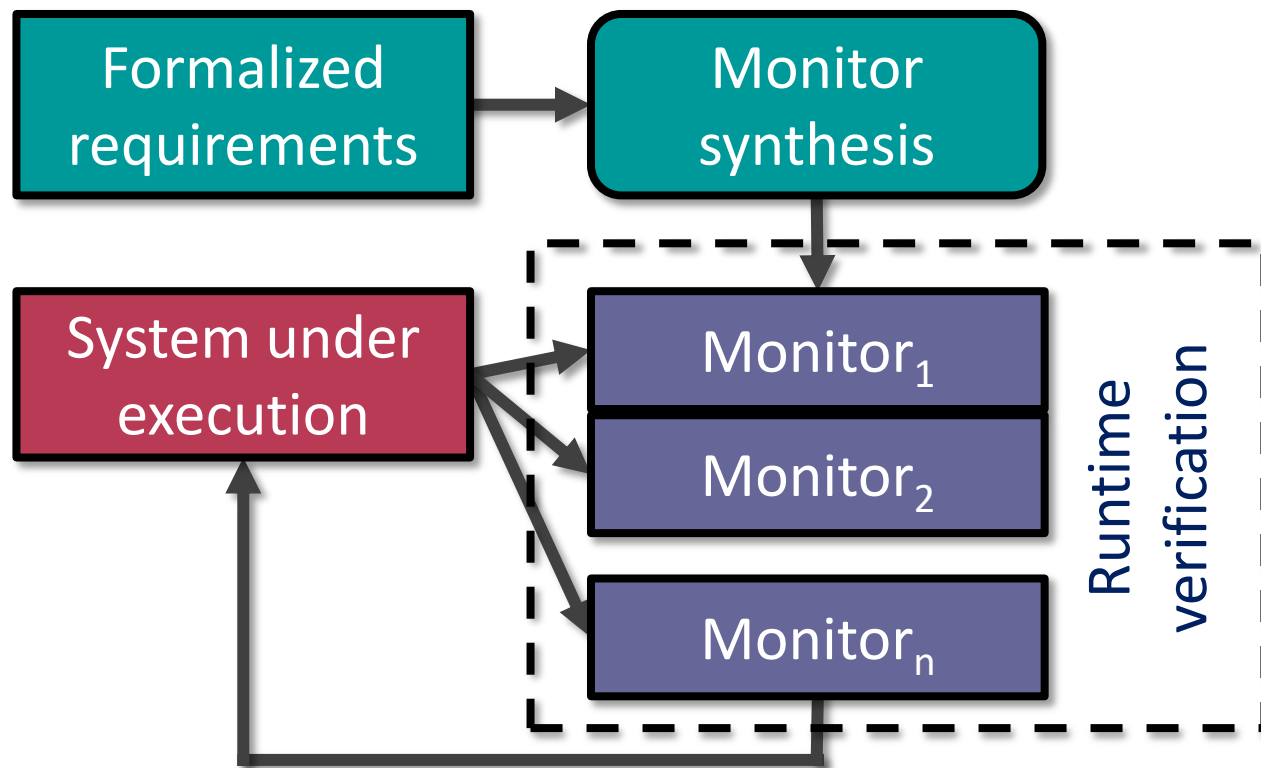
- Dependability and safety requirements
 - IT services: Correct service to be provided
 - Safety-critical systems: Hazardous states to be avoided
- Runtime faults are inevitable
 - Random faults in hardware components
 - Software design, implementation, configuration faults

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 control flow graph (CFG)
 - 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 components
- **Example: Reactive fault handling**
 - Fault detection followed by reaction (e.g., recovery, reconfiguration, setting of safe state, ...)
- Components for runtime fault detection: **Monitors**

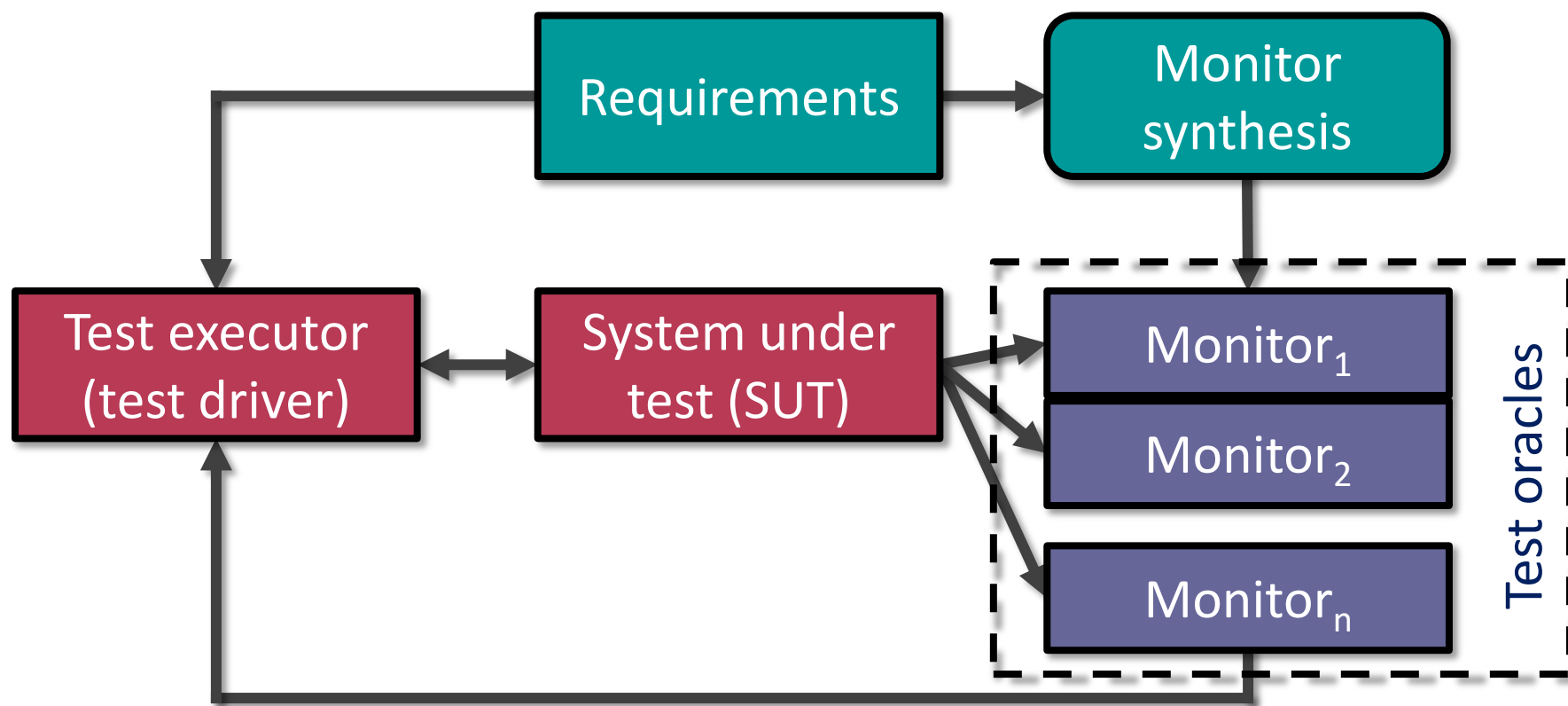
Use case 1: Runtime verification

- 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

○ Formalization of checked properties

- Execution trace based checking of **temporal properties**
 - Temporal logics
 - Reference automata
 - Regular expressions
- **Design-by-contract** based monitoring
 - Executable assertions
- **Specification-less** monitoring
 - 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

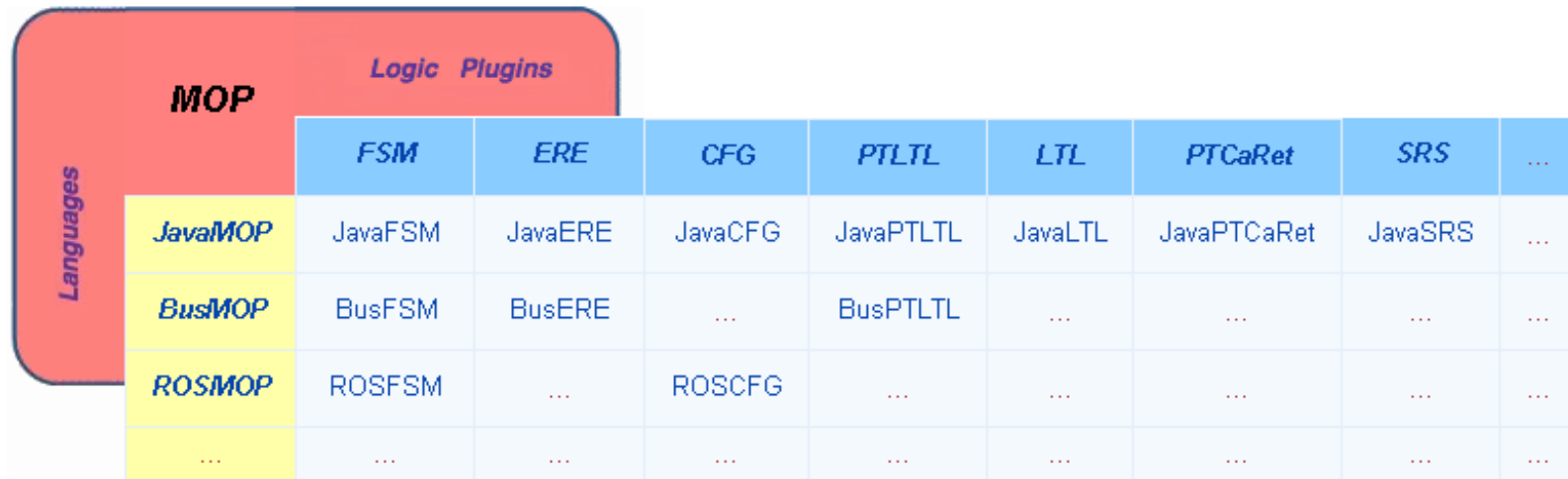
- **Active** and **passive** instrumentation
 - Active: inserting source code snippets into observed code
 - Passive: observation without interference
- Techniques for active instrumentation
 - Aspect-Oriented Programming (AOP)
 - Tracematch: AspectJ extension for trace patterns
- **Synchronous** and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



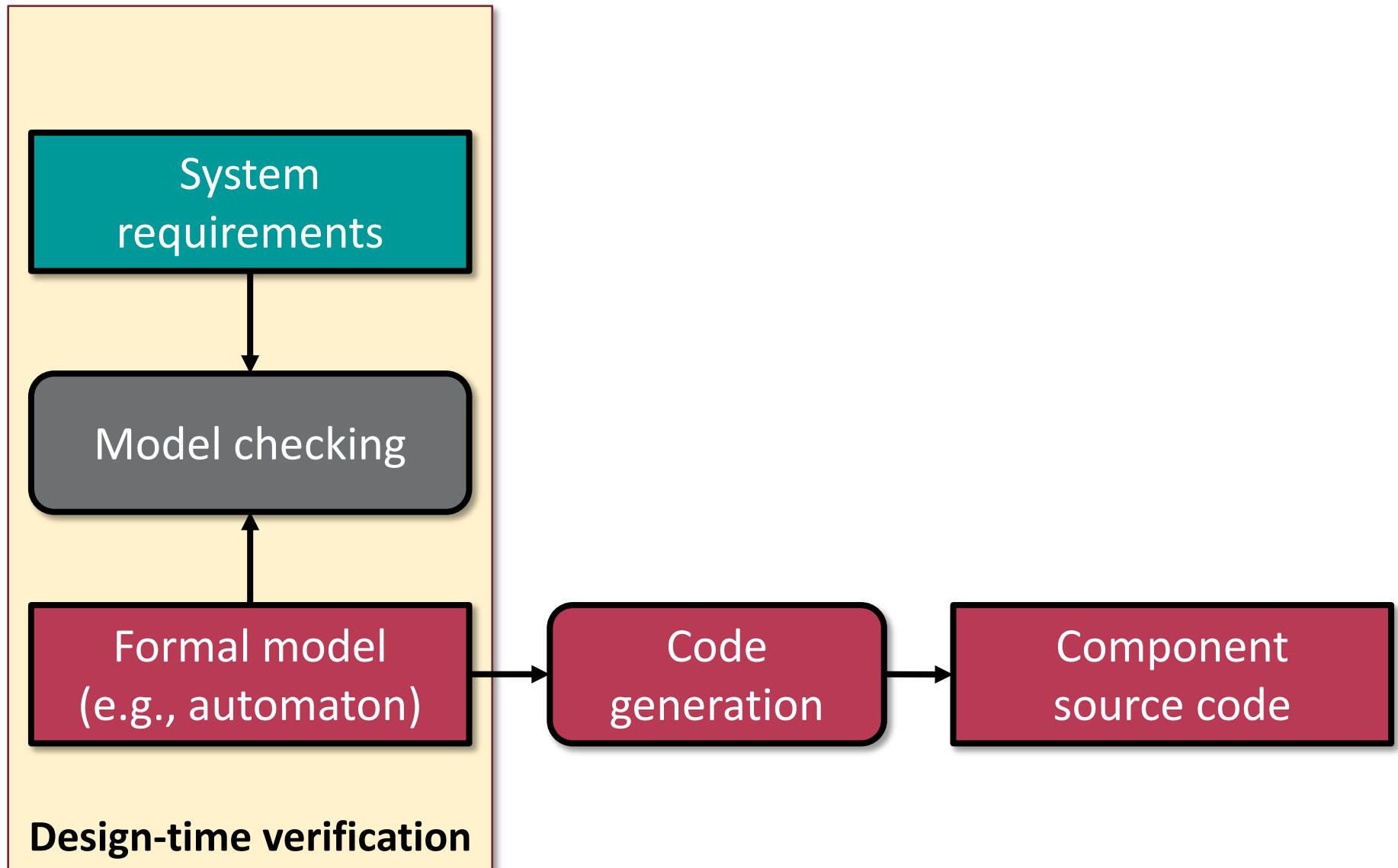
		<i>Logic Plugins</i>							
<i>Languages</i>	<i>MOP</i>	<i>FSM</i>	<i>ERE</i>	<i>CFG</i>	<i>PTLTL</i>	<i>LTL</i>	<i>PTCaRet</i>	<i>SRS</i>	...
	<i>JavaMOP</i>	JavaFSM	JavaERE	JavaCFG	JavaPTLTL	JavaLTL	JavaPTCaRet	JavaSRS	...
	<i>BusMOP</i>	BusFSM	BusERE	...	BusPTLTL
	<i>ROSMOP</i>	ROSMOP	...	ROSCFG

