

ACTIVECHARTSIDE – AN INTEGRATED SOFTWARE DEVELOPMENT ENVIRONMENT COMPRISING A COMPONENT FOR SIMULATING UML 2 ACTIVITY CHARTS

Stefan Sarstedt, Dominik Gessenharter, Jens Kohlmeyer, Alexander Raschke and Matthias Schneiderhan
Department of Programming Methodology and Compiler Construction
University of Ulm
89069 Ulm, Germany
mdd@informatik.uni-ulm.de

KEYWORDS

Software Simulation and Debugging, UML 2 Activity Charts Execution, Model-Driven Architecture

ABSTRACT

Modeling the behavior of software at an abstract level and using supporting simulation facilities offer great advantages in developing software systems.

In this paper we present our implementation and simulation environment ACTIVECHARTSIDE, a tool for object oriented software development. Our approach enables the reuse of large parts of design artifacts for implementation by interpreting UML 2 activity charts, which are used to model the control flow of an application. The central component in our approach is the model interpreter, which is responsible for executing the activity charts. An optional visualization component implemented by the ACTIVECHARTSIDE realizes debugging and simulation functionality to assist quality assurance.

We give a brief overview of our approach and illustrate our tool and its central components by means of a small example.

INTRODUCTION

Modern software development approaches, like the Model-Driven Architecture (MDA) initiative of the Object Management Group (OMG), focus on modeling the static structure and the behavior of a system in an abstract way (for further information see [MDA, 2003]). This aims at improving maintenance and quality and increasing the reuse of components constructed in early process phases. By separating the definition of application systems from the technology they run on, investments made during the design of a system can be preserved for implementation, even if the underlying technology platform changes [McNeile, 2004].

In our approach to Model-Driven Architecture the control flow of the application (i.e., the behavior) is modeled with UML 2 activity diagrams during analysis and design phases. These diagrams are seamlessly reused for the implementation by interpreting them at runtime. Together with generated code of the static structure out of UML 2 class diagrams, this substantially simplifies the creation of applications and leads to a continuous development process from the analysis/design phase to implementation.

To realize this idea we designed and implemented an interpreter for activity charts, accompanied by a model importer and a simulation and debugging component, so that the behavior of an application in terms of the token flow of the activity charts can be executed and visualized. According to the used UML diagram types we call this tool ACTIVECHARTSIDE.

The structure of this paper is as follows: in the following section we briefly describe our approach and rank the introduced tool into it. We then exemplify its overall architecture in detail and give an example, which illustrates how to work with it. In the main part we present our tool ACTIVECHARTSIDE and describe the simulation and debugging functionality. Then we discuss our approach and the tool and compare them to other similar attempts in the field of MDA. The final section gives a conclusion and an outlook to further work.

OUR APPROACH

In this section we give a brief overview of our approach called ActiveCharts, which is necessary to introduce the aforementioned tool and to understand the environmental settings in which the tool is used. For further information see [Sarstedt et al., 2005b] and [Sarstedt et al., 2005a].

Our project aims at simplifying software development by reusing large parts of analysis/design artifacts for

implementation. These reusable artifacts are class diagrams for modeling the static structure of a system and activity charts for specifying the dynamic behavior of those classes. We use abstraction in terms of modeling instead of coding not only for the static structure of an application, but also for its behavior.

By allowing the developer to use regular code within UML 2 activity diagrams for implementation of actions, we tightly integrate modeling and coding tasks while deliberately leaving the respective balancing between code and model to the developer.

The modeled control flow and the structure within the class diagrams are platform independent (corresponding to a platform independent model (PIM) in the sense of MDA), whereas the actions realized in code determine the platform specific parts of the design artefacts, called platform specific model (PSM). We have a well defined interface between elements of the PIM respectively PSM, thus. For this reason, our approach is in between the two main interpretations of MDA: the elaborationist [Kleppe et al., 2003] and the translationist [Mellor and Balcer, 2002] approach. The main difference between these approaches is the level at which a developer can change the functionality of an application. In the elaborationist approach a developer can elaborate the PIM, the PSM, and the code. In the translationist approach the developer can only work out the PIM, which is translated directly into code. In our approach the developer changes the functionality of the system within the PSM, but not within the generated code. Our approach is therefore not elaborationist. It is not translationist either, because the developer can change the behavior within the PSM using regular code.

