Verifying and Monitoring UML Models with Observer Automata

A Transformation-free Approach

ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS'19) in Munich, Germany

Valentin BESNARD [1] Ciprian TEODOROV [2] Frédéric JOUAULT [1]
Matthias BRUN [1] Philippe DHAUSSY [2]

[1] ERIS, ESEO-TECH, Angers, France

[2] Lab-STICC UMR CNRS 6285, ENSTA Bretagne, Brest, France

This work has been partially funded by Davidson.

Table of Contents

Context

Observations

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

Context

Observations

- Increasing complexity and connectivity of embedded systems
  - $\Rightarrow$ Increasing exposure to potential software failures
  - $\Rightarrow$ 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)

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

Overview

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

Overview

Goal

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

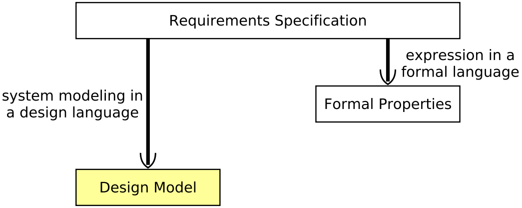
Our previous work [Besnard et al., MODELS 2018]

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

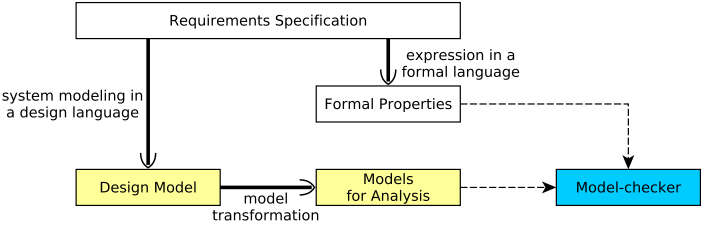
In this work

3. Identify problems on classical **monitoring** approaches
4. Can we address these problems with the model interpreter approach?

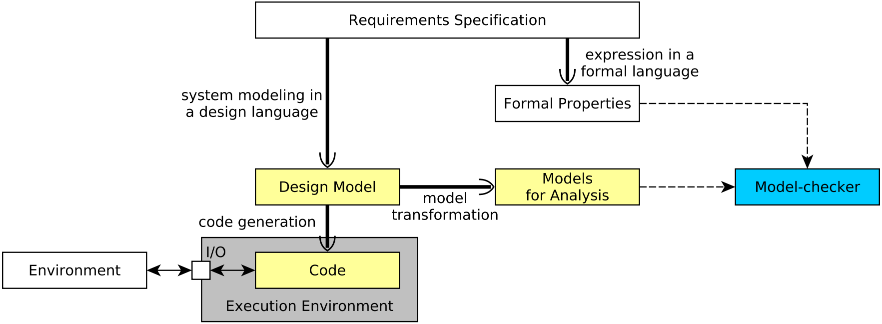
(1) Classical Approach with Model-checking



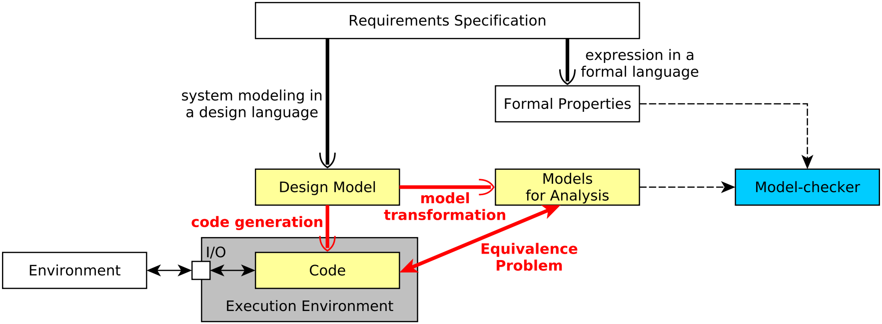
(1) Classical Approach with Model-checking



