Executing and debugging UML models: an fUML extension

Yoann Laurent, Reda Bendraou, Marie-Pierre Gervais

Executing and Debugging UML Models: an fUML extension

Yoann Laurent
LIP6
UPMC Paris Universitas
France
yoann.laurent@lip6.fr

Reda Bendraou
LIP6
UPMC Paris Universitas
France
reda.bendraou@lip6.fr

Marie-Pierre Gervais
LIP6
University of Paris Ouest
Nanterre
France
marie-pierre.gervais@lip6.fr

ABSTRACT

With the widespread of the Model-Driven Development (MDD) and surfing on the success of the Unified Modeling Language (UML), software development is shifting from being code-centric to model-centric. Models become the key artefacts in the software development process. The success of the project relies on the quality of these models. Early detection of errors by debugging and testing these models is mandatory in order to reduce development cost, ensuring quality and preventing rework at later stages. The fUML standard defines the precise semantics for executing a subset of UML models by defining a virtual machine. The models are then directly executed without transformation. However, the virtual machine is defined to execute the model as an atomic action and does not fulfil the requirements for debugging it. We highlight in this paper the limit of the current specification of fUML (v1.0) and propose an approach for extending the virtual machine with the key functionality that enables debugging of fUML models. A working UML debugger prototype has been implemented and the use and evaluation of the approach are made on a case study.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Monitors*

Keywords

fUML, UML, Debugging, Testing, Execution

1. INTRODUCTION

One major advantage of executable models is that once defined, they can be simulated, debugged and validated in short incremental and iterative cycles. Thus, models will not be considered anymore as a rough and abstract representation of the expected system's structure and behavior, but as the system itself. This makes them a powerful asset in the development of complex and critical systems. In order to achieve such goal, modelers require not only precise and unambiguous languages for modeling both the structural and behavioral properties of the system but also the tooling support for executing, testing and debugging these models. In the current state of the art of modeling languages used in the industry, UML appears to be the de facto standard. However in the actual UML specification, the operational semantics remains unclear, imprecise and ambiguous. The semantics is explained in natural language and dispersed through the specification. Due to this fact, research and commercial tools aiming to execute models have to do some assumptions on the precise operational semantics. As a result, the same models may be executed differently from one tool into another.

To overcome this lack, the OMG released fUML 1.0 (Semantics of a Foundational Subset for Executable UML Models) [21], a new standard that precisely defines the execution semantics for a subset of UML 2.3 in a form of an *Execution Model* implemented in a virtual machine. However, in order to ensure the success of the development [14], models need to be verified at early stages of the development lifecycle. One way to do this is to simulate models by testing and debugging them. Although the simulation does not provide any formal proofs, it can significantly increase the confidence in the model, provide system understanding and give an early direct experience with the system being designed [25]. Moreover, Baker et al. [5] show in their case study a reduction of 30%-70% in the time needed to correctly fix bugs using model simulation.

Debugging is a methodical process of finding and reducing the number of bugs or defects in a program thus making it behave as expected [26]. State of the art for software debugging, testing, and verification in the case of code-centric development is well known in the community [12]. In the case of model-centric development and without the additional step of transforming the model to code, only few works address it [23, 17, 8, 7, 10]. However, they do not use the fUML standard for the execution semantics. Nowadays programming without debugging support seems to be inconceivable. Thus, since fUML is the standard for the execution of UML models, the idea of providing the fUML virtual machine with a debugger comes quite naturally. However, the implementation of the fUML virtual machine is defined to execute the model as an atomic action and does not allow any interaction with it. The only goal of the virtual machine is to execute models in an automatic way, i.e. given a value for the inputs parameters of a behavior, the execution returns values for their outputs. The execution is neither

SAC'13 March 18-22, 2013, Coimbra, Portugal.

controllable nor observable in the current specification of fUML [19, 9]. By controllable, we mean that it is not possible to pause and resume the execution. By observable, we mean that it is not possible to know what happens inside the execution (e.g., be aware when a sub-activity starts or when a variable is modified); monitoring of the execution is not available. Thus, it is not possible to build a debugger using the fUML virtual machine straightforwardly. In order to obtain a wider adoption of model execution in the industry, a tool based on a standard which provides correct debugging support is strongly needed.

In this paper we present a critical analysis on the newly defined fUML standard and we address its lacks in terms of controllability and observability, key issues for model execution and debug. An approach is proposed in order to extend the standard with the set of concepts and facilities required to debug models. The paper is organised as follows. Section 2 presents the fUML standard and a critical analysis of the current specification. In Section 3, we propose an extension of the fUML Execution Model enabling controllability and observability of the execution, and we detail the modifications needed in order to incorporate the extension in the Execution Model. Section 4 presents our implemented prototype allowing to test and debug models. The evaluation of the extension and the prototype is presented in Section 5. Finally, related work is addressed in Section 6 and Section 7 concludes by sketching some future perspectives of this work.

