Embedded UML Model Execution to Bridge the Gap between Design and Runtime

First International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems (MDE@DeRun 2018)
co-located with STAF 2018 in Toulouse, France

June 28, 2018

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

Overview

Sommaire

Context

Observations

- Increasing complexity of embedded systems
- Emergence of new needs and applications
- Connection of these systems to networks (IoT)

Context

Observations

- Increasing complexity of embedded systems
- Emergence of new needs and applications
- Connection of these systems to networks (IoT)

Consequences

- Software programs increasingly difficult to design, maintain, and evolve
- Bugs and design faults increasingly difficult to detect and fix

Problem

Issues in terms of design, implementation, and analysis

More and more difficult to:

- Understand diagnosis results in terms of design concepts
- Give runtime feedbacks on the design model
- Visualize and analyze system execution
- Carry out diagnosis activities and runtime measures analysis

Problem

Issues in terms of design, implementation, and analysis

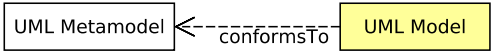
More and more difficult to:

- Understand diagnosis results in terms of design concepts
- Give runtime feedbacks on the design model
- Visualize and analyze system execution
- Carry out diagnosis activities and runtime measures analysis

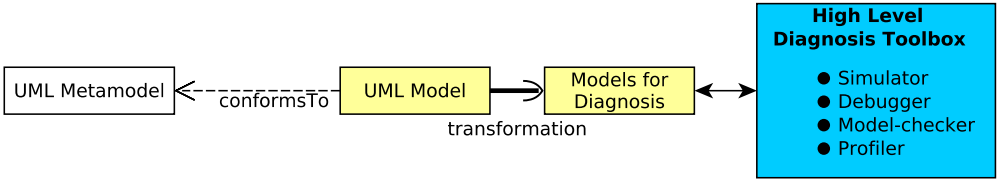
Formulation of the problem

How to link design and runtime concepts?

