

Targeting System Evolution by Explicit Modeling of Control Flows using UML 2 Activity Charts

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

***Abstract**—The Model Driven Development propagates facilitation of modification and maintenance of software. Abstraction and graphical formalization combined with code generation indeed facilitates the Unanticipated Software Evolution, but is often restricted to early phases of software development. Usually, only class stubs are generated from class diagrams. We try to enhance the use of graphical models by making the control flow of an application explicit in activity diagrams which are interpreted and executed by a runtime environment. This allows to incorporate changes concerning the control flow in a transparent and concise way. Since excessive modeling can result in a complex model that is difficult to create and maintain, we do not force the developer to express everything in models; the degree of modeling versus coding can be chosen. We explain how our approach supports integrating different kinds of changes and illustrate this by a small example of an embedded system.*

1. Introduction

Many well-known conventional software development processes used in industry still do not consider changes of requirements that occur during the process. Some processes try to face this problem by focusing on the flexibility of the process (e.g. agile approaches [1]), but these only try to adapt the development workflow to the rapidly changing environment. Even with this flexibility, every modification in the requirements or technologies is difficult and requires a lot of effort to implement since they do not offer techniques for integrating these changes into code.

One possibility to address the problem of changing technologies is abstraction. If an application is modeled with an abstract (graphical) language, it is only necessary to create a new code generator to transfer the modeled logic to a new technology. For example, many CASE-tools implement automatic code generation for different

programming languages out of static UML class diagrams.

Our approach is to enhance the idea of abstraction towards functionality: We use UML 2 activity diagrams to separate the control flow of an application from additional framework code. These abstract artifacts can be modified much easier than hand-written code. It is, of course, not expedient to model every algorithm in graphical notation, since this will become too complex quite fast. Therefore, in our approach it is possible to implement actions of activity diagrams directly in program code. The activity charts are interpreted by a runtime environment, and when the appropriate action is invoked, the hand-written code will be executed.

The next section explains our architecture in more detail before we show how this approach can help to react accordingly on unanticipated software change requests. Section 3 illustrates the possibilities with an example. We discuss our approach in section 4 and compare it with other ideas in section 5. Finally, section 6 gives an outlook to our further work.

2. Architecture

The aim of our project is to make the control flow of applications explicit by using UML 2 activity diagrams without losing the possibility to use regular code. Otherwise, this approach would not be very helpful, since the complexity of graphical models usually increases exorbitantly so that they become useless. To face this problem, we allow the developer to implement actions in regular C# code. This section illustrates the architecture of our system which implements this approach. A more detailed description of it can be found in [2]. Furthermore, we

will explain how this architecture can be used to integrate different kinds of software changes in an easier way.

Figure 1 shows the design of our system. We use UML 2 class diagrams to build the static structure and — as mentioned above — activity charts to model the dynamic behavior of an application. These models can be seen as platform independent insofar as they do not contain any platform or programming language specific information.

The information in the class diagram is used to generate class stubs and code to instantiate associations between classes. They can be completed with code for the corresponding actions and additional “helper” code. A C# compiler which may be enhanced by semantic checks incorporating information out of the diagrams translates the code – which does not contain the control flow of the application – into the executable intermediate language of .NET.

The control flow is managed by a runtime environment that controls the application according to the activity charts. First of all, it imports the charts (in our case from Microsoft Visio 2003) and converts them into an internal representation. After that, it interprets the charts, creates control and data tokens, takes care of signals and concurrency, and executes the appropriate code in the C# classes when an action is invoked.

