Unified LTL Verification and Embedded Execution of UML Models

ACM/IEEE 21\textsuperscript{th} International Conference on Model Driven Engineering Languages and Systems (MODELS ’18) in Copenhagen, Denmark

Valentin BESNARD \textsuperscript{1}  Matthias BRUN \textsuperscript{1}  Frédéric JOUAULT \textsuperscript{1}  
Ciprian TEODOROV \textsuperscript{2}  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 UML Execution Engine
4 Diagnosis Activities
5 Application to the Illustrating Example
6 Conclusion
# Table of Contents

1. Introduction
2. Illustrating Example
3. UML Execution Engine
4. Diagnosis Activities
5. Application to the Illustrating Example
6. Conclusion
Introduction

Context

Observations

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

Context

Observations

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

Consequences on software programs

- More prone to uncertain behaviors, security flaws, and design mistakes
- More safety and security requirements
Introduction

Context

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

Consequences on software programs
- More prone to uncertain behaviors, security flaws, and design mistakes
- More safety and security requirements

Consequence on software development
- Increasing need of verification and validation
Classical UML-based Approaches
Classical UML-based Approaches

- UML Metamodel
  - conformsTo
- UML Model
  - transformation
- Models for Diagnosis
- High Level Diagnosis Toolbox
  - Simulator
  - Debugger
  - Model-checker
  - Profiler
Classical UML-based Approaches

UML Metamodel

conformsTo

UML Model

transformation

code generation

Models for Diagnosis

High Level Diagnosis Toolbox
- Simulator
- Debugger
- Model-checker
- Profiler

Embedded Target

Code
Classical UML-based Approaches

- UML Metamodel
  - conformsTo
  - transformation
  - code generation

- Models for Diagnosis

- Environment
  - Outputs
  - Inputs

- Code

- Embedded Target

- High Level Diagnosis Toolbox
  - Simulator
  - Debugger
  - Model-checker
  - Profiler

- Low Level Diagnosis Toolbox
  - Debugger
  - Profiler
  - Monitor
Some Problems

First issue: Semantic gap between design model and executable code.

- UML Metamodel
- ConformsTo
- UML Model
- Transformation
- Models for Diagnosis

Semantic Gap
- Code Generation
- Environment
- Outputs
- Inputs
- Embedded Target

High Level Diagnosis Toolbox
- Simulator
- Debugger
- Model-checker
- Profiler

Low Level Diagnosis Toolbox
- Debugger
- Profiler
- Monitor
Some Problems

Second issue: Semantic gap between design model and diagnosis model.
Third issue: An equivalence relation between verified formal models and deployed code should be built, proven, and maintained.
Main cause of these problems: Multiple definitions of the modeling language semantics.
Our Approach: A Unified Modeling Language Semantics

![Diagram](attachment:image.png)

- UML Metamodel
- conformsTo
- UML Model
- Embedded Target
Our Approach: A Unified Modeling Language Semantics
Our Approach: A Unified Modeling Language Semantics
Our Approach: A Unified Modeling Language Semantics

- UML Metamodel
- UML Model
- Embedded Interpreter
- Communication Interface
- Outputs
- Inputs
- Environment
- Embedded Target
- UML Diagnosis Toolbox
  - Simulator
  - Debugger
  - Model-checker
  - Profiler
  - Monitor
Our Approach: A Unified Modeling Language Semantics

Other tools are able to execute and analyze models: GEMOC Studio [Bousse et al., 2016], Moliz [Mayerhofer et al., 2012], Moka [Revol et al. 2018], GUML [Charfi et al, 2012], Unicomp [Ciccozzi, 2018], Mbeddr [Voelter et al., 2012], etc.
Our Approach: A Unified Modeling Language Semantics

A single implementation of the language semantics for all activities: simulation, execution, and diagnosis.
Results

Simulation
- Trace-based simulation

Execution
- On bare-metal (without operating system) embedded targets
- On desktop computers

Diagnosis
- State-space exploration
- Deadlock detection
- Model-checking of Linear Temporal Logic (LTL) properties
# Table of Contents

1. Introduction
2. Illustrating Example
3. UML Execution Engine
4. Diagnosis Activities
5. Application to the Illustrating Example
6. Conclusion
Level Crossing Overview

Goal
Ensure safety during the passage of the train
Level Crossing Model Requirements

Deadlock detection
- Ensuring that the model is deadlock free.

