Generation and Enactment of Controllers for Business Architectures using MDA

Günter Graw[1] and Peter Herrmann[2]

[1] ARGE IS KV, graw@iskv.de
[2] University of Dortmund, Peter.Herrmann@udo.edu

Abstract. Model Driven Architecture (MDA) is an initiative of the OMG in which the software development process is driven by various software-related models describing the software to be generated. Moreover, the new upcoming UML 2.0 standard promises to support the execution of models based on several types of actions as well as the inheritance of statecharts. We adapt this new technology in order to generate business controllers. By application of the popular Model View Controller (MVC) architecture, these controllers separate core business model functionality like database management from the presentation and control logic that uses this functionality (i.e., interactive user access). In particular, a controller translates user interactions realized by means of an interactive view into actions on the core business model.
This paper deals with the generation of business controllers applying MDA and UML 2.0 concepts and presents experiences gained in the background of a bigger industrial project. The focus is on statecharts and actions used for the specification and execution of controllers. In particular, in order to deal with the inheritance of statechart diagrams specified for business controllers, we define a couple of transformation rules. These rules support the transformation of abstract PIM statecharts modelling the functionality of business controllers to a flat PSM statechart describing a business controller in a more implementation-like fashion. We outline the application of the transformation rules by means of a business controller example application.

1 Introduction

The Model Driven Architecture (MDA) [5, 12] is the most recent initiative of the Object Management Group (OMG) to facilitate the creation of object-oriented software. This approach has the goal to specify software for different independent domains using abstract high level models. These high level models are specified by means of the UML (Unified Modeling Language, cf. [3]) as specification language, which is another standard adopted by the OMG. The UML models are used as input for the generation of code. MDA distinguishes two different kinds of models: platform independent models (PIM) and platform specific models (PSM). Unfortunately, a drawback of traditional UML was the absence of a sufficiently powerful semantics to specify dynamic behavior, in particular actions.

This disadvantage, however, was overcome in 2001 when the UML 1.4 [15] semantics was extended by an action specification language (ASL) [11], which has the aim to enrich the action semantics of the UML.

As if this breath-taking progress is not enough, the new UML major release for UML 2.0 is currently under standardization. Moreover, there exists a proposal for the UML superstructure [14] describing in particular the dynamic aspects of UML models. This draft contains a rich actions semantics refining the results of [11]. Moreover, the syntax and semantics of interaction diagrams were improved based on experiences of the Message Sequence Chart (MSC) community (cf. [10]).

Since actions of the ASL language are declarative by nature, due to the innovations of UML 1.4 and UML 2.0 the generation of executable models based on UML specifications is possible now. This ability is utilized by the xUML (executable UML) profile [13] which enables the execution of UML models. Meanwhile several companies created tools (e.g., bridgepoint, iCCG) which support the execution of xUML models.

Like us, Agrawal et al. [1] concentrate on generative programming of source code based on MDA. In contrast to us, however, they do not focus on behavioral aspects but on the software process of PIM to PSM transformation which is mainly performed by applying graph grammars on UML class diagrams.

This paper concentrates on the application of MDA and UML 1.4 to UML 2.0 concepts for the synthesis of controllers in business software reflecting our experiences in a bigger project of German health care insurances. A controller is used to influence the interaction of views-based graphical user interfaces (GUI) according to business rules and style guides. In particular, it is responsible for the interchange of data between a view defining a user interface and a model describing the business model. Thus, a controller is a main component of the architecture. It is possible to compose controllers of sub-controllers. The architecture of our business system is based on the well-known Model View Controller (MVC, cf. [4]) pattern which is an important architectural pattern for the fulfillment of business system requirements. In a three tier architecture typical for business applications, controllers responsible for database and workflow access are located in the middle tier which is also called application layer. The interaction between the views, which are residing in the presentation layer (i.e., the upper tier), and the controllers are realized by commands from the views to the controllers (i.e., they follow the so-called command concept).

The controller behavior is specified in a platform-independent fashion by means of UML statechart models. Statechart inheritance is used to refine the models facilitating their reuse. Moreover, we use PIM to PSM transformations of UML models to get platform-specific controller models. These PIM to PSM transformations are carried out by means of graph transformation systems (cf. [2]). Finally, the PSM models are used as input for a code generator creating executable Java code. The generated code realizes the complete state machine of the controller and most of the actions specified in the original UML state chart models. For actions which cannot be generated automatically, code fragments

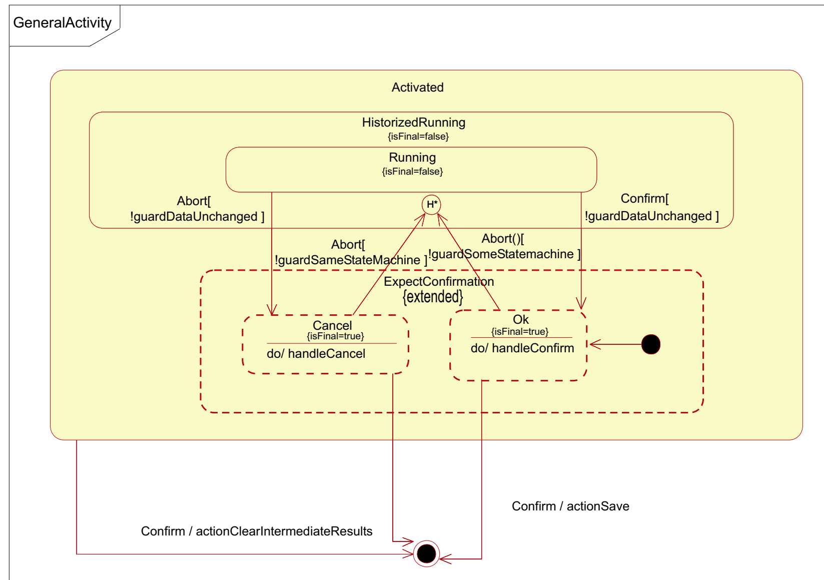

Fig. 1. State chart of the General Activity Controller

are created which, in principle, have to be filled by manually programmed code. Since the tasks realized by this code, however, are often similar, we can avoid additional programming efforts by applying reusable code libraries.

