A Runtime Model for fUML*

Tanja Mayerhofer, Philip Langer, and Gerti Kappel
Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{mayerhofer, langer, gerti}@big.tuwien.ac.at

ABSTRACT

With the introduction of fUML, an OMG standard defining the operational semantics of a subset of UML and the conforming virtual machine, UML models can be used not only for informal design sketching but also for completely building executable systems. Although this has been an important step for UML, the full potential of having executable UML models, such as enabling runtime analysis and adaptation, cannot be realized using the standardized virtual machine due to the lack of the adequate means for accessing important runtime information and controlling the execution of UML models. In this paper, we aim at establishing the necessary basis to overcome this limitation. Therefore, we introduce extensions of the standardized fUML virtual machine in terms of a dedicated trace model, an event model, and a command API. We provide an open-source implementation of the proposed extensions, as well as a model debugger for UML models based on this implementation to demonstrate the feasibility of the presented concepts.

1. INTRODUCTION

Executable models are necessary for bridging the gap between design time models and the running application. To be executable, the semantics of a model has to be specified precisely and completely. Unfortunately, the semantics of many modeling languages is specified only informally leading to imprecision and ambiguity. For a long time, this was also true for UML, the most adopted modeling language in industry [6]. However, UML has evolved recently from a descriptive language that can only be used for informal design sketching to a prescriptive language that can also be used as a programming language if desired [9]. This transition was enabled by the introduction of the fUML[1] standard.

fUML contributes a formal definition of the operational semantics of a key subset of UML 2 in terms of a well-defined virtual machine for executing UML models. In essence, this subset consists of the key parts of UML activities and class diagrams, which are considered sufficient for describing the semantics of the remainder of UML. Although this is a major step towards the utilization of executable UML models, the full potential of UML model execution cannot be exploited, because the standardized virtual machine lacks in providing the means for runtime observation, analysis, and execution control. Moreover, it is currently unclear how the runtime information of executable UML models can be obtained from the virtual machine and how it may be represented adequately in terms of a runtime model. As a result, important applications of models at runtime [1], such as controlling, observing, and adapting the behavior of a system at runtime, cannot be realized using fUML so far.

To address this limitation, we propose a trace model for fUML, which enables the runtime analysis of executed UML models establishing the basis for runtime adaptation. Furthermore, we propose an event model and a command API, which enable to observe and control the model execution process during runtime. We implemented these proposed functionalities by extending the fUML reference implementation[2] and developed a model debugger for UML models to demonstrate the feasibility and sufficiency of our approach.

The remainder of this paper is structured as follows. In Section 2, we summarize related work regarding model execution and runtime models. Our proposed trace model, event model, and command API for fUML are introduced in Section 3 and Section 4. In Section 5, we discuss the validation of the proposed artifacts and conclude with an outlook on future work in Section 6.

2. RELATED WORK

In this section we give an overview on existing approaches for specifying the semantics of UML enabling the execution of UML models. Further, we address existing work regarding the usage of traces as runtime models.

UML semantics. Approaches for specifying the semantics of a modeling language can be classified according to three categories: denotational, translational, and operational semantics [3]. For formalizing the semantics of UML, remarkable approaches have been suggested in the past. One prominent example for a denotational approach is the System Model introduced by Broy et al. [2]. The translational semantics approach has been, for instance, applied by Mellor and Balcer [8] with the introduction of Executable UML.

*This research has been partly funded by our industry partner LieberLieber Software GmbH.

[1] `http://www.omg.org/spec/FUML/Current`

[2] `http://fuml.modeldriven.org`

The fUML standard is the latest attempt to standardize the semantics of UML using an operational semantics approach. As fUML is promoted by OMG, it seems to be promising to become widely adopted.

Trace models. To enable runtime analysis, trace models are used to represent the execution trace of programs. Therefore, various trace formats have been proposed [5]. However, those formats focus on traditional programs and, hence, cannot be used to capture the execution trace of UML models, because the runtime concepts of programs, such as *method calls* and *routine calls*, cannot be transferred directly to runtime concepts of UML models. Moreover, the essential runtime concepts of UML activities, such as *token flows*, are not represented in existing trace formats. As runtime models should be as close to the problem space as possible [1], we argue that a *dedicated trace model for fUML* is necessary to capture the execution of UML models adequately and precisely.