- 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

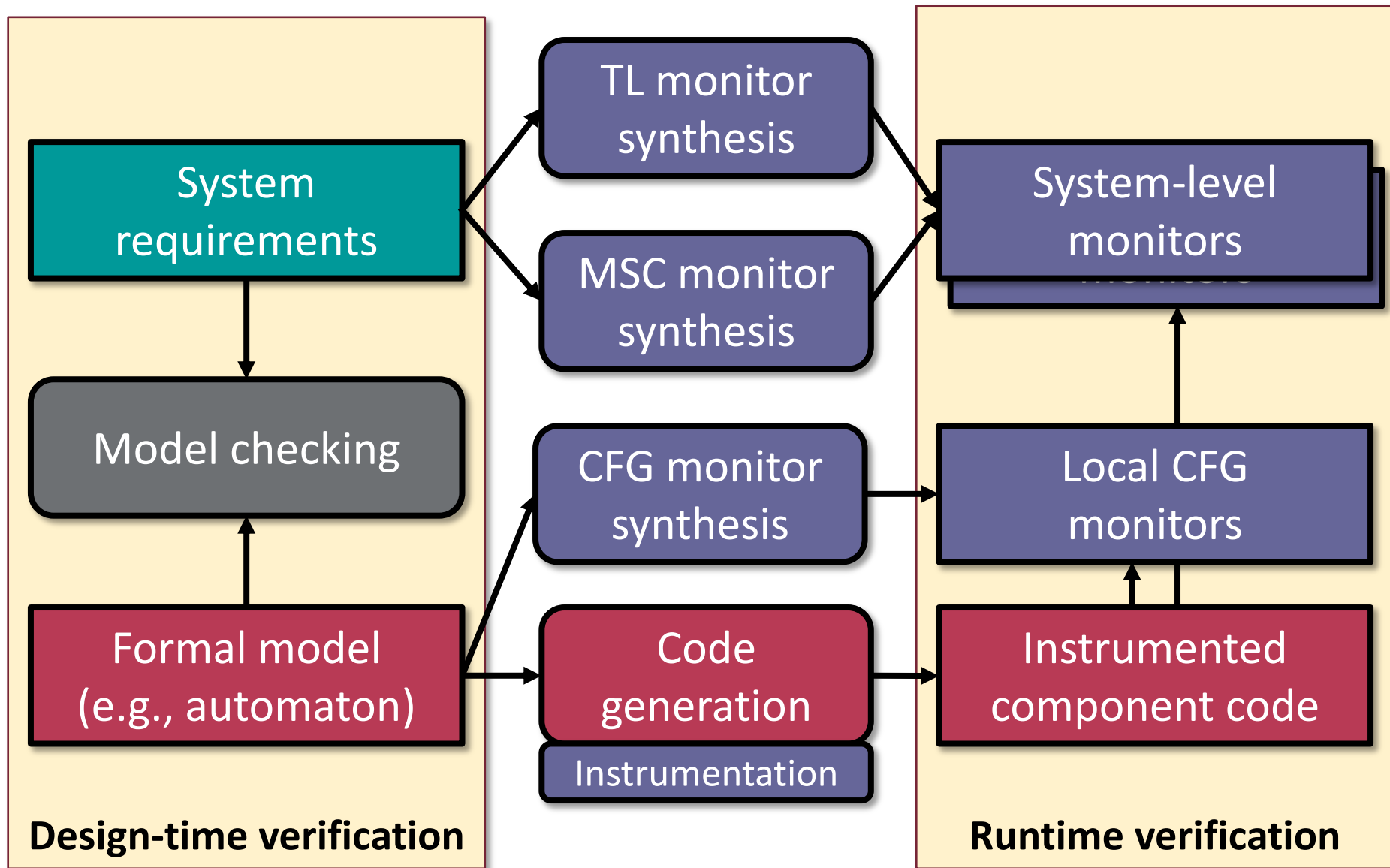
The presented solutions

- 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 (Temporal Logic, TL) properties of interactions
 - Scenario (Message Sequence Chart, MSC) based properties
- Relation to model based design
 - Model based code generation with instrumentation

Overview: Design-time verification

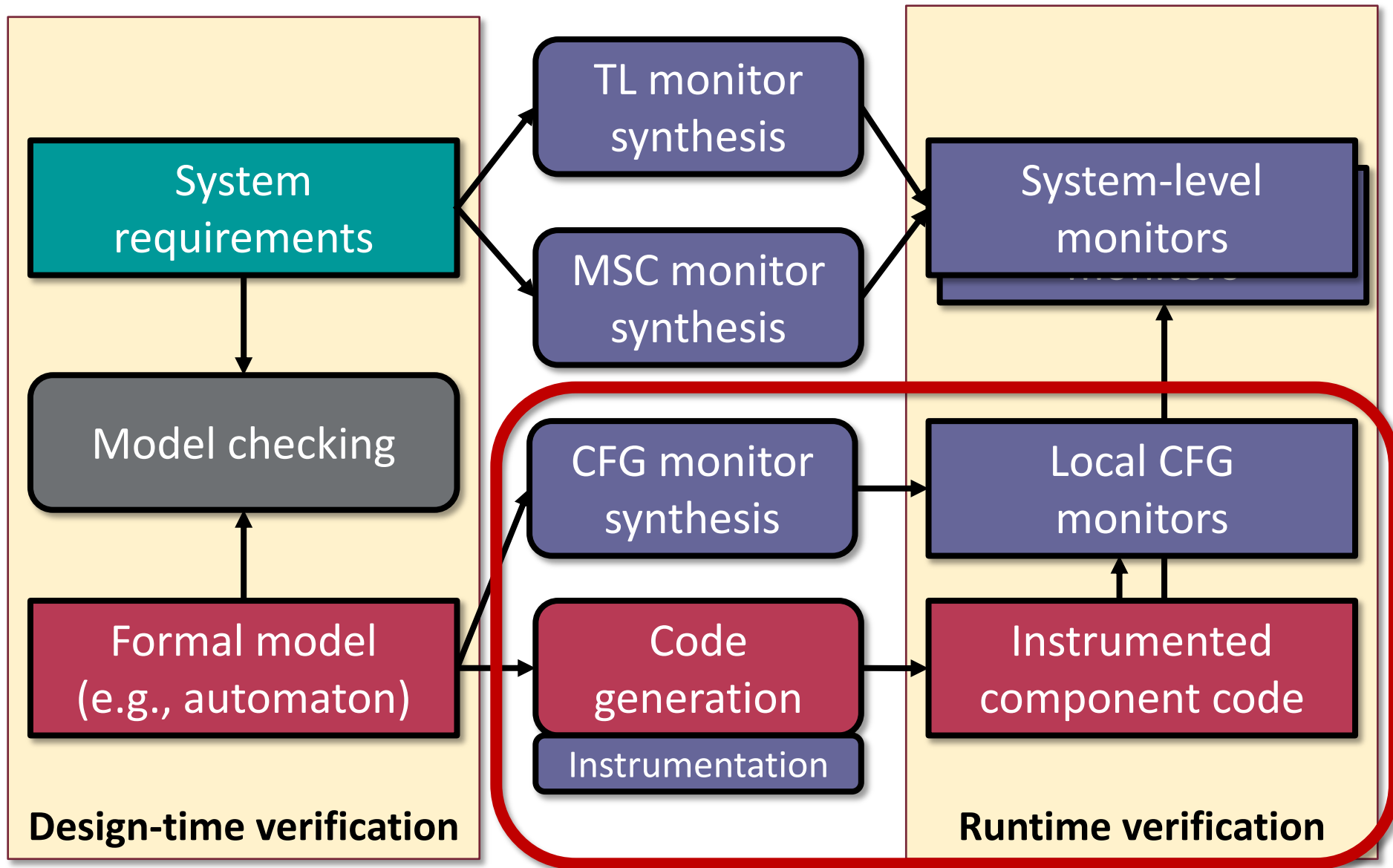


Overview: Runtime verification

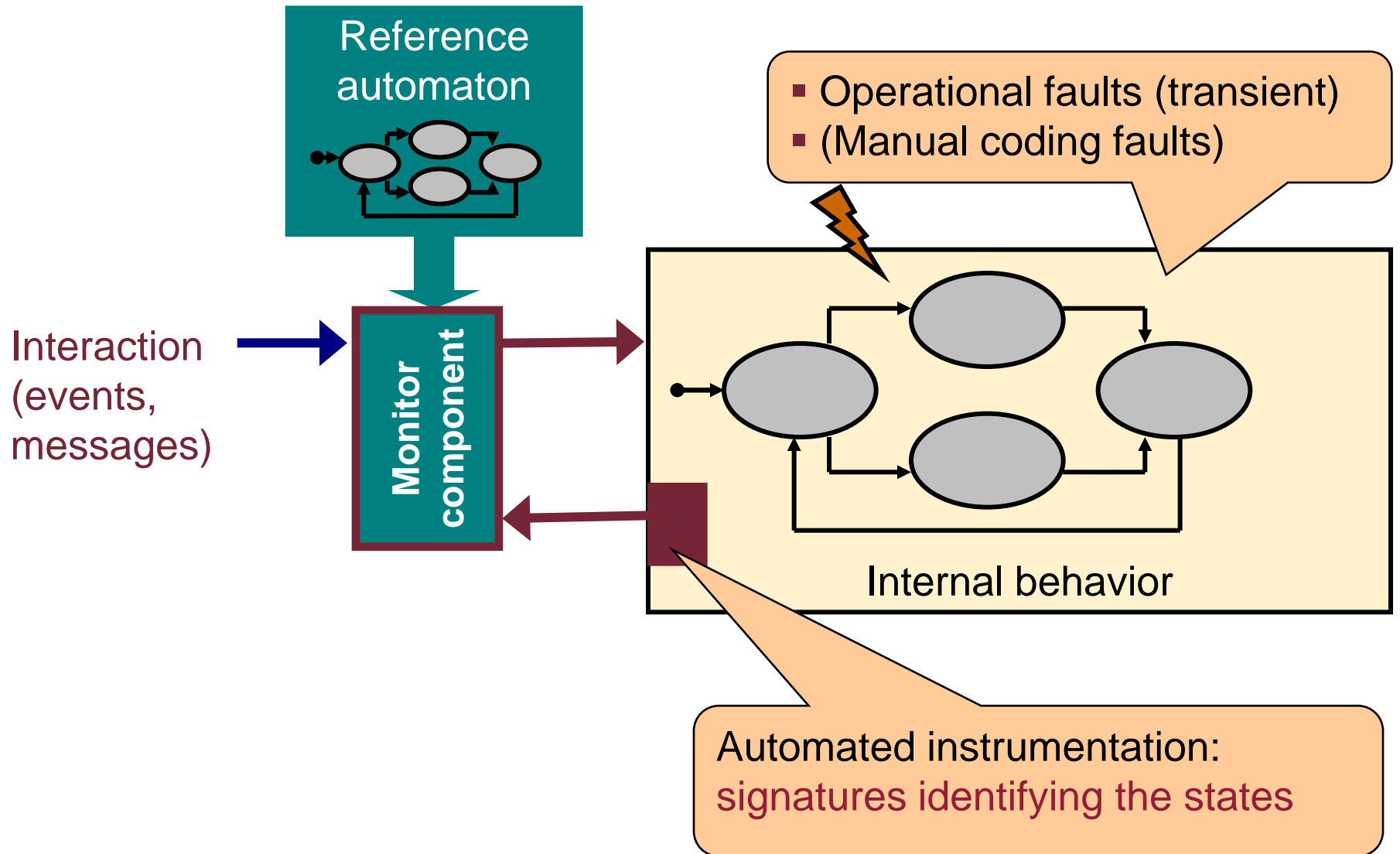


Runtime verification based on reference automata

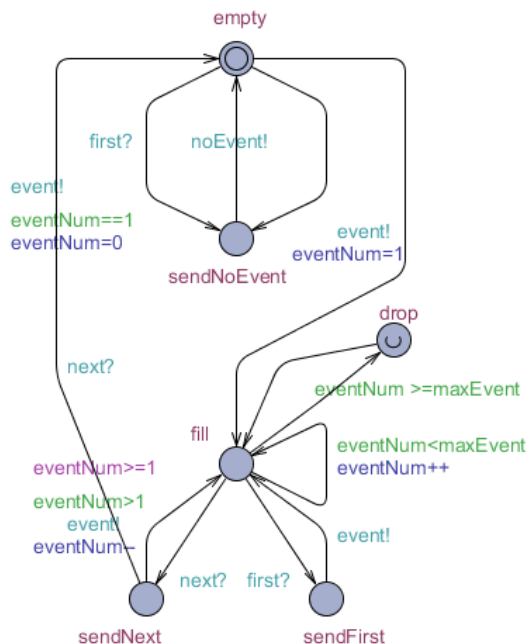
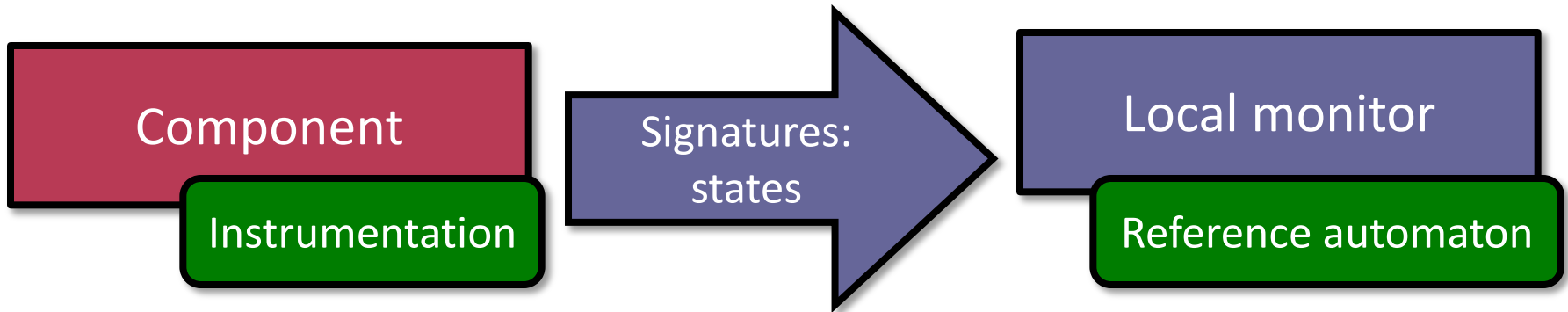
Overview: Runtime verification



