A Scenario-Based Approach to Hierarchical State Machine Design

Francis Bordeleau
School of Computer Science,
Carleton University,
Ottawa, Canada
francis@scs.carleton.ca

Jean-Pierre Corriveau
School of Computer Science,
Carleton University,
Ottawa, Canada
jeanpier@scs.carleton.ca

Bran Selic
ObjecTime Limited,
Kanata, Canada
bran@objectime.com

Abstract

One of the most crucial and complicated phases of real-time system development lies in the transition from system behavior (generally specified using scenario models) to the behavior of interacting components (typically captured by means of communicating hierarchical finite state machines). It is commonly accepted that a systematic approach is required for this transition. In this paper, we overview such an approach, which we root in a hierarchy of "behavior integration patterns" we have elaborated. The proposed patterns guide the structuring of a component's behavior, and help in integrating the behavior associated with new scenarios into the existing hierarchical finite state machine of a component. One of these patterns is discussed at length here.

1. Introduction

Scenario models [1,2,10,11,12,16,18] and communicating hierarchical state machine models [2,9,18,20,22] provide two orthogonal views of real-time systems [18,20,22]. The former view describes system behavior as sequences of responsibilities that need to be executed by components in order to achieve overall system objectives, whereas the latter expresses complete component behavior in terms of states and transitions.

One of the most crucial and complex phases of real-time system design lies in the transition that is required to go from system behavior (defined by means of a set of scenario models) to component behaviors (described by means of communicating hierarchical state machine models) [20,22]. Several factors (*e.g.*, the large number of scenarios, the concurrency and interactions between scenarios [22,24], the unpredictability of external events, *etc.*) contribute to the complexity of this transition. Furthermore, since most industrial systems generally have a long lifecycle, it is very important to build system components that can be easily maintained, reused and extended. It is well accepted that this latter demand further complicates the structuring of the behavior of a component. Also, designers must address nonfunctional requirements, such as performance and robustness. Thus, when considering all these difficult issues, it is generally acknowledged that a systematic approach is required for the specification of the behavior of a component.

In the current literature, some papers (*e.g.*, [12,13,16]), define methods based on *synthesis* algorithms that perform automatic generation of state machines from a set of interaction diagrams. Such methods, in theory, completely automate the transition between interaction diagrams and state machines. Their main advantage, beyond automatic generation of state machines, consists in having scenario models and state machine models in complete semantic synchrony. However, these algorithms do not consider several important design issues such as scenario interactions and state machine structuring; issues, we repeat, directly relevant to component maintainability and reusability. Moreover, none of the existing methods have yet solved the problem of automatically integrating concurrent scenarios in the general case.

In this paper, we propose a different approach to the specification of the behavior of a component, one rooted in the definition of *behavior integration* patterns. Let us elaborate.

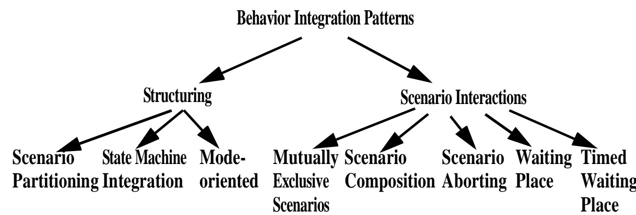
The use of *design* patterns (e.g., [4,5,8,23]) has rapidly increased in industry in the last few years. The "patterns approach" consists in defining a set of solutions that can be applied by designers when facing specific design problems. Patterns can be classified in terms of their application domain, the aspect of system development that they address, and the level of abstraction at which they can be applied.

With respect to application domains, patterns have been defined for both general design problems, and for problems that are specific to some application domains, such as real-time systems (including concurrent and distributed systems) [14,15,19,23,24], CORBA [19] and avionics [14]. With respect to the different facets of system development, patterns have been defined to address enterprise design, process and organization [23], system design [6,8], and

software design [5,6,23]. And, finally, with respect to levels of abstraction, patterns have been defined at the architectural [21] and structural level [5,8,23], behavioral level (*Ibid*.) and programming level [5,8,14,15,19].

