

A New Approach to Combine Models and Code in Model Driven Development

S. Sarstedt, J. Kohlmeyer, A. Raschke, M. Schneiderhan

Dept. of Computer Science
Programming Methodology and Compiler Construction Group
89069 University of Ulm, Germany
Phone Number: +49 (0)731 50-24254
Email: mdd@informatik.uni-ulm.de

***Abstract**—We present an approach to Model Driven Development that combines models and code in a new way. UML 2 activity diagrams created for modeling control flows during analysis and design are seamlessly reused for the implementation phase. A model interpreter executes these diagrams and calls handwritten code at specific points. The degree of functionality modeled and coded can be chosen freely by developers which should improve acceptance of modeling tasks. A few language extensions are introduced into an ordinary C# compiler to achieve a tight integration with the models. We illustrate our approach by means of a small example of an embedded system.*

1. Introduction

To face the known problems in software development (high quality, a short time to market, easy maintenance), especially in the context of rapidly changing requirements and technologies, it is necessary to break new ground. Modern development processes include more or less comprehensive analysis and design phases. During these phases several models are created to a certain extent which usually cannot easily be reused in later phases e.g., implementation and/or testing, and therefore they are not maintained.

One of the recent approaches to address this issue is the model-driven architecture initiative [1] proposed by the Object Management Group (OMG). The main focus of this concept is to bridge the gap between the design and the implementation phase. According to MDA, this should be done by enriching the design artifacts (e.g., UML 2 models) with all required information for building a whole application out of the models and extensions. This platform independent model (PIM) is then transformed into a platform specific model (PSM), including information about data types, database

interfaces, etc., and eventually into code of a specific programming language.

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. Therefore, additional languages were introduced (e.g., OCL, different syntaxes for Action Semantics [2, chapter *Actions*]). These languages are not generally used as they do not offer as many possibilities and add-ons as modern programming languages.

Starting from this problem, our idea is to integrate the model into the regular code and vice versa. We model the control flow of an application with UML 2 activity diagrams which will be interpreted by a runtime environment. Some actions are realized in C# code which will be executed, when the appropriate action is invoked, the others are modeled. Not to be confused with additional (framework) code, we associate the handwritten “action code” with the diagram actions by new keywords. In doing so, the created models are linked to the code and this makes it possible to perform semantic checks during the compilation.

The structure of this paper is as follows: in the next section we describe the overall architecture before we illustrate it with an example in section 3. Then we discuss our ideas in relation to other work and finally we give an outlook to our future work.

2. Architecture

The aim of our project is to integrate UML abstractions into a regular (C#) compiler and tightly integrate modeling and coding tasks while leaving the respective degree to the developer. To reach this goal, we have devised the following two aspects:

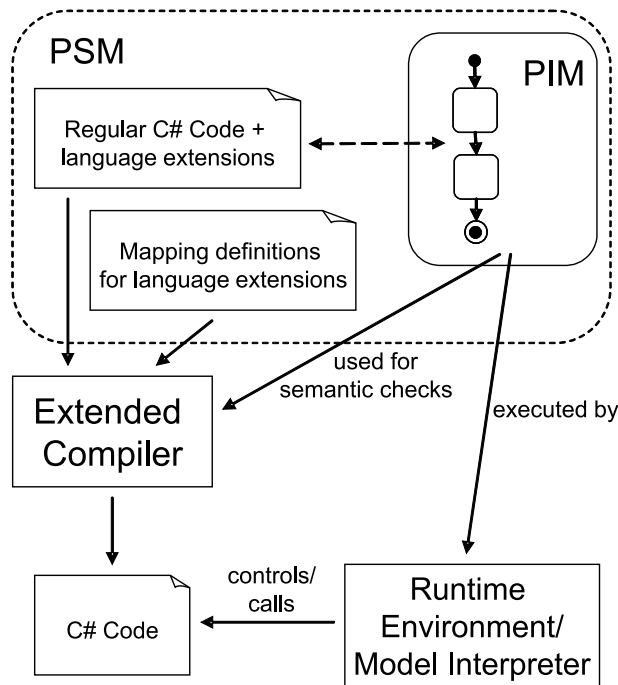


Fig. 1. Architecture

- 1) Making the control flow of an application explicit in UML 2 activity diagrams retaining the possibilities of regular “hand-written” code.
- 2) Associating the “hand-written” code with the diagrams to reflect the original intention.

