Towards one Model Interpreter for Both Design and Deployment

3rd International Workshop on Executable Modeling (EXE 2017)
co-located with MODELS 2017 in Austin, Texas, USA

September 18, 2017

Valentin BESNARD ¹  Matthias BRUN ¹  Philippe DHAUSSY ²
Frédéric JOUAULT ¹  David OLIVIER ³  Ciprian TEODOROV ²

¹TRAME team, ESEO, Angers, France
²Lab-STICC UMR CNRS 6285, ENSTA Bretagne, Brest, France
³Davidson Consulting, Rennes, France
Overview

1 A New Approach for Design and Deployment of UML Models
   - Context
   - Issues
   - Approach
   - Case Study
   - Results

2 Design of the Bare-Metal UML Interpreter
   - Interpreter Design
   - Communication Interface
## Context

### New generation of embedded systems and CPS
- Emergence of new needs
- Connected devices and collaboration on networks (IoT)

### Consequences
- Behavior of systems more uncertain
- Systems more vulnerable to cyber attacks

### Needs
- Simulate, execute, and verify models at early design stage
- Prevent introduction of bugs
Semantic gap: code and diagnosis results difficult to link to the user model (UML Model)

Equivalence: multiple separate definitions of the semantics language not proven equivalent

Diagnosis understandability: results not expressed over UML or code
Root cause of these problems (semantic gap, equivalence, and diagnosis understandability): multiple implementations of UML semantics by transformations towards different formalisms
Our Approach

Key points

- Use of a **single semantics implementation** centralized in a UML model interpreter
  - Avoid multiple implementations of the language semantics by transformations for which we do not know how to prove their equivalence
Our Approach

Solutions

- Semantic gap and equivalence issues: avoided by having only one model
- Diagnosis understandability issue: results directly linked to the UML model
A new issue

A lack of diagnosis tools for this approach that we addressed with an execution control interface (similar to a debugger interface).
Case Study: Level Crossing

Goal

Ensure the safety of all road users during the passage of the train at the intersection of the railroad with the road
Case Study: Level Crossing (Composite Structure Diagram)
Case Study: Level Crossing (State Machines)

RoadSign
- Inactive
  - switchOn / switchOff
    - Active
- Opened
  - close/open
    - Closed
Gate
- NoDetection
  - entranceBegin / send entranceBegin to controller;
  - entranceEnd / send entranceEnd to controller;
  - Detection
TrackCircuit
- Controller
  - Idle
    - entranceBegin / send switchOn to roadSign;
  - TrainDetected
    - approachBegin / send close to gate;
    - Approach
      - approachEnd
      - Passing
        - exitBegin / send open to gate;
        - WaitEndPassing

Deployment

Deployment process

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

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

Deployment process

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

```xml
<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>
```
Deployment

Deployment process

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

```c
UML_Class class__Controller = {
    .c_kind = C_UML_Class,
    .visibility = UML_PUBLIC,
    .name = "Controller",
    .classifierBehavior = (UML_Behavior*)&stateMachine__Controller,
    .isActive = 1
};
```
Deployment

### Deployment process

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

Targets

- PC with a Linux operating system + TCP
Deployment

Targets

- PC with a Linux operating system + TCP
- stm32 on bare-metal + RS232

Frédéric JOUAULT (ESEO)
Deployment

Targets

- PC with a Linux operating system + TCP
- stm32 on bare-metal + RS232
- at91sam7s on bare-metal (microcontroller used by Lego NXT) + RS232 (target used only for simulation)
Simulation

- Connection possible over TCP or RS232 (via UART peripheral)
- Four buttons for the four requests of the communication interface
- Step by step or back-in-time execution available
- **History**: all states encountered are stored
- **Back-in-time execution**: possibility to reload a previous state of the model
State-space exploration

- Use of a breadth first search algorithm
- Level crossing model: 1,825 configurations and 5,793 transitions
## Interpreter Design

### Three components
- **metamodel**: definition of the language semantics
- **model**: representation of the static part of the system
- **interpreter**: representation of the dynamic part of the system and execution support

### Key points
- An interpreter deployable as OS task or process (e.g., Linux) or bare-metal (without OS)
- Each instance of active classes represented as an active object
- Each active object has:
  - An event pool to receive events
  - A current state
  - A store for its attributes
Semantics definition tUML
A subset of Eclipse UML including:
- class diagram
- state machines diagram
- composite structure diagram

Effects and guards
Implemented as OpaqueBehaviors and OpaqueExpressions in a language that enables to:
- send events
- assign values to attributes
Communication Interface

Goal

Solve the lack of specific diagnosis tools by providing a generic API to control remotely the execution of the interpreter.

Four requests

- **Get configuration**: collects the current configuration (memory state) of the interpreter.
- **Set configuration**: loads a configuration as the current memory state of the interpreter.
- **Get fireable transitions**: gets transitions that have their trigger and their guard satisfied in the current state.
- **Fire a transition**: fires a fireable transition of an *ActiveObject*. 

Communication Interface

Possibility to connect existing tools

- No need to implement an ad-hoc toolbox
- Existing tools used and approved for several years
- No formation required for engineers

How to connect a diagnosis tool?

- Implement a TCP client and requests of the communication interface
- Use the connection converter to make the conversion into serial frames
Conclusion

Our contribution

- Use of a single semantics definition to overcome the semantic gap and the equivalence problem between models
- Implementation of a bare-metal UML interpreter
- Definition of a communication interface to enable the use of existing tools and fix the lack of diagnosis toolboxes specific to our interpreter
- Remote control of the model execution with both a simulator and a state-space explorer

Perspectives

- Implementation of formal properties verification
- Connection of this interpreter with a model-checker
- Application of this approach to other languages (e.g., DSLs)