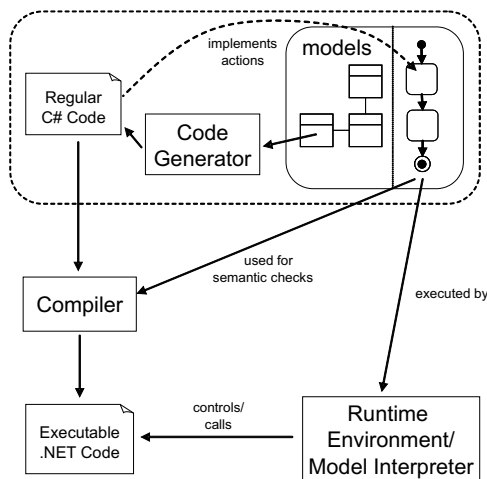


Fig. 1. Architecture

This architecture allows to respond comfortably to different kinds of unanticipated software changes. Due to the partial abstraction, it is comparatively easy to react to changes in technology. It is only necessary to implement a new code generator or possibly translate the runtime environment into another language. This corresponds to the basic idea of Model Driven Architecture [3] and is not fundamentally new — except that we offer an executable semantics for UML 2 activity charts.

Change requests concerning the requirements — and therewith often the design of the application — are much more intrusive and have a high impact. According to

our approach, we can differentiate between the following effects depending on the type of the change request:

- 1) Only static (data) structure has to be changed: new classes, attributes, etc.
- 2) Only the functional behavior has to be changed
 - a) Changes affects only the control flow: new signals, decisions, etc.
 - b) Changes affects only a well-defined scope: algorithms, implementation of an action, etc.
- 3) Structure *and* functional behavior have to be changed: (normal case)
- 4) Partial or complete redesign has to be done: (worst case)

We will show briefly how our approach can handle these different modifications. These ideas will be illustrated by a few examples in the next section, discussed in more detail in section 4 and finally compared with other approaches in section 5.

The first item can be easily dealt with by the use of class diagrams combined with code generation. This is already general practice in modern development environments including more and more powerful refactoring tools.

Changes in the control flow can be handled in a much more transparent way than it would be possible with hand-written code. If the actions were well-defined (that is, like reusable components), it might be possible to leave the underlying code untouched, but only to adapt the activity charts to implement the emerged changes. This obvious transparency of control flow reduces the effort for maintenance radically. The encapsulation of algorithms or certain functionality in actions with well-defined interfaces allows the developer to transparently exchange this code. This corresponds to a regular modularization, but we believe thinking in sequences is more natural.

In most cases, the first and the second item will occur simultaneously (expressed in the third item). If the changes enforce a redesign of the whole project, our approach does not offer a benefit compared to regular coding, but it is also not worse. Nevertheless, it might be possible to minimize the probability for this case by an appropriate design of the control flow.

With this architecture it is even possible to change the control flow during the execution of an application. For a detailed treatment of this topic see section 4. Theoretically it might also be possible for the application itself to change its own control flow during runtime in a transparent way. This offers new options whose potentials we cannot estimate as of now.

3. Example

To show how certain changes in requirements can be handled by our approach, we introduce an example consisting of a simplified alarm device. We first show

an implementation of a basic system and then extend it towards an advanced version in subsection 3.2. Finally, we examine a change of action code in subsection 3.3.

3.1 Simple Alarm Device

The static structure of a simple version of an alarm device is shown by the UML 2 class diagram in figure 2. It consists of a single class called “AlarmDevice” which acts as a controller and has multiple sensors and sirens associated with it. Each class is associated with a behavior in terms of a UML 2 activity diagram.

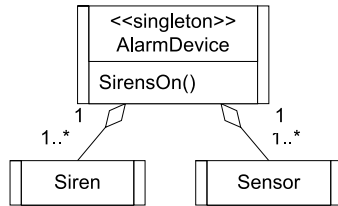


Fig. 2. Static Structure

Figure 3 shows our supported basic elements of UML activity diagrams. For a detailed reference of the UML 2 syntax and semantics, refer to [4].

