An Integrated Formal Model of Scenarios Based on Statecharts

Martin Glinz

Institut für Informatik
University of Zurich
CH-8057 Zurich, Switzerland
glinz@ifi.unizh.ch

ABSTRACT

State automata are an attractive means for formally representing scenarios. Scenarios describe how users interact with a system. However, the current approaches treat every scenario as a separate entity.

This paper introduces a statechart-based model that allows the formal composition of all scenarios of a system into an integrated, consistent model of external system behavior. The scenarios remain visible as building blocks in the model. The meaning of the integrated model is derived from the meanings of the constituent scenarios and from the semantics of the composition rules.

The paper defines the composition rules and shows their application. The capabilities for analyzing and verifying the model are demonstrated. An extension of the scenario model to a general system model is sketched.

1 INTRODUCTION

Using scenarios or use cases for requirements elicitation and representation has received significant attention in the last few years (Jacobson 1992, Rubin and Goldberg 1992, Anderson and Durney 1993, Hsia et al. 1994). A scenario or use case (I consider these two terms as synonyms) is a sequence of interactions between a user and a system. Thus, it describes an aspect of the external behavior of a system from a certain user's viewpoint.

There are several ways to represent a single scenario. Jacobson uses a mostly informal text notation. Rubin and Goldberg introduce a tabular notation of scenario scripts. Hsia et al. (1994) show that a scenario can be adequately represented by a regular language, or equivalently, by a finite state automaton.

The formal or semiformal notations may be rather more difficult to produce and to read than informal ones. Additionally, working with formal notations requires some training. Nevertheless, the increased effort pays off. Formality allows the verification of properties

Published in: Schäfer, W. and Botella, P. (eds.) (1995). Software Engineering - ESEC ’95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. Berlin, etc.: Springer (Lecture Notes in Computer Science 989). 254-271.

of the scenarios and eases the detection of inconsistencies and incompleteness both within and between scenarios. Furthermore, formal notations allow simulation or automatic prototyping of a scenario. Both are of vital importance for requirements validation.

However, the current formal or semiformal approaches model every scenario separately. There is no notation that integrates all scenarios into *one consistent model* of system behavior in such a way that the constituent scenarios are still visible, and can be retrieved as views.

There are three challenges in generating such a model.

- Challenge 1: How can a model be constructed that shows the relationships between the constituent scenarios and leaves their internal structure unchanged?
- Challenge 2: How can the model be used to verify properties of the scenarios and to detect inconsistencies between different scenarios? Can checking be automated, at least partially?
- Challenge 3: Can such an integrated scenario model be extended in a straightforward way to a complete model of system requirements?

In this paper I shall show that challenges 1 and 2 can be met using a statechart based model. In section two, I give a short introduction to statecharts and define their semantics. In sections three and four I present my approach of using statecharts to compose scenarios into an integrated model of external system behavior. In section five I sketch some ideas and preliminary results of how to extend this scenario model to a general systems model that meets challenge 3. In section six I briefly discuss the rationale for choosing statecharts as a means of modeling and composing scenarios.

This paper does not simply propose the use of statecharts as a new technique for scenario representation. The main contribution is to use statecharts as a mechanism for the formal composition of a set of scenarios into an integrated model of system behavior and to show the powerful analysis and verification capabilities of such a model.

2 STATECHARTS

Statecharts (Harel 1987, 1988) are an extension of state-event diagrams to include decomposition and concurrency. State-event diagrams in turn are based on finite state automata. Any state in a statechart can recursively be decomposed into
a) another statechart (hierarchical decomposition)
b) two or more parallel statecharts (concurrency; Harel calls that orthogonality).
At the bottom of the decomposition, all statecharts are ordinary state-event diagrams.

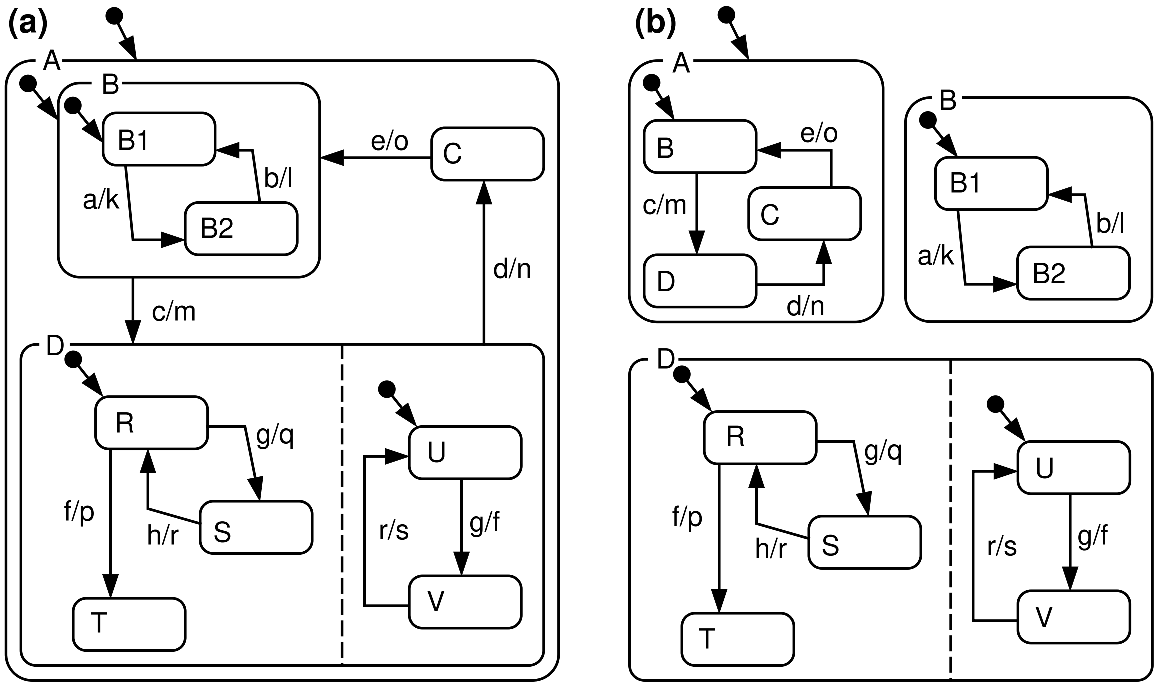
Harel himself defines the principle of statecharts as "statecharts = state diagrams + depth + orthogonality + broadcast-communication" (Harel 1987, p. 233).

The rules for the interpretation of statecharts are mostly those of state-event diagrams: state transitions are triggered by external or internal events. When a transition is triggered, the system leaves its current state, initiates the action(s) specified for that transition, and enters a new state. Any initiation of an action can be interpreted as an occurrence of an internal event. Events are neither saved nor queued. Events that do not trigger a state transition immediately upon their occurrence are lost.

The following additional rules hold for statecharts only: any (external or internal) event is broadcast simultaneously to all state transitions in all statecharts. Within a statechart having concurrent sub-statecharts, the system state is composed of the states of the concurrent sub-statecharts. State transitions between concurrent statecharts are not allowed. (Thus, the expressive power of statecharts is still equivalent to that of finite state automata; see the canonical mapping below.) Concurrent statecharts may synchronize and exchange information using events.

We do not use all features that Harel defines for statecharts. Histories and overlapping states are omitted. The simpler model is sufficient for our purposes and its semantics are easier to define.

