On the Importance of Inter-scenario Relationships in Hierarchical State Machine Design

Francis Bordeleau and Jean-Pierre Corriveau

School of Computer Science, Carleton University, Ottawa, Canada
{francis, jeanpier}@scs.carleton.ca

Abstract. One of the most crucial and complicated phases of system development lies in the transition from system behavior (generally specified using scenario models) to the detailed behavior of the interacting components (typically captured by means of communicating hierarchical finite state machines). This transition constitutes the focus of this paper. Our standpoint is that in order to succeed with this transition, it is not sufficient to grasp the details of individual scenarios, but also to understand the relationships that exist between the scenarios. Our claim is that different types of scenario relationships result in different hierarchical state machine structures. We identify and describe four different types of scenario relationships: containment dependency, alternative dependency, temporal dependency, and functional dependency. We also illustrate how such scenario relationships can be expressed in UML using stereotypes and how they guide the design of the corresponding hierarchical state machines.

1 Introduction

In current object-oriented modeling methodologies [1,2], systems are generally modeled using three apparently orthogonal 'views' namely: structure, behavior, and interactions. Semantically, however, these three views overlap considerably [3,4]. That is, they present much of the same conceptual information from different perspectives and with varying levels of detail. In particular, the behavioral view expresses the complete behavior of each component (or class) of a system in terms of states and transitions, typically organized into hierarchical state machines [5]. Conversely, the interactions view captures how system requirements map onto sequences of responsibilities to be executed by components [3,4,6]. Such sequences are generally referred to as *scenarios*. In UML, such scenarios can be captured at different levels of abstraction using use cases, activity diagrams, sequence diagrams, and collaboration diagrams [7]. In our terminology, we use the term *scenario* to refer to one of the different paths of execution captured within a single *use case*. That is, each scenario forms a specific sequence of responsibilities. We argue in section 2 for the importance of a scenario-driven, as opposed to a use case-driven approach to the design of component behavior.

In a scenario-driven approach to object-oriented modeling [e.g., 8], the scenarios of a system constitute the foundation for the design of the complete behavior of the individual components of this system. That is, whereas each scenario illustrates the behavior of (one or more instances of) a component (viz. class) in a specific context, the state machine associated with this component captures the behavior of that component across all the contexts to which (instances of) it participates. We consider the transition needed to go from scenario models to communicating hierarchical state machine models to be one of the most crucial and complex steps of object-oriented system design. Factors that contribute to the complexity of this transition include:

- **Large number of scenarios.** Typical object-oriented systems are composed of very large sets of scenarios, and each component is usually involved in the execution of many different scenarios. A component's behavior needs to handle each of the scenarios in which this component is involved.
- **Concurrency and interactions between scenarios.** This point follows from the previous one. Scenarios can be concurrent and may interact with each other (e.g., one aborts another or is an alternative to it). This is an aspect of object-oriented systems that makes them particularly complex and difficult to design. More specifically, it is not sufficient to address only the temporal ordering of scenarios. In fact, it is their functional (or equivalently, logical) ordering, and in particular their causal ordering, that must be taken into account in the specification of component behaviors.
- **Scenarios of different types.** Object-oriented systems generally implement scenarios of different types. For example, one may categorize scenarios into those that tackle normal executions ('primary' scenarios), and those that capture alternatives ('secondary' scenarios). Another categorization scheme may focus on the functional aspects of scenarios: control, configuration, service interruption, failure, error recovery, maintenance, etc. Identifying such scenario types may impact on the structuring of component behavior.
- **Maintainability and extensibility of components.** Since most industrial systems have a long lifecycle, it is very important to build system components so that they can be easily maintained and extended. Thus, it is not sufficient to define component behaviors that satisfy the current requirements. It is also desirable to define them so that they facilitate future modifications. Indeed, we contend that the structuring of the behavior of a component is one of the most important factors contributing to component maintainability and extensibility. This constitutes an important nonfunctional requirement that must be considered by designers.

The transition between scenario models and hierarchical state machines constitutes the focus of our work [4,9]. Our standpoint is that in order to successfully proceed from scenarios to state machines, it is not sufficient to grasp the details of individual scenarios, but also to understand the relationships that exist *between* these scenarios. We believe that understanding the relationships that exist between a set of scenarios that are to be implemented in a communicating hierarchical state machine model is one of the fundamentals steps in the design of complex object-oriented systems.

Our thesis is that different types of scenario relationships result in different hierarchical state machine structures [*Ibid.*]. More precisely, for a component *x*, the semantic relationship between two scenarios S1 and S2 to which *x* participates should guide the designer in integrating the behavior associated with *x* in S1 with the behavior of *x* in S2. Indeed, we have proposed elsewhere [*Ibid.*] a hierarchy of *behavior integration patterns* used to provide a systematic approach to the transition from scenarios to hierarchical state machines.

In this paper, we sketch out the scenario relationships at the basis of these patterns. We first summarize, in section 2, our pattern-based approach to the transition from scenarios to state machines. We then introduce, in section 3, a simple example used to illustrate the scenario relationships from which this transition proceeds. Each of these relationships is defined and overviewed in section 4. We review other related approaches in section 5 and discuss the issue of automation. We conclude with a brief discussion of some of the advantages of our approach.