Monitoring on the basis of reference automaton

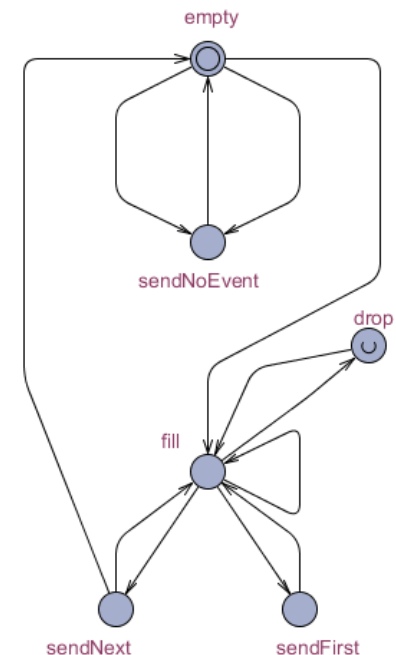


State-based monitoring of timed automata



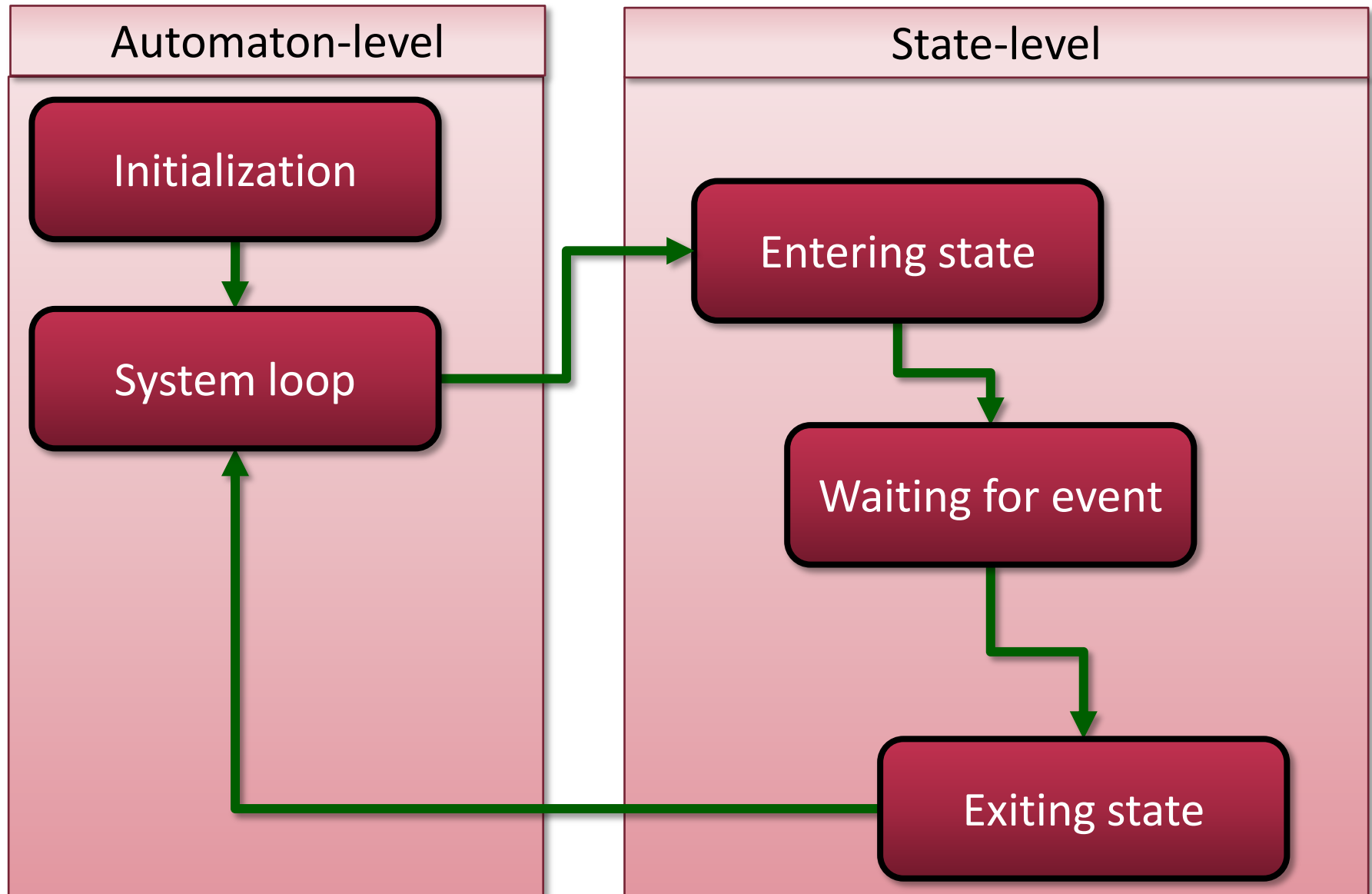
Basis for code generation

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

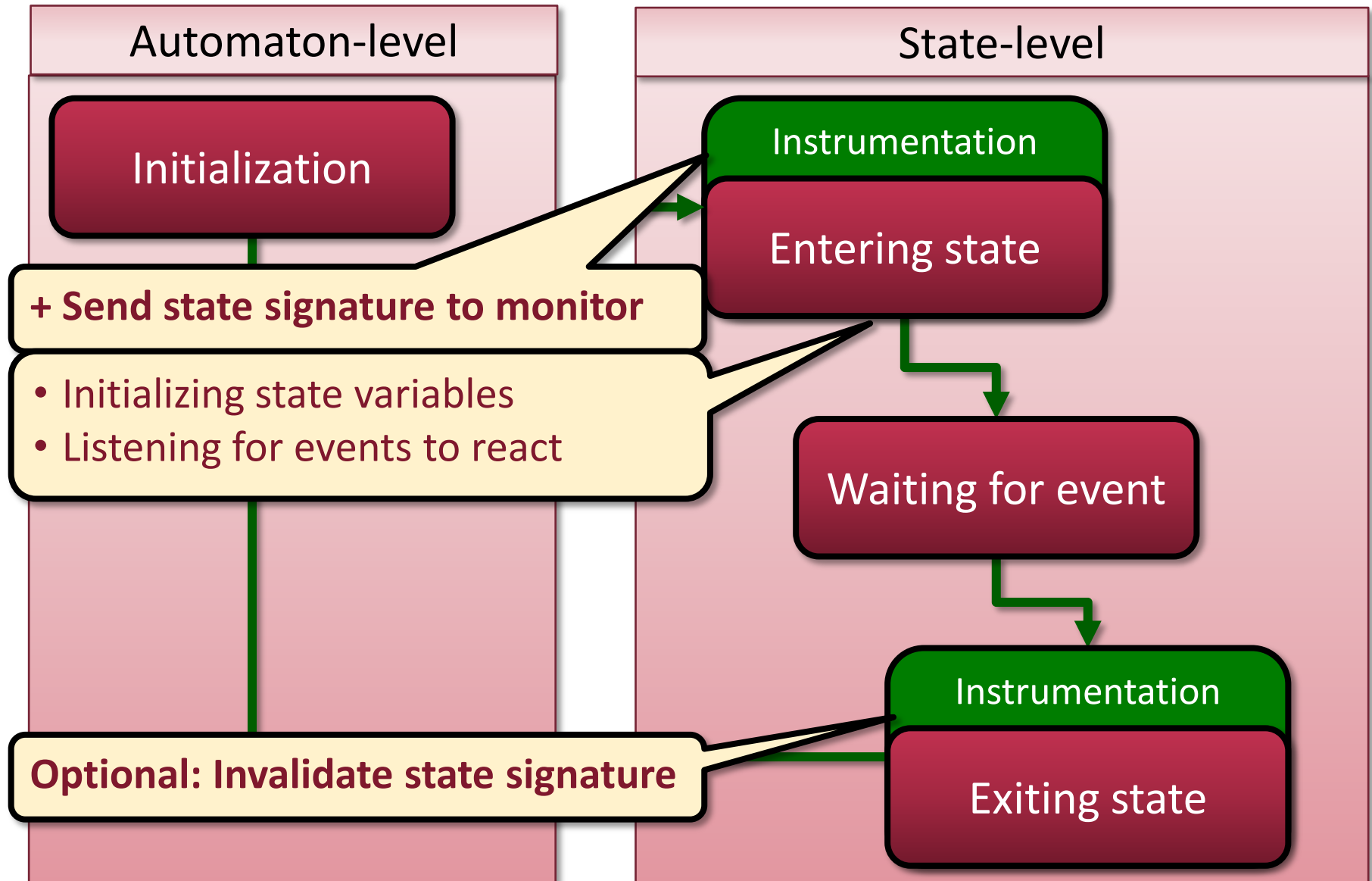


Reference automaton

Instrumentation in the generated source code

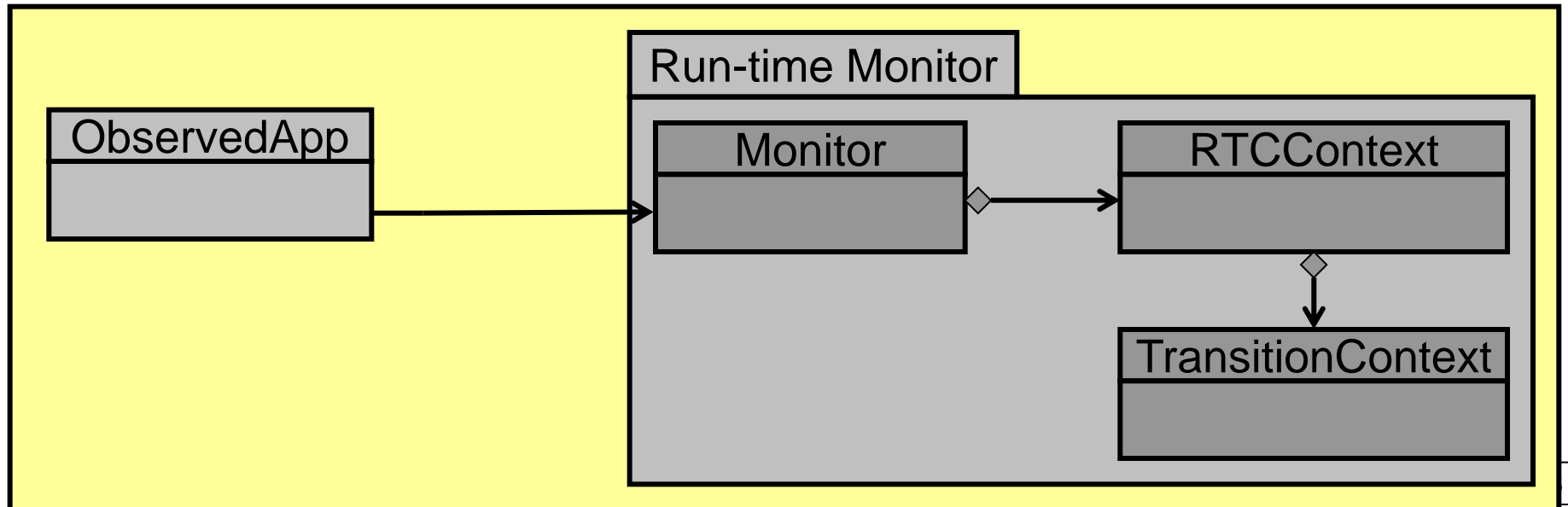


Instrumentation in the generated source code



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

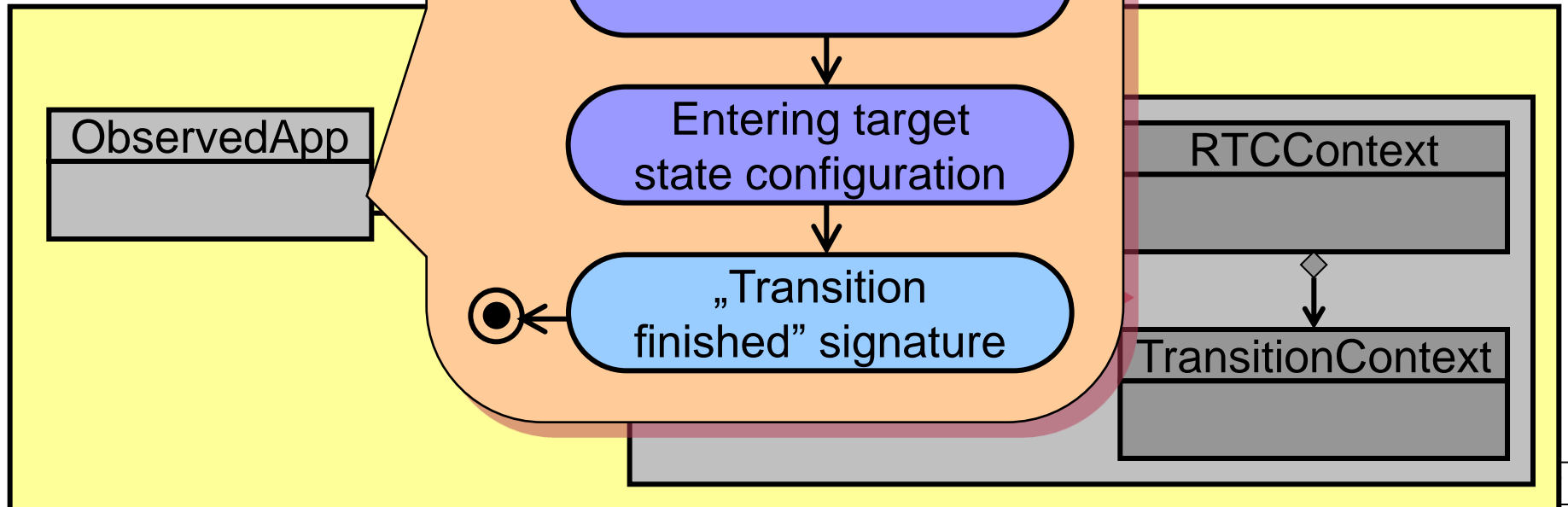


Case study: Modeling the execution of statecharts

- Systematic analysis
 - Explicit information
 - States entered
 - Executed actions
 - Instrumentation

Representation:

Programming



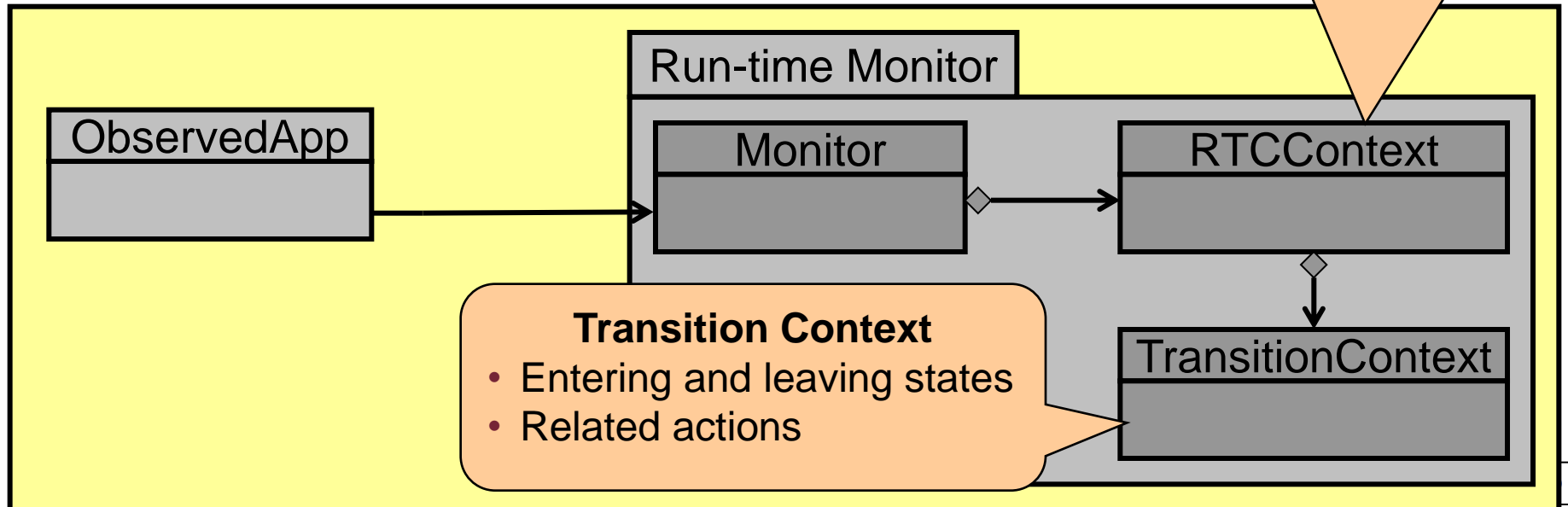
Case study: Monitoring on the basis of statecharts

■ Systematic and transparent instrumentation:

- Explicit information for the monitor
 - States entered and left
 - Executed actions
- Instrumentation: Aspektus-oriented

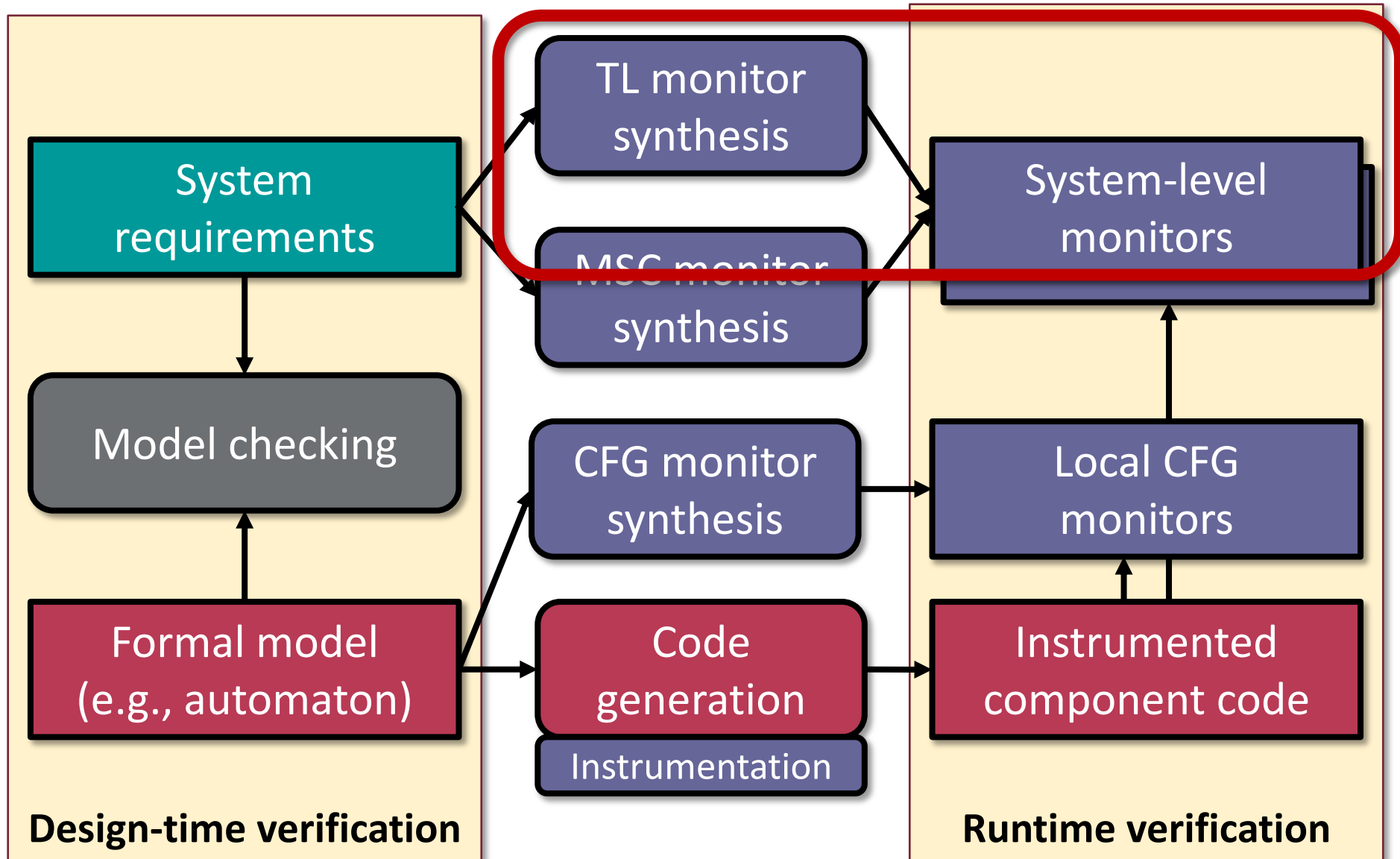
RTC context

- Initialization
- Starting and finishing event processing
- Signals for Transition Context: fired transitions