2. FUML

fUML is an OMG standard that precisely defines the execution semantics of a subset of UML 2.3. The standard defines a basic virtual machine (in the form of pseudo Java-code) enabling compliant fUML models (i.e., UML models using only elements comprised in the fUML subset) to be executed. It can be decomposed in three principle parts (i) the abstract syntax represented by a subset of UML, mainly composed by the `Class Diagram` and most of the `Activity Diagram`; (ii) the Execution Model which defines the execution semantics of the abstract syntax and (iii) the model library which defines primitive types and behaviors. In this section we present these three parts and then we highlight the current limitations of the current specification.

2.1 Abstract syntax

The abstract syntax of fUML is structured in the same way as UML but being a subset of the later, not all packages are included. For modeling the structure, `Classes` are included to deal with the basic modeling concepts and describe the classes of a system, their attributes, operations and relationships among each other. For modeling the behavior, `Actions`, `Activities` and `CommonBehaviors` are partially included. Indeed, some elements with a too high degree of abstraction are excluded from the specification. Thus, the `Activity Diagram` describes how the actions in the system are executed using control and data flow between the actions.

2.2 Execution model

The Execution Model is itself a model, written in fUML, that specifies how fUML models are executed. The execution semantics adopted by fUML is quite similar to Coloured Petri Nets [15] and is based on the principle of offering and consuming object or control tokens between the different activity's constituents (i.e., Activity Nodes and Activity Edges).

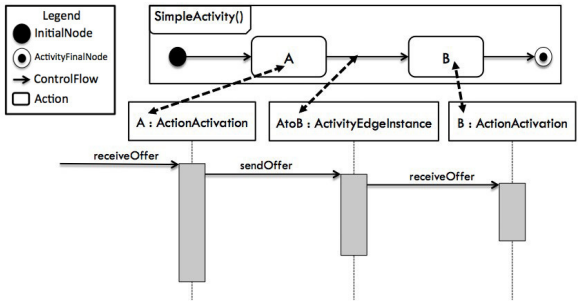

Figure 1: Regular call between semantic elements

To illustrate this, Figure 1 shows a simple activity diagram represented with one `InitialNode`, two `Action` nodes and an `ActivityFinalNode`. Each of these nodes are connected with a `ControlFlow` edge which represents that a node starts after the previous one is finished. The sequence diagram shows the corresponding calls between the nodes in the Execution Model. The diagram is a simplified version of what really happens during the execution and focuses on the interaction between elements. `ActionActivation` and `ActivityEdgeInstance` are the instantiation of the corresponding abstract syntax (see below for the explanation of the instantiation in the Execution Model using the visitor pattern). When the fUML virtual machine invokes this activity, it starts by inserting a token in each `InitialNode` and other nodes with no inputs `ControlFlow`. Then, the nodes with a token (i.e., the `InitialNode` in our example) *fire* (i.e., execute their own behavior) and *sendOffer* on each of their outputs `ControlFlow`. The `ControlFlow` is then able to call on its target node `A` to *receiveOffer*. When the node `A` receives an offer, if first checks if the prerequisites for its execution have been satisfied as determined by *isReady*, if yes, takes the offered tokens from inputs control flow and *fire*. At the end of the *fire* operation, the node directly *sendOffer* on its outputs `ControlFlow`. The execution of an activity is then an extended chain of *sendOffer-receiveOffer-fire-sendOffer* calls between the activity constituents. Each abstract syntax element of an activity diagram has its own semantics. For example, a `DecisionNode` will offer a token only on one of its output edges determined during its *fire* execution. Although the example looks very simple, in order to execute, many actions have to be carried out.

The Execution Model contains a package called `Loci` (Figure 2), which contains the `Locus`, `Executor`, and `ExecutionFactory` classes that model a fUML execution engine and its environment. The `Executor` provides the root abstraction for executing a fUML model. It provides the basic interface for evaluating value specifications and executing behaviors (e.g., Activity). Every execution takes place at a specific `Locus`. A `Locus` can be seen as an abstraction of a physical or virtual computer capable of executing fUML models. Objects and links created during or before an execution are associated to it. The Execution Model is based on the Visitor pattern [11]. Using this pattern, each abstract syntax metaclass has a corresponding visitor class in the Execution Model (named `*Execution` and `*Activation`). In this

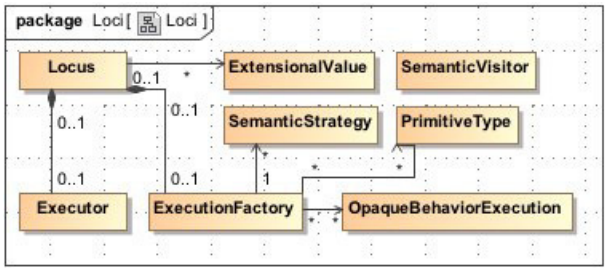

Figure 2: Abstract of the Loci package

paper, we call semantic elements the corresponding visitor class of the abstract syntax. All visitor classes in the Execution Model are descendent, directly or indirectly from the root `SemanticVisitor` class. To create a corresponding visitor instance, the `Executor` uses an instance of the `ExecutionFactory` class located at the execution `Locus`.