2 Proposed Approach

In order to design the (possibly hierarchical) state machine of a complex component, the modeling strategy we propose draws on both the relationships and details of scenarios. The exact notation used for the specification of scenarios (e.g., message sequence charts (MSCs) [10], sequence diagrams [1,7]) does not play a significant role in the use of our integration patterns. However, we remark that, except for Use Case Maps (UCMs) [4,6], few notations *explicitly* capture *inter-scenario* relationships. And yet we have found that such explicit relationships do simplify the application of the integration patterns we suggest.

Our thesis, we repeat, is that, for some component *x*, the integration of a new scenario S1 (in which *x* participates) into the existing state machine *f* of *x* must depend on the pair-wise relationships existing between S1 and the scenarios already handled by *f*. This aspect of our work will be discussed in section 4. For now, we want to summarize the overall approach we are proposing for going from scenarios to state machines. This approach, summarized in Figure 1, consists of three steps:

First, the designer must capture scenarios and their inter-relationships. Recall that a use case may capture a multitude of different scenarios, that is, a multitude of different paths of execution. Thus, a relationship between two use cases expresses a semantic dependency between sets of scenarios, rather than between specific scenarios. Our thesis is that specific inter-scenario relationships play a major role in the design of the hierarchical state machines of the components involved in these scenarios. The exact nature of these relationships is of little import in most existing UML-based modeling processes. In contrast, in the first step of our approach, we are recommending that a designer start by identifying the pair-wise relationships between scenarios. From our standpoint, the exact classification of these relationships is not essential to the application of our strategy; only that each inter-scenario relationship has a specific impact on the design of the state machines associated with the relevant scenario components. In other words, the set of inter-scenario relationships introduced in section 4 is not to be taken as sufficient or complete, but merely as representative. Indeed, we will briefly demonstrate, in that section, that each of the four inter-scenario relationships we put forth has a direct impact on the organization of the state machines of the relevant components.

In UML, scenarios are first captured by means of use cases. The latter are not meant to exist in complete independence from one another: a use case diagram is to capture their relationships, which take the form of unidirectional associations. User-defined stereotypes can be specified for these associations (as for any associations in UML). UML currently proposes four built-in stereotypes [1,7] for use cases, none for their relationships. Moreover, use case relationships are not our primary concern. Instead, we focus firstly on scenario relationships. But use case and scenario relationships clearly overlap semantically.

Second, each use case is refined into a set of diagrams (e.g., sequence diagrams, MSCs) detailing the interactions between the components involved in the scenarios of that use case. In other words, the individual scenarios are now re-expressed in the form of what we will call generically *interaction diagrams*. (The transition between use cases and interaction diagrams may involve several design decisions not discussed here, as well as the use of other models such as class diagrams.) From each such diagram, the designer extracts a state machine for each component that participates in that scenario. We call such state machines *role state machines* for they describe behavior that must be implemented by a component to play a specific role in a specific scenario. From the viewpoint of a single component, we end up with a set of role state machines to integrate into the single state machine of that component.

The third step of our proposal consists in the integration of the role state machines of a component with the existing state machine *f* of that component. In order to integrate a role state machine of scenario S1 with *f*, the designer must establish the pair-wise relationships that exist between S1 and the scenarios already handled by *f*. Each relationship, we repeat, must suggest a specific behavior integration pattern [9].