The first item faces the problem of the gap between the analysis/design and the implementation phases. Usually the control flow of an application is modeled with activity charts. Due to the interpretation of the activity diagram by the runtime environment, it is no more necessary to (re-)code this control flow in a programming language. Thus, the modeled diagrams can directly be reused in the implementation phase.

Since these charts are defined hierarchically, it is possible to refine each action with a new activity diagram. But, in some cases (see section 3) it is more useful to describe an action in regular C# code. 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. This is even more useful, as some tasks are almost impossible or at least too extensive to be modeled (e.g., GUI-Extensions, complex computations, database access, etc.). The computed results can be used to affect the control flow at decision nodes, for example. The explicit control flow in form of a graphical notation makes it much easier to maintain the source code. If the actions were well-defined (that means, like reusable components), it even might be possible to leave the underlying code untouched, but only to adapt the activity

chart(s) to implement emerged change requests from customers. In addition, the control flow can be simulated before any action is implemented. This provides the opportunity for rapid prototyping.

The second aspect is realized by the introduction of new keywords. These keywords were adapted from the action semantics specification and couples the code closer to the model by reflecting the original intention. For example, an implemented action is labeled as an “action” and not declared as a method. First of all, this avoids confusion, but beyond this, it is now possible for the compiler to perform semantic checks integrating all information from the model and code together. We only use a few keywords to keep the original language simple and clear. Otherwise, this would have been counterproductive to the demand on acceptance.

Figure 1 illustrates how these aspects are realized in our system. Including the information about the control flow in the form of activity charts, an extended compiler translates the coded actions (augmented with the new primitives) into ordinary C# code. For this operation, additional mapping definitions for the new keywords were used. Finally, the already mentioned runtime environment is executed together with the generated code. The runtime environment interprets the model(s) and maps the modeled control flow into a real application flow. It handles not only primitive constructs of the activity charts but also signal treatment, parallel processing, and data flows. At specific points (the actions) the corresponding code is executed and — if leaving an interruptible region, for example — also aborted.

3. An Example of Model and Code Integration

We motivate our approach with an example of a simplified alarm device. A detailed reference of UML 2 syntax and semantics used in this section can be found in [2].

The static structure of our alarm device consists of a class called “Controller” that manages the system and is a singleton (i.e. there exists only one instance at runtime). Associated with that class are sensors and sirens. All classes are “active”, which means that instances run in their own threads of execution. Each class is also associated with a behavior in terms of a UML 2 activity diagram.

3.1 Explicit Modeling of Control Flow

Figures 2 and 3 show UML 2 activity diagrams modeling the control flow for a system function covering heartbeat processing. For brevity we omit other parts of the behavior dealing with break-in detection and alarm signaling.

Each sensor issues a heartbeat signal every five seconds (illustrated by the UML timer symbol) so that

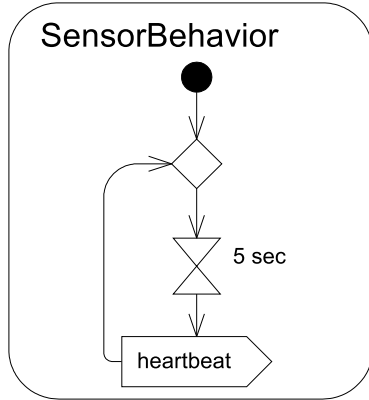


Fig. 2. Sensor Behavior

the controller knows the sensor is still connected and alive — otherwise one could simply cut the cable to it without consequences. In UML terms, this means that an instance of a “heartbeat” signal, which is a subtype of the UML signal class, is being created and, in our case, sent to the controller class. The controller behavior shown in figure 3 first initiates an “init” action responsible for initialization and then waits for those heartbeats. If one heartbeat was received, it does some processing, shown by the action “process heartbeat”, which will be discussed below. If there was no heartbeat signal from any sensor within ten seconds, an alarm will be issued and alarm processing will be performed. The signal receiver and timer symbols are surrounded by an interruptible region [2] and outgoing edges are interrupting edges. The UML semantics define that, if one of those interrupting edges is passed, the other actions of the region are aborted. This is useful in our case since the timer event should not execute any more until the next loop of the controller (see figure 3), if a heartbeat was received.