Architecture of the ActiveCharts Approach

Figure 1 shows an overview of our simulation and visualization architecture. As it is common in modern software development processes, the static structure of a system is modeled using UML 2 class diagrams [UML, 2004]. These models – drawn in Microsoft Visio 2003 using special shapes – are translated into code by a generator (shown as the “Class Generator”-tool in the figure). The generated code implements all attributes and associations shown in the diagram, including code to handle modifications (i.e., addition and removal of objects) of those relationships. Because partial classes [Microsoft Developer Network, 2005] are used, additional C# code adding methods or other attributes can (and should) be written in separate files. The Microsoft C# compiler merges all matching class definitions when compiling the files, leading to easier development cycles because modifications of the static

structure, and therefore regeneration of its code, leaves custom C# code untouched. If the generated classes are to be extended for some details such as attributes, methods or super classes which are not to be shown in the models, it is possible to use separate partial class definitions that will not be affected by the recreation of code. By using this feature, generated code and implementations of the developer are clearly separated.

It is important to note that the class diagrams we use should preferably not contain any method declarations. This is neither unusual nor a restriction since we consider the class diagrams to basically provide a conceptual view on the system, omitting any unnecessary implementation details; these are added using custom code. Other authors in the field of OOA/OOD in principle agree in our opinion [Larman, 2005].

Application control flow is modeled using UML 2 activity charts (see “Dynamics” in figure 1). Therefore, each class that should have its own behavior has an associated activity, describing its functionality. UML actions – more precisely, UML `CallBehaviorActions` – are used to invoke custom C# code written by the developer. In figure 1 this is indicated by a `CallBehaviorAction` named “DoSomething” and its associated method declaration `void DoSomething()` in the lower left. To connect classes to activities we make use of a standard extension mechanism, UML tags, which designate key/value pairs [UML, 2004]. Activities are also drawn in Visio 2003 and translated into a XML representation by our tool. When the compiled program is finally executed, a runtime component (“ActiveCharts Runtime”, implemented as a dynamic link library) reads the model file and executes the activities when needed. During execution, custom code is called if an UML `CallBehaviorAction` is reached in an activity.

To integrate import, visualization and debugging of activity executions, the `ACTIVECHARTSIDE` has been developed. When the IDE is running, activity executions can be controlled and visualized. If the project is executed without the IDE, the application runs stand-alone without visualization and flow debugging possibilities. In this paper we focus on the IDE.

An Example

Figure 2 shows the static structure of a simple two hand molding press. It consists of a single controller which is associated with a piston, which can move up and down. The movement of the piston is initiated by two buttons to be pressed by the left and right hand, respectively. Its current position is reflected by an attribute called `position`.