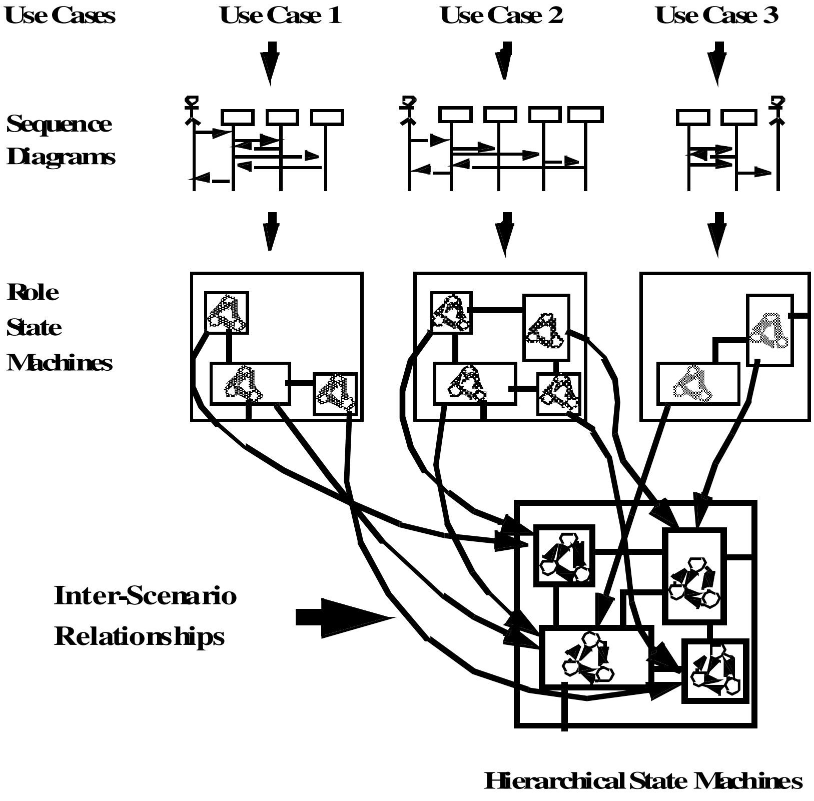

Fig. 1. From Scenarios to Hierarchical State Machines

Fig. 2 gives our current hierarchy of behavior integration patterns (which is discussed at length elsewhere [4]). This hierarchy includes a set of general patterns that can be used as a starting point to define a more complete catalogue of integration patterns in specific development contexts. The proposed 'scenario interaction' design patterns proceed from the temporal and logical relationships that may exist between a role to integrate and the roles already handled by the behavior of a component. For example, with respect to such relationships, two roles of a component (or equivalently their corresponding scenarios) may be mutually exclusive, or one may abort another, or wait for another, or follow another, etc. Conversely, the Mode-Oriented Pattern is one that addresses the structuring of a component's behavior. More specifically, it puts forth an overall hierarchical architecture for the design of a component whose behavior can be thought of as a sequencing of modes.

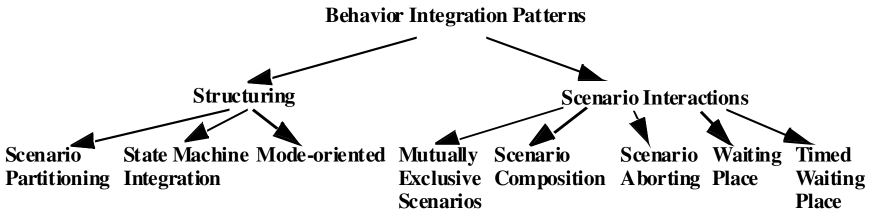

Fig. 2. A Hierarchy of Behavior Integration Patterns

Finally, our catalogue also offers two process patterns: the Scenario Partitioning Pattern and the State Machine Integration Pattern. Such process patterns aim at guiding the designer through the typical steps to be followed when grouping or ungrouping sets of behaviors within a same component. For example, the State Machine Integration Pattern spells out the four steps to use when trying to decide how and where to integrate a new role into an existing hierarchical state machine. (See [4] for more details.)

3 A Simple Example

In this paper, we use a simple Automatic Teller Machine (ATM) system to illustrate the different scenario relationships we introduce in the next section. This ATM system is a conventional one that allows for withdraw, deposit, bill payment, and account update.

The ATM system is composed of a set of geographically distributed ATMs and a Central Bank System (CBS), which is responsible for maintaining client information and accounts; and for authorizing and registering all transactions. Each ATM is composed of an ATM controller, a card reader, a user interface (composed of a display window and a keypad), a cash dispenser, an envelope input slot (used for deposit and bill payments), and a receipt printer. The ATM controller is responsible for controlling the execution of all ATM scenarios, and for communicating with the CBS. For simplicity, we will only consider here the behavior of the ATM Controller.

In this paper, we consider only the following scenarios:

- A start-up scenario that describes the steps required for bringing the system to its operational state. The start-up scenario requires two input messages from the system administrator: a start-up message that triggers the configuration of the ATM, and a start message that starts the ATM. The start message can only be input after completion of the configuration.
- An initial configuration scenario that describes the sequence of steps that are required to properly configure an ATM. These steps include the configuration of each components of the ATM system, and the establishment of a communication with the CBS.
- A *Transaction* scenario that captures the sequence of responsibilities common to all transactions. It includes reading the card information, verifying the PIN (Personal identification Number), getting the user transaction selection, executing the required transaction, printing a receipt, and returning the card.
- One scenario for each of the different types of transactions offered by the ATM system: withdraw, deposit, bill payment, and account update. Each scenario gives the details of a specific transaction, as well as a set of relevant alternatives.
- A shutdown scenario that describes the steps to be carried out when closing down the ATM. The shutdown steps include turning off the different ATM components, and terminating communication with the CBS.

4 On Scenario Relationships and Integration of Behavior

We are suggesting that the exact inter-scenario relationship between two scenarios determine how these scenarios are to be integrated into a hierarchical state machine [4,9]. Our goal here is not to be exhaustive, but instead to demonstrate the importance of each of the four inter-scenario dependency relationships we introduce namely: containment, alternative, temporal and functional.

4.1 Containment Dependency

A containment dependency exists between a scenario S1 and a scenario S2, if S2 is used in the description of S1. Examples of this type of relationship include stubs in UCMs [4,6], and the "uses" relationship defined by Jacobson [12]. This relationship is essential for the development of complex systems as it allows for the recursive decomposition (or composition) of scenarios into other scenarios.

