Activity-Driven Synthesis of State Machines*

Rolf Hennicker and Alexander Knapp

Ludwig-Maximilians-Universität München
{hennicker,knapp}@ifi.lmu.de

Abstract. The synthesis of object behaviour from scenarios is a well-known and important issue in the transition from system analysis to system design. We describe a model transformation procedure from UML 2.0 interactions into UML 2.0 state machines that focuses, in contrast to existing approaches, on standard synchronous operation calls where the sender of a message waits until the receiver object has executed the requested operation possibly returning a result. The key aspect of our approach is to distinguish between active and inactive phases of an object participating in an interaction. This allows us to generate well-structured state machines separating "stable" states, where an object is ready to react to an incoming message, and "activity" states which model the computational behaviour of an object upon receipt of an operation call. The translation procedure is formalised, in accordance with the UML 2.0 meta-model, by means of an abstract syntax for scenarios which are first translated into I/O-automata as an appropriate intermediate format. Apparent non-determinism in the automata gives rise to feedback on scenario deficiencies and to suggestions on scenario refinements. Finally, for each object of interest the corresponding I/O-automaton is translated into a UML 2.0 state machine representing stable states by simple states and activity states by submachine states which provide algorithmic descriptions of operations. Thus the resulting state machines can be easily transformed into code by applying well-known implementation techniques.

1 Introduction

Scenario-based approaches describe system behaviour in terms of typical interactions between several objects participating, for instance, in a single use case. Scenarios are particularly useful in the analysis phase since they focus on the overall collaboration of objects to perform a particular task. However, scenarios do not show the complete behaviour of a single object which is left to the design phase where the objects' lifecycles can be described by state machines.

We propose a rigorous method to transform a set of scenarios, represented by UML 2.0 sequence diagrams, into state machines. Our general assumption is that each scenario is simple in the sense that it focuses only on one interaction sequence at a time. Hence, we will deliberately not consider more expressive notations for sequence diagrams (like, e.g., alternatives) which add computational complexity at the cost of clarity; cf. the discussion on the specification of conditional behaviour by Fowler [1].

* This research has been partially supported by the GLOWA-Danube project (01LW0303A) sponsored by the German Federal Ministry of Education and Research.

Additional behaviour can be shown in separate sequence diagrams for (secondary) scenarios. The task then is to transform the set of sequence diagrams into a set of state machines, each showing the complete behaviour of a single object across the scenarios.

There are many approaches in the literature suggesting various strategies and solutions for state machine synthesis and analysis, like, e.g., [2,3,4,5]; see [6] for an overview. These approaches deal with asynchronous communication in the sense that the sender of a message is immediately ready for further activation and, in contrast to synchronous communication, does not wait until the receiver has executed its reaction to the incoming operation call. We claim that, as a consequence, the resulting state machines do not provide an adequate design model if we consider standard applications with synchronous operation calls and returns. The goal of this paper is to provide a synthesis algorithm that takes into account synchronous calls and their corresponding execution traces such that the resulting state machines can be easily transformed into a standard implementation with a single thread of control.

Our method is centred around the treatment of object activations that occur (in a sequence diagram) when an object has received an incoming message. During an activation an object may send messages to other objects, wait for corresponding results and finally provide a return value. We hence focus on reactive system objects where inactive phases, in which an object is waiting for an incoming message, and active phases, in which an object reacts to an incoming message, alternate. Inactive phases are considered as "stable" states. They may be given a name which will be used for matching different scenarios when generating state machines. During the translation different activations represented in different scenarios but caused by the same incoming message (after the same stable state) will be integrated into a single activity of an object. Activities can be considered as procedures in the sense of "Executable UML" [7]. They are modelled by UML 2.0 submachines with one entry point and, in general, several exit points representing different possible results (inferred from the different scenarios).[1] The overall state machine representing the life-cycle of an object is then obtained by integrating stable states and "activity" states (represented by submachine states). The generated state machines exhibit a general pattern with alternating stable and activity states. Any outgoing transition from a stable state leads to the entry point of an activity state and is labelled by an incoming message; upon completion of an activity a transition is fired which connects an exit point of the activity state with the next stable state.

Technically, our transformation sets out from a set of UML 2.0 interactions which are formalised, in accordance with the UML 2.0 meta-model, in terms of an appropriate abstract syntax for scenarios (see Sect. 2). Taking these scenarios as input, our synthesis procedure consists of the following four steps which are iteratively performed for each (system) object *o*.

1. *Projection of scenarios:* For each scenario the communications in which the object *o* under consideration participates are extracted. Such projections are usually computed in state machine synthesis approaches.
2. *Generation of behaviours from projected scenarios:* Each projected scenario is transformed into an equivalent but differently structured representation, called

[1] We do not use UML 2.0 activity diagrams to model activities because, in contrast to UML 1.x, the activity diagrams of UML 2.0 are not specialisations of state machine diagrams.

behaviour. A behaviour groups, for each incoming message, all subsequent outgoing messages sent by *o* into one activation capturing "the reaction" of object *o* for each synchronous operation call according to a particular scenario (see Sect. 3).

3. *Integration of behaviours into I/O-automata:* After the first two steps there is still one behaviour of system object *o* for each scenario. The different behaviours across all scenarios are now integrated on the basis of common (stable) states. The result is represented by an I/O-automaton with the incoming messages as input and the corresponding object activations (together with a return) as output (see Sect. 4.1). The non-deterministic transitions of the I/O-automaton serve as a basis for generating suggestions for scenario changes which are, in particular, directed towards behavioural completion (see Sect. 4.2).
4. *Translation of I/O-automata into UML 2.0 state machines:* Finally, the generated I/O-automaton is transformed into a UML 2.0 state machine with stable states and activity states. The activity states integrate possible different reactions to the same incoming message in a particular stable state (which may still have been present in an I/O-automaton) into one single activity (see Sect. 5). It is worth to note that, according to the abstract nature of the I/O-automata, translations into different concrete target representations are possible (e.g., generating instead of activity states procedural expressions of some action language [7]).