The `ExecutionFactory` maintains the set of primitive behaviors available to be called. In fUML, primitive behaviors are defined syntactically as instances of `OpaqueBehavior`. For each `OpaqueBehavior` instance representing a primitive behavior, the execution factory maintains a corresponding prototype instance of `OpaqueBehaviorExecution`. The `ExecutionFactory` contains also some instances of `SemanticStrategy` to deal with the semantic variation points of UML (e.g., event dispatch scheduling).

To configure the execution environment, it is necessary to instantiate a set of collaborating objects from classes within the Execution Model that provide the initial execution environment.

2.3 Limitations

The major limitation of the Execution Model comes from the fact that the execution is not controllable. The `Executor` only provides few operations to evaluate a value specification and to execute synchronously or asynchronously a behavior. Then, given a value for its input parameters, the execution returns values for its output behaviors. As explained in Section 2.2 and shown on Figure 1, the execution is computed directly through the chain of calls between the semantic elements. No interaction or insight about what happens is provided. The execution of a behavior can be seen as an atomic operation in the Execution Model since no interactions with the execution are provided.

Another limitation comes from the fact that the Execution Model is not observable. During the execution the only information available is some printing from each semantic elements of the Execution Model (e.g., inside the *fire* method of the `ActionActivation`, the printing corresponds to "fire node *nodeName*"). Obviously this printing is useful for debugging the implementation of the fUML Execution Model but is not suitable to observe the execution. It is not possible to set listeners on elements (e.g., be aware when a sub `Activity` starts), or on objects instantiated on the `Locus` (e.g., be aware when an attribute of an object is modified). More generally, monitoring the execution is not possible. We are only aware when behavior starts and when it finishes. These limitations prevent to link the execution to a prototyping user interface to understand the end-user use cases and to find interaction bottlenecks.

fUML also lacks of some generic extension mechanisms

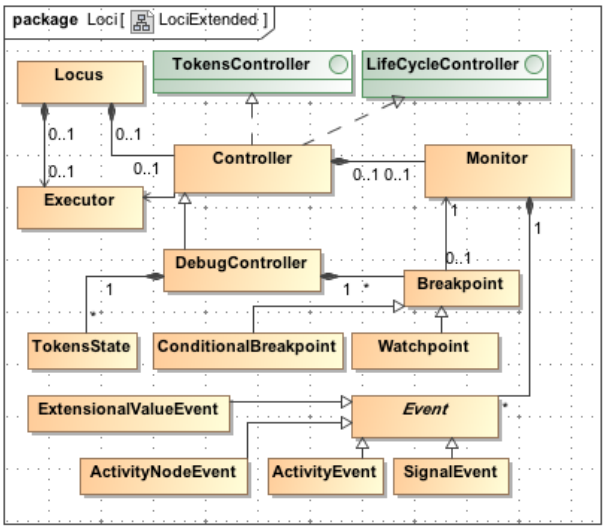

Figure 3: Outline of the fUML extension

for customizing the semantics of modeling concepts. For example, the use of UML profiles is not addressed in the standard.

Moreover, as highlighted in the specification, there are some semantics areas which are not explicitly constrained by the Execution Model: (i) the execution is agnostic about time, both discrete and continuous time can be used; (ii) the semantics of concurrency includes an implicit concept of concurrent threading, the execution tool only needs to respect the various creation, termination, and synchronization constraints of the model (i.e., a `ForkNode` in the model does not imply to run multiple threads); (iii) the semantics of inter-object communications mechanisms assumes that all communications are perfectly reliable and deterministic. The result is that some execution tools can still be conformant to the semantics specified by the Execution Model for fUML and semantically vary in the above areas while executing the same model. This lack of constraints is an important problem of fUML since it breaks the interoperability of tools, the same model may not be executed in the same way on different tools.

The purpose of this work is not to address all of these limitations but to focus on the limitations which prevent to debug models using the fUML virtual machine.

3. FUML EXTENSION

This section presents our extension to fUML in order to control and to observe the Execution Model at runtime.

3.1 Tokens and Lifecycle Controller

Section 2.3 shows that the fUML execution is considered as an atomic action with no control over it. To bring controllability in the Execution Model, we propose to add to the `Locus` a new class `Controller` allowing to control the execution (Figure 3).

The key idea is to redirect the communication call from the semantic elements to the `Controller`. The communication corresponds to the methods which offer and consume objects and control tokens (e.g, `receiveOffer`, `sendOffers`...). Figure 4 shows the call of methods using a con-

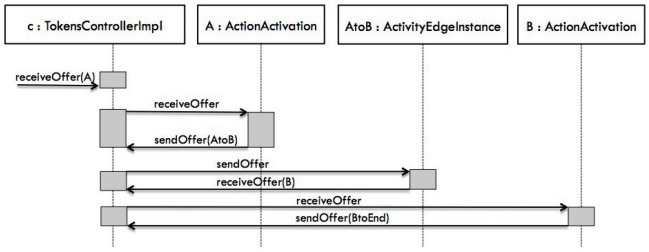

Figure 4: Modified call between semantic elements

troller. When the action `A` finishes, instead of directly do `sendOffer` to their outputs edges, the node `A` sends this information to the controller. The controller is then able to transmit the information itself to the edge. The implementation of the controller on Figure 4 mimics the normal Execution Model and just transmits the information. Using a controller gives many benefits. By interfacing the controller with a graphical interface, it is possible to control the execution in a finer grain. For instance, blocking the execution on each `receiveOffer` method corresponds to setting a breakpoint on each node before its execution. The controller is then able to give insights about what happens during the execution of the behavior. The `TokensController` interface contains the methods which consume and offer tokens between the activity constituents.

