Extending an ORDBMS: The StateMachine Module

Wolfgang Mahnke[2] Christian Mathis[2] Hans-Peter Steiert[1]

[1]DaimlerChrysler AG
Research & Technology
P.O.Box 2360
89013 Ulm
Germany
hans-peter.steiert@daimlerchrysler.com

[2]University of Kaiserslautern
P.O.Box 3049
67653 Kaiserslautern
Germany
{mahnke|mathis}@informatik.uni-kl.de

Abstract

Extensibility is one of the mayor benefits of object-relational database management systems. We have used this system property to implement a StateMachine Module inside an object-relational database management system. The module allows the checking of dynamic integrity constraints as well as the execution of active behavior specified with the UML. Our approach demonstrates that extensibility can effectively be applied to integrate such dynamic aspects specified with UML statecharts into an object-relational database management system.

1 Motivation

Object-relational database management systems (ORDBMS) enable us to extend their off-the-shelf functionality by self-developed extension modules [6]. This allows us to adopt an ORDBMS to the needs of several application domains. Our demonstration illustrates how this feature can be exploited to equip an ORDBMS with a module which handles dynamic aspects inside the database system.

In our SERUM project [3], we use an object-oriented specification as an input to generate data storage components. Since UML [4] offers a rich set of object-oriented constructs and is widely used, we have chosen it as specification language. Class diagrams are used to describe the data structures, and the mapping of these structures to an object-relational schema is straightforward. Each class of the class diagram is mapped to a row type and a typed table in the database schema. Inheritance relationships can be mapped onto table inheritance (see [2] for details and problems with multiple inheritance). Methods declared in the class diagram are implemented as user-defined routines (UDRs) in the ORDBMS, and associations can be established in several ways, e.g., by using foreign keys. These mappings do not only apply to the SERUM context, but are a general approach when implementing an object-relational schema on the basis of an object-oriented specification.

In addition, the capabilities of ORDBMSs allow to consider dynamic modelling aspects, too. UML statechart diagrams are an appropriate way to specify these aspects. In the context of ORDBMS-based data storage components, statecharts can be used for two purposes [5]:

- Observing dynamic integrity constraints
  In contrast to static integrity constraints describing the correctness of a database state, dynamic integrity constraints restrict the transitions between different states. For this purpose, valid sequences of operations, pre- and post-conditions for transitions, and state-dependent invariants can be specified. Whenever such a constraint is being violated, the conflicting operation has to be prevented.
- Implementing active behavior
  Statecharts can be used to specify functionality in form of actions which have to be executed when the triggering event occurs. There are predefined events in an ORDBMS, e.g., INSERT, UPDATE, or DELETE events on tables, which can be used by SQL triggers. Furthermore, application-specific events can be defined and triggered by UDR calls or, in turn, by the actions of the statecharts.

UML statecharts specify dynamic aspects of a class or an operation. We restrict our demonstration to statecharts de-

