Extending the Standard Execution Model of UML for Real-Time Systems

Abderraouf Benyahia[1,2], Arnaud Cuccuru[2], Safouan Taha[1], François Terrier[2], Frédéric Boulanger[1], and Sébastien Gérard[2]

[1] SUPELEC Systems Sciences (E3S) Computer Science Department, 91192 Gif-Sur-Yvette cedex, France
[2] CEA LIST, 91191 Gif-Sur-Yvette cedex, France
{abderraouf.benyahia,safouan.taha,frederic.boulanger}@supelec.fr,
{arnaud.cuccuru,francois.terrier,sebastien.gerard}@cea.fr

Abstract. The ongoing OMG standard on the "Semantics of a Foundational Subset for Executable UML Models" identifies a subset of UML (called fUML, for Foundational UML), for which it defines a general-purpose execution model. This execution model therefore captures an executable semantics for fUML, providing an unambiguous basis for various kinds of model-based exploitations (model transformation, code generation, analysis, simulation, debugging etc.). This kind of facility is of great interest for the domain of real time systems, where analysis of system behavior is very sensible. One may therefore wonder if the general-purpose execution model of fUML can be used to reflect execution semantics concerns of real-time systems (e.g., concurrency, synchronization, and scheduling.). It would practically mean that it is possible to leverage on this precise semantic foundation (and all the work that its definition implied) to capture the precise execution semantics of real-time systems. In this paper, we show that this approach is not directly feasible, because of the way concurrency and asynchronous communications are actually handled in the fUML execution model. However, we show that introducing support for these aspects is technically feasible and reasonable in terms of effort and we propose lightweight modifications of the Execution model to illustrate our purpose.

Keywords: fUML, MDD, Model Simulation, Concurrent systems, Real-time systems.

1 Introduction

Profiles are the default UML extension mechanism for tailoring UML2 to specific application domains, from both syntactic and semantic terms. Extending UML2 syntax is well achieved, with explicit stereotype definitions capturing the syntactic extensions. Unfortunately, the semantic extensions (potentially implied by a profile) have not yet reached a similar degree of formalization. They usually take the form of a natural language description, just like the semantic description of the UML2 metamodel. The informal nature of this description leaves the door open to several (potentially contradictory) interpretations of a given model and does not lend itself to unambiguous model-based exploitations. This is particularly critical when considering

M. Hinchey et al. (Eds.): DIPES/BICC 2010, IFIP AICT 329, pp. 43–54, 2010.

complex notions such as time and concurrency, which are central issues to the design of real-time and embedded software.

Things should however evolve with the ongoing OMG standard on the semantics of a foundational subset for executable UML models [2]. This standard indeed defines a formal operational semantics for a subset of UML2 called fUML (foundational UML). The operational semantics of fUML takes the form of an executable UML model called "Execution Model" (that is to say, a UML model defined with elements from the fUML subset[1]), which is precise enough to be considered as an interpreter for fUML models. While foundational, this subset includes non-trivial mechanisms carrying concurrent and asynchronous execution semantics, such as active objects (i.e., objects with their own execution thread) and asynchronous communications via signal passing. These notions are essential when considering concurrent real-time systems, such as in the MARTE profile [1] (Modeling and Analysis of Real-Time and Embedded systems) and in particular in its HLAM sub-profile (High Level Application Modeling), which provides support for designing concurrent real-time systems with extensions inspired by the concept of real-time active object [4][5][6].

Our long term objective is to reflect timed and concurrent execution semantics as introduced in HLAM by extending the general-purpose Execution Model of fUML. Ideally, this extension would first rely on fUML mechanisms for concurrency and asynchronous communications, and then add support for time. This extended Execution Model would typically provide support for model-based simulation, a design technique that has proven useful for rapid prototyping of real-time and embedded systems [7][8].