2 State Charts in UML

States can be used in the UML to define the attributes of an object and its behavior in a rather fine-grained way. Here, we apply them in a more abstract fashion in order to model the current situation of an object and its reaction on incoming events. As depicted in Fig. 1, a state description in UML (e.g., *Cancel* or *Ok*) contains an unambiguous name. Moreover, one can add action identifiers which are accompanied by the keywords *entry*, *exit*, or *do*. Based on these keywords the actions are carried out during entering, leaving, resp. remaining in the current state.

In UML, transitions between states can be provided with a statement containing an event name, a guard condition, and action identifiers. A transition is executed if the event specified, in the event name, occurs and the guard condition specified in the statement holds as well. Here, a *call* resp. *send* event is triggered if a call or send action is fired (cf. Sec. 3). In contrast, a change of an object attribute leads to a *change* event whereas a *timed event* refers to a certain real time constraint. In contrast, so-called *completion transitions* or *triggerless transitions* depending on a *completion event* are carried out without an external

trigger. A *completion event* fires if an *entry* or *do* action terminates. It is preferred against other events in order to prevent deadlocks. Furthermore, one can allow the deferral of events. If an event cannot be processed in the current state, it is stored in an event queue and can be used later. During the execution of a transition, the actions identified in the transition statement are carried out.

Similar to Harel's statechart diagrams [8], one can define so-called *composite states* composed from substates (e.g., the state *ExpectConfirmation* consisting of the substates *Cancel* and *Ok*) which can contain substates as well. A *composite state* can be a *nested state* corresponding to the OR-states in statechart diagrams. If an incoming transition of the nested state is fired, exactly one of its substates gets active.

A special class of states are pseudostates which have to be left immediately after being reached. Therefore, pseudostates must not contain *do* actions which are only executed if the state remains active for a while. Well-known pseudo nodes are *initial states*. In contrast, *termination states* are not pseudostates since an object remains in this state after reaching it. In nested states, *history states* (e.g., the state H^* in state *HistorizedRunning*) can be applied to store the lastly visited substate of a nested state. By executing an incoming transition of a *history state* the substate stored by it is reached.

To model the processing of events and, correspondingly, the selection of transitions, UML [14] defines a special state machine which is based on the run-to-completion semantics. According to this semantics, only one event may be processed at a point in time and the processing cannot be interrupted by other events. By special state configuration information the state machine describes which state resp. substates are currently active.

Statecharts in UML 2.0 can be inherited. This is reflected in the statechart diagrams by marking the states which are subject to effects of inheritance by dashed lines.

3 Actions in UML 2.0

A major improvement of UML 2.0 is the new Action Semantics defining a metamodel (cf. [14]) for action-based description languages. In contrast to traditional OCL, it facilitates the description of dynamic behavior enabling the generation of implementation code from UML models (cf. [5]). The Action Semantics does not define a particular syntax for action statements but more abstract action class definitions which can be realized by applying various different syntaxes. The standard distinguishes concrete actions from abstract metamodel action class definitions which refer to sets of similar but different action definitions. In concrete syntaxes only concrete actions may be used. Altogether, three main action classes are defined:

- Invocation-oriented actions refer to the object operation calls.
- Read- and write-oriented actions are devoted to the management of object attributes and links.

– Computation-oriented actions are used to compute output values from a set of input arguments.

The invocation-oriented actions are described by an abstract metamodel action class *InvocationAction*. Another, more specialized abstract action class is *CallAction* which is inherited from *InvocationAction*. *CallAction* describes object operations with call parameters and return values. *CallOperationAction* is a relevant concrete action inherited from *CallAction* which realizes operation calls at other objects by triggering the behavioral steps (e.g., a transition of the state machine introduced in Sec. 2) related to the operation in the called object.

Read- and write-oriented actions are distinguished in actions to maintain object attributes and in actions managing object references.

Computation-oriented actions map input arguments directly to output values. An important computation-oriented action is *ApplyFunctionAction* which encapsulates a primitive function. The action arguments are mapped to function arguments and the function result is made available at the output pins (i.e., parameters) of the action. During the computation of the primitive function the executing object is blocked and cannot interact with its environment.

4 Transformation and Generation in MDA

In the Model Driven Architecture (MDA) [12], a PIM (Platform Independent Model) is a model based specification of the structure and functionality on an abstract level neglecting technical details. In our project the PIM, which stems from the domain of health care insurance, can be used for implementations on the platforms of different insurance companies. Moreover, the validation of model correctness is supported, since a PIM supports the technology independent specification of the system. In contrast to this, the PSM (Platform Specific Model) is technology dependent with respect to a certain operating system, programming language, middleware, or application server. Source code of a particular application is generated based on the technology-depending data contained in a PSM.