By taking advantage of the `TokensController`, it is possible to add some lifecycle control to the execution by temporally blocking the transmission of tokens. The `LifeCycleController` interface defines methods to start, stop, pause and resume the execution.

The `Controller` implements both `TokensController` and `LifeCycleController` and keeps a reference to the `Executor` to be able to run parts of the model without explicitly writing entry points (e.g, writing a "main" activity to test a sub-activity).

The `Controller` owns a `Monitor` which keeps records of all received `Event` during the execution. `ActivityEvent` and `ActivityNodeEvent` correspond to the entry/exit point of an activity and node. `SignalEvent` happens when signals are sent to the event pools (e.g, when a node `SendSignalAction` is executed). `ExtensionalValueEvent` corresponds to the execution of a node from the `IntermediateActions` of the `Actions` package (e.g, `CreateObjectAction`, `ReadStructuralFeatureAction`, `WriteStructuralFeatureAction`...) ; these actions create, modify, read and delete objects or links instantiated at the `Locus` (Figure 2). By using the `Monitor`, it is possible to observe the execution by listening to `Event` happening in the Execution Model.

The Java Virtual Machine proposes a Tool Interface (JVM TI) which provides both a way to inspect the state and to control the execution of running applications. It supports the full breadth of tools that need access to the JVM state. We argue that the fUML virtual machine should empower the use of both the `Controller` and `Monitor` concepts inside the Execution Model to bring such flexibility, controllability and observability during the execution. Thus, the `Controller` answers the lack of controllability while the `Monitor` handles the need of observability.

3.2 Debug Controller

In a debugger, setting breakpoints corresponds to an intentional stopping or pausing place in a program. More generally, putting a breakpoint on the model is a means of acquiring knowledge about the execution. During the interruption, the modeler can inspect the environment to find out whether the executed model is working as expected. A watchpoint corresponds to a specific breakpoint, such that the reading, writing, or modification of a specific variable triggers it. The `DebugController` which extends the `Controller` handles a list of `Breakpoint`. By using the `Monitor` and comparing the `Event` happening during the execution with the `Breakpoint` signature, the `Controller` can check if the `Breakpoint` holds and then pause the execution. `Breakpoint` and `ConditionalBreakpoint` are set on the nodes of an activity while `Watchpoint` are set on instantiated objects and links.

`DebugController` defines methods directly related to the need of a debugger. One important point to effectively debug is to be able to roll-back the execution. Since the execution formalism is based on the Coloured Petri Nets, tokens and their contents on the system represent the actual execution state. The `DebugController` saves information about the position and the contents of the tokens (at every change or under certain conditions in order to keep memory efficiency in the case of big models) in a `TokensState`. The `DebugController` is then able to roll-back the execution by swapping all the tokens with an old `TokensState`. The `DebugController` proposes also to change the `SemanticStrategy` (the semantic variation points of UML) and the value of objects and links instantiated in the `Locus` dynamically during the execution. By rolling-back the execution and changing the semantic variation points or/and the value of objects and links, it is possible to re-execute a part of the model and to compare the obtained results.

Additionally, the `DebugController` defines methods to dynamically insert, move and delete tokens from nodes. Instead of following the normal execution defined by the workflow of the activity (i.e., tokens offered and consumed by the activity's constituents), these methods allow to directly jump at different locations in the activity by moving tokens.

3.3 Modification of the Execution Model

Section 3.1 and 3.2 present the extension of the fUML Execution Model. In order to incorporate this extension in the Execution Model, two minor modifications are needed : (i) using the `Controller`, we need to change in the semantic elements every call of method on which tokens transit (e.g, `receiveOffer`) to the equivalent method in the `Controller` (`TokensController` interface); (ii) concerning the monitoring of `Event`, in the same way that the Execution Model contains some printing in each semantic elements, we need to add some calls to the `Monitor` to notify which events are happening (e.g, the execution of a `ReadStructuralFeatureAction` node will trigger the occurrence of an `ExtensionalValueEvent` on the `Monitor`). The drawback of these modifications is that they require to modify many semantic elements, even if the modification is trivial.

Another approach is to use aspect-oriented programming (AOP) [16], by setting an around advice on every call of methods on which tokens transit, it is possible to bypass the call and to redirect it to the `Controller` [13]. Moreover, it is possible to weave aspects before or after the execution of methods in order to trigger a specific `Event` to the `Monitor`.