In the ATM example, a containment dependency exists between the Transaction scenario and each of the scenarios corresponding to the steps of that scenario, as illustrated in Fig. 3.

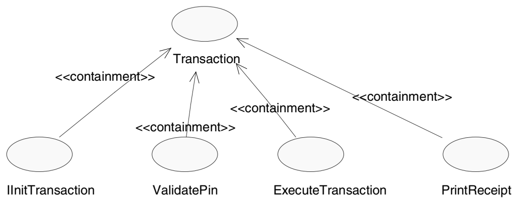

Fig. 3. The Containment Stereotype

At the hierarchical state machine level, such dependencies can be captured explicitly through a simple strategy, namely, defining a state machine for each contained scenario, and placing such state machines as substates of the state corresponding to the container scenario. In this paper, hierarchical state machines are described using the UML notation [1]. In our example, we obtain the following state machine for the **transaction** state (which is to handle a transaction) of the ATM controller (whose top-level state machine will tackle other aspects of the controller, such as starting up and shutting down, as shown in Figure 8 later).

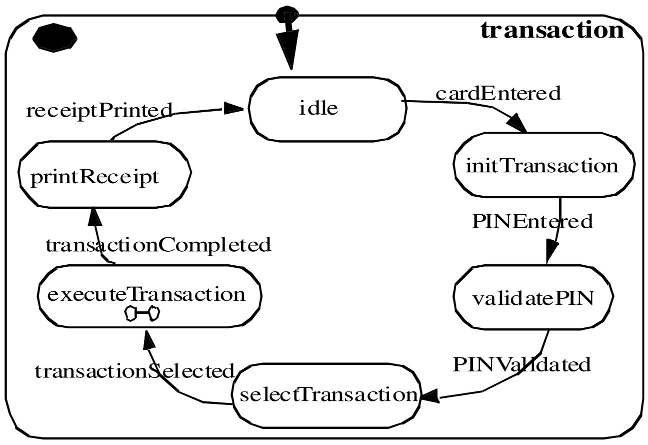

Fig. 4. A Hierarchical State Machine for the Steps of Using an ATM

The containment dependency is a simple inter-scenario relationship that appears to be semantically sufficient for the hierarchical organization of the state machine of a component. In particular, we believe it is unnecessary to introduce a more complex inter-scenario relationship that would somehow involve the concept of inheritance (as Jacobson's *extends* relationship [12], now called *refines* in UML [7]). The semantics of such an "extends" is somewhat problematic (see [11]) and, more importantly from our standpoint, would not directly bear on the design of a state machine.

4.2 Alternative Dependency

A use case outlines "normal mainline of behavior as well as variants such as alternate paths, exception handling [, etc.]" [8]. When a use case is broken down into individual scenarios, it is essential that these alternate paths be semantically tied to the mainline of behavior. For this purpose, we use an inter-scenario relationship we call the *alternative dependency*. In a use case diagram, this relationship is captured using a stereotype of the same name. For example, in our ATM, the *Transaction* scenario must be linked to its alternative, the *Invalid PIN* scenario. To capture this in a use case diagram, it is simplest to have a use case for each scenario and tie these scenarios with the appropriate stereotyped association, as shown in Fig. 5.

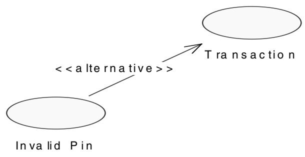

Fig. 5. The Alternative Stereotype

Most importantly, we claim that an alternative dependency suggests a specific strategy to integrate the behavior of a component in the alternate scenario with the behavior of this component for the mainline scenario. The solution typically simply consists in associating transition(s) for the alternate scenario to the state machine handling the mainline behavior. From the existing transaction state machine of the ATM controller (left side of Figure 6), adding the handling of the Invalid Pin scenario simply consists in adding the PINInvalid transitions (as shown in bold on the right side of Figure 6).

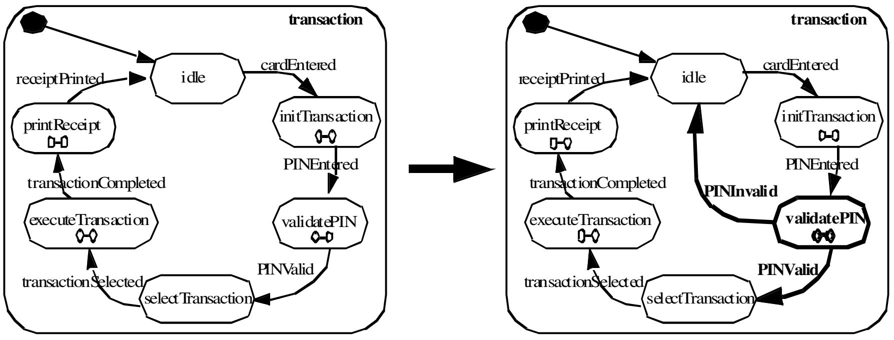

Fig. 6. Integrating the Invalid Pin Scenario into the Transaction Scenario of the State Machine of an ATM

Finally, we remark that, semantically, we found it much easier to deal with alternatives using scenarios, that is, with respect to paths of execution (i.e., sequences of responsibilities), than by referring to use cases. The problem lies in the fact that a use case does not necessarily have a single point of termination, whereas a scenario does. Thus, having a use case as an alternative to another can be quite complicated to understand. (See [11] for further discussion of the semantic difficulties of use-cases.) Conversely, a scenario is an alternative to another if they have a common starting point but distinct end points, much like the different paths of a related path set in Use Case Maps [6].

