OVERCOMING THE LIMITATIONS OF SIGNAL HANDLING WHEN SIMULATING UML 2 ACTIVITY CHARTS

Stefan Sarstedt
Department of Programming Methodology and Compiler Construction
University of Ulm
89069 Ulm, Germany
Phone: +49 (0)731 50-24254
eMail: stefan.sarstedt@uni-ulm.de

ABSTRACT

Our approach to Model Driven Development uses UML 2 activity charts for simulating and implementing the control flow of an application. Signal handling is an important concept of these diagrams for synchronizing actions. We identified multiple problems related to buffering and distributing signals as well as to specifying targets for UML `SendSignalAction`s and `BroadcastSignalAction`s. We propose solutions for these issues by defining UML tags to configure activities and nodes and by introducing *SignalPath*, a subset of the XPath language, for querying the object graph at runtime to obtain targets for send actions. Our approach has successfully been implemented in our simulation environment for UML 2 activity charts.

INTRODUCTION

Modern software development efforts often rely on using executable models, making it possible to ease implementation, do rapid prototyping and to test application behavior in advance. A popular notation for modeling object oriented systems is the Unified Modeling Language (UML, see [Larman, 2005] and [Oestereich, 2002]), whose version 2 has recently reached its final state [UML, 2004].

Our approach to Model Driven Development uses UML 2 class diagrams for modeling the static structure and activity charts for specifying the dynamic behavior of a system (see [Sarstedt et al., 2005] for a more detailed discussion). The control flow, which is modeled by activity diagrams, is executed by a runtime component and code is called at specific points, which leads to a strong integration of modeling and coding. A tool has been developed that is able to simulate and debug the application control flow by allowing to execute the individual steps of a chart. Our simulation software will be presented at the tools presentation track and exhibition at the ECMDA 2005 conference [ECMDA, 2005].

Important concepts of activity charts are explicit signal sending and receiving actions that are used to synchronize control flow. While developing our simulation tool, we discovered various deficiencies in the final UML 2 specification [UML, 2004] regarding signal modeling and handling in the following areas:

- specifying signal targets
- buffering of signals
- distribution of signals
- notational problems

In the following sections we will describe and target each of these problems by using standard UML extension mechanisms. These concepts are already implemented in our simulation tool.

The structure of this paper is as follows. In the first section we describe our approach to modeling and simulation of a software system. A small example of an embedded system is included by which our solution will be discussed. In the following sections we show the shortcomings of the UML 2 specification regarding signal handling and how to overcome them. In the last sections we discuss our results and provide a conclusion as well as an outlook to future work.

OUR APPROACH TO SIMULATION

Our simulation environment uses activity charts for modeling the application control flow. These diagrams describe a sequencing of actions among which sending and receiving signals are the most important ones for synchronizing application flow. Each class can be associated with an activity (making it the "context class"

of its activity) that is started when an object is created. This way, each class can have multiple objects each with its own behavior instance, called "activity execution". Each execution has its own context object (these terms are taken from the official UML 2 specification [UML, 2004]).

An Example

Figure 1 shows the static structure of a simple alarm device. The device consists of a single controller class which has multiple sensors and sirens associated to it. The latter classes have behaviors which are described by activity diagrams shown in figure 2.

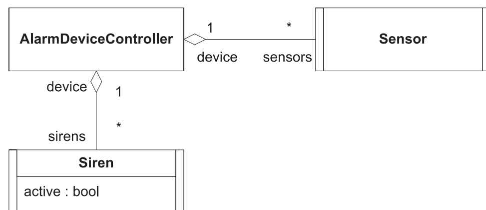

Figure 1: Static Structure of an Alarm Device

When a sensor object is created its behavior is started. This behavior executes code (with a `CallBehavior-Action` named "Detect Break-In") that detects a break-in. Triggered by the detection, an alarm signal is sent by using a UML 2 `SendSignalAction`. After handling a reset, the sensor object returns to detecting break-ins.

The siren behavior shown in the right part of figure 2 waits for an alarm signal to be received (using a UML 2 `AcceptEventAction`) and then executes code that turns the siren on ("Make Noise").

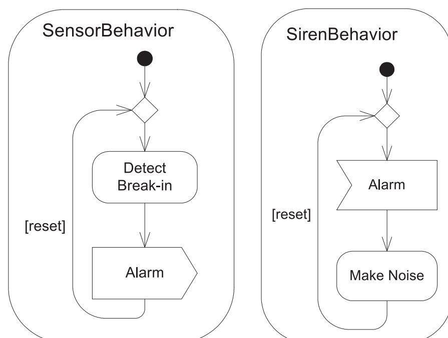

Figure 2: Behaviors of Sensors and Sirens

There can be multiple sensors and sirens (represented by individual objects) active at the same time, meaning that each one executes in its own context.

DEFICIENCIES OF THE UML 2

Although signal handling plays a central role in behavioral modeling in the UML, there are various shortcomings in the specification due to missing or ambiguous information. Additional problems emerge from inadequate usability of the models.