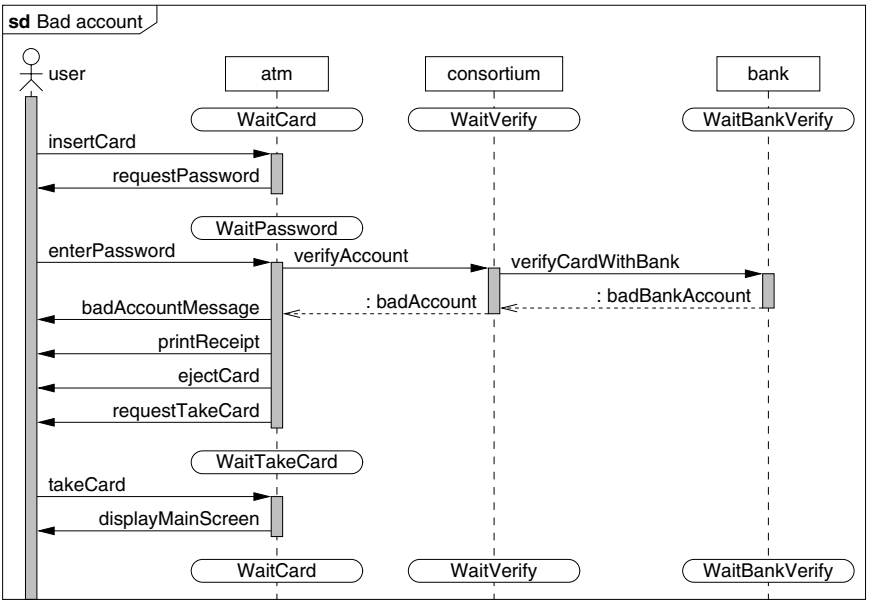
For ease of comparison of our synthesis procedure with the approaches from the literature (see Sect. 6) we base our description on a widely used automatic teller machine (ATM) example [2,5,8].

2 Scenarios

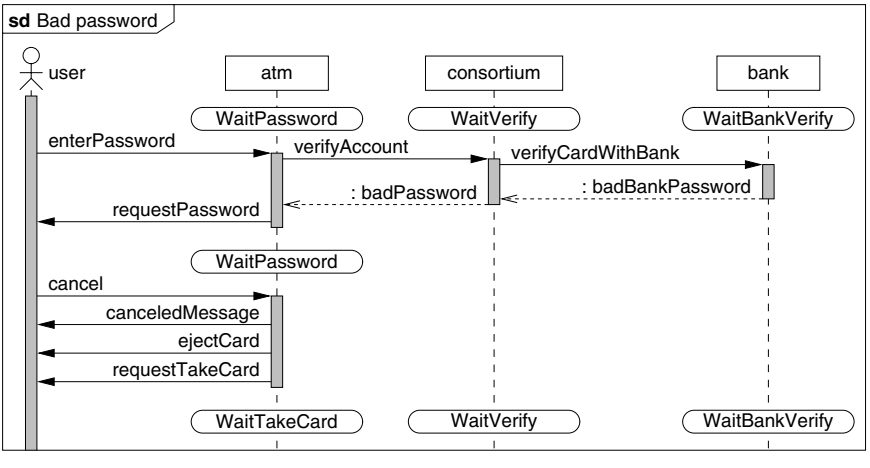
We introduce the sequence diagram language for describing scenarios by the well-known ATM example [2,5,8]. In the following we consider the case where an atm object reacts to a user who has inserted a card by validating the card with the help of the objects consortium and bank. The UML 2.0 sequence diagrams in Fig. 1(a) and Fig. 1(b) detail two possible scenarios which have been formulated in [2,5] with the difference, as pointed out in Sect. 1, that we consider here messages as synchronous operation calls which may provide return values.[2]

A scenario describes a sequence of communications between scenario participants. For scenario participants we distinguish between *user actors* (headed by stick figures) and *system objects* (depicted by boxes). A communication consists of a synchronous *operation call* (shown above a solid line with filled arrow head) and a *return* message with a *value* (shown above a dashed arrow with open arrow head). An operation call on a system object causes an *activation* (grey vertical rectangle) of the system object. Before and after an activation a system object is in a certain *state* which can be left implicit or be named explicitly (shown in a rounded rectangle).

[2] Note that both scenarios can be considered as "secondary" scenarios since they describe variations of the normal behaviour described by a primary scenario which is not considered here but could easily be included.