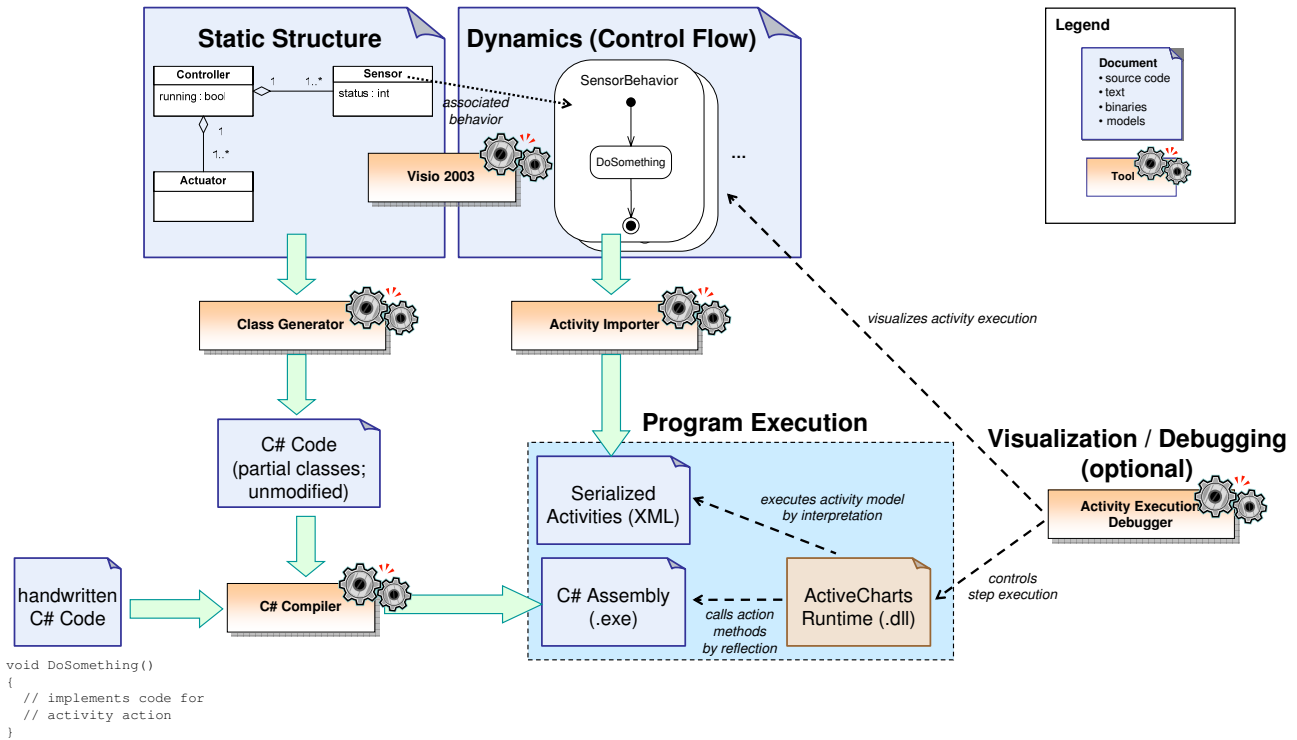


Figure 1: ActiveCharts Architecture



Figure 2: Static Structure of a Molding Press

The requirements for this molding press are as follows:

- The piston starts moving downwards if its two buttons are pressed within one second.
- When the piston moves downwards and any button is released before the “Point of no return” is reached (which is located at 3/4 of the total distance), the piston stops and moves upwards again to its starting position.
- If the “Point of No Return” has been reached, the piston continues moving downward, even if any button is released afterwards.
- When the piston has reached the bottom position it stops and returns to its starting position.

These requirements are implemented by the activity chart in figure 3 which shows the behavior for the press-controller. The link between the class and its behavior is established by using the UML tag specification “{behavior=PressControllerBehavior}” which

is shown in figure 2 below the class name for the press-controller class.

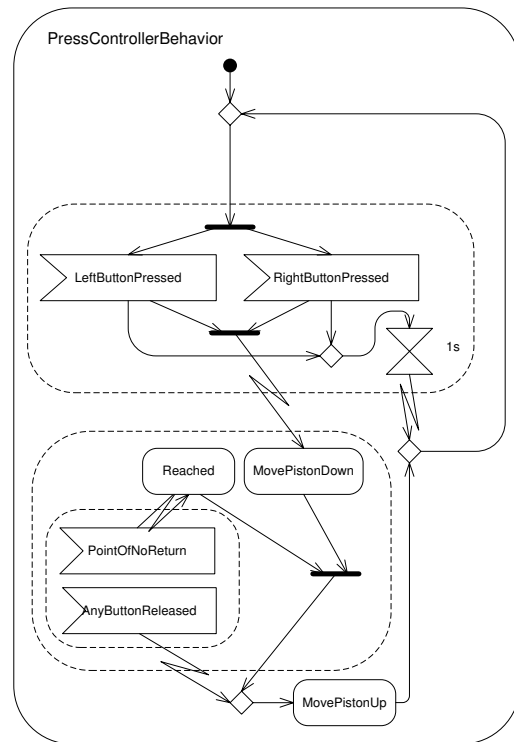


Figure 3: Behavior of the Press-Controller

When the press is instantiated (there is only one object for it) its associated behavior is started, yielding a new activity execution. The controller object is called the “context class” for this execution, since all actions and guards are evaluated in this context. Application flow begins at the `InitialNode` (● symbol) of the activity. At the following `ForkNode` (—) the flow is split into two parallel branches each one activating an `AcceptEventAction` (◻) which waits for a button pressed signal to be received (for a complete reference of all UML elements used refer to [UML, 2004]).