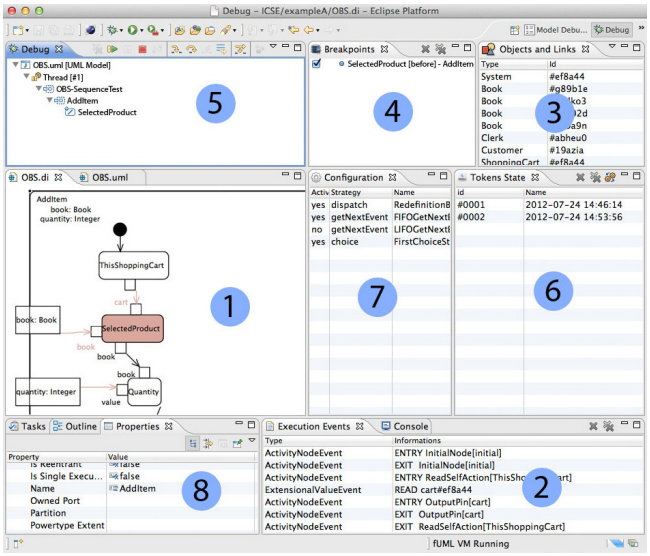

Figure 5: Prototype of the UML debugger using the fUML virtual machine

The major advantage of this approach is that it does not require to modify the current specification of fUML. The aspects are woven in the Execution Model at runtime.

Then, we propose the two solutions to modify the Execution Model in order to incorporate the controller. The AOP solution is clearly a better way since it does not require to modify the actual Execution Model but only extending it with aspects. It is worth noticing that our extension of fUML does not modify the normal semantics execution of fUML, i.e. with or without our extension, an fUML model will be executed the same way (see Section 5.1).

4. PROTOTYPE

The prototype we developed is currently provided as an Eclipse EMF plugin. It accepts UML models in the form of XMI files as input. It uses the fUML Reference Implementation [27] with the extension proposed in Section 3 and uses the Papyrus plugin [1] for the graphical representations of the model. We have chosen the AOP approach from Section 3.3 in order to extend the fUML Execution Model. The intent of this prototype is to assist UML modelers by simulating and debugging their models using the operational semantics of the fUML virtual machine. The models are directly interpreted without an intermediate transformation step.

Figure 5 shows a screenshot of the prototype executing a test of the Online Book Store (OBS), taken from [20]. The prototype follows the concepts of the Eclipse Java debugger and offers equivalent views for rich UML-debugging capabilities. A description of each view is provided below.

Model (label 1) It displays the model being executed using both the Eclipse UML2 tree editor plugin and the Papyrus plugin. Both plugins are extended to show information and the tokens present in the activity diagram. The node ready to be executed are highlighted on the diagram. By selecting nodes, it is possible to add breakpoints. It allows also to directly run an activity from the model without explicitly writing entry points. If the activity contains input parameters, the plugin will ask the user to enter them.

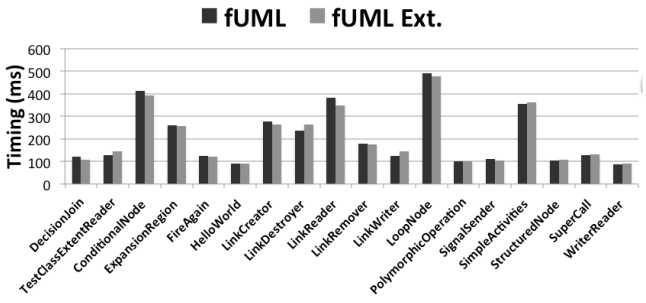

Figure 6: Timing execution comparison between fUML and fUML with our extension

Execution Events (label 2) It shows information about the events happening during the execution. It represents the detailed trace-log of the simulation.

Objects and Links (label 3) It shows all the created objects and links. By selecting an item in the view, it is possible to set a watchpoint on it or see attributes values in the `Properties` view. It is also possible to manually create, remove and modify these values.

Breakpoints (label 4) It displays all the breakpoints and watchpoints currently set during the execution. It is possible to remove and disable breakpoints.

Debug (label 5) It shows the current call stack, displaying the nodes ready to be executed in a tree of the called sub-activities. The debugger supports both breakpoint based and stepwise debugging. Multiple actions are available to control the execution such as resume, suspend, terminate, step into, step over and step return.

Tokens State (label 6) It shows all the saved `TokensState`. The `TokensState` are saved manually. By selecting a `TokensState` in the list, it is possible to roll-back the execution to this precedent state.

Configuration (label 7) It displays the available and the used semantic variation points of the Execution Model. It allows to change the variation points before or during the execution.

Properties (label 8) It displays information about the selected item from other views.

5. EVALUATION

This section presents the evaluation of our fUML extension by first, comparing our prototype with other debuggers from the literature and finally, by testing it with a case study.

5.1 The extension of the Execution Model

We empirically evaluated our extension on 30 fUML models from the fUML Reference Implementation [27]. The goal of these models is to test the implementation of fUML. The size of the models is varying, ranging from few nodes to 50 nodes. The purpose of this evaluation is twofold: (i) to verify that our extension does not break the semantic execution of fUML and (ii) to evaluate the additional workload added by our extension.