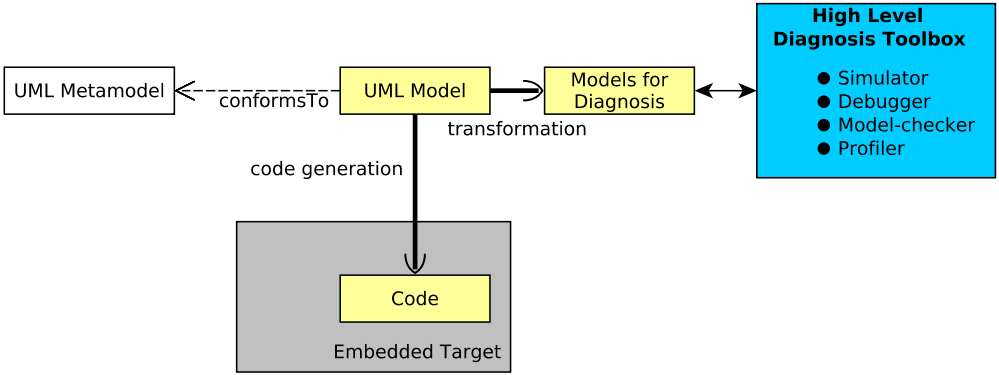
Classical Approaches and their Issues



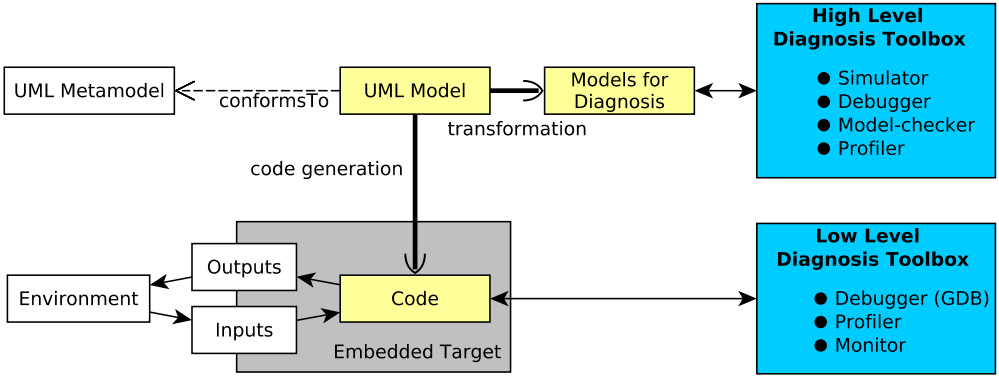
Classical Approaches and their Issues



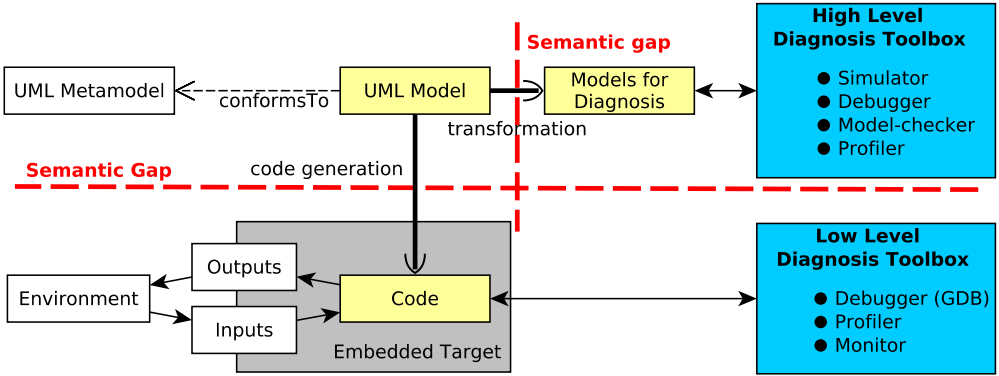
Classical Approaches and their Issues



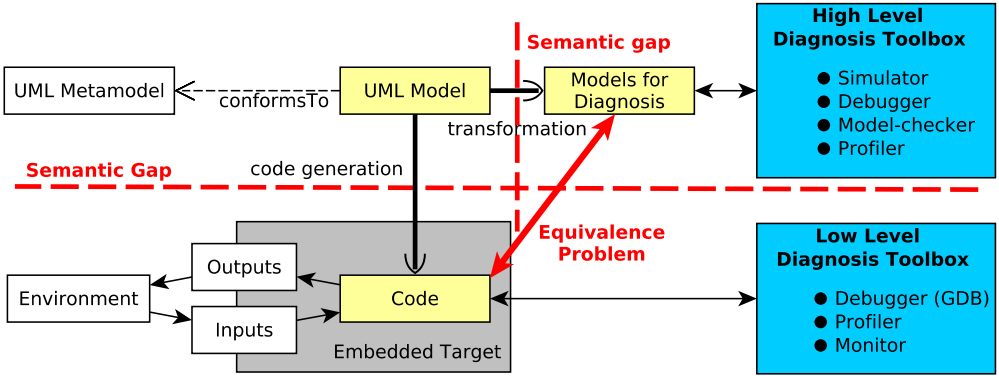
Classical Approaches and their Issues