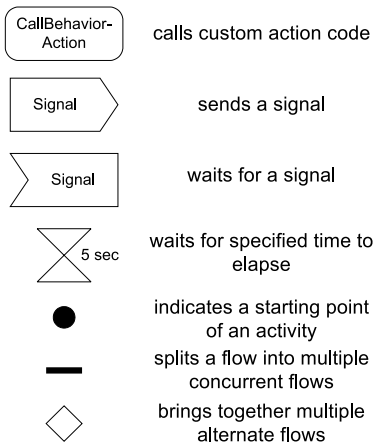


Fig. 3. Basic UML Activity Chart Elements

Our basic system requires that if some sensor detects a break-in, all sirens are switched on. The according behaviors are shown in figures 4 and 5. A sensor action detects a break-in (depicted by a rounded rectangle) and then sends an event called “building break-in” to the alarm device behavior, which in turn waits for those events – shown by an AcceptEventAction [4]. The device thereupon turns the sirens on and waits for the alarm state to be cleared by the operator. It finally returns to the initial waiting state.

As already described in the previous section, all behavioral models are interpreted by a runtime component to control the behavior of the application. Additional code is only needed for implementing the actions. An

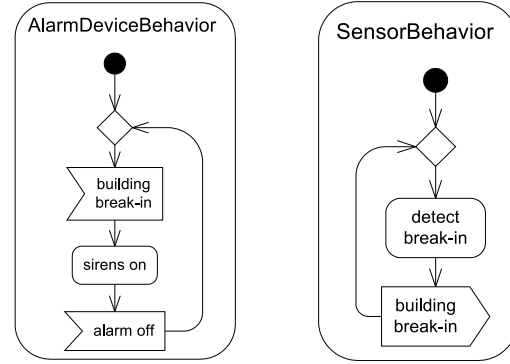


Fig. 4. Basic Alarm Device Fig. 5. Basic Sensor

example of such an action implementation is presented in figure 6, where sample code for “sirens on” (see figure 4) is shown. This operation is located in the class “AlarmDevice”, as can be seen in the class diagram. The link between the action and the method is ensured by same names. Other methods are not shown here for brevity reasons.

```

class AlarmDevice {
    private ArrayList Sirens;
    public void SirensOn()
    {
        for(int i=0;i<Sirens.Count;i++)
        {
            Sirens[i].On();
        }
    }
}
  
```

Fig. 6. Hand-written Code for Activating all Sirens

3.2 Extending the Alarm Device

A customer could require the alarm device to be able to handle different zones, like a living room, a cellar, or a garage. It should be possible to enable and disable the zones separately (in case of being at home, one would perhaps only want the cellar to be monitored), but, if any sensor of an active zone detects an incident, all sirens shall be triggered – even those of currently inactive zones.

This requirement demands a non-trivial extension of the basic version where normally larger parts of the code would have to be rewritten. In our case, this requirement can be handled by solely adjusting the structure and behaviors modeled by UML diagrams.

First, the static structure must be extended to reflect zones. Figure 7 shows the updated class diagram. Sensors and sirens are now connected to a zone, zones are controlled by the singleton “AlarmDevice” class.

The alarm device and sensor behaviors must only be marginally changed (see figures 8 and 9): a new signal

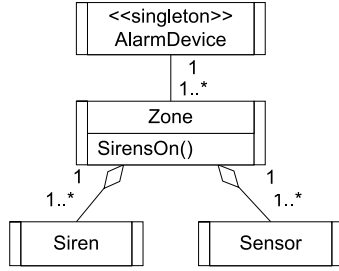


Fig. 7. Advanced Static Structure

event is introduced to model a “zone break-in”. The activation of sirens are now handled by the zone where they belong to instead of the alarm device. To initiate the alarm in the other zones, the device now sends a “building break-in” to all zones.