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

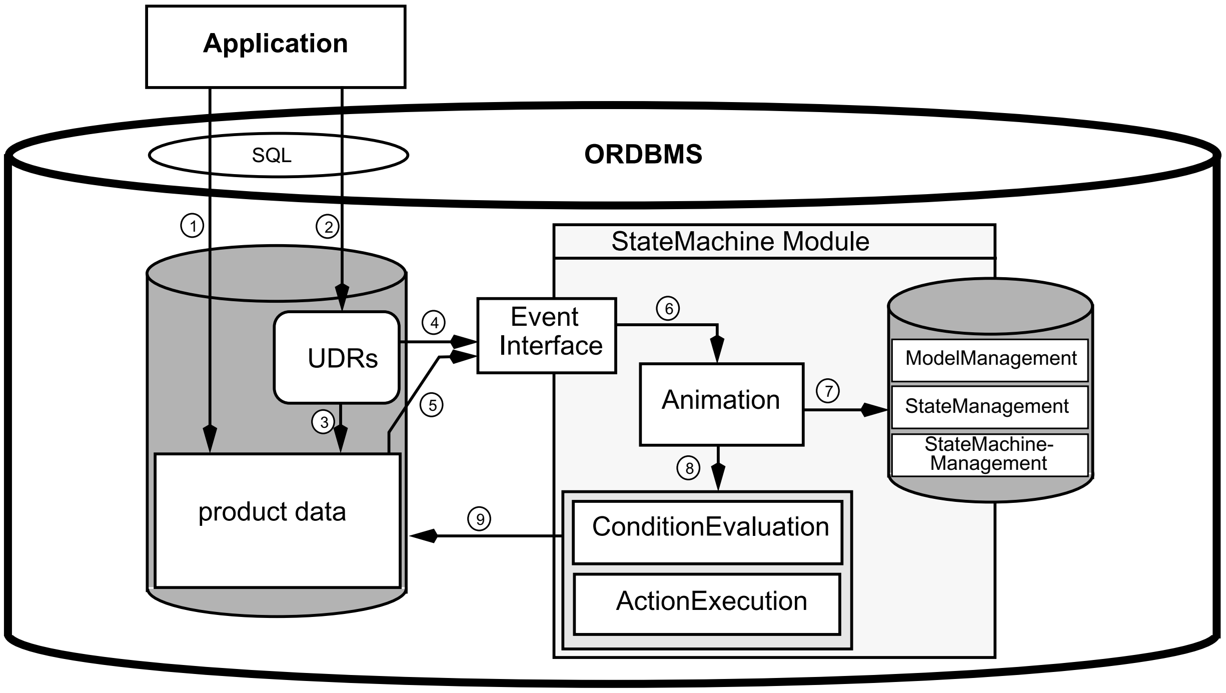

Figure 1. Architecture of the StateMachine Module

fined on classes and do not deal with the dynamic aspects related to the operations.

In the following, a short overview of the architecture of our StateMachine Module is given and two example statecharts are introduced, one specifying dynamic integrity constraints, and the other specifying active behavior. In our demonstration, we present both examples and illustrate the internal processes of the StateMachine Module.

2 Architecture of the StateMachine Module

Fig. 1 illustrates the architecture of the StateMachine Module. The product data and their related UDRs are shown at the lefthand side. The specification, expressed as a UML model, is stored as part of the StateMachine Module database tables, denoted as ModelManagement in Fig. 1. Applications can access the product data directly (1) or via UDRs (2,3). Both ways, calling a UDR or manipulating the product data directly, can trigger an event, which is send to the Event Interface of the StateMachine Module (4,5). To create these events we integrate specific event calls in the UDRs and use SQL triggers for INSERT, UPDATE, and DELETE operations on the typed tables of the product data. In the SERUM project, this integration is straightforward, since the typed tables and the UDRs are generated. In other environments, additional efforts are required to assure that each event is called correctly.

When an event arrives at the Event Interface, it is passed on to the Animation component (6). Depending on its type and the current state (stored in the StateManagement component), it may be processed directly or delayed. By storing it in an event queue of the StateMachineManagement component, it can be processed later on. Whenever initiating a state machine, the StateMachineManagement uses a subscription mechanism to keep information for the events to be expected. Since each state machine responding to the event received has to be animated, the Animation reads its specification and its current state (7). While performing the animation, conditions have to be evaluated and actions have to be executed (8), both steps referring to the product data (9). The manipulation of the product data, in turn, can trigger new events to be handled. As soon as the animation process for the state machine under consideration has been finished, its new state is saved by the StateManagement.

As ORDBMS we used the IBM Informix Internet Foundation.2000 [1] to implement the StateMachine Module. Most functions of the module are coded as Java UDRs. Object-relational concepts like type hierarchies, overloading and polymorphism helped managing the different event types occurring in the UML-based specification.