While the rationale for this approach sounds quite obvious, we believe that it cannot be directly put into practice. Our main obstacle concerns the way concurrency (i.e., active objects) and asynchronous communications (i.e., via signals) are actually supported. While the fUML specification of Execution Model leaves the door open to support some slightly different execution paradigms by including a few explicit semantics variation points (section *8.2.2* of [2]), no key variation points are defined regarding concurrency and asynchronous communications. Furthermore, the Execution Model does not identify an explicit entity responsible (such as scheduler) for the management of concurrent entities. In order to properly handle these aspects, some modifications are needed in the Execution model. The main contribution of this article is to propose such lightweight modifications. These propositions can be considered as a first step towards our long-term objective: reflecting the execution semantics of real-time systems by specializing the fUML execution model.

In section 2, we start by highlighting fUML limitations. In section 3, we discuss works related to model-based simulation of concurrent systems. We show how principles underlying these approaches could be integrated in the standard Execution Model of UML. In section 4 we propose a modification of the Execution Model, which mainly consists in introducing an explicit scheduler. Section 5 then concludes this article and sets guidelines for future research.

[1] In order to break circularity, some of the fUML elements have a formal axiomatic description.

2 Limitations of fUML Regarding Support for Concurrency and Asynchronous Communications

As explained in the introduction to this article, fUML [2] formalizes the execution semantics of a subset of the UML2 metamodel. Particularly, this subset contains mechanisms for the description of concurrent systems (i.e., classes can be active. See [12], section *13.3.8* for more details). It also includes support for the specification of asynchronous communications (i.e., Signal, SendSignalAction, SignalEvent, see [12], section *13.12.24*, *11.3.45* and *13.3.25*). The semantic formalization, called Execution Model, takes the form of a UML model specified with the fUML subset itself, simply by considering the fact that the fUML execution engine is a particular executable fUML model. It defines the operational procedure for the dynamic changes required during the execution of a fUML model. In the following section, we start by providing an overview of the Execution Model. Then, we discuss limitations of the Execution Model regarding the management of concurrent executions.

2.1 Overview the fUML Execution Model

The Execution Model has been defined following the Visitor design pattern [11], where almost each class of the Execution Model has a relationship with a class from the fUML syntax subset (except for a package called *Loci*, where classes *Locus*, *Executor* and *ExecutionFactory* are not visitors, and are just used for setting up the execution engine).

Each visitor class of the Execution Model basically provides an interpretation for the associated fUML class, and therefore explicitly captures the corresponding execution semantics. Globally, the Execution Model can be considered as the model of an interpreter for UML models specified with the fUML subset. Figure 1 illustrates a part of this global architecture. It represents the relationship between syntactic elements of the fUML subset (left-hand side of Figure 1) and corresponding visitors of the Execution Model (right-hand side part of Figure 1). For example, the execution semantics associated with the concept of *Class* (which is part of the fUML subset) is defined by the class *Object* from the execution model.

