Verifying and Monitoring UML Models with Observer Automata
A Transformation-free Approach

ACM/IEEE 22\textsuperscript{th} International Conference on Model Driven Engineering Languages and Systems (MODELS’19) in Munich, Germany

Valentin BESNARD \textsuperscript{1} Ciprian TEODOROV \textsuperscript{2} Frédéric JOUAULT \textsuperscript{1}
Matthias BRUN \textsuperscript{1} Philippe DHAUSSY \textsuperscript{2}

\textsuperscript{1} ERIS, ESEO-TECH, Angers, France
\textsuperscript{2} Lab-STICC UMR CNRS 6285, ENSTA Bretagne, Brest, France

This work has been partially funded by Davidson.
# Table of Contents

1. Introduction

2. Illustrating Example

3. Expressing Properties as UML Observer Automata

4. Monitoring Activities

5. Application to the Illustrating Example

6. Conclusion
Introduction

Context

Observations

- Increasing complexity and connectivity of embedded systems
  - Increasing exposure to potential software failures
  - Increasing difficulty to detect, understand, and fix software failures

Need for V&V at all design stages
- Testing or proving that a system satisfies its expected properties
  - Possibly relying on environment abstractions (inputs to consider and execution platform)

Need for runtime monitoring
- Detecting safety property violations at runtime (with the actual environment)
  - Making it possible to trigger safe system recovery procedures
Context

Observations
- Increasing complexity and connectivity of embedded systems
  ⇒ Increasing exposure to potential software failures
  ⇒ Increasing difficulty to detect, understand, and fix software failures

Need for V&V at all design stages
- Testing or proving that a system satisfies its expected properties
  - Possibly relying on environment abstractions
    (inputs to consider and execution platform)
Introduction

Context

Observations
- Increasing complexity and connectivity of embedded systems
  ⇒ Increasing exposure to potential software failures
  ⇒ Increasing difficulty to detect, understand, and fix software failures

Need for V&V at all design stages
- Testing or proving that a system satisfies its expected properties
  - Possibly relying on environment abstractions
    (inputs to consider and execution platform)

Need for runtime monitoring
- Detecting safety property violations at runtime (with the actual environment)
- Making it possible to trigger safe system recovery procedures
Goal

Provide a technique to execute models on embedded targets with facilities to perform model-checking and runtime monitoring on these models.
Overview

Goal
Provide a technique to execute models on embedded targets with facilities to perform model-checking and runtime monitoring on these models

Our previous work [Besnard et al., MODELS 2018]
1. Identified problems on classical model-checking approaches
2. Introduced a solution based on a model interpreter
Goal

Provide a technique to execute models on embedded targets with facilities to perform model-checking and runtime monitoring on these models.

Our previous work [Besnard et al., MODELS 2018]

1. Identified problems on classical model-checking approaches
2. Introduced a solution based on a model interpreter

In this work

3. Identify problems on classical monitoring approaches
4. Can we address these problems with the model interpreter approach?
(1) Classical Approach with Model-checking

- Requirements Specification
  - expression in a formal language
- Formal Properties
- Design Model
  - system modeling in a design language
(1) Classical Approach with Model-checking

- Requirements Specification
  - system modeling in a design language
  - expression in a formal language

- Design Model
  - model transformation

- Formal Properties

- Models for Analysis

- Model-checker
(1) Classical Approach with Model-checking

- Requirements Specification

  - Formal Properties
    - expression in a formal language

  - Design Model
    - system modeling in a design language

  - Models for Analysis
    - model transformation

- Model-checker

  - Execution Environment
    - Environment
      - Code
        - I/O
  
  - Code generation
(1) Classical Approach with Model-checking (Problems)

Problems: Two semantic gaps and an equivalence problem caused by transformations of the design model into different languages
(2) Our Approach with Model-checking [Besnard et al., MODELS 2018]

- Requirements Specification
  - expression in a formal language
  - system modeling in a design language
  - Design Model
  - Formal Properties
(2) Our Approach with Model-checking [Besnard et al., MODELS 2018]
A unique definition of the language semantics for verification activities and model execution
(3) Classical Approach with Monitoring

*Diagram*

- **Requirements Specification**
  - System modeling in a design language
  - Expression as observer automata
  - Expression in a formal language
  - Formal Properties
  - Transformation

- **Design Model**
  - Observes
  - Monitors Model
(3) Classical Approach with Monitoring

Introduction

Valentin BESNARD (ESEO-TECH)

MODELS'19

September 19th, 2019 8 / 31
(3) Classical Approach with Monitoring (Problems)

1. Semantic gap between monitors model and monitors code
(3) Classical Approach with Monitoring (Problems)

1. Semantic gap between monitors model and monitors code
2. Languages used to express monitors and design models are usually different
(3) Classical Approach with Monitoring (Problems)

- Semantic gap between monitors model and monitors code
- Languages used to express monitors and design models are usually different
(4) Our Approach with Monitoring