(1) Classical Approach with Model-checking

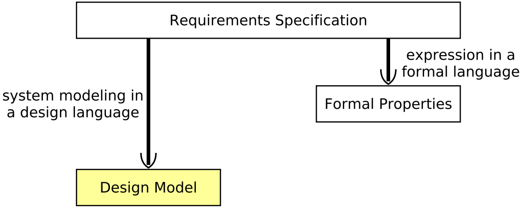


(1) Classical Approach with Model-checking (Problems)

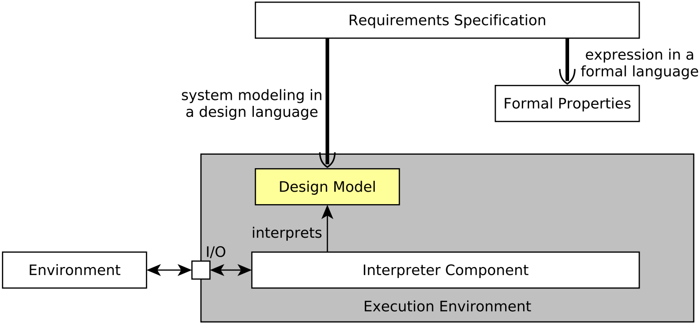


Problems: Two semantic gaps and an equivalence problem caused by transformations of the design model into different languages

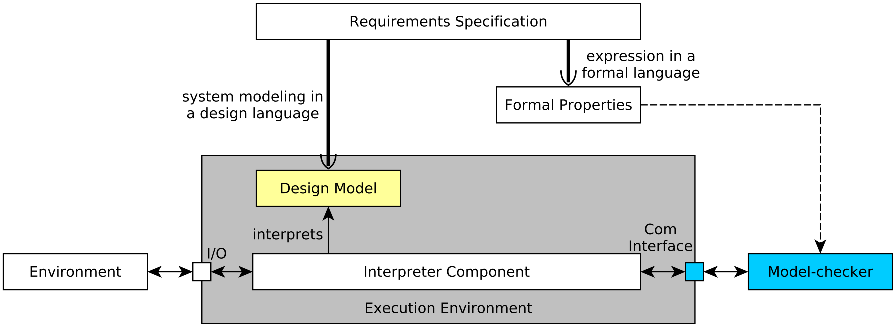
(2) Our Approach with Model-checking [Besnard et al., MODELS 2018]



(2) Our Approach with Model-checking [Besnard et al., MODELS 2018]

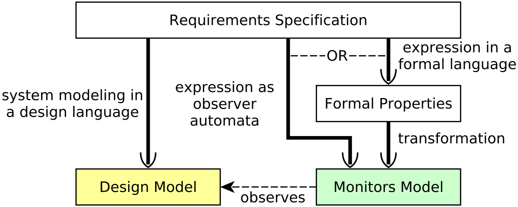


(2) Our Approach with Model-checking [Besnard et al., MODELS 2018]

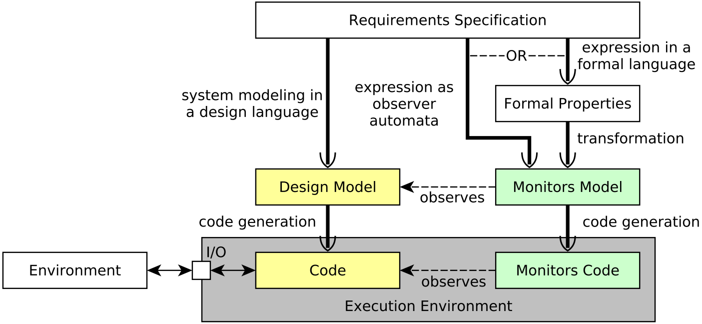


A unique definition of the language semantics
for verification activities and model execution