In the domain of software modeling, the usage of model-based traces (or traces in general) has been proposed already to serve as runtime models enabling to reason about the execution, as well as dynamic analysis and adaptation. Most of these approaches assume that the running application is realized by generated code. In this scenario the runtime information available on the code level has to be mapped to the more abstract model level. Maoz [7] proposed an approach to derive the required mapping code automatically from models and weave it into the application code using aspect-oriented programming. A similar approach was developed by Vogel et al. [10], who proposed to use model transformation techniques to synchronize the runtime information with the models that provide different views on the running system. Another approach was taken by Ghezzi et al. [4], who proposed to extract formal models from the runtime behavior of a running application. As the fUML virtual machine interprets the models directly, neither an extraction of behavior models nor a mapping of code-level runtime information to the model level is necessary.

3. TRACE MODEL

Traces record the execution of a system and provide an abstract representation of its runtime behavior. Thereby, a trace provides the information necessary for analyzing a system's execution, which constitutes the crucial basis for several applications, such as dynamic adaptation [1]. To enable these important applications for executable UML models, we elaborated a dedicated trace metamodel capable of recording the model execution carried out by the fUML virtual machine. Thereby, we aimed at fulfilling the requirement that a recorded trace provides sufficient information for comprehending, reconstructing, and replaying an execution. For instance, a trace model should be expressive enough to allow answering the following questions.

1. **Input/output relationship**: Which inputs have been provided for the execution of an activity node / activity and which outputs have been produced?
2. **Chronological order**: In which chronological order have the activity nodes of an activity been executed?
3. **Logical order**: In which logical order have the activity nodes of an activity been executed? In other words, the executions of which activity nodes have enabled the execution of another specific activity node?

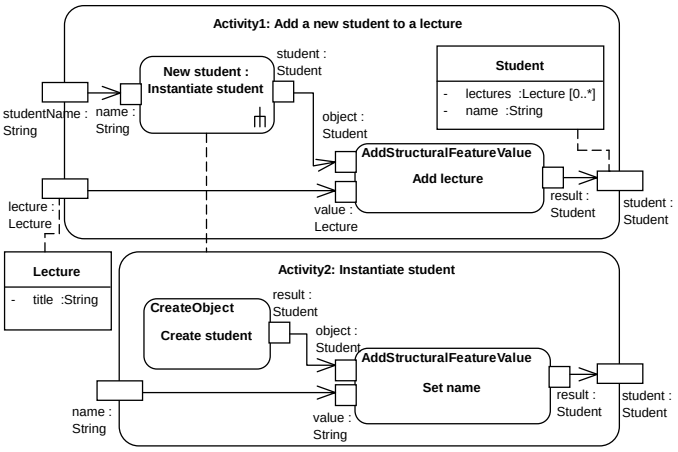

Figure 1: Example of a UML activity

4. **Edge traversal**: Across which edge has an input for the execution of an activity node been provided?

For illustrating our elaborated trace metamodel, we use the UML activity example depicted in Figure 1. In this example the UML activity *Activity1: Add a new student to a lecture* first instantiates an object of the class *Student* by calling the activity *Activity2: Instantiate student* (which also initializes the *name* attribute of the *Student* object with a given String value), adds a given *Lecture* object to its list of *lectures* and provides the *Student* object as activity output.