Figure 1 shows a hierarchy of statecharts together with an explanation.



Explanation of symbols. States and statecharts are denoted by rectangles, transitions by arrows. x/y denotes a triggering event x and a triggered action y for a state transition. Arrows originating in a black dot denote initial states on the corresponding nesting levels. A dotted line separates concurrent statecharts. Notations **(a)** and **(b)** are equivalent.

Sample Interpretation. The sequence of external events <Start, a, c, h, g, h, d, e> produces the following sequence of actions: <–, k, m, –, q and f, r and s, n, o>, stepping through the following sequence of states: <B1, B2, R||U, S||V, R||V, R||U, C, B1>. '–' means no action, X||Y means both in state X and Y. R||V is a transient state that is left immediately when it is entered. This is due to internal event r which, when generated by the transition from S to R, immediately triggers the transition from V to U.

Figure 1. A set of statecharts

Statecharts can be given well defined formal semantics. In the context of this paper, a detailed understanding of the subtleties of timing and state transition semantics is not necessary. Readers who are not interested in formal foundations may therefore skip the rest of this section with the exception of figures 2 and 3 and their explanations.

We start with the definition of timing rules. Let t: $X -e/a_1,\ldots,a_n\rightarrow Y$ be a transition from state X to state Y which is triggered by event e and produces outputs $a_1,\ldots,a_n$. Let e happen at time t_e on the time scale of non-negative real numbers. Then state X is left at time t_e. State Y is entered at $t_e + \varepsilon_0$, $\varepsilon_0 > 0$. Output a_i is produced at time $t_e + \varepsilon_i$, $\varepsilon_i > \varepsilon_0$ and $\varepsilon_i > \varepsilon_j$ for all $i > j$, $1 \le i \le n$. If a_i is an event that triggers another state transition (for example, in a concurrent statechart) then this transition completes in the interval $t_e + \varepsilon_i \le t < t_e + \varepsilon_{i+1}$. The ε_i are chosen such that for any event e' happening at time $t_{e'} > t_e$ and for any δ with $0 < \delta < | t_{e'} - t_e|$ holds $\varepsilon_0 < \varepsilon_1 < \varepsilon_2 < ... < \varepsilon_{n+1} < \delta$ (at time $t_e + \varepsilon_{n+1}$, transition t and all transitions triggered by outputs a_i, $1 \le i \le n$ have completed). This last condition means that entering the new state(s) and producing the outputs happen in an arbitrarily small time interval after t_e, where nothing else can happen. With this condition and with the additional assumption that at no point in time can more than one event happen, we have a *quasi-synchronous timing paradigm*: a state transition takes time, but the time interval is infinitesimally short. Leveson et al. (1994) use a similar paradigm for their specification language RSML (which is derived from statecharts).

For statecharts, this quasi-synchronous paradigm has advantages over the synchronous one which is commonly used for state automata. (Synchronous means that the complete state transition happens at one point in time.) The quasi-synchronous paradigm avoids nondeterminism and counterintuitive behavior in concurrent statecharts and simplifies the canonical mapping (see below). For example, consider statechart D of Figure 1b. Under the synchronous paradigm, event g would nondeterministically trigger either transitions U –g/f→ V and R –g/q→ S or U –g/f→ V and R –f/p→ T. The latter one is counterintuitive.

On the other hand, there is a disadvantage, too: endless transitions can occur. For example, the transitions t_1: X –r/s→ Y and t_2: Y –s/r→ X form a never-ending loop if one of them is triggered. Therefore, any such cyclic chains of self-triggering state transitions must be avoided. However, this is no severe restriction. In reality, where every state transition takes some real time, we would have an endless loop in this situation, too. Moreover, the property that every state transition terminates in a finite number of steps can be proved for a given statechart if necessary.

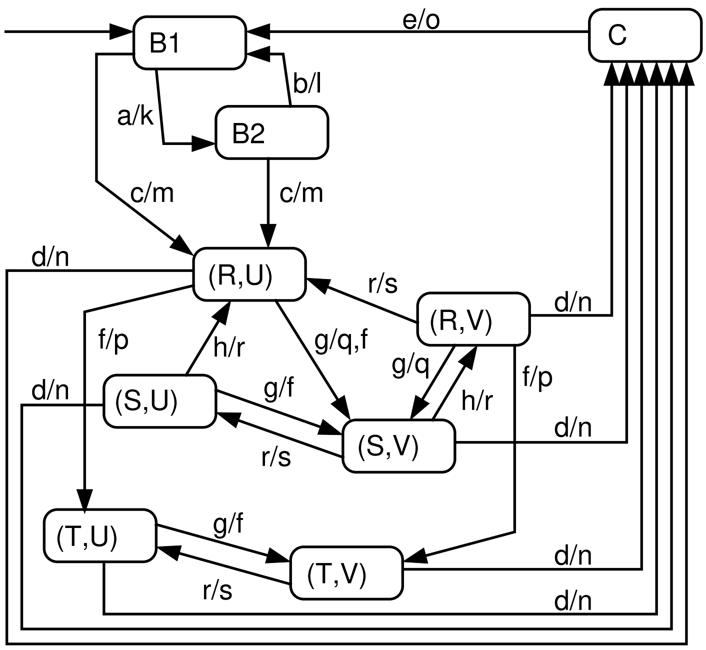
On the basis of the timing rules introduced above, we can now define the semantics of state transitions in statecharts. We do this by a *canonical mapping* from a given statechart hierarchy to a plain state-event diagram. This diagram is interpreted as a finite state automaton with quasi-synchronous timing. Properties and semantics of this automaton define the properties and semantics of the given statechart hierarchy in a proper and unambiguous way. The algorithm of this canonical mapping is described in a condensed form below. Figure 2 illustrates the canonical mapping for the statechart of Figure 1.

The canonical mapping flattens the hierarchy by recursive bottom-up insertion of state diagrams. Every superstate S is replaced by its constituent state diagram SD. State transitions to S are replaced by equivalent transitions to the initial state of SD. Any state transition from S to a state T is replaced by transitions from every state in SD to T.
Mapping concurrent statecharts is more complicated. Let statechart S consist of n concurrent statecharts $SC_1,...,SC_n$. SC_1 to SC_n are first transformed into flat state diagrams $SD_1,...,SD_n$, applying the canonical mapping recursively. Then these state diagrams are replaced by the Cartesian product of all concurrent states. The set of initial states E_i forms the initial state $(E_1,...,E_n)$ of the new state diagram. The state transitions in $SD_1,...,SD_n$ are mapped to state transitions on the elements of the Cartesian product state diagram as follows: let $X_1,...,X_n$ be any set of states with X_i state in SD_i, $1 \le i \le n$. Let t_i be any state

transition X_i $-e/a\rightarrow$ Y_i which is triggered by event e and initiates action a. Then t_i is mapped as follows:

(i) If there do not exist any states X_j, Y_j in SD_j, $i\neq j$ with t_j: X_j $-e/a_j\rightarrow$ Y_j (*) i.e. event e triggers no transition in any other diagram SD_j from state X_j to a state Y_j), then t_i is mapped to transitions $(X_1,...,X_{i-1},X_i,X_{i+1},...,X_n)$ $-e/a\rightarrow$ $(X_1,...,X_{i-1},Y_i,X_{i+1},...,X_n)$ for all state tuples where X_i and Y_i occur and condition (*) holds

(ii) If $X_{j1},...,X_{jk}$ is the largest subset of the states $X_1,...,X_n$ for which there exist transitions t_{jl}: X_{jl} $-e/a_{jl}\rightarrow$ Y_{jl} for all $1\leq l \leq k$ (i.e. event e triggers transitions $t_{jl},..., t_{jk}$ with actions $a_{jl},...,a_{jk}$ concurrently), then t_{jl} is mapped to transitions $(X_1,...,X_{j1},..,X_{jl},..,X_{jk},...,X_n)$ $-e/a_{j1},...,a_{jk}\rightarrow$ $(X_1,...,Y_{j1},..,Y_{jl},..,Y_{jk},...,X_n)$ for all state tuples where the subsets $X_{j1},...,X_{jk}$ and $Y_{j1},...,Y_{jk}$ occur.