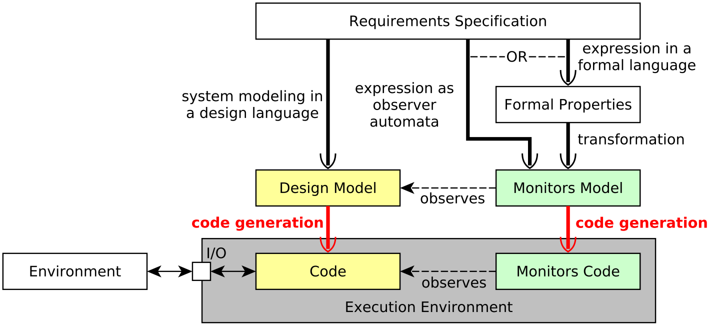
(3) Classical Approach with Monitoring



(3) Classical Approach with Monitoring

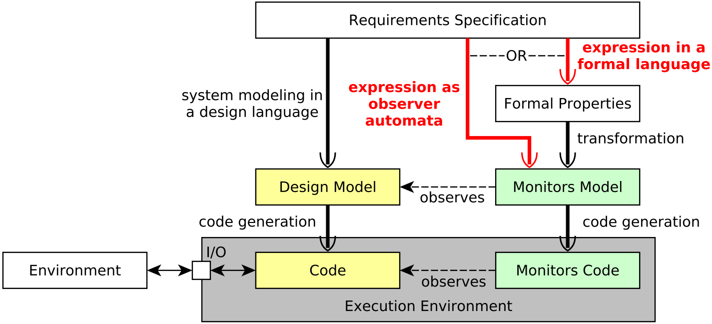


(3) Classical Approach with Monitoring (Problems)



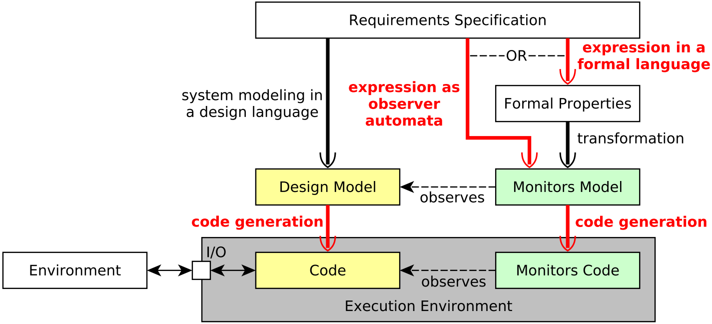
1. Semantic gap between monitors model and monitors code

(3) Classical Approach with Monitoring (Problems)



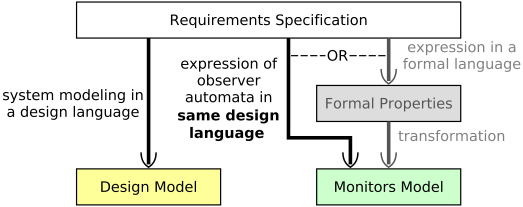
1. Semantic gap between monitors model and monitors code
2. Languages used to express monitors and design models are usually different

(3) Classical Approach with Monitoring (Problems)

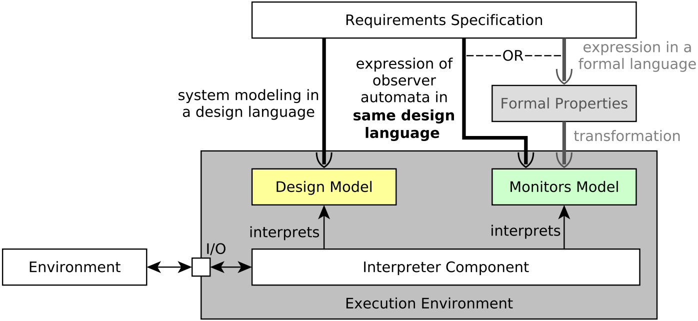


1. Semantic gap between monitors model and monitors code
2. Languages used to express monitors and design models are usually different