Figure 2 depicts our elaborated trace metamodel for capturing traces of the execution of UML activities by the fUML virtual machine. The metaclasses colored white represent model elements of the executed UML activity itself. A `Trace` contains a list of `ActivityExecutions` that represent the execution of UML activities. An `ActivityExecution` consists of a chronologically ordered list of `ActivityNodeExecutions`, which represent the execution of activity nodes. A `CallActivityNodeExecution` is a special kind of an `ActivityNodeExecution` that represents the call of a behavior that may cause the execution of another activity. `ActivityExecutions`, as well as `ActivityNodeExecutions`, can have inputs and outputs represented by the metaclasses `ParamterInput` / `ParameterOutput` and `Input` / `Output`, respectively. `ParamterInputs` and `ParameterOutputs` reference `ObjectTokenInstances` that carry `ValueInstances` containing `Values`, such as String values or objects. Note that the `ValueInstances` refer to deep copies of the `Values` to maintain a self-contained copy of the input and output data. `UserParameterInput` is a subtype of `ParameterInput` that is used to represent the input provided by the user when starting the execution of activities. `Inputs` and `Outputs` of `ActivityNodeExecutions` can additionally refer to `ControlTokenInstances` representing the control flow.

To exemplify the usage of this trace metamodel, Figure 3 shows an excerpt of the trace resulting from the execution of the activity example introduced in Figure 1. This excerpt only includes details about the execution of the activity *Activity2*. Using this trace, the before stated questions regarding its execution can be answered as follows.

1. Input/output relationship. The inputs and outputs of an execution are recorded through the metaclasses `Input` / `Output` for the execution of activity nodes, as well as `ParameterInput` / `ParameterOutput` for the execution of activities.

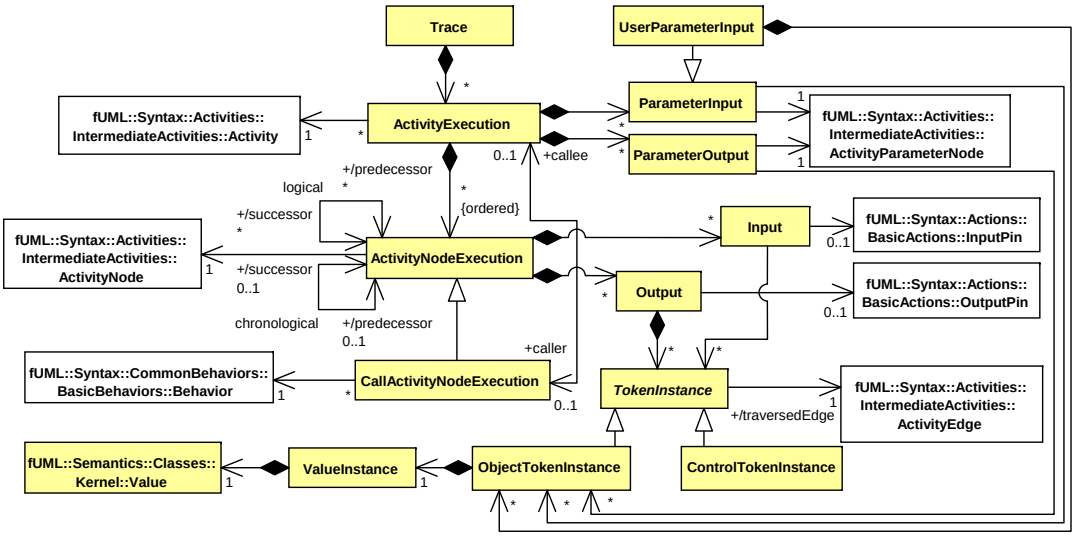

Figure 2: Trace metamodel

In the trace excerpt we can see that the execution of the activity node *Create student* had no inputs and the object *v2* of the type *Student* as output (1a). The execution of the activity *Activity2* had the String value *v1* "Alice" as input (1b) and produced an object *v3* of the type *Student* for which the property *name* was set to "Alice" as output (1c).

2. Chronological order. The chronological order of activity node executions is captured using the derived reference `chronological`. This reference can be derived from the order in which the `ActivityNodeExecutions` are contained by the corresponding `ActivityExecution`. The following OCL expression is used to derive the chronological predecessor.

```
context ActivityNodeExecution::chronologicalpredecessor :
  ActivityNodeExecution def:
if self.activityExecution.activityNodeExecution->
  indexOf(self) <> 1
then self.activityExecution.activityNodeExecution->
  at(self.activityExecution.activityNodeExecution->
    indexOf(self) - 1)
else null
endif
```

Using this OCL expressions, we can derive that the execution of the activity node *Create student* is the chronological predecessor of the execution of the node *Set name* (2).