3 Observing Dynamic Integrity Constraints

We introduce a short example to demonstrate the use of the StateMachine Module to observe dynamic integrity constraints. In the following, we restrict ourselves to a single class, called `Aircraft`. An excerpt of the corresponding statechart diagram is illustrated in Fig. 2. A flight of an aircraft starts in the `PrepareToTakeOff` state. In this

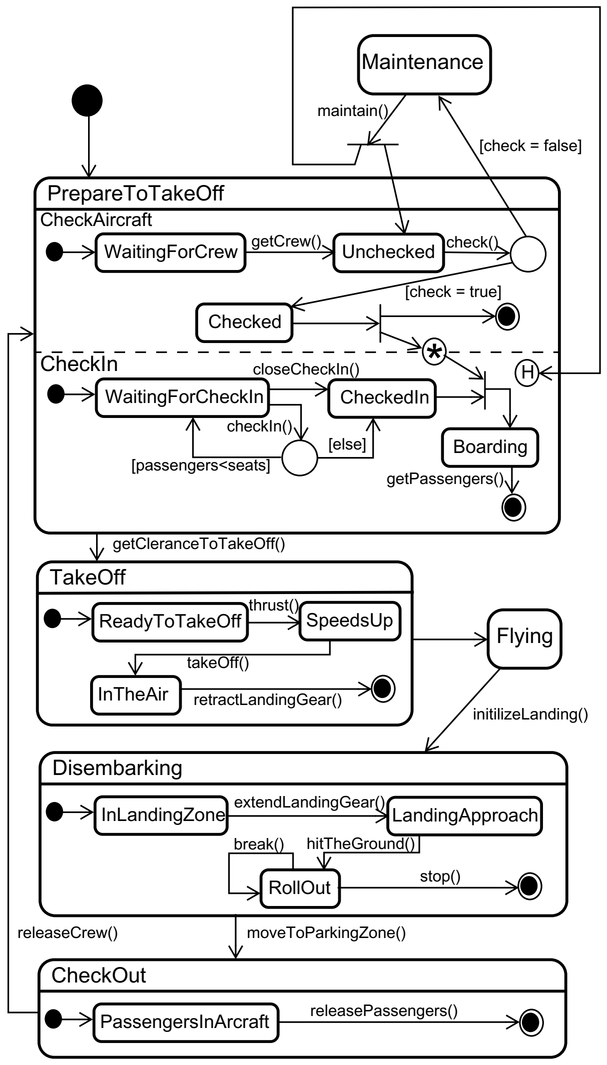

Figure 2. Statechart of an Aircraft

state, the aircraft can be checked, while the passengers can check in, represented by parallel substates. The boarding can only start after the aircraft is successfully checked. Therefore, we use a sync-state between the parallel substates. As soon as the boarding is completed, the aircraft can take off. Upon arrival at the destination, the aircraft can disembark and the passengers can check out. By releasing the crew, the aircraft comes again into the `PrepareToTakeOff` state.

Most transitions of the example are triggered by function calls. Hence, the statechart describes in which state a specific function can be called. An error occurs if a function is called while the aircraft is in a state where the call is not allowed. For example, the `retractLandingGear()` function can only be called when the aircraft is in the air.

There are two possibilities for handling errors. One is to define an error state, and whenever an unexpected event occurs, the error state is entered. The other is to raise an exception which leads to a rollback of the operation. Because we do not want an aircraft entering the error state, we use the second one in our example.

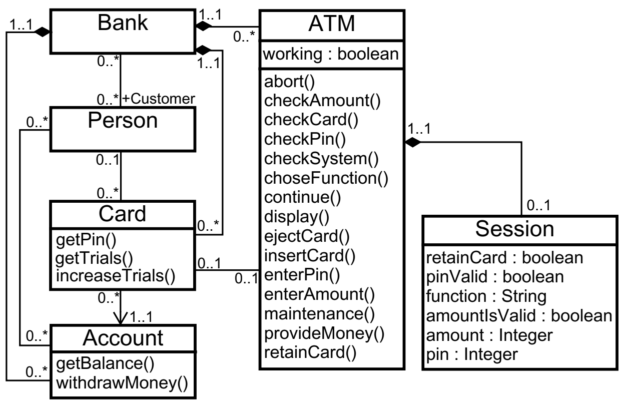

Figure 3. Class Diagram of Example Scenario

4 Implementing Functionality

Our second example deals with the implementation of active behavior. We use a statechart diagram to implement the behavior of an ATM (Automated Teller Machine). In this scenario, the class diagram involves more than just one class, as illustrated in Fig. 3. Each ATM manages a session object, representing the current session, and belongs to a bank. The bank manages accounts and has customers. These customers can use their bank card to access the ATM.