(a) Scenario: Bad account



(b) Scenario: Bad password

Fig. 1. ATM example

The abstract syntax of our scenario language, which conforms to a subset of UML 2.0 interactions [9], is rendered in the following BNF grammar where we assume the domains *SystemObject* of system objects, *User* of user actors, *State* of states, *Operation* of operations, and *Value* of typed values.

$$\begin{aligned} \mathit{Scenario} &::= \mathit{Communication}^* \\ \mathit{Communication} &::= \mathit{UserCommunication} \mid \mathit{SystemCommunication} \end{aligned}$$

$$
\begin{aligned}
\textit{UserCommunication} &::= \textit{UserMessage Return} \\
\textit{UserMessage} &::= \textit{Object Operation User} \\
\textit{SystemCommunication} &::= \textit{State SystemMessage Return State} \\
\textit{SystemMessage} &::= \textit{Object Operation SystemObject} \\
\textit{Return} &::= \textit{Value} \mid \mathsf{void} \\
\textit{Object} &::= \textit{SystemObject} \mid \textit{User}
\end{aligned}
$$

In a sequence of communications, a user communication represents a message from a sending object to a receiving user actor together with its return; and a system communication a message from a sending object to a receiving system object again with its return. The first (pre-)state in a system communication represents the state of the receiving system object before actually receiving the message, the second (post-)state the state after having sent the return to the incoming message. We require that the post-state of a system communication equals the pre-state of the next system communication with the same receiving system object, which disallows spontaneous state changes on reactive system objects. The reaction of a system object o to an incoming message, i.e., its subsequent activation, is given implicitly by the sequence of all communications with o as sender before the next incoming message to o arrives. Hence, we do not consider nested activations caused by call-backs. Finally, we require that all returns are type correct in the sense that for a given operation either all return messages have no return value (represented by void) or all return messages have a return value of the same type.

Table 1. Tabular representation of the scenarios

(a) Scenario: Bad account

Pre state (rcv.)	*Sender*	*Operation*	*Receiver*	*Return*	*Post state (rcv.)*
WaitCard	user	insertCard	atm	void	WaitPassword
–	atm	requestPassword	user	void	–
WaitPassword	user	enterPassword	atm	void	WaitTakeCard
WaitVerify	atm	verifyAccount	consortium	badAccount	WaitVerify
WaitBankVerify	consortium	verifyCardWithBank	bank	badBankAccount	WaitBankVerify
–	atm	badAccountMessage	user	void	–
–	atm	printReceipt	user	void	–
–	atm	ejectCard	user	void	–
–	atm	requestTakeCard	user	void	–
WaitTakeCard	user	takeCard	atm	void	WaitCard
–	atm	displayMainScreen	user	void	–

(b) Scenario: Bad password

Pre state (rcv.)	*Sender*	*Operation*	*Receiver*	*Return*	*Post state (rcv.)*
WaitPassword	user	enterPassword	atm	void	WaitPassword
WaitVerify	atm	verifyAccount	consortium	badPassword	WaitVerify
WaitBankVerify	consortium	verifyCardWithBank	bank	badBankPassword	WaitBankVerify
–	atm	requestPassword	user	void	–
WaitPassword	user	cancel	atm	void	WaitTakeCard
–	atm	canceledMessage	user	void	–
–	atm	ejectCard	user	void	–
–	atm	requestTakeCard	user	void	–

Using a tabular notation, similar to the one suggested in the UML 2.0 superstructure specification [9, App. E], the sequence diagrams for the scenarios of the ATM example are represented by the two sequences of communications shown in Tab. 1. For missing returns void has been filled in. In our example, all states have user-defined names, but in general this is not necessary and states which were left implicit in the graphical representation of the sequence diagrams would be considered to be pairwise different and would be equipped with different artificial names. The symbol "$-$" is used in user communications where no states are needed for the user actor.

For deriving the behaviour of a given system object across many scenarios, we assume an ordering on the given set of scenarios such that the pre-state of the first communication of the system object in a successive scenario is already present as a state in one of its predecessor scenarios. For instance, the pre-state WaitPassword of atm in the second scenario Bad password occurs in the first scenario Bad account.

The scenario language differs from the MSC-based languages used in [5], [3], or [2] by three main concepts: the distinction between user actors and system objects, the use of activations and the use of return values. On the other hand, as discussed in Sect. 1, we deliberately do not include more complex constructs for interaction composition.

3 Generating Behaviours from Scenarios

For the synthesis of state machines from scenarios we focus (iteratively) on a single system object for which the different scenarios have to be integrated. In a first step, similarly to all other synthesis algorithms, a projection operation discards communications in a scenario that are not relevant for the system object o under consideration. More formally, given a system object o and a scenario S, the *projection* of S to o is defined as the scenario $proj(S, o)$ which consists of all those communications of S where o is either the sending or the receiving object. In our running example, the projection $proj(\text{Bad account}, \text{atm})$ to the system object atm yields the sequence of communications in Tab. 1(a) with the fifth line removed, and the projection $proj(\text{Bad password}, \text{atm})$ to atm yields the sequence of communications in Tab. 1(b) with the third line removed. If we focus on the system object consortium the projections $proj(\text{Bad account}, \text{consortium})$ and $proj(\text{Bad password}, \text{consortium})$ yield the communications shown in Tab. 2.

In the second step we transform for each system object each single projected scenario into an equivalent but differently structured representation, called *behaviour*, where for

Table 2. Projection of the scenarios to consortium

(a) Scenario: Bad account

Pre state (rcv.)	*Sender*	*Operation*	*Receiver*	*Return*	*Post state (rcv.)*
WaitVerify	atm	verifyAccount	consortium	badAccount	WaitVerify
WaitBankVerify	consortium	verifyCardWithBank	bank	badBankAccount	WaitBankVerify