3. Logical order. The logical order of activity node executions can be derived from the input/output relationships between them and is recorded in the trace using the derived reference `logical`. The execution of an activity node is the logical predecessor of the execution of another activity node if it provides some input to it. The following OCL expression derives the logical predecessor of an activity node execution.

```
context ActivityNodeExecution::logicalpredecessor :
  ActivityNodeExecution def:
ActivityNodeExecution::allInstances()->select(a |
  a.output.tokenInstance->select(t |
    self.input.tokenInstance->includes(t))->size() > 0)
```

In our example the execution of the node *Create student* is the logical predecessor of the execution of the activity node *Set name* because it provides the object *v2* as input (3).

4. Edge traversal. Tokens transport the values that have been generated as output of an activity node execution to the (logical) successor nodes via activity edges. The information concerning the edge that was traversed by a token is represented by the derived reference `traversedEdge` and can be obtained by the following OCL expression.

```
context TokenInstance::traversedEdge : ActivityEdge def:
ActivityEdge::allInstances()->select(e |
  (e.source = Output::allInstances()->select(o |
    o.tokenInstance->includes(self)).outputPin) and
  (e.target = Input::allInstances()->select(i |
    i.tokenInstance->includes(self)).inputPin))
```

Based on this OCL expression, we can derive for our example that the object token instance *ot2* carrying a *Student* object traversed the object flow edge *edge1* from the output pin *op1* "result" of the action *Create student* to the input pin *ip1* "object" of the action *Set name* (4).

4. EVENT MODEL AND COMMAND API

Based on a trace model (cf. Section 3), it is possible to analyze the runtime behavior of activities. However, for allowing to observe and control the execution during runtime, an event model and a command API are required in addition. The main requirement for an event model is that each change of the runtime state triggers a corresponding event to enable observing applications to react accordingly. Besides, to provide a flexible way to control the execution, we require a rich but minimal set of commands.

For classifying the possible types of events, we introduce the event metamodel depicted in Figure 4. `TraceEvents` inform about the state of the execution. `ActivityEvents` indicate the start (`ActivityEntryEvent`) and the end (`ActivityExitEvent`) of an activity execution. Accordingly, `ActivityNodeEvents` signalize the start (`ActivityNodeEntryEvent`) and the end (`ActivityNodeExitEvent`) of the execution of an activity node. In addition to these entry and exit events, we introduce `SuspendEvents` indicating that the activity execution was suspended. An activity execution is suspended either if the activity is executed step-wise and an execution step was completed, or if a breakpoint, which was set for a specific activity node, was hit during the activity execution. In the former case a `SuspendEvent` is issued and in the latter a `BreakpointEvent`. `ExtensionalValueEvents` notify about the creation, destruction, and modification of extensional values (i.e., objects and links) and `FeatureValueEvents` denote the modification of an extensional value's feature value.