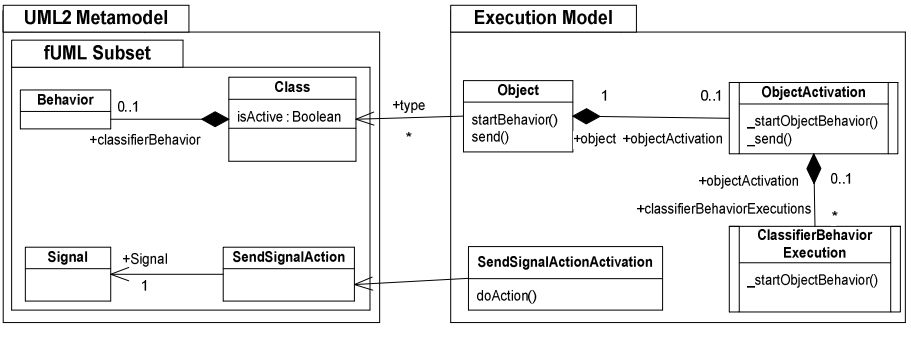

Fig. 1. The global architecture of execution model

It is important to notice that the Execution Model exploits the mechanisms provided by fUML for concurrency and asynchronous communications. For example, classes *ObjectActivation* (which encapsulates the execution of an event dispatch loop, enabling a given active object to react to event occurrences) and *ClassifierBehaviorExecution* (which encapsulates the concurrent execution of the classifier behavior associated with the type of an object) are active classes, i.e., classes whose instances have their own thread of control. In principle, the Execution Model thus explicitly captures the concurrent aspects of fUML execution semantics. In practice, however, the management of concurrency is buried inside the architecture of the fUML Execution Model. Regarding our preliminary objective, this is an important limitation of the fUML Execution Model: The place where concurrency is handled in the Execution Model must be accessible and explicit, so that it can be conveniently tailored to the needs of particular application domains. In the two following sections, we first discuss this limitation and its relationship with the usage of Java as a concrete notation for the description of behavioral aspects of the fUML Execution Model (i.e., mainly, behaviors associated with operations of classes from the Execution Model). Then, we more generally discuss the absence, in the architecture of the Execution Model, of explicit mechanisms for scheduling and synchronizing instances of concurrent entities (i.e., active objects).

2.2 On the Actual Java Specification of the Execution Model

UML activities are the only behavioral formalism supported by fUML. In the Execution Model, they are practically used to specify the implementations of every operation and/or classifier behaviors. However, for significant behaviors, these diagrams quickly become large and complex and thus hard to understand. Instead of using such complex graphical notation (or defining from scratch a new textual notation for activities), the authors of the fUML specification have used Java as a concrete textual notation for capturing behavioral aspects of the Execution Model, respecting a strict "Java to Activities" mapping (see. Appendix A of [2] for details).

In other words, Java statements should just be considered as a concrete and concise textual syntax for UML activities. Nevertheless, the positive side effect regarding the choice of Java is that the Execution Model takes an executable form, which could be used as a model interpreter for UML models respecting the fUML subset. A reference implementation is thereby provided by Model Driven Solutions [3]. However, the "Java to Activities" mapping (defined in Appendix A of [2], and followed for the definition of the Execution Model) does not consider native Java threading mechanisms. fUML mechanisms related to concurrency and asynchronous communications (e.g., active objects, signal emissions, etc.) are simply depicted using syntactic conventions, with no explicit manifestation of the Java Thread API. For example, a call to the operation *_send()* of class *ObjectActivation* (depicted in the right-hand side of Figure 1) is the Java mapping for a SendSignalAction, which normally corresponds to an asynchronous signal emission. Therefore, an interpreter strictly conforming to the Java implementation of the Execution Model can only interpret fUML models as sequential Java programs (e.g., a call to *_send()* remains a synchronous and blocking Java call).

To be clear, the fact that the resulting Java implementation is mono-threaded and purely sequential is not a fundamental issue *per se*. Indeed, as we will see in the Related Works section, most state-of-the-art simulation tools are also sequential and mono-threaded. However, these tools include explicit mechanisms for simulating concurrency, usually with a well indentified entity which is responsible for triggering the execution of the various behaviors, according to a given scheduling policy. The real issue with the current architecture of the Execution Model is that there are no equivalent mechanisms, and that executions obtained via the Execution Model are purely sequential. Let us illustrate this issue with a simple example.

The example illustrated in Figure 2 describes a simple application model that we want to simulate using the fUML Execution Model. It contains two active classes (C1 and C2) whose instances will communicate via signal exchanges (S1 and S2). The classifier behaviors of C1 and C2 are respectively described by activities C1Behavior and C2Behavior. C1 asynchronously sends a signal S1 to C2, and then waits for a reception of a signal S2 from C2. On the other side, C2 waits to receive a signal s1 from C1. After the reception, it asynchronously sends a signal S2 to C1.

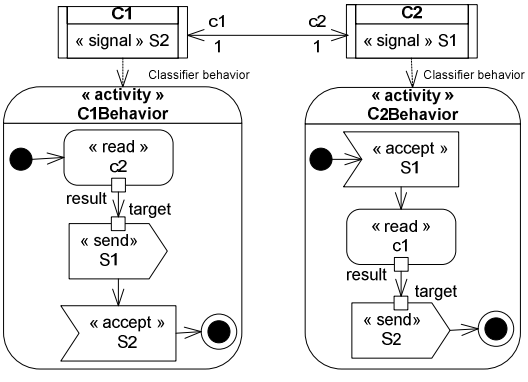

Fig. 2. fUML model of a simple asynchronous system

Figure 3 shows a sequence diagram of a sequential execution trace respecting the java statements of the operational fUML Execution Model. The hypothesis for this execution trace is that two active objects c1:C1 and c2:C2 have been created, and that c2 has been started before c1[2]. Lifelines appearing in the sequence diagram of Figure 3 represent instances of classes from the fUML execution model. The interactions between these lifelines show how the model specified in Figure 2 is actually interpreted by the fUML Execution Model (in this case, all the execution is carried out in one thread).