Sequence of mapping steps for this example.

(1) The concurrent statecharts in D are replaced by the Cartesian product of their states. Transitions R–g/q→S and U–g/f→V are mapped by rule (ii) for state combination (R,U) and by rule (i) for the other state combinations. All other transitions are mapped by rule (i).

(2) Replace B and D by their constituent state diagrams. B1 becomes the new initial state, transition B–c/m→D is replaced by B1–c/m→(R,U) and B2–c/m →(R,U). Transition D–d/n→C is mapped in the same way.

(3) Replace A by its constituent state diagram.

Figure 2. Result of canonically mapping the statechart hierarchy of Fig. 1 to a plain state diagram, giving the exact semantics of these statecharts

When working with statecharts, especially for statechart composition, some particular kinds of state transition conditions are useful. Their notation is given in Figure 3.

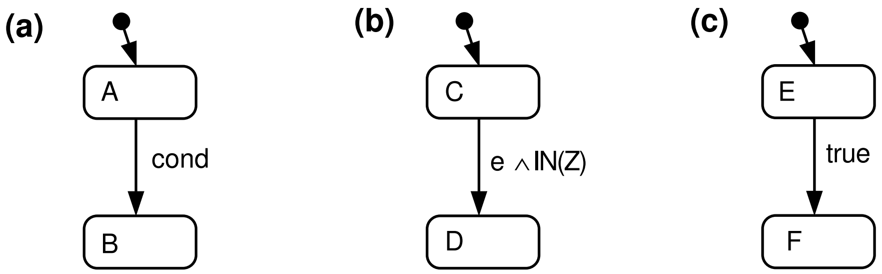


Figure 3. Special kinds of state transitions (the semantics is given below in the text)

In Fig. 3 (a), a state transition from A to B occurs when condition cond becomes true. If cond is already true when the system enters state A, this state is immediately left to enter state B. Fig. 3 (b) shows a combination of an event with a special kind of condition. A state transition from C to D occurs when event e occurs and the system is in state Z. Z must be a substate of C. This can be used to specify that a complex statechart can be left only from a specific terminating substate. Fig. 3 (c) shows an unconditional state transition. Whenever the system enters state E, it immediately leaves E to enter F. State E is redundant, i.e. it could be removed from the model without changing its semantics. However, such redundant states occur (and make sense) when composing statecharts out of components without modifying these components.

3 COMPOSITION OF SCENARIOS

3.1 Describing Single Scenarios

Every single scenario is modeled with a statechart. Normally, this statechart will be a plain state diagram. Only large scenarios might already require some hierarchical depth.

In order to present concrete examples, we introduce a sample application.

Sample Application: The Department Library System

The system shall support a department library, where students themselves can take books from the shelves, read them, borrow / return books, and query the library catalogue.
Every book has a barcode label. Additionally, there is an electronic safety device located under that barcode label which can be turned on and off by the system.
When a library user wants to borrow a book, she takes it to the check-out station. There she first scans her personal library card. Then she scans the barcode label of the book. If she has no borrowed books that are overdue, the systems registers the book as being borrowed by her and turns off the electronic safety device of that book. Several books can be checked out together. The check-out procedure is terminated by pressing a 'Finished' key.
The check-in procedure for returning a book works in a similar way.
In order to be admitted to the library, a potential user must first be registered by the library personnel. The user has to identify herself and must provide her personal data. The system registers this data and produces a personal library card for her.
Similar procedures are to be provided for deleting and updating a user's registration.
At the exit of the library, there is a security gate. When a user tries to leave the library carrying with her a book that has not been checked out properly, the system sounds an alarm and locks the exit door. By pressing an emergency button, the exit door can be unlocked in case of emergency.
For the sake of simplicity, the requirements for determining and fining users having overdue books are not specified. Obviously, the system works the same for male users.
The library personnel interacts with the system when maintaining the library catalogue, when registering / updating / deleting users, and when querying the status of users or books.

As a first example, we consider the scenarios concerning library users. There are seven scenarios: Get registered, Borrow books, Return books, Query catalogue, Exit library, Update personal data, and Get deleted.

In Figure 4, the scenarios Borrow books (Fig. 4 a) and Get registered (Fig. 4 b) are modeled as simple statecharts (in fact, plain state diagrams).

As we want to compose scenarios into an integrated model of external behavior, we model every scenario as a statechart having exactly one initial and one terminal state. The cyclic model of Hsia et al. (1994) with identical initial and terminal state is not suited for composition.