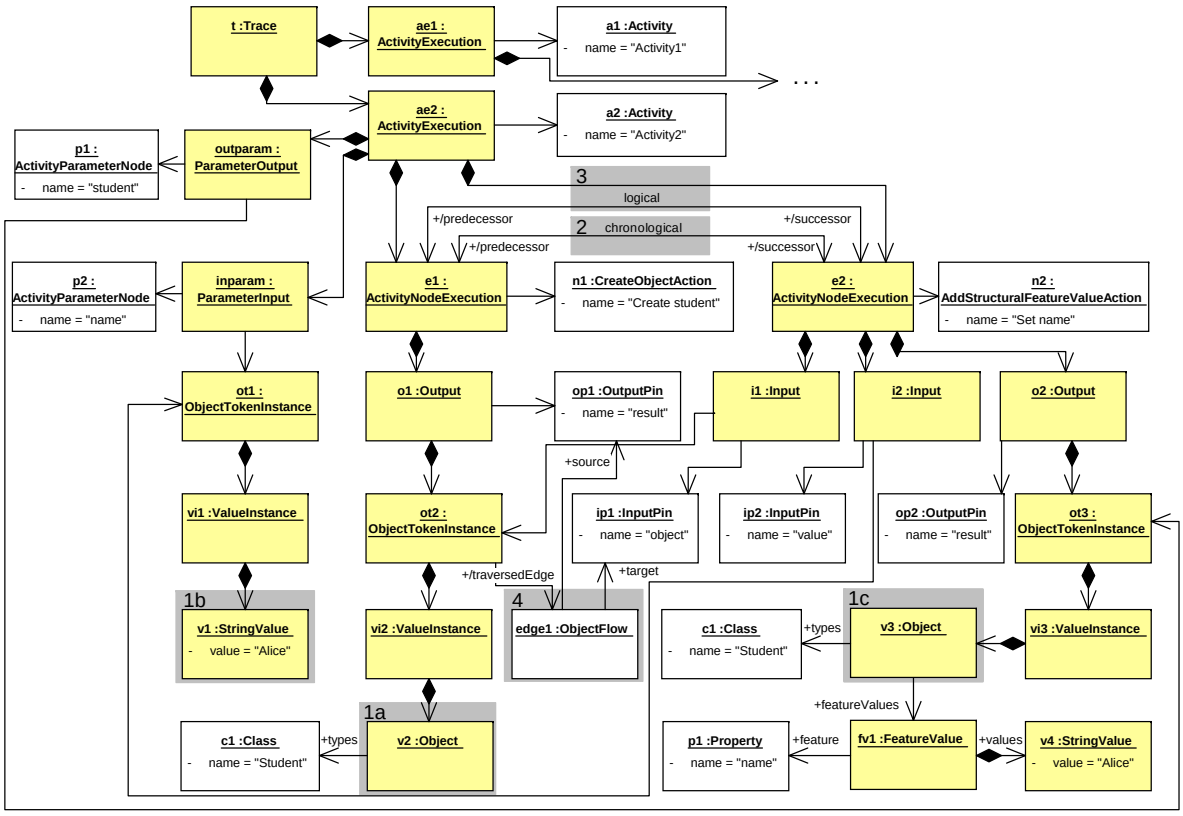

Figure 3: Example of a trace model (excerpt)

To allow applications to receive the notifications about the event occurrences, we provide an observer interface `ExecutionEventListener` that needs to be implemented. Listeners can register at the fUML execution environment realized by a class called `ExecutionContext`, which manages registered listeners and provides them with execution events.
Besides enabling applications to receive events regarding the state of activity executions, we also introduce a dedicated command API to be used for controlling the execution, which is again provided by the class `ExecutionContext`. An excerpt of the available commands is depicted in the following listing and discussed below.

```
void execute(Behavior activity, Object context,
        ParameterValueList input)
void executeStepwise(Behavior activity, Object context,
        ParameterValueList input)
void nextStep(int executionID, ActivityNode node)
void resume(int executionID)
```

To start the execution of an activity, the commands `execute` and `executeStepwise` can be used. The `execute` command triggers the execution of the activity until either a breakpoint is hit or the end of the activity is reached. Using the command `executeStepwise`, the activity execution can be carried out step-wise. In this case the execution is suspended not only if a breakpoint is hit, but also after the completion of each execution step. The first step in an activity execution is completed, when the initially enabled nodes of the activity have been determined. This is indicated by a `SuspendEvent`. A `SuspendEvent` provides an *executionID* that identifies the current execution of the activity. This is necessary because the same activity can be executed several times in parallel. This *executionID* can be further used to continue the particular activity execution using the commands `nextStep` and `resume`. The command `nextStep` instructs the virtual machine to trigger the execution of the next enabled activity node. As multiple activity nodes can be enabled at the same time, it is possible to provide a specific node to be executed in the next step. As the command `nextStep` only performs a single step, a `SuspendEvent` is triggered directly after the activity node has been executed. If this is not desired, the command `resume` can be used to continue the execution with the provided *executionID* until either a breakpoint is hit or the execution terminates.
Further, we introduced methods that enable the management of breakpoints, as well as the retrieval of information about an activity execution, such as the current enabled nodes and the current trace.
Figure 5 exemplifies the sequence of events and commands exchanged between the `ExecutionContext` and an `ExecutionEventListener` using the execution of the activity introduced in Figure 1. The *listener* starts the execution of the activity *Activity1* using the command `executeStepwise`. Consequently, the *context* starts the execution of the activity, determines the initially enabled nodes of the activity, and informs the *listener* about this by raising an `ActivityEntryEvent`, as well as a `SuspendEvent`. The `ActivityEntryEvent` informs the *listener* about the start of the execution of the activity *Activity1* and provides the *executionID* that identifies this execution. The `SuspendEvent` indicates the completion of an execution step and carries the information about the nodes that were enabled in the last step. In our example only one node was enabled, namely the action *New student*. As stated before, the `SuspendEvent` informs about the suspension of the activity execution and indicates that the *listener* can now use the commands `nextStep` or `resume` to continue the execution. In the example, the *lis-*

