Model-Driven Development with ACTIVECHARTS

Tutorial

Stefan Sarstedt <sarstedt@acm.org>
Url: http://activecharts.informatik.uni-ulm.de
eMail: mdd@informatik.uni-ulm.de

Summary

This tutorial aims to give an overview over the ACTIVECHARTS project currently under development at the Programming Methodology and Compiler Construction group in the Department of Computer Science at the University of Ulm, Germany. Our project aims at simplifying software development by reusing large parts of analysis/design artefacts for implementation. These reusable artefacts consist of UML 2 class diagrams for modeling the static structure of a system and UML 2 activity charts for specifying the dynamic behavior of those classes. Activities are associated with context classes and interpreted by a runtime component which facilitates development because the modeled application control flow has not to be implemented again using handwritten code.

Our toolset is based on C#/.NET 2 for programming, and Visio 2003 for drawing class and activity diagrams.

Table of Contents

1 Introduction

Modern software development processes include more or less comprehensive analysis and design phases where various models describing the static structure and the dynamics of a system are created. The primary motivation is to gain a deeper understanding (for discussions with developers and other stakeholders) and to provide documentation for the system. Starting from these models, the implementation is created. Unfortunately, these models are often not maintained during later phases of the project which leads to divergence of technical documentation and the final implementation, which furthermore impedes system maintenance.

Model Driven Development [MDD] tries to bridge the gap between analysis/design and implementation by enriching the design artifacts (e.g. UML 2 models) to enable building parts of the application out of these models automatically. A recent initiative, called MDA (Model Driven Architecture [MDA]), even has the ultimate goal to generate the whole application out of models. MDA proposes to differentiate between platform independent models (PIMs) which descibe the systems' functionality independently of the underlying technical infrastructure. These models are then transformed by tools and transformation languages to platform specific models (PSMs), which include information about specific technology, e.g. Java/C# or MySQL/Oracle.

In concretizing the idea of model-driven development, it becomes obvious that the graphical information on its own is not sufficient to properly describe the functionality of a system. Additional languages have been introduced to describe imperative parts of the software, but these mostly exotic languages are only seldom used. We also think that it is not feasible to solely rely on graphical abstractions, since models quickly become too complex to understand, leading to similar effects as it is the case with unreadable code [SKRS1] [SKRS2].

Starting from these problems, our approach to Model Driven Development combines models from the analysis/design phases and regular (handwritten) C# code [CSharp] from the implementation phase in a new way. We model the control flow of an application with UML 2 activity diagrams [UML05] which are interpreted by a runtime environment. Some actions are realized in C# code which will be executed when the appropriate action is invoked. Analysis/Design artifacts are therefore seamlessly reused for implementation which leads to narrowing the gap between these phases, better documentation and therefore easier maintenance.

The structure of this document is as follows. In section 2 we describe the architecture, features and prerequisites of our system. Section 3 deals with basic modeling tasks followed by a description of the ACTIVECHARTSIDE in section 4. The following two sections further deepen the concepts by addressing advanced modeling and simulation capabilities of our system. The final sections deal with FAQ, a technical reference and pointers to additional literature.

INTENDED AUDIENCE

We assume the reader to be familiar with basic UML 2 modeling, especially with concepts of class and activity diagrams as well as object oriented programming with C#. Knowledge of Microsoft Visual Studio .NET 2005 and Microsoft Visio 2003 are also helpful.

2 Architecture and Overview

The main goal of our approach is making the control flow of an application explicit in UML 2 activity diagrams retaining the possibilities of regular "hand-written" code implementing activity actions. It is no more necessary to (re-)code this control flow in C# since the models are executed by a runtime component. This degree of functionality described by models versus functionality described by code can be chosen freely by the developer, which should improve acceptance of modeling tasks and is particularly useful because not every aspect of a system can and should be modeled graphically. Method return values can also influence paths taken in the diagram.

Figure 2-1 shows an overview of our approach. As it is common in modern software development processes, the **static structure** of a system is modeled using UML 2 class diagrams. 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 (addition and removal of objects) of those relationships. Because partial classes [MSDN] are used, additional C# code adding methods or other attributes can (and should) be written in separate files. The ordinary 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. It is important to state that the class diagrams we use should not contain any method declarations, i.e. they are ignored by our code generator. This is not unusual 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 have similar opinions [Larman04].

Application **control flow** is modeled using UML 2 activity charts (see "Dynamics" in the figure). Therefore, each class that should have its own behavior has an associated activity, describ-

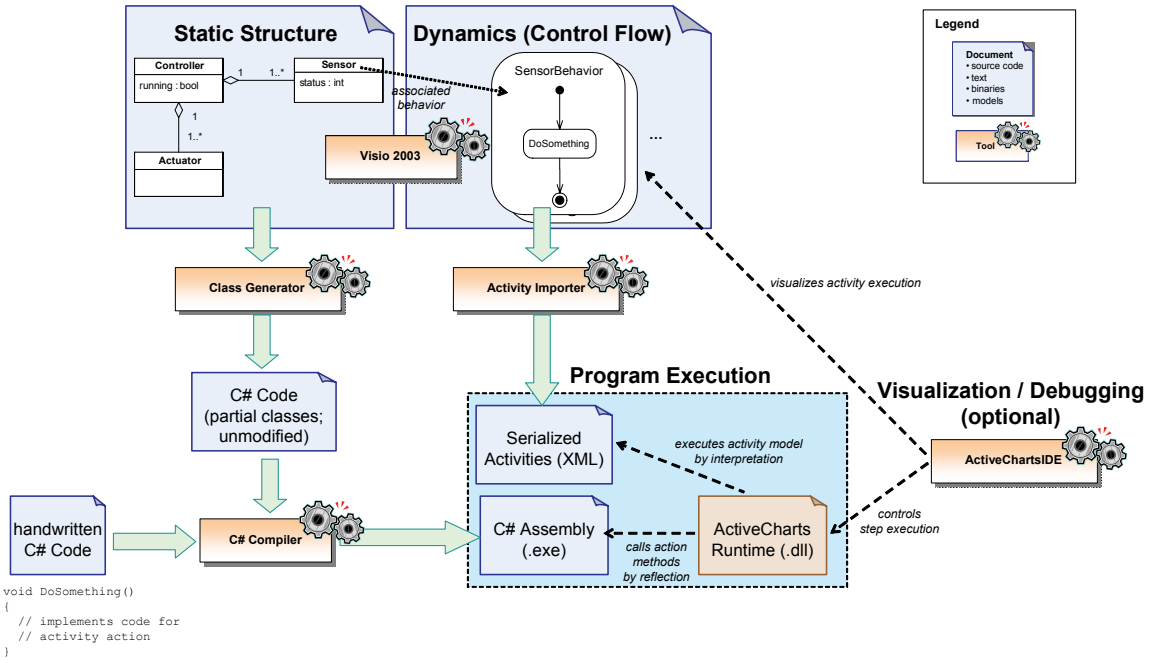


Figure 2-1: ACTIVECHARTS approach

ing its functionality. UML actions – more precisely, UML `CallOperationActions` – are used to invoke custom C# code written by the developer. In Figure 2-1 this is indicated by an action named "DoSomething" and its associated method declaration "`void DoSomething()`" in the lower left. To connect classes to activities we make use of UML tags, a standard extension mechanism of the UML (not shown in the figure, see section 3.4 for details). 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 ("ActiveChart 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 CallOperationAction is reached in an activity.

To integrate import, visualization and debugging of activity execution, a tool called ACTIVECHARTSIDE (see section 4) has been developed. When started, activity executions can be controlled and shown in the diagram previously drawn. Values of data tokens flowing between actions can also be examined.

After a short summary of the features and prerequisites of ACTIVECHARTS in the remainder of this part, section 3 shows how to set up and use the tools for basic modeling tasks.

FEATURES

- Importing UML 2.0 class diagrams and UML 2.0 activity charts from Visio 2003
- Code generation from UML 2.0 class diagrams, untouched custom code
- Interpreting UML 2.0 activity charts, supported elements include
    - initial, flow final and activity final nodes
    - fork, join, decision and merge nodes
    - calloperation- and callbehavior actions
    - both control and object flow
    - nested activities (i.e. activities calling other activities)
    - interruptible regions
    - accept event actions with signal trigger and time trigger
- Debugging activity charts

PREREQUISITES

- Microsoft .NET 2 SDK
- Microsoft Visual Studio .NET 2005 (optional)
- Microsoft Visio 2003 for drawing class and activity diagrams and for simulation functionality

CONVENTIONS

Things you should especially take care of are written this way.

Keyboard commands, e.g. *Ctrl+F1* are marked in italics.

```
Code is written in a typewriter font.
```

3 Basic Modeling

This section introduces basic modeling of static structure (section 3.2), control flow (section 3.3) and how to connect both formalisms together.

3.1 Getting Started with Visio 2003

ACTIVECHARTS uses special Visio 2003 stencils (consisting of graphical elements, called "shapes", e.g. Class, Activity, Relationship) for modeling tasks. For ease of use, a Visio template named "`ActiveCharts.vst`" – which already contains all relevant shapes – is provided. After opening the template, a blank drawing like the one shown in Figure 3-1 is created.

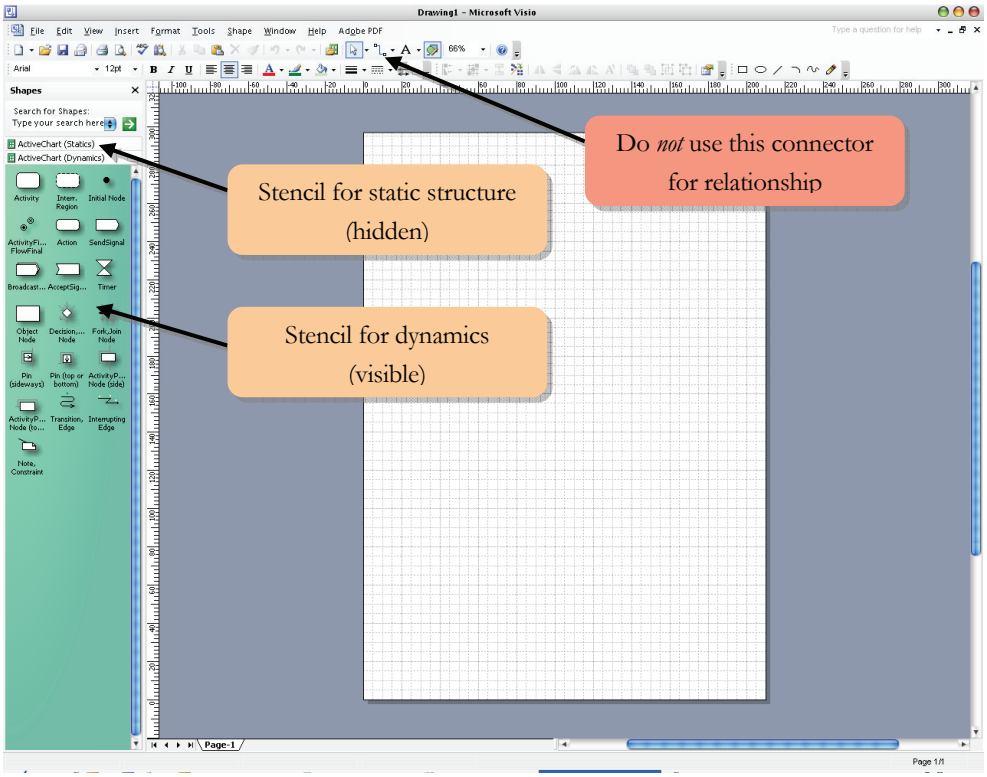


Figure 3-1: ACTIVECHARTS Visio 2003 Template

To start modeling, drag the desired shape on the drawing surface. There are also shapes for connecting your shapes, e.g. "Transition Edge" (for modeling control flow in activities) or "Routable Relationship" (for associations between classes).

Do *not* use other means to connect the shapes like the built-in Visio "Connector Tool", shown in Figure 3-1 in the topmost toolbar, since ACTIVECHARTS will not recognize those connections, even though they look similar.

3.2 Modeling the Static Structure

A specification of a software system consists of a description of its static and dynamic parts. The coarse grained static structure (architecture) describes components and their relationships. For intra component modeling, i.e. the fine grained static structure, it is common to use UML class diagrams. There are several levels of abstraction a class diagram can be utilized [Fowler03] for this purpose:

1. **Conceptual Level**
   Used for modeling concepts/"things"/roles of a domain. This is how class diagrams should be initially used in the analysis phase of a project to gain a deeper understanding of the area of application.
2. **Specification Level**
   Used for modeling interfaces between software components, independently of implementation specifics.
3. **Implementation Level**
   Modeling of software classes which correspond directly to programming language code (like C# or Java).

Our view of class diagrams is mainly the conceptual one, since this is the most beneficial level. Analysis classes can be reused independent of rapidly changing implementation technology, which is also the motivation for defining platform independent models (PIMs) in the notion of the Model Driven Architecture [MDA]. For pragmatic reasons we do not forbear to make use of data types, which strong advocates of conceptual modeling dictate. The additional layer of indirection of mapping abstract data types to their respective implementation types seems not useful in our approach, although it would not be complex to realize.

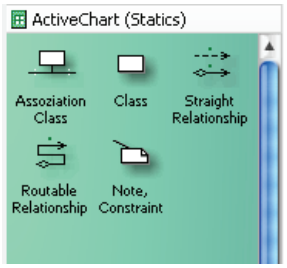


Figure 3-2: Statics stencil

To model the static structure of the inner workings of a system, the "ActiveCharts (Statics)"-stencil shown in Figure 3-2 is used. It consists of the following elements:

- Classes
- Straight Relationships and Routable Relationships
- Association Classes
- and Notes

The next sections describe how to model these elements in detail. For more convenient modelling ensure that spell checking and "AutoCorrect" options are turned off in Visio 2003.

3.2.1 Classes

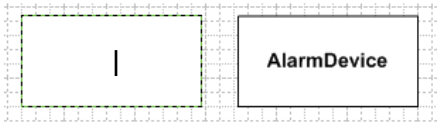


Figure 3-3: Naming classes

After dragging the class shape from the "ActiveChart (Statics)" stencil, you can name it by double-clicking the shape.


All class names are converted to uppercase by ActiveCharts, since this is the C# naming convention.