(b) Scenario: Bad password

Pre state (rcv.)	*Sender*	*Operation*	*Receiver*	*Return*	*Post state (rcv.)*
WaitVerify	atm	verifyAccount	consortium	badPassword	WaitVerify
WaitBankVerify	consortium	verifyCardWithBank	bank	badBankPassword	WaitBankVerify

each incoming message (received in some state) the subsequent outgoing messages and the final return are grouped into one activation. After an activation has finished the object (possibly) changes its state. Thus, each (projected) scenario can be transformed into a block structure where each block consists of a pre-state, an incoming message, the corresponding activation and a post-state. The following grammar captures this intuition:

$$
\begin{aligned}
\mathit{Behaviour} &::= \mathit{Block}^* \\
\mathit{Block} &::= \mathit{State\ InMessage\ Activation\ State} \\
\mathit{Activation} &::= \mathit{OutMessage}^*\ \mathit{Return} \\
\mathit{InMessage} &::= \mathit{Operation} \\
\mathit{OutMessage} &::= \mathit{Operation\ Object\ Return}
\end{aligned}
$$

From a scenario $S \in \mathit{Scenario}$ and a system object $o \in \mathit{SystemObject}$, the operation $beh(S, o)$ computes the *behaviour* of o in S, by first projecting S to o and then collecting all messages sent by o between two subsequent messages received by o:

$$
\begin{aligned}
&beh : \mathit{Scenario} \times \mathit{SystemObject} \to \mathit{Behaviour} \\
&beh(S, o) = act(proj(S, o), o) \\
&\\
&act : \mathit{Scenario} \times \mathit{SystemObject} \to \mathit{Behaviour} \\
&act(\varepsilon, o) = \varepsilon \\
&act(\langle \mathit{pre}, \langle \mathit{snd}, \mathit{op}, o\rangle, \mathit{ret}, \mathit{post}\rangle\ cs, o) = \langle \mathit{pre}, \mathit{op}, \langle \mathit{outs}, \mathit{ret}\rangle, \mathit{post}\rangle\ \ act(\mathit{rest}, o) \\
&\qquad \text{where } (\mathit{outs}, \mathit{rest}) = collect(cs, o) \\
&\\
&collect : \mathit{Communication}^* \times \mathit{SystemObject} \to \mathit{OutMessage}^* \times \mathit{Communication}^* \\
&collect(\varepsilon, o) = (\varepsilon, \varepsilon) \\
&collect(\langle \mathit{pre}, \langle \mathit{snd}, \mathit{op}, \mathit{rcv}\rangle, \mathit{ret}, \mathit{post}\rangle\ cs, o) = \\
&\qquad \begin{cases} (\varepsilon, \langle \mathit{pre}, \langle \mathit{snd}, \mathit{op}, \mathit{rcv}\rangle, \mathit{ret}, \mathit{post}\rangle\ cs) & \text{if } \mathit{rcv} = o \\ (\langle \mathit{op}, \mathit{rcv}, \mathit{ret}\rangle\ \mathit{outs}, \mathit{rest}) & \text{if } \mathit{snd} = o \\ \qquad \text{where } (\mathit{outs}, \mathit{rest}) = collect(cs, o) & \end{cases}
\end{aligned}
$$

where $\langle \mathit{pre}, \langle \mathit{snd}, \mathit{op}, \mathit{rcv}\rangle, \mathit{ret}, \mathit{post}\rangle \in \mathit{Communication}$ and $cs \in \mathit{Communication}^*$; ε denotes the empty sequence; sequence composition is denoted by juxtaposition; and angle brackets compound syntax fragments. The behaviours of atm and consortium for our running example scenarios Bad account and Bad password are given in Tab. 3.

Note that the construction of activations, based upon the function $collect$, marks a significant methodological and technical difference to the approaches in [5], [2], and [3], which has a crucial impact on the construction of UML 2.0 state machines described below.

4 Integrating Behaviours into I/O-Automata

The behaviours constructed in the last section are still split according to the original set of scenarios. The goal of the next step is to integrate for each system object its computed set of behaviours on the basis of shared states; these shared states must have been determined already by the modeller of the scenarios by giving them the same names. For the integration of behaviours we use as an intermediate format

Table 3. Behaviours for atm and consortium in each scenario

(a) Behaviour of atm in scenario Bad account

Pre state	*Message in*	*Messages out*	*Return*	*Post state*
WaitCard	insertCard	⟨requestPassword, user, void⟩	void	WaitPassword
WaitPassword	enterPassword	⟨verifyAccount, consortium, badAccount⟩		
		⟨badAccountMessage, user, void⟩		
		⟨printReceipt, user, void⟩		
		⟨ejectCard, user, void⟩		
		⟨requestTakeCard, user, void⟩	void	WaitTakeCard
WaitTakeCard	takeCard	⟨displayMainScreen, user, void⟩	void	WaitCard

(b) Behaviour of atm in scenario Bad password

Pre state	*Message in*	*Messages out*	*Return*	*Post state*
WaitPassword	enterPassword	⟨verifyAccount, consortium, badPassword⟩		
		⟨requestPassword, user, void⟩	void	WaitPassword
WaitPassword	cancel	⟨canceledMessage, user, void⟩		
		⟨ejectCard, user, void⟩		
		⟨requestTakeCard, user, void⟩	void	WaitTakeCard

(c) Behaviour of consortium in scenario Bad account

Pre state	*Message in*	*Messages out*	*Return*	*Post state*
WaitVerify	verifyAccount	⟨verifyCardWithBank, bank, badBankAccount⟩	badAccount	WaitVerify