Both actions are contained within an `Interruptible-ActivityRegion`, depicted by a dashed rectangle with rounded corners [UML, 2004]. If any one button is pressed, a timer gets activated waiting for one second. If the other button is not pressed within this time span, the region is aborted, meaning that all actions contained in it are terminated. Flow then proceeds to waiting for button presses again.

If both buttons have been pressed accurately, flow proceeds to the action “MovePistonDown” which is implemented by handwritten code that basically calls the activity “MoveDown” shown in figure 4. This behavior contains a loop, incrementing the piston position and therefore moving it downwards.

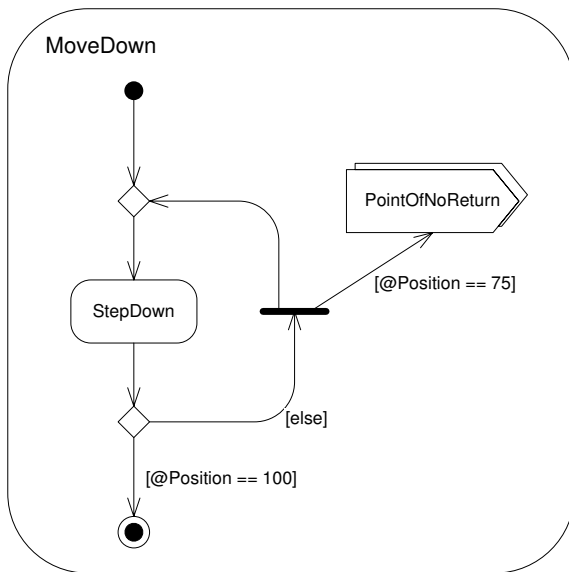


Figure 4: Move Down Activity for Piston

If an according position has been reached, a “PointOfNoReturn” signal is sent preventing a button release to abort the piston movement in the controller (see figure 3). This is also achieved by using an `InterruptibleActivityRegion` in the model.

The ACTIVECHARTSIDE

The ACTIVECHARTSIDE (see figure 5) is a tool that supports software development as described by MDA. It consists of three main components (also see the corresponding tool icons in figure 1):

1. Activity Importer

This component translates activity and class diagrams into a XML-based representation and afterwards creates an in-memory metamodel of the contained activities. This metamodel is then used for execution.

2. Class Generator

This code generator is used to translate classes together with their attributes and associations into code.

3. Activity Execution Debugger

This component is responsible for visualization of the simulation. The crucial point of this task is the communication with Visio (by using the COM automation interface) that is required for animating activity diagrams. The IDE acts as the superior manager for the “ActiveCharts Runtime” (which manages the execution of activities, see figure 1) and offers a visual simulation control panel to the developer.

The simulation and debugging process is usually started by running the project’s executable. The ACTIVECHARTSIDE calls the main method of the executable where the initialization of the project should be done. For every instance of an active class, an activity execution is created and the corresponding diagram is opened and used for animation of control and data flow. Whenever an action is reached, the “ActiveCharts Runtime” calls the corresponding method of the context class instance.

A special feature of the ACTIVECHARTSIDE is the “standalone simulation” of activities avoiding any method calls. This feature enables simulation of diagrams where no classes have been defined, i.e., the methods representing actions do not have to be implemented. This is especially useful for early prototyping of activities.

Token Flow Visualization

The interpretation of models and the computation of steps following the modeled control flow is discussed in detail in this section. When using the IDE’s debugger-controlled visualization for interactive step-by-step execution, both the current state of an activity execution