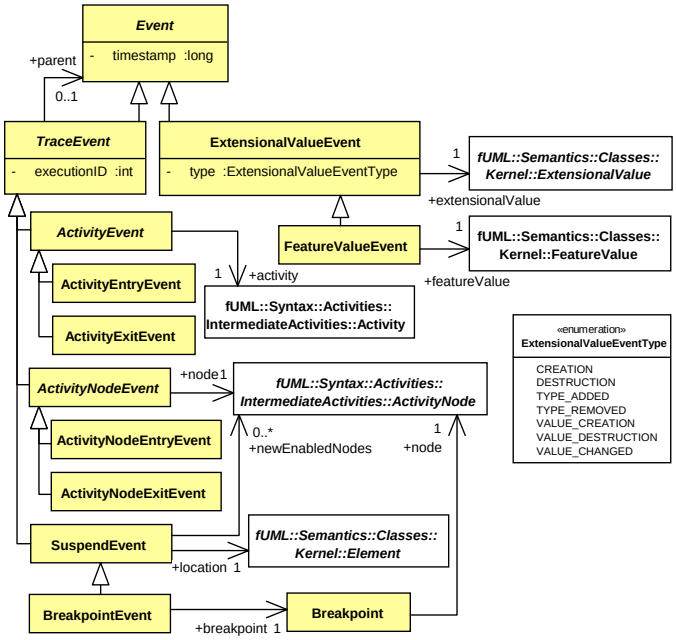

Figure 4: Event metamodel

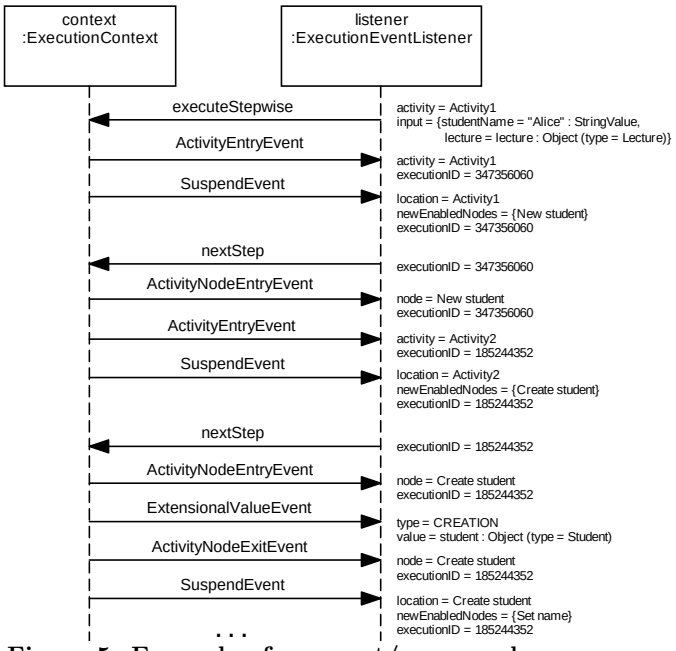

Figure 5: Example of an event/command sequence

tener uses the `nextStep` command to instruct the *context* to perform the next execution step. This triggers the execution of the *New student* node that calls the activity *Activity2*. Accordingly the *context* delivers an `ActivityNodeEntryEvent` informing about the start of the execution of the activity node *New student*, an `ActivityEntryEvent` informing about the start of execution of the activity *Activity2*, as well as a `SuspendEvent` informing about the suspension of the execution and the new enabled node *Create student*. Again the *listener* uses the command `nextStep` resulting in the execution of the enabled node *Create student*. An `ActivityNodeEntryEvent` tells the *listener* that the execution of this node started, the `ActivityNodeExitEvent` informs about the completion of the execution of this node, and the `SuspendEvent` indicates the completion of the execution step. After the `ActivityEntryEvent`, an `ExtensionalValueEvent` is issued, which informs the *listener* about the creation of a new object of the type *Student*.

5. VALIDATION

In this section, we report on our investigations concerning the feasibility and sufficiency of the proposed trace model, event model, and command API for fUML. In particular, we aim at answering the following research questions.

1. **Feasibility**: Is it feasible to obtain the necessary information from the *fUML reference implementation* for creating the trace model and the event model, and is it possible to extend the *fUML reference implementation* for realizing the presented set of commands for controlling the execution of UML activities?
2. **Trace model**: Is the information provided by the trace model a sufficient basis for reasoning about the runtime and for identifying the steps that lead to a specific execution state of a UML activity?
3. **Event model**: Are the provided events and the information they carry sufficient for observing the state changes thoroughly during the execution of a UML activity?
4. **Command framework**: Is the set of commands sufficient for controlling the activity execution flexibly?

Research Question 1. We assessed the feasibility by creating a prototypical implementation of the proposed trace model, event model, and command API based on the reference implementation of the standardized fUML virtual machine. Although the reference implementation offers a complete virtual machine for fUML-conforming activities, it currently neither provides the means for accessing the runtime information during the execution of a UML activity