These two models can be considered as platform independent (PIM) in the sense of MDA [1]. Instead of handcoding or transforming these models into code, like other MDA approaches propose, they are being executed by an interpreter (see section 2 and [3]).

During execution, other parts of regular code are called. As an example, consider the action called “process heartbeat” shown in figure 3. Since many sensors may be active at runtime and each sensor sends heartbeat signals to the controller, it must be assured that each one is taken into account when computing timeouts. If there was a fixed number of sensors (which is not possible here because of different configurations of sensor sets), one could define separate heartbeat signal types for each sensor and duplicate the relevant model portion shown, however one would lose clarity and maintainability of the model. Instead we decided to model only one type of heartbeat and do all processing in

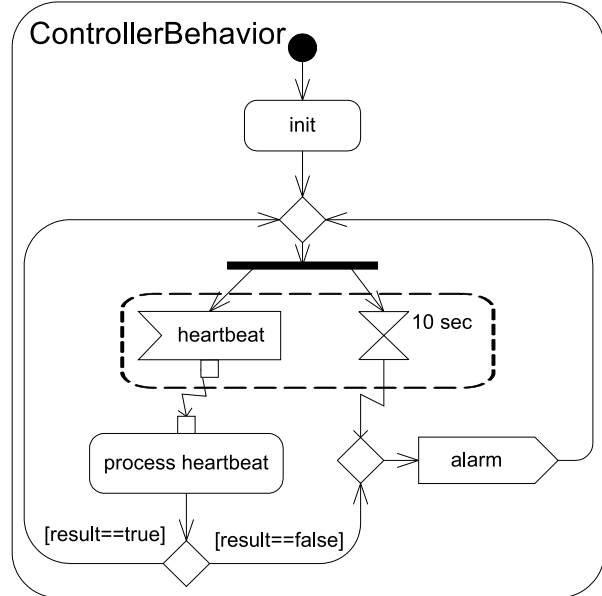


Fig. 3. Controller Behavior

a separate action called “process heartbeat”. This action has to do the following:

- 1) See which sensor sent the signal; this information can be coded as an attribute of our subclass of the UML signal class
- 2) For the sensor that has sent the signal: update a timestamp variable to the current system time
- 3) Loop through all other sensors and see, if there is one that has not notified the controller for a longer time; if there is one, indicate a failure (which in term will result in issuing an alarm, see figure 3)
- 4) If there is no such “failing” sensor, indicate success of the action

For implementation purposes, this algorithm could be further refined by another activity (see figure 5) that is called by the action. Parameters and dataflow are shown as boxes resp. arrows. Dashed lines mark special activity regions, acting as `for`- and `if`-constructs, see [2]. This would be suitable if we would follow a pure “translationist” approach of MDA (see section 4 and [1]). Although possible to read and comprehend, it takes a lot of time to create and maintain the diagram compared to normal programming language code. A direct comparison can be seen in figures 4 and 5. Obviously, this handwritten code is much more concise and readable which indicates that not every aspect of a controller should be modeled but instead coded in a traditional way.

3.2 Language Extensions

We argue for a tight integration of models and handwritten code. A compiler should take both as input to check syntax and semantics of the models and code

```

action process_heartbeat(receivedSignal : heartbeat)
{
  Sensor triggeredSensor = receivedSignal.GetSensor();
  Time tnow = System.CurrentTime();
  bool result = true;

  for(int i=1; i<=CurrentTimestamps.GetCount(); i++)
  {
    if (Sensors[i] == triggeredSensor)
      CurrentTimestamps[i] = tnow;
    else
      if (CurrentTimestamps[i] < tnow-6)
        result = false;
  }
  return result;
}