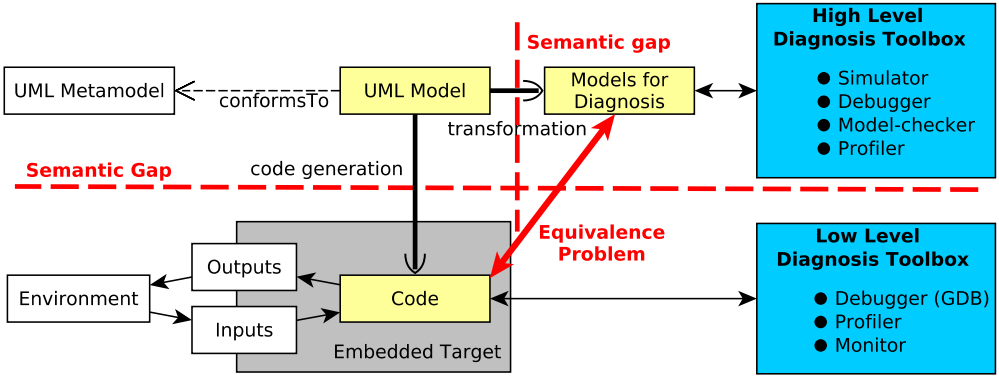
Classical Approaches and their Issues



Classical Approaches and their Issues

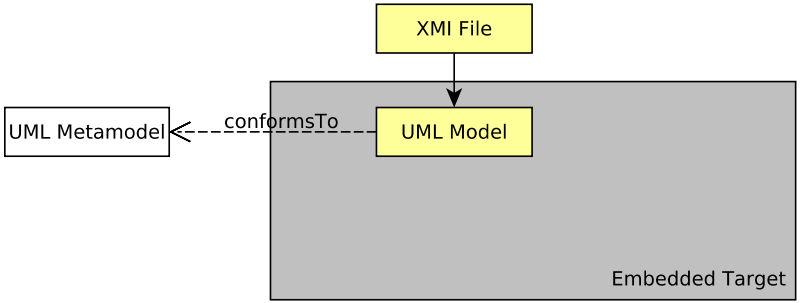


Classical Approaches and their Issues

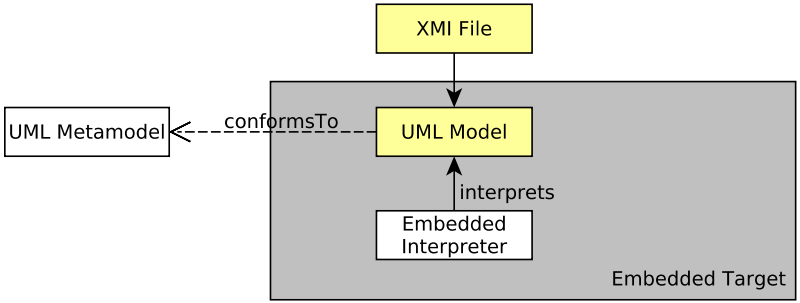


Main cause: Execution and diagnosis formalisms usually differ from the design formalism.