Transformations are important in MDA. A transformation consists of a set of techniques and rules which are used for the modification of a model m_1 in order to obtain a model m_2. We distinguish three kinds of transformations:

– PIM to PIM transformations are used to get different representations of the same PIM.
– PIM to PSM transformations are applied to obtain the PSM representing a refinement of a specific PIM. This kind of transformation is sometimes called mapping.
– PSM to PSM transformations represent a means to get a different representation of a PSM.

Transformations may either be performed manually in iterative elaborations or be automated by means of transformation tools. Often, the automatic transformation of models is performed on the base of templates.

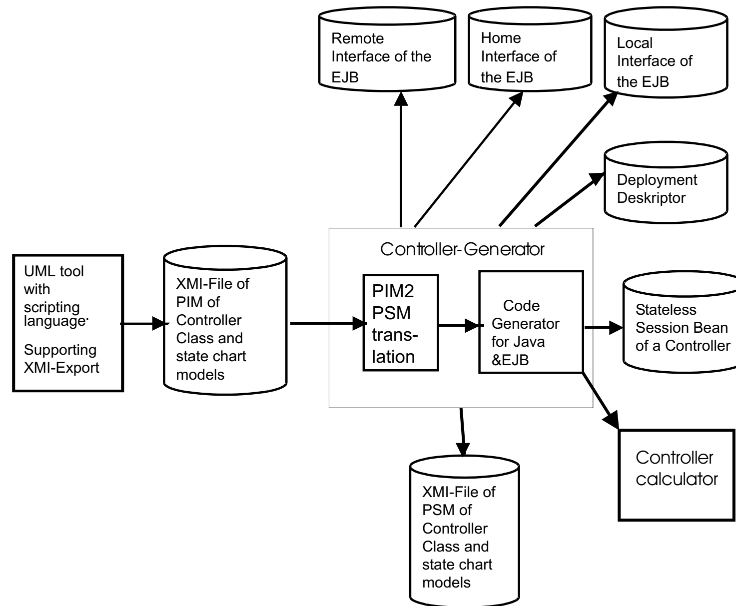

Fig. 2. Integration of Transformation and Generation Tools

In our approach, we apply a set of tools as depicted in Fig. 2 which are tightly integrated. The integration of a UML modelling tool and the controller generator is performed by means of the XML Metadata Interface (XMI). XMI is a standardized format to represent UML models (cf. [16]). We use a standard UML tool for the export of class and statechart diagrams representing the PIMs of business controllers in the XMI format. In particular, by means of UML class diagrams we define inheritance trees of business controller classes each describing a business controller with a specific functionality defined by the user requirements. Moreover, for each controller class we design a statechart diagram modelling the behavior (cf. Sec. 7).

The second tool is the controller generator consisting of several transformation and generation tools developed within the project. One of these transformation tools is used to perform PIM to PSM transformations by the application of graph rewriting rules. Since it exports the resulting PSM models as XMI files, these can be displayed by an appropriate modelling tool.

Finally, the code generator is used to generate executable Java code from the PSM. It is able to generate distributed applications based on Enterprise Java Bean (EJB) 2.0 technology (cf. [17, 18]) and creates the artifacts like interface and descriptor files. The transformation tool is tightly coupled with the Java Code generator.

Moreover, we are experimenting with tools supporting a developer in getting special views of the code generated from the PIM of a business controller. E.g., a small viewer, the so-called controller calculator is able to show all transitions on a given state of a statechart and helps to clarify the execution order of transitions and identifies potential conflicts. This supports the traceability from generated code back to the PIM which is very important for model driven development [19]. Furthermore, it gives some support in estimating the effect of a statechart change before a new XMI export and generator run is started.

5 PIM to PSM Transformation of Business Controller Statecharts

In the following, the PIM to PSM transformation based on graph rewrite rules is explained. Graph rewrite systems (cf. [2]) consist of a set of graph rewrite rules. Each rewrite rule is a tuple of two graph patterns which are called pre-pattern and post-pattern. In our approach, these rules are applied to UML state chart diagrams. A rule may be fired if a state chart diagram contains a subgraph which is an instance of the rule's pre-pattern. This subgraph is replaced by the corresponding instance of the post-pattern (i.e., instances of nodes and vertices carrying identical identifiers in the pre- and post-patterns are retained in the graph transformation). In the Figs. 3 to 7 we quote a number of graph rewrite rules. Here, the pre-patterns are listed on the left side and the post-patterns on the right.

The transformation of PIMs to PSMs is modelled by inheriting the statecharts specifying the behavior of the PIMs and PSMs. Unfortunately, before the submission of the UML 2.0 superstructure document [14] only limited information about the state chart inheritance semantics was available. For instance, the UML 1.4 specification proposed only three different policies dealing with the inheritance issue of state charts. These policies refer to subtyping, implementation inheritance, and general refinement. A more valuable source for insights with respect to statechart inheritance is proposed by Harel and Kupferman [9]. In contrast to UML 1.4, the UML 2.0 superstructure specification [14] contains clear recommendations how to deal with state chart inheritance:

> "A state machine is generalizable. A specialized state machine is an extension of the general state machine, in that regions, vertices and transitions may be added, regions and states may be redefined (extended: simple states to composite states and composite states by adding states and transitions), and transitions can be redefined."