On the right-hand side of Figure 3, the instance of *ClassifierBehaviorExecution* represents the execution of the classifier behavior of c2. Once it is started, it performs the *AcceptEventAction*. From the Execution Model standpoint, It consists in registering an *EventAccepter* for S1 within a list of waiting event accepters (i.e., call to operation

[2] Another fundamental limitation of this sequential Java interpretation is that it is non-deterministic. The resulting execution trace will be different if c1 is started before c2.

register()). It captures the fact that the execution of c2 is now waiting for an occurrence of S1. However, the execution of c2 does not actually wait for an occurrence of S1 (i.e., with the strict interpretation of the Java statements, the *ClassifierBehaviorExecution* is not executed on its own thread). Instead, it returns to the main activity, which continues the execution by starting the classifier behavior of c1. The execution flow of c2's *ClassifierBehaviorExecution* will be further continued, after an explicit notification. On the left-hand side of Figure 3, when the classifier behavior of c1 starts (i.e., call to *execute()* emitted by the *ActivityExecution*), it executes the *SendSignalAction*. The semantics associated with the *SendSignalAction* is captured in the execution model by calling the operation *send()* of target object c2, which in turn calls the operation *send()* of *ObjectActivation*. It results in adding a signal instance s1 to the event pool associated with the object activation of c2.

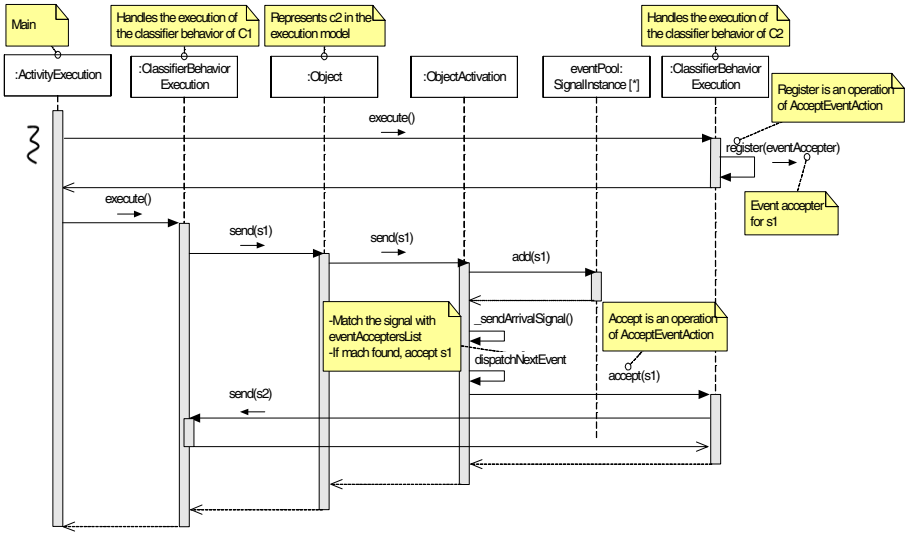

Fig. 3. Execution trace from a sequential implementation of the Execution Model

In order to notify the *ClassifierBehaviorExecution* of c2 that a signal is available for dispatch (and therefore that its execution flow can potentially be continued if there is a matching *EventAccepter*), a call to *_send(new ArrivalSignal())* is emitted, which in turn causes a call to *dispatchNextEvent()*. This operation dispatches a signal from the event pool and matches it against the list of waiting event accepters. If an event accepter matches, a call to the *accept* operation of the *AcceptEventAction* is performed and the classifier behavior of c2 continues the execution by sending signal S2 to c1. The execution of this *SendSignalAction* results in a call to operation *send()* on target object c1, which in turn implies the sequencing of operations described above.