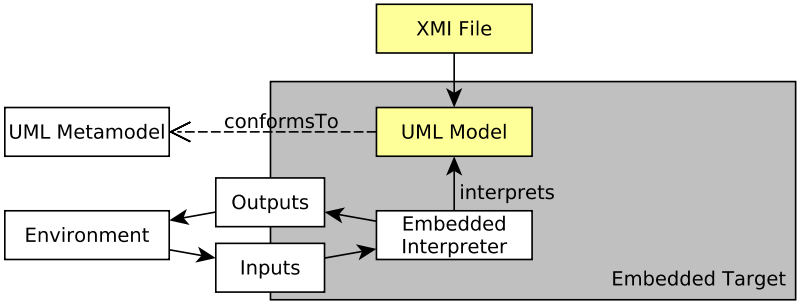
Our Approach



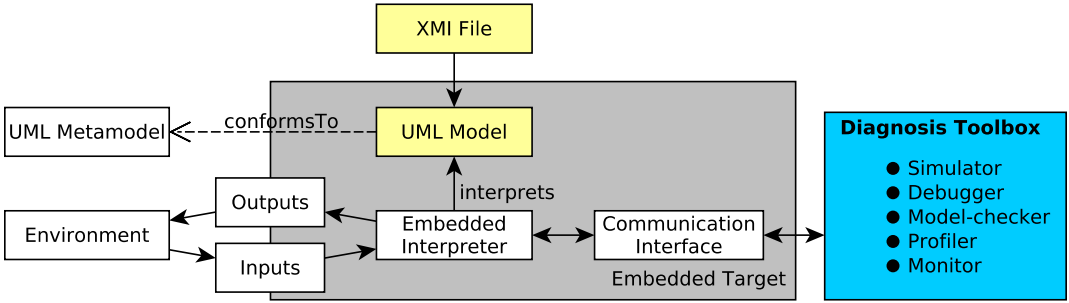
Our Approach



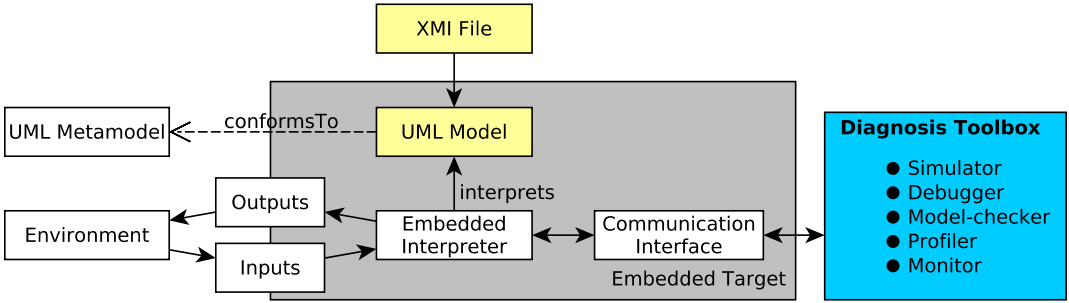
Our Approach



Our Approach



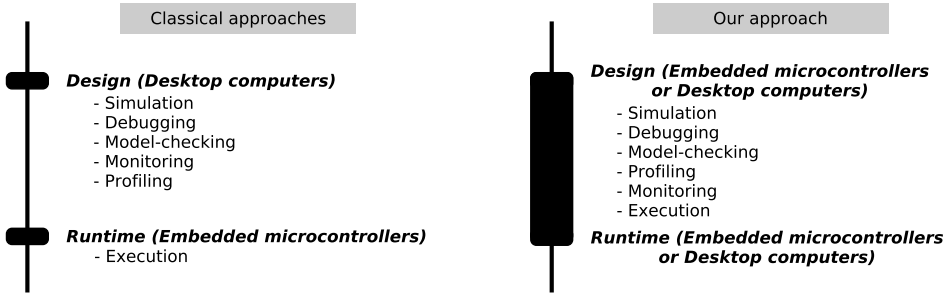
Our Approach



Solution: Execute and analyze the UML design model.

Interactions between *Design* and *Runtime*

- Continuum between *Design* and *Runtime* thanks to the use of:
  - The same model
  - The same execution semantics



Interactions between *Design* and *Runtime*

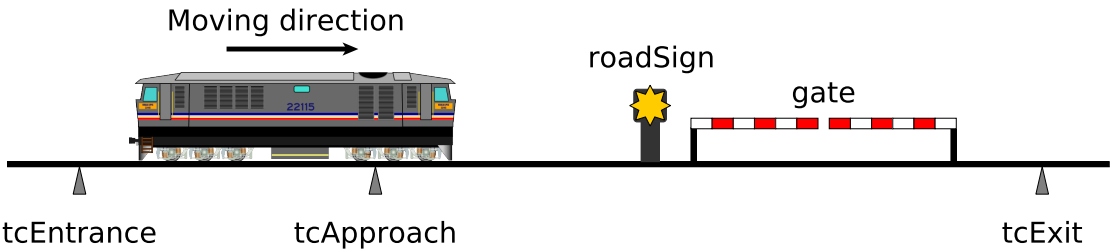
- Continuum between *Design* and *Runtime* thanks to the use of:
  - The same model
  - The same execution semantics
- Multiple advantages:
  - Easier interactions between *Design* and *Runtime*
  - No semantic gap
  - Direct expression of diagnosis results in terms of UML concepts
    - Simplifying addition of feedback on the model

Interactions between *Design* and *Runtime*

- Continuum between *Design* and *Runtime* thanks to the use of:
  - The same model
  - The same execution semantics
- Multiple advantages:
  - Easier interactions between *Design* and *Runtime*
  - No semantic gap
  - Direct expression of diagnosis results in terms of UML concepts
    - Simplifying addition of feedback on the model
- Expected results:
  - Increased development quality
  - Increased productivity
  - Reduced time-to-market