These effects of statechart inheritance may be directly applied to PIM to PSM transformations. In particular, we distinguish the addition of new states to statecharts, the refinement of existing states, the overwriting of existing states, the addition of new transitions, and the redefinition of transitions as classes of transformation steps. For each class we defined a set of graph rewrite rules some of which are introduced below. A statechart may be extended by adding a new

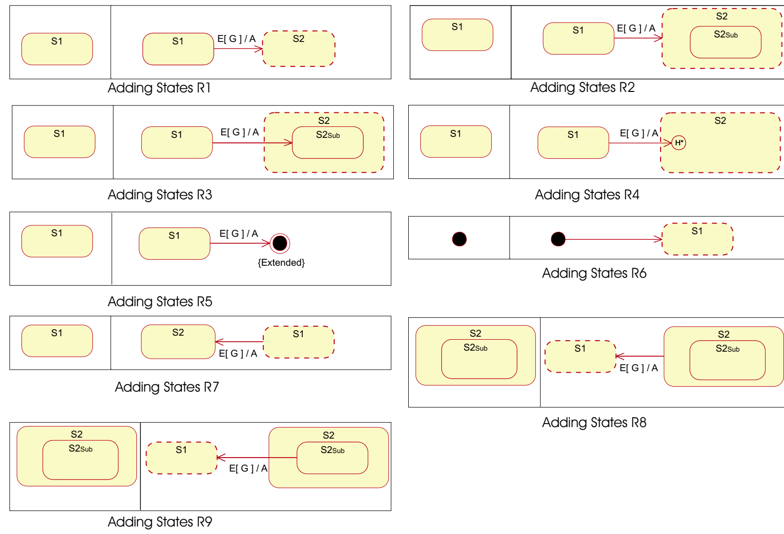

Fig. 3. Adding State Rules

state which is shown by the graph rewriting rules Adding State Rules (ASR) 1-9 depicted in Fig. 3. In order to avoid the addition of isolated states, which could never be reached in the execution, we assume that newly added states have at least one incoming or outgoing transition which is also added by executing a rule. While this restriction is not fundamental, it prevents the introduction of useless model elements which is of particular importance in complex industrial projects. The graph rewrite rule ASR1 handles the extension of a state chart adding a new simple state. ASR2 and ASR3 deal with the addition of a newly introduced nested state where the incoming transition is either connected to the nested state or to a substate of the nested state. ASR4 realizes the addition of a nested state containing a history state. By ASR5 a new final state is added whereas ASR6 handles the addition of an initial state. Due to the UML semantics, however, it is not permitted to add more than one additional initial state. The rewriting rules ASR7 to ASR9 are symmetric with respect to the rules ASR1 to ASR3 but introduce transitions using the new states as source states.

Simple states may be refined to nested states by adding new substates which is performed by the Refining State rules (RSR) 1-3 depicted in Fig. 4. The rules reflect that in PSMs the use of nested states makes only sense if they contain at least two substates. RSR1 handles the transformation of a simple state into a nesting state which contains an initial state and another connected state as substates. The rule RSR2 deals with the introduction of a nested state containing two new substates. Finally, RSR3 handles the addition of a final state into a nested state. Moreover, one can collapse an already existing nesting state consisting of only one substate together with the linked initial and final states

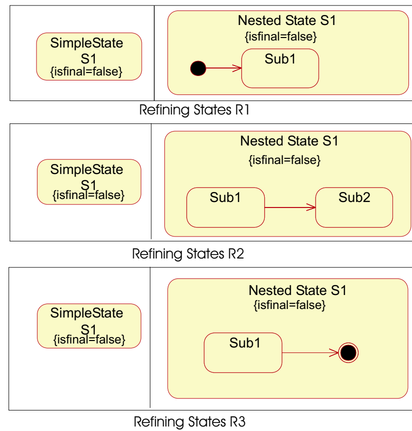

Fig. 4. Refining State Rules

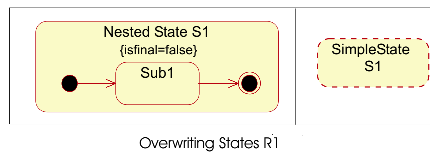

Fig. 5. Overwriting State Rule

to a simple state. This procedure is called overwriting and can be performed by application of the Overwriting State Rule 1 which is listed in Fig. 5.

Moreover, new transitions between existing states might be inserted which is realized by Adding Transition Rules (ATR) like the ATRs 1-5 of Fig. 6. The source and target states of an added transition can be of any type of states supported by our approach. If between the source and target state, however, already an existing transition exists which is fired by the same event as the new transition, the addition may only be done if the existing transition cannot be redefined. This is expressed by the tagged value *isFinal* of the existing transition which has to carry the value *true*. The rules listed in Fig. 6 describe this special case. The rule ATR1 handles the addition of a new transition between two simple states whereas ATR2 and ATR3 describe the addition of a transition in the context of nested states. In particular, ATR2 deals with the introduction of a transition on the nested state while ATR3 handles the addition of a transition on a substate. ATR4 realizes the addition of a transition to a history state of a nested state and ATR5 performs the addition of a transition to a final state. The rules listed in Fig. 6 describe the special case that a non-redefinable transition between the source and target states with an identical event already exists. Similar rules for unconnected source and target states and for existing transition with different events are also available. If a source and a target node are linked by a redefinable transition, one can apply the Redefining Transition