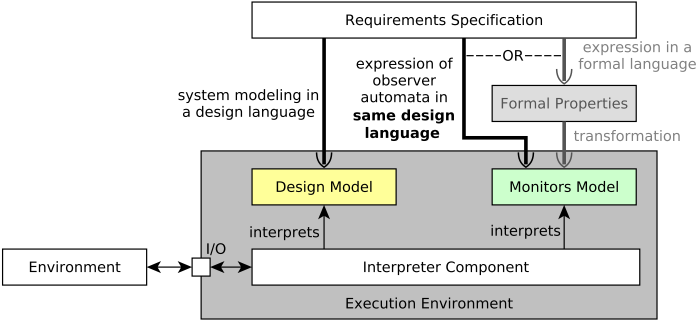
(4) Our Approach with Monitoring



(4) Our Approach with Monitoring



(4) Our Approach with Monitoring



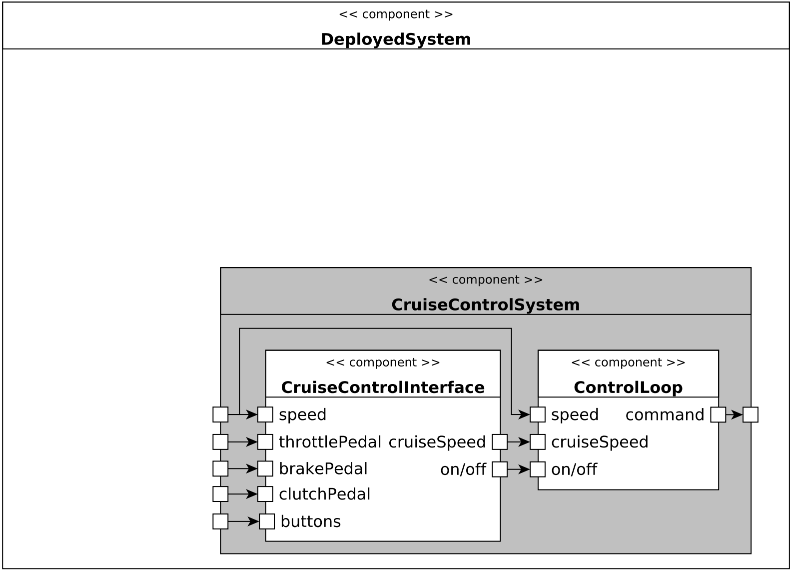
The same component interprets both design and monitors models:

1. No semantic gap
2. Only one language to express system and monitors models

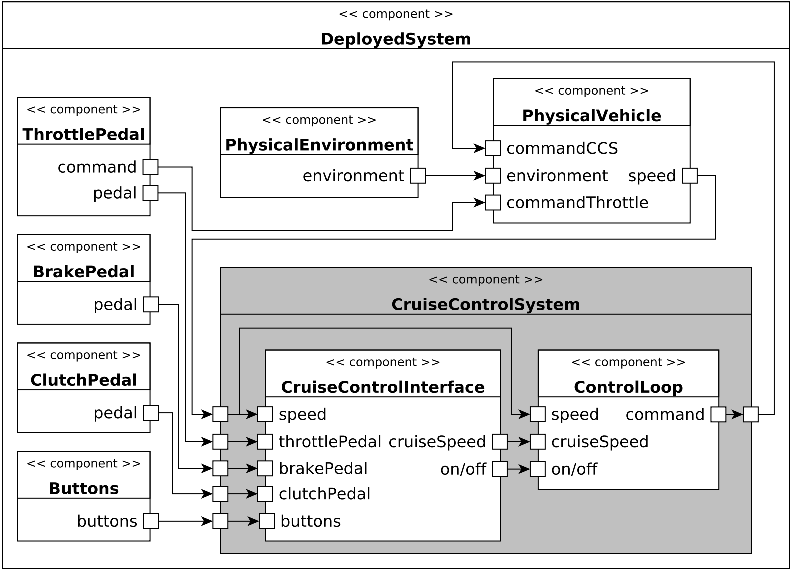
Table of Contents

Table of Contents

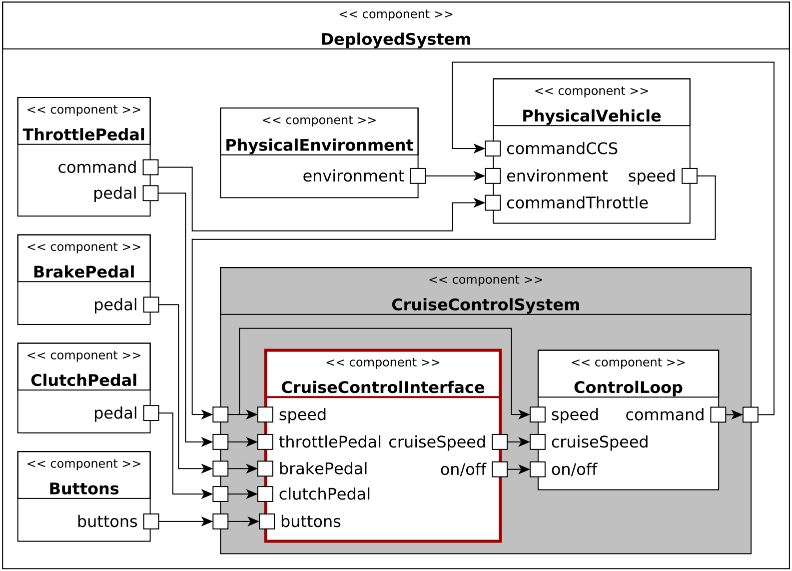
Cruise Control Overview



Cruise Control Overview



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

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

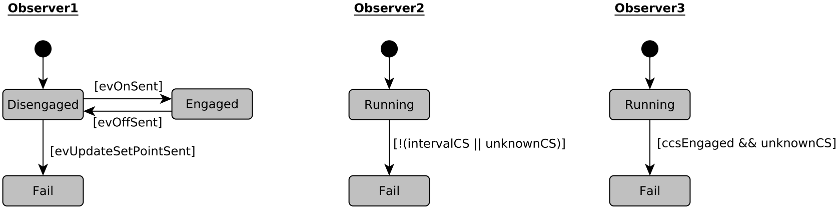
Expressivity = safety properties (something bad happens)

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

UML Observer Automata

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.



UML Observer Automata (Interpretation for Analysis Activities)

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.

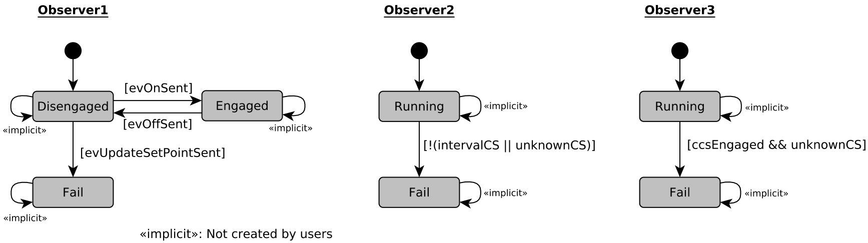


Table of Contents

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.

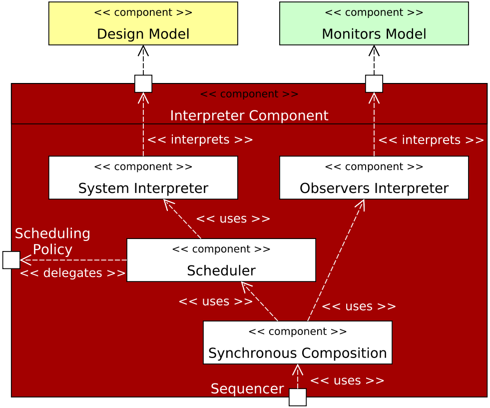
- At each step, a synchronous transition must be fired
- A synchronous transition is composed of:
  - One transition of the system
  - One transition per observer automaton