```

Fig. 4. Code for heartbeat processing

```

action init()
{
  File f = File.Open(„SensorConfiguration“);
  foreach(Record r in f)
  {
    Sensor s = new Sensor(r.Info);
    relate this to s;
    callbehavior s;
  }
}

```

Fig. 6. Code for system initialization

against each other. We introduce a simple keyword called **action** [3] which is marked bold in figure 4 to indicate handwritten code that is supposed to be an action implementation that will be called from an activity execution by our runtime component. An extended compiler can prevent direct calls to this code from other parts of the program. This is not possible with an ordinary combination of model and code where parts are realized as simple callback functions. In addition to that, the number of formal parameters and their types can be checked against the input- and output pins shown in the calling action “process heartbeat” in figure 3.

Figure 3 also shows an action called “init” for system initialization tasks. In this function we want to set up sensors according to data found in a configuration file. Also the according sensor behavior should be started. The code is shown in figure 6. In addition to the before mentioned **action**-keyword, we introduce **relate...to**, which establishes a link between two objects and **callbehavior**, that initiates a behavior (in our case starts an activity) [3]. The advantage of those constructs which are partly taken from the action semantics speci-

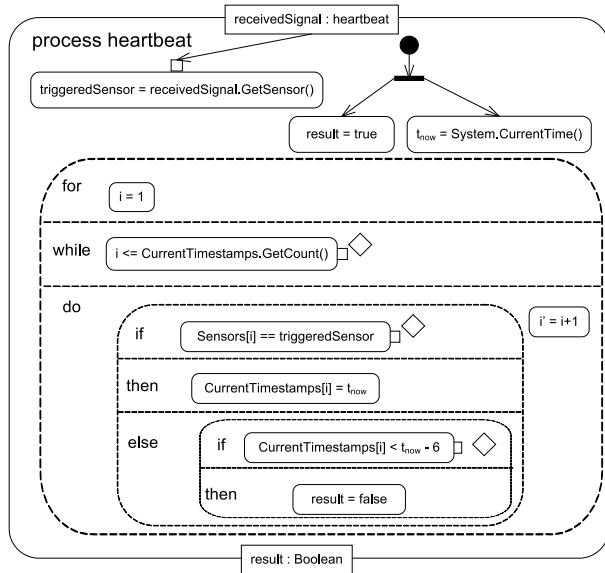


Fig. 5. Detailed activity for heartbeat processing

fication [2], is the higher abstraction level a programmer can use. One could for example imagine a translation of `relate` to a simple `myCollection.add()` or to a database SQL `INSERT` statement. These translations are specified using mapping definitions that also belong to the PSM (see figure 1). Again, the programmer is free to use ordinary programming language constructs and framework methods for the programming task.

4. Discussion and Related Work

There are two main interpretations of MDA: the elaborationist [1] and the translationist [4] approach. The chief difference of these approaches is the level where a developer can change the functionality of the system. In the elaborationist approach a developer can elaborate the PIM, the PSM, and the code. This leads to a round-trip engineering problem. In the translationist approach the developer can elaborate the PIM, which is translated directly into code. We have a PIM and a PSM, too (see figure 1 in section 2). Our approach is in between the two main interpretations, anyhow. The developer changes the functionality of the system within the PSM, but not within the code. Our approach is therefore not elaborationist. It is not translationist either, because the developer can change the functionality with our extended language within the PSM; this contradicts the ideas of the translationist approach. We avoid the round-trip engineering problem of the elaborationist approach by integrating the design artefacts into the implementation of our system, as described in section 2. The documents of the design phase are up to date and consistent with the code. The critics of the translationist approach say it is suitable only for special domains. In our approach, a variety of modeling techniques (e.g. state charts) can be

integrated by elaborating the singular actions with these techniques. We think our approach is therefore suitable for a large number of domains. The actions, enriched with pre- and postconditions, can be used as precise interface definitions of the components used by different experts with their corresponding modeling techniques. In this spirit, we define a domain specific language (DSL) for UML 2 activity and class diagrams.