4.3 Temporal Dependency

This third type of inter-scenario relationship is the strongest we propose from a semantic viewpoint for it allows several specific specializations, which pertain to the temporal ordering of the scenarios (*e.g.*, one scenario *excludes*, *waits for*, *aborts*, *rendezvous* or *joins* another, two scenarios run concurrently, etc.).

As a first example, we want to capture the fact that the shut down scenario must follow the start up scenario (as captured in the partial use case diagram of Fig. 7.).

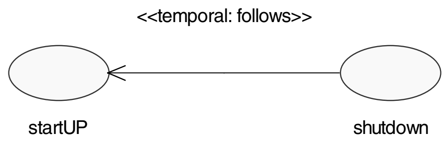

Fig. 7. The Temporal: follows Stereotype

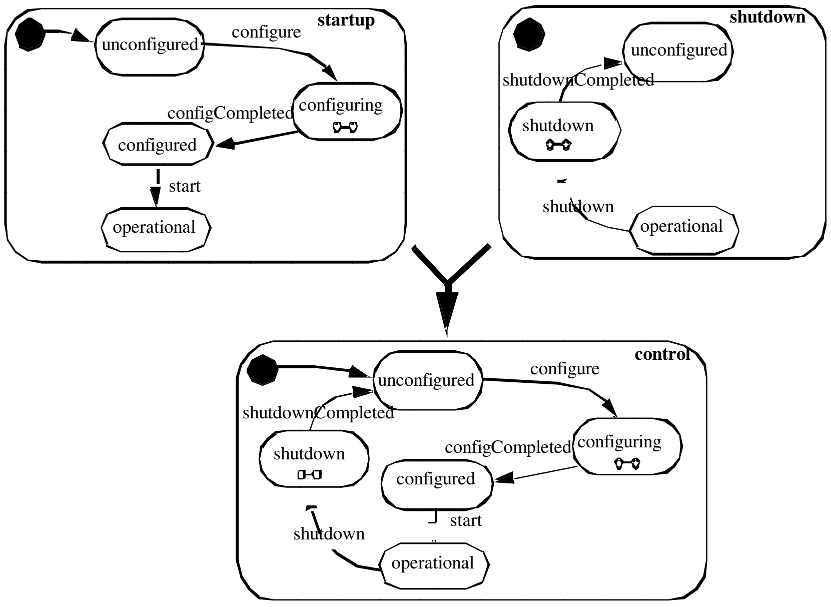

Fig. 8. Control State Machine

The consequences of this relationship on the design of the state machine of the ATM controller are quite direct: each scenario corresponds to a role state machine. These machines must be integrated in such a way that the end of the first one *enables* (via a state, or a precondition, etc.) the start of the second. Here, for example, the end state of *startUp* is the starting state of *shutDown*. The result of integrating the *startUp* and *shutDown* scenarios into a *control* state machine is illustrated in Fig. 8.

The pair-wise specification of such temporal relationships may be cumbersome. For example, we would like to model the fact that the *Withdraw*, *Deposit*, and *Pay Bill* scenarios contained in the *Transaction* scenario are to be mutually exclusive. Though this could be achieved through a multitude of associations, we instead suggest an alternative strategy that draws on another key UML concept, that of a package [see [1, 7]). Here, the “mutual exclusion” temporal relationship may be captured by simply putting the different relevant scenarios in a package, which is itself assigned the stereotype *MutualExclusion.*

Again, the key point to understand is that this temporal *MutualExclusion* inter-scenario relationship has direct impact on the design of the corresponding hierarchical state machine. In our example, this relationship suggests the use of the *Mutually Exclusive Scenario* behavior integration pattern of Figure 2. This pattern organizes the different mutually exclusive scenarios as states (here, of the ATM controller) that are accessed through a choice point (see [7]), as illustrated in Fig. 9.