Synchronous Composition

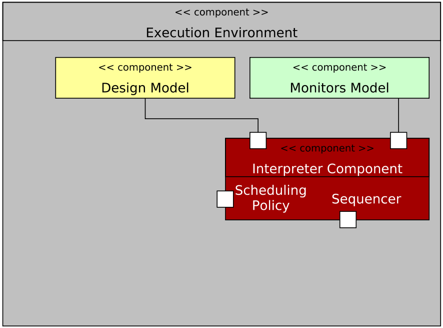
Principle

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**

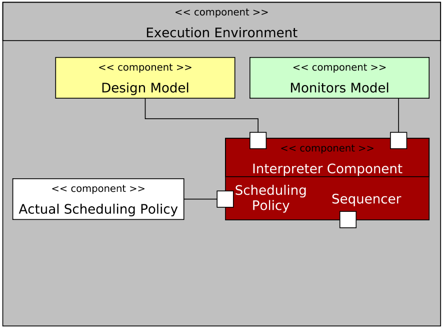


Runtime Monitoring with UML Observer Automata



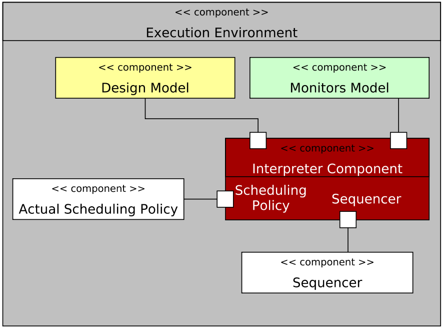
Runtime Monitoring with UML Observer Automata

- Use the actual scheduling policy (e.g., round robin on active objects)



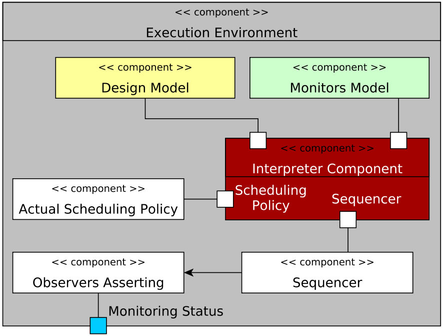
Runtime Monitoring with UML Observer Automata

- Use the actual scheduling policy (e.g., round robin on active objects)
- Use the execution sequencer that fires synchronous transitions in loop



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



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
  - `[] !|OBSERVER_FAIL(obs)|`

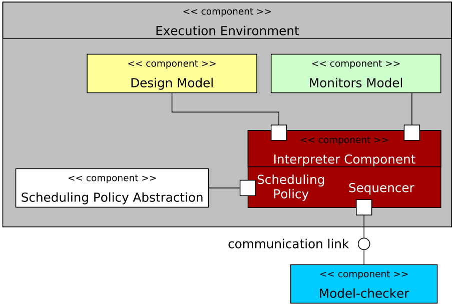
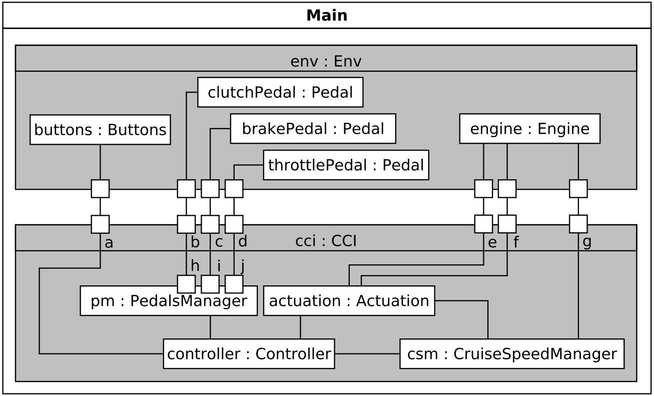


Table of Contents

Cruise Control Interface Model Under Verification

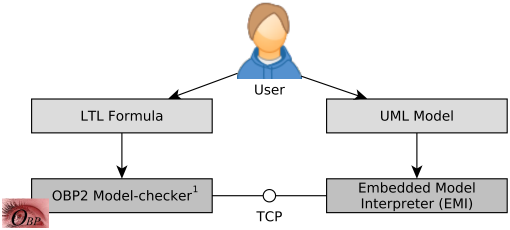
model under verification = system model + abstract environment model