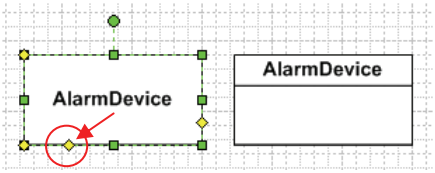


Figure 3-4: Adding attributes (a)

To add attributes, first single-click the shape, drag the yellow spot marked in Figure 3-4 upwards…

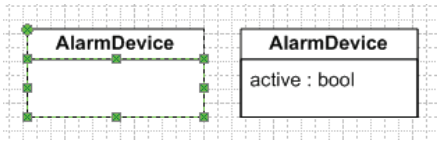


Figure 3-5: Adding attributes (b)

…then click on the attributes compartment (the rectangle below the class name) and either start typing (e.g. "`active : bool`") or press *F2* to get a text cursor to modify the text. Add additional attributes in the same compartment – by using *F2* to enter the editing mode.

Our Visio stencils do not check syntax or semantics of what you type for your attributes. Errors will arise later when trying to generate code from your models.

3.2.2 Relationships

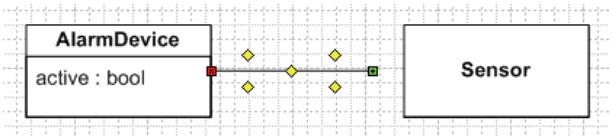


Figure 3-6: Connecting relationships

You can add relationships by dragging either the "Straight Relationship" or "Routable Relationship" shape to your classes. Notice that only ends marked red are successfully connected to the respective class.

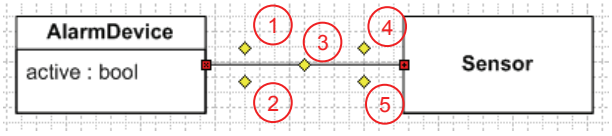


Figure 3-7: Multiplicities and roles (a)

You specify multiplicities and role names by adding text at the positions marked by a yellow spot (Figure 3-7). The positions (1) and (2) resp. (4) and (5) can be either used for multiplicities or roles, number (3) is used for association names. Click on the desired spot to enter text. To edit text already typed, single click the spot and press *F2*. This does not work for spot number (3); to enter or edit text here, you have to double-click the association.

If no multiplicities are defined, "0..1" is used as a default. If no roles are specified, the respective class name with a lowercase first letter is used. All association ends are public.

Figure 3-8: Multiplicities and roles (b)

Figure 3-8 on the left shows how multiplicities, role- and association-names are used.

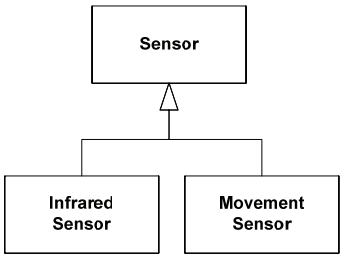
Figure 3-9: Aggregation and composition

You can also add aggregation or composition symbols to express whole-part-relationships, although our generator does not treat them in a special way at the moment. To specify the type of relationship (aggregation, composition, …), right-click the association and choose the appropriate type from the popup-menu.

Figure 3-10: Navigation

To provide navigation directions, you also use the popup-menu ("Navigable Forward"

and "Navigable Backward" types). These directions define visibilities within the modeled classes. In our example in Figure 3-10, the `AlarmDevice`-class can see `Sensor`-instances but not the other way round.

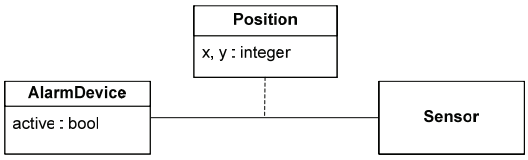


Generalizations can also be modeled by right-clicking the association and choosing the "Generalization"-type.

Figure 3-11: Generalization

Note that only single inheritance is allowed in our class diagrams, since the target language does not support multiple inheritance.

3.2.3 Association Classes

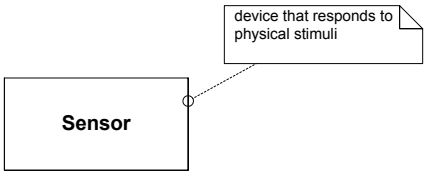


Association classes add additional information to associations. See the example in Figure 3-12, where position information (x- and y-coordinates) is associated with an `AlarmDevice-Sensor`-pair.

Figure 3-12: Association classes

Code generated for association classes consists of "`Pair<B,C>`"-ends, where `B` is the opposite end class and `C` the association class.

3.2.4 Notes



Notes add comments to shapes. They currently have no meaning for code generation.

Figure 3-13: Notes

3.3 Modeling the Application Control Flow

To model the dynamics of a system, the "ActiveCharts (Dynamics)" stencil is used, which is also available when you open the Visio template file "`ActiveCharts.vst`".

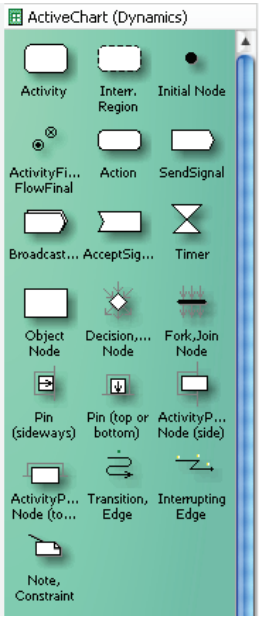


Figure 3-14: Dynamics stencil

Activities dictate the possible sequencing of Actions by connecting these and other kinds of nodes using ControlFlows and ObjectFlows. The following kinds of elements are treated in this section:

- **Control Nodes**
  ControlNodes influence flow of execution, e.g. by forking into parallel flows, or branching flow based on guard values.
  - InitialNode, ActivityFinalNode
  - ForkNode, JoinNode
  - DecisionNode, MergeNode
- **Actions**
  Actions are the primary elements of interest in activities since this is the place where some behavior execution is actually happening. There are different kinds of actions, the most important one being action calling custom code, i.e. methods that are implemented by a developer.
  - CallOperationAction, CallBehaviorAction
  - AcceptEventAction (Timer)
- **Flows**
  Tokens are transported to other nodes by passing edges.
  - ControlFlow

Advanced UML2 activity chart elements are discussed in section 6.

3.3.1 UML 2 Activity Chart Semantics

The semantics of UML2 Activity Charts is very loosely based on the token flow semantics of the well-known Petri Nets. Tokens – drawn in this document by green circles – tell where current locus of control reside. There can be many tokens active in one activity in parallel, meaning that behavior is concurrently executing.

According to the current UML2 specification [UML05], there are different steps when executing an action:

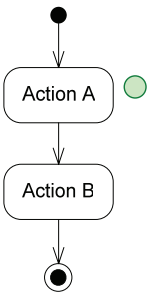


Figure 3-15: "Enabled"-Step

Enabled
If an execution of an action is *enabled*, it means that the action has been called and is currently running.

Note, that we only show CallOperationActions here for illustration; other kinds of actions (AcceptEventAction, SendSignalAction, …) which have other symbols behave the same way!

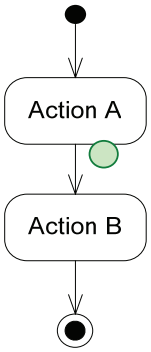


Figure 3-16: "Terminated"-Step

Terminated
After an action has finished executing, it is *terminated*.

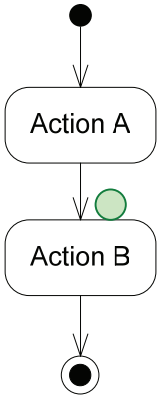
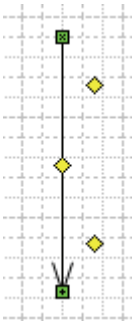


Figure 3-17: "CreateExecution"-Step

CreateExecution
For an action that is about to start ("Action B" in the figure on the left), is in a state called *create execution*. The next state for this action will be the before mentioned *enabled* step.

Tokens never rest at Control Nodes (like Decision, Merge, Fork, Join), except Intial Nodes. Transitions only occur between Actions (CallBehaviorAction, AcceptEventAction, SendSignalAction, …) and are always atomic.

If tokens appear to be on wrong positions during simulation you have to **adjust the token positions on the shapes** manually. All actions and edges have position markers for tokens:



Activity edges for example have markers for "upper" and "lower" token positions. After adjusting, you have to re-import your diagrams to reflect the new configuration.

3.3.2 Timers

Timers (more precisely: UML2 AcceptEventActions with a TimeTrigger) are used to suspend token flow for a specific amount of time. Figure 3-18 shows some examples of Timers:

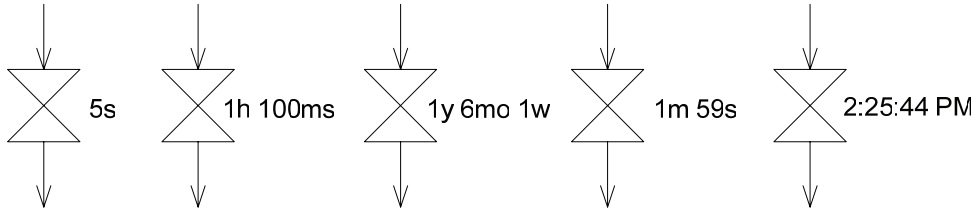


Figure 3-18: Examples of Timers

Differences are made between relative and absolute times. The first four examples show relative time specifications: when a token reaches the timer, it waits for the time to elapse, e.g. 1 minute and 59 seconds ("1m 59s"-caption). The example on the right ("2:25:44 PM") depicts an absolute time: when a token arrives, it waits for the clock showing "2:25:44 PM" and then continues flowing. The following table shows the abbreviations that can be used for timers in ACTIVECHARTS.

TIME SPECIFICATION	ABBREVIATION IN ACTIVITY DIAGRAM
Millisecond	`ms`
Second	`s`
Minute	`m`
Hour	`h`
Day	`d`
Week	`w`
Month	`mo`
Year	`y`
absolute Time	`hh:mm:ss PM (or AM)`

3.4 Connecting Static Structure to Dynamics

Activities describe behavior for UML classes. Therefore, each activity has to be associated with its corresponding class – named "**context class**" – which is achieved using a special UML tag [UML05], which is a well-known extension mechanism of the UML. Figure 3-19 shows such a link. Below the name of the class, the tag "`{behavior=<name of activity>}`" is found. Furthermore, the "Siren"-class is marked "active" in Visio 2003 by using its popup menu. This further indicates that this is a behavioral class.

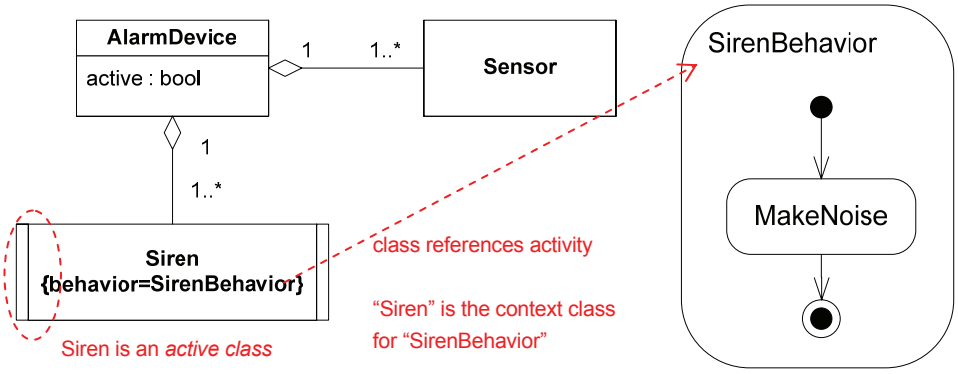


Figure 3-19: Connecting activities with classes

The class generator (see section 4.2) notices those specifications and generates special code to import the activity diagrams from the XML representation automatically. Additionally, a "`StartBehavior()`" method is generated for each active class, which a developer can call in his project to initiate the behavior associated with that class. This is normally not necessary since a behavior is started automatically upon creating an instance of a class. An optional UML tag called "`{autostart=false}`" can be used in the class to disable automatic starting of the behavior (see Figure 3-20). In that case, an additional call of the "`StartBehavior()`"-method must be made to initiate the activity.

Figure 3-20: Enable automatic starting of a behavior

3.5 Using Guards

Edges can be annotated with guards, which describe a condition under which a token can pass this edge. Guards can make use of attributes and properties defined in an activities' context

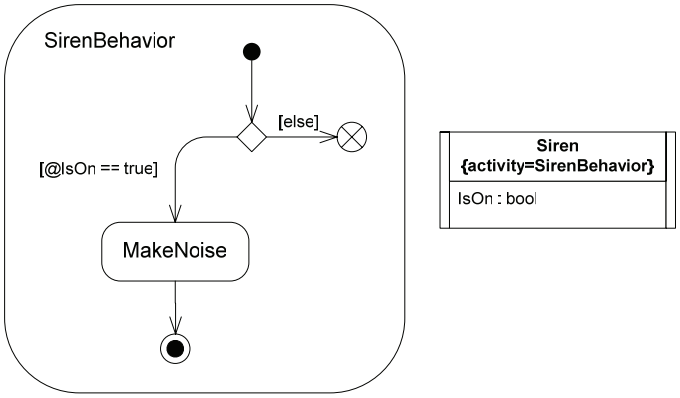


Figure 3-21: Using guards with context class

class.

Figure 3-21 shows how to use guards. All guards must be placed in brackets ("`[…]`"). Attributes or properties from the context class *must* be prefixed by '@' (e.g. "`@IsOn`"), otherwise the execution yields an error when evaluating the guard at runtime. The "**`[else]`**" guard can be specified to indicate an alternative flow, when no other edge can be passed. When writing guards, normal C#-syntax for boolean expressions can be used. Data flowing over edges can also be referenced, see section 6.2 for details.

4 Getting started with ACTIVECHARTSIDE

ACTIVECHARTSIDE is the central application for class diagram import, code generation, activity diagram import and serialization, as well as model simulation and debugging.

4.1 Creating a new Project

The first step for all tasks is creating an ACTIVECHARTS-project. Note that only one project can be created per folder. A project folder contains settings, DLLs and all generated C# source and XML files necessary for execution. When using an IDE like Visual Studio .NET, it is recommended that the ACTIVECHARTS project folder and the Visual Studio project folder (e.g., of a Windows Application) where the created files and additional code are compiled, are identical.