Runtime verification based on temporal logic properties

Overview: Runtime verification

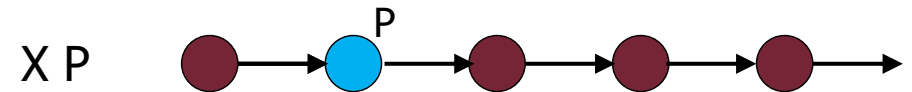


Linear temporal logic properties

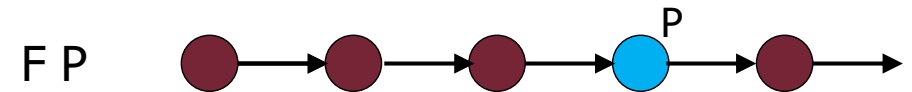
Elements of the linear temporal logic (LTL):

- Atomic propositions: State properties P , Q , ...
- Boolean operators: \wedge , \vee , \neg , \Rightarrow
- Temporal operators: X , F , G , U , informally:

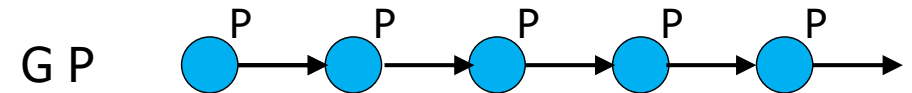
- $X p$: “neXt p ”
 p holds in the next state



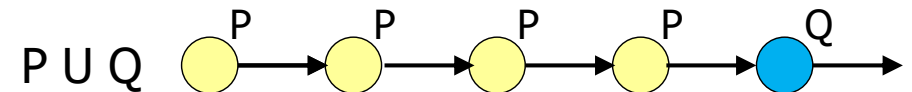
- $F p$: “Future p ”
 p holds eventually
on the subsequent path



- $G p$: “Globally p ”
 p holds in all states
on the subsequent path



- $p U q$: “ p Until q ”
 p holds at least until q ,
which holds at the subsequent path

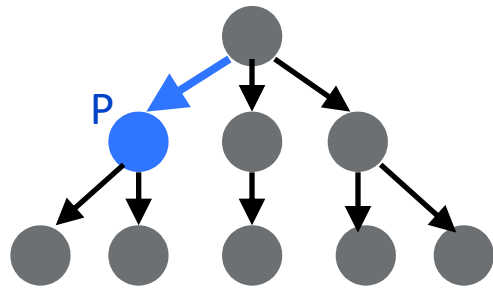


Branching temporal logic properties

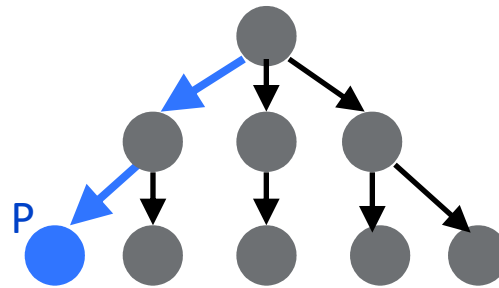
Quantifiers on paths starting from a given state:

- $E\ p$ (Exists p): **there exists** a path on which p holds
- $A\ p$ (for All p): **for all** paths from the state p holds

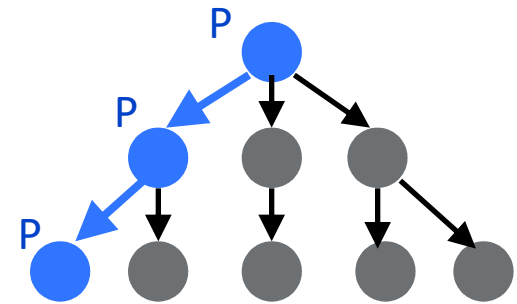
Combined with LTL temporal operators: CTL*, CTL



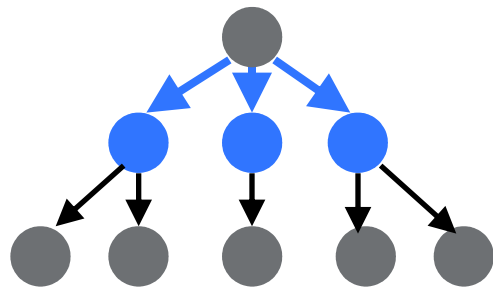
EX P



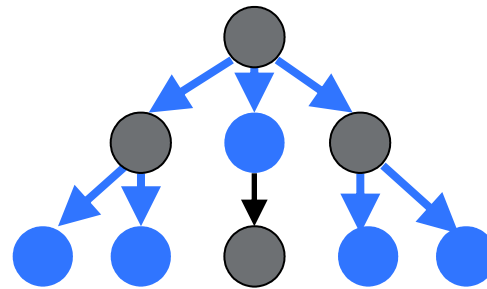
EF P



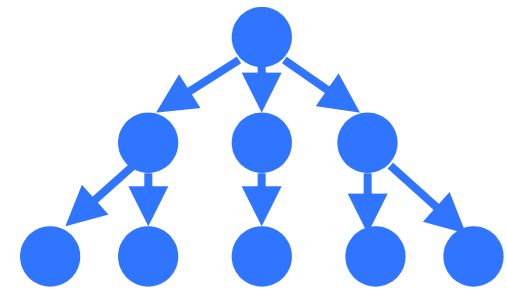
EG P



AX P



AF P

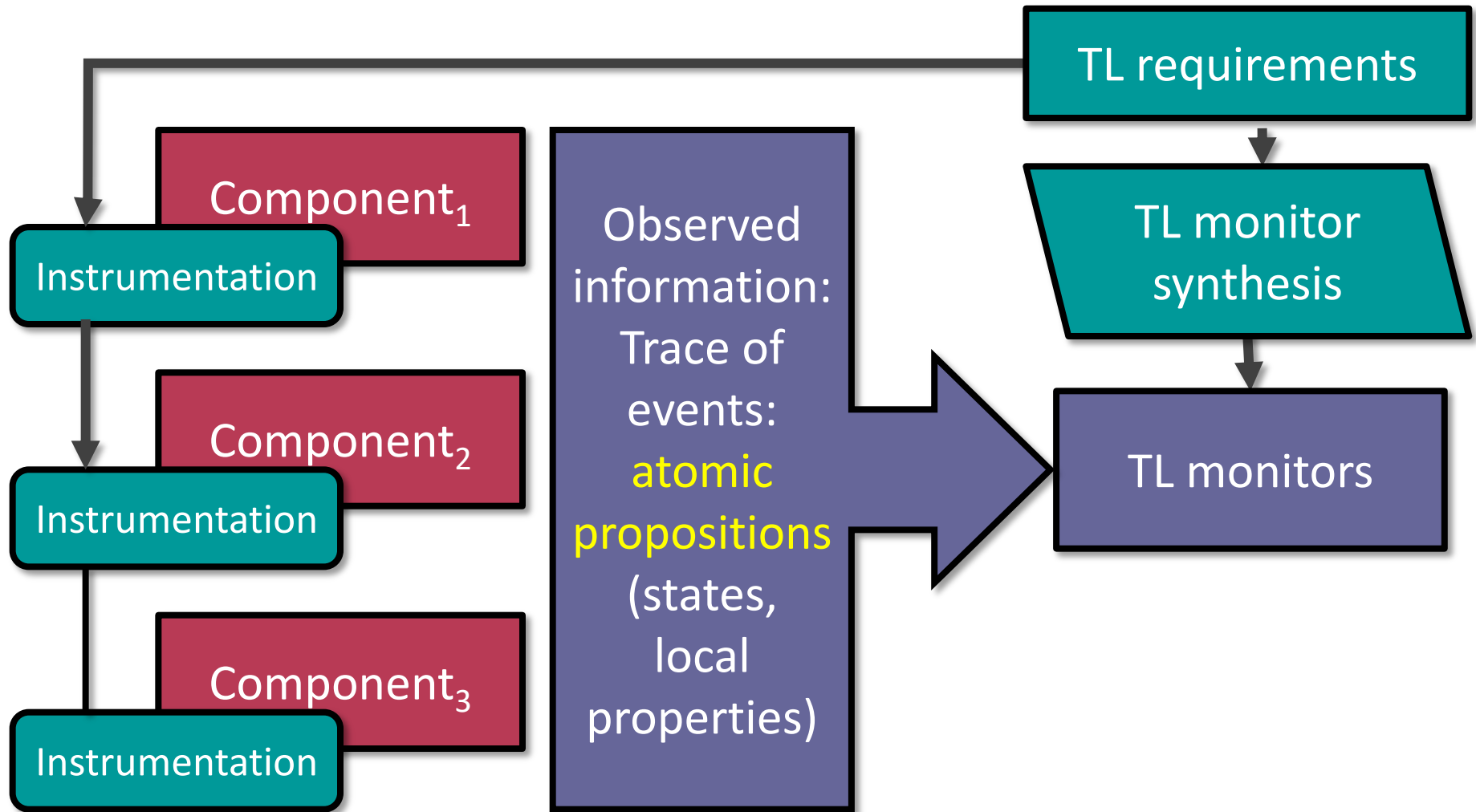


AG P

Temporal logic based properties

- Properties: Sequence and reachability of states/events
 - **Safety** properties: Invariants for all states
 - **Liveness** properties: Reachability of desired states
- Runtime checking LTL properties
 - Use case: Checking **observed trace** in runtime
 - Finite or infinite trace (continuously operating systems)
- Runtime checking CTL properties
 - Use case: Checking the **paths explored** during testing
 - Each path is an execution on the basis of test inputs
 - Path quantifiers (exists, forall) can be evaluated

Setup of TL based monitoring

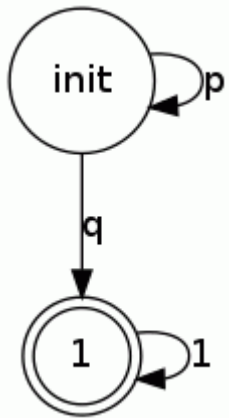


Monitor synthesis for LTL properties (1)

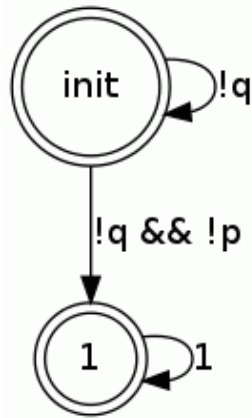
Basic idea: Construction of two accepting automata

- A^φ : accepts event sequences on which the **original property** holds
- $A^{\neg\varphi}$: accepts event sequences on which the **negated property** holds

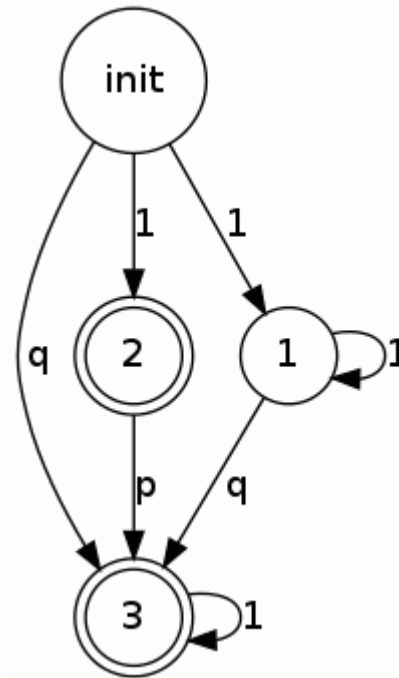
E.g.: $p \cup q$



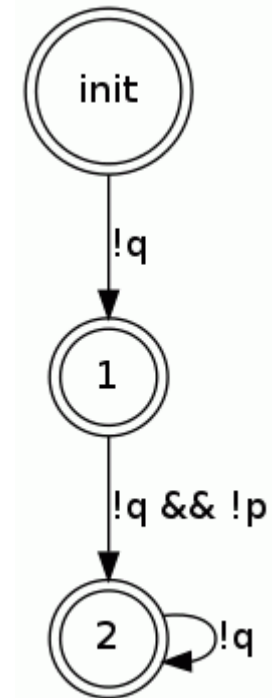
$\neg(p \cup q)$



$(X p) \vee (F q)$



$\neg((X p) \vee (F q))$



Here ! denotes \neg ,
&& denotes \wedge ,
1 denotes true

Note: Only those states and transitions are shown which contribute to the accepted language

Monitor synthesis for LTL properties (2)

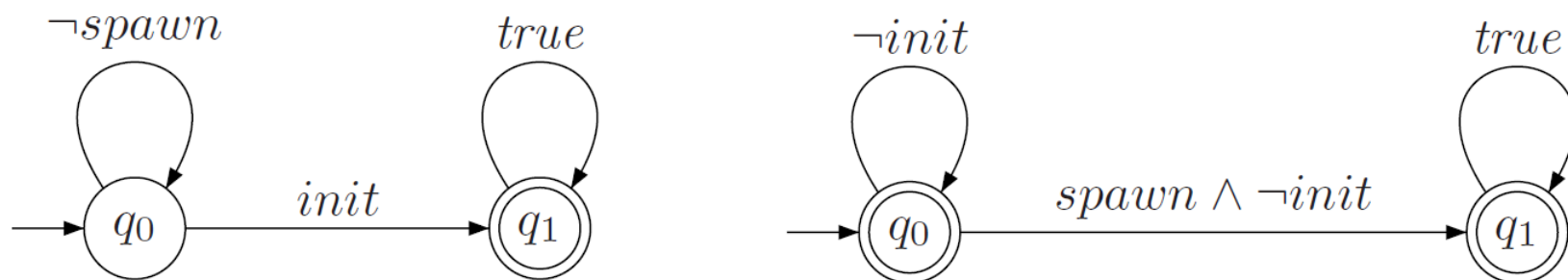
- Labeling states of the automata
 - “Acceptable” state: There is a continuation of the event sequence which may lead to an accepted run (when the property holds)
- **Output** after a sequence of events checked by both automata:
 - “ \perp ” **false** (error detected): Reached state is **not acceptable by A^φ**
 - There is no continuation on which the property holds
 - “T” **true** (property found): Reached state is **not acceptable in $A^{\neg\varphi}$**
 - There is no continuation on which the negated property holds
 - “?” **inconclusive** (no output): **Acceptable by both automata**
 - There are continuations on which the property holds / violated
- **Synthesis of the monitor**: Constructing a **product automaton** from the two automata A^φ and $A^{\neg\varphi}$ in form of an FSM
 - A^φ and $A^{\neg\varphi}$ are first determinized, then the product FSM is minimized