nor does it allow for controlling the execution. To establish the basis for this functionality, we had to extend the reference implementation. We used AspectJ to weave the necessary extensions into the original code without altering it. Nevertheless, by leveraging these extensions, we succeeded in realizing the means for building the trace model, for issuing events whenever a state change of the runtime model occurred, and for step-wise execution. The source code of our implementation is available at our project website[3].

Research Questions 2–4. The application domains of runtime models are very diverse. As a consequence, it may depend on the specific application whether runtime models may be considered sufficient or not. However, one application domain that demands for very precise runtime information, that depends heavily on event notifications during the runtime, and that requires a powerful mechanism for controlling the execution is *debugging*. Thus, we chose to assess the sufficiency of the proposed artifacts by implementing a debugger for UML models based on our extended fUML virtual machine. This implementation, which is available on our project website, is integrated with the Eclipse Debug framework[4] and allows debugging activities created with the Papyrus diagramming editor[5].
A debugger usually depicts the state of the program being debugged in terms of *threads*, *stack frames*, and *variables*. A *thread* is a sequence of actions that may execute in parallel

[3] `http://www.modelexecution.org`
[4] `http://www.eclipse.org/eclipse/debug`
[5] `http://www.papyrusuml.org`

with each other. In the context of fUML, multiple activity nodes may be enabled at the same time, for instance, if the control flow of an activity exhibits fork nodes. Hence, we consider each concurrently enabled node to run in an own thread. For deriving the currently running threads, we may obtain the enabled nodes of a suspended activity execution easily from the `ExecutionContext`. However, a new enabled node does not always constitute a new thread, because the node activation may also result from resuming an already existing thread. To obtain this information, the trace model can be used, which contains for each enabled node an instance of `ActivityNodeExecution` having `Inputs`. If an existing thread concerns the logical predecessor of this `ActivityNodeExecution` instance, this thread may be updated to the new enabled node. If it has several predecessors for which threads exist, one of them may be updated and the others are terminated. If we do not find a corresponding thread or all corresponding threads have been assigned to other enabled nodes already, we create a new thread.