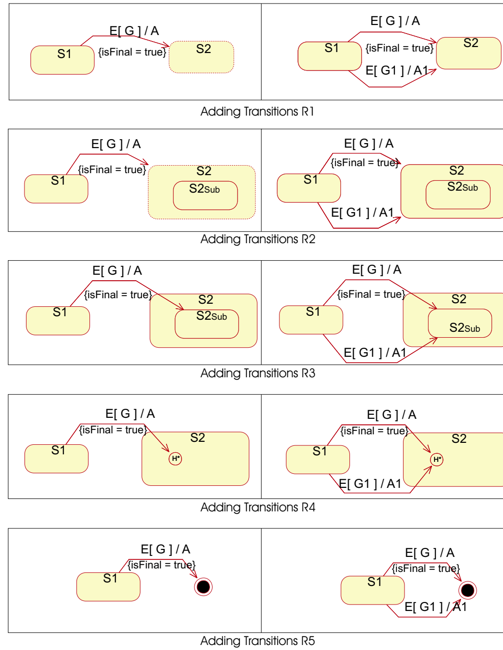

Fig. 6. Adding Transition Rules

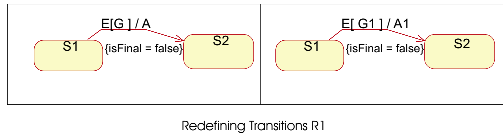

Fig. 7. Redefining Transition Rule

Rule (RTR) 1 which is depicted in Fig. 7. The rule may be only executed if the tagged value *isFinal* is set to *false*. By its application the guard and the action of the transition are altered.

Our tool applies the rules in the following order: At first, RSRs are executed followed by OSRs. Thereafter, the ASRs and ATRs are fired. The graph rewrite rules-based transformation terminates with the application of RTRs. In order to transform a controller PIM to the corresponding PSM, at first the PSMs of its superclasses have to be created by rule applications. Based on these transformation results the tool transforms the state chart of the controller class. The rules are programmed in Java implementing an algorithm visiting the nodes and transitions of a design model parsed from the XMI representation while performing the PIM to PSM transformation.

If a large number of new transitions is added by the controller model transformations, nondeterminism of the transitions in the resulting model may increase.

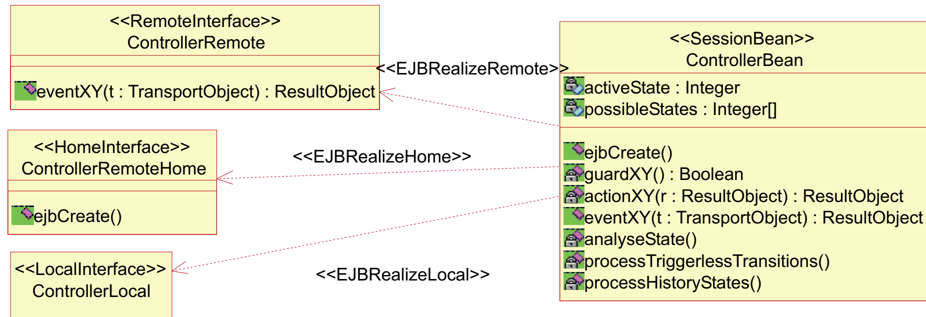

Fig. 8. Structure of a Controller Bean and the Interfaces

To support the code generator in resolving this nondeterminism, every transition is supplied by a weight factor indicating the depth of the controller class containing the transition in the class inheritance hierarchy. The weight is a natural number, which is as higher as deeper the controller of the statechart is positioned in the inheritance hierarchy. The weight is made persistent in a tagged value *weight* of the according transition. This value will be used by the code generator to create a useful transition order in the generated stateful session bean realizing the controller. Here, transitions with a higher weight will be prioritized. Moreover, the code generator prefers transitions with guards to transitions without guards. Triggerless transitions without guards have the lowest priority.

6 PIM to PSM Transformation of Business Controller Classes

In the following, we focus on the structural aspects of the PIM to PSM transformation of a business controller. A business controller is realized by a stateful session bean. Session beans are EJB components providing client access to a business system. In stateful session beans, moreover, the current state of a conversation is maintained whereas stateless session beans have no capability to persist states. A stateful session bean, representing the controller of a PSM, includes the code of a state machine interpreting the statechart of the according PSM Class. The class diagram depicted in figure 8 shows the structure of the PSM of a controller. In table 1 the rules for the creation of a controller PSM are presented which we will explain in the following. Firstly, attributes of a PIM Controller class become member variables of the Controller PSM. Here, we use an algorithm to create the flat representation of all states of all controller states as enumeration type. Moreover, an attribute is introduced to the controller PSM keeping the active state of the statechart which is an element of the enumeration type (Rule *PSM_R1*).

In our architecure, UML models of Java GUI widgets are used to specify the mapping of selected Java Swing events to logical events of the controller

Rule Nr.	Rule description	Target
PSM_R1	Creation of enumeration type for states of a controller bean	Controller PSM
PSM_R2	Creation of operations for events of the according state machine of a controller bean	Controller PSM
PSM_R3$_i$	Set of rules for the creation of transport objects transmitted by an event	Controller PSM
PSM_R4	Creation of operations for actions of a controller bean	Controller PSM
PSM_R5	Creation of operations for guards of a controller bean	Controller PSM
PSM_R6$_i$	Set of rules for the creation and of the *analyseState* operation of a state machine	Controller PSM
PSM_R7$_i$	Set of rules for the creation of the *processTriggerlessTransitions* operation of a state machine	Controller PSM
PSM_R8$_i$	Set of rules for the creation of the *processHistoryStates* operation of a state machine	Controller PSM
PSM_R9	Creation of a remote interface of a controller session bean	Controller PSM
PSM_R10	Creation of a local interface of a controller session bean	Controller PSM
PSM_R11	Creation of the home interface of a controller session bean	Controller PSM
PSM_R12	Creation of an event operation in the remote interface	Controller PSM
PSM_R13	Creation of an operation *ejbCreate* in the home interface	Controller PSM
PSM_R14	Creation of an operation to start the execution of a sub-controller in the local interface	Controller PSM

Table 1. Controller PIM To PSM Transformation Rules

statechart. This mapping is specified by tagged values. Event operations which are called from the views of the GUI of a controller are used to transport values in so called transport objects to the state machine. Vice versa, new data values are sent back to the views as well as result objects containing error states and new values in order to refresh of GUI information. According event operations are introduced to the PSM (Rule *PSM_R2*). The PSMs for transport objects are created by a different generator which is not subject to this paper (Rule *PSM_R3*$_i$).