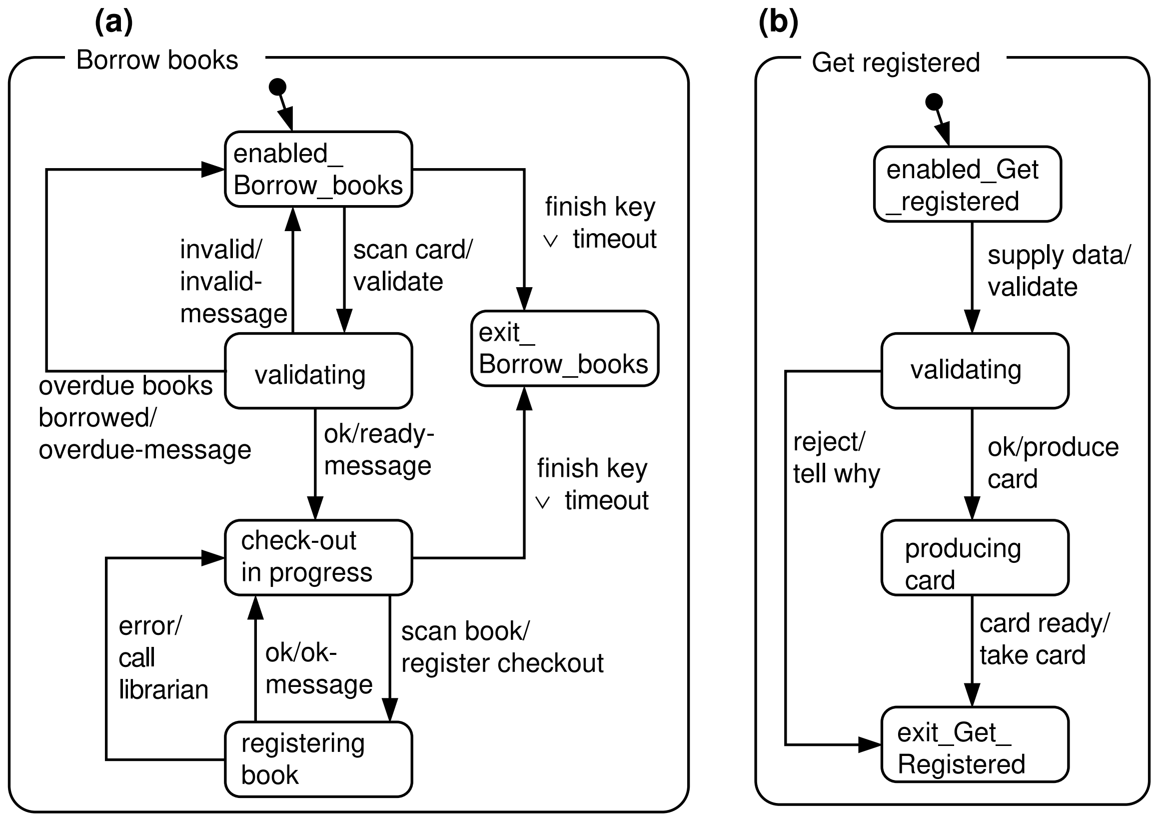


Figure 4. Statecharts modeling the scenarios “Borrow books” (a) and “Get registered” (b). $x \vee y$ means event x or event y

3.2 Principles of Scenario Composition

We assume that we have defined a set of disjoint single scenarios using statecharts or state diagrams as described above. (The treatment of overlapping scenarios is described in section 3.4.) Now we want to integrate these scenarios into a single model.

With respect to the order of execution, scenarios can be related in four ways (let A, B be scenarios): B after A (sequence), either A or B (alternative), A followed n times by itself (iteration), and A concurrent with B (concurrency). For our Library user scenarios, we assume that these scenarios are related as depicted in Figure 5. As a notation, I use a Jackson style diagram with a straightforward extension to include concurrency. Such diagrams can also be used to validate the assumptions about scenario relationships.

From the theory of structured programming we know that single-entry-single-exit constructs can be composed easily and systematically: higher level structures are built by concatenating and nesting blocks according to the relationships sequence, alternative, iteration, and – for non sequential programs – concurrency. Thus, scenarios must be composable in the same way if we model them as single-entry-single-exit constructs, i.e. if they all have exactly one initial and one terminal state.

In the sequel, I will define statechart templates for composing statecharts according to the four kinds of relationships. In order to simplify the notation, I first introduce the notion of closed statecharts. These are statecharts with exactly one initial and one terminal state.

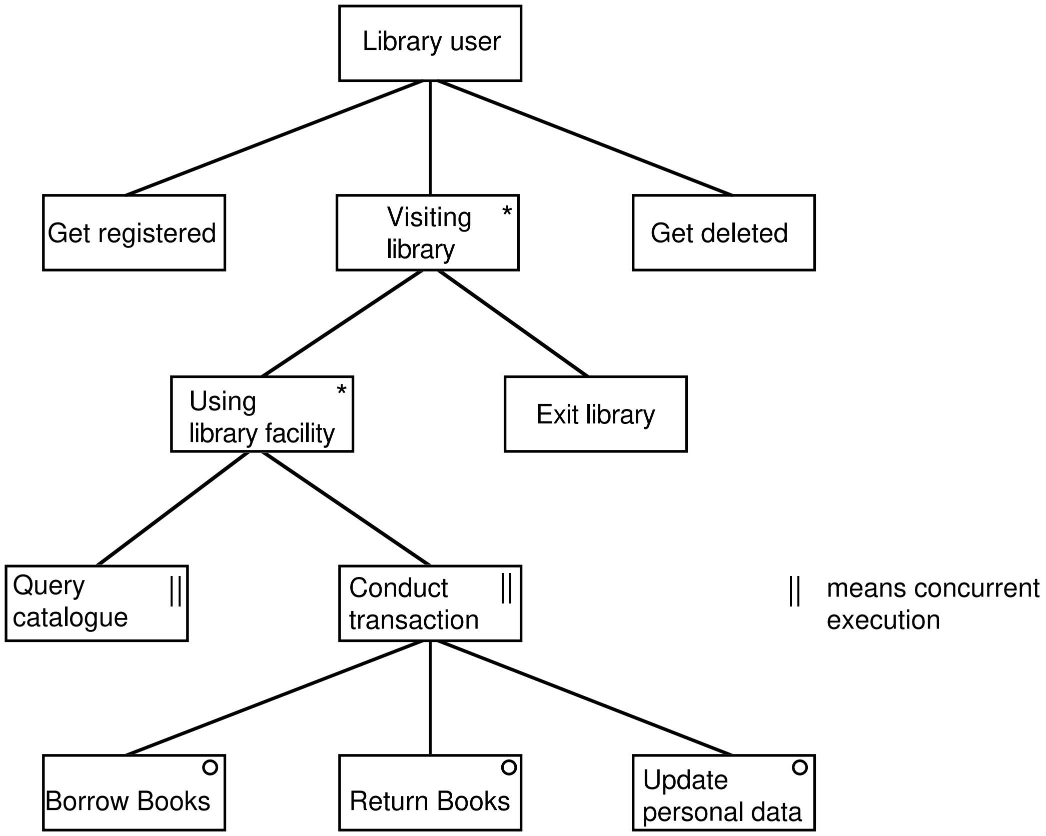


Figure 5. Relationships between the scenarios for library users in a Jackson style notation

DEFINITION. A *closed statechart* is a statechart which has exactly one entry state and which can be left if and only if it is in exactly one exit state. By convention, the entry state of a closed statechart A is named enabled_A. The exit state is named exit_A.

Figure 6 shows a shorthand notation for closed statecharts. The bars at the top and the bottom of the chart symbolize the entry and exit states, respectively. Since we model every scenario with exactly one entry and one exit state (see 3.1 above), all our scenario statecharts are closed ones.

3.3 Composing Scenario Statecharts

Now we can define the four composition templates for scenario statecharts as described in Figure 7. As the semantics of the concurrency template is not intuitively clear, it is explained using a normal statechart (Fig. 7 e). For alternative and iteration, conditions must be defined that determine the alternative to be taken and the number of iterations, respectively. The iteration template models a WHILE-DO iteration. States with no triggering condition are unconditional transitions, i.e. they have the trigger “true”.