System requirements
1. The Gate is closed when the Train is on the level crossing.
2. The light of the RoadSign is active when the Train is on the level crossing.
3. The Gate finally opens after being closed.
4. The light of the RoadSign is finally turn off after being activated.
Level Crossing Model (Class Diagram)
Level Crossing Model (Composite Structure Diagram)

Illustrating Example

Moving direction
gate
roadSign
railwaySign

SUS

controller:Controller

roadSign:RoadSign
Controller_RoadSign

gate:Gate
Controller_Gate

tcFar:EntranceTC
Controller_EncounterTC

tcClose:EntranceTC
Controller_EncounterTC

Controller_EncounterTC

Controller_RailwaySign

railwaySign:RailwaySign

tcExit:ExitTC
Controller_WayOutTC
Illustrating Example

Level Crossing Model (State Machines)

Controller

Idle

/ nbEngagedTrains = 0;

WaitRoadSignOn

roadSignOn / FarDetection

CloseDetection

gateClosed / send(switchOff, railwaySign);

WaitRailwaySignOn

railwaySignOn / send(open, gate);

nbEngagedTrains--;

WaitGateOpen

gateOpen / send(switchOff, roadSign);

WaitRoadSignOff

roadSignOff /

exit [nbEngagedTrains == 1] / send(switchOn, railwaySign);

entrance [nbEngagedTrains == 0] / send(switchOn, roadSign);

RoadSign

Inactive

switchOn / send(railwaySignOn, controller);

Active

switchOff / send(railwaySignOff, controller);

Gate

Open

close / send(gateClosed, controller);

Closed

open / send(gateOpen, controller);

EntranceTC

Detection

activation / send(entrance(id), controller);

ExitTC

Detection

activation / send(exit(id), controller);
Table of Contents

1 Introduction
2 Illustrating Example
3 UML Execution Engine
4 Diagnosis Activities
5 Application to the Illustrating Example
6 Conclusion
UML Interpreter Design

Interpreter

GuardEvaluator

EffectInterpreter

EventPool

FifoEventPool

OrderedListDeferredEventPool

ActiveObject

* activeObjects

* eventPool

Model

Peers

Store

uses

uses

model

peers

store
Loading the Runtime Model at Compile-Time
Loading the Runtime Model at Compile-Time

**Principle**

UML Model

.text (program memory)
.data (data memory)

**Implementation**

UML Model (XMI)

1. class Controller behavesAs SM {
2.    stateMachine SM {}
3. }

Controller
Loading the Runtime Model at Compile-Time

**Principle**
- UML Model

**Implementation**
1. UML Model (XMI)
2. UML to C Serializer
3. UML Model (in C)
4. Data Types for Action Language
5. Interpreter Source Code
6. Source Code (in C)
7. `Controller` class
   ```
   class Controller behavesAs SM {
   stateMachine SM {};
   }
   ```
8. `UML_Class` class
   ```
   UML_Class controller_class = {
   .c_kind = C_UML_Class,
   .visibility = UML_PUBLIC,
   .name = "Controller",
   .classifierBehavior = (UML_Behavior*) &controller_SM,
   .isActive = 1
   };
   ```
Loading the Runtime Model at Compile-Time

**Principle**

- UML Model

**Implementation**

- UML Model (XMI)
  - UML to C Serializer
  - UML Model (in C)
    - Data Types for Action Language
    - Interpreter Source Code
    - Source Code (in C)
  - C Compiler
    - Runtime Model
    - Executable Interpreter
    - Executable Code

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

```c
1 UML.Class controller_class = {
2    .c_kind = C_UML_Class,
3    .visibility = UML_PUBLIC,
4    .name = "Controller",
5    .classifierBehavior = 
        (UML_Behavior*) &controller_SM,
6    .isActive = 1
7 };
```
Loading the Runtime Model at Compile-Time

Principle:
- Opaque Expressions
- Opaque Behaviors
- UML Model

Implementation:
- UML Model (XMI) → UML to C Serializer
  - UML Model (in C)
    - Data Types for Action Language
    - Interpreter Source Code
  - Source Code (in C)
  - C Compiler
  - Runtime Model
  - Executable Interpreter
  - Executable Code

opaqueBehavior = 'send(close, gate);'
Table of Contents

1 Introduction

2 Illustrating Example

3 UML Execution Engine

4 Diagnosis Activities

5 Application to the Illustrating Example

6 Conclusion
UML Model Diagnosis: Goals and Requirements

Our goals

- Simulate the model

Requirements to achieve these goals

Get fireable transitions
Fire transition
Get configuration

Diagnosis Tool
Controllable Interpreter
UML Model Diagnosis: Goals and Requirements