Each action of the controller's statechart is transformed into an operation with an argument and return type which are both of type ResultObject (Rule *PSM_R4*). A ResultObject is used to retransmit resulting object values to a View of the GUI. For the guards of the statechart parameterless operations with return type boolean are generated (Rule *PSM_R5*). In the project we have the convention that operations for guards and actions are already modelled in the class of a controller PIM. Although this is not necessary, it helps to keep track of what is going on.

Moreover, some operations to realize the correct behavior of controller state machines are required. The method *analyseState* is used to enact entry and exit actions as well as to start doActivities of a state (Rule *PSM_R6*$_i$). Furthermore,

this operation calls the operation *handleTriggerlessTransition* which is responsible to select triggerless transitions. This selection is based on the result of guard evaluation of the transitions enabled in the active state of the controller state machine as well as the order defined for conflicting transitions (Rule PSM_R7_i). The operation *processHistoryStates* which is also called the method *analyseState* which is responsible for the handling of history states (Rule PSM_R8_i).

The PIM to PSM transformation creates also the home, remote, and local interfaces for the session bean of a controller as shown in figure 8 (Rules *PSM_R9*, *PSM_R10*, and *PSM_R11*). Every event operation is added to the remote interface (Rule *PSM_R12*). An *ejbCreate* operation is added to the home interface (Rule *PSM_R13*). Operations dealing with the enaction of sub-controllers, which are modelled as operations of a controller class are added to the local Interface (Rule *PSM_R14*). Moreover, a doActivity is modelled in a state of the statechart of a controller PIM to compose a controller and a sub-controller. Since a controller requires information about sub-controller termination each sub-controller provides termination information to its controller. To handle different cases of sub-controller termination special transitions are added to the PIM of a controller.

The java code generator generates method frames or complete methods for each operation of the PIM. While, generally, a frame has to be filled by manually created programming code, we can often take reusable code which is inherited from the superclasses of the controller hierarchy applying the inheritance properties of session beans. For read- and write-oriented actions (cf. Sec. 3), the Java method is completely generated whereas for computation-oriented actions only a method frame is created. The PSM operations of guards cause the generation of an according method frame. In contrast to this, the methods for event operations are completely generated since the PIM to PSM transformation of a controller statechart provides all of the required information. This is also applicable to the methods for the operations *analyseState*, *processTriggerless* and *processHistoryStates* of a controller PSM which are completely generated.

A part of the code generator, the so-called *interface generator*, generates the home, local, and remote interfaces which provide the local and remote access to methods of the stateful session bean realizing a controller. Finally, the *descriptor generator* is used to generate the deployment descriptor of a controller bean describing the interfaces of the bean and their behavior. In particular, the deployment descriptor contains the bean's name and type, the names of the home, remote and local interfaces. Moreover, the transaction type of the bean and the transaction attribute of the methods of an interface are declared. A transaction attribute is generated as *required* by default.

7 Business Controller Example

In this section examples of business controller syntheses used in an enterprise application are presented. Fig. 9 depicts the class diagram showing the inheritance hierarchy of business controllers. The abstract class *BaseController* is the

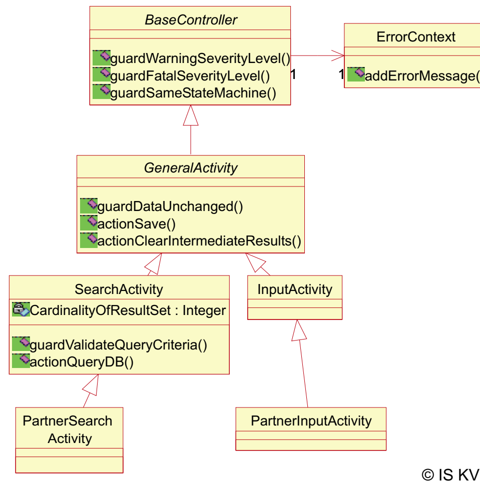

Fig. 9. Class diagram of the business controller inheritance hierarchy

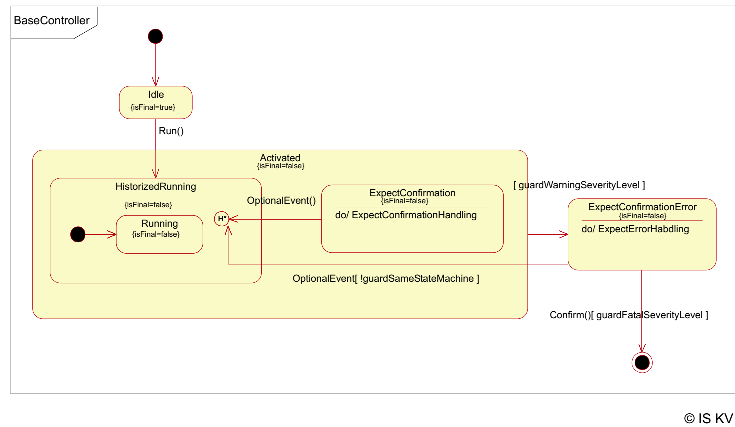

Fig. 10. Statechart of the *Base Controller*

root of the class diagram. It is responsible for the error and exception handling as well as for the management of dialogue confirmations. The statechart diagram of *BaseController* outlined in Fig. 10 contains elementary states which are provided for derived business controllers. The controller which is initially in the state *idle* is activated by the event *Run()* and proceeds to the state *Activated*. In particular, the substate *HistorizedRunning* is reached. In a later statechart inheritance, *HistorizedRunning* is refined by the introduction of additional substates in order to store the history of the controller behavior. This is reflected by the history state in *HistorizedRunning*. Moreover, *HistorizedRunning* contains the substate *Running* which is a place holder for later refinements describing the system functionality. Another substate of *Activated* is the state *ExpectConfirma-*