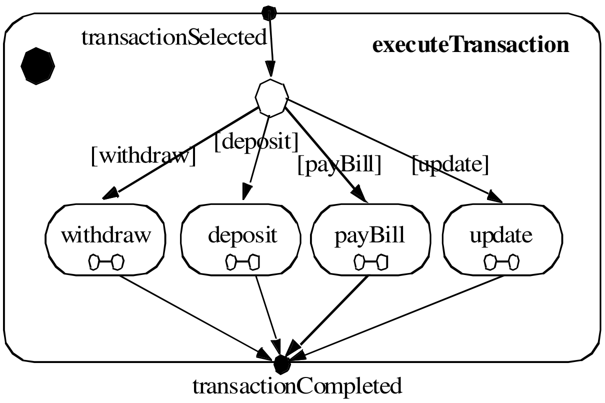

Fig. 9. Mutual Exclusion Example

4.4 Functional Dependency

This last relationship we propose here is used to capture the coexistence of two or more scenarios inside a same conceptual (or logical) regrouping called a *cluster.* This cluster corresponds to a specific ‘aspect’ of the system. Our design experience in real-time event-driven systems suggests regroupings such as control, configuration, communication, error recovery, normal operation, etc. For example, the *start-up* and *shutdown* scenarios are both part of the *control* cluster of the ATM system, and the deposit and withdraw scenarios are both part of the *operational* cluster. Such clusters can be explicitly captured in a use case diagram through the use of packages. More specifically, a set of scenarios (expressed as individual use cases) can be regrouped into a package. In this case, as with mutual exclusion, the use of a package saves us the trouble of defining a multitude of associations between the scenarios.

The functional dependency is the weakest form of dependency between scenarios as it mostly rests on the viewpoint of the designer. But, from a top-down perspective, establishing early functional dependency between scenarios corresponds to the well-accepted design technique called separation of concerns, which is often required for scalability reasons: each aspect (i.e., group/package of scenarios) may be further decomposed into smaller sets of related scenarios. For example, a subset of the control scenarios may be related to system restart (with different types of restart), while another subset may be relate to error handling (including both error detection and error recovery) and yet another may be related to preparing the system for reconfiguration. Furthermore, the set of configuration scenarios may be composed of several distinct subsets of scenarios, each related to a different type of system configuration. And, in our ATM example, even the set of 'normal' operation scenarios (i.e., withdraw, deposit, etc.) may be composed of different types of system functionality. Consequently, even apparently simple component behavior can become rather complex to design when having to address issues such as robustness, reliability and extensibility. And thus, the importance of capturing functional dependency.

Furthermore, as usual, our claim is that such an inter-scenario relationship has direct impact on the design of the relevant state machine(s). More specifically, in order to integrate scenarios contained in different clusters, we use the *State Machine Integration* pattern of 0. This pattern (see [9] and [4] for details) suggests that each aspect in which a component participates be given its own hierarchical state in the behavior of that component. For example, after analyzing the overall set of ATM scenarios, scenarios have been partitioned into four subsets: normal operation, control, maintenance, configuration, and secure communication. The result of the partitioning is given in Fig. 10. (For readability purposes, only some of the scenarios are shown in this figure. See [4] for more details.)

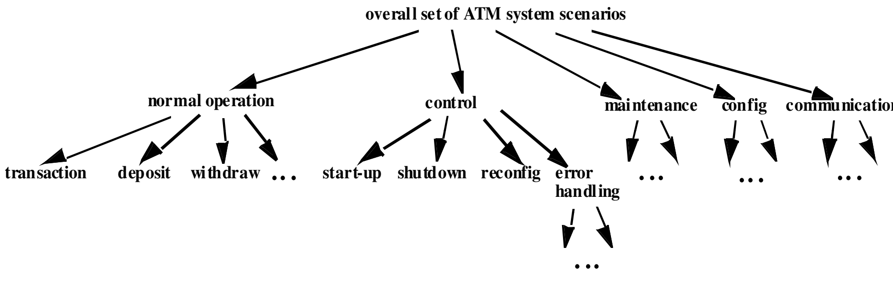

Fig. 10. Scenario clusters for the ATM

Then, state machines are defined on a per cluster basis. For example, in the current case, we separately defined a state machine for the control (Fig. 8) and transaction (Fig. 4) cluster. The final step of the *State Machine Integration* pattern consists in integrating the state machines related to the different clusters in a single hierarchical state machine. This last step is illustrated in Fig. 11 where the control state machine and the transaction state machine (top part of Fig. 11) are integrated into a single hierarchical state machine given in the bottom part of Fig. 11: the transaction state machine becomes the state machine of the operational state of the control state machine. For this purpose, the operational state of the control state machine is modified to become a composite state.

The actual integration is more complex than what is suggested by this figure. When integrating the state machines corresponding to different scenarios, it is essential to verify (e.g., through pre- and post-conditions) that the inner workings of each scenario are preserved.