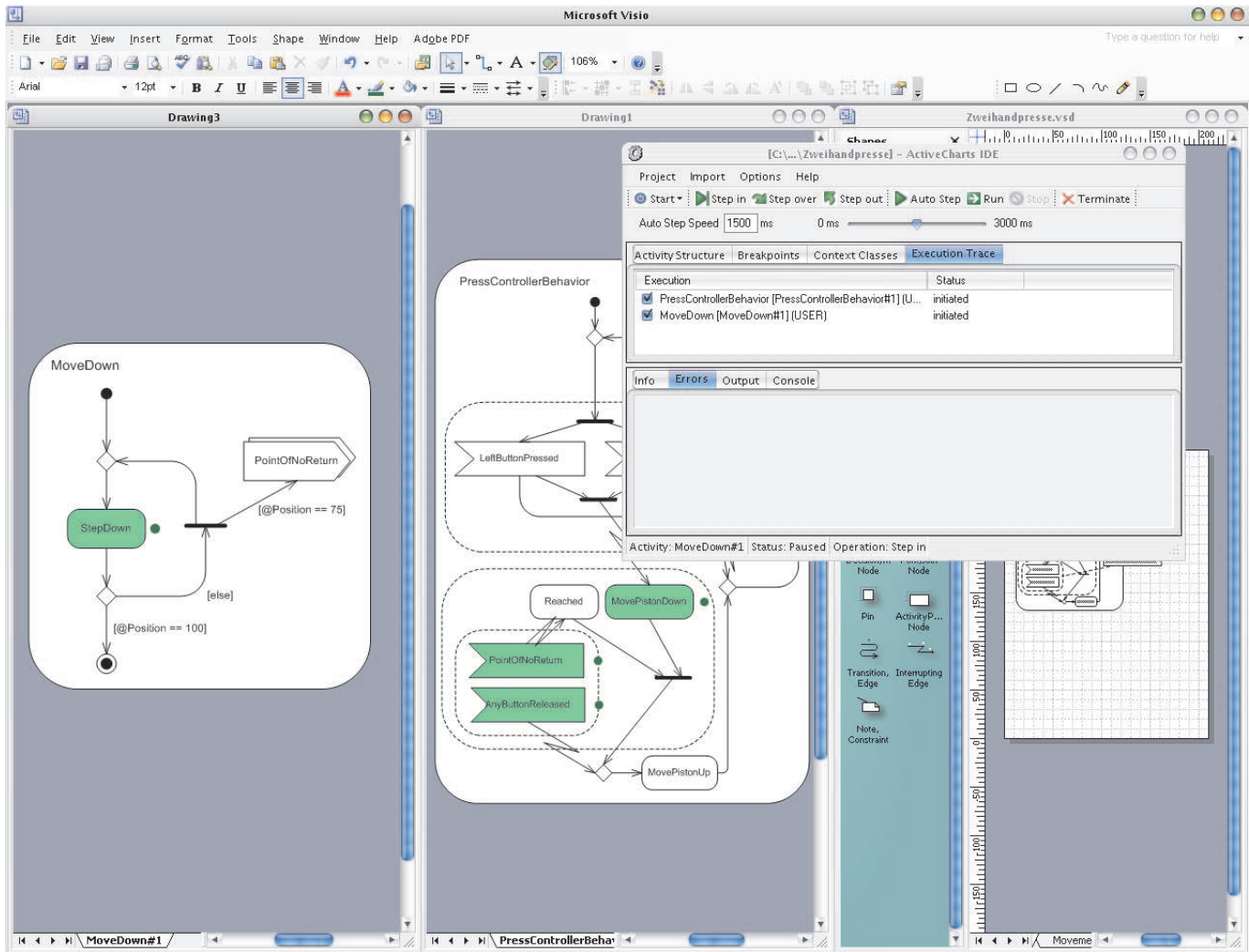


Figure 5: ACTIVECHARTSIDE and Visio 2003

and the changes caused by the last atomic step are displayed by animations in the activity diagrams. In case of continuous stepping, the creation or deletion of every token can be observed as well as the flow of tokens along the edges of the simulated activity.

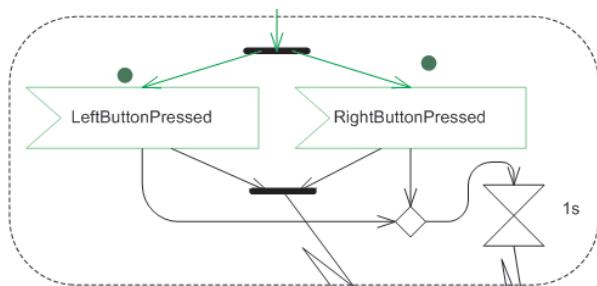


Figure 6: Visualization of a Token Flow and Creation of two action executions

Whenever a token flows from a source position towards a target (see figure 6), the whole path is displayed by changing the color of the affected edges. Actions for which executions have been created are shown in different colors, too.