tion which will carry confirmations of dialogue events in refined states. Finally, the statechart diagram contains the state *ExpectConfirmationError* modelling error handling. It is reached by a transition accompanied by the guard *guardWarningSeverityLevel*. Thus, the transition fires only if the error exception is at least of the severity level *Warning*. According to the severity level the controller may either return to the last active state of historized running or abort due to a fatal error. This is expressed by the transition guards *guardSameStateMachine* resp. *GuardFatalSeverityLevel*. The guards are implemented by the three boolean operations which in the class diagram are contained in the operation compartment of class *BaseController*. This class is associated with the class *ErrorContext* receiving all error messages which result from actions triggered by the business controller.

An inheritance of *BaseController* is the abstract controller class *GeneralActivity* which is mainly responsible for the handling of confirmations initiated by a user in a session. The associated statechart diagram of this controller class was already presented in Figure 1. With respect to *BaseController* the substate *ExpectConfirmation* was refined by adding the two substates *Cancel* and *Ok*. These substates contain do-actions *handleCancel* resp. *handleConfirm* each starting a sub-controller which deals with cancellation or confirmation performed by the user. Moreover, four transitions were added in order to handle logical abort and confirm events of a dialogue. The transitions from the state *Running* to the added states which rely on an *Abort* resp. *Confirm* event refer to the guard *guardDataUnchanged* checking if business data changed since carrying out the last transition. Two other transitions connecting the new states to *Running* are carried out after receiving an *Abort* event if the guard *guardSameStateMachine* holds.

The concrete class *SearchActivity* models a controller which is able to perform a search in a database. The class has an attribute *CardinalityOfResultSet* of type integer holding the number of result entries of a database query. The operation *guardValidateQueryCriteria* is required to check if a query on a database uses the required criteria of the query correctly. Moreover, there is an operation *actionQueryDB* which is an *ApplyFunctionAction* (cf. Sec. 3) performing a query on a database. In the corresponding statechart of *SearchActivity* the state *ValidateSearchResult* and transitions handling the query in a database are added.

In a further refinement of *SearchController* (e.g., the class *PartnerSearchActivity*), controllers with respect to specific business requirements are modelled. Here, the operation *guardValidateQueryCriteria* is overwritten by an operation referring to a concrete database application while *actionQueryDB* is refined by an operation containing a concrete database query.

Below, we sketch the PIM to PSM transformation of the statechart of the controller class *SearchActivity*. The resulting PSM of *SearchActivity* is shown in Fig. 11. Moreover, the depicted statechart models the states and transitions of the PIM. We present the states and transitions added to the statechart of the *GeneralActivity* class (cf. Fig. 1) by thick lines. For transformating the PIM to

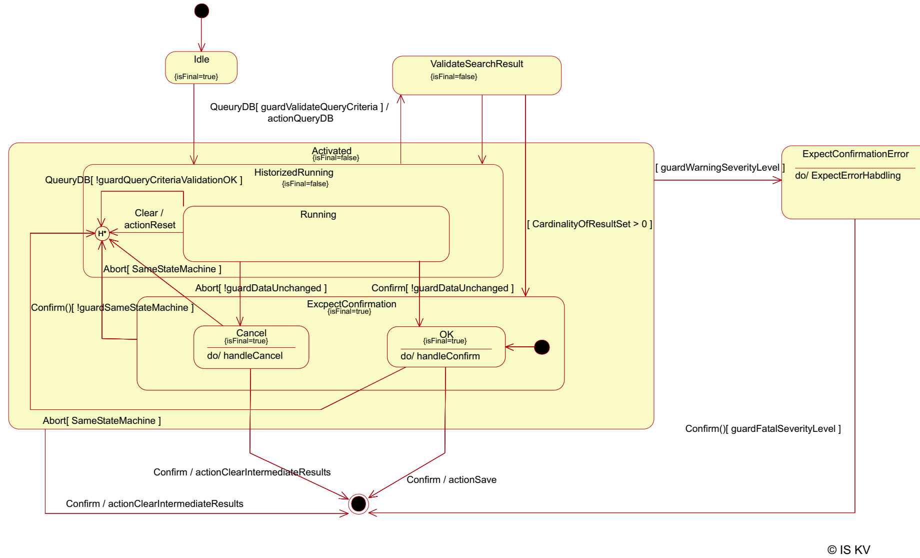

Fig. 11. State chart of the Search Activity PSM

the PSM of *SearchActivity*, we had first to transform the PIM of the superclasses. While for the class *BaseController* no transformations had to be performed, we applied the listed rules to the statechart of the superclass *GeneralActivity* in the following order:

1. Refine nested state *expectConfirmation* by adding the substate *Cancel* which has the action *handleCancel* and the transition from *Running* to *Cancel* (RSR2).
2. Add the substate *Ok* to the nested state *expectConfirmation* which has the action *handleConfimation* and a transition from *Running* to *Ok* (ASR1).
3. Add the initial substate to the nested state *expectConfirmation* with a transition to *Ok* (ASR6).
4. Add a transition from state *Cancel* to the final state (ATR5 variant).
5. Add a transition from state *Ok* to the history state H^* (ATR4 variant).
6. Add a transition from the nested state *Activated* to the final state with the event *Confirm* and the action *actionClearIntermediateResults* (ATR5 variant).