Sommaire

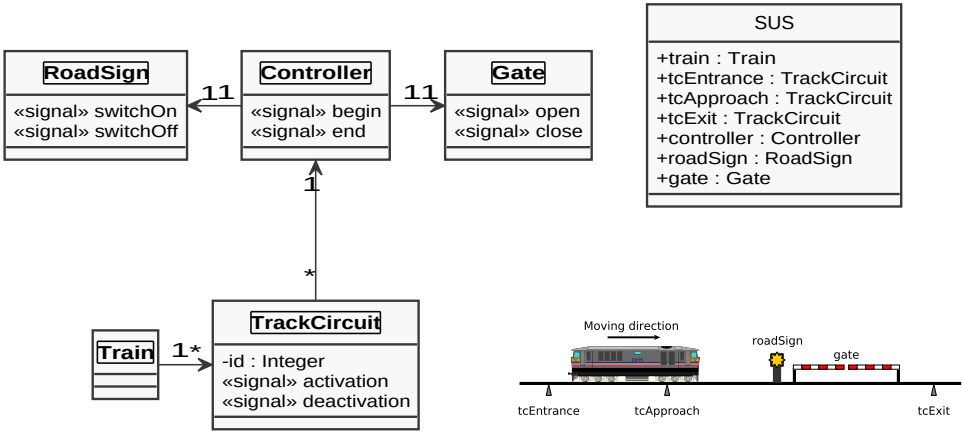
Motivating Example: Level Crossing



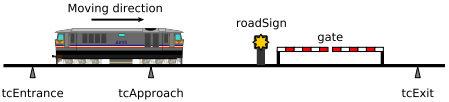
Goal

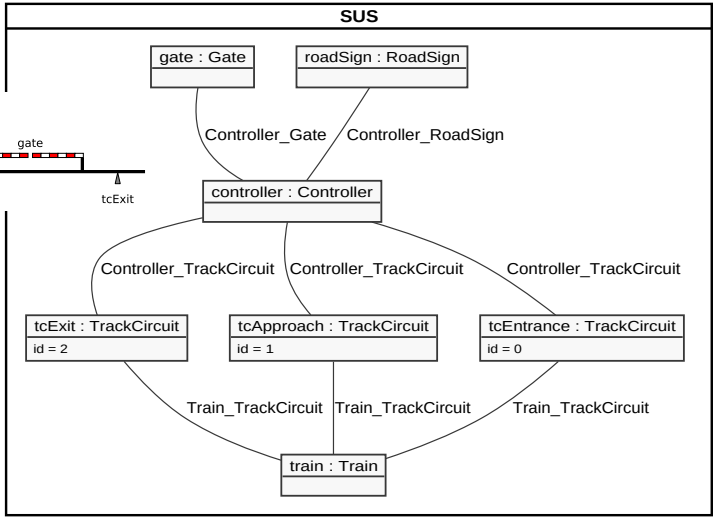
Ensure safety during the passage of the train

Motivating Example: Level Crossing (Class Diagram)

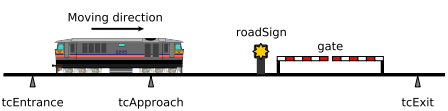


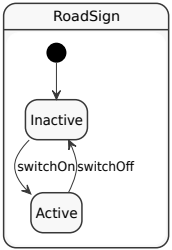
Motivating Example: Level Crossing (Composite Structure Diagram)

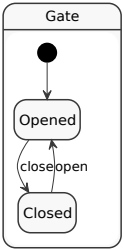


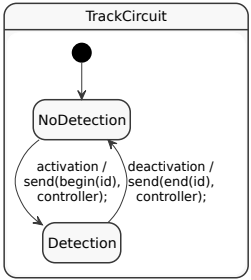


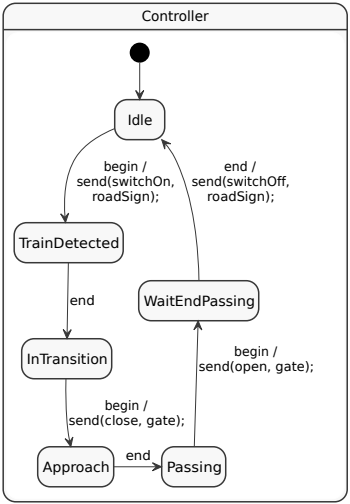
Motivating Example: Level Crossing (State Machines)











Sommaire

Interactions with Design Tools

Two kinds of interactions

- Online interactions
- Offline interactions

Interactions with Design Tools

Two kinds of interactions

- Online interactions
- Offline interactions

Three interaction modes

- Simulation (online)
- Execution followed by offline trace analysis
- Debugging (online)

Interactions with Design Tools

Two kinds of interactions

- Online interactions
- Offline interactions

Three interaction modes

- **Simulation (online)**
- Execution followed by offline trace analysis
- Debugging (online)

Communication Protocol

Application layer protocol

Composed of 5 kinds of requests:

- Get configuration
- Set configuration
- Get fireable transitions
- Fire transition
- Reset interpreter

Communication Protocol

Application layer protocol

Composed of 5 kinds of requests:

- Get configuration
- Set configuration => Enable back-in-time simulation
- Get fireable transitions
- Fire transition
- Reset interpreter

Communication Protocol

Application layer protocol

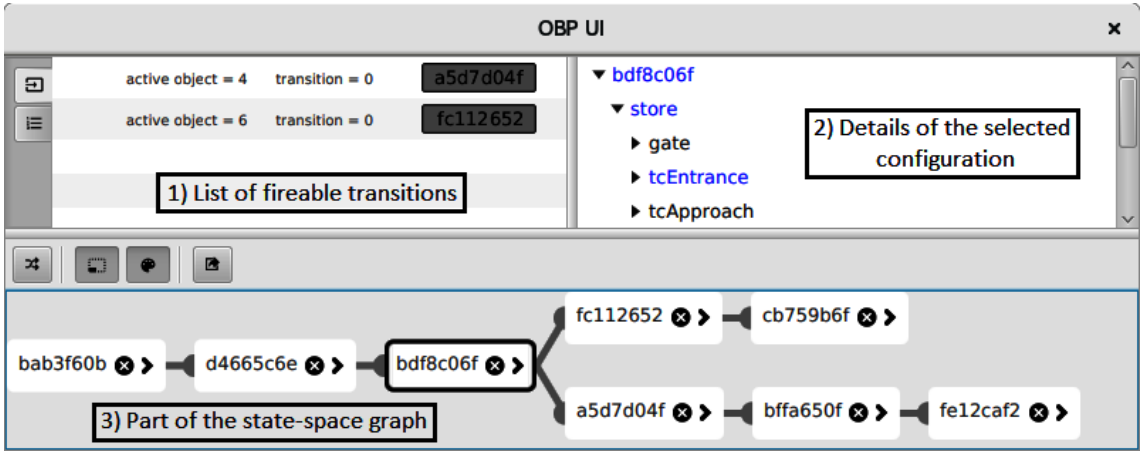
Composed of 5 kinds of requests:

- Get configuration
- Set configuration => Enable back-in-time simulation
- Get fireable transitions
- Fire transition
- Reset interpreter

Optimizations

- The *diff mode* enables to exchange only the parts that are different for get and set configuration requests
  - previous configuration + diff(previous, new) = new configuration

Simulation


OBP UI
active object = 4
transition = 0
a5d7d04f
active object = 6
transition = 0
fc112652
1) List of fireable transitions
bdf8c06f
store
gate
tcEntrance
tcApproach
2) Details of the selected configuration
bab3f60b
d4665c6e
bdf8c06f
fc112652
cb759b6f
a5d7d04f
bffa650f
fe12caf2
3) Part of the state-space graph

Interactions with Design Tools

Two kinds of interactions

- Online interactions
- Offline interactions

Three interaction modes

- Simulation (online)
- **Execution followed by offline trace analysis**
- Debugging (online)

Execution

1. Instrumentation of the interpreter code with C macros called when:
   - An active object sends an event
   - A state machine updates its current state
   - An attribute updates its value (optional)
   - No overhead if no trace required

Execution

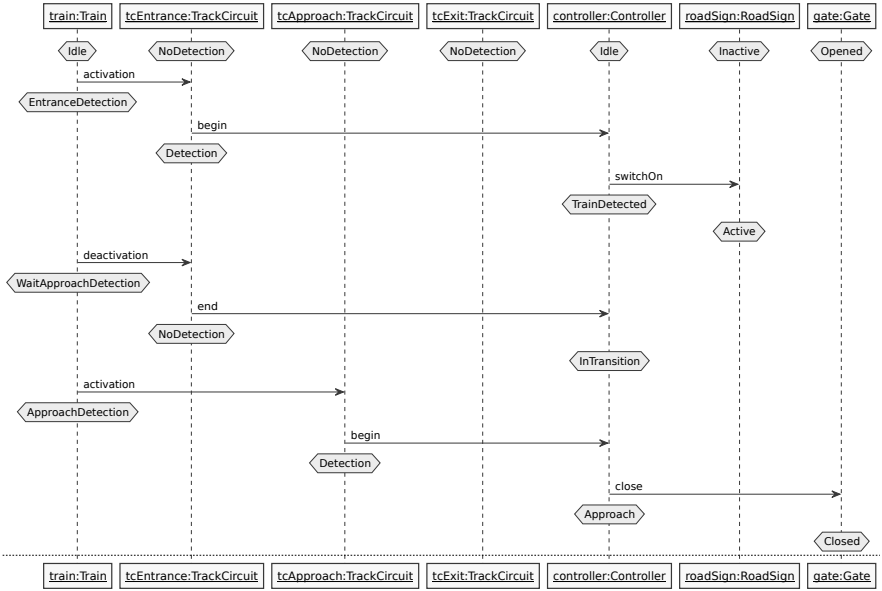
1. Instrumentation of the interpreter code with C macros called when:
   - An active object sends an event
   - A state machine updates its current state
   - An attribute updates its value (optional)
   - No overhead if no trace required
2. Choose the output formalism at compile-time (e.g., PlantUML)