(d) Behaviour of consortium in scenario Bad password

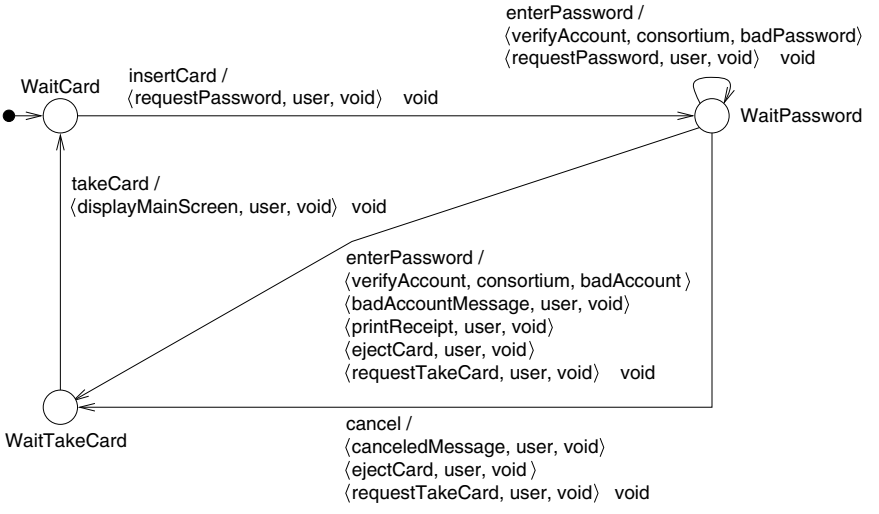
Pre state	*Message in*	*Messages out*	*Return*	*Post state*
WaitVerify	verifyAccount	⟨verifyCardWithBank, bank, badBankPassword⟩	badPassword	WaitVerify

I/O-(input-/output-)automata before we finally construct concrete state machines. To use I/O-automata has several advantages: First, the integration process can be defined in terms of standard techniques for joining I/O-automata. Moreover, I/O-automata provide an abstract representation which is appropriate for feedback on problems in the integration process which can either be resolved by human manipulation of the scenarios or by choosing a default integration strategy. Finally, the intermediate representation paves the way for transforming scenario models into different concrete notations, like, in our case, UML 2.0 state machines or the "Executable UML" [7] or LTSA [5].

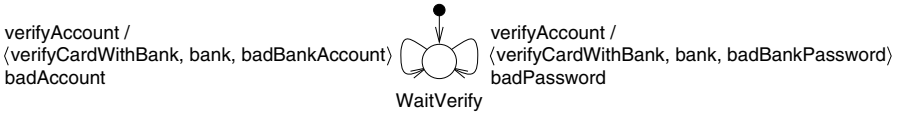
4.1 I/O-Automata

Formally, an *I/O-automaton* is a quadruple (Z, In, Out, δ) with Z its *states*, *In* the *input alphabet*, *Out* the *output alphabet*, and $\delta \subseteq Z \times In \times Out \times Z$ the *transition relation*. An *I/O-automaton with initial state* is a quintuple $(Z, In, Out, \delta, z_0)$ where (Z, In, Out, δ) is an I/O-automaton and $z_0 \in Z$ is the *initial state*.

Each single behaviour of a system object constructed in Sect. 3 can be seen as an I/O-automaton where the states in a behaviour are directly taken as the states of the automaton, the input messages as the input to the automaton, the activations, i.e., the sequences of output messages with final returns, as the output of the automaton, and each block as a transition. Given a scenario S and a system object o, the function $io(S, o)$ constructs



(a) Integrated I/O-automaton of atm



(b) Integrated I/O-automaton of consortium

Fig. 2. Integrated I/O-automata for atm and consortium

an I/O-automaton $(Z, \mathit{In}, \mathit{Out}, \delta)$ for the behaviour of o in S as follows: Z is given by the set of states in $beh(S, o)$; *In* is given by the set of in-messages in $beh(S, o)$; *Out* is given by the activations in $beh(S, o)$, i.e., the pairs of sequences of out-messages and returns; and δ is defined by requiring $(\mathit{pre}, \mathit{in}, (\mathit{outs}, \mathit{ret}), \mathit{post}) \in \delta$ iff $\langle \mathit{pre}, \mathit{in}, \langle \mathit{outs}, \mathit{ret} \rangle, \mathit{post} \rangle$ is a block in $beh(S, o)$.

The integration $intio(S_0, \{S_1, \ldots, S_n\}, o)$ of a given scenario S_0 with further scenarios $S_1, \ldots, S_n$ with respect to a system object o is now simply the (joined) I/O-automaton with initial state $(Z_0 \cup \cdots \cup Z_n, \mathit{In}_0 \cup \cdots \cup \mathit{In}_n, \mathit{Out}_0 \cup \cdots \cup \mathit{Out}_n, \delta_0 \cup \cdots \cup \delta_n, z_0)$ with $io(S_i, o) = (Z_i, \mathit{In}_i, \mathit{Out}_i, \delta_i)$ and z_0 the pre-state of the first block in $beh(S_0, o)$.

Figure 2(a) shows the integrated I/O-automaton of the atm object (with initial state) for the scenarios of our running example. Similarly, Figure 2(b) shows the integrated I/O-automaton of the consortium object.

4.2 Feedback

The integration of scenarios into a single I/O-automaton with initial state will, in general, result in a non-deterministic automaton. On the one hand, non-determinism reflects under-specification and thus is intentional. On the other hand, non-determinism can also be a symptom for incompleteness or errors in the original scenarios. Indeed, we would

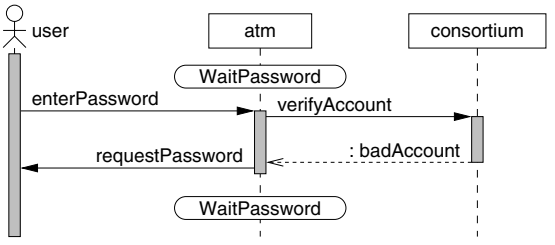


Fig. 3. Non-deterministic scenario Bad password (fragment)

expect different reaction sequences to an incoming message to be justified by different source states or by different returns in the sequence of outgoing messages of the reaction, and the user of the synthesis procedure will be warned about the possible error.