Beyond these technical details, it is important to notice here that this sequential propagation of operation calls will finally result in a valid execution trace (i.e., an execution trace respecting control and data dependencies expressed between actions in the application model being simulated). Basically, once an action execution terminates, it

will simply trigger the execution of another action that can logically be executed after it. The problem here is that the mechanisms which determine the next action to be scheduled is buried inside the implementation of each *ActionExecution* visitor class. If we want the Execution Model to be easily customizable for the real-time domain (which is our primary objective), we clearly need to extract this scheduling aspect from visitor classes, and add an explicit entity that would be responsible for scheduling the execution of actions. Once the entity which is responsible for scheduling action executions is clearly identified, it can be easily specialized to capture various execution schemes, corresponding to various execution semantics (i.e., semantics implied by a profile definition). Perceptive readers may wonder whether the need for an explicit scheduler is the consequence of the sequential Java implementation.

If we make abstraction of the actual Java statements and the way they would be interpreted by a Java compiler (i.e., sequential propagation of synchronous and blocking operation calls), the classifier behavior of each active object c1 and c2 is theoretically started asynchronously and performed on its own thread. What is important to notice is that active objects are simply started by the Execution Model, and finish their execution once their associated classifier behavior terminates. There is neither a well identified entity in the Execution Model describing scheduling rules, nor synchronization primitives that could be used by the scheduler to synchronize running active objects (e.g., operations or signal receptions that could be associated with class *Object* of the Execution Model depicted in Figure 1).

This architecture is not well suited to our primary objective: Specializing the Execution Model in order to reflect concerns of the real-time domain. For this purpose, we believe that introducing an explicit and well-identified entity responsible for scheduling active objects and/or action executions is mandatory, along with well-identified primitives for synchronizing and scheduling concurrent entities. Existing solutions (discussed in the next section) in model-based simulation of concurrent systems could inspire the modifications required by the Execution Model.

3 Related Works

In the field of Hardware Description Languages (HDLs), designers have already been facing the issue of simulating hardware systems (which are intrinsically concurrent) on design platforms which are typically not concurrent. SystemC [9, 10] is a representative example of solutions put into practice in this domain in order to solve this issue. It basically consists of a set of C++ extensions and class definitions (along with a usage methodology), and a simulation kernel for executing them. These extensions include handling of concurrent behaviors, time sequenced operations and simulation support. The core of SystemC is based on an event-driven simulator, where processes are behaviors and events are synchronization points that determine when a process must be triggered. The SystemC scheduler controls the timing, the order of process execution and handles event notifications. It provides primitives to synchronize and notify processes (e.g., *wait()* and *notify()* primitives). Concretely, similar mechanisms could be easily integrated in the fUML Execution Model, by adding a scheduler and primitives like *wait()* and *notify()* (which would be associated with class *Object*).

More generally, in the field of model-based simulation of concurrent systems, generic approaches such as Ptolemy [13] and ModHel'X [14] should also be considered. Ptolemy focuses on modeling, simulation, and design of concurrent, real-time, embedded systems. This approach is based on the notion of *actors* which communicate through an interface which hides their internal behaviour and is composed of *ports*. Models are built from actors and *relations* between their ports, with a *director* in charge of interpreting the relations between ports and the values available on the ports. The director of a model gives the execution semantics of the model as the rules used to combine the behaviors of its component actors. In fact, a director may represent a family of execution semantics and may have parameters such as a scheduling policy. Ptolemy comes with a number of directors ranging from Synchronous Data Flow for discrete time signal processing to Continuous Time for modeling physical processes. It supports a Discrete Event model of computation which is similar to the execution model of SystemC, as well as a Process Network model of computation in which asynchronous processes are synchronized on the availability of their inputs (contrary to CSP, producing data is never blocking, only getting data may block a process if the data is not yet available). In Ptolemy, actors are autonomous entities with a behavior which may be executed in its own flow of control. However, in many models of computation, actors are activated in sequence according to a static or dynamic schedule. What is important to notice here is that the Director / Actor architecture of Ptolemy is flexible enough to support multiple models of computation, that is to say multiple execution semantics. Regarding the fUML Execution Model, a similar architecture could be adopted: Active objects and/or action executions could be considered as actors, and the explicit entity responsible for scheduling their execution could be a kind of Ptolemy director. Defining a specialization of the Execution Model for a given application domain (i.e., explicitly capturing the execution semantics implied by a profile) would therefore basically come to extending corresponding classes in the execution model, and overloading or implementing some of their operations.