Fig. 4 illustrates the statechart diagram. Conditions are defined on most transitions. The actions of the transitions define the intended behavior of the ATM. For example, if the ATM is in the state `WaitingForCard`, it expects the function call `insertingCard(card)`. As soon as the function is called, the transition is initiated by triggering an event and the action `checkCard(card)` is executed. This function sets `retainCard` of the current session. If `retainCard` is false, the ATM displays `′Your account is locked′` and changes to the `RetainingCard` state. Afterwards it displays `′Your card is retained′`, calls `retainCard(card)`, and moves into the `SystemCheck` state. If `retainCard` is true, the ATM displays `′Please insert PIN′` and moves into the state `WaitingForPin`.

This example shows that statechart diagrams can easily be used to implement active behavior of database objects.

5 Presentation Details of the Demonstration

Since the StateMachine Module runs completely inside the ORDBMS, it is hard to visualize its internal processes.

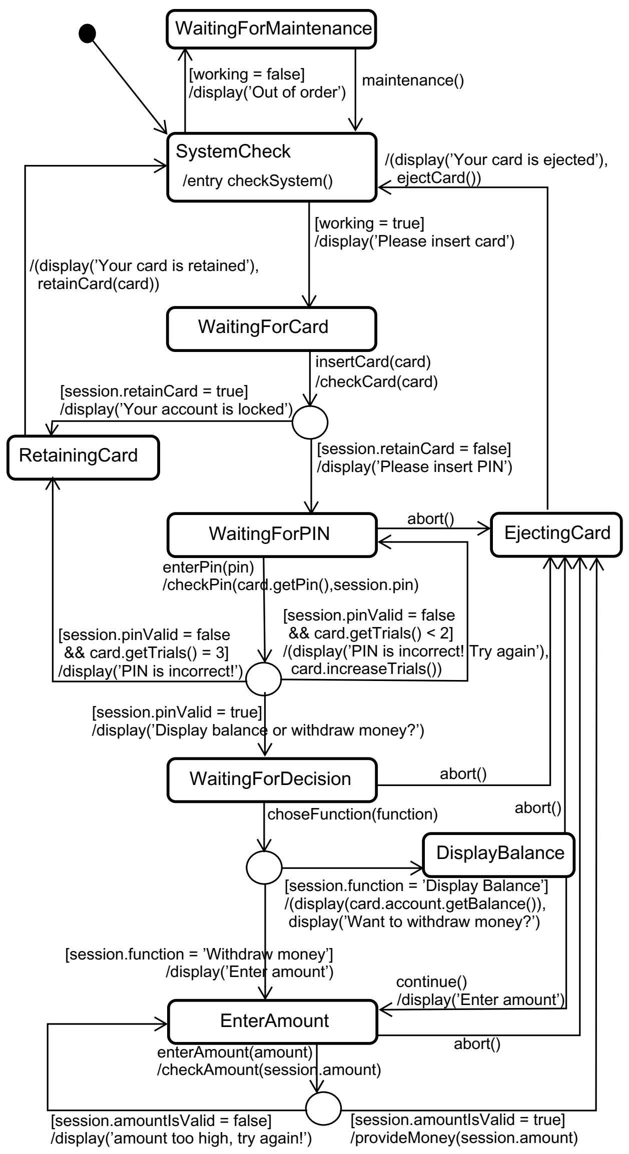
WaitingForMaintenance
[working = false]
/display('Out of order')
maintenance()
SystemCheck
/entry checkSystem()
/(display('Your card is ejected'),
ejectCard())
[working = true]
/display('Please insert card')
/(display('Your card is retained'),
retainCard(card))
WaitingForCard
insertCard(card)
/checkCard(card)
[session.retainCard = true]
/display('Your account is locked')
RetainingCard
[session.retainCard = false]
/display('Please insert PIN')
WaitingForPIN
abort()
EjectingCard
enterPin(pin)
/checkPin(card.getPin(),session.pin)
[session.pinValid = false
&& card.getTrials() < 2]
/(display('PIN is incorrect! Try again'),
card.increaseTrials())
[session.pinValid = false
&& card.getTrials() = 3]
/display('PIN is incorrect!')
[session.pinValid = true]
/display('Display balance or withdraw money?')
WaitingForDecision
abort()
abort()
choseFunction(function)
DisplayBalance
[session.function = 'Display Balance']
/(display(card.account.getBalance()),
display('Want to withdraw money?')
[session.function = 'Withdraw money']
/display('Enter amount')
continue()
/display('Enter amount')
EnterAmount
abort()
enterAmount(amount)
/checkAmount(session.amount)
[session.amountIsValid = false]
/display('amount too high, try again!')
[session.amountIsValid = true]
/provideMoney(session.amount)