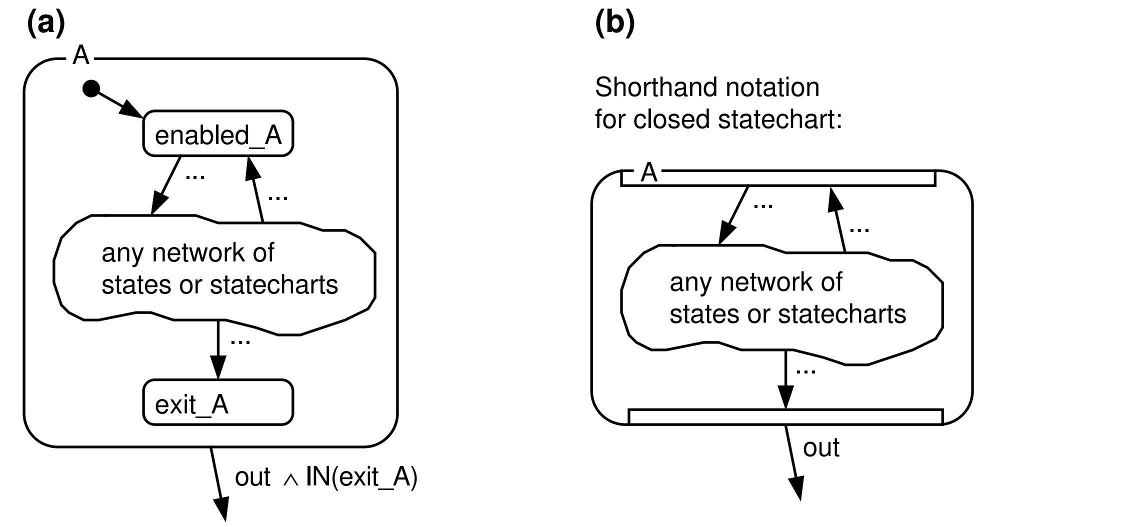


Figure 6. A closed statechart in normal notation (a) and in a shorthand notation (b)

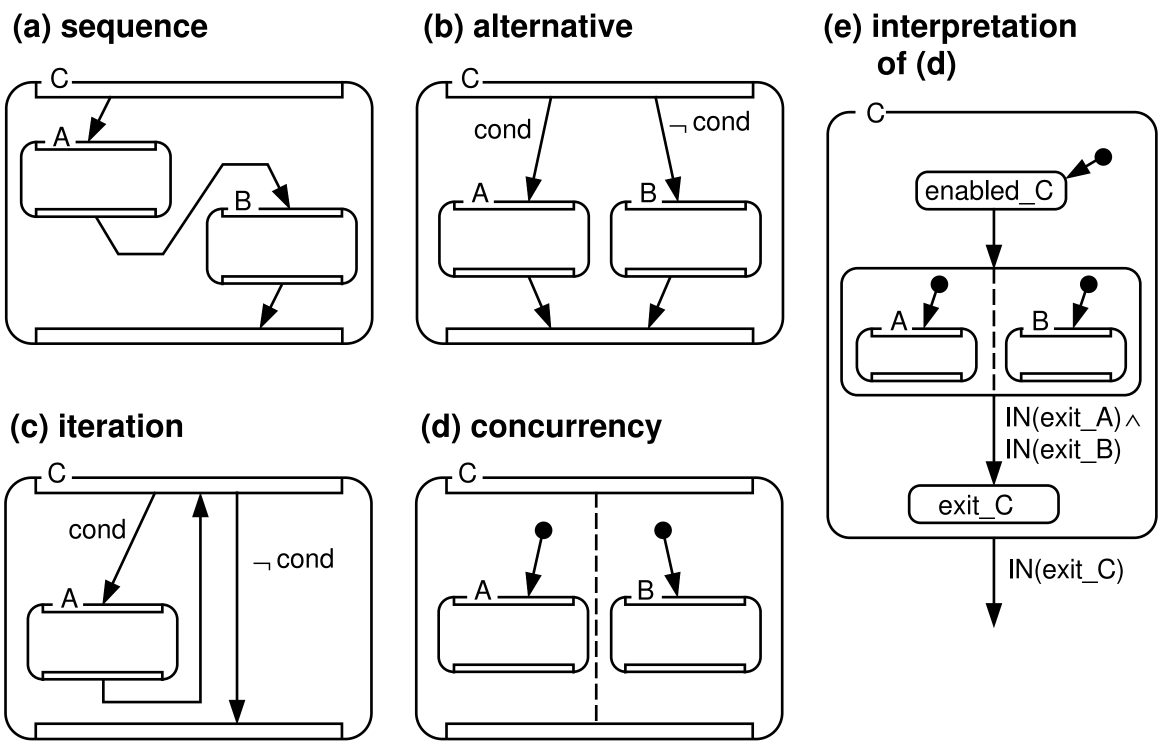


Figure 7. Statechart templates for scenario composition

Using these templates, we can now model all scenarios concerning library users together in a single statechart model (Fig. 8). The construction is guided by the ordering of the scenarios given in Figure 5. Get Registered is followed by an iteration of Visiting library which is labeled Admitted to library. This iteration in turn is followed by Get deleted. In a second step, Visiting library is refined by an iteration of Using library facility, followed by Exit library. In the next step, Using library facility is specified to be a concurrent composition of Query catalogue and Conduct transaction. In the last step, the latter scenario is refined to be an alternative of Borrow Books, Return Books, and Update personal data. Instead of working top down, construction could have proceeded bottom up, too.