Like Ptolemy, ModHel'X defines a unique generic simulation engine to support all MoCs. Consequently, ModHel'X is well adapted for heterogeneous systems modeling. It adopts a model-based approach where the whole behavior is represented by a set of blocks, ports and unidirectional lines. A snapshot-based execution engine is proposed for interpreting this structure. As described in [14], a model execution is a sequence of snapshots. To compute each snapshot, the algorithm provides the model with inputs from the environment and builds an observation of the outputs, according to its current state. This process has a generic structure: first, choose a component to observe, then observe its behavior in response to its inputs, and propagate this observation according to the relations within the model structure. This generic algorithm for executing models relies on such primitive operations which can be refined for each model of computation. The semantics of these operations define the semantics of the model of computation. Indeed, ModHel'X has a more generic execution engine and provides a finer grain description of models of computation than Ptolemy. Concretely, we could also get inspiration of this architecture to modify the fUML Execution Model. A class encapsulating the snapshot-based execution engine could be integrated in the Execution Model, and specializing the Execution Model for a given application domain would basically come to provide particular implementations for the operations described above.

Coupling with existing and more static approaches such as TimeSquare [16] could also be considered. TimeSquare provides an environment for modeling and analyzing timed systems. TimeSquare supports an implementation of the Time Model introduced in the UML MARTE profile and the CCSL language (Clock Constraint Specification Language). It displays possible time evolutions as waveforms generated in the standard VCD format. These evolutions constitute a scheduling trace.

TimeSquare takes as input an UML model and a CCSL model applied to the UML model. The CCSL model is used to specify time constraints and apply a specific behavioral semantics on a model. The result produced by TimeSquare is a sequence of steps (Scheduling Map) that can be used by external tools for analysis/simulation purposes. Concretely, coupling the fUML Execution Model would mean that a CCSL model must be generated for a given application model, and that the generated model reflects the time and concurrent semantics of the application domain for which a profile is defined. Scheduling maps generated by TimeSquare could then be "played" by the Execution Model. Again, modifications in the architecture of the Execution Model would be required, and would mainly consist in adding an explicit entity responsible for triggering executions of active objects and actions, with respect to the scheduling map generated by TimeSquare.

4 Introducing an Explicit Scheduler in the fUML Execution Model

In section 2, we have shown that the executions performed by the fUML Execution Model are purely sequential. We have highlighted the absence of an explicit entity responsible for scheduling the execution of actions. We have identified in section 3 different approaches for modeling and simulation of concurrent systems. Each approach contains an entity and primitives to control behavior executions. We propose in this section a lightweight modification of the Execution Model following this general idea. The goal is to break the sequential execution and provide the ability to control the start of each action execution, in a way that can be easily overloaded (so that it is possible to cope with multiple scheduling policies). We introduce for this purpose an explicit scheduler into the Execution model, as illustrated in Figure 4.

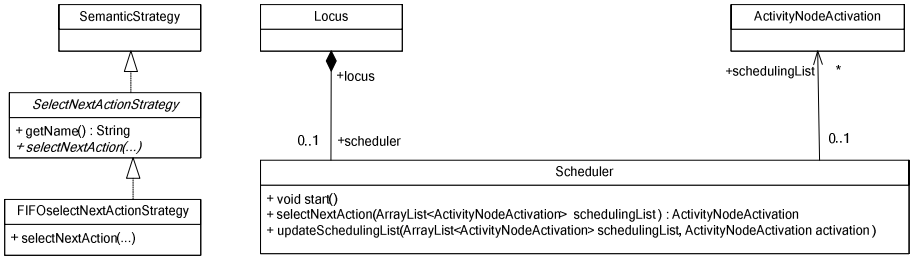

Fig. 4. Description of the Scheduler in fUML Execution Model

The class *Scheduler* manipulates a list of *ActivityNodeActivation* (i.e., this class represents the visitor class of UML::Action) depicted by the property *schedulingList*, which contains the list of all actions ready to execute (i.e., an action is ready to execute if all its control and data tokens are available). *Scheduler* offers several operations that can be used to control executions of actions. These operations are called in the body of *start ( )* which actually start the behavior of the scheduler. The operation *selectNextAction( )* determines the next action to be executed, by extracting an element from *schedulingList*, according to a given scheduling policy. The operation *updateSchedulingList ( )* determines the potential successors for the last executed action (i.e., with respect to control and data dependencies within the executed activity) and adds them to the scheduling list.