Figure 4. Statechart of an ATM

Therefore, we have implemented a Java application running outside the ORDBMS, which performs two tasks:

- First, it serves as a front-end for manipulating product data, i.e., inserting new objects or calling UDRs. Thereby, it creates new events for the StateMachine Module.
- Second, it visualizes the internal processes running inside the StateMachine Module. For this purpose, it observes one object of the product data. The current attribute values of the object and an animated statechart diagram, where the current state is highlighted, are displayed. Whenever an event for this object occurs, the application visualizes the state transitions of the statemachine instance.

  To get the information about the animation, it reads the data from the log file written by the StateMachine Module.

In our demonstration, we will present the aircraft scenario to prove that the StateMachine Module can reasonably be used to observe dynamic integrity constraints. We will also demonstrate the ATM scenario in order to prove the usability of the StateMachine Module to effectively implement functionality inside the ORDBMS by specifiing a statemachine.

6 Conclusions

Our StateMachine Module allows to control dynamic aspects inside an ORDBMS. The introduced examples, presented as UML statechart diagrams, illustrate different possibilities for the use of dynamic modelling aspects: dynamic integrity constraints and active behavior of database objects.

Nevertheless, there are some difficulties related to the integration of dynamic features into the ORDBMS. In an environment using a *generated* database schema, it is easy to create the required events for the StateMachine Module by appropriate generator actions. In other environments, this may turn out to be a problem, because the programmer may fail to provide them. What we have not considered so far, are performance restrictions. Specifying complex dynamic integrity constraints may lead to a high run-time burden for the constraint checking. Because this functionality is integrated in the StateMachine Module, the query optimizer can not relieve this problem.

Despite these problems, our demonstration confirms that an ORDBMS can be extended successfully to manage dynamic aspects of database objects.

7 Literature

[1] IBM: Getting started with the IBM Informix Dynamic Server. Version 9.30. August 2001.

[2] Mahnke, W., Steiert, H.-P.: The potential of ORDBMS in design environments (in German). GI-Fachtagung CAD 2000 - Kommunikation, Kooperation, Koordination, Berlin, March 2000.

[3] Mahnke, W., Ritter, N., Steiert, H.-P.: Towards Generating Object-Relational Software Engineering Repositories. 8. GI-Fachtagung „Datenbanken in Büro, Technik und Wissenschaft“ (BTW'99), Freiburg, March 1999.

[4] OMG: OMG UML v. 1.3, OMG Document ad/99-06-08.

[5] Steiert, H.-P.: Aspects of the generic design of ORDBMS-based data storage components (in German). Dissertation, Department of Computer Science, University of Kaiserslautern, September 2001.

[6] Stonebraker, M., Brown, M.: Object-Relational DBMSs: Tracking the next great Wave. Morgan Kaufmann, 1999.