Our goals

- Simulate the model (with rollback for back-in-time simulation)
- Explore the model state-space
- Detect deadlocks

Requirements to achieve these goals

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

Diagram:

- Diagnosis Tool
  - Get fireable transitions
  - Fire transition
  - Get configuration
- Controllable Interpreter
  - Set configuration
**UML Model Diagnosis: Goals and Requirements**

**Our goals**
- Simulate the model (with rollback for back-in-time simulation)
- Explore the model state-space
- Detect deadlocks
- Verify formal properties via model-checking

**Requirements to achieve these goals**

```
Diagnosis Tool
  Get fireable transitions
    Fire transition
    Get configuration
  Set configuration
    Evaluate Predicate
```

```
Controllable Interpreter
```
Diagnosis Design

Design of an application layer protocol over:

- TCP connection
- Serial connection (e.g., UART, USB)
A formal property consists of:

- Atomic propositions (i.e., predicates related to model concepts)
  → Compiled into executable code by the converter
  → Evaluated by the controllable interpreter

- Logical operators used to link atomic propositions together
  → Evaluated by the diagnosis tool (model-checker)
# Table of Contents

1. Introduction
2. Illustrating Example
3. UML Execution Engine
4. Diagnosis Activities
5. Application to the Illustrating Example
6. Conclusion
Application to the Illustrating Example

Level Crossing Model Under Verification

Environment

System

Controller

Entrance Detection
[nbEngagedTrains == 0] / send(switchOn, roadSign);

Idle

WaitRoadSignOn

roadSignOn / 

Far Detection

entranceDetection / send(close, gate);

nbEngagedTrains++;

Close Detection

gateClose / send(switchOff, railwaySign);

Train

Exit Detection
[nbEngagedTrains == 1] / send(switchOn, railwaySign);

WaitRailwaySignOn

railwaySignOn / send(open, gate);

nbEngagedTrains--;

WaitGateOpen

gateOpen / send(switchOff, roadSign);

railwaySign

Passing

Active

switchOff / send(authorization, train);

Inactive

switchOn / send(railwaySignOn, controller);

RoadSign

RailwaySign

Gate

EntranceTC

ExitTC

Active

Inactive

Opened

Closed

Detection

Activation / send(entranceDetection(id), controller);

Activation / send(exitDetection(id), controller);
Model-Checking of the Level Crossing Model

Expression of Properties into LTL

1. $[] \neg (\text{trainIsPassing} \land \text{gateIsOpen})$
2. $[] \neg (\text{trainIsPassing} \land \text{roadSignIsOff})$
3. $[] (\neg \text{gateIsClosed} \rightarrow \Diamond \text{gateIsOpen})$
4. $[] (\neg \text{roadSignIsOn} \rightarrow \Diamond \neg \text{roadSignIsOff})$
Model-Checking of the Level Crossing Model

Expression of Properties into LTL

1. \[ \neg (\text{trainIsPassing} \land \text{gateIsOpen}) \]
2. \[ \neg (\text{trainIsPassing} \land \text{roadSignIsOff}) \]
3. \[ (\neg \text{gateIsClosed} \rightarrow \Diamond \neg \text{gateIsOpen}) \]
4. \[ (\neg \text{roadSignIsOn} \rightarrow \Diamond \neg \text{roadSignIsOff}) \]

Expression of Atomic Propositions

- \( \text{trainIsPassing} = | \text{train.state} == \text{PASSING} | \)
- \( \text{gateIsClosed} = | \text{gate.state} == \text{CLOSED} | \)
- \( \text{gateIsOpen} = | \text{gate.state} == \text{OPEN} | \)
- \( \text{roadSignIsOn} = | \text{roadSign.state} == \text{ACTIVE} | \)
- \( \text{roadSignIsOff} = | \text{roadSign.state} == \text{INACTIVE} | \)
Experiments

1https://plug-obp.github.io/
Experiments

Diagnosis of the level-crossing model on:
- Desktop computer
- STM32 discovery

1https://plug-obp.github.io/
Experiments

Using the two implementations of the event pool
- the FIFO implementation that drops ignored events
- the ordered list implementation that defers ignored events

https://plug-obp.github.io/
Results - Simulation

[Diagram of simulation results showing execution of BFS Explorer with various nodes and edges, including timestamps and event occurrences.]
## Results - State-space Exploration