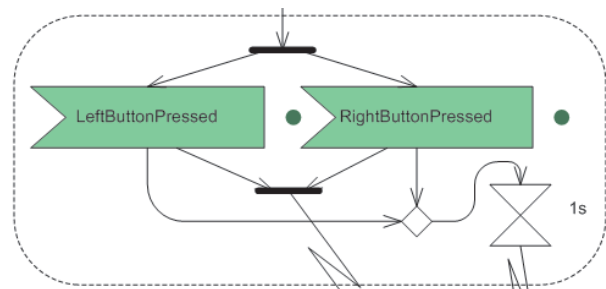


Figure 7: Visualization of Enabling two Elements

After the creation of an action execution, it is enabled.

This is shown by positioning the token next to the enabled element and by changing the element's appearance to indicate this action as currently executing (see figure 7). Enabling an action execution means creating a thread and calling the method that implements the action. While the code for an action is executed, the IDE waits for the called method to complete. After termination of the method (and hence of the action execution), tokens are displayed on each outgoing edge where they remain until they flow to the edges' targets or are destroyed (see figure 8, right part).

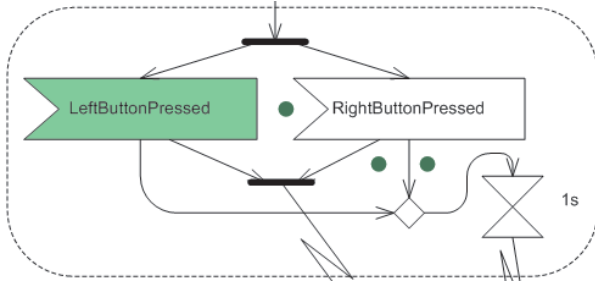


Figure 8: Visualization of an Element's Termination

Tokens are destroyed when an activity's final node is reached or when an `InterruptibleActivityRegion` containing the element to which a token is associated is left via an interrupting edge. In this case, all tokens inside the `InterruptibleActivityRegion` disappear and all nodes located inside are shown as "aborted". Running actions are canceled by stopping the thread that has been created for the action execution (see figure 9).

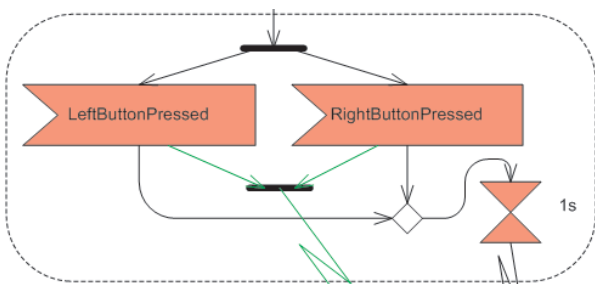


Figure 9: Visualization of aborted Elements after both Buttons have been pressed within a Second

Each situation illustrated by figures 6 to 9 can be observed when executing the project executable using the "Step-in" or "Step-over" functions. In the next section, we describe the functions provided for controlling an execution in detail.

Controlling the simulation

There are various possibilities to control step execution with the "step toolbar" of the `ACTIVECHARTSIDE` "Activity Execution Debugger"-component:



- The **"Step-in"** function performs a single step of execution. When an initial step takes place in a newly created execution, a new Visio document is automatically opened for animation. If there is already an execution open for an activity, only an additional tab for the new execution is created. All executions of a specific activity are distinguished by unique identifiers of the form

<name of activity>#<execution number>

(for an example, see the bottom left of figure 5: "MoveDown#1"). When an execution terminates (i.e., by reaching an `ActivityFinalNode`), the tab resp. document is automatically closed.

- The **"Step-over"** function allows to skip steps of an execution. When pressing "Step-over" just after a `CallBehaviorAction` has been enabled, all steps in this subactivity are not visualized. Note that steps occurring in other executions are nevertheless shown, which means that the effect of "Step-over" may be delayed.
- When pressing **"Step-out"** all remaining steps in the current execution are no more shown.
- **"Auto-step"** performs "Step-in" automatically in a loop. It can be interrupted by using "Stop". The step-delay can be adjusted using the slider and textbox below the toolbar.



