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

<table>
<thead>
<tr>
<th>Languages</th>
<th>FSM</th>
<th>ERE</th>
<th>CFG</th>
<th>PTLTL</th>
<th>LTL</th>
<th>PTCaRet</th>
<th>SRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaMOP</td>
<td>JavaFSM</td>
<td>JavaERE</td>
<td>JavaCFG</td>
<td>JavaPTLTL</td>
<td>JavaLTL</td>
<td>JavaPTCaRet</td>
<td>JavaSRS</td>
</tr>
<tr>
<td>BusMOP</td>
<td>BusFSM</td>
<td>BusERE</td>
<td>…</td>
<td>BusPTLTL</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>ROSMOP</td>
<td>ROSFSM</td>
<td>…</td>
<td>ROSCFG</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

- **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

- System requirements
- Model checking
- Formal model (e.g., automaton)
- Code generation
- Component source code

Design-time verification
Overview: Runtime verification

Design-time verification

System requirements

Model checking

Formal model (e.g., automaton)

Runtime verification

System-level monitors

Local CFG monitors

Instrumented component code

System-level monitors

MSC monitor synthesis

CFG monitor synthesis

TL monitor synthesis

Instrumentation

Code generation
Runtime verification based on reference automata
Overview: Runtime verification

System requirements

Model checking

Formal model (e.g., automaton)

Design-time verification

TL monitor synthesis

MSC monitor synthesis

System-level monitors

Local CFG monitors

Instrumented component code

Runtime verification

Code generation

Instrumentation
Monitoring on the basis of reference automaton

**Reference automaton**

**Monitor component**

- Interaction (events, messages)
- Internal behavior

- Automated instrumentation: signatures identifying the states

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

Basis for code generation

Reference automaton
Instrumentation in the generated source code

Automaton-level

- Initialization

  + Send state signature to monitor
    - Initializing state variables
    - Listening for events to react

Optional: Invalidate state signature

State-level

- Entering function
- Waiting function
- Exit function
- Instrumentation

System loop:
- Waiting function
- Instrumentation
- Exit function
- Send state signature to monitor
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: Monitoring on the basis of statecharts

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

Firing of a transition

- "Starting transition" signature
- Leaving source state configuration
- Execution of actions
- Entering target state configuration
- "Transition finished" signature
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 programming

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

Transitions Context
- Entering and leaving states
- Related actions
Case study: Monitoring on the basis of statecharts

- Reference for RTC Context

### Diagram

- **Uninitialized**
  - InitStarting
  - InitEntry [ieOK]
  - InitFinishing[ifOK]

- **Initialization**
  - EvtProcStarting [epsOK]
  - EvtProcFinishing [epfOK]

- **Stable**
  - TrStarting [tsOK] / createTrCtx

- **Transient**
  - Dispatch
Case study: Monitoring on the basis of statecharts

- **Reference for RTC Context**

  - Invalid condition for the step according to the statechart semantics

**Diagram:**
- States:
  - Uninitialized
  - Initialization
  - Stable
  - Transient

- Transitions:
  - initStarting
  - initEntry
  - initFinishing
  - evtProcStarting
  - evtProcFinishing
  - trStarting
  - dispatch

**Context:**
- Fault detected
- Reference for RTC Context
Case study: Monitoring on the basis of statecharts

- **Reference for Transition Context**

- **Exiting states**
  - exitState [xsOK] / markInactive
  - trAssociated [taOK]

- **Entering states**
  - enterState [esOK] / markActive
  - trFinishing [tfOK]
Case study: Monitoring on the basis of statecharts

- Reference for Transition Context

- Invalid condition for the step according to the statechart semantics
Runtime verification based on temporal logic properties
Overview: Runtime verification

- **System requirements**
- **Model checking**
  - **Formal model** (e.g., automaton)
- **Design-time verification**
  - **Code generation**
  - **Instrumentation**
- **Runtime verification**
  - **System-level monitors**
  - **Local CFG monitors**
  - **Instrumented component code**
  - **TL monitor synthesis**
  - **MSC monitor synthesis**
  - **CFG monitor synthesis**
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

Component$_1$
Instrumentation

Component$_2$
Instrumentation

Component$_3$
Instrumentation

Observed information:
Trace of signatures:
- elements of atomic propositions
  (states, events)

TL requirements

TL monitor synthesis

TL monitors
Monitoring LTL expressions

- Preprocessing: Normal form of expressions
  - Only $\land$, $\neg$, $X$ 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: $F \ p = \text{true} \ U \ p$, $G \ P = \neg (F \neg p)$