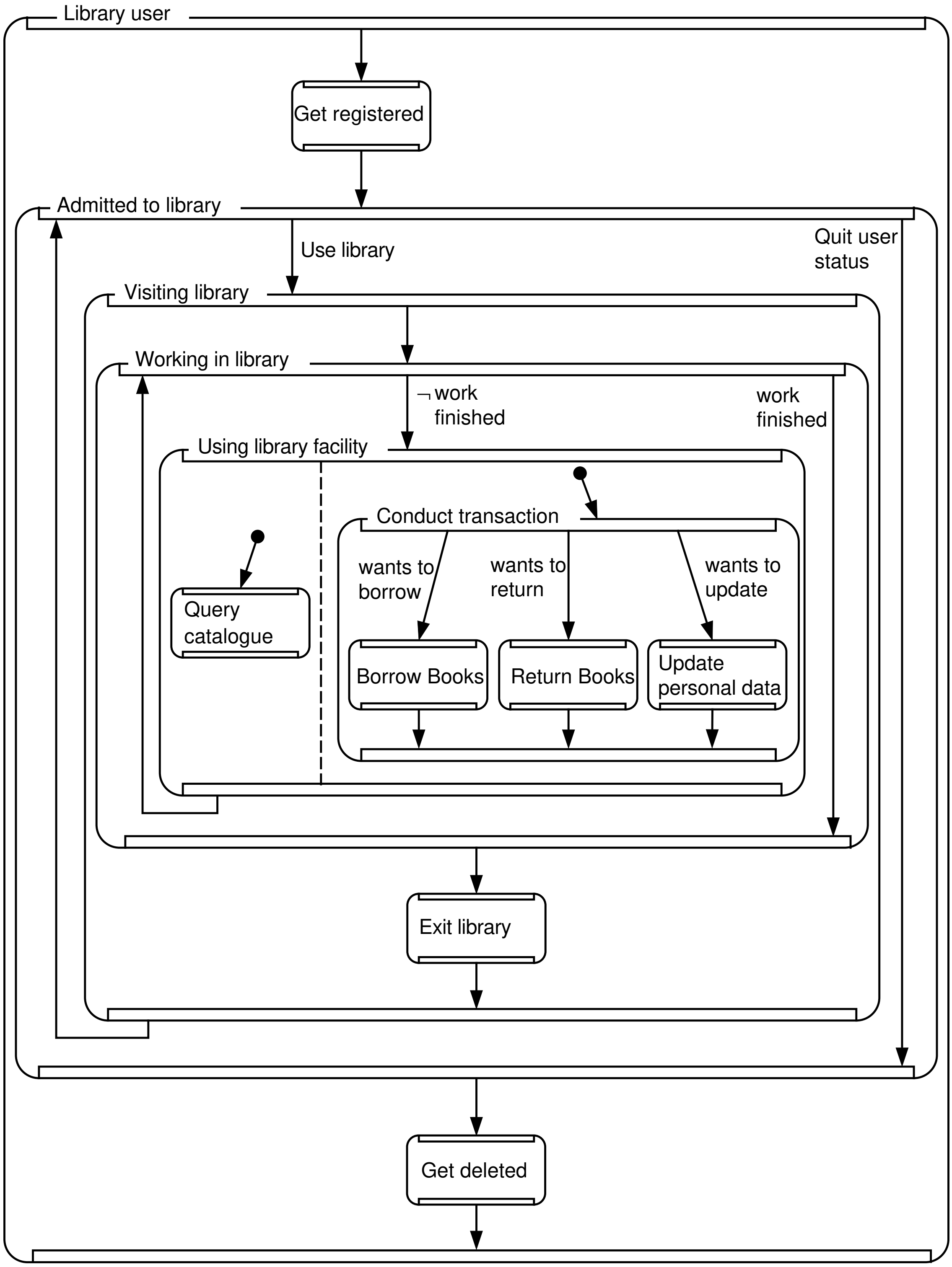


Figure 8. Integrated model of all scenarios for Library user

The complete scenario model of the library is modeled on an abstract level in Fig. 9. The detailed models for Librarian and Exit gate are still to be specified.

Figures 8 and 9 illustrate that the notation supports both bottom-up and top down development of scenario models. Components in any state of completion (from unspecified via informal texts to full-fledged statecharts) fit together well in the composition framework described above. As every component of the model is a statechart, a model under development is always analyzable and executable, no matter how complete it already is.

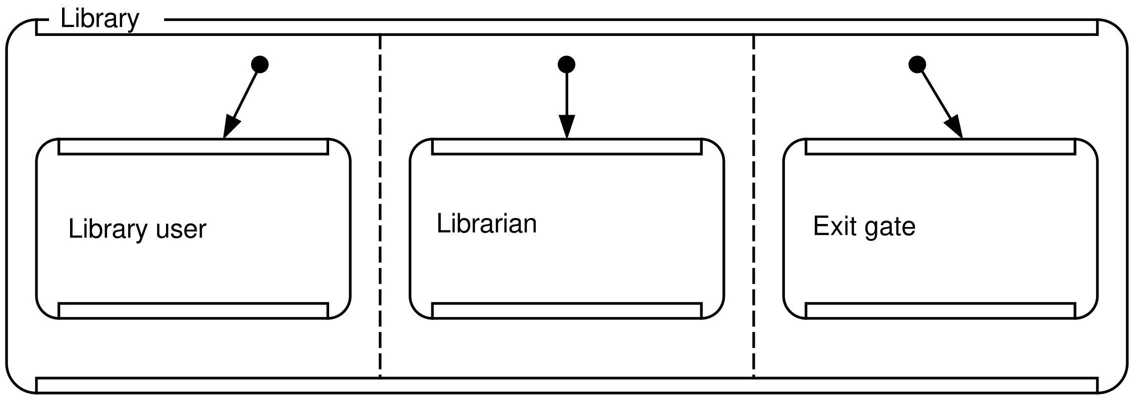


Figure 9. Scenario model of complete library system on an abstract level

3.4 Treating Overlapping Scenarios

Scenario composition as described above applies to disjoint scenarios only. This is due to the fact that the states of the component scenarios must be disjoint for proper composition. In my approach, states represent steps in the user-system interaction process. As long as different scenarios model disjoint portions of this process, there is no problem.

However, we also may have overlapping scenarios. Typically, this situation occurs when scenarios describe variants or facets of the same portion of the process (for example, normal execution and exception situations). In this case, the overlappings must be resolved prior to composition. This can be accomplished in two ways: (1) The overlapping scenarios can be decomposed into mutually disjoint sub-scenarios. (2) They can be fused into one single scenario. The concurrency features of statecharts are particularly helpful to accomplish the fusion of scenario variants.

3.5 Executing Scenario Models

Every statechart model is executable. The canonical mapping (see section 2) defines a finite state automaton which provides the required semantics for execution. However, the events in scenario models will frequently not be specified formally enough for direct execution. For example, when executing the scenario Borrow books (Fig. 4 a), the analyst must decide which of the three events (ok, invalid, overdue books borrowed) shall occur when being in state Validating. Therefore, execution must be interactive or script-driven. In the former case, the analyst interactively decides which events shall occur in a given state. In the latter case, the analyst annotates the states of the model with event scripts prior to execution.

Parts of the model which have not yet been specified can be treated in two ways during execution: for any unspecified statechart, the simulator running the execution may either insert a transition from the entry to the exit state and issue a warning message or, in an interactive execution session, it may request the analyst to supply an exit trigger.

4 MODEL ANALYSIS AND VERIFICATION

One of the principal advantages of a formal or at least semiformal model is that some of its properties can be formally analyzed and verified. The integrated scenario model allows analysis and verification of the complete behavioral specification of a system. This goes far beyond the analysis capabilities we had until now for isolated scenarios.

By exploiting the properties of statecharts, we can
- detect deadlocks
- determine reachability of states
- verify required mutual exclusions
- find inconsistent events and actions, particularly in inter-scenario relationships
- uncover incompleteness

in the behavioral specification of a complete system.

4.1 Deadlocks

Deadlocks occur when concurrent scenarios mutually wait for internal events to happen (see Fig. 10 a for a simple example). Such deadlock situations can be detected using a formal procedure. When applying the canonical mapping (see section 2) to the scenario model, any deadlock becomes equivalent to the existence of a non-redundant state that can be left by the occurrence of internal events only. As no internal event can occur when the system is in a non-redundant state, the system is deadlocked (Fig. 10 b).

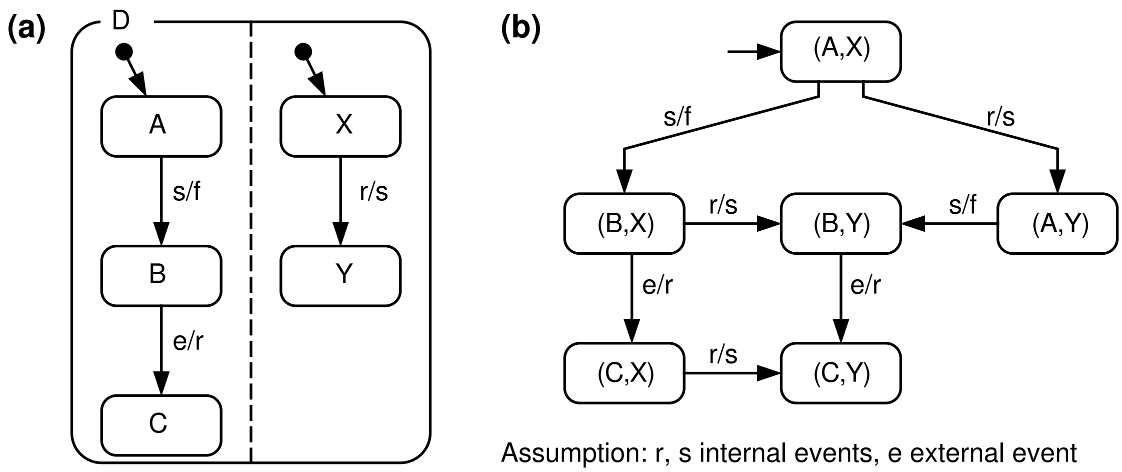


Figure 10. (a) Two concurrent statecharts with a deadlock. (b) Result of canonical mapping of statechart D, showing the deadlocking state (A,X)

4.2 Reachability

A necessary condition for the reachability of a state T from another state S is that there exists at least one path of state transitions from S to T. This property can easily be checked: the canonical mapping is applied to the scenario model. The resulting state diagram is analyzed with an algorithm to determine paths in directed graphs. However, the existence of a path of transitions is not a sufficient condition for reachability. Additionally, there must exist a feasible sequence of events triggering the transitions of at least one existing path. The existence of such a sequence of events must be determined by the analyst. The statechart model greatly simplifies this task because the set of existing paths and the set of possible events leading through any of these paths can be generated from the model.