Suppose that for some source state and an input in the integrated I/O-automaton two sequences of outgoing messages of the following form appear:

$$m_1 \dots m_{k-1} \ \langle op_k, rcv_k, ret_k \rangle \ \langle op_{k+1}, rcv_{k+1}, ret_{k+1} \rangle \ m_{k+2} \dots c_l$$
$$m_1 \dots m_{k-1} \ \langle op_k, rcv_k, ret'_k \rangle \ \langle op'_{k+1}, rcv'_{k+1}, ret'_{k+1} \rangle \ m'_{k+2} \dots c'_n$$

If $ret_k = ret'_k$, but $op_{k+1} \neq op'_{k+1}$ or $rcv'_{k+1} \neq rcv'_{k+1}$ the user will be informed that ret_k and ret'_k should be different, in order to ensure deterministic behaviour. As a simple example, consider the scenario fragment in Fig. 3, which modifies the scenario Bad password of Fig. 1(b) by using as a return for verifyAccount the same value badAccount that has been used in the scenario Bad account. Then, the different continuations between the first and the (changed) second scenario indicate non-determinism which should be resolved by the user.

Similarly, suppose that some source state *pre* and an input *in* is followed by two sequences of outgoing messages, subsequent returns, and successor states of the following form:

$$\langle pre, in, \langle m_1 \dots m_n, ret \rangle, post \rangle$$
$$\langle pre, in, \langle m_1 \dots m_n, ret' \rangle, post' \rangle$$

If $ret \neq ret'$, the user will be warned that either *ret* should be the same as ret' or the activation sequences before should be different, because after exactly the same activation (for the same in-message and the same pre-state) there is no obvious reason to provide different return values. Analogously, if $ret = ret'$ but $post \neq post'$ a warning is issued.

5 Translating I/O-Automata into UML 2.0 State Machines

The generated I/O-automaton for the integrated behaviour of a system object in scenarios can be seen as a UML 2.0 state machine keeping the state-transition structure and only turning in-messages into triggers and sequences of out-messages with a return into effects. The drawback of this mere adaptation of the notation to UML is that it shows different activations following the same incoming message on different transitions retaining unnecessary non-determinism.

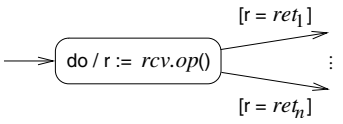


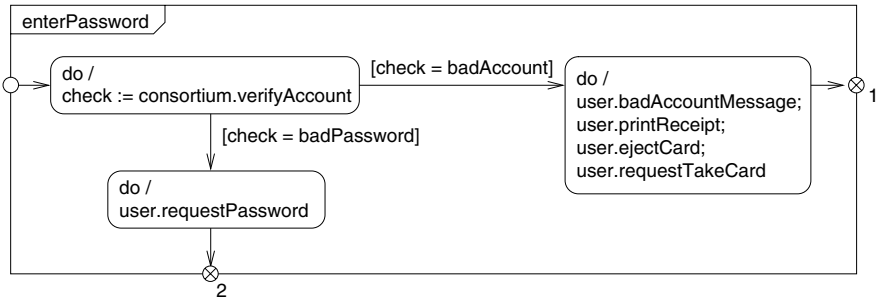
Fig. 4. Actions as state machine fragment

Thus, in order to obtain a comprehensive representation of the activity that follows an incoming message we transfer activations to state machines by discerning two kinds of states: *stable* states, in which a system object waits for a message; and *activity* states, in which the reaction to an incoming message is processed. The different sequences of outgoing messages and the subsequent returns mark different exits to such an activity state, in general leading to different (stable) successor states. In UML 2.0, submachine states provide the necessary structure for the activity states, with entry and exit points (shown as circles and crossed circles) encapsulating the internal behaviour of the contained state machine; simple states represent stable states. In fact, activity states capture procedures in the sense of "Executable UML" [7], but make case distinctions in procedure executions graphically explicit.

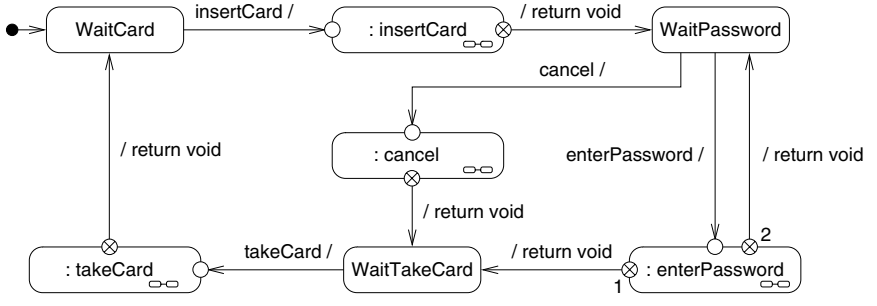
For integrating the reaction to an incoming message $in \in In$ in a state $z \in Z$ of an I/O-automaton with initial state $(Z, In, Out, \delta, z_0) = intio(S_0, \{S_1, \ldots, S_n\}, o)$ into a submachine, we first turn the outgoing operation calls in the activation set $R(z, in) = \{out \in Out \mid \exists z' \in Z \,.\, (z, in, out, z') \in \delta\}$ into state machine fragments: If (op, rcv) is a pair of an operation and a receiving object such that $\langle op, rcv, ret\rangle$ occurs in an $out \in R(z, in)$ with some return ret and if $(m_1 \ldots m_k \ \langle op, rcv, ret_1\rangle) \ldots (m_1 \ldots m_k \ \langle op, rcv, ret_n\rangle)$ are all occurrences of (op, rcv) in $R(z, in)$ after a common prefix $m_1 \ldots m_k$ with $n > 0$ different returns $ret_1, \ldots, ret_n$, we construct a state machine fragment $M(z, in, m_1 \ldots m_k, op, rcv)$ of the form in Fig. 4 with r an auxiliary variable. In a next step, the different state machine fragments for out-messages in the activation set $R(z, in)$ are assembled into a single submachine: The transition for [r = ret_i] of $M(z, in, m_1 \ldots m_k, op, rcv)$ is merged to the incoming transition of $M(z, in, m_1 \ldots m_k \langle op, rcv, ret_i\rangle, op', rcv')$. Finally, we define an entry point for each $M(z, in, \varepsilon, op, rcv)$ and exit points for each $M(z, in, m_1 \ldots m_k, op, rcv)$ with $m_1 \ldots m_k$ a maximal sequence of out-messages in $R(z, in)$. The result of applying this procedure to enterPassword in the state WaitPassword in Fig. 2(a) is depicted in Fig. 5(a).