Example: Monitor for an LTL property

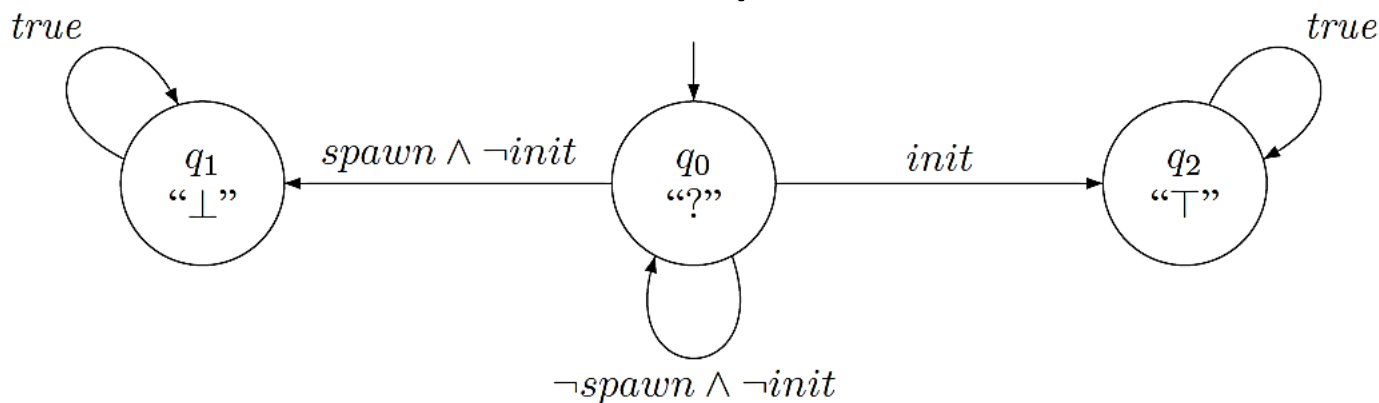
- A process does not get spawned before it is initialized:

$$\varphi = \neg \text{spawn} \cup \text{init}$$

- Automata A^φ and $A^{\neg\varphi}$:

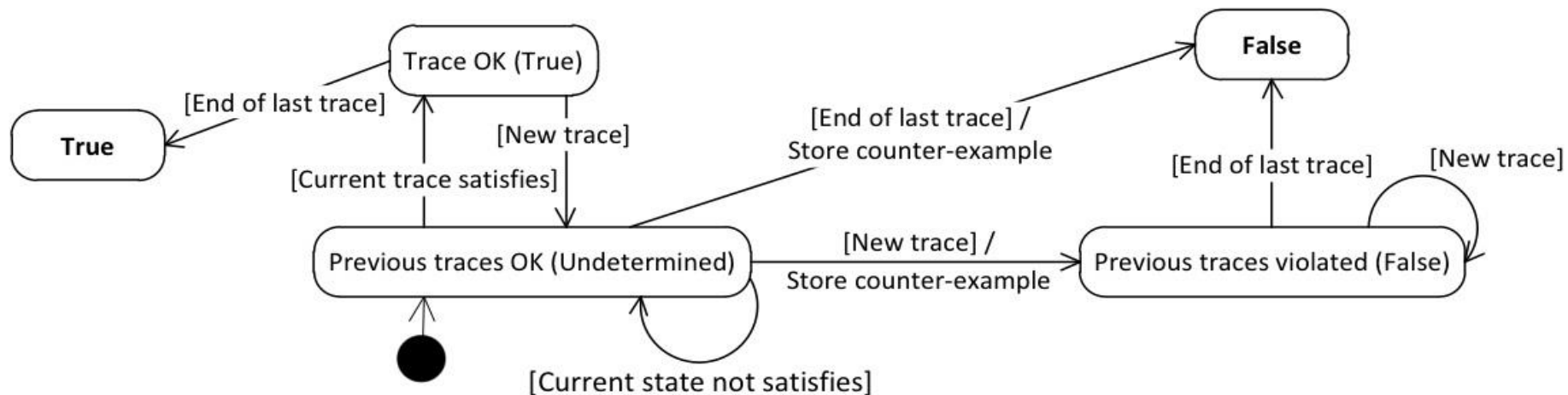


- Product FSM with monitor output:



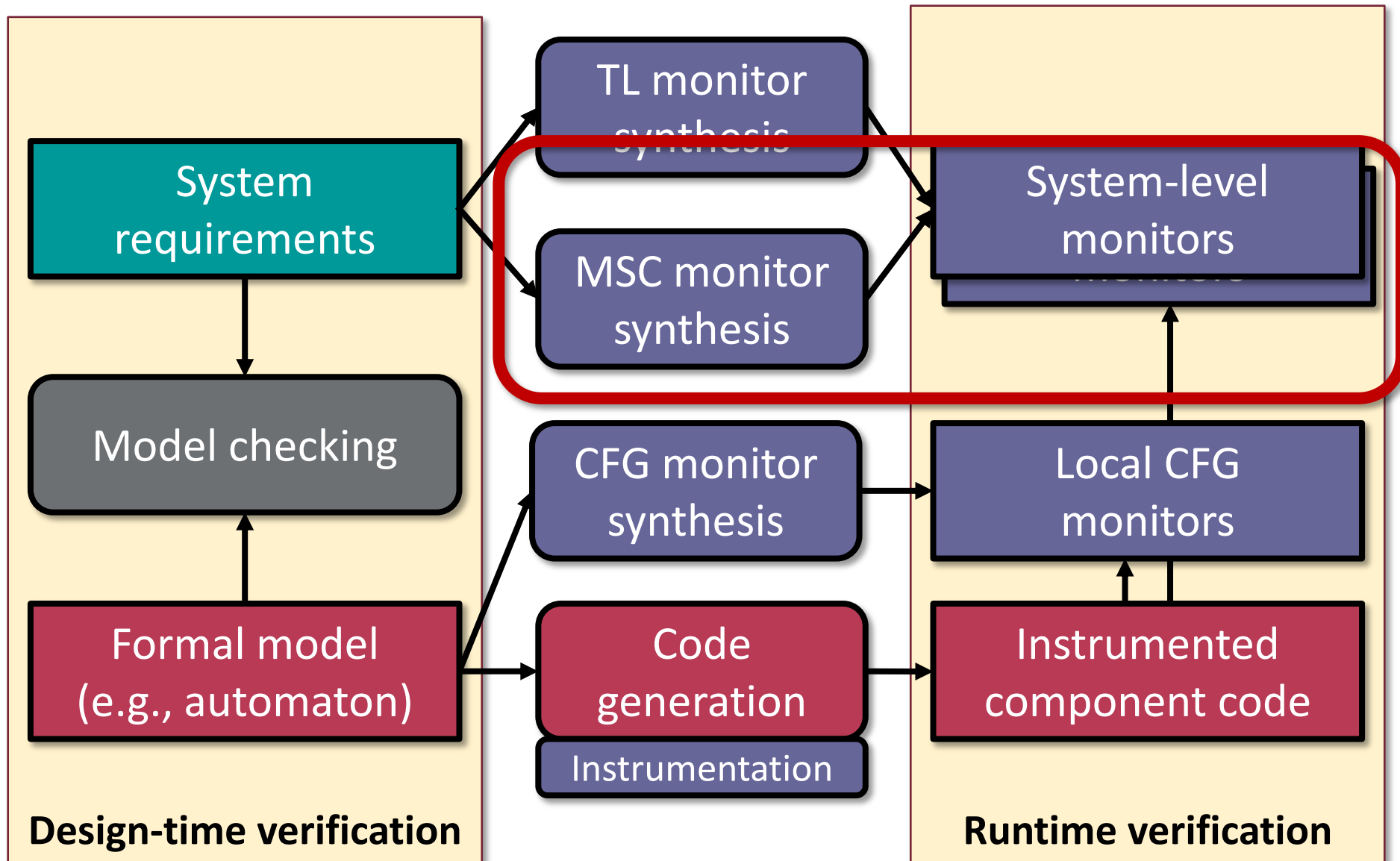
CTL based monitoring

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



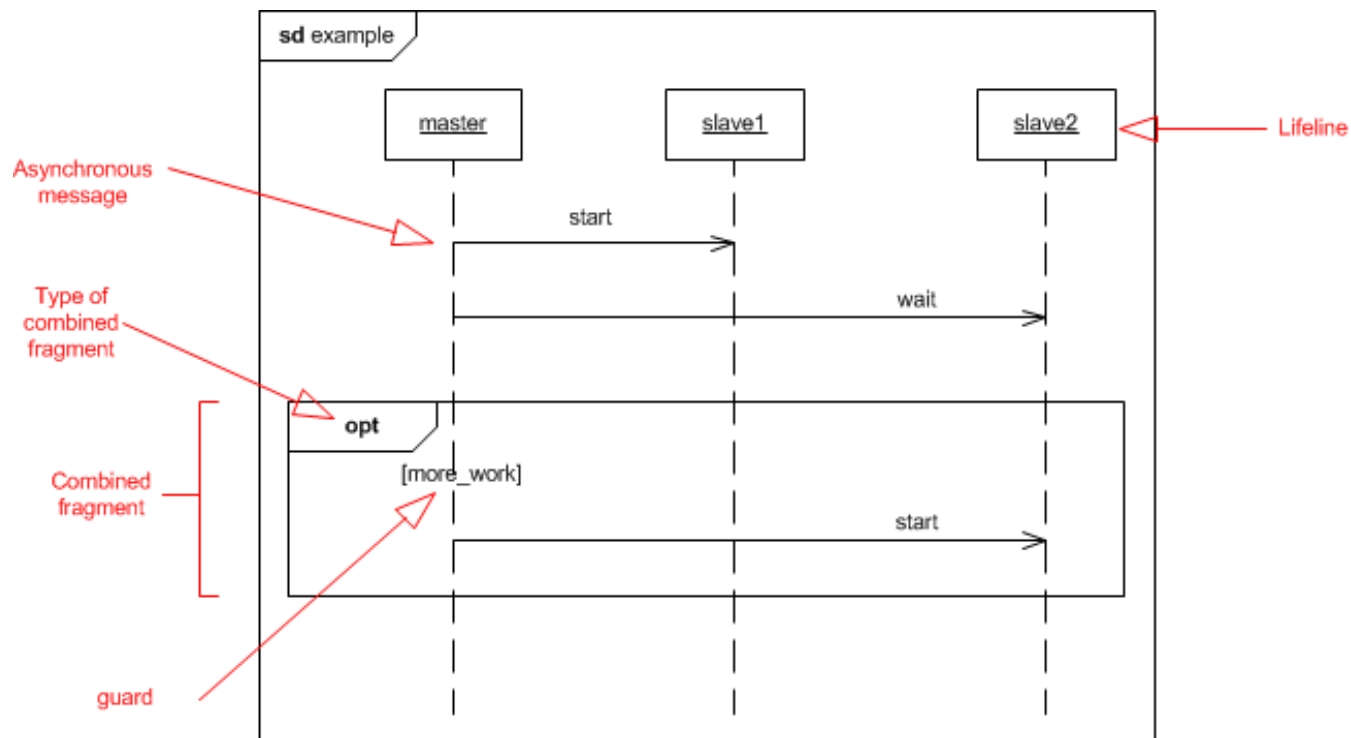
Runtime verification based on sequence diagrams