We run these models using both the fUML Reference Implementation and through our prototype without setting any

breakpoints (on a MacBook Air 2011 with the Intel Core i5 processor and 4 GB of RAM). We obtained the same outputs from both executions assuring us that the semantic execution remains the same. Figure 6 shows the timing execution with and without our extension. We take the average timing of 100 executions on each model. Due to the CPU's scheduling and the use of small models, the timing execution of the same model can differ from one execution into another. It can be difficult to measure the difference using the extension. Thus, the measured timing execution remains close due to the fact that our extension only adds a lightweight workload by means of additional calls from and to the controller.

5.2 Debuggers requirements

Following the state-of-art of debugging features [24, 22] and by testing many of the most important language-debuggers available nowadays (i.e., debuggers for C, C#, Java ...), we synthesize the most important features that every debuggers should support:

Controlling controlling the execution of the program (i.e., step-by-step, pause, resume...).

Breakpoints pausing the program at a specified place.

Conditional Breakpoints breakpoints that are triggered when reached only if a particular condition is true.

Watchpoints "data" breakpoints that are triggered when a variable is read or written.

Visualization visualizing graphically or textually the current position in the program.

State-reading the ability to inspect variables of the running program.

State-modifying the ability to modify the variables of the running program.

Trace-log the ability to log what happens during the execution.

Jumping the ability to continue the execution at a different location in the program.

Reverse-execution the ability to roll back the execution to a previous program state.

Hot code replace the ability to modify some parts of the code without restarting the debugging from the start.

However, debugging models and code is quite different. Debugging models happens at a higher level of abstraction and implies some conceptual differences. For example, instead of visualizing threads and operation call stack, the visualization corresponds to the activity diagram with tokens and offers positions. Variable corresponds to the objects and links instantiated in the `Locus` and so on. Moreover, UML is full of *semantic variation points* which explicitly identify the areas where the semantics are intentionally under specified to provide leeway for domain-specific refinements. The debuggers have to consider the choice of these semantic variation points. One particularity of the models over the code is that some aspects of a system can be modeled very precisely whereas others parts remain imprecise leading to incomplete models [18]. Though, models still need to be debugged at early stage to ensure quality.

Thus, the debuggers have to propose views adapted to models and support also some additional features related to the model world:

Incomplete models the ability to debug incomplete models.

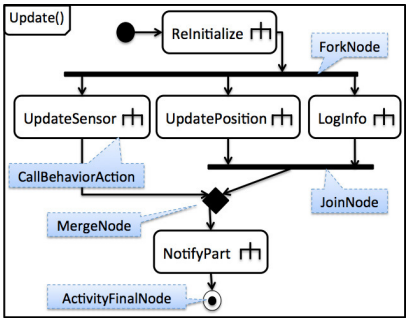

Figure 7: Excerpt of the *Update* activity

Semantic variation points the ability to choose and change dynamically the variation points.

Table 1 shows these requirements handled by our prototype (column 1). Two requirements which imply deeper change of the architecture of the virtual machine remains open. As currently defined by the OMG, the Execution Model attempts to execute valid and syntactically verified models as inputs [21]. However, these assumptions are not always satisfied when debugging *incomplete models*, which results in errors during the instantiation (e.g., *NullPointerException*) in the virtual machine. Concerning the *hot code replace* feature (e.g., modifying the sequencing of actions of an activity at runtime), early experimentation reveals that it is possible, but still requires a heavy extension of the Execution Model. Due to space limitation, this high-level feature is not presented here.

5.3 Case study

The use of our prototype has been evaluated on a simple model. This model has been chosen because, it is representative of a case in which the use of a debuggers is particularly useful.

Figure 7 shows the *Update* activity from a robotic system. This activity is in charge of updating the position of the robot and adapting the sensor accordingly. The activity starts by some re-initialization by calling *ReInitialize*, then starts in parallel using a `ForkNode` three sub-activities (`CallBehaviorAction`): (i) *UpdateSensor* updates the sensor for adapting the move of the robot; (ii) *UpdatePosition* updates the position of the robot on the map; (iii) *LogInfo* logs the current information. The `JoinNode` is present to synchronize the execution of *UpdatePosition* and *LogInfo*. When both calls are done, all the input flows of the `JoinNode` are offered, then the `JoinNode` can fire. The `MergeNode` can fire when one of its inputs is offered. Before the activity finishes, *NotifyPart* is called to notify the other parts of the systems that the position has been updated.

Using our prototype and executing this activity stepwise, the modeler realizes that sometimes it does not execute the activity as expected. By rolling back the execution and executing multiple times this activity, it happens that many unintended execution paths exist in the diagram. Table 2 shows some of the execution paths starting after the `ForkNode` to the execution of the `ActivityFinalNode`. Only the `CallBehaviorAction` are presented on the table. There are two issues : (i) *NotifyPart* may be fired 2 times (e.g., execution paths 3 and 6); (ii) *UpdateSensor* or *UpdatePosition* and *LogInfo* may be never fired (e.g., execution paths 1 and