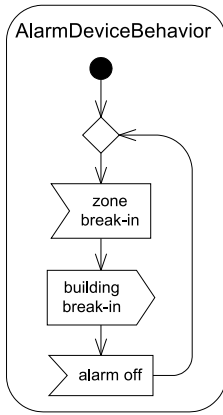


Fig. 8. Advanced Alarm Device

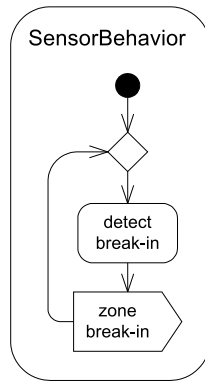


Fig. 9. Advanced Sensor

The zone behavior is shown in figure 10. It waits for either a “zone break-in” or a “building break-in” and then turns its own sirens on. If one of those signals is received, the waiting action of the other signal is interrupted — modeled by using an UML interruptible activity region [4]. This is essential, because the sirens of the zone that detected the break-in would otherwise be activated twice. At the end, all zones again wait for the alarm to be turned off.

To incorporate these changes into our basic application, no further action concerning the control flow is necessary: the adaption is reflected by our model interpreter at runtime. The only changes that need to be made concern the static structure of the application. The “Zone”-class is introduced and the method “SirensOn()” has to be moved from the “AlarmDevice”-class to the “Zone”-class. These changes can be easily made using refactoring and code generation tools.

3.3 Modification of Action Code

Requests dealing with changes that affect only a well-defined scope (e.g. algorithms, see section 2) are

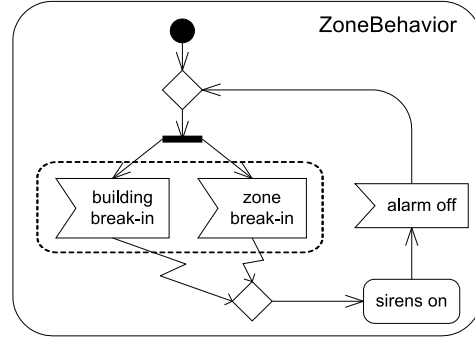


Fig. 10. Zone Behavior

equally easy to incorporate, just like with conventional programming. If it should for example be possible to disable single sirens, modifications shown in figure 11 and figure 12 are needed.

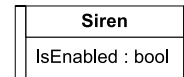


Fig. 11. Modified Siren Class

The “Siren”-class has to be extended with an attribute that indicates the enabling state. The code for “SirensOn()” must be changed accordingly to only switch on sirens that are enabled. In contrast to a pure translationist approach of MDA (see also discussion in section 4), we offer the possibility to use hand-written code to make these kinds of changes. The activity diagrams do not have to be altered in this case.

```

class Zone {
    private ArrayList Sirens;
    public void SirensOn()
    {
        for(int i=0;i<Sirens.Count;i++)
        {
            if (Sirens[i].IsEnabled)
                Sirens[i].On();
        }
    }
}

```

Fig. 12. Modified Code for Activating all Sirens

Another possibility to handle this change request would be to use a technique like AOP [5], for example.

4. Discussion

To handle complexity and maintainability in a rapid changing environment, different techniques were developed in software engineering. The encapsulation of information and functional duty is a fundamental precept in building software, ranging from methods and functions in imperative programming languages, over to the

use of object orientation with its techniques like classes and inheritance, and up to modern components. In our approach we define a new abstract method of encapsulation: using activities and actions. In the following we discuss how the handling of certain unanticipated changes is affected by our approach.

In section 2 we defined certain types of change requests. These categories of changes can be handled more or less in a practicable way.

Changes within the static structure (see section 2, category 1) can be easily adapted by remodeling the class diagram. This can be seen in section 3, see figure 2 with the original and figure 7 with the reworked structure of the alarm device. New application code will be generated out of the model and compiled from there. In most cases the modification of the static structure of an application affects the functional behavior, so that change requests resulting in effects of category 1 appear rarely.

In changes of the functional behavior (category 2) we distinguish between two subcategories: changing the control flow (a) or changes in a well-defined scope (b). If it is possible to map the modification to category 2a, its implementation is simple and can be done in an effective way. The modification could, for example, be the change of the flow or a new signal. By adapting the relevant activity chart this modification can be handled easily and the change of the functional behavior is implemented. This is, for example, shown in section 3, where the sensor behavior is extended, and only the break-in signal changes (see figure 5 and figure 9). In this case, it is even possible to make the necessary modifications at runtime. They are immediately reflected during execution due to the runtime environment. This, of course, can only be done by developers, not by end-users. Moreover, situations when changing only functionality in the activity diagrams are rare, and therefore we can use this benefit only in a very restricted manner.