<table>
<thead>
<tr>
<th></th>
<th>FIFO (drops)</th>
<th>OrderedList (defers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nb configurations</td>
<td>173</td>
<td>122</td>
</tr>
<tr>
<td>Nb transitions</td>
<td>276</td>
<td>209</td>
</tr>
</tbody>
</table>

State-space graph with FIFO

State-space graph with OrderedList
## Results - Deadlock Detection

<table>
<thead>
<tr>
<th></th>
<th>FIFO (drops)</th>
<th>OrderedList (defers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nb configurations</td>
<td>173</td>
<td>122</td>
</tr>
<tr>
<td>Nb transitions</td>
<td>276</td>
<td>209</td>
</tr>
<tr>
<td>Nb deadlocks</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

![State-space graph with FIFO](image1)

![State-space graph with OrderedList](image2)
## Results - LTL Model-checking

<table>
<thead>
<tr>
<th>Property</th>
<th>FIFO (drops)</th>
<th>OrderedList (defers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>![trainIsPassing &amp;&amp; gateIsOpen]</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>![trainIsPassing &amp;&amp; roadSignIsOff]</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>![gateIsClosed -&gt; &lt;&gt; gateIsOpen]</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>![roadSignIsOn -&gt; &lt;&gt; roadSignIsOff]</td>
<td>❌</td>
<td>✔️</td>
</tr>
</tbody>
</table>

✔️: Property verified       ❌: Property violated

## Execution performance

Verification of the 4 properties on a desktop computer\(^1\) in 1.71 seconds

---

\(^1\)Intel® Core\(^\text{TM}\) i7-8550U CPU at 1.80GHz with 4 cores, 16GB DDR4 2400MHz RAM, running a Linux OS
# Table of Contents

1. Introduction
2. Illustrating Example
3. UML Execution Engine
4. Diagnosis Activities
5. Application to the Illustrating Example
6. Conclusion
Conclusion

Our contribution

- Use the same operational semantics implementation for execution and LTL verification
- What is checked during model diagnosis is what is executed at runtime

---

1Preliminary study: https://plug-obp.github.io/experiments/
Conclusion

Our contribution
- Use the same operational semantics implementation for execution and LTL verification
- What is checked during model diagnosis is what is executed at runtime

Limitations
- No support for UML activities
- No evaluation of the resource overhead of the interpreter

1Preliminary study: https://plug-obp.github.io/experiments/
Conclusion

Our contribution

- Use the same operational semantics implementation for execution and LTL verification
- What is checked during model diagnosis is what is executed at runtime

Limitations

- No support for UML activities
- No evaluation of the resource overhead of the interpreter

Perspectives

- Support multiple action languages (e.g., UML activities / Alf)
- Integrate the tool with UML modelers (e.g., Papyrus)
- Apply this approach to other domain-specific languages (e.g., Capella in Eclipse PolarSys)

\[1\] Preliminary study: https://plug-obp.github.io/experiments/
Thank you for your attention
Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov.
Towards one Model Interpreter for Both Design and Deployment.
In 3rd International Workshop on Executable Modeling (EXE 2017), Austin, United States, September 2017.

Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy.
Embedded UML Model Execution to Bridge the Gap Between Design and Runtime.

Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale.
Execution Framework of the GEMOC Studio (Tool Demo).

Asma Charfi Smaoui, Chokri Mraidha, and Pierre Boulet.
An Optimized Compilation of UML State Machines.
In ISORC - 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, Shenzhen, China, April 2012.

Federico Ciccozzi.
Unicomp: A Semantics-aware Model Compiler for Optimised Predictable Software.
Alexandre Duret-Lutz and Denis Poitrenaud.  
Spot: An extensible model checking library using transition-based generalized Büchi automata.  

Andreas Gaiser and Stefan Schwoon.  
Comparison of Algorithms for Checking Emptiness on Büchi Automata.  

Frédéric Jouault, Ciprian Teodorov, Jérôme Delatour, Luka Le Roux, and Philippe Dhaussy.  
Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD.  
Génie logiciel, 109, June 2014.

Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco Pol, Stefan Blom, and Tom Dijk.  
Ltsmin: High-performance language-independent model checking.  

Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraida, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier.  
Papyrus UML: an open source toolset for MDA.  