Having defined submachines for the activation sets $R(z, in)$ an integrated state machine can now be synthesised by introducing stable states from the I/O-automaton as simple states and connecting these by transitions to activity states as submachine states referencing the submachines from $R(z, in)$. For the two ATM scenarios the result of the translation of the I/O-automaton of atm in Fig. 2(a) is shown in Fig. 5(b). The states WaitCard, WaitPassword, and WaitTakeCard are the stable states, the states : insertCard, : enterPassword, : cancel, and : takeCard are the activity states of the state machine.

It is worth noting that for each system object the separation of stable and activity states leads to a sequential behavioural design model which can be directly implemented using the state pattern [10] or a straightforward implementation by means of state variables. The latter approach represents the incoming messages of a system object by methods which show a case distinction according to the stable states and



(a) UML 2.0 submachine for reaction of atm to enterPassword.



(b) UML 2.0 state machine of atm with stable and activity states.

Fig. 5. Synthesised state machines for atm

implement the behaviour of the activity states. Using Java as implementation language and recording the current state in an enumeration-typed variable `currentState`, the implementation of `enterPassword` takes the following form:

```
void enterPassword() {
  switch (currentState) {
    case WAIT_PASSWORD:
      VerificationResult check = consortium.verifyAccount();
      if (check == VerificationResult.BAD_ACCOUNT) {
        user.badAccountMessage(); ...
        currentState = State.WAIT_TAKECARD;
        return;
      }
      if (check == VerificationResult.BAD_PASSWORD) {
        user.requestPassword();
        currentState = State.WAIT_PASSWORD;
        return;
      }
      break;
    default:
  }
}
```

6 Related Work

To our knowledge, in contrast to all other approaches to state machine synthesis our method sets out from scenarios with a clear distinction between inactive (stable) states and activation phases that follow as a reaction to a synchronous operation call. Our approach is activity-driven in the sense that during the transformation process different activations occurring in different scenarios but following the same incoming message (in the same stable state) are integrated into one single activity which models the behaviour of an operation across many scenarios. Such a model can be easily translated into a sequential program.

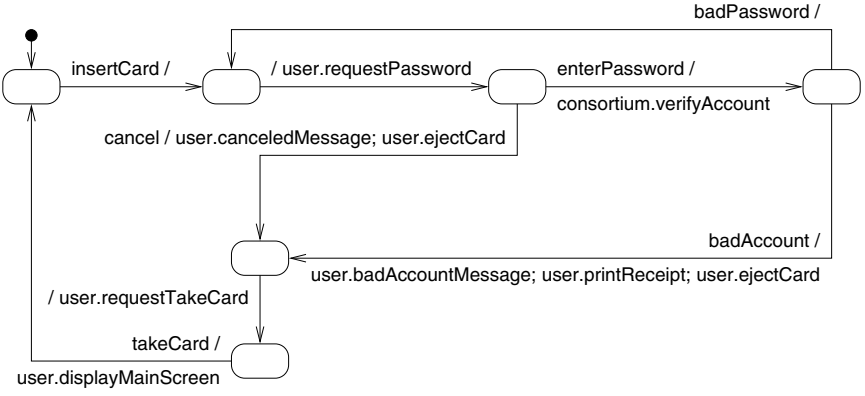
In order to compare our results with the literature we can use the same ATM case study which, in an asynchronous environment, is modelled by the same scenarios as shown in Fig. 1, but deleting all states and replacing all synchronous and return messages by asynchronous messages [2,5] (we have omitted the initial outgoing displayMainScreen message). This example is also the basis for the detailed comparison in [5]. The crucial difference to the synchronous case is that instead of the return values badAccount and badBankAccount of the operations verifyAccount and verifyCardWithBank, respectively, now (call-back) messages are used to indicate the result of a verification.[3]

For the integration of the asynchronous scenarios different strategies have been proposed in the literature. According to [6], these approaches can mainly be categorised into synthesis algorithms which are based on matching conditions and algorithms which are based on matching events or actions.