In a second step, we transformed the statechart specifying the *SearchActivity* PIM by application of the following rules:

1. Add state *ValidateSearchResult* with a transition from state *HistorizedRunning* (ASR1).
2. Add a transition from state *Running* to the history state H^* with the event *Clear* and the association action *actionClear* (ATR 4 variant).

3. Add a transition from state *Running* to the history state H^* with the event *QueryDB* and the guard *guardQueryCriteriaValidationOK* (ATR 4 variant).
4. Add a transition from state *ExpectConfirmation* to the history state H^* with the event *QueryDB* and the guard *guardQueryCriteriaValidationOK* (ATR 4 variant).
5. Add a triggerless transition from *ValidateSearchResult* to state *HistorizedRunning* (ATR3 variant).
6. Add a triggerless transition from *ValidateSearchResult* to state *ExpectConfirmation* (ATR3 variant).

All in all, the transformation tool performed the rule application automatically and needed about 125 ms.

8 Concluding Remarks

In this paper we pointed out that new features of UML 2.0 like statechart inheritance and the action semantics extensions are useful means to model business controllers. Moreover, we focussed on MDA-based generation of business controller code. In particular, we use graph rewrite rules for PIM to PSM transformation. The effort to create the transformation tool and the code generator for controllers was about 2 man years and the amount to build the framework of business controllers and the PIM to PSM transformation was about 3 man months. The result is a fast and useful system which reduces the efforts of business controller design drastically. We applied our approach in a first industrial project for health care insurances. Our generated metrics showed that 90 % of the necessary application code could be generated by the application of transformation and generation tools. Moreover, about 60 % of the framework code for the controllers were automatically generated. Performance measurements show that generated controller code is performant. The maintenance of the code is comparatively easy since a lot of information is available in models which makes its complexity more manageable. In spite of the generally exponential complexity of graph isomorphisms, the PIM to PSM transformations of the project could be performed within 400 ms for a standard sized controller due to hard coding of the transformation rules.

In the future, we have to increase the flexibility of rule adoptions which are of interest not only for business controllers. In particular, tools and languages for a suitable generation and application of the transformation rules are of interest in agile software development processes. In complete, we plan to spend several man years of application development using the framework and the transformation tools. Moreover, since the bidirectional traceability from models to code is very fundamental for model driven development appropriate tools are required. We will provide special tools to prepare information concerning traceability in special views, which allow a quick access to the necessary information without using a debugger for Java sources. At third, we will explore the possibility to use the model-based approach for formal-based software development (cf. [6, 7]).

References

1. A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, and G. Karsai. Generative Programming via Graph Transformations in the Model-Driven Architecture. In *OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, Nov. 2002. Available via WWW: www.softmetaware.com/oopsla2002/mda-workshop.html.
2. R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In *Handbook on Graph Grammars and Computing by Graph Transformation, Volume 2*. World Scientific, 1999.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, Chichester, 1996.
5. D. S. Frankel. *Model Driven Architecture : Applying MDA to Enterprise Computing*. Wiley Europe, Jan. 2003.
6. G. Graw and P. Herrmann. Verification of xUML Specifications in the Context of MDA. In J. Bezivin and R. France, editors, *Workshop in Software Model Engineering (WISMEUML'2002)*, Dresden, 2002.
7. G. Graw, P. Herrmann, and H. Krumm. Verification of UML-Based Real-Time System Designs by Means of cTLA. In *Proc. 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC2K)*, pages 86–95, Newport Beach, 2000. IEEE Computer Society Press.
8. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
9. D. Harel and O. Kupferman. On the Behavioral Inheritance of State-Based Objects. In *Proc. Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, pages 83–94, Santa Barbara, 2000. IEEE Computer Society Press.
10. Ø. Haugen. MSC-2000 Interaction Diagrams for the New Millennium. *Computer Networks*, 35(6):721–732, 2001.
11. Kennedy Carter. *Action Semantics for the UML*. Available via WWW: www.kc.com/as_site/home.html.
12. A. Kleppe, W. Bast, and J. Warmer. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.
13. S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.
14. OMG. *UML: Superstructure v. 2.0 – Third revised UML 2.0 Superstructure Proposal*, OMG Doc# ad/03-04-01 edition, 2003. Available via WWW: www.u2-partners.org/uml2-proposals.htm.
15. Open Management Group. *UML Metamodel 1.4*. Available via WWW: www.omg.org/uml.
16. P. Stevens. Small-Scale XMI Programming: A Revolution in UML Tool Use? *Journal of Automated Software Engineering*, 10(1):7–21, Jan. 2003.
17. Sun Microsystems. *Enterprise Java Beans Technology — Server Component Model for the Java Platform (White Paper)*, 1998. java.sun.com/products/ejb/white_paper.html.
18. Ed Roman. *Mastering EJB*, 2002. Available via WWW: www.theserverside.com.
19. Krzysztof Czarnecki, Simon Helsen *Classification of Model Transformation Approaches*, 2003. *Proceedings of the OOPSLA 03 MDA Workshop*. Available via WWW: http://www.softmetaware.com/oopsla2003/czarnecki.pdf.