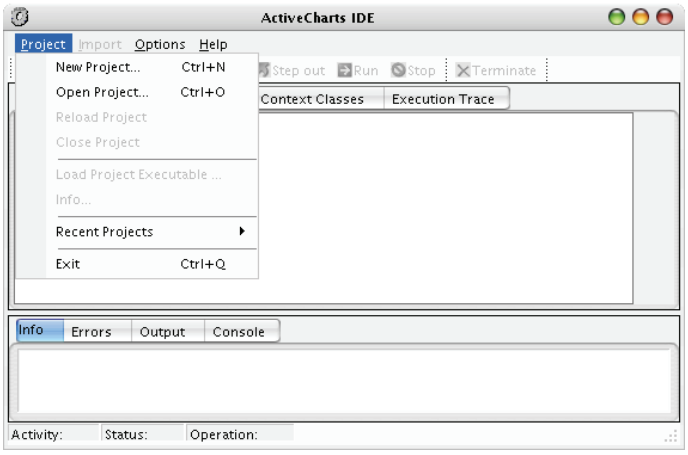
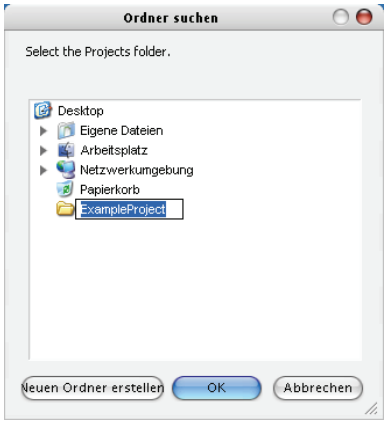


Figure 4-1: Project menu

To continue in this tutorial, create a new project in a folder called "ExampleProject":



When you look at the contents of the "ExampleProject" folder, there should now be the following files in it:

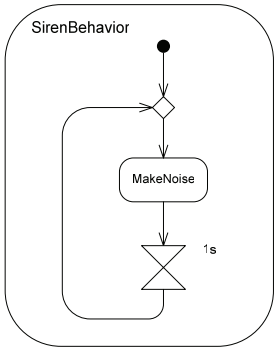
File	Description
`ActiveCharts.acp`	Main project file with some settings
`ActiveChartsConfig.xml`	Project specific settings for the execution engine

`ActiveCharts.dll`
`log4net.dll`
`ModelManager.dll`
`UmlMetamodel.dll`
`XmlImport.dll`
	DLLs needed for executing the project when used without the ACTIVECHARTSIDE

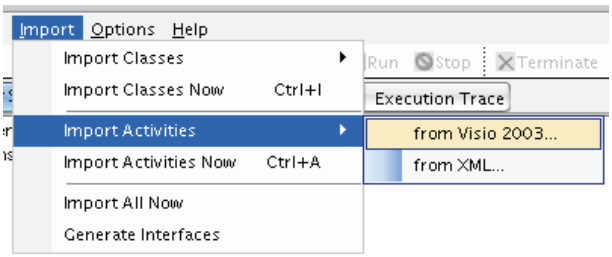
4.2 Importing and Executing Activity Diagrams

4.2.1 Importing a diagram

Draw a simple activity diagram using the Visio 2003 template mentioned in section 3.1 and save your new drawing in the project folder (it is not necessary to use the project folder to keep the drawings, but we recommend to do so):

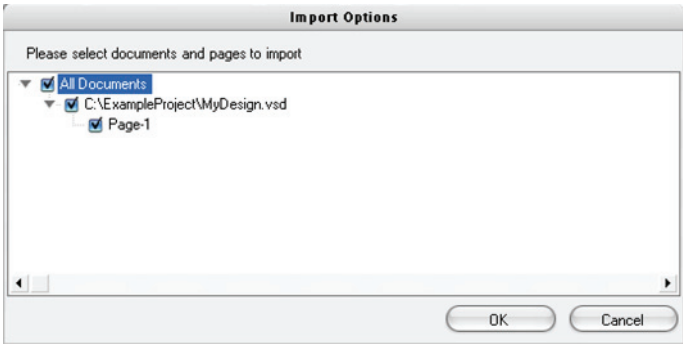


Now, import the diagram using the "`Import->Import Activities->from Visio 2003...`" menu entry:



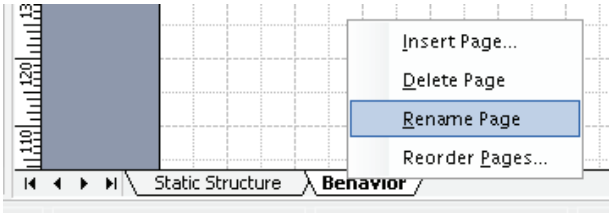
The following dialog will come up where you can select the Visio documents and tabs to import from. A Visio document can consist of multiple tabs which should be used to separate class and activity diagrams. Diagrams can also be distributed across multiple tabs; this way, large activities or class networks can be divided into coherent parts.

Since there is currently only one document and tab in our example, check all boxes:



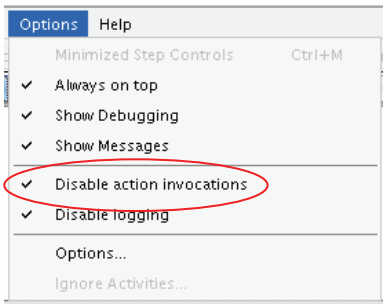
To prevent this dialog from showing up every time you want to import diagrams, you can use "`Import Activities Now`" (or its corresponding keyboard shortcut *Ctrl+A*) from the "`Import`"-menu. The last import settings you made are then used.

For easier identification of the corresponding tab(s), you should rename them by right-clicking the tabname at the bottom of the document and using the "Rename Page" menu entry. Use, for example, identifiers like "Behavior" or "Static Structure" for naming activity and class diagrams, respectively:



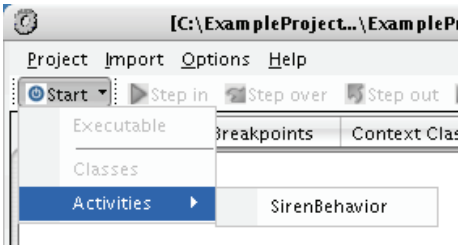
4.2.2 Executing the diagram

Before starting to execute the diagram, you have to make sure that CallOperationActions (like, "`MakeNoise`" in our activity) are *not called*, since there is currently no context class for "Siren-Behavior" available that contains code for this method (since we didn't create and compile one yet – see the next section for an example on this).

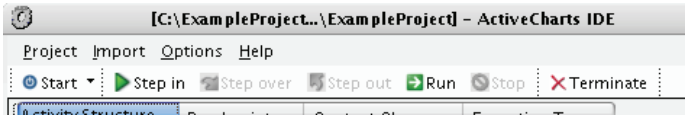


In fact, this setting is made *automatically* when starting an activity that has no context class associated, because it would otherwise result in a runtime error when the method is not found. When invoking such an activity, the above shown menu entry is furthermore disabled to prevent action calls accidently to be enabled.

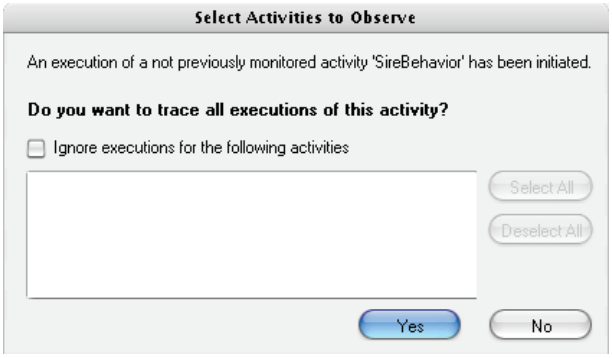
Like classes, activities must also be *instantiated*, every instance represents an **ActivityExecution** [UML05] that is executed concurrently. After ActiveChartsIDE imported those activities, you can create instances using the "`Start`" menu shown below:



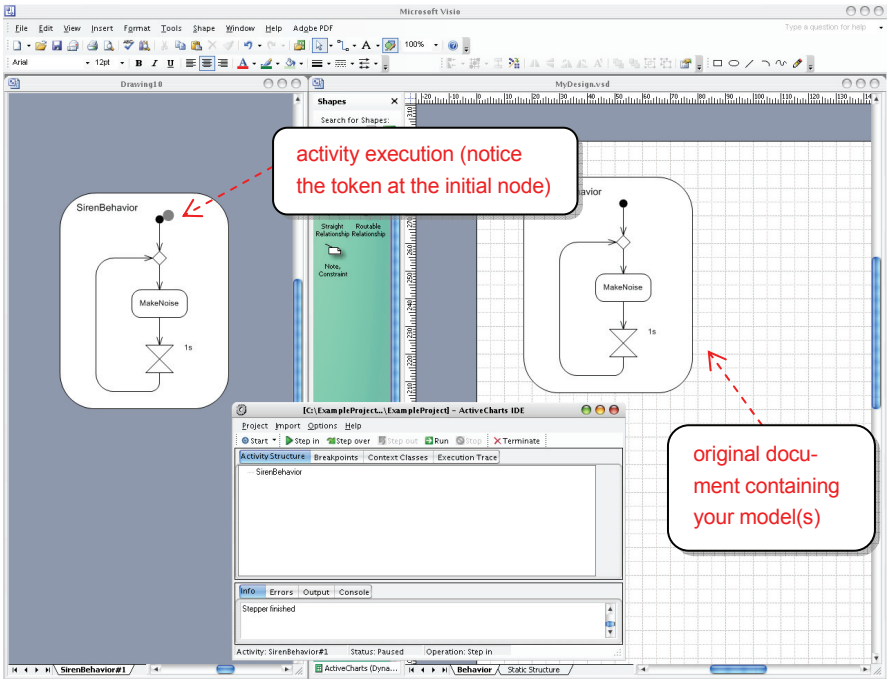
Select "`Start->Activities->SirenBehavior`" to create an instance of the activity we have drawn. Now that we have at least one instance that can be executed, the Step-Control-toolbar is enabled:



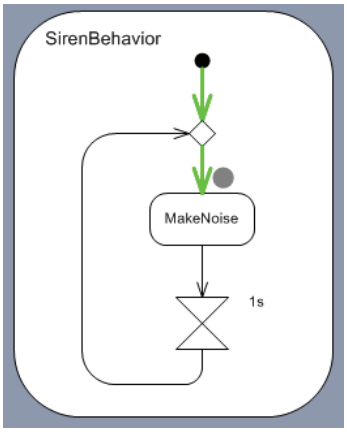
When you click "`Step in`" the first time a new activity is initiated, a dialog asks you, if you want to observe executions of this type of activity. You can also pre-select other types to trace in the future:



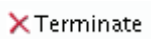
If you click "Yes", a new Visio document is created showing the new execution of the selected activity instance:



When you step through the diagram, edges get highlighted as tokens flow through it. For example, after clicking "`Step in`" again, the CallOperationAction "`MakeNoise`" is now in "CreateExecution"-state (see section 3.3.1 for a description of transition states of an activity diagram). After a step has been executed, the whole simulation is suspended until you click Step again.



Note that a call to "`MakeNoise`" has no effect due to the setting we previously made. When the timer action is activated, no further step is possible until the duration specified (one second in our example) has elapsed. This is indicated by the Step buttons being temporarily disabled (greyed out).

You can terminate all running executions at any time by using the 

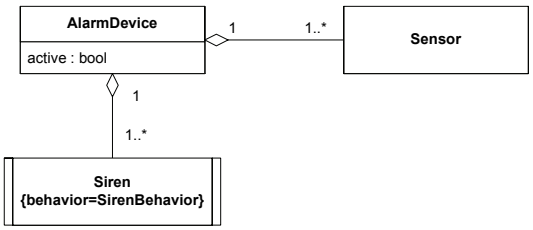
-button on the toolbar. All Visio documents showing executions are closed.

4.3 Using Class Diagrams

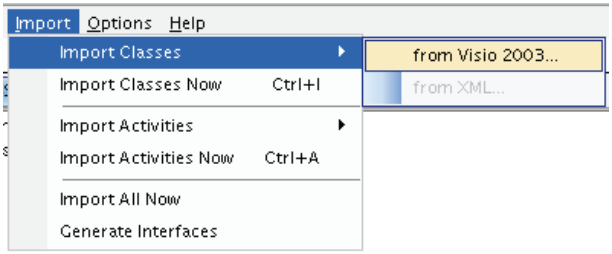
4.3.1 Generating Code for Classes

For activity diagrams to be useful – i.e. to enable CallOperationActions to invoke your custom code – you need to associate a class to your activitiy, called "context class" (see section 3.4). This section explains how to implement and use those classes in ActiveCharts.

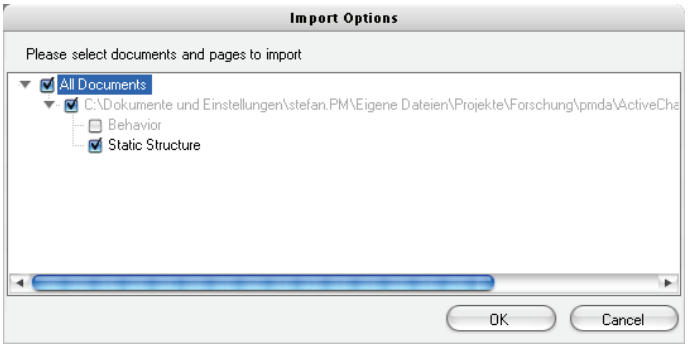
We will use the example of our simple alarm device with sensors and sirens, for which we already defined a behavior in the previous section:



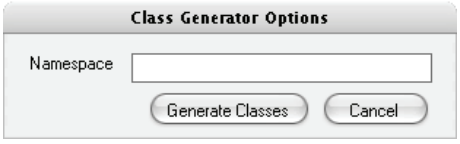
The "Siren"-class will be connected to "SirenBehavior", which is depicted by our special UML tag. We can now import the classes and generate code from it using ACTIVECHARTSIDE:



As it was the case with activities, you have to select the Visio documents and tabs where your classes are drawn:



After clicking on "Ok", you can specify a namespace name to use for the generated classes. For now, leave it blank:



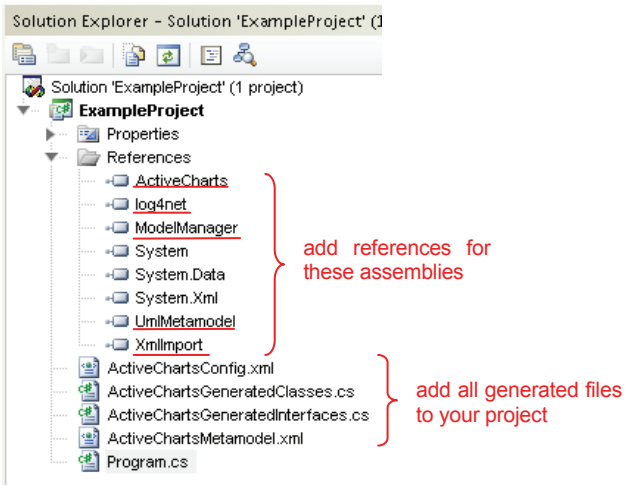
After clicking on "Generate Classes", a file called "`ActiveChartsGeneratedClasses.cs`" is created in the project folder, containing the partial class definitions for your diagram (again, you can use the "`Import->Import Classes Now`"-shortcut from the menu or use *Ctrl+I*). The generated code looks as follows:

```
[Serializable]
public partial class Siren : IIdentifiable
{
    Relationships

    IIdentifiable interface implementation

    Constructors

    implementation of UmlAssociation: Siren(Siren)[1..*]<-><> AlarmDevice(AlarmDevice)[1..1]

    // associated activity
    [NonSerialized]
    private ActivityExecution currentActivityExecution = null;

    public ActivityExecution CurrentActivityExecution
    {
        get
```

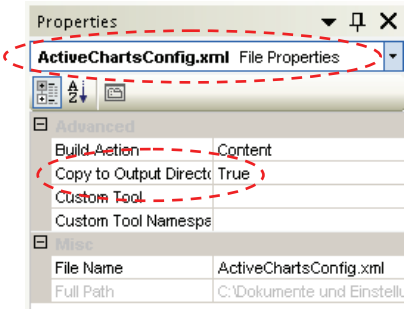
Notice the partial class definition for "Siren", where all attributes and relationships drawn are implemented by automatically generated code.