The implementation of a change request of category 2b is marginally more difficult. In case the change can be mapped to actions, the integrated code has to be changed. This is illustrated by an example in section 3.3.

In most cases, unanticipated changes concern both the static structure and the functional behavior (category 3). The developer has to identify the involved parts of the application and make the necessary changes. However, these changes can be easily done with modifications of the diagrams and subsequent code generation, as described in section 3.

A fundamental change request can lead to a partial or a complete redesign of the application (change of category 4, see section 2). By modeling the control flow of an application with UML 2 activity charts we bridge the gap between design and implementation. These diagrams, created during the design phase of the software development process, are seamlessly reused for the implementation phase. Therefore, if a redesign

is necessary, we shorten the time developers need to implement the respective application.

By-and-by the underlying technology of software systems changes. For example, operating systems upgrade over the years or new releases of virtual machines are adopted. Custom software could become useless if new requirements to the underlying technology appear. To handle the problem of changing technologies, we use a platform independent model (PIM), according to the Model Driven Architecture initiative from the Object Management Group [3]. For the transformation into a platform specific model (PSM) and further into implementation code various templates can be used, each for a specific target platform. Thus, just like in the vision of MDA, only templates have to be adapted or constructed if the underlying technology changes. Therefore, we can react appropriately to this kind of changes with our approach. In our approach this vision of the model-driven development is substantiated: we integrate models and code. Thereby, we are in between the two main interpretations of MDA, which are the elaborationist [6] and the translationist [7] approach. The chief difference between these approaches is the level at which a developer changes the functionality of the system. In the elaborationist approach a developer can elaborate the PIM, the PSM, and the code. In the translationist approach the developer can only elaborate the PIM, which is translated directly into code. Our approach is not elaborationist, as the developer changes the functionality of the system within the PIM, but not within the code. It is not translationist either, because the developer can change the functionality using regular C# code within the PSM; this contradicts the idea of the translationist approach. We use the advantages of both approaches: the functionality is modeled abstractly to handle complexity, yet the developer is free to use code for certain purposes, and therefore, not all of the behavior has to be modeled. The implementation of unanticipated changes benefits from this approach, as mentioned above.

Finally, however, there are also a few drawbacks: Since modeling activity charts in our approach is essential, developers have to learn the new “language”. They have to understand semantics and modeling techniques of the UML 2 activity diagrams in detail. Experience and exercise will be necessary to react properly to upcoming changes. Certainly, applications created with our approach will become less efficient than hand-written ones. This circumstance stems from the increased abstraction.

5. Related Work

Another way of handling unanticipated changes is to address this problem through component composition (e.g. see [8], [9], and [10]). The reusable components are assembled in various ways to accomplish user requirements. We have a kind of orchestration possibility

in our approach, too: the actions in our activity diagrams, enriched with pre- and postconditions, can be seen as different components. We therefore offer a possibility of orchestrating single components, therefore. The way in which the components interact is defined precisely in the flow of the activity diagrams, supplemented by the use of signals for synchronization purposes.

To react appropriately to changing requirements and therefore to changes in functionality, some ideas like AOP [5] try to separate orthogonal aspects (e.g. persistence, logging) from each other and from the application logic. The authors of [11] propose to view the ability to update a system at runtime as an aspect. Although this approach offers many interesting possibilities, it is not always useful and in some cases inefficient (e.g. transactional logic in persistence abstraction [12]). Instead of extracting certain concerns (e.g. persistence or live update) out of software, we extract the functional behavior of an application itself as a control flow modeled with UML 2 activity charts to handle complexity.