However, to the best of our knowledge, there exists no patterns that address the difficult problem of integrating a set of possibly concurrent and interacting scenarios into a set of component behaviors. In this paper, we describe one of the several behavior integration patterns we have identified [2] to help designers define communicating hierarchical state machines from scenario models. Figure 1 gives an overview of these patterns.

Figure 1. Hierarchy of behavior integration patterns



We will focus here on the *State Machine Integration* pattern.

This paper is structured as follows. In section 2, we describe the Automatic Teller Machine (ATM) system used to illustrate. In section 3, we describe our general approach for hierarchical state machine design. In section 4, we describe the State Machine Integration pattern. Finally, in section 5, we summarize and discuss our experience of using the pattern approach decsribed here in an industrial context.

2. Example: an ATM system

In this presentation, an Automatic Teller Machine (ATM) system is used to illustrate the *State Machine Integration* pattern. 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 a ATM controller, a card reader, a user interface (composed of a display window and a keypad), a cash dispenser, an envelop 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 this paper, we consider the following scenarios:

- A start-up scenario that describes the steps required to bring the system to its operational state. These steps include the configuration of each components of the ATM system, and the establishment of a communication with the CBS.
- An abstract *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 transaction offered by the ATM system: withdraw, deposit, bill payment, and account update. Each of these scenarios gives the details of a specific transaction, as well as a set of relevant alternative scenarios.
- A shutdown scenario that describes the steps to be carried out when closing down the ATM. The shut down steps includes turning off the different ATM components, and terminating communication with the CBS.

3. Proposed Approach

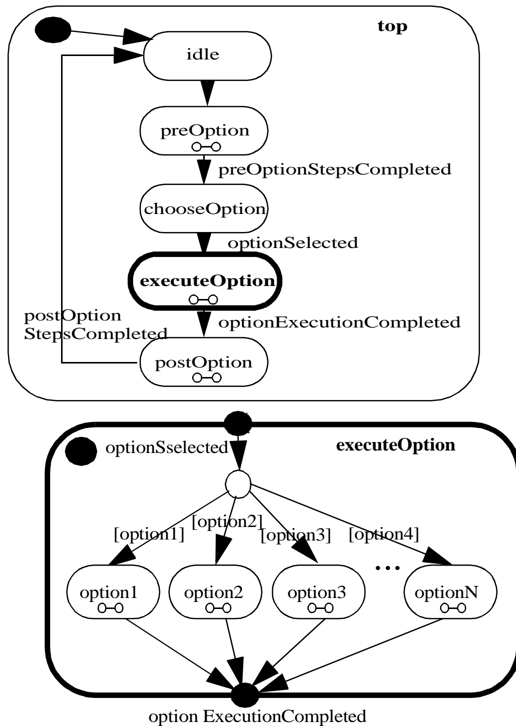
In order to design the hierarchical state machine of a complex component, the behavior integration patterns we propose draw on both the inter-scenario relationships and details contained in scenario models such as interaction diagrams [11]. We remark that, except for Use Case Maps (UCMs) [2,3,17], few notations *explicitly* capture inter-scenario relationships. For detailed interaction diagrams, we use Message Sequence Charts [10] (MSCs). However, the patterns we have developed [2] are not dependent on these two specific notations, but rather on the semantics of scenarios. Let us elaborate.

We claim that the integration of a new scenario S1 into an existing hierarchical state machine *f* must depend on the pair-wise relationships existing between S1 and the scenarios already handled by *f*. We identify three important types of inter-scenario relationships:

- The *Scenario Interaction* relationship. This type of relationship is the strongest of the three from a semantic viewpoint. It exists between scenarios that interact in a specific manner (*e.g.*, one scenario *excludes*, *waits for*, *aborts*, *rendezvous* or *joins* another). These specific interactions, we repeat, can be captured in UCMs. We suggest that the exact interaction relationship between two scenarios determines how these scenarios are to be integrated into a hierarchical state machine [2]. For example, if two scenarios are taken to be mutually exclusive, then they will be integrated using the *Mutually Exclusive Scenario* pattern, which suggests how to

organize the relevant hierarchical state machine[1] (using a *choice point*) (*Ibid.*). This pattern is summarized in Figure 2 below. In this figure, the right-hand side state machine (*executeOption*) illustrates the part of the hierarchical state machine that ensures mutual exclusion between the scenarios that correspond to the different options (each option scenario is encapsulated in an *option* composite state), while the left-hand side state machine (*top*) illustrates the higher level state machine that leads to the execution of an option.