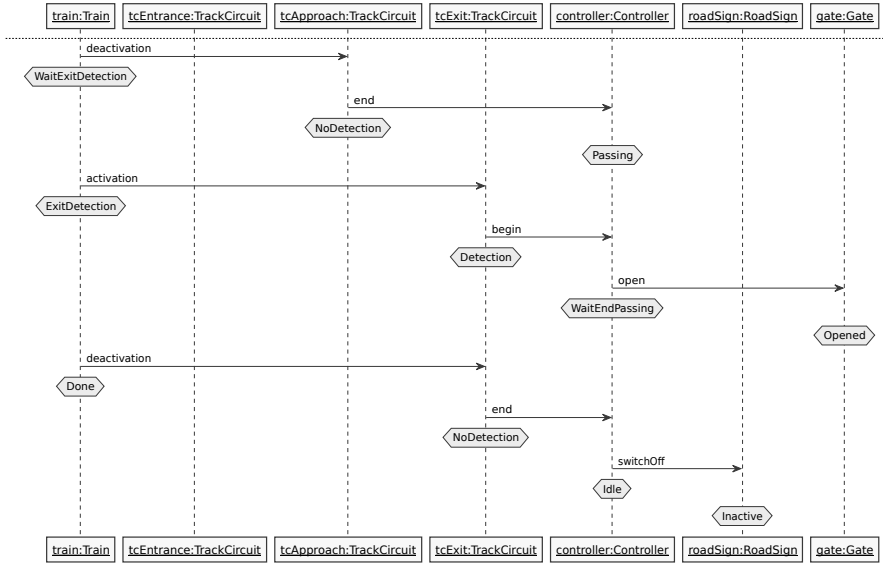
Execution

1. Instrumentation of the interpreter code with C macros called when:
   - An active object sends an event
   - A state machine updates its current state
   - An attribute updates its value (optional)
   - No overhead if no trace required
2. Choose the output formalism at compile-time (e.g., PlantUML)
3. Print the trace at runtime:
   - On the standard output
   - In a text file

Execution

1. Instrumentation of the interpreter code with C macros called when:
   - An active object sends an event
   - A state machine updates its current state
   - An attribute updates its value (optional)
   - No overhead if no trace required
2. Choose the output formalism at compile-time (e.g., PlantUML)
3. Print the trace at runtime:
   - On the standard output
   - In a text file
4. Generate a graphical diagram (Message Sequence Chart) of the trace


train:Train
tcEntrance:TrackCircuit
tcApproach:TrackCircuit
tcExit:TrackCircuit
controller:Controller
roadSign:RoadSign
gate:Gate
Idle
NoDetection
NoDetection
NoDetection
Idle
Inactive
Opened
activation
EntranceDetection
begin
Detection
switchOn
TrainDetected
Active
deactivation
WaitApproachDetection
end
NoDetection
InTransition
activation
ApproachDetection
begin
Detection
close
Approach
Closed
train:Train
tcEntrance:TrackCircuit
tcApproach:TrackCircuit
tcExit:TrackCircuit
controller:Controller
roadSign:RoadSign
gate:Gate


train:Train
tcEntrance:TrackCircuit
tcApproach:TrackCircuit
tcExit:TrackCircuit
controller:Controller
roadSign:RoadSign
gate:Gate
deactivation
WaitExitDetection
end
NoDetection
Passing
activation
ExitDetection
begin
Detection
open
WaitEndPassing
Opened
deactivation
Done
end
NoDetection
switchOff
Idle
Inactive
train:Train
tcEntrance:TrackCircuit
tcApproach:TrackCircuit
tcExit:TrackCircuit
controller:Controller
roadSign:RoadSign
gate:Gate

Interactions with Design Tools

Two kinds of interactions

- Online interactions
- Offline interactions

Three interaction modes

- Simulation (online)
- Execution followed by offline trace analysis
- **Debugging (online)**

Debugging

Debugging loop

Check if there is a command sent by the debugger to process:

- If a command is received, process the command
- If no command is received, continue execution

Debugging

Debugging loop

Check if there is a command sent by the debugger to process:

- If a command is received, process the command
- If no command is received, continue execution

Functionalities

- Optional back-in-time debugging
- Change current state of active objects and inject events into event pools

Debugging

Debugging loop

Check if there is a command sent by the debugger to process:

- If a command is received, process the command
- If no command is received, continue execution

Functionalities

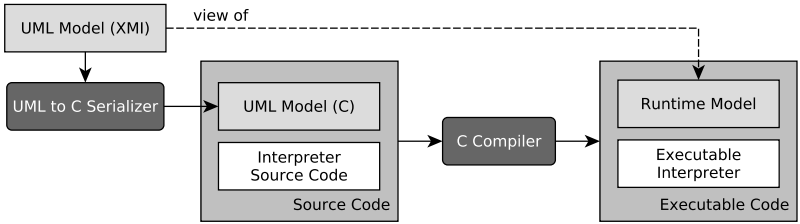
- Optional back-in-time debugging
- Change current state of active objects and inject events into event pools

Limitations

- Cannot debug opaque behaviors and opaque expressions implemented as C functions
- No support for adding/removing breakpoints yet