- When pressing **"Run"**, all executions are executed without visualizing any token flow. Run may be interrupted by "Stop" or when a breakpoint is reached.
- The **"Stop"** button interrupts "Auto-step", "Run" or the implicit continuous call of steps that are performed when "Step-over" or "Step-out" is done. The activity execution is recovered, i.e., the number and position of tokens is updated and the last step is visualized. The information about steps that have been made in between are not displayed.
- The **"Terminate"** button aborts the project's executable and terminates all activities.

Characteristic Debugger Tasks

Beyond controlling and visualizing execution steps, the ACTIVECHARTSIDE supports characteristic debugger tasks allowing the user to explore data structures and values of context class objects at runtime. Using reflection, the whole context class as well as all of its associations can be viewed and retraced by the IDE. In future versions, even the manipulation of context class members will be supported.

An execution trace shows all running and terminated activity executions in a tree structure that represents caller/callee relationships between activities by assigning a called activity under the calling activity as a child element. However, this relationship does not mean that a called activity is terminated when the calling activity terminates.

The execution trace is also used to select activity executions to be watched. By checking or unchecking an activity execution, the debugger is told to visualize steps for executions or to discard them. Visio-windows that refer to unwatched activity executions are closed and remaining windows are resized to a maximum in order to offer a high degree of clarity for the visualization.

The ACTIVECHARTSIDE supports breakpoints. Breakpoints are set by clicking elements in an activity chart. Whenever steps are performed automatically in a loop (see “Step-over”, “Step-out”, “Auto-Step” and “Run” in the previous section), this loop is ended immediately when reaching an element that is associated with a breakpoint.

DISCUSSION AND RELATED WORK

In our approach and, on that account, in our IDE we use activity charts to model the behavior of an application. Another possible and common way of modeling the functional behavior of software systems are state charts. They are popular for modeling in the field of embedded systems. A multitude of tools have been established to design, simulate and debug embedded systems (e.g. Stateflow [The Mathworks – Stateflow, 2005] or Statemate [I-Logix – Statemate, 2005]). The possibility of code generation out of the used diagrams — mainly state charts — is best available technology and widely used.

Nevertheless, we decided to use UML 2 activity diagrams to model the behavior of an application for several reasons: first of all, we want our approach to be able to be used not only for the creation of embedded

systems, but also for the design and building of information systems like a reservation system, for example. We believe, activity diagrams are more suitable for this application area than state charts because of the following reasons: in activity diagrams there are explicit *data flows*, which can be used to model the data used in an application, whereas state charts do not offer this potentiality. This is an important issue when designing applications that handle lots of data, considering for example bookings in the above mentioned reservation system. Moreover, coming from use cases — as many software development processes for object oriented applications propose, e.g. the Rational Unified Process (RUP), [IBM – RUP, 2005] — it is a natural proceeding to expand use cases with activity diagrams. The explicit actions used in these diagrams can be seen as interactions between the different roles with the system, among other things.

In our ACTIVECHARTSIDE tool we use Microsoft Visio 2003 to draw the diagrams. We tested several — as the producer claim — UML 2 compliant modeling tools like EclipseUML 2.0 [Omondo – EclipseUML, 2005], Poseidon 3.0 [Gentleware – Poseidon, 2005] and InnovatorObject from Innovator 8.1 [MID – Innovator, 2005], but non of these satisfied us, as the use of elements especially within the activity diagrams was restricted. It was not possible to draw interruptible regions or timing signals for example, which are in our opinion fundamental for the usability of these diagrams. For this reason we decided to use a comfortable drawing application, Visio 2003.

Other research groups in the field of MDA propose their approaches and tools: in the area of web enabled applications the DASL project [Goldberg, 2002] presents a new modeling language (Distributed Application Specification Language), which combines UML state charts and self created UML like charts with additional Java-like code. They do not use activity charts, nor support any simulation or visualizing component during analysis or design phases.

Within the C-Lab [Schattkowsky and Müller, 2005] project, UML state and sequence diagrams are used to model the behavior of an embedded system. These diagrams are executed on a technology called UML Virtual Machine, according to the Java Virtual Machine. It is possible to run the models on different platforms, therefore. It is not possible to enrich singular actions with additional code within their approach.