A *stack frame* represents the runtime context of a suspended thread, which is in our context the current activity node being executed. A stack frame may have child stack frames that represent nodes that called the activity where the current node resides in. For deriving the stack frame hierarchy, we again make use of the trace model by navigating from the instance of `ActivityNodeExecution` representing the execution of the current node to its containing instance of `ActivityExecution` and check whether this `ActivityExecution` refers to an `CallActivityNodeExecution` instance using the reference `caller`. If this applies, we add a child stack frame, which indicates that the current activity node is executed in the context of an activity that has been called from another node. This procedure can be repeated until the upper most activity execution is reached.

A *variable* denotes data values that are in the scope of the current stack frame. In the domain of activities, variables refer to values that reside on input pins. Hence, obtaining the variables is easily possible by looking up the `ActivityNodeExecution` instance representing the current enabled node and read the `ValueInstances` of the `ObjectTokenInstance` via the referenced `Input` instance.

Apart from depicting the current state of a suspended execution, a debugger should empower users to step-wise resume a suspended execution using the commands *step into*, *step over*, *step return*, *resume*, and *terminate*. In the context of fUML, *step into* and *step over* should cause the execution of the next node in the currently selected thread. However, if this node is a call action, *step into* should cause the thread to suspend directly before the first node of the called activity is executed, whereas *step over* should execute the entire called activity and suspend after the call action itself has been executed. Accordingly, *step return* should execute the entire current activity. This can be realized easily using the proposed event model and command API. Therefore, we call `nextStep` repeatedly until a certain event is issued. For instance, in the case of *step return*, `nextStep` is called until an `ActivityExitEvent` for the current activity is received. The commands *resume* and *terminate*, as well as breakpoints, are supported directly by the command API.

In summary, we conclude that the proposed trace model, event model, and command API are sufficient for building a debugger offering all the functionality that is required for step-wise executing activities, as well as for inspecting the current state of a suspended activity execution. Moreover, we argue that the trace model contains much more details, which build a solid basis also for other tools that rely on runtime information of activity executions, such as runtime analysis and adaptation.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented an extension of the standardized fUML virtual machine in order to enable accessing important runtime information and controlling the execution of UML models. Therefore, we introduced a dedicated *trace model* acting as a basis for reasoning about the execution of a model, an *event model* for observing state changes during the execution, as well as a *command API* providing the means for controlling the execution flexibly. With these extensions, we aim at establishing the basis for exhausting the full potential of having executable UML models, such as enabling runtime analysis and performing adaptations at runtime. We provide an open-source implementation of the proposed extensions and of a model debugger based on this implementation to demonstrate the feasibility of the presented concepts. In future work, we plan to validate the usefulness of the proposed extensions more thoroughly by building more sophisticated runtime analysis tools based on our implementation. In particular, we aim at developing techniques for validating whether changes applied to executed UML models invalidate existing traces, as well as for testing the behavior of a model based on execution traces.

7. REFERENCES

[1] G. S. Blair, N. Bencomo, and R. B. France. Models@ run.time. *IEEE Computer*, 42(10):22–27, 2009.

[2] M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. Definition of the system model. In *UML 2 Semantics and Applications*, pages 61–93. Wiley, 2009.

[3] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai. Challenges and directions in formalizing the semantics of modeling languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.

[4] C. Ghezzi, A. Mocci, and M. Sangiorgio. Runtime monitoring of functional component changes with behavior models. In *Proc. of MODELS'11*, pages 152–166, 2012.

[5] A. Hamou-Lhadj and T. C. Lethbridge. A metamodel for the compact but lossless exchange of execution traces. *Softw. Syst. Model.*, 11(1):77–98, 2012.

[6] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proc. of ICSE '11*, pages 471–480, 2011.

[7] S. Maoz. Model-based traces. In *Proc. of MODELS'08*, pages 109–119, 2009.

[8] S. J. Mellor and M. J. Balcer. *Executable UML: A foundation for model-driven architecture*. Addison Wesley, 2002.

[9] B. Selic. The less well known UML. In *Formal Methods for MDE*, volume 7320 of *LNCS*, pages 1–20. Springer Berlin / Heidelberg, 2012.

[10] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental model synchronization for efficient run-time monitoring. In *Proc. of MODELS'09*, pages 124–139, 2010.