Figure 2. Structure of the scenario option state machine

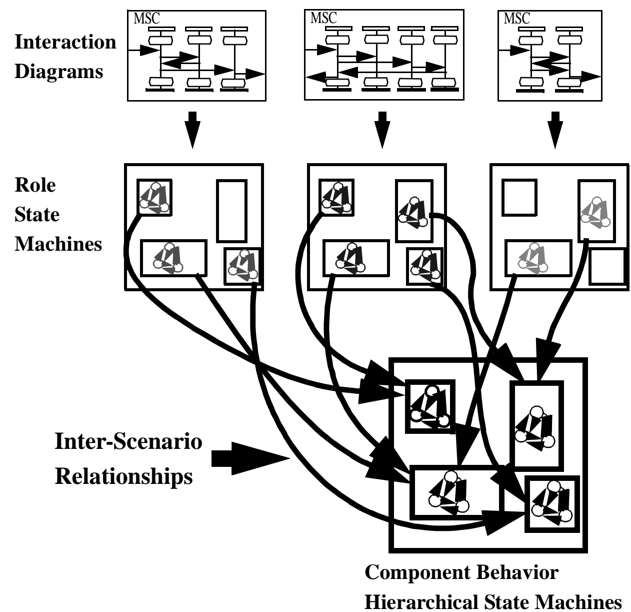


- The *scenario dependency* relationship. A scenario dependency relationship exists between a scenario S1 and a scenario S2, if scenario S2 is used in the description of S1. Examples of this type of relationship include stubs in UCMs [2,3], and the "uses" and "refines" relationships defined by Jacobson [11]. In the ATM system, such a relationship exists between the abstract *Transaction* scenario and each of the scenarios that correspond a specific transaction (i.e. withdraw, deposit, bill payment, and account update).
- The *scenario clustering* relationship. This relationship is used to capture the coexistence of two or more scenarios inside a same conceptual regrouping called a *cluster*. This regrouping corresponds to a specific 'aspect' of the system. At this point in time, aspects that we have observed to lead to such regrouping include 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.

1. In this paper, hierarchical state machines are described using the UML notation [20].

The proposed integration patterns define the behavior of components using a two-step approach. First, we use the details of message sequence charts [10] (MSCs) to define state machines on a per scenario basis. The resulting state machines are called *role state machines* as they describe behavior that must implemented by a component to play a specific role in a scenario. Second, we consider the inter-scenario relationship information to compose the state machines obtained in the first step into more complex hierarchical state machines. This approach is summarized in Figure 3.

Figure 3. From a Set of Scenarios to a Set of Component Behaviors



4. The State Machine Integration Pattern

The *State Machine Integration Pattern* specifically addresses the design of a hierarchical state machine from a set of simpler state machines. That is, the main issue that needs to be resolved in this pattern is the one of integrating a set of state machines associated with different scenarios, into a single hierarchical state machine. Our presentation roughly follows the style of Gamma *et al.* [8].

Motivation

New scenarios constantly need to be integrated in existing systems to satisfy new requirements. From our perspective, the integration of a new scenario in a system results in the integration of a new set of role state machines in the system. That is, the component behavior of several components will need to be modified to handle new sequences of actions, new sequences defined by means of what we call role state machines [2]. For this reason, state machine integration constitutes a main issue in real-time system design.

Problem

Integrating a set of new role state machines into an existing component behavior is a difficult task. Often, the structure of the hierarchical state machine of a component is not capable of adapting to the integration of new scenarios. That is, if the component behavior is not properly structured, the integration of a new scenario may require a major restructuring of the component's hierarchical state machine. The cost of major restructuring is very high both in terms of time and possible new errors. Indeed, when restructuring a state machine, designers must ensure that the resulting component behavior can still correctly execute all the scenarios already integrated in the component. Unless a systematic approach is used, major restructuring usually entails major regression testing.