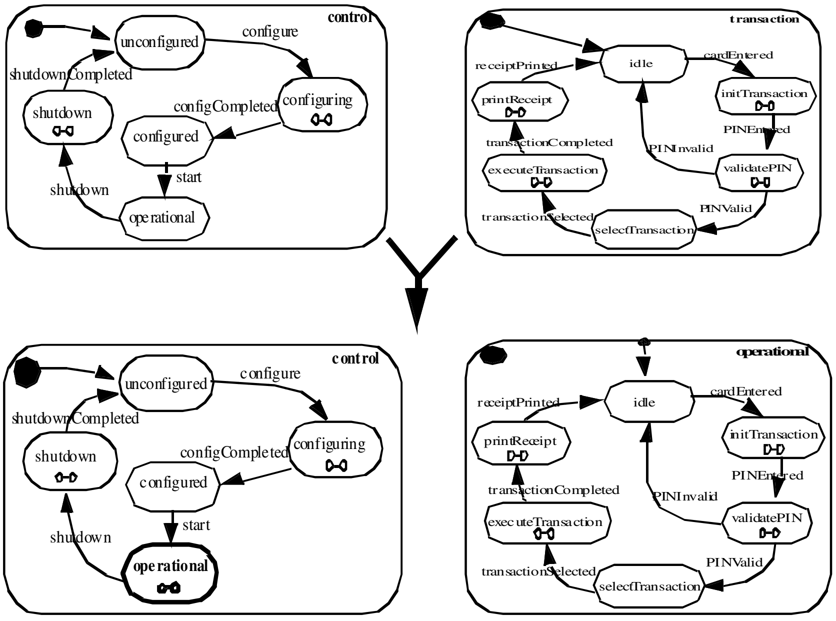

Fig. 11. Scenario Integration Pattern

5 Comparison with Other Approaches

In the current literature, several papers address the problem of defining state machines from different types of interaction diagrams (i.e., scenario models). But most, if not al, ignore scenario inter-relationships. In [13], Haugen discusses a general approach for making the transition between an MSC model [14] and an SDL model [15]. Other papers ([16, 17, 20]) propose an automatic synthesis approach. That is, they aim at explaining how to automatically generate state machines from different types of interaction diagrams. For example, the work of Koskimies and Makinen [16] focuses on the definition of a transition between interaction diagrams and flat (i.e., non-hierarchical) state machines. It defines a synthesis algorithm for the automatic generation of such state machines from a set of sequence diagrams. Leue and Mehrmann [17] offer a similar approach between MSC'96 diagrams [18] and ROOM [19] hierarchical state machines. Also, Harel and Kugler [20] present an automatic synthesis approach centered on the use of live sequence charts (LSCs), a specification technique they define as an extension of interaction diagrams. LCSs are to be used to specify both mandatory and forbidden scenarios.

In contrast, our work centers on the design of hierarchical finite state machines from scenarios *and their relationships*. We believe that automation is ill-suited for such a task. Let us elaborate. Contra synthesis approaches, we recall that, generally, scenario models are incomplete. Thus, the corresponding state machines may typically have to be augmented with states and/or transitions in order to account for behavior not explicitly modeled in the scenarios. In other words, most often, state machines are not semantically equivalent to the scenarios from which they proceed, but in fact constitute ‘semantic extensions’ to these scenarios. But the very nature of state machines makes them difficult to modify: for example, one new transition may generate indeterminism. Thus, we repeat, it is essential to immediately structure component behavior for extensibility. And, in our opinion, this is not achieved in synthesis approaches. That is, we believe that an automatically generated state machine is seldom ‘semantically customizable’. This is especially true for flat state machines.

The generation of a flat state machine from some algorithm is undoubtedly useful for activities such as validation and verification. However, we find it absolutely crucial to use hierarchical state machines for designing components: hierarchical state machines not only enable iterative development of the behavior of a component, but also promote its understandability, testability and scalability. Whereas a flat state machine, regardless of its number of states, is suitable for exhaustive path analysis, hierarchical state machines allow for the clustering of different behavioral facets of a component (often referred to as roles) into distinct hierarchical states of this component. Such a separation of concern is required for the understanding of complex behavior, and for its evolution, two essential characteristics of a good design. Put another way, we need hierarchical states to decouple clusters of states from each other in order to i) understand their corresponding roles and ii) ease the evolution of each such role (in particular with respect to behavior not explicitly captured in scenario models).

Subsuming this argumentation, we find our conviction that component design is an act of creation guided by a multitude of decisions typically lost in the automatic generation of a (typically flat) state machine from some other (often incomplete) model or specification.

The idea of patterns ([22, 23, 24]) has been put forth in several subfields of software engineering (e.g., analysis, architecture, design, testing) as a means to document typical sets of decisions. Work on behavioral patterns (e.g., Gamma et al. [23]) has mostly focus on inter-component behavior. Indeed, few researchers have focused on patterns for the design of the behavior of a component. Douglass's work on dynamic modeling (chapter 12 of [25]) is a noticeable exception: he is concerned with “useful ways to structure states rather than objects” (Ibid., p.588). And he clearly advocates hierarchical state machines to do this: “many of the state patterns use orthogonal components to separate out independent aspects of the state machine.” (Ibid., p.589).

The 18 state behavior patterns put forth by Douglass form a catalogue of typical behaviors for a component. Let us consider, for example, the Polling State Pattern. “Many problems require active, periodic data acquisition. The object processing this data often has states associated with the problem semantics that are unrelated to the acquisition of data... The Polling State Pattern solves this problem by organizing the state machine into two distinct orthogonal components. The first acquires the data periodically. The second component manages the object state... This allows data handling to scale up to more complex state behavior.” (Ibid., p.593).