The following issues have been addressed by our group:

1. According to the UML 2 metamodel, targets of a `SendSignalAction` can only be provided by an explicit object parameter (via an `InputPin` [UML, 2004]) making it impossible to figure out the target solely by looking at the diagram. In addition, this mandatory parameter is often omitted in models to make them more concise – even in examples contained in the UML specification.

2. The way the targets of a `BroadcastSignalAction` are determined is considered to be a "semantic variation point" according to the official specification, which has to be dealt with in a useful way in a concrete implementation.

3. The UML specification describes that buffering takes place at an "object" without stating what is meant by this term.

4. There are often scenarios where buffering of signals can lead to incorrect behavior of a system or to overly complex models.

5. Distribution of signals to "objects" must be taken into account but is not discussed at all.

6. It is not possible to model a broadcast of signals graphically with UML 2, since there is no symbol proposed for this type of action.

In the next section we will discuss our approach to each of these problems.

OUR SOLUTION TO SIGNAL HANDLING

To provide the necessary configuration properties in UML models to be able to overcome the deficiencies, we use the standard "tag"-mechanism for extending the UML [UML, 2004]. Tags designate simple key/value pairs which add additional information to UML model elements. By creating our own tags our activity charts execution engine is able to adjust the handling of signals at runtime accordingly.

Target Specification

To overcome the problems of explicit target specification for `SendSignalAction` and `BroadcastSignalAction` (issues 1 and 2) we propose to use a subset of the well-known XPath language, called *SignalPath*, that we use for querying our current object graph, resulting in a set of nodes where the signal should be sent to. XPath has previously been successfully applied not only for searching Xml documents, which is its primary intention, but also for iterating over arbitrary object networks (see [Saxon, 2003] and [Sudarsan and Gray, 2005]). An XPath expression consists of multiple steps, each one (starting from a "context object", which is not to be confused with the conforming UML term) resulting in a node set that is processed by the next step. Predicates can be applied at each step to filter the current set. See [Kay, 2004] for a comprehensive overview over the XPath language, including node sets and filter predicates.

To attach the targeting information to a send-action in an UML diagram, we introduce a new tag called `signalPath` that is written in a comment. To also overcome issue 6 (no graphical representation for a `BroadcastSignalAction`) we propose to use a new symbol, which is a twofold `SendSignalAction`. Figure 3 shows the revised SensorBehavior from figure 2 which incorporates our new concepts:

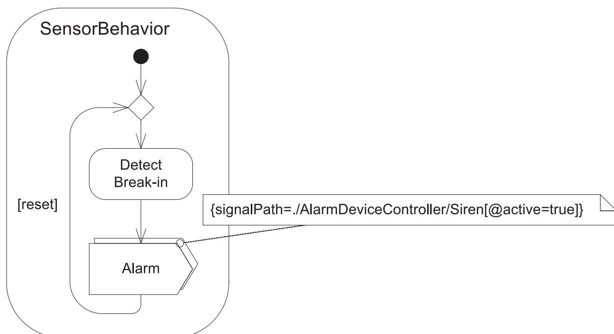

Figure 3: Using SignalPath Expressions

By specifying the `SignalPath`-expression

```
./AlarmDeviceController/Siren[@active=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 `AlarmDeviceController` has firstly to be traversed – resulting in a node-set that contains only the single `AlarmDeviceController` 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 `[@active=true]` only those siren objects remain in the result set, that have its `active` attribute set to `true`. Thus, the targets of alarm signal encompass all active sirens that can be reached by our current object graph. Since a set of nodes is targeted instead of a single one, we have to use a broadcast instead of a simple `SendSignalAction`.

To type-check SignalPath expressions and finally query the objects, we use the classes metamodel and the reflection capabilities of C#.

Signal Buffering

According to [UML, 2004], all signals are buffered at an "object" without further clarifying what is exactly meant by this (see issue 3). An obvious interpretation would be to consider "object" to be the context object of an activity, but that would contradict the specification in that there can be activities without context objects (see the UML activities metamodel [UML, 2004]). We therefore consider it useful to use an *activity execution* as the location to buffer signals. When there is any `AcceptEventAction` that waits for a signal contained in the queue, the signal will be removed and the action is terminated. When using SignalPath-expressions, the activity executions can easily be located, since each object can have at least one activity execution associated with it at runtime. Therefore, we can still use queries over the classes metamodel for targeting these executions.

Reconsider figure 2. When multiple sensors detect a single alarm (e.g. a glass breaking and a movement sensor could have successively been activated) each of them sends a signal to the sirens. Since a siren only handles one signal at a time, the other ones are buffered leading to misbehavior after a `[reset]` - namely instantly making noise again although the same break-in has already been handled (issue 4). We therefore propose that it must be possible to disable buffering for all or only certain types of signals. This issue could be targeted using standard UML notation by putting each `AcceptEventAction` in its own (sub-)activity which is called by the primary activity. Since buffering takes now only place when the `AcceptEventAction` together with its sub-activity is activated, this would in fact lead to correct handling (although signal distribution among activities is not targeted either, see below), but also to overly complex models.

To keep models clear, we introduce a new tag called `bufferSignals` for this purpose in figure 4:

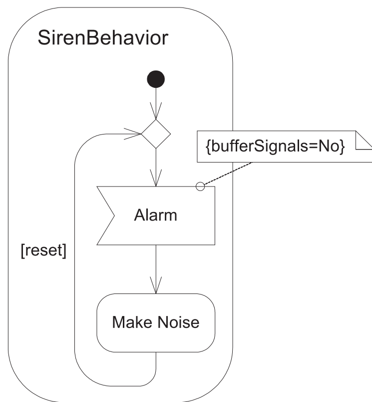

Figure 4: Disable Buffering of Signals

By setting this property to `No`, buffering is disabled for any `Alarm` signal that is received by `SirenBehavior`. Only when the `AcceptEventAction` of type `Alarm` is active when receiving a signal, it will be handled. All other signals are discarded, leading to a correct behavior for the alarm device.

To disable buffering for all signals in an activity, the `bufferSignals`-tag can also be attached to the activity itself. This global setting can again be overridden by local tags having values set to `Yes`, making the buffering configuration flexible.

Signal Distribution

UML 2 activity diagrams can also be nested, i.e. activities may call other activities for making models more clear. When sending signals to a "top-level"-activity it is not obvious if those signals are to be distributed to the contained activities (see issue 5). Consider for example the following behavior for the `AlarmDeviceController`:

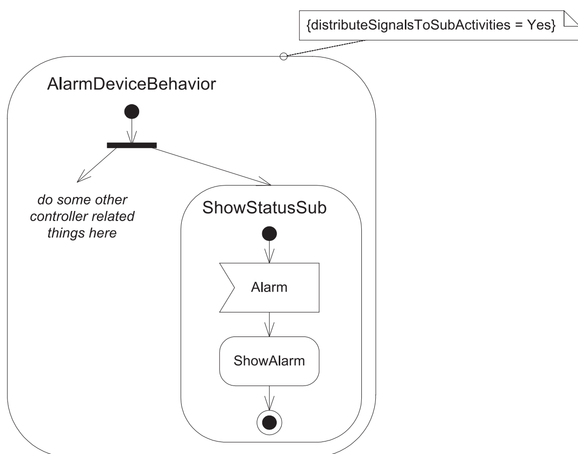

Figure 5: Signal Distribution

The behavior contains another one called `ShowStatus-Sub` which is responsible for showing the alarm status on a panel (putting activities into other ones is just a notational abbreviation, in fact it acts as a substitute for a call [UML, 2004]). To handle signals properly, a tag called `distributeSignalsToSubActivities` can be added to the top-level activity. When set to `Yes`, all signals will be sent to all contained activities. This setting can again be overridden for individual sub-activities in the same way as it was the case with buffer-specifications. Also note that " | `./AlarmDeviceController`" must be added to the SignalPath expression shown before to make the sensors send the alarm also to the controller and not only to the sirens.

DISCUSSION

Our decision to use a subset of the popular XPath language to find target objects for `SendSignalAction`s and `BroadcastSignalAction`s was inspired by Object-XPathNavigator [Saxon, 2003], which uses XPath to query object graphs. By adding this information directly to the send actions in the model, it becomes obvious where the signals are sent. Normally, explicit object pins have to be used in the model, which still leaves the actual type of target unclear until runtime. See also [Sudarsan and Gray, 2005] for a recent work that uses XPath to query metamodels.

Issues of targeting, buffering and distribution of signals in UML 2 activity charts have not yet been discussed elsewhere to the best of our knowledge. We provide an elegant way to specify these properties by using the standard tag extension mechanism of the UML.

We also offer a solution to broadcast actions that are considered to be a "semantic variation point" in the current UML specification. This also includes a graphical symbol, which is needed for use in UML diagrams.

CONCLUSION AND FUTURE WORK

We presented an approach to overcome the limitations regarding signal handling in the UML 2. We discussed issues of targeting, buffering and distribution of signals in activity diagrams and provided a solution that uses UML tags and a subset of the XPath language. In our future work we want to utilize role names in addition to class names for SignalPath expressions. Furthermore, the XPath subset will be extended to include custom user functions which makes target selection even more flexible.

Our approach has been successfully implemented in a simulation tool for UML 2 activity charts which will soon be available for download.

References

[ECMDA, 2005] ECMDA (2005). European Conference on Model Driven Architecture. http://www.ecmda-fa.org.

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

[Larman, 2005] Larman, C. (2005). *Applying UML and Patterns*. Prentice Hall.

[Oestereich, 2002] Oestereich, B. (2002). *Developing Software with UML*. Addison-Wesley Professional.

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

[Saxon, 2003] Saxon, S. (2003). XPath Querying Over Objects with ObjectXPathNavigator. http://msdn.microsoft.com/library/default.asp? url=/library/en-us/dnexxml/html/xml03172003.asp.

[Sudarsan and Gray, 2005] Sudarsan, R. and Gray, J. (2005). Meta-Model Search: Using XPath to Search Domain-Specific Models. In *The 2005 International Conference on Software Engineering Research and Practice (SERP '05)*.

[UML, 2004] UML (2004). Object Management Group, UML 2.0 Superstructure Specification, Document 04-10-02. This is supposed to be the final version.