- System modeling in a design language
- Expression of observer automata in the same design language
- Expression in a formal language
- Formal properties
- Transformation

Diagram:
- Requirements Specification
- Design Model
- Monitors Model
(4) Our Approach with Monitoring

- **Requirements Specification**
  - System modeling in a design language
  - Expression of observer automata in the same design language
  - Expression in a formal language
  - Formal Properties
  - Transformation

- **Environment**
  - Interprets

- **Design Model**
  - Interprets

- **Monitors Model**
  - Interprets

- **Interpreter Component**
  - Execution Environment
(4) Our Approach with Monitoring

The same component interprets both design and monitors models:
- **1. No semantic gap**
- **2. Only one language to express system and monitors models**
Table of Contents

1 Introduction

2 Illustrating Example

3 Expressing Properties as UML Observer Automata

4 Monitoring Activities

5 Application to the Illustrating Example

6 Conclusion
Table of Contents

1 Introduction

2 Illustrating Example

3 Expressing Properties as UML Observer Automata

4 Monitoring Activities

5 Application to the Illustrating Example

6 Conclusion
Cruise Control Overview
Cruise Control Overview

DeployedSystem

ThrottlePedal
command
pedal

BrakePedal
pedal

ClutchPedal
pedal

Buttons
buttons

PhysicalEnvironment
environment

commandCCS

environment

speed

commandThrottle

CruiseControlSystem

CruiseControlInterface

speed
throttlePedal
cruiseSpeed
brakePedal
clutchPedal
buttons

ControlLoop

speed
cruiseSpeed
command
on/off

on/off
Cruise Control Overview
Cruise Control Interface Requirements

System requirements

1. After the detection of an event that turns the control loop off and until a contrary event is sent, the cruise control interface should not try to send new cruise speed setpoints.

2. The cruise speed setpoint should not be below 40 km/h or above 180 km/h.

3. When the system is engaged, the cruise speed setpoint should be defined.

Design model

Made using a UML subset that can be represented by:

- Class diagram
- Composite structure diagram
- State machines
# Table of Contents

1. Introduction

2. Illustrating Example

3. Expressing Properties as UML Observer Automata

4. Monitoring Activities

5. Application to the Illustrating Example

6. Conclusion
UML Observer Automata

Expressed directly in the design language

- UML class + UML state machine with *fail* states
- Extension of the expression language to read objects of the system and their properties

Requirements on observer automata

- Read-only access to system objects
- UML observer state machines must be:
  - **Deterministic** to avoid introducing non-determinism in the observed system execution
  - **Complete** to avoid blocking the system execution

Expressivity = safety properties (something bad happens)

- Analysis of finite execution traces for monitoring (current run)
- Verification problem reduced to a reachability problem (observer *fail* states)
Cruise control interface requirements

1. After the detection of an event that turns the control loop off and until a contrary event is sent, the cruise control interface should not try to send new cruise speed setpoints.
2. The cruise speed setpoint should not be below 40 km/h or above 180 km/h.
3. When the system is engaged, the cruise speed setpoint should be defined.
Cruise control interface requirements

1. After the detection of an event that turns the control loop off and until a contrary event is sent, the cruise control interface should not try to send new cruise speed setpoints.

2. The cruise speed setpoint should not be below 40 km/h or above 180 km/h.

3. When the system is engaged, the cruise speed setpoint should be defined.
Synchronous Composition

Principle

Each time a transition of the system model is fired, each observer automaton also makes a step to follow the system execution.

- At each step, a synchronous transition must be fired
- A synchronous transition is composed of:
  - One transition of the system
  - One transition per observer automaton
Synchronous Composition

Principle

Each time a transition of the system model is fired, each observer automaton also makes a step to follow the system execution.

- At each step, a synchronous transition must be fired
- A synchronous transition is composed of:
  - One transition of the system
  - One transition per observer automaton
- The UML semantics extension on which our approach relies
- Synchronous transitions are built on-the-fly for an efficient execution
Monitoring Activities

Runtime Monitoring with UML Observer Automata

Use the actual scheduling policy (e.g., round robin on active objects)
Use the execution sequencer that fires synchronous transitions in loop
Check the current state of each observer at each step

Valentin BESNARD (ESEO-TECH)
MODELS'19
September 19th, 2019 20 / 31
Use the actual scheduling policy (e.g., round robin on active objects)
Use the actual scheduling policy (e.g., round robin on active objects)
Use the execution sequencer that fires synchronous transitions in loop
Use the actual scheduling policy (e.g., round robin on active objects)

Use the execution sequencer that fires synchronous transitions in loop

Check the current state of each observer at each step
Additional Usage: Model-checking with UML Observer Automata

- Use an abstraction of the scheduling policy to explore the whole model state-space
- The model-checker only has to use a reachability algorithm
  - $[] ! \text{OBSERVER\_FAIL}(\text{obs})$
# Table of Contents