To capture several scheduling policies that could correspond to different execution semantics, we rely on the strategy pattern proposed by the Execution model, itself based on the class *SemanticStrategy* (for more details about the strategy pattern, see [11]). In the fUML execution model, *SemanticStrategy* is used to address semantic variation points of UML, with a refinement of this class for each semantic variation point of UML (e.g., there is a class called *GetNextEventStrategy*, which is introduced to address the UML semantic variation point related to the selection of an event from an object's event pool). Fixing a given semantic variation point then comes to refine the corresponding strategy class, by providing an implementation for the operation capturing the strategy.

Following this pattern, supporting different scheduling policies amounts to refine the class *SelecNextActionStrategy* (see Figure 4) for each new policy and to overload the *selectNextAction()* operation to capture the underlying behavior. In our case, we introduce the class *SelecNextActionStrategy*, whose operation *selectNextAction()* is overloaded in order to encapsulate the behavior of one particular scheduling policy. For example, *FIFOSelectNextActionStrategy* is a concrete class that implements a simple FIFO strategy (i.e., by "FIFO", we simply mean that actions are executed respecting their order of appearance in a list of action activations such as *shedulingList*). In order to plug the scheduler onto the fUML execution model, we also modify the behavior of *ActivityNodeActivation* in order to let the scheduler determine the next action to be executed after a given *ActivityNodeActivation* finishes the execution of its visited action. Figure 5 shows a sequence diagram of an interaction trace between the scheduler and an action. The scheduler executes the operation *selectNextAction ( )* that chooses one action from its scheduling list according to a certain policy. Its implementation actually consists in delegating the choice to a *SelectNextActionStrategy* class (in this case, the policy is the one of *FIFOSelectNextActionStrategy*. Note that the *Loci* class dynamically determines the various semantic strategy classes to be used, provided it has been correctly configured before launching the execution). Then, the scheduler triggers the execution of the selected action. The behavior of the selected action is performed by the operation *doAction( )*. The operation *sendOffer( )* then propagates tokens to the next actions that can logically be executed after it, but it does not trigger anymore the execution of these actions. The scheduler indeed calls *updateSchedulingList()* to add these potential successors into the scheduling list. The next action to be executed is selected by calling *selectNextAction*(). This behavior is repeated until the scheduling list becomes empty (i.e., the execution of the activity is finished).

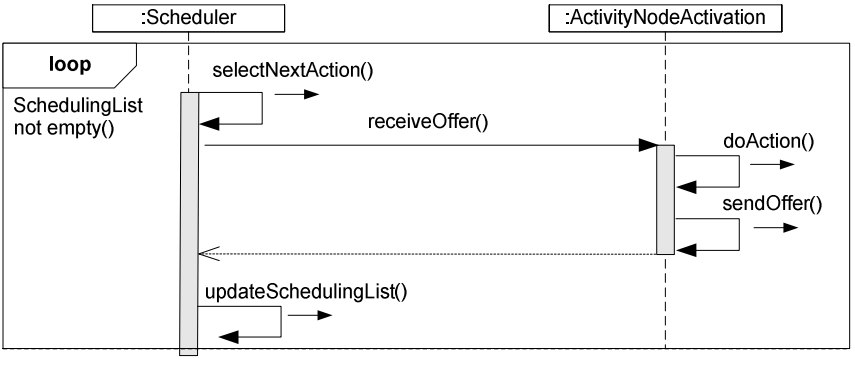

Fig. 5. Execution trace of scheduler interactions with action

5 Conclusion

The ongoing OMG standard on the semantics of a foundational subset (fUML) for executable UML models defines a general-purpose execution model for a subset of UML. This subset includes non trivial mechanisms, carrying concurrent and asynchronous execution semantics (e.g., active objects, signals, etc). Our objective was to evaluate how far the current definition of the fUML Execution Model can support formalization of concurrent and temporal semantic aspects required for real time embedded system design and analysis. As shown in the study, the current form of the fUML execution model is not suited to this objective, mainly due to the way concurrency and asynchronous communications are actually handled.