4.3 Mutual Exclusion

DEFINITION. Two statecharts or plain states A, B are mutually exclusive, if and only if the system cannot concurrently be both in A and B.

LEMMA. Let A, B be statecharts or plain states within a statechart S and let S' be the result of the canonical mapping of S. A and B are mutually exclusive if and only if conditions (i) or (ii) hold:

(i) A and B are not concurrent statecharts or states

(ii) Any product state in S' containing both A and B is either redundant (i.e. it is immediately left upon entry) or it is unreachable from every other state.

The procedure to verify both conditions of the above lemma for a given pair of statecharts or states is obvious. Thus, required mutual exclusions between scenarios or parts of them can easily be verified in the scenario model.

4.4 Inconsistent Events and Actions

When working bottom up, it will frequently happen that the same events and actions are named differently in different parts of the model. A common glossary of events and actions contributes to the reduction of such errors, but normally does not eliminate them completely. When the model components are composed to form an integrated scenario model, such inconsistencies show up and can be corrected. At the latest any attempt to execute the model will show that expected transitions do not happen due to misnamed events and actions.

4.5 Incompleteness of the Model

When the scenarios are composed to an integrated model, missing parts (e.g. unspecified behavior in certain cases or missing transitions) are much easier to detect than in a set of separately modeled scenarios. Nevertheless, the statechart paradigm allows for intended, well controlled incompleteness during construction of the model (see end of section 3.3).

4.6 Tool Support

The verification of absence of deadlocks, reachability and mutual exclusion is based on formal procedures, so these procedures can be automated. Together with an execution simulator we thus can construct a powerful set of tools to support and partially automate the task of model validation and verification.

5 DIRECTIONS OF FURTHER RESEARCH

5.1 Including Formal Models of External Agents

The possible behavior of any external agent (human or machine) can be modeled using closed statecharts, too. Such models, in particular those of humans, will include nondeterminism, i.e. there will be states with different transitions triggered by the same event. If this is the case, a randomly chosen transition from the set of enabled transitions will be taken. All external agent statecharts together with the system model statechart can be composed to a super statechart using the concurrency composition. This super statechart may then be analyzed to verify properties in the interaction between external agents and the system.

For example, deadlocks can also occur when some external agent produces an external event e only after the system has generated some action a and, on the other hand, the system waits for the occurrence of e in order to initiate a. Deadlocks of this kind can be detected when analyzing the super statechart.

Reachability of states becomes decidable in the super statechart. Details are beyond the scope of this paper.

5.2 Extending the Scenario Model to a General System Model

Scenarios are related not only by their execution order, but also by data. For example, the outcome of validation in Borrow books depends on the results of previous executions of Borrow books and Return books for the same user.

Therefore, a really complete description of the external behavior of a system requires not only a scenario model, but also a specification of the data and functionality of the system. The latter can be done by an object model that specifies properties and behavior of the objects in the system domain.

The challenge is, whether the two specifications can be integrated into one model, using a common foundation of concepts, notations, and semantics. A single unified model for both scenarios and domain objects has considerable advantages over separate models:

- only one requirements model must be created and maintained,
- more and deeper analyzes and verifications can be made,
- the model can be used as a basis for the generation of full-fledged prototypes,
- a single notation lowers communication barriers between requirements engineers and customers.

My future research will concentrate on the definition of such a universal model, combining the ideas of object models and statecharts.

Statecharts can be extended with object modeling capabilities in different ways (Coleman, Hayes, and Bear 1992, Glinz 1993). The basic idea in my approach is to define a composition hierarchy for objects which is equivalent to that of statecharts: every object is considered to be a state. The state of elementary objects may be refined into a state diagram. Independent objects have concurrent states. Complex objects have an object decomposition which is equivalent to a statechart. Thus, the overall behavior of the object model can be interpreted using statechart semantics.

Figure 11 gives a first impression of how a model that integrates scenarios and domain objects could look.