4.3.2 Setting up a Visual Studio .NET Project for your Files

Before being able to add methods to this class using additional partial class definitions, you first have to create a Visual Studio .NET project in your ACTIVECHARTS project folder (note that the Visual Studio project actually has to be created in the *parent folder* to be located in the current project folder!) and add all generated files by using the "`Project->Add->Existing Item...`" menu and references to all copied assemblies by using the "`Add Reference...`" popup menu to it (see section 4.1 for explanations of these files):



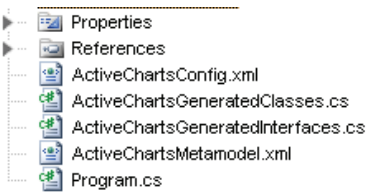
The generated XML-files must also be copied to the executable directory upon project compilation. To enable this, you have to set the "Copy to Ouput Directory" property to "True" for the files "ActiveChartsConfig.xml" and "ActiveChartsMetamodel.xml" (if this file does not exist, you haven't yet successfully imported an activity- or classdiagram):



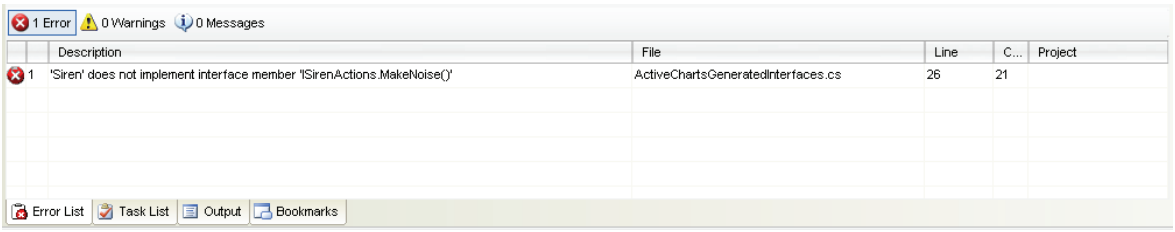
These tasks must only be made once when the Visual Studio .NET project is created but are absolutely necessary for proper execution of the project.

4.3.3 Add Action Methods to your Classes

ACTIVECHARTSIDE automatically generates interfaces from action invocations in activity diagrams. These methods are added to interfaces which the corresponding context class must implement. To be able to generate those interfaces, you must first import a class diagram to get the mappings between classes and activities and finally the corresponding activity diagrams. Interfaces are generated at the end of the activity diagram import and will be written to a file called "ActiveChartsGeneratedInterfaces.cs" in your project folder (you can also re-generate the interfaces by choosing "`Import->Generate Interfaces`".) In your Visual Studio .NET project, add this file to your solution:



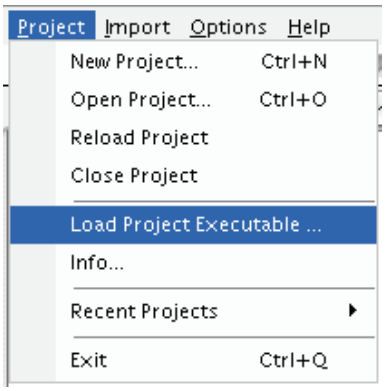
If you try to compile your project without implementing the action methods defined in the interfaces, you get a compilation error:



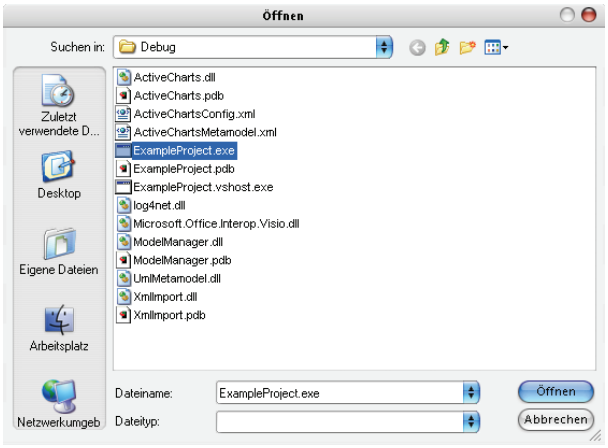
You can now simply add these methods to the generated partial classes. Consider, for example, an additional C# file for your project called "Program.cs". Add a new partial class definition to it for the (already in the file "ActiveChartsGeneratedClasses.cs" existing) "Siren"-class and add a method "`MakeNoise()`" (implementing the corresponding interface method) to it:

```
Program.cs
Siren                                   MakeNoise()
using ...

public partial class Siren
{
    public void MakeNoise()
    {
        ActiveCharts.ActiveChartsConsole.WriteLine("Hello ActiveCharts!");
    }
}

namespace ExampleProject
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

This method will then be executed automatically by our runtime when execution of your activity diagram reaches the "MakeNoise"-CallOperationAction. For now, leave the "`Main()`"-method blank and compile your project. To be able to find your method, you must point ACTIVECHARTSIDE to the executable "`ExampleProject.exe`" (created by Visual Studio .NET) using the "`Project->Load Project Executable...`" menu entry in ACTIVECHARTSIDE:

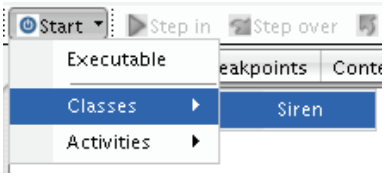


Select "ExampleProject.exe", normally located in the "`...\bin\Debug`" subfolder in your project folder:

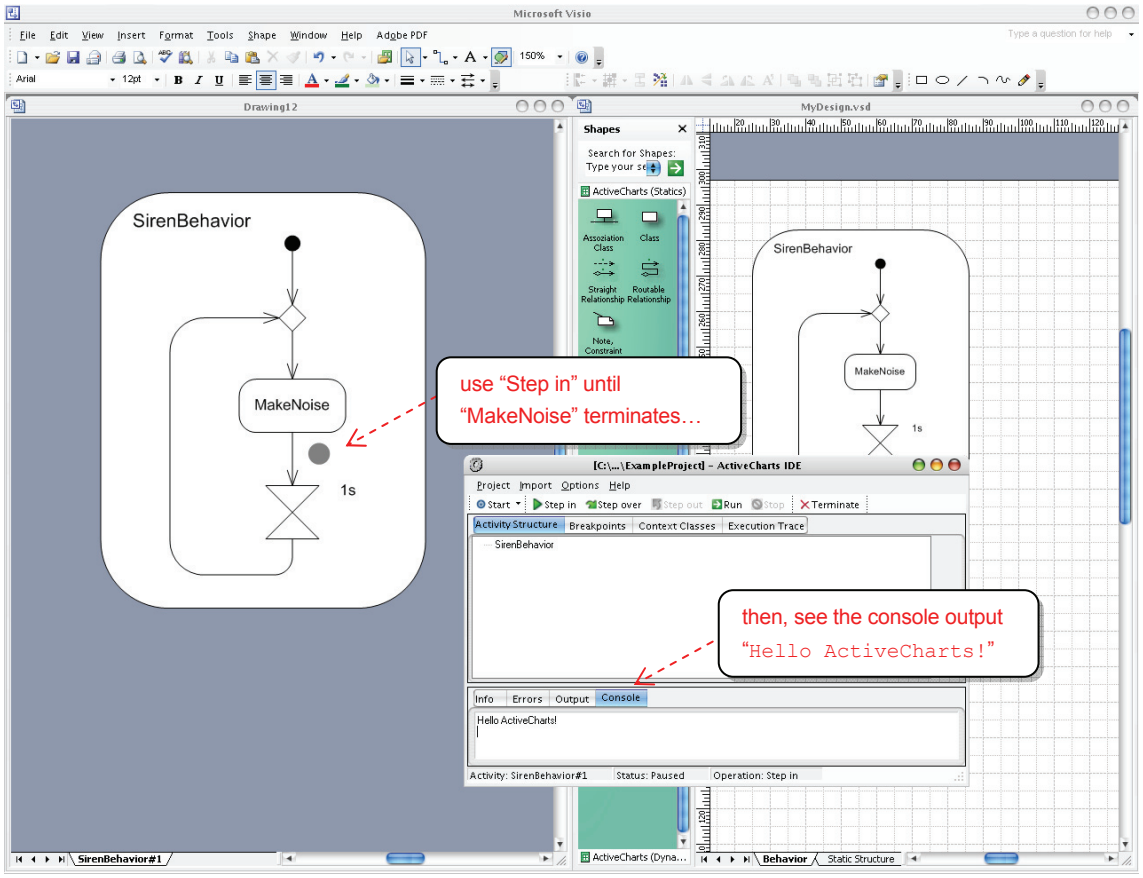


Once you loaded a project assembly, this information is stored in your project file so that this assembly is loaded everytime you open the project. You can associate another assembly with your project simply by using the "`Load Project Executable`" item again.

Now that the class code is available to our IDE, you can instantiate objects from it using the "Start"-menu. Note, that the submenu "Classes" is now available, containing entries for all classes that have an associated behavior (only the "Siren"-class at the moment):



After selecting "Siren" from "`Start->Classes`" and using "Step in" from the toolbar you will see that your code is called when the "MakeNoise" CallOperationAction has been reached:



Congratulations! You managed to execute your first ACTIVECHARTS project!

5 Controlling Execution

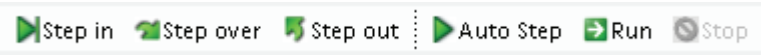
There are two modes of executing an ACTIVECHARTS-project:

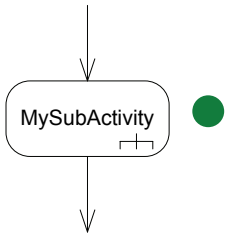
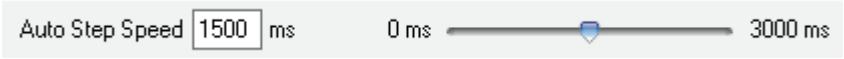
1. **Simulation**
   In this mode you use the ACTIVECHARTSIDE to control and visualize the execution of your project, as exercised in the section before. You can make use of functions like "Step-In", "Step-Out", observe executions of interest, create breakpoints, etc. to debug your application flow. This is done during analysis, design and testing of your code.

2. **Direct Execution (standalone)**
   Your project can also be executed completely independent of the ACTIVECHARTSIDE. In this case, no visualization takes place but the model is executed "behind the scenes" to control your application. To use direct execution, just compile your Visual Studio .NET project and run it in a normal way. The ACTIVECHARTS-runtime notices that the IDE is not connected and disables all visualization and step-control functionality automatically. Direct execution is used for production release of your application.

The following section shows all important elements and functions of the IDE in detail.

5.1 Step Functions



Function	Description
Step in	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 numbered using a "`<name of activity>#<execution number>`" scheme. When an execution terminates (i.e. by reaching an activity final node), the tab resp. document is automatically closed.
Step over	 When pressing "Step over" just after a CallBehaviorAction calling a subactivity (see section 6.2) has been enabled, all steps in this sub activity are not visualized. Note that steps occuring in other executions are nevertheless shown, which means that the effect of "Step over" is maybe delayed.
Step out	When pressing "Step out" all remaining steps in the current execution are no more shown.
Auto Step	Performs →"Step in" automatically in a loop. Can be interrupted by using →"Stop". The step-delay can be adjusted using the slider and textbox below the toolbar: 
Run	Runs all executions without visualizing any token flow. Is interrupted by →"Stop" or when a breakpoint is reached.
Stop	Interrupts →"Auto Step" or →"Run". The current step will always finish executing.

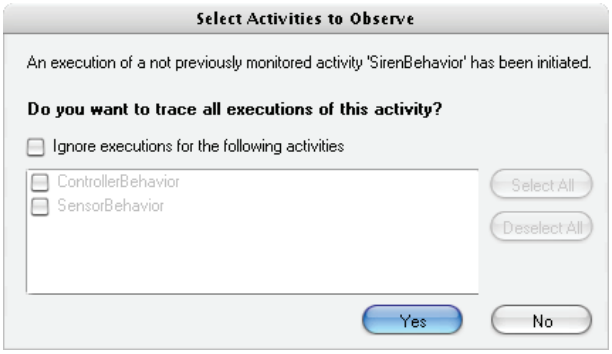
Note that only observed executions are taken into account when using the step functions (except "Run", which covers all executions). See section 5.2 for details.

For more comfort, you can hide the ACTIVECHARTSIDE main window and show a minimal step-control panel by pressing *Ctrl+M* or selecting "Minimized Step Controls" from the "Options" menu:

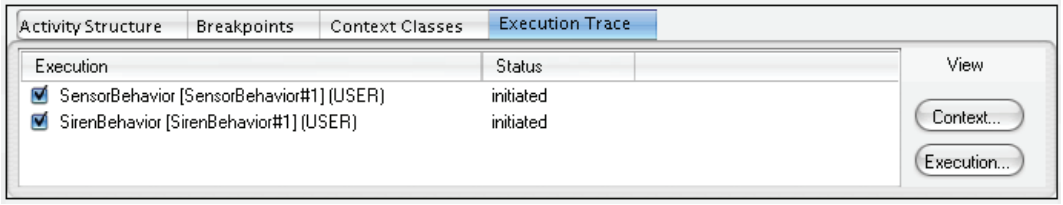
To restore the main window, just close the panel or press *Ctrl+M* again.

5.2 Observing Executions of Interest

Depending on the total number of activities in your project you would not normally want to trace executions of all types of activities at the same time. As already mentioned in section 4.2.2, you can select activities to monitor in a dialog that comes up every time a new activity is executed:

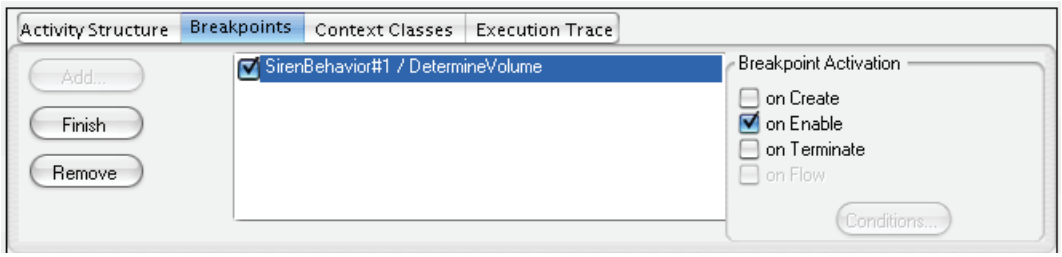


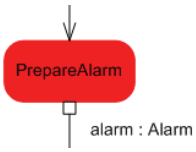
You can furthermore hide executions currently active using the "Execution Trace" window:



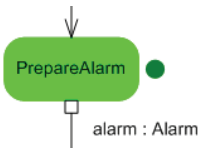
5.3 Using Breakpoints

To ease debugging your application flow, you can set breakpoints on actions or flows. Click on the "Breakpoints"-tab and select "Add…". You can now click on elements in your currently acitve executions to set breakpoints on them. When you have finished setting breakpoints, click "Finish". When a breakpoint is highlighted in the list, you can also specify or change the activation condition.





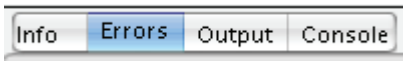
After setting a new breakpoint, the corresponding action or flow edge is marked red.



A breakpoint that has been reached is marked green.

5.4 In Case of an Application Error

When a serious application error occurs that prevent ACTIVECHARTSIDE from continuing execution, an error message will be shown on the "Errors"-tab. The IDE has normally to be restarted to operate properly.



In case of an error, send

- the error message (see the "Errors"-tab)
- the file "`ActiveChartsExecutionLog.txt`" (located in the ACTIVECHARTSIDE executable directory)
- the models that produce the error (if possible)
- a short description when and under what condition the error occurred
- and the ACTIVECHARTSIDE version you used (can be found under "`Help->About ActiveChartsIDE...`")

to:

mdd@informatik.uni-ulm.de

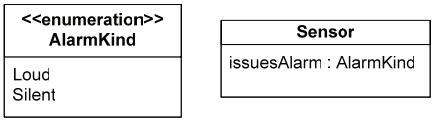
6 Advanced Modeling and Code Generation Concepts

The following sections introduce advanced concepts of static and dynamic modeling with ActiveCharts.

6.1 Advanced Concepts for the Static Structure

6.1.1 Enumerations

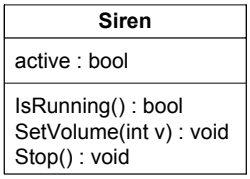
You can make use of enumerations in your class diagram. To define a new enumeration type, simply use a class shape and add the stereotype "<<enumeration>>" in ist heading. Define the enumeration values in the attribute compartment.



The new type can then be used in you class definitions.

6.1.2 Methods

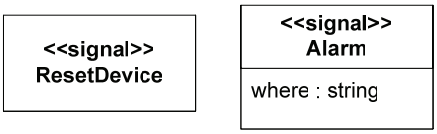
Method declarations are normally defined in your own program code. If you want a declaration to appear in the class model, create a new compartment to add method declarations:



Normal C# syntax is used for specifying parameter and return types. Note that return types must be separated with a colon (similar to attribute declarations), which is the UML style to define methods. After importing the class diagram, these methods are added to the generated interfaces file (see section 4.3.3). You can now implement them in your custom code files.

6.1.3 Signals

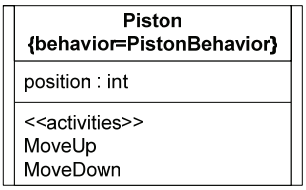
Signals are an important concept for synchronizing control flow in activity diagrams. Before any signal can be used (see section 6.2.4), they have to be defined in a class diagram:



The stereotype "<<signal>>" is used to declare a signal of a specific type. Attributes can also be added to signals, enabling more powerful synchronization mechanisms in your diagrams.

6.1.4 Specifying additional Activities for Classes

A class can only have a single associated activity (defined by the "behavior"-tag), which describes the behavior of instances of tha class. Additional activities can be defined in a compartment called "<<activities>>":



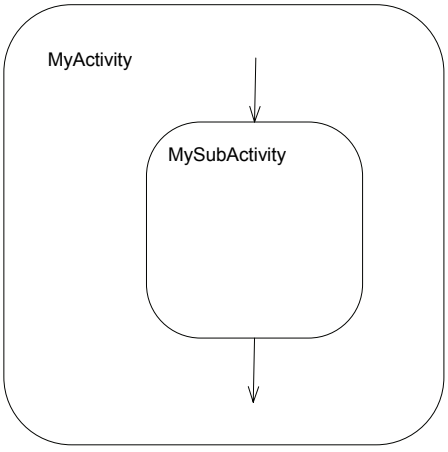
Methods are generated to initiated these activities in a convenient way (see file "ActiveCharts-GeneratedClasses.cs").

When calling one of the generated methods (which encapsulate an activity call) from an own action method, it is obligative to define the `[ActiveCharts.HandlesAbort]` attribute on that action method (see section 7.1 for a description of this attribute). The reason is that activity aborts must be handled in a correct way.

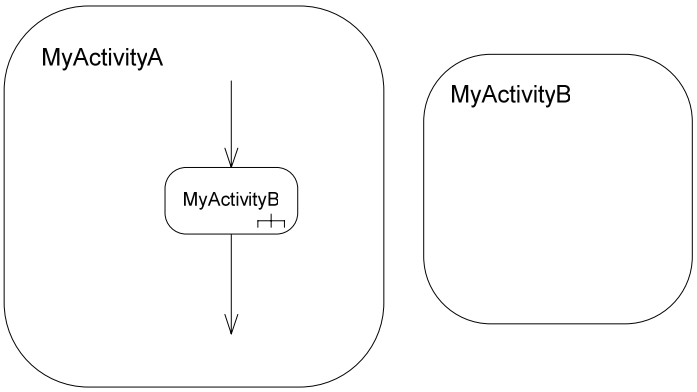
6.2 Advanced Control Flow Modeling

6.2.1 Structuring Activities into Sub-Activities

For improving readability and enabling model reuse, activities can be decomposed into sub-activities.



In fact, modeling of „inner" activities is not supported by the UML 2 activities metamodel, but our drawing is translated to a CallBehaviorAction which calls the activity accordingly. If an activity must be called from more than one place, an explicit modeling of a CallBehaviorAction must be used:



Notice the small fork symbol at the CallBehaviorAction "MyActivityB" which appears when you select "Call Activity" from the context menu.

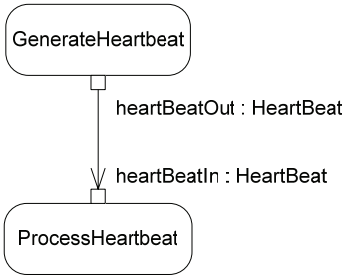
When calling other activities, **the context object remains the same**, i.e. the context object for the execution of "MyActivityA" is passed to the activity execution of "MyActivityB". This also means that you have to take care that all action methods from all called activities are properly implemented by this single context class. Since activities can be called from various activities, a kind of *polymorphism* is enabled.

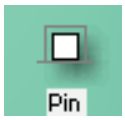
6.2.2 Object Flow

Instead of or in addition to control flow, UML 2 also offer the possibility to use data flow (call"object flow" in UML) in activity diagrams.

6.2.2.1 Using Pins

Object flows are defined by connecting "Pins" to UML elements, like Actions, for example:





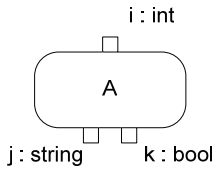
The Pin-shape in the Visio 2003 "ActiveCharts (Dynamics)"-stencil is used for modeling pins. To connect pins, normal transition edges are used. The current contents of a pin (i.e. its data) can be viewed by placing the cursor over it in the simulation.

Pins for outgoing object flows (i.e. sources) are called OutputPins, those for incoming flows (i.e. targets) are InputPins. All Pins *must* be named by selecting them and pressing *F2*. Each name *must* be of the form "`parameterName : parameterType`", the parameter type being a legal C# type (system or a self defined one).

After importing, the following interface methods have been created by ACTIVECHARTSIDE:

```
HeartBeat GenerateHeartbeat();
void ProcessHeartbeat(HeartBeat heartBeatIn);
```

Notice the return and parameter types of the action methods which match the Pin-definitions in the diagram. Since there is only a single OutputPin, the return type of the "GenerateHeartbeat"-method is used. If there are multiple OutputPins, additional C# "out"-parameters are utilized:



This action yields a method

```
string A(int i, out bool k);
```

respective

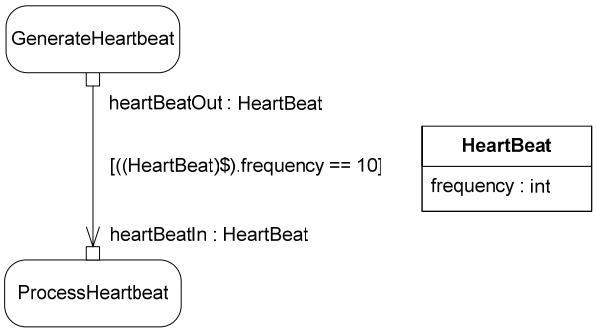
```
bool A(int i, out string j);
```

depending on the order the pins have been drawn by the developer.

When implementing Action methods with pins, you *must* use the same parameter names as specified by the modeled action respective the generated interface. Otherwise, ActiveCharts is not able to map object flow to parameters in a correct way!

6.2.2.2 Accessing Object Flow in Guards

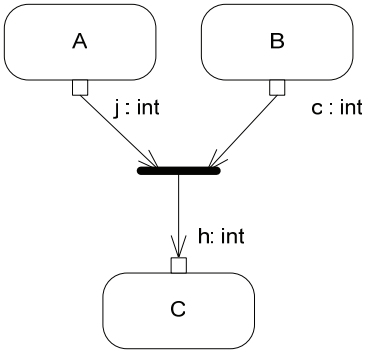
Data which is transported by object flow edges can be accessed in guards by using the "$" symbol:



Additionally, a *type cast* of the variable is needed in any case since ActiveCharts cannot compute the correct type of data flows at the current time.

6.2.2.3 ObjectFlow passing JoinNodes

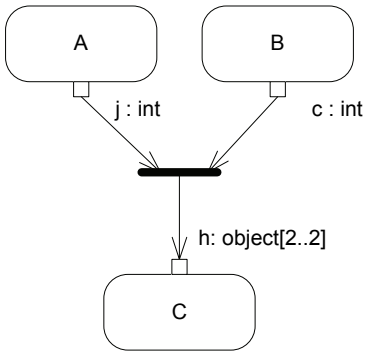
When multiple object flows are joined using a JoinNode, tokens are not merged (as in case when joining control flows only) but passed each to the following object node.



In this example, both outputs of action "A" and "B" are transferred to "C", which is executed twice.

6.2.2.4 Upper/Lower specifications on Pins

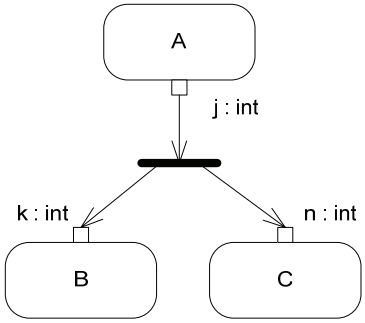
To prevent action "C" from the previous example to start two times, a specification of upper/lower bounds for the InputPin "h" can be used:



Action "C" does not start until the number of tokens offered has not reached the „lower" value. It then takes at most "upper" tokens to call the method. "upper" must always be greater than or equal to "lower". Note that the pin-type currently must be "object" if upper/lower bounds are used. Action "C" can then cast the array contained in the object to the required type.

6.2.2.5 ObjectFlow passing ForkNodes

When an object flow reaches a ForkNode, the data is transferred across each outgoing edge:

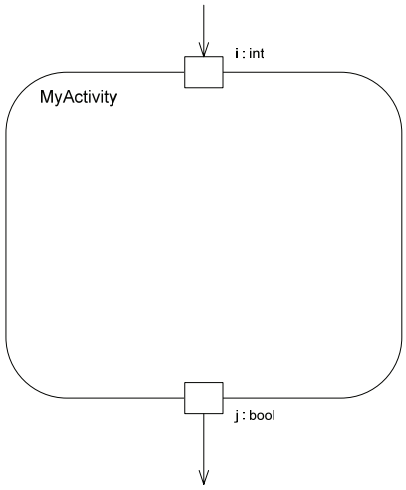


In case of reference types being handled, ActiveCharts calls the "`Clone()`" method to clone the object if it implements the `ICloneable` interface.

If a class used in an object flow does not implement the `ICloneable` interface, the same reference is passed across all edges, meaning that modifications of this object in an action possibly affects actions executing in parallel.

6.2.2.6 Using Activity Parameters

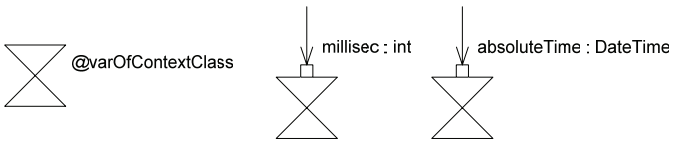
Activities can also be supplied with parameters. The "Activity Parameter Node" shape is used for this purpose:



Note that when calling activities from other activities by using a CallBehaviorAction (see section 6.2.1), the number, types and names of activity parameters and InputPins of the CallBehaviorAction calling the activity must match.

6.2.3 Advanced Timers

In addition to specifying exact time values for Timers (see section 3.3.2), you can also reference attributes declared in your context class or use object flow data:

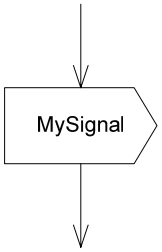


The variable is evaluated every time a create execution step is executed on the timer. When using object flow data, relative time specifications can be used by adding an InputPin of type "int". Absolute time specifications are used by specifying a "DateTime" type.

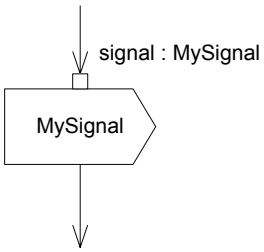
6.2.4 Signals

6.2.4.1 Sending Signals

Signals can be used to synchronize control flow. Before any signal can be sent, it must be defined in a class diagram (see section 6.1.3). It can then be used in a SendSignalAction:



Note that control flow immediately proceeds to the following action (meaning, the send is asynchronous), even if no action is listening for the signal. If there is an InputPin "`signal : Signal`" (or one of its subtypes) is defined, it is assumed to be a signal instance, which is then used:



Sometimes it is useful to construct signal instances on your own, for example if your signal class defines attributes which have to be set preliminarily. In case there is no such InputPin defined, an empty instance using the default constructor of the generated class for the signal is created.

Signals using the SendSignalAction symbol are sent to the *current* activity execution if there is no InputPin called "`target : ActivityExecution`" and are also *buffered* by default.

If you want to send signals to multiple other executions (even to executions of other classes) you must the the **BroadcastSignal**-shape:

When using this action, the signal is sent to every execution currently active. For possibilities to narrow the set of targets, see section 8.

To send signals from your custom code (e.g. to initiate some action from GUI event handlers) you can use the following code:

```
<object>.SendSignal(<signal instance>);
```

Note that this method is only available for objects that have its own behavior.

To broadcast a signal to all executions of all activities you use

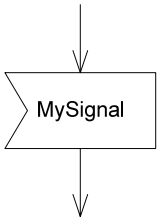
```
Activity.BroadcastSignalToAll(<signal instance>);
```

To broadcast a signal to all executions of a specifiy activity you use

```
<activity>.BroadcastSignal(<signal instance>);
```

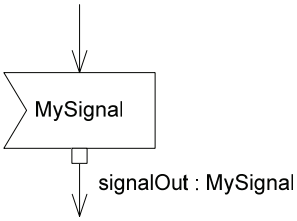
6.2.4.2 Receiving Signals

To wait for signals, an AcceptEventAction is used:



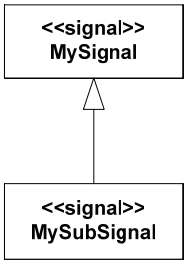
If an AcceptEventAction is enabled, it blocks until a signal is found in the buffer and then proceeds to the next action. An AcceptEventAction with **no incoming control flow** it is *immediately* enabled (more precisely, a "create execution step" is performed) if the containing activity starts. If it terminates due to receipt of a signal it is re-enabled automatically, meaning it is always able to accept signals. An exception to these rules are AcceptEventActions contained in InterruptibleActivityRegions, see section 6.2.5.

If an AcceptEventAction has an **OutputPin** defined, the received signal is available to following actions:



If more than one AcceptEventAction is able to handle a signal, only one – randomly determined – active action will receive it.

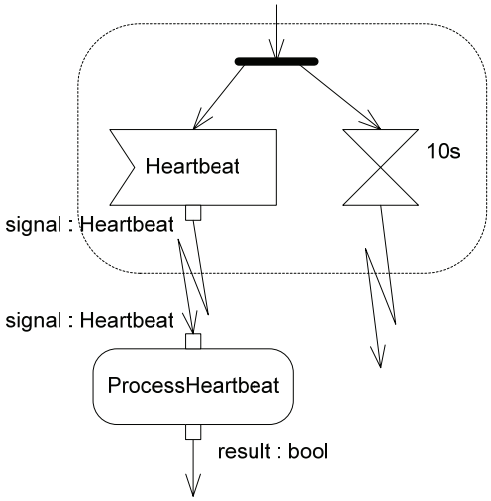
Signal types can also form a **hierarchy**:



Signals of subtypes are also handled by AcceptEventActions for supertypes.

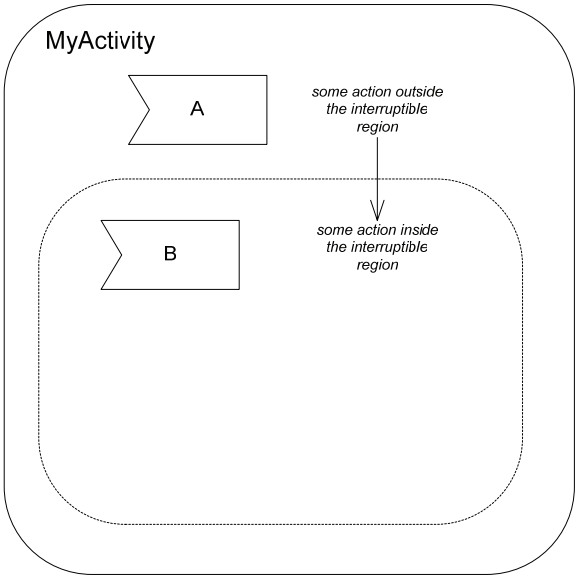
6.2.5 Interruptible Activity Regions

Interruptible Activity Regions mark a set of nodes which are aborted when a special transition edge – called interrupting edge – is passed. When a token flows along an edge of this kind, all tokens currently contained in the region are removed and all actions are terminated immediately.



No more than one interrupting edges can be passed in one step; if there is more than one edge, any one will be taken nondeterministically. Note that other "normal" transitions included in the current step, which lead to the outside of the interruptible region, are executed in a normal way, i.e. they are not interrupted by the concurrent interrupting flow.

As already described in section 0, AcceptEventActions with no incoming flow are enabled immediately when the containing activity starts. An exception to this rule are AcceptEventActions contained in an InterruptibleActivityRegion, which **are not enabled until some flow enters the region**:



Action A is enabled as soon as the activity starts, whereas B is only activated when a flow enters the region. If Action A terminates, it is re-enabled immediately, B is only **re-enabled only as long as there is at least one active node left in the region**.

7 Advanced Execution

7.1 Controlling Action Method Behavior

There are several issues that need perhaps to be handled when implementing action methods with your own C# code:

- Actions are **aborted** when leaving interruptible activity regions or when an activity is terminated.

- When executing in "Simulation Mode", i.e. when using the ACTIVECHARTSIDE to debug your application, your code is **suspended** after a step has been executed and shown in the diagram. The code is not resumed until one of the Step buttons is clicked again.

The thread your own code runs on is controlled by the ActiveCharts runtime. If it is critical to abort or suspend your code due to synchronisation problems for example, it is possible to get notified when your code should be suspended or aborted. To enable this feature, you use the `[ActiveCharts.HandlesAbort]` attribute before your action method if you just want to handle abort events, or the `[ActiveCharts.HandlesSuspendAndAbort]` attribute if you want to handle suspend and abort events. You can then react to the events in your code in the following way:

```
using ActiveCharts;
…
[HandlesAbort]
public void MoveDown()
{
    using (MethodExecutionState methodExecutionState =
           MethodExecutionStateManager.RetrieveMethodExecutionState())
    {
      // do your (long running) stuff here
      // …
      if (methodExecutionState.HasRequest &&
          (methodExecutionState.StateRequest == MethodExecutionState.State.Abort))
      {
           // handle your own abort here
           // …
           methodExecutionState.NotifyStateReached();
      }
    }
}
```

Note that it is *required* to call the `NotifyStateReached()` method as soon as you handled the suspend or abort request. If you omit the call, ActiveCharts will block infinitely. If you call this method after a suspend request, your application gets blocked and is resumed automatically by ActiveCharts.

Normally it is only useful to handle suspend and abort events in cases of long running actions. If your action only consists of a few lines of uncritical code with a small total execution time, custom event handling is not needed.

If you are calling a generated activity method (see section 6.1.2), the usage of `[ActiveCharts.HandlesAbort]` is obligative.

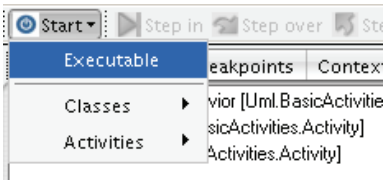
7.2 Writing and using your own Application Executable

In section 4 ("Getting Started") we invoked an activity directly by using the "`Activities`" entry from the "`Start`"-menu. This procedure is normally used when modeling activities for the first time to quickly check the flow graphically. In ordinary application development, you have a main program that sets up your objects (having some with and others without activities and action methods), and then showing some kind of user interface to interact with the application:

```
public partial class Form1 : Form
{
    private AlarmDevice alarmDevice;

    public Form1()
    {
        InitializeComponent();

        alarmDevice = new AlarmDevice();
        //alarmDevice.StartBehavior(); (if "autostart" was set to "false")

        Siren siren = new Siren();
        alarmDevice.siren.Add(siren);
        //siren.StartBehavior(); (if "autostart" was set to "false")

        Sensor sensor = new Sensor();
        alarmDevice.sensor.Add(sensor);
    }

    private void buttonActivateSensors_Click(object sender, EventArgs e)
    {
        foreach (Sensor sensor in alarmDevice.sensor)
        {
            sensor.activated = true;
        }
    }

    private void buttonExit_Click(object sender, EventArgs e)
    {
        ActiveCharts.ActiveChartsEngine.Exit();
    }
}
```

It is useful to initialize your objects in the constructor of your windows form, for example. Note that the behavior of an object starts immediately as it is instantiated. An exception to this being if the "`autostart`" tag is set to "`false`" for this class; for an example see section 3.4. In that case you have to start the bahavior manually using the `<instance>.StartBehavior()` method.

After compiling your application, you can use your executable and directly run it from ACTIVECHARTSIDE. To do so, you initiate your "`.exe`"-file by using "`Start->Executable`":



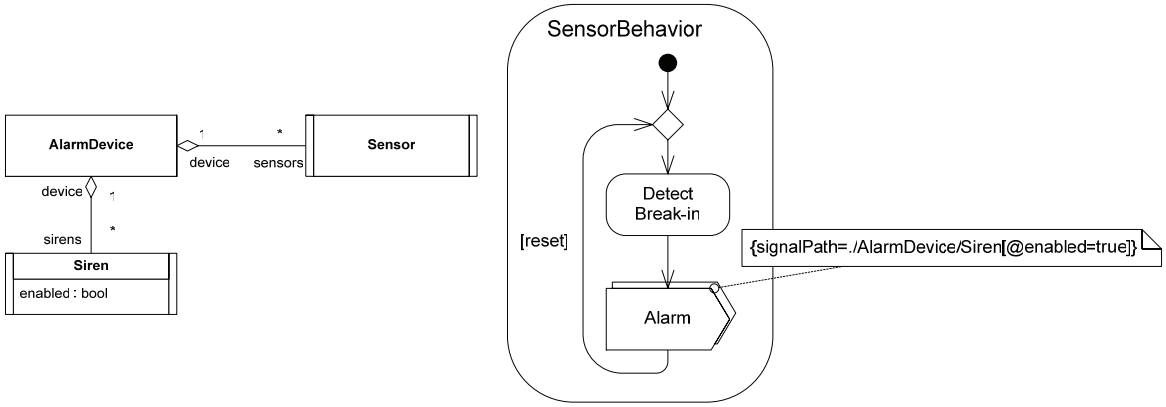
When exiting your application you should take care of proper termination of the ActiveCharts runtime by calling the `ActiveCharts.ActiveChartsEngine.Exit()` method.

8 Signal Paths

The way targets of BroadcastSignalActions are computed is considered to be a "semantic variation point" due to the official UML 2 specification. ActiveCharts sends signals to all activity executions of all activities by default. To specify the set of targets exactly we introduce *SignalPath* expressions (see [Sarstedt05]) which are a subset of XPath [UML05] Object Management Group, UML 2.0 Superstructure Specification, http://www.omg.org/cgi-bin/doc?formal/05-07-04

[XPath] expressions. Using SignalPath expressions, ActiveCharts traverses the current object graph (from the class diagram) and computes target activity executions.

To attach the targeting information to a send-action in an UML diagram, we introduce a UML tag called "`signalPath`" that is written in a comment:



By specifying the SignalPath-expression

```
./AlarmDevice/Siren[@enabled=true]
```

we state, that, starting from the current context object (the sensor object belonging to the current SensorBehavior activity execution), the "* to 1" association to `AlarmDevice` has firstly to be traversed, resulting in a node-set that contains only the single `AlarmDevice` object. When evaluating the next part of the expression, the "1 to *" association to `Siren` is examined, yielding a node set of all sirens. After evaluating the filter predicate `[@enabled=true]` only those siren objects remain in the result set, that have its "`enabled`" attribute set to true. Thus, the targets of alarm signal encompass all active sirens that can be reached by our current object graph.

The syntax of valid SignalPath expressions is as follows:

```
<SignalPathExpression> ::= (“.” | <RelativePathExpression>) (“|” <SignalPathExpression>)*
<RelativePathExpression> ::= “./” <StepExpression> (“/” <StepExpression>)*
<StepExpression> ::= <ClassName> (<FilterPredicate>)*
<FilterPredicate> ::= “[“ <CSharpPredicate> “]”
```

where

- `<ClassName>` must be a valid class name from the class diagram
- `<CSharpPredicate>` must be a valid C# boolean expression where all occurrences of context class attributes are preceded by a "@" symbol.

Note that all expressions are relative to the current context class.

9 Adjusting Semantics

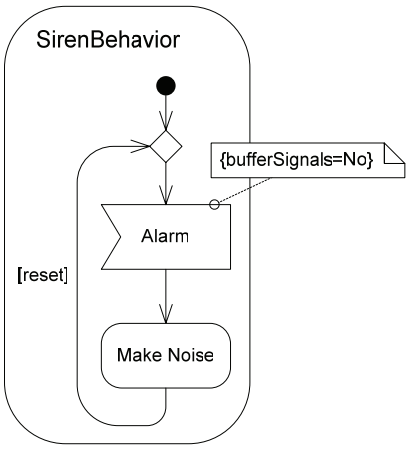
Due to various unclear points in the current UML 2 specification, we decided to leave choice of interpretation to the developer regarding some important issues. We make use of UML tags to configure the behavior of various UML model elements. Tags denote key/value pairs of the form "`{<key>=<value>(,<key>=<value>)*}`". In the following sections we describe configuration possibilities for signal buffering and handling of AcceptEventActions in nested InterruptibleActivityRegions.

9.1 Signal Buffering

Signals are buffered by activity executions by default. Since in some scenarios this can lead to misbehavior of the models, we offer the possibility to configure buffering using UML tags.

9.1.1 Disable Buffering

To disable buffering for certain types of signals, you can use the "`bufferSignals`" tag in the following way:

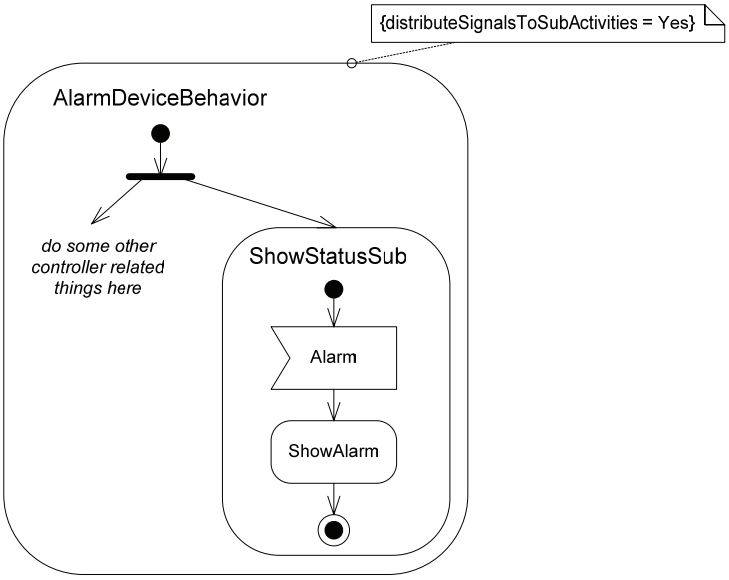


If signal buffering should be disabled for all types of signals, you can attach this tag to the surrounding activity. This setting can be overidden locally by setting "`bufferSignals`" to "`Yes`" at AcceptEventActions.

The default buffering setting is "Yes" for all types of signals.

9.1.2 Configuring Distribution

When using sub-activities (see section 6.2.1), it is not clear if signals sent to the "parent" activity are to be distributed to sub-activities. To configure this behavior a tag called "`distribute-SignalsToSubActivities`" can be used:



Signals sent to „AlarmDeviceBehavior“ are now distributed to the embedded activity. Note that this behavior would also be in effect when using a CallBehaviorAction to initiate another activity. This setting can be overridden at individual CallBehaviorActions.

The default distribution setting is "`No`".

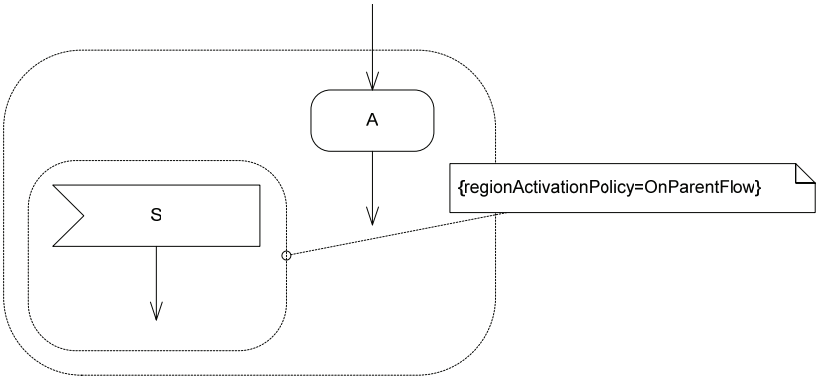
9.1.3 Signal Replacement

Signal instances of one type currently in the signal buffer can be replaced by a newly incoming signal when the tag "`replaceSignals`" is set to "`Yes`" for an AcceptEventAction. This tag can also be applied globally at the activity level. The replacing mechanism defaults to "`No`".

9.2 AcceptEvents in Interruptible Regions

9.2.1 Activation of AcceptEvents

Since we allow InterruptibleActivityRegions to be nested, it is not clear from the official UML specification how activation of AcceptEventActions without incoming edges is to be handled. We decided to introduce a tag called "`regionActivationPolicy`" stating when activation of an AcceptEventAction should actually happen. Legal values are "`OnParentFlow`", meaning that the Action is to be enabled when a flow enters the parent region and "`OnRegionFlow`", meaning that the Action is activated when a flow enters the current region (this is the default).



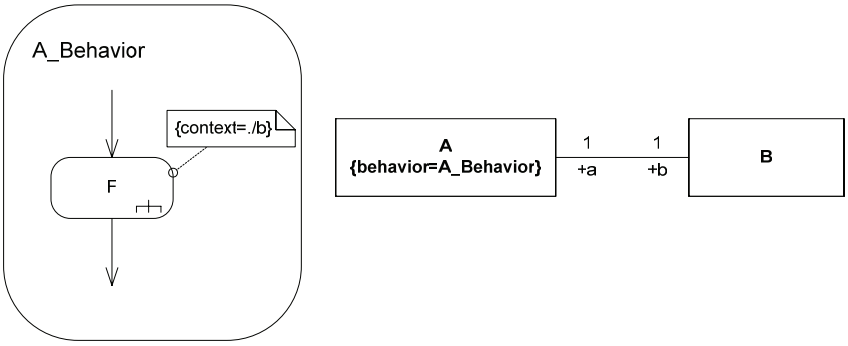
In the previous figure, AcceptExentAction "S" is activated as soon as the flow enters the parent region which activates CallOperationAction "A". Note that configuration of subregion activation only makes sense for nested regions, not for the top-level region.

9.2.2 Re-Activation of AcceptEvents

AcceptEvents without incoming edges are also subject to re-activation when they have terminated. For actions contained in InterruptibleActivityRegions this is only useful as long as there is at least one active node in the region. For nested regions, it is not clear if re-activation should happen when the parent region or the subregion has active nodes, and can therefore be configured using the "`regionReactivationPolicy`" tag. Legal values are "`OnRegionActive`" (the default) or "`OnParentActive`".

9.3 Changing context classes in CallActions

When using CallBehaviorActions or CallOperationActions, always the context class of the current ActivityExecution is used. If you want to use another object as a context object of a CallAction, you specify an XPath expression over the current object tree which must yield a single object:

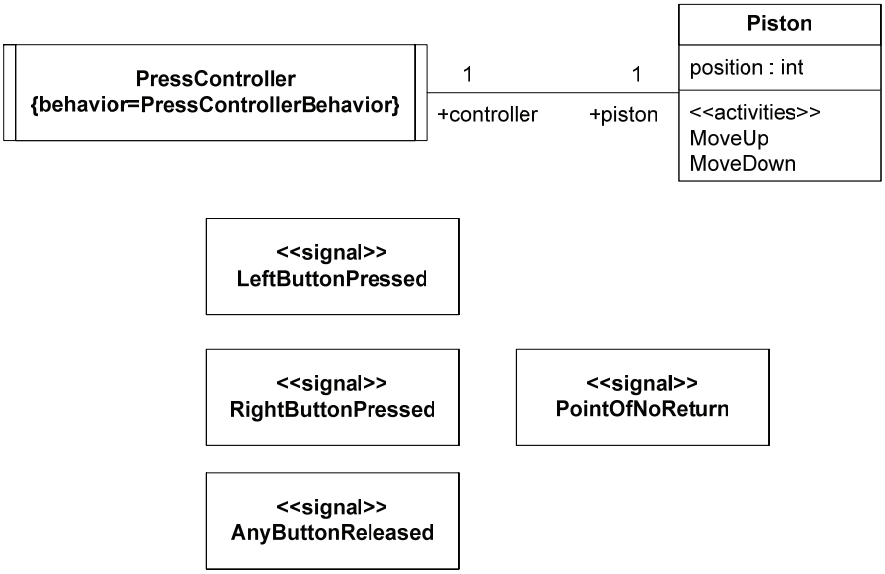


10 Examples

10.1 Molding Press

10.1.1 Description and Static Structure

The following figure shows the static structure of a simple molding press. It consists of a single controller which is associated with a piston. The current position of the piston (which can move up and down) is reflected by an attribute called "`position`" contained in the class:

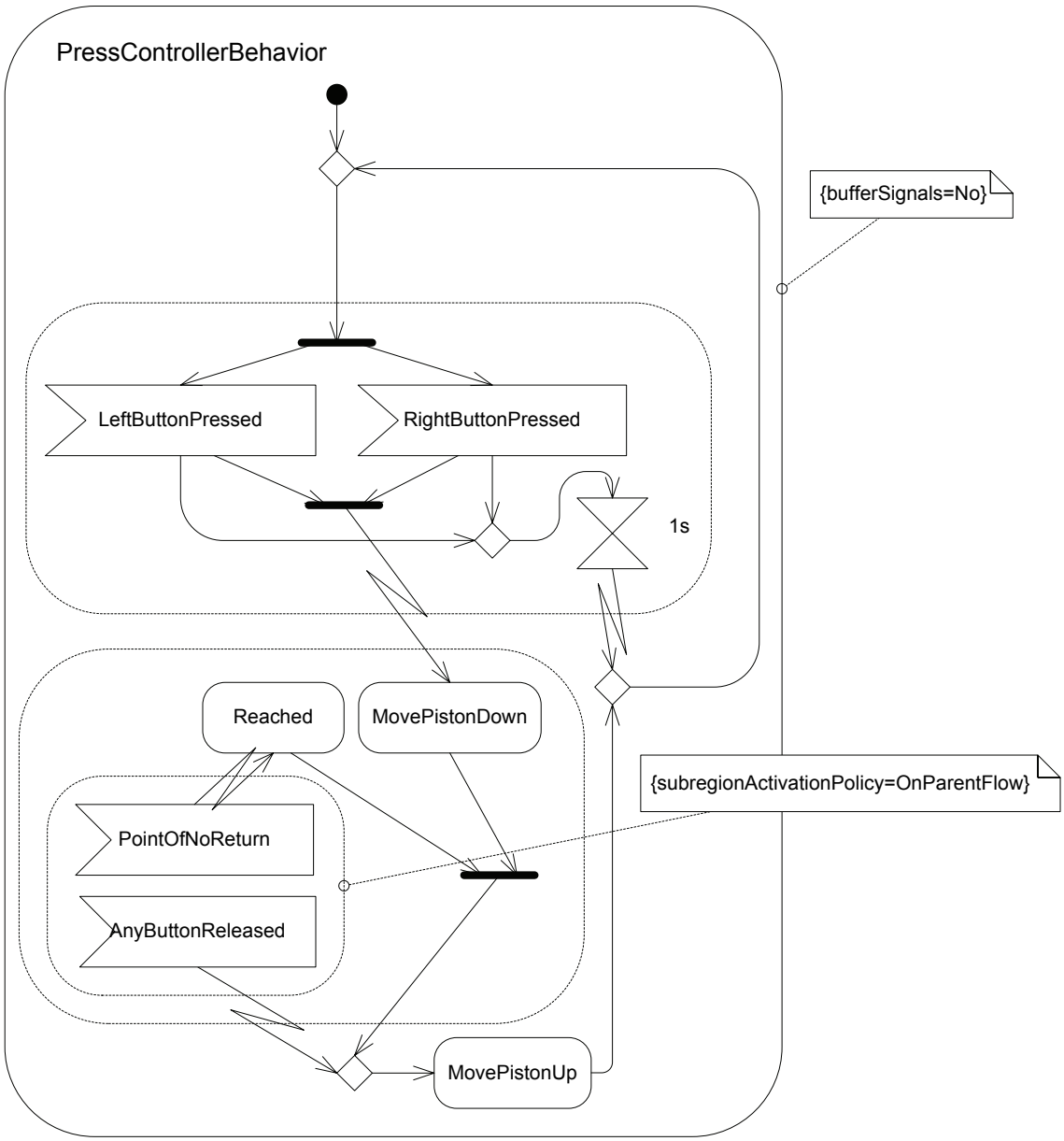


The requirements for a molding press are as follows:

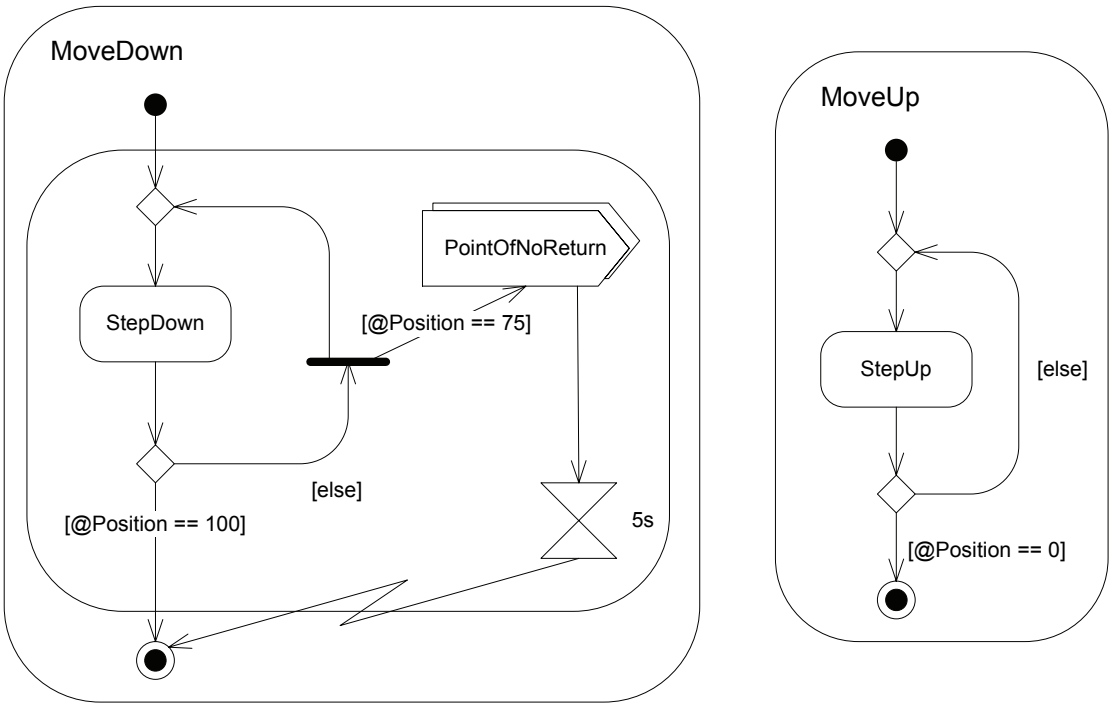
- The piston starts moving downwards if its two buttons are pressed within 1 second. If the time span is greater, both buttons have to be released before starting again.
- When the piston moves downwards and any button is released until the "Point of no return" is reached (which is located at ¾ of the total distance), the piston stops and moves upward again in its starting position.
- If the "Point of no Return" has been reached, the piston continues moving downward even if any button is released after this time.
- When the piston reached the bottom position it stops and moves upward to its starting position.

The following sections contain the complete behaviors and an excerpt showing the most important code (e.g. the GUI is missing).

10.1.2 Controller Behavior



10.1.3 MoveUp / MoveDown Behaviors for Piston



10.1.4 Application Code

```
public partial class PressController
{
	[HandlesAbort]
	public void MovePistonDown()
	{
		piston.MoveDown();
	}