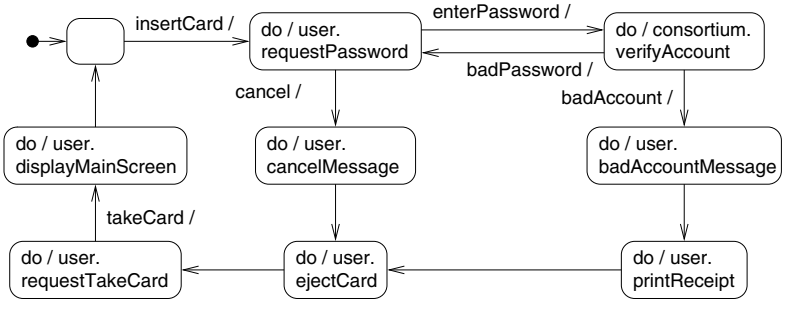
As an instance of the first group, the integration procedure of Whittle and Schumann [2] is based on matching of pre-/post-conditions that the user has to provide (for incoming and outgoing messages) as an input to the transformation process. As a result, Whittle and Schumann obtain for the atm the state machine in Fig. 6(a) which has a completely different structure than the activity-based state machine for synchronous communication shown in Fig. 5. Also the hierarchical state machine developed in a last step in Whittle and Schumann's algorithm does not follow an activity-based integration strategy but groups states according to values of state variables in pre-/post-conditions. The approaches by Krüger et al. [3] and Uchitel et al. [5] are based on a similar strategy; here, explicit states, like in our scenarios, have to be provided, and Uchitel et al. also take into account combinations of basic scenario blocks.

The Fujaba approach to integrate scenarios into a state machine, proposed by Maier and Zündorf [4], is based on matching of send actions. In this way one would obtain for the above scenarios the state machine in Fig. 6(b) which in turn is not aimed at exhibiting the computational behaviour of operations. The SCED/MAS algorithm by Mäkinen and Systä [11] falls in the same category of event matching procedures, but also uses a learning procedure for synthesis.

[3] In the asynchronous approach, [2,5] distinguish also a further scenario where the user cancels the transaction before the consortium has called back the atm, which is not possible in the synchronous approach.



(a) According to Whittle and Schumann [2]



(b) According to Maier and Zündorf [4]

Fig. 6. Integrated atm state machine in asynchronous approaches

7 Conclusions and Future Work

We have described a model transformation procedure for synthesising an integrated UML 2.0 state machine from scenarios given as UML 2.0 interactions. The approach focuses on reactive objects, where activities are triggered by incoming synchronous messages. The transformation procedure constructs an intermediate I/O-automaton for the behaviour of a single object integrating the scenarios. The non-deterministic transitions of the I/O-automaton provide the basis for systematic feedback to the user of the transformation about integration problems, which will iteratively lead to more complete and refined scenarios. Finally, the integrated I/O-automaton is translated into a UML 2.0 state machine which describes the overall behaviour of an object distinguishing between stable and activity states. The activity states are represented by UML 2.0 submachines (with entry- and exit-points) modelling the reaction of an object to an operation call (occurring in a stable state).

A related distinction has been suggested by Tenzer and Stevens [12] who use protocol state machines to specify the permissible sequences of operation calls occurring in stable states and method state machines to model the execution of the actions of an operation. However, Tenzer and Stevens do not model state dependent reactions on

operation calls and do not focus on a synthesis procedure but rather on the modelling of recursive calls and call-backs which we have not considered yet. Further steps to make our synthesis algorithm more complete concern message parameters, which can be easily added, and an extension to scenarios with both synchronous and asynchronous messages, which should work along the same lines as the current approach.

Finally, let us remark that our approach can be adjusted to a component-based framework where scenarios are used to identify provided and required interfaces of components and the synthesis procedure generates UML 2.0 protocol state machines for the ports of a component. In particular, the encapsulation of activities allows for independent refinement, of operation behaviour, on the one hand, and of the protocol on the other.

References

1. Fowler, M.: UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, Boston–&c. (1997)
2. Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. In: Proc. 22nd Int. Conf. Software Engineering (ICSE'00), IEEE Press (2000) 314–323
3. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In Rammig, F.J., ed.: Distributed and Parallel Embedded Systems, Kluwer Academic, Boston–Dordrecht (1999) 61–71
4. Maier, T., Zündorf, A.: The Fujaba Statechart Synthesis Approach. In: Proc. 2nd Int. Wsh. Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'03), Portland, Oregon (2003)
5. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Trans. Softw. Eng. **29** (2003) 99–115
6. Liang, H., Dingel, J., Diskin, Z.: A Comparative Survey of Scenario-based To State-based Model Synthesis Approaches. In: Proc. 5th Int. Wsh. Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06), Shanghai (2006) 5–12
7. Mellor, S.J., Balcer, M.J.: Executable UML — A Foundation for Model-Driven Architecture. Addison-Wesley, Boston–&c. (2002)
8. Blaha, M., Rumbaugh, J.: Object-Oriented Modeling and Design with UML. 2nd edn. Pearson Education, Upper Saddle River, N. J. (2005)
9. Object Management Group: Unified Modeling Language: Superstructure, version 2.0. (2005) http://www.omg.org/cgi-bin/doc?formal/05-07-04(06/12/28).
10. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. Addison-Wesley, Boston–&c. (1995)
11. Mäkinen, E., Systä, T.: MAS: An Interactive Synthesizer to Support Behavioral Modelling in UML. In: Proc. 23rd IEEE Int. Conf. Software Engineering (ICSE'01), IEEE Computer Society (2001) 15–24
12. Tenzer, J., Stevens, P.: Modelling Recursive Calls with UML State Diagrams. In Pezzè, M., ed.: Proc. 6th Int. Conf. Fundamental Approaches to Software Engineering (FASE'03). Volume 2621 of Lect. Notes Comp. Sci., Springer, Berlin (2003) 135–149