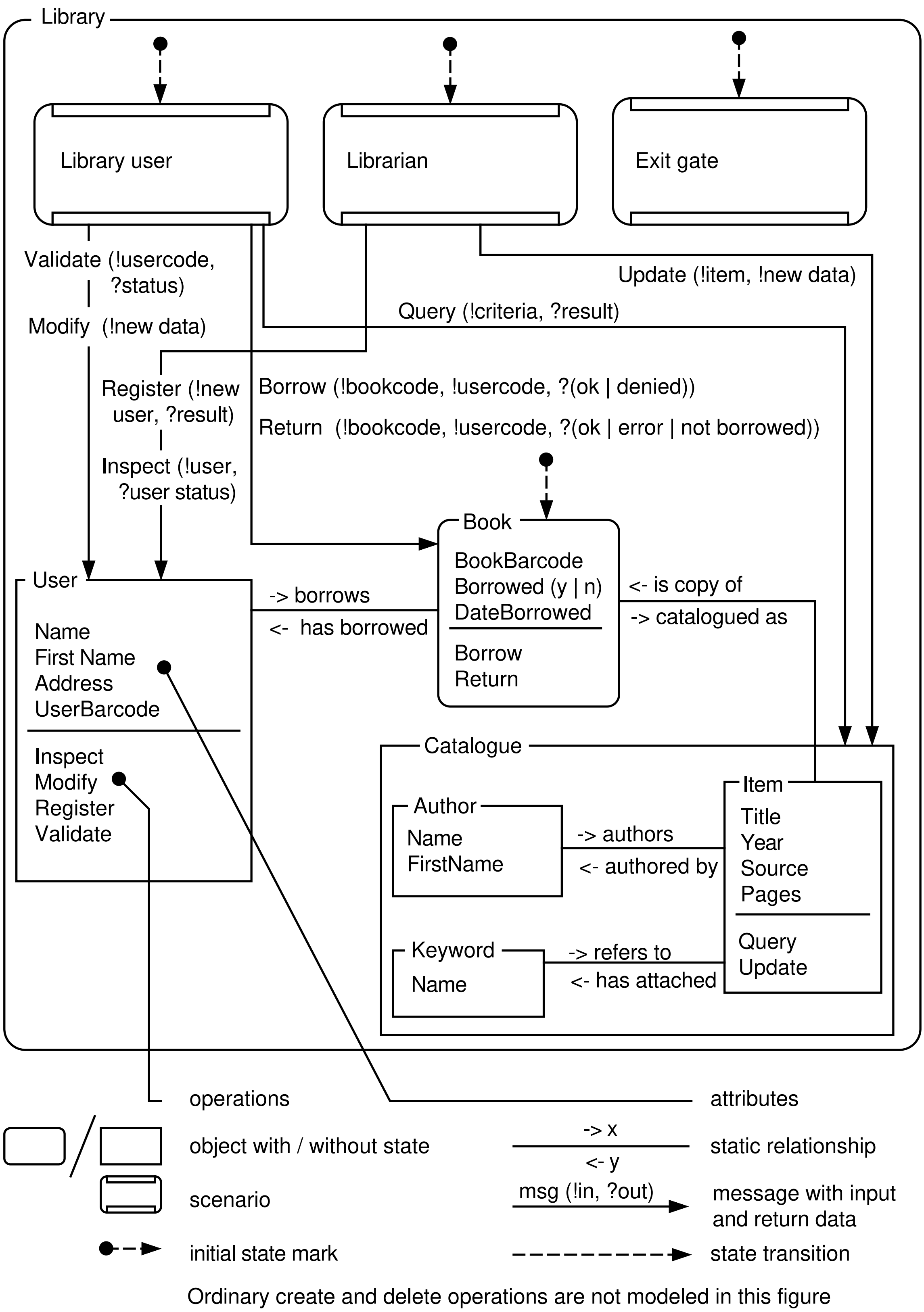


Figure 11. Sketch of a unified system model for library application (see text for further explanation)

I will first give some comments on concepts and notations. Any stored data in a system constitutes a part of the overall system state. However, it does not make sense to represent this data as hierarchies of explicitly modeled states. The number of required states would explode immediately. Instead, data is modeled as attributes of objects. The objects should be interpreted as representatives of classes, i.e. of a set of objects with equal structure and behavior.

We need to represent static relationships, communication (by messages), and state transitions in the same model. In order to clearly distinguish the three, state transitions are now drawn by dotted lines. Furthermore, it is convenient to modify the notation of concurrency, because all domain objects are concurrent. In Figure 11, we assume that at a given level of decomposition, all objects and scenarios with an initial state mark have a concurrent internal behavior.

The potential benefit of integrating scenarios and domain objects in a single model is demonstrated using the process of borrowing a book. The object model describes what happens to a book when it is borrowed (using the objects Book and User and the relationship between them), but does not specify who uses the Borrow operation in which context. The scenario Borrow books describes how borrowing a book looks from a library user's viewpoint, but does not specify the data dependencies between the scenario currently running and previous ones. Both views together yield the complete picture, making the interactions and dependencies between scenarios and objects visible and analyzable.

5.3 Validating the Model

We plan to validate the usability and applicability of the statechart approach to scenario modeling and composition first by re-modeling text-based scenarios from an industrial project. At the same time, we are looking for industrial partners willing to try our approach on their projects. Our role will be to coach these projects and to evaluate the ease as well as the difficulties the persons involved had with this kind of modeling.

6 DISCUSSION

6.1 Why Statecharts?

Principally, any notation that allows the expression of states and state transitions can be used for the representation of scenarios. However, the special features of statecharts concerning parallelism, hierarchy, event handling, and graphic representation make statecharts a particularly appealing choice when compared with other approaches.

With *plain state automata*, the number of states explodes as soon as several parallel scenarios have to be integrated. Therefore, the practical use of state automata is limited to the representation of single scenarios. However, state automata form a sound theoretical basis for all statechart-style approaches.

Programming languages that provide event handling and synchronous parallelism have similar expressive power as the statechart notation presented in this paper. In my opinion, however, a graphical representation of structures is more illustrative than a textual one. It might be a good idea to define both a graphic and a textual notation. The latter could be useful to specify scenarios on a detailed level.

When scenarios are represented by grammars (Hsia et al. 1994), scenario composition can be achieved by nesting and parallel parsing of grammars. The latter requires a special construct for rules that have to be parsed in parallel. However, a grammar notation of an integrated set of scenarios will quickly become unreadable as the set grows in size. Grammars could potentially be used as an internal notation from which suitable external representations (for example, scenario traces) are produced.

Petri Nets are another candidate notation. However, Petri Nets lack the abstraction capabilities of statecharts. Furthermore, the usual requirement of ignoring events that are unsolicitated in the current state forces the analyst to clutter a Petri Net model with a lot of transitions in order to consume these events.

6.2 Notation vs. Method

The approach presented in this paper is primarily a *notation* that allows the representation of a set of scenarios in a unified, consistent model. It is *not* a method how to solve the semantic integration problems that arise when separately developed scenarios have to be brought together. However, it helps to spot these problems. Furthermore, a lot of integration problems can be avoided if a statechart model is used from the beginning to model every scenario as a part of a global model of behavior.
Thus, we have a situation similar to the one in information modeling: there, for example, the entity-relationship model provides an excellent notation for integrated data models, but does not provide a method of achieving the integration of different, separately analyzed views of the data.

6.3 The Role of Abstraction

Due to the abstraction and decomposition properties of statecharts, a complete scenario model does not become a monster of complexity. On any level, details can be omitted. These details can be modeled in separate lower level diagrams or can be omitted completely from the model if considered to be unimportant. Figure 9 shows a model on an abstract level where Figure 8 provides the details for one component of the abstract model. These abstraction capabilities become particularly important when unifying scenario and object models. For example, Figure 11 would be unreadable if the details of Library user, Librarian, and Exit gate had to be modeled in the same diagram.

7 CONCLUSIONS

In this paper, I have generalized the idea of representing scenarios as state automata. Working with statecharts provides us with a well defined formal basis for model construction, analysis, and execution.

The scenario model provides a powerful notation for *documenting* scenarios and for *understanding* their *interrelationships*. The model integrates single scenario models without modifying them. Thus, all constituent scenarios are still visible in the integrated model and can be retrieved as views. Formality in the model allows the *verification* of important properties and contributes to a *better understanding* and *validation* of scenarios.

Furthermore, this approach has the potential for integrating domain object models with scenario models into a single, unified model of system requirements.

REFERENCES

Anderson, J. S., B. Durney (1993). Using Scenarios in Deficiency-Driven Requirements Engineering. *Proc. IEEE Int. Symposium on Requirements Engineering*, San Diego. 134-141.

Coleman, D., F. Hayes, S. Bear (1992). Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering* **18**, 1 (Jan 1992), 9-18.

Glinz, M. (1993). Hierarchische Verhaltensbeschreibung in objektorientierten Systemmodellen – eine Grundlage für modellbasiertes Prototyping. In: Züllighoven, H. et. al. (eds.): *Requirements Engineering '93: Prototyping*. (Report 41 of the German Chapter of the ACM). Stuttgart: Teubner. 175-192. [Hierarchical Description of Behavior in Object-Oriented System Models – a Foundation for Model-Based Prototyping (in German)]

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Sci. Computer Program.* **8** (1987). 231-274.

Harel, D. (1988). On Visual Formalisms. *Communications of the ACM* **31**, 5 (May 1988). 514-530.

Hsia, P., J. Samuel, J. Gao, D. Kung (1994). Formal Approach to Scenario Analysis. *IEEE Software* **11**, 2 (March 1994). 33-41.

Jacobson, I., M. Christerson, P. Jonsson, G. Övergaard (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach.* Amsterdam, etc: Addison-Wesley.

Leveson, N.G. et al. (1994). Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering* **20**, 9 (Sept. 1994). 684-707.

Rubin, K.S., A. Goldberg (1992). Object Behavior Analysis. *Communications of the ACM* **35**, 2 (Sept 1992). 48-62.