	Academic			*Commercial*		
	Our prototype	Dotan et al.	Populo	Cameo Simulation Toolkit	AM|USE 2.0	Eclipse Java Debugger
Generic features						
Controlling	**yes**	yes	yes	yes	yes	yes
Visualization	**callstack**, **graphical**	callstack, graphical	callstack, textual	callstack, graphical	graphical	callstack, textual
Breakpoints	**yes**	yes	yes	yes	yes	yes
Cond. Breakpoints	**yes**	no	no	yes	no	yes
Watchpoints	**yes**	no	no	no	no	yes
Trace-log	**yes**	yes	yes	yes	yes	no
State-reading	**yes**	yes	yes	yes	yes	yes
State-modifying	**yes**	no	no	no	no	yes
Jumping	**yes**	no	no	no	no	no
Hot code replace	**no**	no	no	no	no	yes
Reverse-execution	**yes**	no	no	no	no	no
Models features						
fUML semantics	**yes**	no	no	yes	yes	-
Incomplete models	**no**	partial	no	no	no	-
semantic variation points	**yes**	no	partial	no	no	-

Table 1: Comparison of UML Debuggers using direct simulation

N.	Intended?	Execution path
1	no	*UpdateSensor→NotifyPart*
2	yes	*UpdateSensor→UpdatePosition→LogInfo→NotifyPart*
3	no	*UpdateSensor→LogInfo→UpdatePosition→NotifyPart→NotifyPart*
4	no	*UpdatePosition→LogInfo→NotifyPart*
5	yes	*UpdatePosition→UpdateSensor→LogInfo→NotifyPart*
6	no	*UpdatePosition→UpdateSensor→LogInfo→NotifyPart→NotifyPart*
7	...	...

Table 2: Some execution paths of the *Update* activity

4). This problem is known as a *lack of synchronization*, i.e. joining the concurrent fork path with a merge structure. A lack of synchronization at a merge structure results in unintentional multiple activations of nodes that follow the merge node. Moreover, a lack of knowledge from the modeler concerning the semantics of the `ActivityFinalNode` leads to the second issue. Indeed, when a `ActivityFinalNode` fires, the whole activity terminates even if some control tokens remain on the diagram. For example, the execution path 1 shows that the `ActivityFinalNode` has been fired before *UpdatePosition* and *LogInfo* receive their offer (*receiveOffer*) from the `ForkNode`. It can be difficult for a naive modeler to understand why the activity does not work properly without the debugging capabilities.

The modeler corrects the problem by removing the `MergeNode` and changing the target of the output `ControlFlow` of *UpdateSensor* to the `JoinNode`. The `JoinNode` can then synchronize the execution of *UpdateSensor*, *UpdatePosition* and *LogInfo* before continuing to *NotifyPart*. The modeler also sets a breakpoint on the *Update* activity and re-starts the debugging to directly test this part of the system.

Using our prototype, the modelers can get a better understanding on how fUML is executing their models, and solve some conceptual and architectural issues in their models. Both novices and experienced modelers can benefit from the debugging functionality, helping novices to identify improper use of modeling constructs and experienced modelers to better understand the designs that other modelers create.

6. RELATED WORK

There are many research and commercial tools enabling the execution of UML models. The execution approach can be divided into two fields: the transformations of the models into executable code such as *Executable UML* introduced by Mellor et al. [20], and the direct interpretation. Due to space limitations, we only present the work based on the direct interpretation.

The first work was from Riehle et al. [23] which present the architecture of a virtual machine for UML that interprets directly the model without transformation steps. Dotan et al. [17, 8] present a UML debugger based on their own generic model execution engine. The debugger proposes to execute the model stepwise, set breakpoints, inspect variables, and visualize the execution. Moreover, it supports the execution of incomplete models, i.e. when there is missing information regarding the next step, the debugger asks the user what do to. Crane et al. [7] propose an interpreter for UML 2 Actions and Activities. This work was done after the refinements of the activity package in the second version of UML, when the semantics changes to the Petri-Net one. The interpreter offers analysis capabilities, random execution, reachability properties, assertion and deadlock checking. Fuentes et al. [10] present Populo, a UML model debugger compliant with the UML 2 standard for executing models. The particularity of Populo is that the UML action language can be customised to meet the needs of a specific profile.

However, these works are not based on the new fUML standard and have done some assumptions on the precise operational semantics which creates tool-interoperability and extensibility problems. Moreover, the semantics richness of these approaches are less complete than fUML, many simplifications have been done and the semantic variation points are never addressed. Some features such as the roll-back support and the configuration of the semantic variation points are only supported by our approach.

Mayerhofer [19] explains some limitations of fUML but does not provide any solution to it. They present their research direction that has given rise to the *moliz* project [3] which aims to implement a novel model execution environment based on fUML.

Recently, some commercial UML tools have implemented fUML in their own environment enabling execution and basic debugging capabilities (e.g., The Cameo Simulation Toolkit for the MagicDraw tool and AM|USE 2.0 by LieberLieber for the Enterprise Architect tool). They offer some non-standard functionalities such as the support of state machine (W3C SCXML standard) and multiple action languages (e.g., Javascript, Ruby, Python...). These tools are compared to our prototype on Table 1.