Another tool in the MDA context is Fujaba [Fujaba, 2005], a tool suite developed at the University of Paderborn, Germany. The Fujaba Tool Suite supports the generation of Java sourcecode out of the design, which consists of UML class diagrams and so called story diagrams, which are not an element of the

UML. They are a self defined combination of activity and collaboration diagrams. A dynamic object browsing system (DBOS) is integrated in Fujaba for simulation purposes. The objects with their attributes and their relations are visualized within an object diagram.

ACTIVECHARTSIDE is not restricted to be used with Visio for visualization. As it is in large parts independent from the used editor, only a few interfaces have to be re-implemented with arguable effort for the IDE to be able to use another editor for drawing and simulating activity charts.

CONCLUSION AND FUTURE WORK

In this paper we introduced a new simulation environment called ACTIVECHARTSIDE within our approach to Model-Driven Architecture: by simulating and visualizing UML 2 activity charts, which are used to model the control flow of an application, the creation of software is benefitted. By means of an example we introduced our simulation environment and its architecture and showed in which way the UML 2 activity diagrams can be simulated by visualizing the token flow.

We used our tool in a practical course in the summer term of 2005 at the University of Ulm. This led to an intensive testing and evaluation of the ACTIVECHARTSIDE, which resulted in improvements of the usability and robustness of the tool. To ensure quality during modeling, we will enrich the tool with automated syntactic and semantic checks on the models.

References

- [Fujaba, 2005] Fujaba (2005). Fujaba: Tool Suite 4. <http://www.fujaba.de/>.
- [Gentleware – Poseidon, 2005] Gentleware – Poseidon (2005). Gentleware: Poseidon. <http://www.gentleware.com/index.php>.
- [Goldberg, 2002] Goldberg, B. (2002). Practical Guide to the Ace Application Specification Languages. Sun Microsystems.
- [I-Logix – Statemate, 2005] I-Logix – Statemate (2005). I-Logix Statemate website. <http://www.ilogix.com/statemate/statemate.cfm>.
- [IBM – RUP, 2005] IBM – RUP (2005). IBM: Rational Unified Process. <http://www-306.ibm.com/software/awdtools/rup/>.
- [Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture – Practice and Promise*. Addison–Wesley.
- [Larman, 2005] Larman, C. (2005). *Applying UML and Patterns*. Prentice Hall.
- [McNeile, 2004] McNeile, A. (2004). MDA: The Vision with the Hole? <http://www.metamaxim.com/download/documents/MDAv1.pdf>.
- [MDA, 2003] MDA (2003). Object Management Group, MDA Guide 1.0.1, Document 03-06-01, June 2003.
- [Mellor and Balcer, 2002] Mellor, S. J. and Balcer, M. J. (2002). *Executable UML: A Foundation for Model Driven Architecture*. Addison–Wesley.
- [Microsoft Developer Network, 2005] Microsoft Developer Network (2005). Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes. <http://msdn.microsoft.com/msdnmag/issues/04/05/c20/default.aspx>.
- [MID – Innovator, 2005] MID – Innovator (2005). MID: Innovator. <http://www.mid.de/de/>.
- [Omondo – EclipseUML, 2005] Omondo – EclipseUML (2005). Omondo: EclipseUML. <http://www.omondo.com/>.
- [Sarstedt et al., 2005a] Sarstedt, S., Kohlmeyer, J., Raschke, A., and Schneiderhan, M. (2005a). Targeting System Evolution by Explicit Modeling of Control Flows using UML 2 Activity Charts. In *Proceedings of the International Conference on Programming Languages and Compilers (PLC '05), Technical Session on Support for Unanticipated Software Evolution*.
- [Sarstedt et al., 2005b] Sarstedt, S., Raschke, A., Kohlmeyer, J., and Schneiderhan, M. (2005b). A New Approach to Combine Models and Code in Model Driven Development. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '05), International Workshop on Applications of UML/MDA to Software Systems*.
- [Schattkowsky and Müller, 2005] Schattkowsky, T. and Müller, W. (2005). Model-based design of embedded systems. In *Proc. Design Automation and Test in Europe DATE 2005*, Munich, Germany.
- [The Mathworks – Stateflow, 2005] The Mathworks – Stateflow (2005). The Mathworks – Stateflow website. <http://www.mathworks.com/products/stateflow/description1.jsp>.
- [UML, 2004] UML (2004). Object Management Group, UML 2.0 Superstructure Specification, Document 04-10-02.