	[HandlesAbort]
	public void MovePistonUp()
	{
		piston.MoveUp();
		Reset();
	}

	public void Reached()
	{
		piston.PointOfNoReturnReached = true;
	}

	public void Reset()
	{
		piston.PointOfNoReturnReached = false;
	}
}

public partial class Piston
{
	public enum PistonMovement { Up, Down, Idle };
	public PistonMovement PistonState;
	private bool pointOfNoReturnReached = false;

	internal bool PointOfNoReturnReached
	{
		get
		{
```

```
                return pointOfNoReturnReached;
            }
            set
            {
                pointOfNoReturnReached = value;
            }
        }

        internal void SetState(PistonMovement state)
        {
            PistonState = state;
        }

        public void Init()
        {
            position = 0;
        }

        public int Position
        {
            get
            {
                return position;
            }
            set
            {
                position = value;
            }
        }

        public void StepUp()
        {
            Position--;
        }

        public void StepDown()
        {
            Position++;
        }
}
```

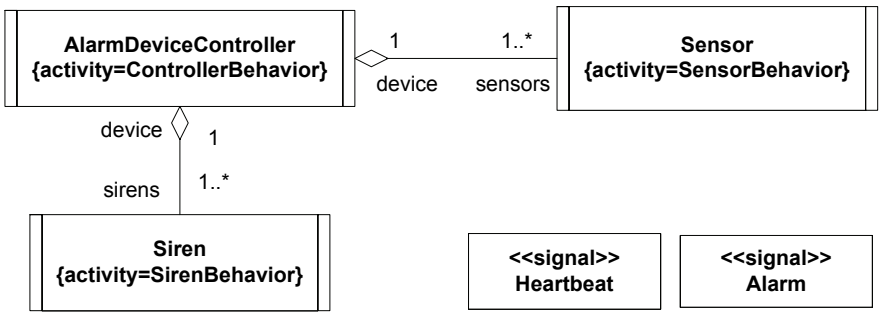
Code to configure and start the molding press:

```
controller = new PressController();
piston = new Piston();
controller.piston = piston;
```

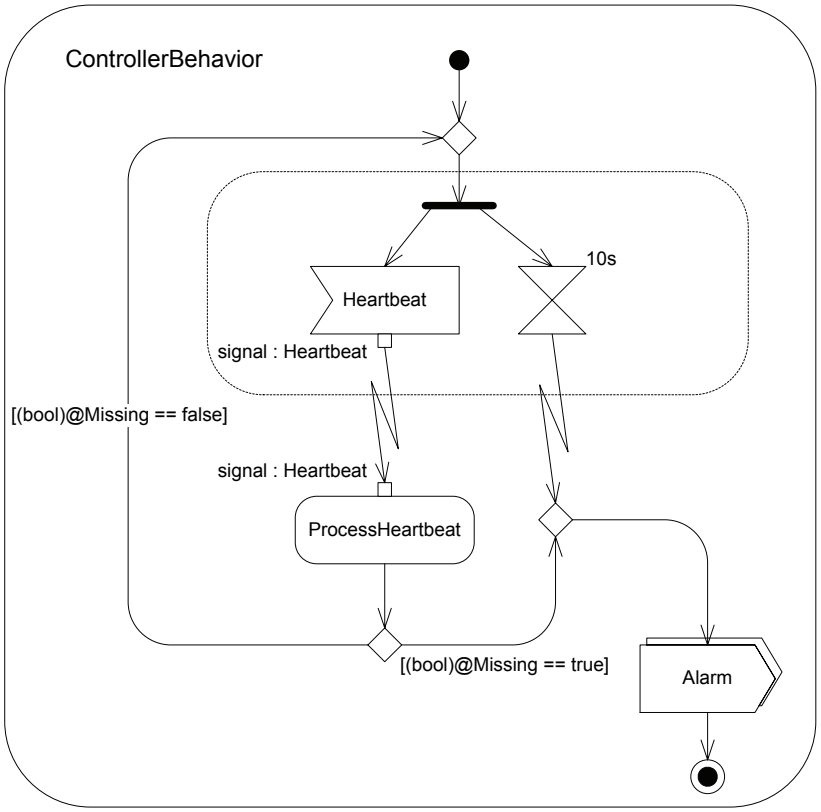
10.2 Alarm Device (Heartbeat)

10.2.1 Description and Static Structure

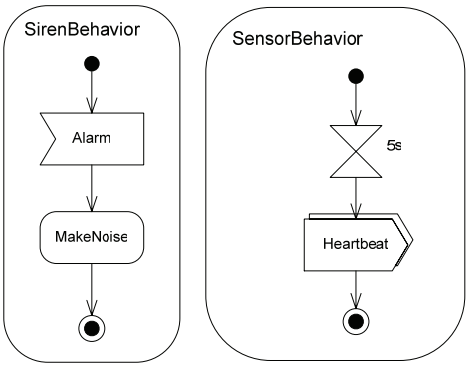
A simple alarm device consists of multiple sirens and sensors. Each sensor issues a “Hearbeat” signal indicating it is still alive. If such a signal misses for 10 seconds, an alarm is issued by turning on all sirens.



10.2.2 Controller Behavior



10.2.3 Siren and Sensor Behaviors



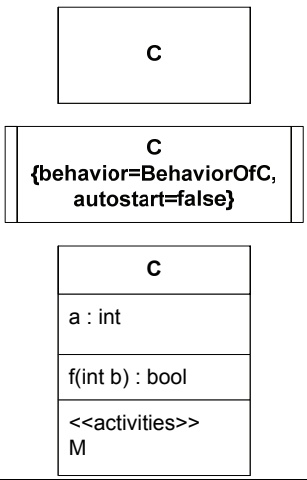
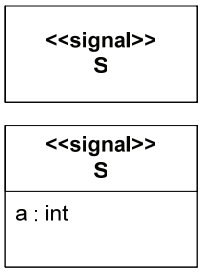
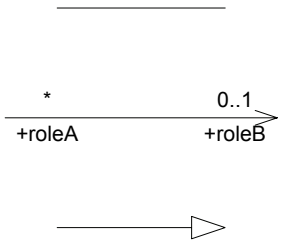
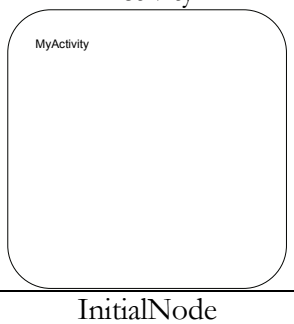
For testing purposes, our sensor only sends one Heartbeat signal and then dies.

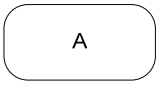
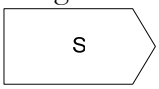
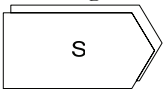
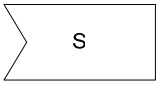
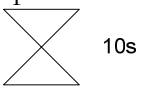
10.2.4 Application Code

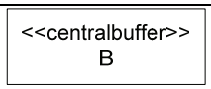
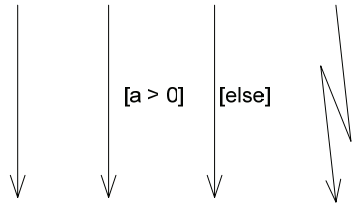
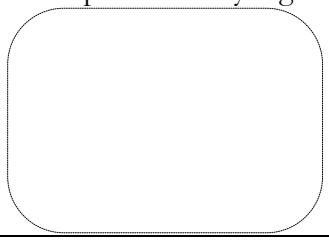
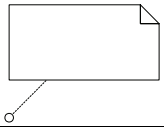
```
public partial class AlarmDeviceController
{
	private Dictionary<Sensor, DateTime> TimeStamps =
				new Dictionary<Sensor, DateTime>();
	public bool Missing = false;

	public void ProcessHeartbeat(Heartbeat signal)
	{
		Sensor triggeredSensor = (signal.source.context as Sensor);

		if (!TimeStamps.ContainsKey(triggeredSensor))
			TimeStamps.Add(triggeredSensor, DateTime.Now);

		foreach(Sensor sensor in this.sensors)
		{
			if (sensor == triggeredSensor)
				TimeStamps[sensor] = DateTime.Now;
			else
			{
				if (TimeStamps[sensor] < DateTime.Now.AddSeconds(-6))
				{
					Missing = true;
					return;
				}
			}
		}
	}
}

public partial class Siren
{
	public void MakeNoise()
	{
		ActiveCharts.ActiveChartsConsole.WriteLine("Alarm!");
	}
}

namespace AlarmDeviceExample
{
	public partial class MyForm : Form
	{
		public MyForm()
		{
			InitializeComponent();

			AlarmDeviceController controller =
					new AlarmDeviceController();
			Sensor sensor = new Sensor();
			controller.sensors.Add(sensor);
			Siren siren = new Siren();
			controller.sirens.Add(siren);
		}

		private void buttonExit_Click(object sender, EventArgs e)
		{
			ActiveCharts.ActiveChartsEngine.Exit();
		}
	}
}
```

11 Technical Reference

11.1 UML 2 Shapes for Visio 2003

UML 2 Element	Description
Class 	Models "concepts" of a domain. Classes can have an associated activity (specified with the behavior-tag) which is automatically started unless the optional "autostart" attribute is set to "false". Classes with behavior must be marked "active" (right click on shape to get this option). Classes can have attributes, methods and additional associated activities. Special methods are generated to initiate these activities from custom code.
Signal  *(use a class shape and type the „<<signal>>" stereotype specification yourself)*	Defines a signal type for use in SendSignal-, BroadcastSignal- and AcceptEventActions. Use a normal class shape for modelling signals.
Relationship 	Models relationships among classes. Normal associations can have multiplicities and role names on each side. Generalization relationships can be used to model single inheritance only in ActiveCharts.
Activity 	An activity models behavior for its context class. Activities can contain or call sub-activities (see section 6.2.1).
InitialNode ●	When a new activity execution is created, a new token is placed on all InitialNodes (there can be more than one) in the current

	activity.
ActivityFinalNode 	If a flow reaches an ActivityFinalNode, the current activity execution is terminated and all remaining tokens are destroyed.
FlowFinalNode  *(subshape of ActivityFinalNode)*	A FlowFinalNode destroys tokens. Other tokens in the current activity execution are not affected.
CallOperationAction  *(for calling custom code)*	Own code is implemented by action methods which are called by CallOperationActions. Name and parameters must match.
CallBehaviorAction  *(for calling an activity)*	Activities are called by CallBehaviorActions with a small fork symbol in the lower left.
SendSignalAction 	Signals sent using SendSignalActions are always targeted to the current execution. For targeting others, use BroadcastSignal instead. If an input parameter is defined, it must contain a signal instance to send.
BroadcastSignalAction 	Broadcast a signal to all executions of all activities at once. If an activity cannot handle this type of signal, it is ignored. Special expressions can further restrict possible targets, see section 8. If an input parameter is defined, it must contain a signal instance to send.
AcceptEventAction  *(for SignalEvents)*	Waits for events of type S or subtypes of S. AcceptEventActions without incoming edges are subject to special enabling and re-enabling policies, see sections 0 and 6.2.5. If an output parameter has been defined, it contains the received signal instance.
AcceptEventAction  *(for TimeEvents)*	Waits for the annotated time to elapse, see sections 3.3.2 and 6.2.3 for valid time specifications.
ForkNode/JoinNode	Splits / Joins flows.
DecisionNode/MergeNode 	Branching / Unification of flows. Outgoing edges of DecisionNodes should have non-overlapping guards, but this is not required. An optional `[else]`-guard can be used.
InputPin/OutputPin 	Models input- and output parameters for actions. Interfaces are generated that match the names/number and types of parameters. When implementing custom actions, names must be identical.
ActivityParameterNode 	Models input- and output parameters for activities.
CentralBufferNode	A CentralbufferNode buffers data tokens.

 *(you have to use the "ObjectNode" shape and type "<<centralbuffer>>" for yourself)*	
ControlFlow/ObjectFlow  *(non interrupting without/with guards and interrupting)*	Control or data tokens flow along activity edges. Guards can be used at all edges. When an interrupting edge is passed, all interruptible activity regions left are aborted.
InterruptibleActivityRegion 	An InterruptibleActivityRegion contains nodes that can be aborted by interrupting edges.
Note 	Notes can be attached to each element to supply comments and node configuration information (see section 8).

11.2 ACTIVECHARTS API

Purpose	Function
Miscellaneous Functions	
Writing to the ActiveChartsIDE Console	`ActiveCharts.ActiveChartsConsole.WriteLine(<string>)`
Exiting ActiveCharts properly in your application	`ActiveCharts.ActiveChartsEngine.Exit()`
Starting the activity of a class where "`autostart`" is set to "`false`".	`<object>.StartBehavior()`
Signal Handling Functions	
Send signal from your code to a target object	`<object>.SendSignal(<signal instance>)`
Get the current activity execution from an object	`<object>.CurrentActivityExecution`
Broadcast a signal to all executions of a all activities	`Activity.BroadcastSignalToAll(<signal instance>)`
Broadcast a signal to all executions of a specific	`<activity>.BroadcastSignal(<signal instance>)`

activity	
Method State Execution Handling Functions	
Get the method execution state for a method using `[HandleAbort]` or `[HandleSuspendAndAbort]` attributes	`MethodExecutionStateManager.RetrieveMethodExecutionState()`
Check if there is a request pending in the current method execution state	`<methodExecutionStateReference>.HasRequest`
Query the method execution state request	`<methodExecutionStateReference>.StateRequest` (can have the values `MethodExecutionState.State.Abort` or `MethodExecutionState.State.Suspend`)
Notify that the state is reached	`<methodExecutionStateReference>.NotifyStateReached()`

Literature

[Bock03]: Conrad Bock: "UML 2 Activity and Action Models", in Journal of Object Technology, vol. 2, no. 4, July-August 2003, pp. 43-53. http://www.jot.fm/issues/issue_2003_07/column3

[CSharp]: Robinson S., Nagel C. et al: Professional C# 3rd Edition, Wrox Press, 2004

[Evans03]: Evans, E.: Domain-Driven Design, Addison Wesley, 2003

[Fowler03]: Fowler, M.: Uml Distilled, Addison-Wesley Professional, 2003

[Larman04]: Larman, C.: Applying UML and Patterns, Prentice Hall PTR, 2004

[MDA]: Model Driven Architecture, http://www.omg.org/mda

[MDD]: Model Driven Development, http://www.mdsd.info

[MSDN]: Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes, http://msdn.microsoft.com/msdnmag/issues/04/05/c20/default.aspx

[Sarstedt05] Sarstedt S., Overcoming the limitations of Signal Handling when Simulation UML 2 Activity Charts. The 2005 European Simulation and Modelling Conference (ESM 2005).

[SKRS1] Sarstedt S., Kohlmeyer J., Raschke A. and Schneiderhan M.: A New Approach to Combine Models and Code in Model Driven Development. The 2005 International Conference on Software Engineering Research and Practice (SERP'05), Workshop on Applications of UML/MDA to Software Systems.

[SKRS2] Sarstedt S., Kohlmeyer J., Raschke A. and Schneiderhan M.: Targeting System Evolution by Explicit Modeling of Control Flows using UML 2 Activity Charts. The 2005 International Conference on Programming Languages and Compilers (PLC'05), Technical Session on Support for Unanticipated Software Evolution.

[UML05] Object Management Group, UML 2.0 Superstructure Specification, http://www.omg.org/cgi-bin/doc?formal/05-07-04

[XPath] Kay, M. (2004). XPath 2.0 Programmer's Reference. Wrox Press.