- Separating two parts of the expressions:
  - Present-time part: Boolean expressions of atomic propositions
  - Next-time part: Expression after an $X$ operator
  - Basic rule: $P \ U \ Q = Q \lor (P \land X (P \ U \ Q))$
    - present
    - 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: \( P \cup Q = Q \lor (P \land X(P \cup Q)) \)

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

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

Next-time expression to be evaluated later
Evaluating a trace of atomic propositions

- Checking of $P \cup Q = Q \lor (P \land X(P \cup Q))$
- 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:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a \land_3 b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>${\top, \bot, ?}$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>${\top, \bot, ?}$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$? ?$</td>
<td>${\top, ?}$</td>
<td>$?$</td>
</tr>
<tr>
<td>${\top, ?}$</td>
<td>$?$</td>
<td>$?$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$a$</th>
<th>$\neg_3 a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$?$</td>
<td>$?$</td>
</tr>
</tbody>
</table>
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

- **System requirements**
- **Model checking**
- **Formal model (e.g., automaton)**

**Design-time verification**

- **Model checking**
  - **Formal model (e.g., automaton)**
  - **Code generation**
    - **Instrumentation**

**Runtime verification**

- **TL monitor synthesis**
- **MSC monitor synthesis**
- **CFG monitor synthesis**
- **Local CFG monitors**
- **System-level monitors**

**System-level monitors**

**Instrumented component code**
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

- Component\(_1\)
  - Instrumentation
- Component\(_2\)
  - Instrumentation
- Component\(_3\)
  - Instrumentation

Trace of signatures: **Elements of interactions** (messages, events, variable and clock values)

MSC requirements

MSC monitor synthesis

MSC monitors for components
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

Diagram:

- Control flowchart with transitions:
  - `alt` with outputs: `humanDetected`, `animalDetected`
  - `assert` with output: `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 property is satisfied.
Basic patterns to construct the monitor

- **Alternative:**

- **Parallel:**

- **Optional:**
Steps of monitor synthesis

Message Sequence Chart requirement

Observer automaton

MSC monitor source code

Common Execution Context
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

System requirements

Model checking

Formal model (e.g., automaton)

Design-time verification

System-level monitors

MSC+ monitor synthesis

Local CFG monitors

Instrumented component code

Runtime verification

Code generation

Instrumentation

CFG monitor synthesis
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

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

- **Real environment**
- **Simulator**

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

**Requirements**

**Runtime traces**

**Trace evaluation by monitors**
Formalization of requirements

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

![Diagram showing context and scenario views]

- Context view (context fragments)
- Scenario view (events and actions as messages)

- 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**

- CF2
  - SUT: Robot
    - tooClose
  - L: LivingBeing
  - AE: AppearEvent

- CF3
  - SUT: Robot
    - near
  - L: LivingBeing

**Scenario view**

- sd REQ2
- Perception
- SUT: Robot
- Actuators

- assert
- loop(0,*)
- hornBell

- { Context: CF2 }
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

Alt scenarios

sd REQ1

alt

humanDetected

animalDetected

assert

stop

SUT : Robot

{ Context: CF1

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

**With communication and control functions**

- **(50,000 state changes)**

**Complex control functions:**

- Less than 12% overhead

**Simple control functions:**

- Larger overhead can be expected

---

**Code skeleton only**

- **(500,000 state changes)**

**No instrumentation**

- No instrumentation

**Local monitoring**

- Local monitoring

**CTL monitoring**

- CTL monitoring

**LSC monitoring**

- LSC monitoring

**All**

- All
Code (memory) overhead

- Code size on the mbed platform

<table>
<thead>
<tr>
<th></th>
<th>Code skeleton only</th>
<th>With communication and control functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>No instrumentation</td>
<td>99%</td>
<td>102%</td>
</tr>
<tr>
<td>Local monitoring</td>
<td>100%</td>
<td>102%</td>
</tr>
<tr>
<td>CTL monitoring</td>
<td>101%</td>
<td>101%</td>
</tr>
<tr>
<td>LSC monitoring</td>
<td>102%</td>
<td>102%</td>
</tr>
<tr>
<td>All</td>
<td>103%</td>
<td>103%</td>
</tr>
</tbody>
</table>

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: $O(NIM^2M^2)$
    - $N$: 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

Formalized requirements

- Temporal logic (LTL, TCTL)
- Message Seq. Charts (MSC)
- MSC + context scenarios

Monitor synthesis

- TL monitors
- MSC monitors
- Scenario monitors