As can be seen in section 3, figure 5, a too much of modeling can result in non acceptance of a modeling approach. In our approach the degree of functionality modeled and coded is at free choice (see section 2, aspect 1). This certainly benefits the acceptability of the approach. The graphical notation of the functionality is analogical to business modeling, so that non computer experts seem to understand this modeling fast. This facilitates the communication between customer and software developer. The acceptance is advanced by the potentiality of simulating the models in early analysis and design phases of the software development process. Errors within these critical initial phases can be detected early by prototyping. In addition, our approach supports extensibility and maintenance (see [5]).

The team of the DASL project [6] combines UML state charts and self created UML like charts with additional code. In the DASL project the charts can be used to generate the deployment code out of the models; a simulation is not possible. To add code to their models it is necessary to learn their Java-like language and their SQL-like query language. In our extended compiler we integrate only a few new keywords.

Another proposal is Structured Analysis (SA) [7]. However, SA is different from our method: the use of object orientation and its benefits like inheritance is not supported. Dataflow diagrams provide the basis of SA. UML activity charts make it possible to understand the workflow of an application, which is not supported by dataflow diagrams. One of the weaknesses of SA is the use of natural language, which makes formal checks of correctness and the simulation of the behavior of the modeled system impossible. Another weakness of SA is the gap between design and implementation, which we avoid as described in section 3.

C-Lab introduced a model-based, platform independent design method [8] to model the behavior of embedded systems using a combination of UML state and sequence diagrams. This enables the creation of UML models which are suitable for direct execution on a technology called UML Virtual Machine. Like the Java Virtual Machine, models should be able to be exported to other platforms. They neither use activity charts nor support additional code for single actions.

By implementing the two aspects mentioned in section 2, the models become especially important for the implementation. Therefore, the developers have to understand the semantics and modeling techniques of the UML 2

diagrams in detail. At some points the semantics of activity diagrams are ambiguous and the scope of the diagrams is too large, so that modeling becomes very difficult. Exercise and experience in modeling with the UML 2 will be necessary to work with our approach.

5. Summary and Future Work

In this paper we introduced a new approach of model-driven development: by modeling the control flow of an application with UML 2 activity charts and a tight integration of models and code with new keywords, we bridge the gap between design and implementation during software development.

We currently write an interpreter for activity charts, in which the token flow of an application can be visualized and simulated. The UML 2 metamodel for activity charts provides a basis for this interpreter, so that we needed a representation of this metamodel in code. We implemented major parts of the activity charts intermediate semantics, extended with some aspects from the complete semantics, e.g. interruptible regions. Moreover, we focus in our work on analyzing the differences of activity charts and state charts in general and their diverse fields of application. This is accompanied by the analysis of the different semantics of UML 2 state charts and activity charts.

We will evaluate our approach in a practical course in the summer term of 2005 at the University of Ulm. Our goal is to compare it with techniques from a conventional software development process. In the near future, we intend to build an integrated development environment, consisting of editors for the diagrams and the simulation tool.

References

- [1] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture — Practice and Promise*. Addison-Wesley, 2003.
- [2] “Object Management Group, UML 2.0 Superstructure Specification, Document 04-10-02,” 2004.
- [3] S. Sarstedt, J. Kohlmeyer, A. Raschke, and M. Schneiderhan, “Modellgetriebene Softwareentwicklung mit UML 2 Aktivitätsdiagrammen,” University of Ulm, Tech. Rep., 2005, to be published (in german).
- [4] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.
- [5] S. Sarstedt, J. Kohlmeyer, A. Raschke, and M. Schneiderhan, “Targeting System Evolution by Explicit Modeling of Control Flows using UML 2 Activity Charts,” in *PLC’05 - International Conference on Programming Languages and Compilers, Technical Session on Support for Unanticipated Software Evolution*, 2005, to appear.
- [6] B. Goldberg, “Practical Guide to the Ace Application Specification Languages.” Sun Microsystems, 2002.
- [7] T. DeMarco, *Structured Analysis and System Specification*. Prentice-Hall, 1979.
- [8] T. Schattkowsky and W. Müller, “Model-based design of embedded systems.” In *Proc. Design Automation and Test in Europe DATE 2005*, Munich, Germany, 2005.