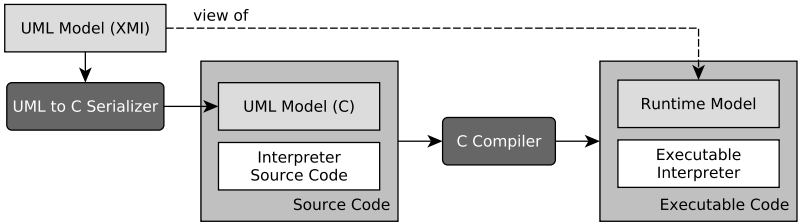
Sommaire

Loading UML Models



- Serialization of the UML model
  - Generate only *data* and not functions (*program*) from both structural (e.g., classes), and behavioral (e.g., state machines) parts of the model
    - Except for opaque expressions and opaque behaviors, for each of which a function is generated

Loading UML Models



- Serialization of the UML model
  - Generate only *data* and not functions (*program*) from both structural (e.g., classes), and behavioral (e.g., state machines) parts of the model
    - Except for opaque expressions and opaque behaviors, for each of which a function is generated
- Compilation of the model (*data*) and of the interpreter (*program*) into executable code
  - Can be seen as compile-time model loading

Interpretation of UML models

Operational semantics (subset of UML)

- A unique implementation of the semantics in the interpreter
- Aiming at compatibility with:
  - Precise Semantics for UML Composite Structures (PSCS)
  - Precise Semantics for UML State Machines (PSSM)

Interpretation of UML models

Operational semantics (subset of UML)

- A unique implementation of the semantics in the interpreter
- Aiming at compatibility with:
  - Precise Semantics for UML Composite Structures (PSCS)
  - Precise Semantics for UML State Machines (PSSM)

Execution loop

- Evaluate fireable transitions according to events and interactions with the environment
- Fire a fireable transition

Interpretation of UML models

Operational semantics (subset of UML)

- A unique implementation of the semantics in the interpreter
- Aiming at compatibility with:
  - Precise Semantics for UML Composite Structures (PSCS)
  - Precise Semantics for UML State Machines (PSSM)

Execution loop

- Evaluate fireable transitions according to events and interactions with the environment
- Fire a fireable transition

Targets

- Desktop computers (e.g., PC with a Linux operating system + TCP)
- Embedded microcontrollers on bare-metal (e.g., stm32f4 discovery + RS232)

Sommaire

Conclusion

Our contribution

- Uses the same operational semantics implementation for all activities
- Facilitates interactions between design and runtime concepts
- Expresses diagnosis results in terms of design concepts

Conclusion

Our contribution

- Uses the same operational semantics implementation for all activities
- Facilitates interactions between design and runtime concepts
- Expresses diagnosis results in terms of design concepts

Perspectives

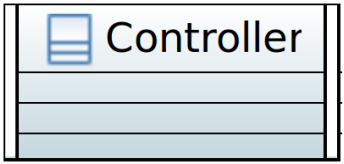
- Support for adding/removing breakpoints
- Support a larger subset of UML
- Integration with UML modelers (e.g., Papyrus)
- Evaluation of the interpreter overhead

Thank you for your attention

Appendix: Loading UML Models

Deployment process

- Design of the level crossing model in Eclipse UML (graphically with Papyrus or textually with tUML)




```
1 class |Controller| behavesAs SM {
2   stateMachine SM {}
3 }
```

Appendix: Loading UML Models

Deployment process

- Design of the level crossing model in Eclipse UML (graphically with Papyrus or textually with tUML)

```
<packagedElement xmi:type="uml:Class"
  xmi:id="_hcP2cJFrEeeKv5ZjdgN-yQ"name="Controller"
  classifierBehavior="_hcXyQJFrEeeKv5ZjdgN-yQ" isActive="true">
  <ownedBehavior xmi:type="uml:StateMachine"
    xmi:id="_hcXyQJFrEeeKv5ZjdgN-yQ" name="SM">
  </ownedBehavior>
</packagedElement>
```

Appendix: Loading UML Models

Deployment process

- Design of the level crossing model in Eclipse UML (graphically with Papyrus or textually with tUML)
- Serialization into C language as struct initializers

```
1 UML_Class class__Controller = {
2   .c_kind = C_UML_Class,
3   .visibility = UML_PUBLIC,
4   .name = "Controller",
5   .classifierBehavior = (UML_Behavior*)&stateMachine__Controller,
6   .isActive = 1
7 };
```

Appendix: Loading UML Models

Deployment process

- Design of the level crossing model in Eclipse UML (graphically with Papyrus or textually with tUML)
- Serialization into C language as struct initializers
- Model linked at build time with the interpreter