Overview: Runtime verification

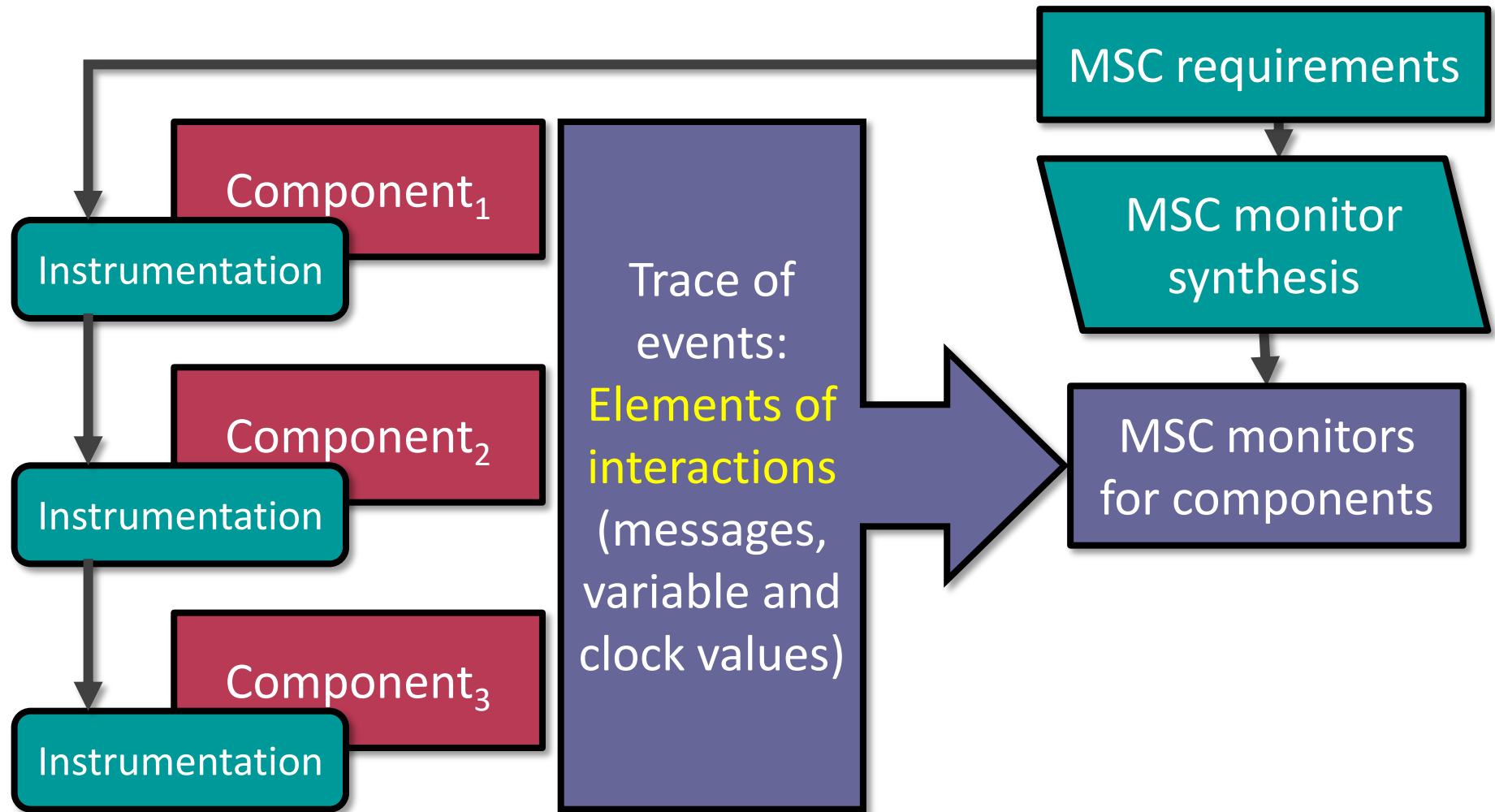


MSC based properties

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

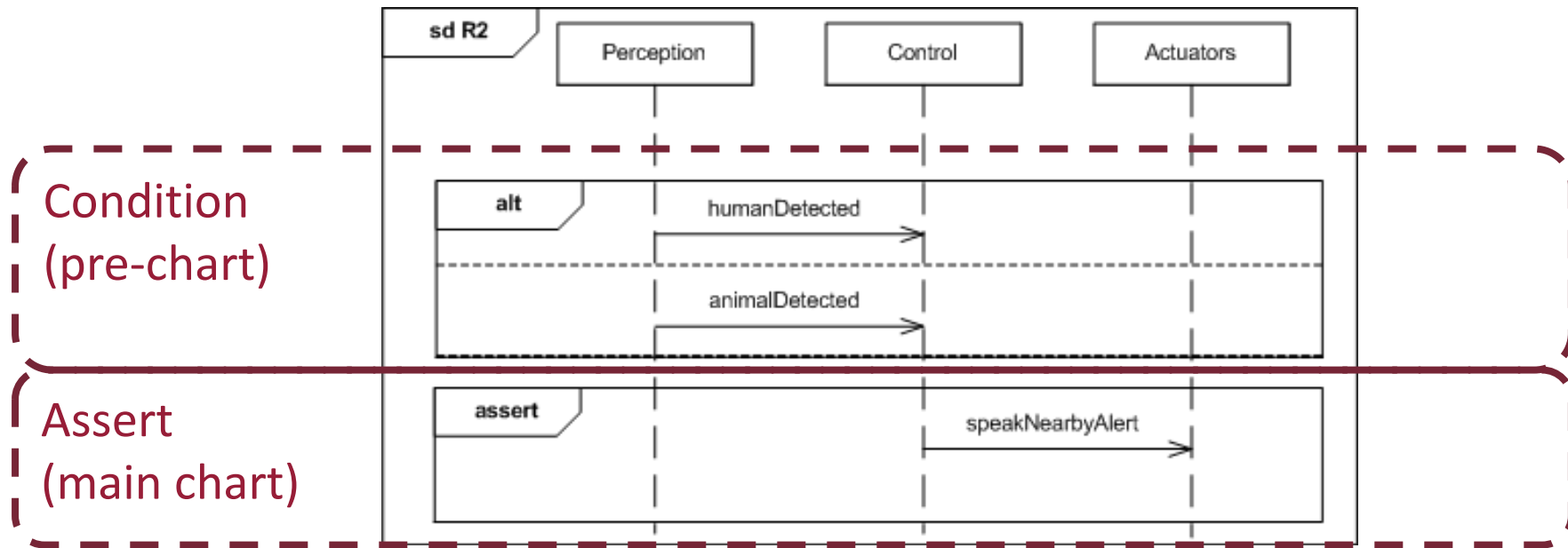


Setup of MSC based monitoring



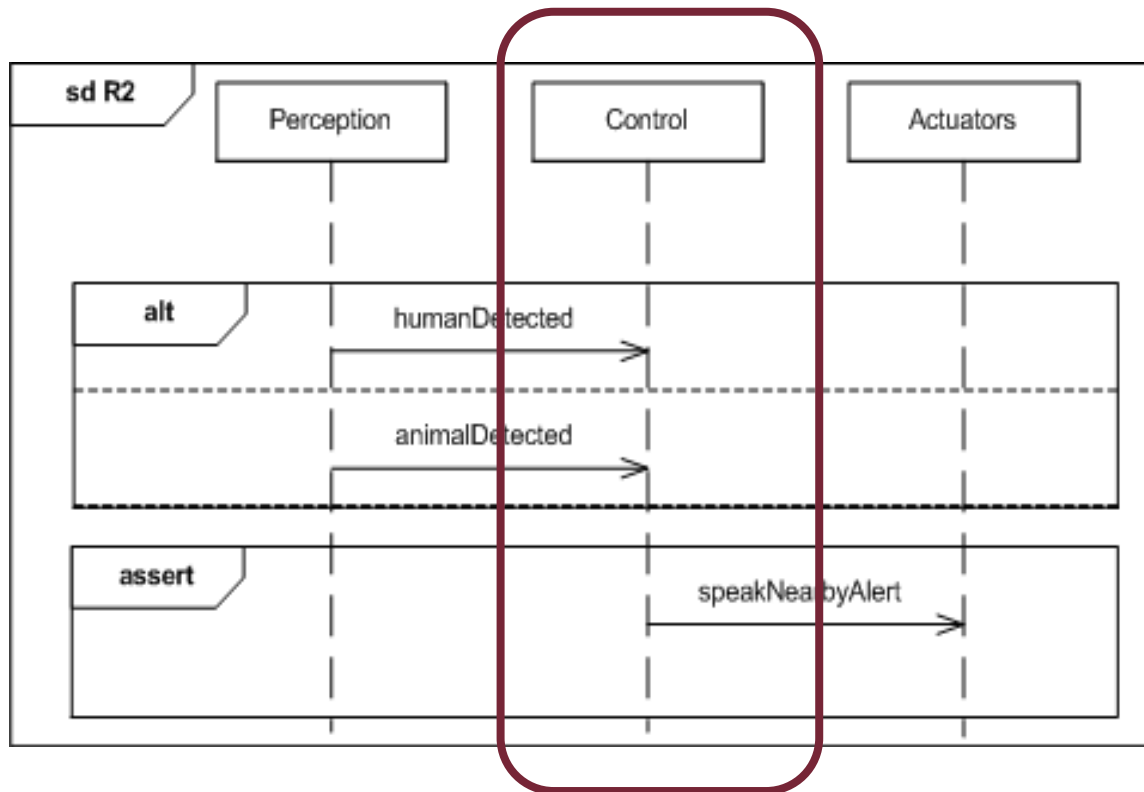
Restrictions and extensions

- 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

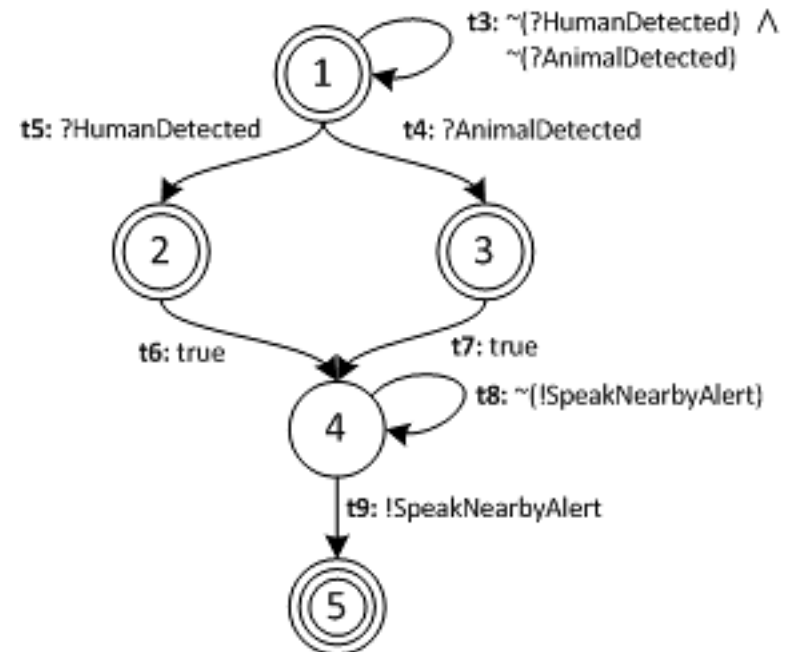
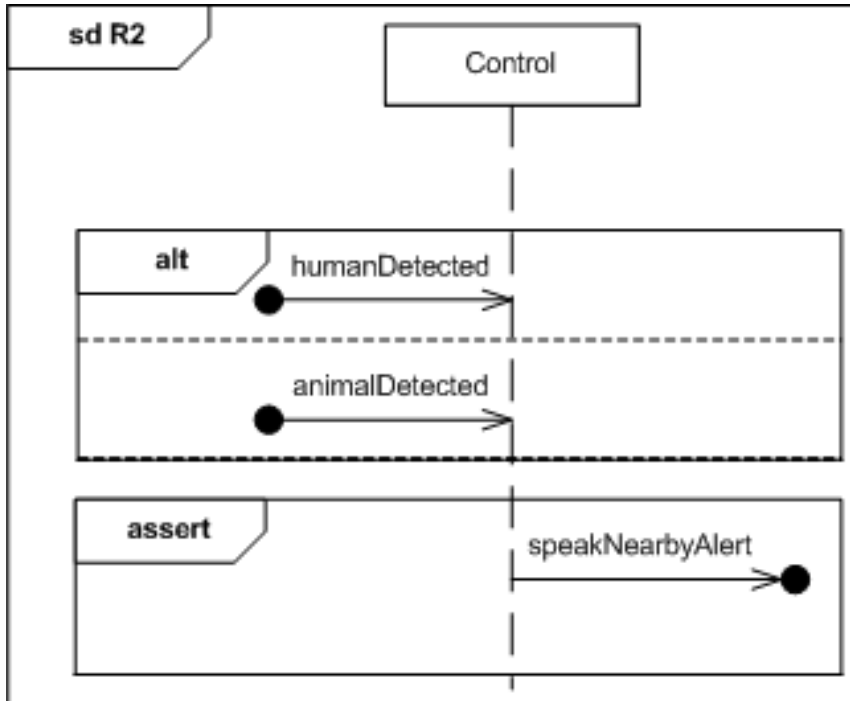
- Monitor constructed here: Observing a single lifeline (single component)



Monitoring on the basis of an MSC

- Observer automata constructed on the basis of the MSC lifeline

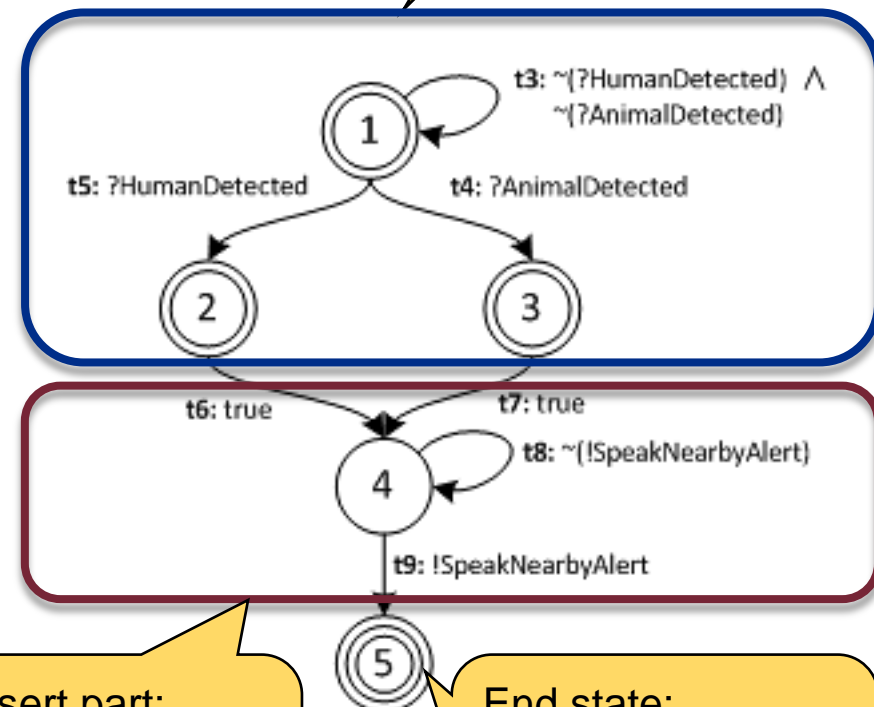
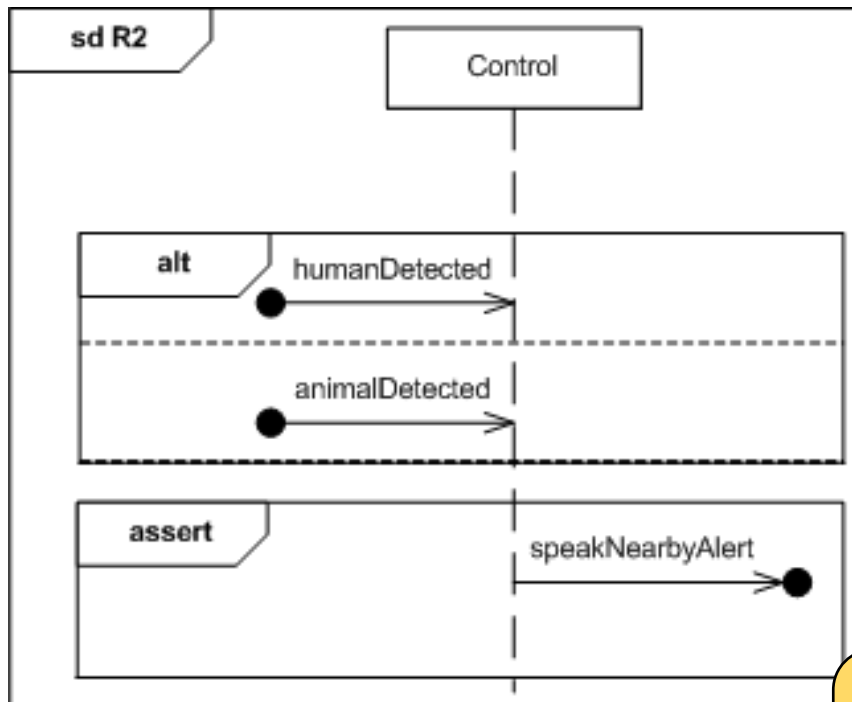
- Input events and messages, e.g., ?humanDetected
- Output actions and messages, e.g., !speakNearbyAlert



Role of condition and assert part

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

Condition part:
Not matching
means property
is not triggered

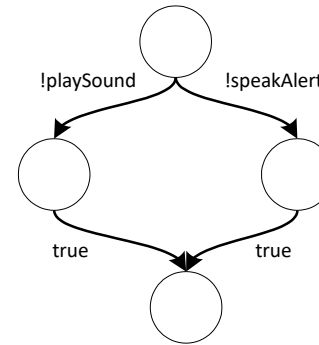
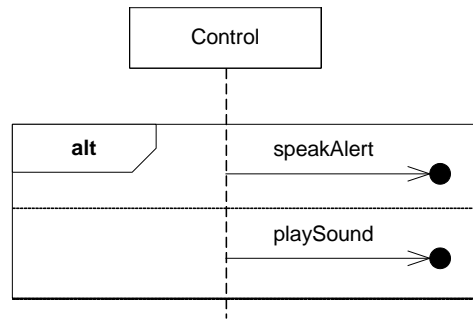


Assert part:
Not matching
means property
is violated

End state:
Reaching it means
that property is
satisfied

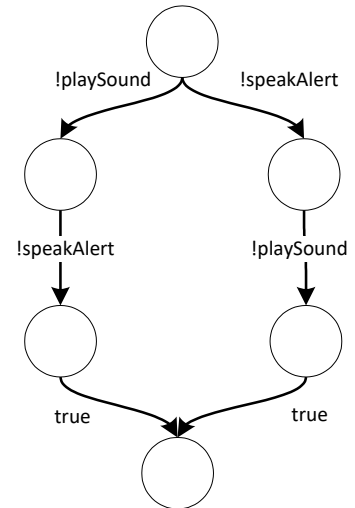
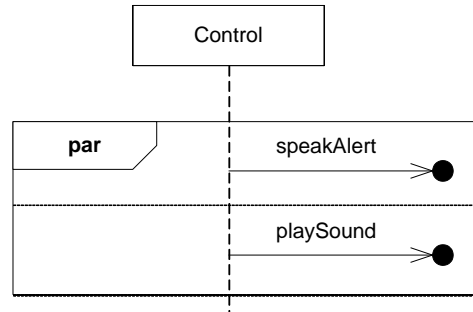
Basic patterns to construct the monitor

■ Alternative:

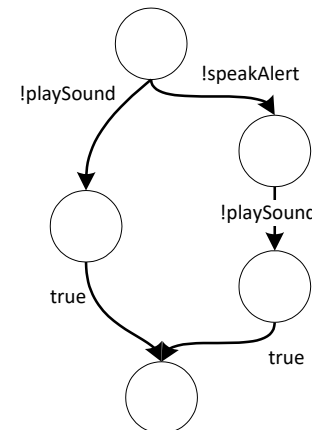
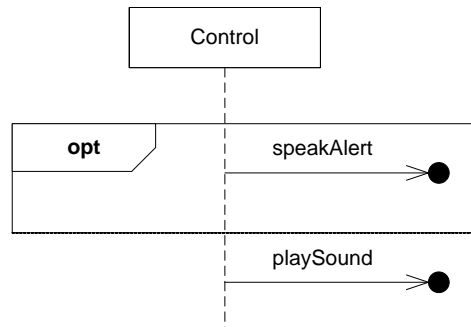


(Negative edges are omitted)