We have mainly shown that the current architecture of the fUML Execution Model suffers the lack of explicit mechanisms for manipulating and synchronizing concurrent entities. Existing solutions for embedded system simulation indicate that it is possible to provide much more adapted and realistic solutions. We proposed some concrete modifications regarding the architecture of the fUML Execution Model, inspired by these solutions. We took care of minimizing changes in the architecture, so that we can leverage as much as possible on the existing Execution Model (and all the work that its definition implied). The proposed solution is mainly intended to show that a modification of the fUML Execution Model is technically feasible and reasonable in terms of efforts. However, further experiments are still required to validate the proposed modifications. Additionally, this solution only reflects executions by a single unit of computation (i.e., mono-processor). The case of executions onto multiple processing units will be investigated in future works.

Another important aspect which has not been detailed in this article concerns the simulation of time in the Execution Model, which is currently not supported. Time is indeed considered as a semantic variation point within the fUML Specification (Subclause *2.3* of [2]). Consequently, a wide variety of time models could be adopted, including discrete or continuous time. fUML does not make any assumptions about the sources of time information and their related mechanisms. Therefore, to support timed execution semantics and underlying timing properties (e.g., ready time, period, deadline, etc.), it is necessary to extend the Execution Model with both necessary

syntactic and semantic concepts. Time is a central aspect to our work. Resolving the concurrency issues of the fUML Execution Model by adopting solutions similar to those proposed in the Related Works could therefore, in the same move, provide a solution for the Time issue of the Execution Model. Ultimately, our goal is to provide a kind of methodological and tooled framework for the definition of UML profiles, where the semantic specializations of UML implied by a profile will take as much considerations as syntactic specializations.

References

1. OMG. A UML profile fore MARTE: Modeling and Analysis of Real-Time Embedded systems Version 1.0. (2009)
2. OMG. Semantics of Foundational Subset for Executable UML models FTF-Beta2 (2009)
3. Model driven solution, `http://portal.modeldriven.org/content/fuml-reference-implementation-download`
4. Agha, G.: Actors: a model of concurrent computation in distributed system. MIT Press, Cambridge (1986)
5. Selic, B., Ward, P.T., McGee, G.G.: Real-Time Object-Oriented Modeling. Wiley, John & Sons, Inc. (October 1994), ISBN-13: 9780471599173
6. Terrier, F., Fouquier, G., Bras, D., Rioux, L., Vanuxeem, P., Lanusse, A.: A real time object model. In: International Conference on Technology of Object Oriented Languages and Systems, TOOLS Europe 1996, Paris, France, Février (1996)
7. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software 91(1), 127–144 (2003)
8. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time systems in BIP. In: 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12 (2006)
9. Open SystemC Initiative. SystemC 2.0.1 Language Reference Manual (2004)
10. SystemC. Official web site of SystemC community, `http://www.systemc.org/`
11. Gamma, Helm, Johnson, Vlissides: Design Patterns: Elements of Resuable Object-Oriented Software, pp. 163–174, 331–344. Addison-Wesley, Reading (1995)
12. OMG. Unified Modeling Language: Superstructure. version 2.2. formal/2009-02-02 (2009)
13. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – The Ptolemy approach. Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software 91(1), 127–144 (2003)
14. Boulanger, F., Hardebolle, C.: Simulation of Multi-Formalism Models with Mod-Hel'X. In: Proceedings of ICSTW 2008, pp. 318–327. IEEE Comp. Soc., Los Alamitos (2008)
15. Executable UML/SYSML semantics. Model Driven Solutions. Final project report (November 2008)
16. André, C., Ferrero, B., Mallet, F.: TimeSquare: a Multiform Time Simulation Environment. In: Sophia Antipolis and Formal Analysis Workshop (Décembre 2008)