Such a description, and in particular the need for hierarchical states and the goal of conventionalizing behavior, emphasizes the closeness of Douglass's work to ours. But there is a significant difference between our approaches: his catalogue of patterns proceeds from his identification of a set of typical behaviors (viz., detailed roles) often encountered in components of object-oriented systems (e.g., polling, waiting rendezvous, balking rendezvous, timed rendezvous, random state, etc.). In contrast, we do not try to catalogue typical behaviors of a component but instead conventionalize the transition from scenarios to hierarchical state machines. More precisely, we start with a set of scenarios that express (albeit perhaps partly implicitly) the behavior of their components. We do not assume that the behavior of a component is necessarily typical, that is, that it matches a specific detailed role. Instead, we assemble the behavior of a component from the scenarios in which it is used, leaving room for the combination of detailed roles within a same component, as well as for semantic customization. Furthermore, our approach is iterative: we do not require that all relevant scenarios be known before starting to design the behavior of a component. Instead, new roles can be progressively integrated into the behavior of an existing component.

6 Conclusions

In this paper, we have presented a three-step approach to a systematic transition from scenarios to hierarchical state machines. This approach rests on the definition of inter-scenario relationships (such as containment, alternative, temporal and functional dependency) and of the corresponding families of behavior integration patterns, which we have very briefly introduced but discuss at length elsewhere ([4]).

We are currently working on formalizing scenario temporal and functional dependency. For the former, existing work on temporal logic and notations such as LOTOS [25] serves as a starting point. Our goal is to focus on temporal expressions that have direct impact on state machine design. Similarly, for functional dependency, existing work on causal relationships (in particular in Artificial Intelligence) is scrutinized in order to find forms that bear on state machine design.

Industrial experience with the approach we propose suggests the following benefits:

- Reducing the time required to design complex component behavior
- Increasing the quality of the design of complex component behavior
- Reducing the time required to test complex component behavior
- Reducing undesired scenario interactions
- Increasing traceability between scenarios and detailed component behavior

To conclude, let us remark that the approach described in this paper is constantly evolving as a result of its use in industry, as well as in academia. In particular, our method is followed by the Versatel group of CML Technologies, as well as by our software engineering students at Carleton University. It is also at the basis of a recent university-industry NSERC funding initiative with Nortel Networks to define a standard development process in the context of the Wireless Intelligent Networkss (WIN) standard definition.

References

1. Object Management Group: OMG Unified Modeling Language Specification.
2. Henderson-Sellers, B. and Firesmith, D.: The Open Modeling Language Reference Manual, Cambridge University Press, 1998.

3. Corriveau, J.-P.: Traceability and Process for Large OO projects, IEEE Computer, pp. 63-68, 1996 (also available in a longer version in: Technology of Object-Oriented Languages and Systems (TOOLS-USA-96), Santa-Barbara, August 1996.
4. Bordeleau, F.: A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines. Ph.D. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1999. (Available at http//www.scs.carleton.ca/~francis)
5. Harel, D.: StateCharts: A Visual Formalism for Complex Systems, Science of Computer Programming, Vol. 8, pp. 231-274, 1987.
6. Buhr, R. J. A. and Casselman R. S.: Use Case Maps for Object-Oriented Systems, Prentice Hall, 1996.
7. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.
8. Kruchten, P.: The Rational Unified Process: An Introduction, Addison-Wesley, 1999.
9. Bordeleau, F., Corriveau, J.-P., Selic, B.: A Scenario-Based Approach for Hierarchical State Machine Design, Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'2000), Newport Beach, California, March 15 - 17, 2000.
10. ITU (1996) : Message Sequence Charts (MSC'96), Recommendation Z.120, Geneva.
11. Firesmith, D.: The Pros and Cons of Use Cases, ROAD, May 1996.
12. Jacobson, I. et al.: Object Oriented Software Engineering, Addison-Wesley, 1994.
13. Haugen, O.: MSC Methodology, SISU II Report L-1313-7, Oslo, Norway, 1994.
14. ITU (1993) : Message Sequence Charts (MSC'93). Recommendation Z.120, Geneva.
15. ITU (1992): Specification and Description Language (SDL'92), Recommendation Z.100, Geneva.
16. Koskimies, K., Makinen, E.: Automatic Synthesis of State Machines from Trace Diagrams, Software-Practice and Experience, vol. 24, No. 7, pp. 643-658 (July 1994).
17. Leue, S., Mehrmann, L., Rezai M.: Synthesizing ROOM Models From Message Sequence Charts Specifications, TR98-06, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, 1998.
18. ITU (1996) : Message Sequence Charts ('96), Recommendation Z.120, Geneva.
19. Selic, B., Gullickson, G., and Ward, P.T.: Real-time Object-Oriented Modeling, Wiley, 1994.
20. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications, Proceedings of Fifth International Conference on Implementation and Application of Automata (CIAA2000), Le
21. tutre Notes in Computer Science, Springer-Verlag, July 2000.
22. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: A System of Patterns, Wiley, 1996.
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns-Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
24. Vlissides, J., Coplien, J.O., Kerth, N.L.: Pattern Languages of Program Design, Addison-Wesley, 1996.
25. Douglass, B.: Doing Hard Time. Addison-Wesley, 2000.
26. ISO LOTOS standard: IS 8807 (http://www.iso.ch/cate/d16258.html)