1. Introduction
2. Illustrating Example
3. Expressing Properties as UML Observer Automata
4. Monitoring Activities
5. Application to the Illustrating Example
6. Conclusion
model under verification = system model + abstract environment model
Experiments

- Compare verification results obtained with:
  - LTL formulae

\[\text{[Teodorov et al., 2017]} \quad \text{https://plug-obp.github.io/}\]
Experiments

- Compare verification results obtained with:
  - LTL formulae
  - UML observer automata

1[Teodorov et al., 2017] https://plug-obp.github.io/
Experiments

- Compare verification results obtained with:
  - LTL formulae
  - UML observer automata
- Use the same UML observer automata to make runtime monitoring

\[^1\text{Teodorov et al., 2017}]\ https://plug-obp.github.io/
Expression of properties as LTL formulae

1. \([\Box ((|evOffSent| \text{ and } !|evOnSent|) \rightarrow (!|evUpdateSetPointSent| \text{ W } |evOnSent|))]
2. \([\Box (|intervalCS| \text{ or } |unknownCS|)]
3. \([\Box (|ccsEngaged| \rightarrow !|unknownCS|)]

Expression of properties as UML observer automata

Observer1
- States: Disengaged, Engaged, Fail
- Transitions: [evOffSent] from Disengaged to Engaged, [evOnSent] from Engaged to Disengaged, [evUpdateSetPointSent] from Engaged to Fail

Observer2
- States: Running, Fail
- Transitions: [!(intervalCS || unknownCS)] from Running to Fail

Observer3
- States: Running, Fail
- Transitions: [ccsEngaged && unknownCS] from Running to Fail
Results - Model-checking

<table>
<thead>
<tr>
<th>LTL Formulae</th>
<th>UML Observer Automata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>✓</td>
</tr>
<tr>
<td>Property 2</td>
<td>✓</td>
</tr>
<tr>
<td>Property 3</td>
<td>✗</td>
</tr>
</tbody>
</table>

✓: Property verified  ❌: Property violated

Analysis of the counter-example

Events \textit{resetCS} and \textit{disengage} could be processed in any order
⇒ Bad event interleaving

Model state-space

46,444,386 configurations linked by 82,734,350 transitions
## Results - Monitoring

<table>
<thead>
<tr>
<th></th>
<th>Initial Model</th>
<th>Fixed Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>Property 2</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>Property 3</td>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

- **G**: No failure detected
- **G**: Failure detected

### Overhead of the monitoring infrastructure

- **Execution performance**: +6.5%
- **Memory footprint**: +1.2%
Results - Monitoring

<table>
<thead>
<tr>
<th>Property</th>
<th>Initial Model</th>
<th>Fixed Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Property 2</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Property 3</td>
<td>✗</td>
<td>✔️</td>
</tr>
</tbody>
</table>

- ✔️: No failure detected
- ✗: Failure detected

**Execution performance**

- Estimation of the overhead:
  
  \[
  overhead \approx 6.5 + \frac{1}{nb_{ao}} \sum_{i=1}^{N} \frac{nb_{states_i}}{nb_{outgoings_i}}
  \]

- Relative cost of observer automata decreases as the size of the system model increases.
Table of Contents

1. Introduction
2. Illustrating Example
3. Expressing Properties as UML Observer Automata
4. Monitoring Activities
5. Application to the Illustrating Example
6. Conclusion
Conclusion

Problems

1. Semantic gap between monitors model and monitors code
2. Languages used to express monitors and design models are usually different
Conclusion

Problems
1. Semantic gap between monitors model and monitors code
2. Languages used to express monitors and design models are usually different

Proposed solution
- Express properties as UML observer automata directly in the design language
- Embed these monitors with our model interpreter

Results
- No more semantic gap
- Only one language to express system and monitors models
⇒ Helps engineers verify and monitor the embedded systems they are designing
Conclusion

Problems
1. Semantic gap between monitors model and monitors code
2. Languages used to express monitors and design models are usually different

Proposed solution
- Express properties as UML observer automata directly in the design language
- Embed these monitors with our model interpreter

Results
1. No more semantic gap
2. Only one language to express system and monitors models
   ⇒ Helps engineers verify and monitor the embedded systems they are designing
Conclusion

Benefits

- The same UML observer automata can be used for model verification and runtime monitoring
- The use of formal verification techniques by engineers is facilitated
Conclusion

Benefits
- The same UML observer automata can be used for model verification and runtime monitoring
- The use of formal verification techniques by engineers is facilitated

Drawbacks
- Only observed failures can be detected
- Monitoring overhead (does not impede scalability)
## Conclusion

### Benefits
- The same UML observer automata can be used for model verification and runtime monitoring
- The use of formal verification techniques by engineers is facilitated

### Drawbacks
- Only observed failures can be detected
- Monitoring overhead (does not impede scalability)

### Perspectives
- Extend expressivity of guards in UML observer automata
- Integrate other model-based specification formalisms
Thank you for your attention