To model functional behavior, Statecharts are a popular and well-investigated formalism [13]. Although these charts have well-defined semantics, it requires quite a lot of practice and experience to model functional behavior in Statecharts. In our opinion, thinking in actions and decisions is more intuitive than thinking in states. Therefore, activity charts are easier to learn and understand even by customers. In our further work, we want to compare these two formalisms in detail.

In [14] the authors describe an approach to address unanticipated software by enabling the end-users to change the software behavior from the user interface. In contrast, although the focus of our approach is not to support changes or updates at runtime, we support this aspect anyway, even though in a very restricted way, as mentioned in the previous section.

Modeling the functional behavior of an application with activity charts is similar to workflow modeling. In contrast to workflows management systems, our modeling targets the behavior of a software system, not the flow of work or collaboration between individuals or departments [15]. Therefore, we neither have a document management system nor a monitoring system for business processes. We use some aspects of the workflow technology, yet we adapt and exercise them in another environment.

6. Conclusion

We described a method which can be used to target Unanticipated Software Evolution: By explicit modeling the control flow of an application with UML 2 activity charts, we provide a possibility to handle unanticipated changes in a practicable manner.

By means of an example we showed how certain change requests can be handled with our approach.

The implementation of specific unanticipated changes benefits from abstracting the functional behavior of an application with graphical models. Our approach to model-driven development allows us not only to properly react to changing requirements, but also to handle ever-changing underlying technologies. In addition, we discussed the advantages and disadvantages according to the handling of unanticipated changes and related our approach to others.

The formal semantics of UML 2 activity charts we implemented in our approach will be treated in a future publication. We will evaluate the aspect of handling unanticipated changes in our approach in a practical term at the University of Ulm. It is our ambition to compare the technical assistance used in our approach to the conventional way of handling changes during software lifetime.

References

- [1] K. Beck and M. Fowler, *Planning Extreme Programming*, 1st ed. Addison-Wesley Professional, 2000.
- [2] S. Sarstedt, A. Raschke, J. Kohlmeyer, and M. Schneiderhan, "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 (UMSS'05)*, 2005, to appear.
- [3] (2003) Object Management Group, MDA Guide 1.0.1. [Online]. Available: <http://www.omg.org/docs/omg/03-06-01.pdf>
- [4] (2004) Object Management Group, UML 2.0 Superstructure Specification. [Online]. Available: <http://www.omg.org/cgi-bin/doc?ptc/2004-10-02>
- [5] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.
- [6] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture — Practice and Promise*. Addison-Wesley, 2003.
- [7] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.
- [8] I. Sora, N. Janssens, P. Verbaeten, and Y. Berbers, "A Component Composition Model to Support Unanticipated Customization of Systems," in *First International Workshop on Unanticipated Software Evolution (USE2002)*, Malaga, Spain, June 2002.
- [9] J. Buckley, "Adaptive Component Interfaces," in *First International Workshop on Unanticipated Software Evolution (USE2002)*, Malaga, Spain, June 2002.
- [10] J. Gorinsek, S. V. Baelen, Y. Berbers, and K. D. Vlaminck, "Managing Quality of Service during Evolution Using Components Contracts," in *Second International Workshop on Unanticipated Software Evolution (USE2003)*, Warsaw, Poland, April 2003.
- [11] J. Gustavsson, T. Staijen, and U. Assmann, "Runtime Evolution as an Aspect," in *First International Workshop on Foundations of Unanticipated Software Evolution (FUSE2004)*, Barcelona, Spain, March 2004.
- [12] S. Soares, E. Laureano, and P. Borba, "Implementing Distribution and Persistence Aspects with AspectJ," in *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2002.
- [13] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.
- [14] C. Letondal and U. Zdun, "Anticipating Scientific Software Evolution as a Combined Technological and Design Approach," in *Second International Workshop on Unanticipated Software Evolution (USE2003)*, Warsaw, Poland, April 2003.
- [15] W. van der Aalst and K. van Hee, *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.