Names of ports

a) cciButtonsPort
b) cciClutchPedalPort
c) cciBrakePedalPort
d) cciThrottlePedalPort
e) cciOnOffPort
f) cciSpeedPort
g) cciCruiseSpeedPort
h) pmClutchPedalPort
i) pmBrakePedalPort
j) pmThrottlePedalPort

Experiments

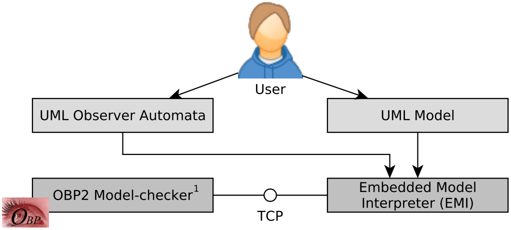


Experiments

- Compare verification results obtained with:
  - LTL formulae

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

Experiments

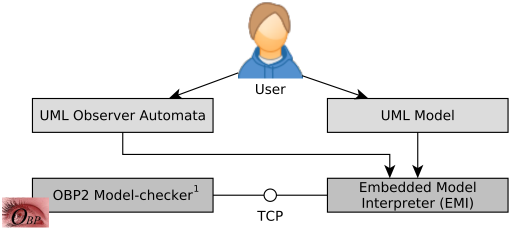


Experiments

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

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

Experiments



Experiments

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

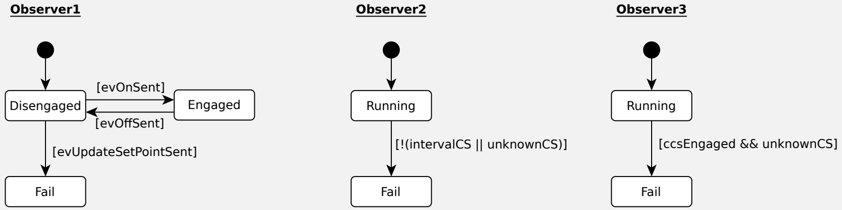
[1] [Teodorov et al., 2017] https://plug-obp.github.io/

Model-Checking of the Level Crossing Model

Expression of properties as LTL formulae

1. `[] ((|evOffSent| and !|evOnSent|) -> (!|evUpdateSetPointSent| W |evOnSent|))`
2. `[] (|intervalCS| or |unknownCS|)`
3. `[] (|ccsEngaged| -> !|unknownCS|)`

Expression of properties as UML observer automata



Results - Model-checking

	LTL Formulae	UML Observer Automata
Property 1	✓	✓
Property 2	✓	✓
Property 3	✗	✗

✓: Property verified ✗: Property violated

Analysis of the counter-example

Events *resetCS* and *disengage* could be processed in any order
$\Rightarrow$ Bad event interleaving

Model state-space

46,444,386 configurations linked by 82,734,350 transitions

Results - Monitoring

	Initial Model	Fixed Model
Property 1	● (green)	● (green)
Property 2	● (green)	● (green)
Property 3	● (red)	● (green)

● (green): No failure detected ● (red): Failure detected

Overhead of the monitoring infrastructure

- Execution performance: +6.5%
- Memory footprint: +1.2%

Results - Monitoring

	Initial Model	Fixed Model
Property 1	● (green)	● (green)
Property 2	● (green)	● (green)
Property 3	● (red)	● (green)

● (green): No failure detected   ● (red): 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

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

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

$\Rightarrow$ 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

Bibliography

Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy.
Unified LTL Verification and Embedded Execution of UML Models.
In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, Copenhagen, Denmark, October 2018.

OMG.
Unified Modeling Language, December 2017.
https://www.omg.org/spec/UML/2.5.1/PDF.

Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux.
Environment-driven reachability for timed systems.
International Journal on Software Tools for Technology Transfer, 19(2):229–245, Apr 2017.

Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy.
Past-Free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis.
Software Testing, Verification and Reliability, 26(7):516–542, 2016.