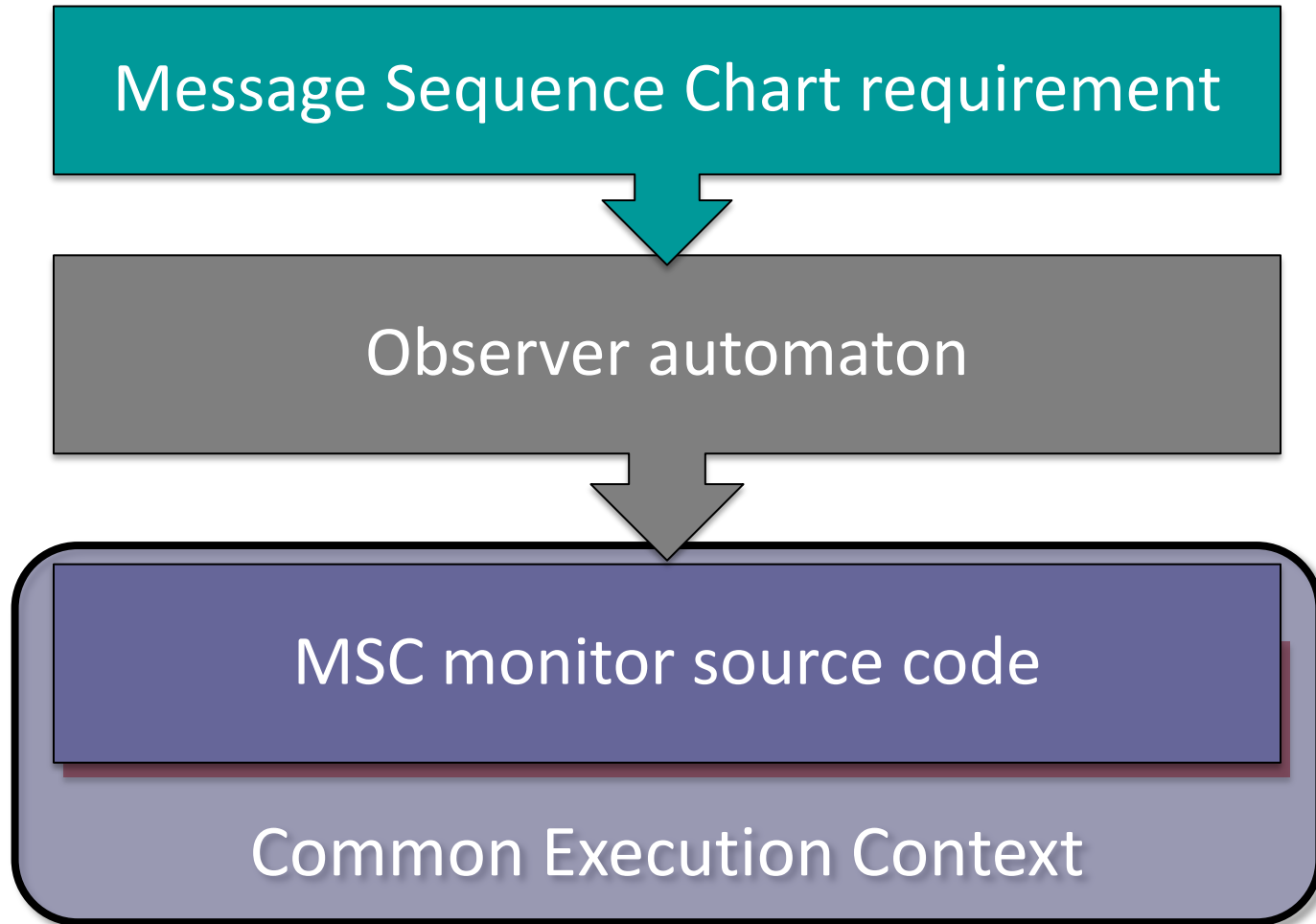
■ Parallel:



■ 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

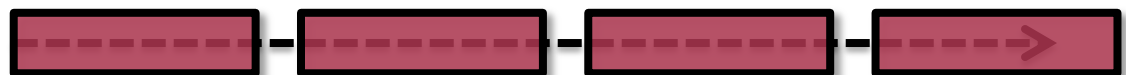
- Initial



- Invariant

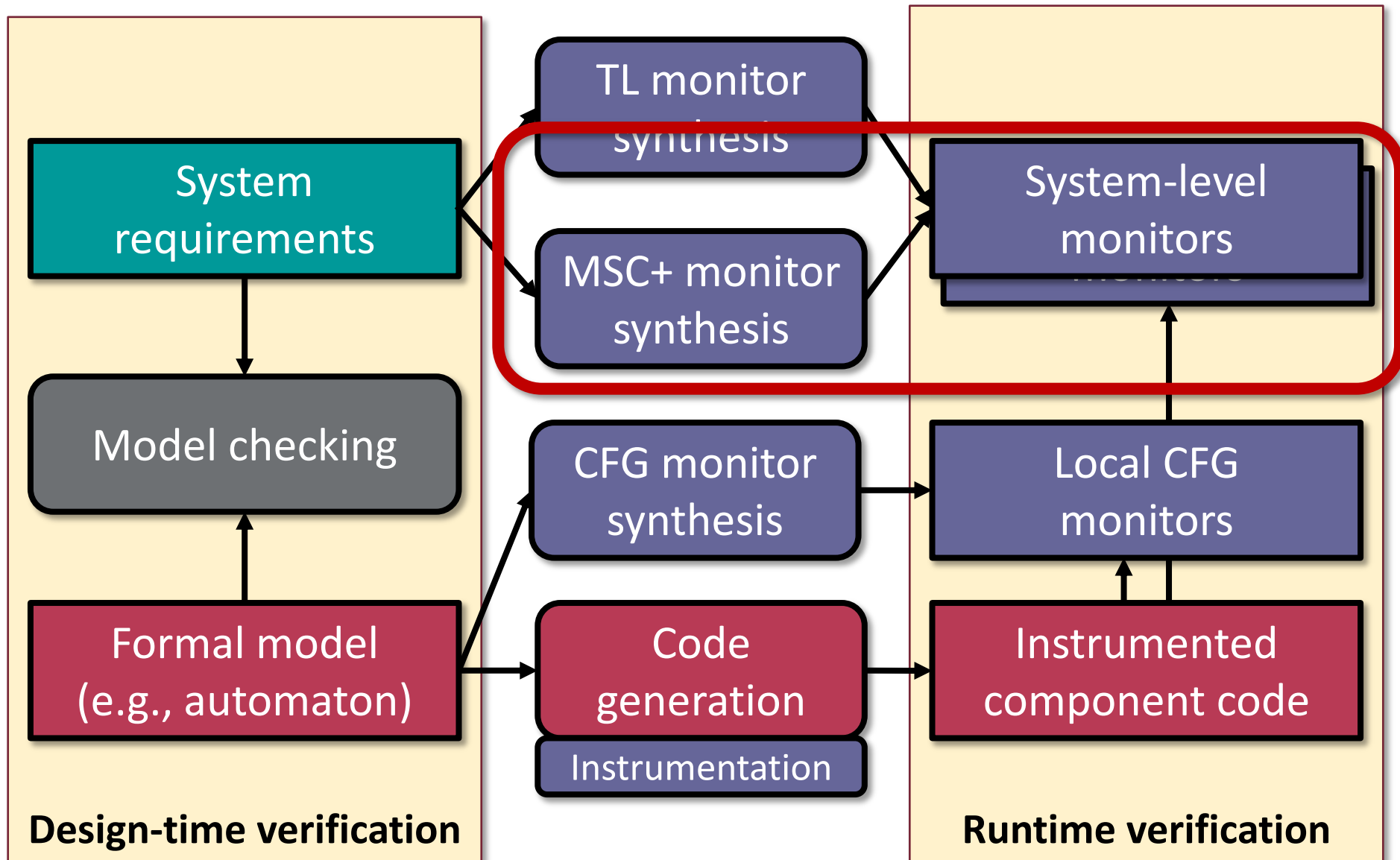


- Iterative



Runtime verification based on scenario and context description

Overview: Runtime verification



New challenges

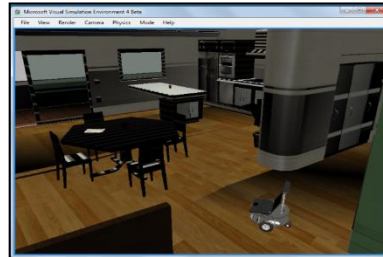
- Monitoring autonomous systems
 - **Context-aware** behaviour (perceived environment)
 - Adaptation to changing context (decisions, strategy)
- Specification of requirements: **Scenarios**
 - Behaviour: Sequences of **events / actions** with condition (pre-chart) and assertion (main chart)
 - Including 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

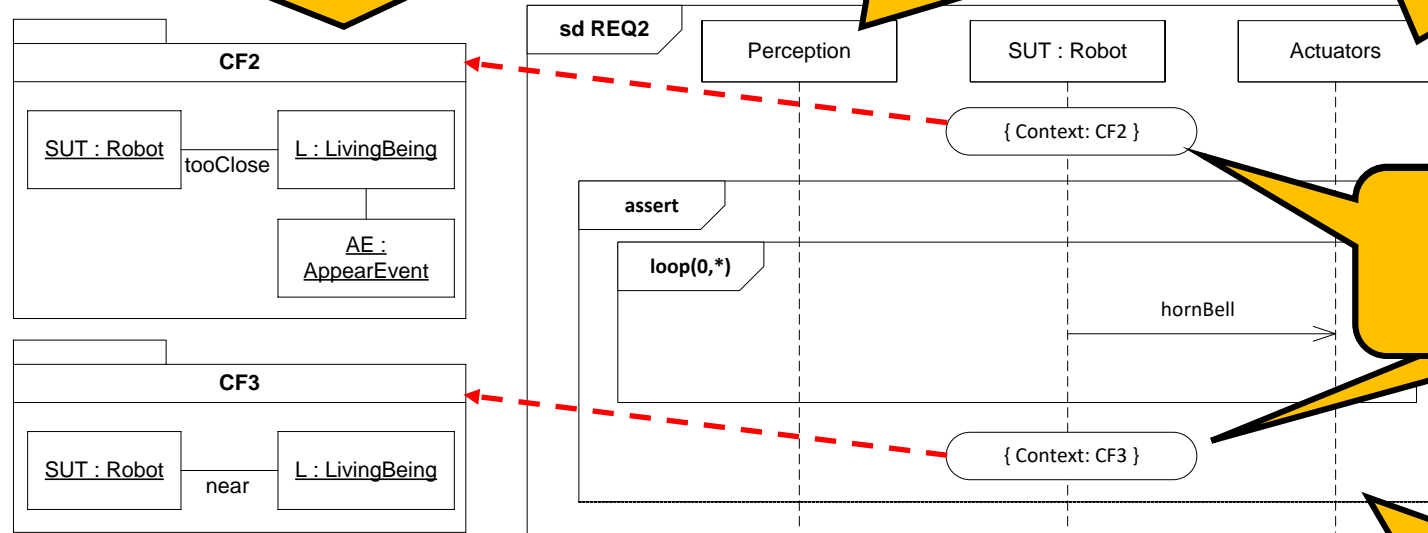
Formalization of requirements

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

Context of the SUT (objects and relations) at a given point

Events coming from the sensors

Actions sent to the actuators



Reference to a context fragment

Mandatory behavior

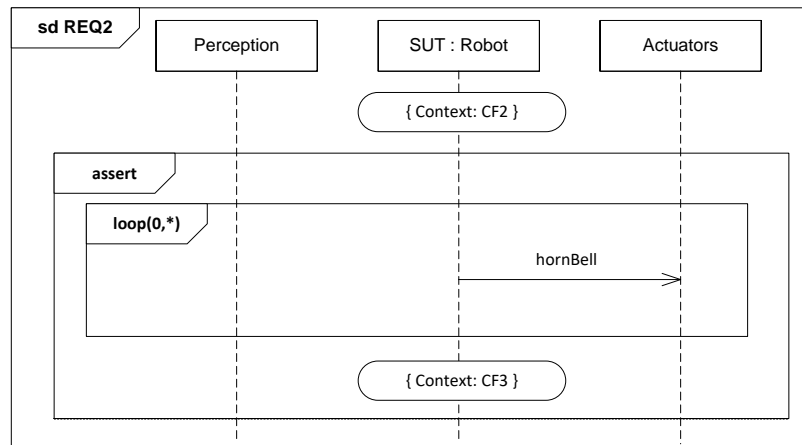
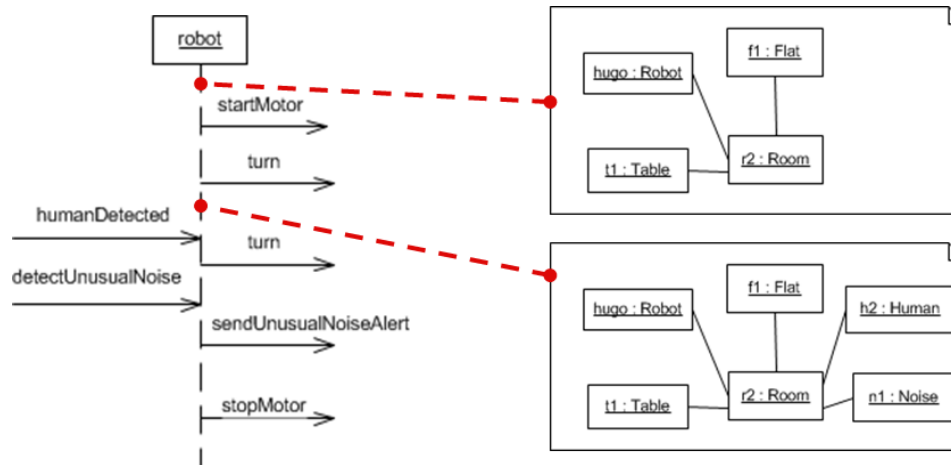
Context view
(context fragments)

Scenario view
(events and actions as messages)

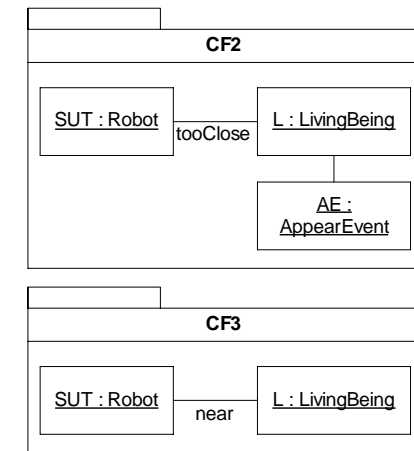
Tasks of the monitor

Observed trace:

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



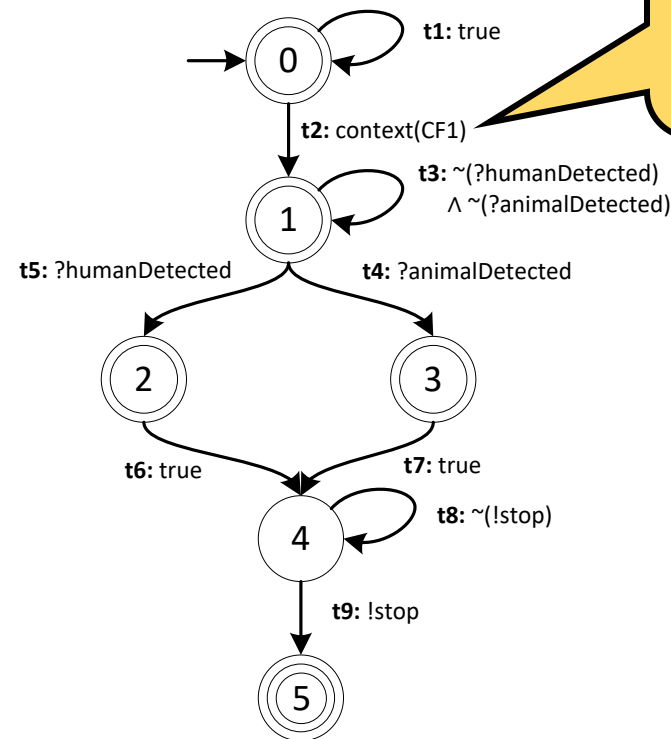
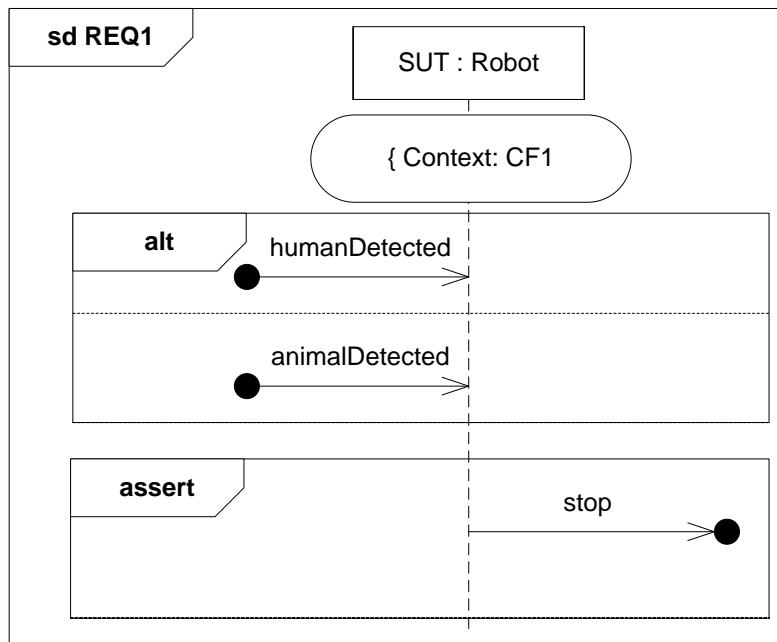
Matching messages:
Observer automaton



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

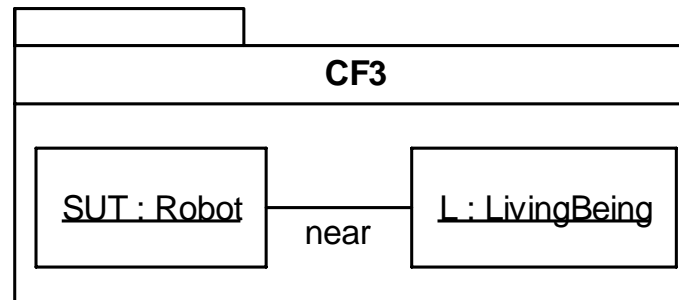


Context matching is included in checking

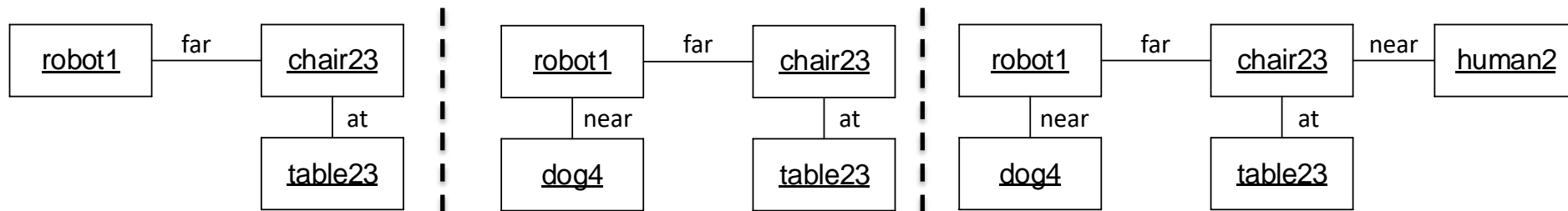
Context matching as graph matching

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

Context fragment
(in requirement):



Observed trace (with abstract relations):



Handling abstract relations

- Peculiarities in requirement properties

- Abstract relations (e.g., “near”)

- Hierarchy of objects (e.g., “dog” is a “living being”)

- Handling peculiarities in the monitor



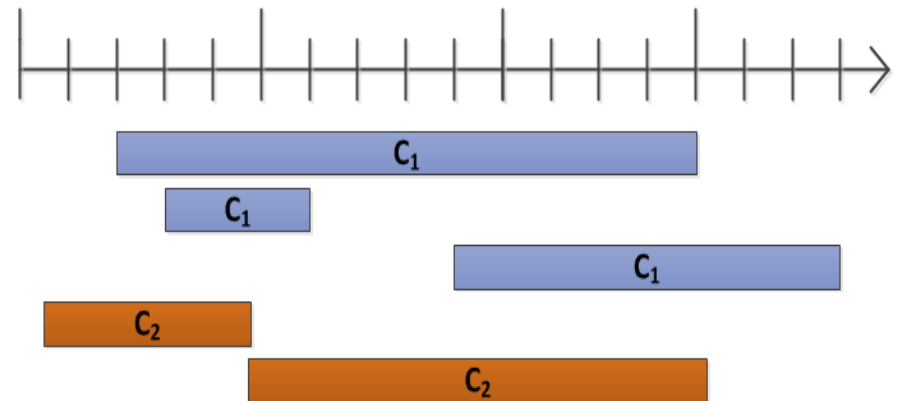
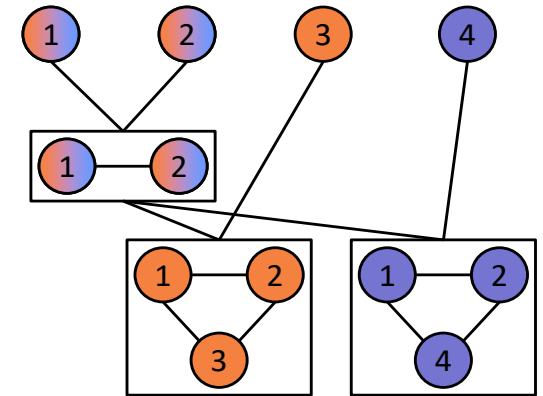
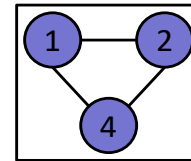
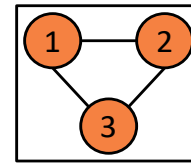
- Preprocessing the trace to derive abstract relations



- Using compatibility relation when matching context elements

Specific problems of graph matching

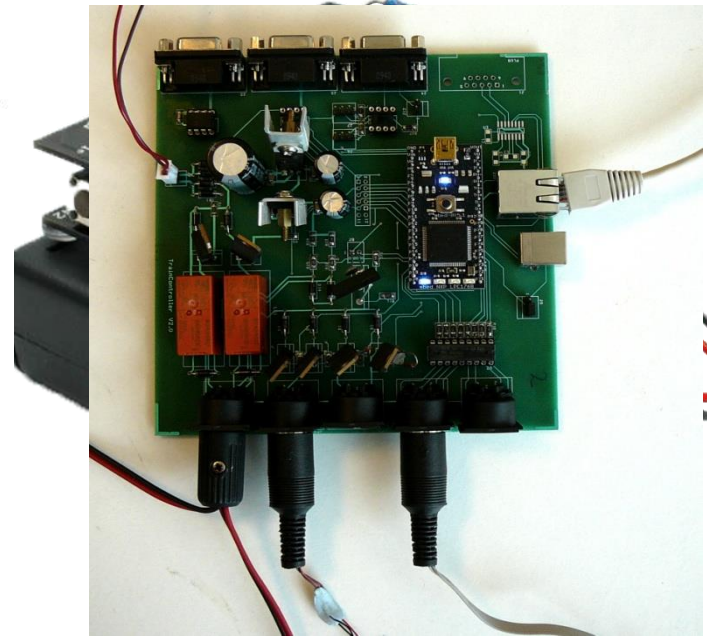
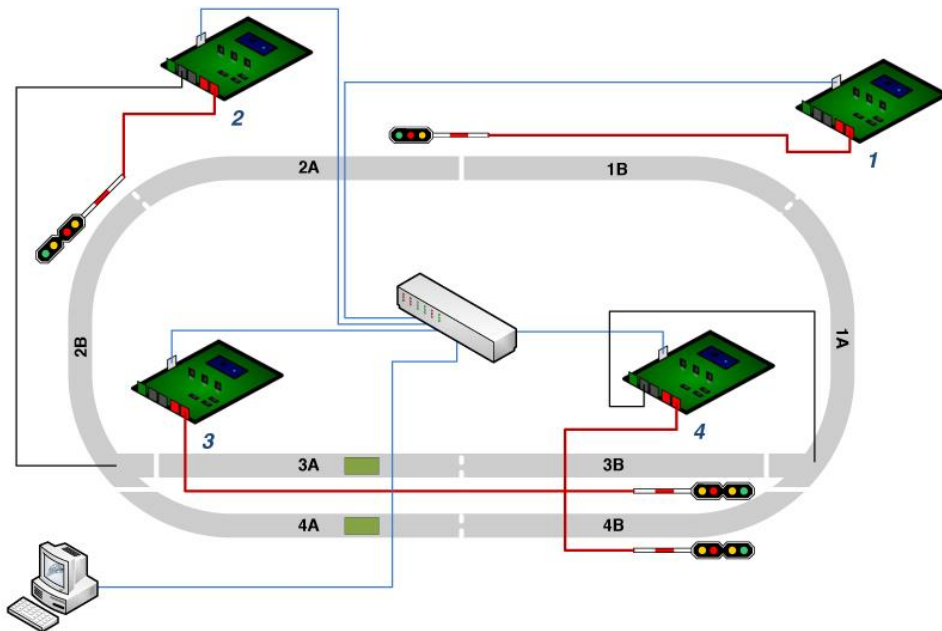
- Matching **all requirement scenarios** to a trace
 - Decomposition of the context fragments to store and match common parts only once
- Matching context fragments of requirements **at each step** of the trace
 - Concurrent threads of monitors (evaluations) are started when matchings are detected



Implementation experience

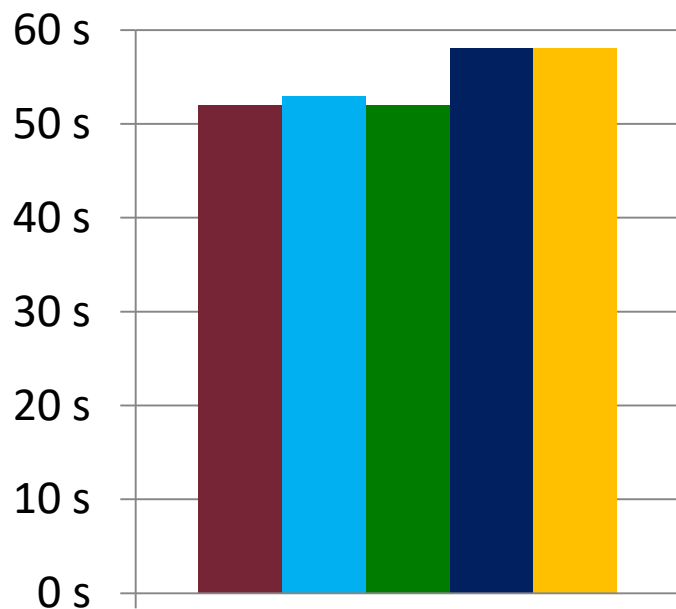
Implementation of TL and LSC monitoring

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



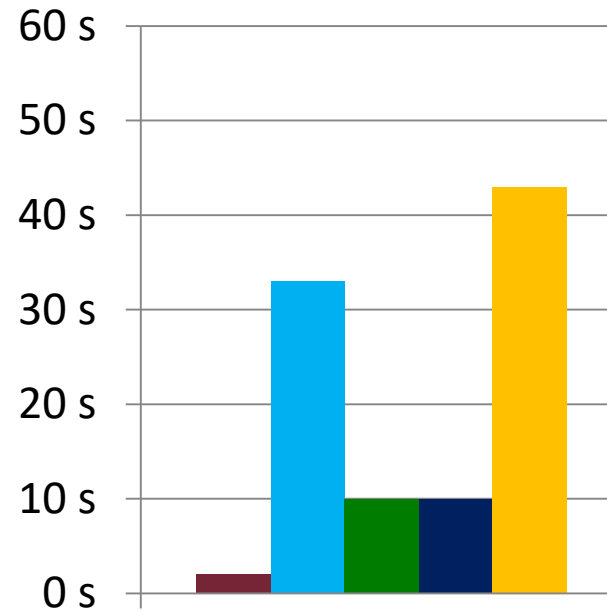
Time overhead

- Execution time on the mbed platform



With communication and
control functions
(50.000 state changes)

Complex control functions:
Less than 12% overhead

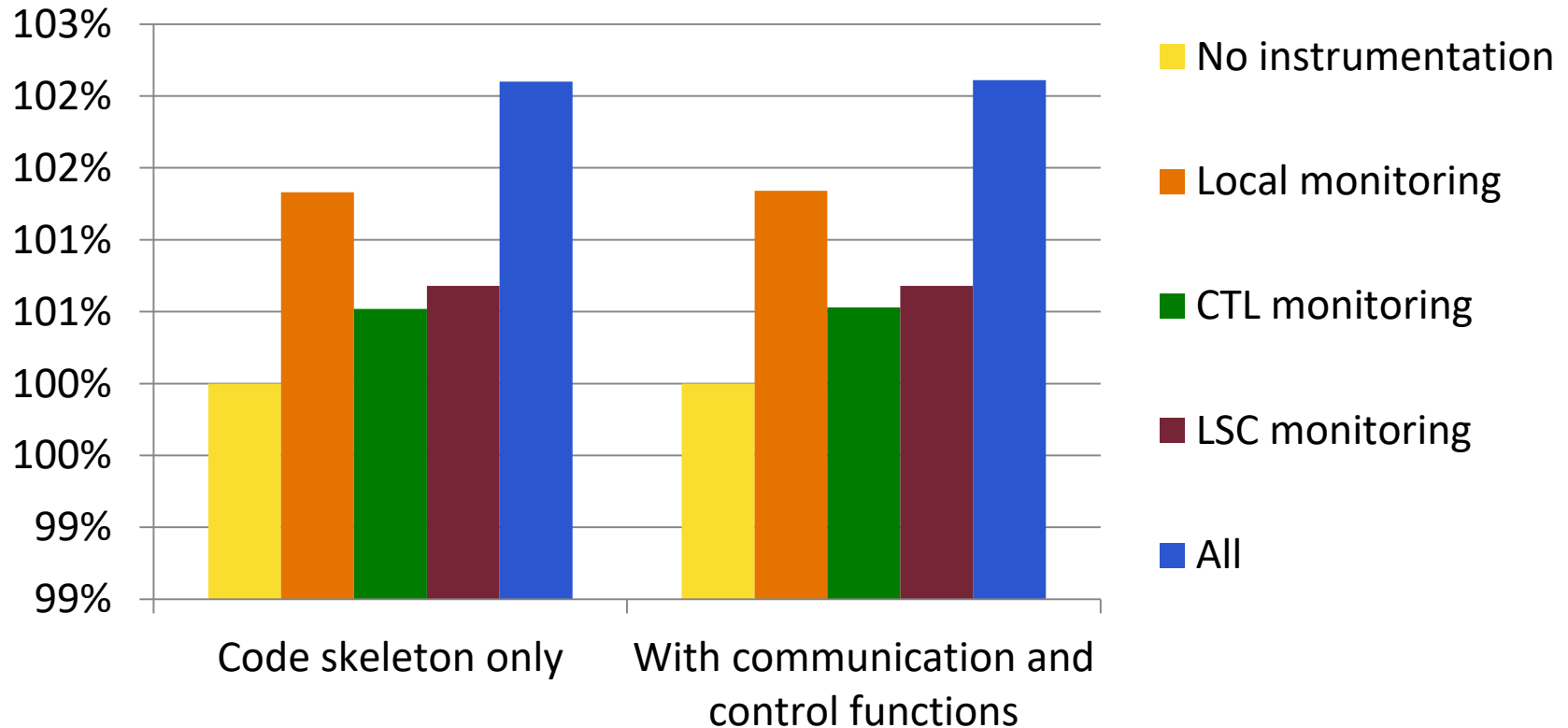


Code skeleton only
(500.000 state changes)

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 UML2 (Eclipse)
 - 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