Concerning the approach which extend the fUML stan-

dard, Ellner et al. have also observed the lack of "human interaction" (i.e., controllability) while using fUML to enact software process models [9]. The authors propose some extensions to support distributed execution, suspending and resuming execution on different nodes and a request extension to interact with the fUML virtual machine. Benyahia et al. [6] present a lightweight modification of fUML to cope with the need of real-time systems concerning the scheduling and concurrency within the Execution Model. They introduce a `Scheduler` class to dispatch actions and therefore break the sequential execution of fUML.

Abdelhalim et al. [4] present an approach to represent an fUML models into the process algebraic specification language CSP (Communicating Sequential Processes) and use the FDR (Failures-Divergences Refinement) model-checker to check if the model is deadlock free. When a deadlock is found, a counter-example trace which led to the deadlock is generated. The verification of the models is done through automatic model-checking instead of simulation as we promote it in our approach.

7. CONCLUSIONS AND FUTURE WORK

This paper presented an approach which extends the fUML Execution Model with a controller to enable the implementation of a debugger that takes advantage of the fUML virtual machine. The approach covers the major features of code-based debuggers such as stepwise execution, setting of breakpoints/watchpoints, inspection and manipulation of values, trace-logs, roll-back, and interactive visualisation of the execution.

The developed prototype allows to directly simulate UML models to gain system understanding, detect errors, and check whether the system conforms to engineer expectation and user requirements. Moreover, it can significantly reduce development cost and time by ensuring quality and preventing rework at later stages.

It is worth noticing that our prototype is modeling-tool independent and that the proposed extensions of fUML do not change the regular execution semantics.

The new controller is not only useful for debugging purposes, it also brings more flexibility, controllability and observability to the Execution Model raising the possibility of using the operational semantics of fUML to more areas (e.g, using fUML as an engine to execute processes and workflow). Moreover, due to the fact that generally software engineers lack of skills in formal techniques, we investigate the use of a Model Checker in our prototype to analyse automatically the behaviors of activities in order to detect deadlocks, livelocks and to calculate reachability coverage.

8. ACKNOWLEDGMENTS

The author's work is supported by the Galaxy (ANR - French National) project [2].

9. REFERENCES

[1] Papyrus. `http://www.eclipse.org/papyrus`.

[2] Galaxy project. `http://galaxy.lip6.fr/`, 2009.

[3] moliz. `http://www.modelexecution.org/`, 2012.

[4] I. Abdelhalim, J. Sharp, S. Schneider, and H. Treharne. Formal verification of tokeneer behaviours modelled in fuml using csp. *Formal Methods and Software Engineering*, pages 371–387, 2010.

[5] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context-Motorola case study. *Model Driven Engineering Languages and Systems*, pages 476–491, 2005.

[6] A. Benyahia, A. Cuccuru, S. Taha, F. Terrier, F. Boulanger, and S. Gérard. Extending the standard execution model of UML for real-time systems. *Distributed, Parallel and Biologically Inspired Systems*, pages 43–54, 2010.

[7] M. Crane and J. Dingel. Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 8. ACM, 2008.

[8] D. Dotan and A. Kirshin. Debugging and testing behavioral UML models. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 838–839. ACM, 2007.

[9] R. Ellner, S. Al-Hilank, J. Drexler, M. Jung, D. Kips, and M. Philippsen. A FUML-based distributed execution machine for enacting software process models. In *ECMFA*, pages 19–34, 2011.

[10] L. Fuentes, J. Manrique, and P. Sánchez. Execution and simulation of (profiled) UML models using Pópulo. In *Proceedings of the 2008 international workshop on Models in software engineering*. ACM, 2008.

[11] E. Gamma. *Design patterns: elements of reusable object-oriented software*. 1995.

[12] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 2002.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *ACM Sigplan Notices*, volume 37, pages 161–173. ACM, 2002.

[14] W. Humphrey. Managing the software process. *Reading/MA*, 1989.

[15] K. Jensen. Coloured petri nets. *Petri nets: central models and their properties*, pages 248–299, 1987.

[16] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es):154, 1996.

[17] A. Kirshin, D. Dotan, and A. Hartman. A UML simulator based on a generic model execution engine. *Lecture notes in computer science*, 4364:324, 2007.

[18] A. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[19] T. Mayerhofer. Testing and debugging UML models based on fUML. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012.

[20] S. Mellor, M. Balcer, and I. Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures*. 2002.

[21] OMG. Semantics of a foundational subset for executable uml models (FUML) version 1.0. `http://www.omg.org/spec/FUML/`, 2011.

[22] N. Papoylias. High-level debugging facilities and interfaces: Design and developement of a debug-oriented ide. *Open Source Software: New Horizons*, pages 373–379, 2010.

[23] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a UML virtual machine. *ACM SIGPLAN Notices*, 2001.

[24] J. Rosenberg. *How debuggers work: algorithms, data structures, and architecture*. John Wiley & Sons, Inc., 1996.

[25] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.

[26] E. Shapiro. Algorithmic program debugging. *Dissertation Abstracts International Part B: Science and Engineering,*, 43(5):1982, 1982.

[27] The ModelDriven Community. Foundational UML reference implementation. `http://portal.modeldriven.org/project/foundationalUML`, 2007.