We have observed that problems in the structuring of hierarchical state machines often result from a lack of understanding of scenario relationships, or a lack of a systematic approach in expressing those scenario relationships in terms of state machine constructs. And thus, the integration of a new state machine by a designer who does not have a good understanding of the overall system behavior often results in a component's state machine that is difficult to maintain and extend. Conversely, if a component's state machine is properly structured, the impact of integrating a new scenario should be limited to a well-defined subset of the overall hierarchical state machine of that component.

Applicability

The State Machine Integration pattern can be used for the design of hierarchical state machines in any type of component that is composed of a set of existing state machines.

Forces

- Allow for the design of component behavior from existing state machines.
- Structure hierarchical state machines (i.e., component behavior) so that the integration of new role state machines is performed at minimal cost.
- Provide a scaleable integration strategy: the cost of maintaining the component and the cost of adding new scenarios (i.e., role state machines) should not increase exponentially as the size of the component grows.
- Promote high cohesion of the scenarios addressed in a composite state.
- Maintain traceability between state machine structure and scenarios [7].
- Increase component behavior maintainability and extensibility by structuring hierarchical state machines in a way that is consistent with the partitioning of scenarios over several design iterations.

Solution

The State Machine Integration pattern defines a component's behavior as a set of integrated simpler state machines, each of which is associated with a set of scenarios. More specifically, the approach taken in this pattern is similar to the approach used in system structure design, where a system is defined as a set of communicating components. In the case of system structure, the overall behavior of the system is the result of component integration. In this pattern, a hierarchical state machine is defined as a set of simpler state machines, where each state machine implements a set of scenarios. In this case, the overall behavior of a component is the result of state machine integration.

The starting point for the application of this pattern is a set of role state machines, each of which being associated with a specific scenario[1], that must be integrated in an existing component's hierarchical state machine. Because the *control* state machine controls the ability of a component to perform all other functions, the top level of the component's hierarchical state machine is typically taken up by the *control* state machine. Then, the other state machines, like the *normal operation* state machine or the *configuration* state machine, are integrated in the appropriate composite state of the *control* state machine. (The State Machine Integration pattern is completely recursive.)

The guiding principle behind this pattern consists in separating the two important aspects of scenario models: individual scenario description, and inter-scenario relationships (discussed above). The structure of a component's hierarchical state machine is then defined so that it reflects the relationships between scenarios. Thus, the State Machine Integration pattern can be described as a four steps process:

1. Each role state machine is associated with (that is, is an implementation of) a specific scenario. Therefore, there exists a one-to-one relationship between a role state machine and a scenario. In the description of the current pattern, we use the terms *role state machine* or *scenario* depending on the aspect we want to emphasize.

1. Analyze the relationships between the scenarios that are to be integrated and the ones already implemented in the system.
2. Determine *where*, i.e. in which state, in a component's hierarchical state machine, the role state machine associated with a new scenario must be integrated. The decision proceeds from the analysis of the relationships between the scenario corresponding to a role state machine to integrate and the scenarios currently handled by the existing state machine.
3. Having established the state *s* in which to integrate the role state machine *m* and the relevant inter-scenario relationships, use these relationships to integrate *s* in *m*. Recall that a specific relationship lead to specific integration strategy (see Figure 1.). The application of such a strategy produces a new hierarchical state machine in which the current role state machine has been integrated.
4. Before proceeding to the integration of the next role state machine, some testing must be performed, as well as some possible restructuring. More specifically, it is imperative to verify i) that past and new scenarios of the component are still handled correctly and II) that no unwanted interactions between scenarios have developed. Such a verification is not easy due to the nondeterministic nature of events. As for restructuring, it may involve grouping a set of states into a composite state, defining state entry or exit actions, distributing the set of actions executed on a transition over a set of transition segments, etc. (see [2] for further details).

Example

We now apply the State Machine Integration pattern to the design of the ATM controller component. This component is responsible for controlling all aspects of the ATM. It coordinates the work of the different ATM components, and communicates with the CBS to carry out transactions.

For the purpose of illustrating the State Machine Integration pattern, we consider the two main control scenarios: *startup* and *shutdown*, and the set of *transaction* scenarios. The integration of other scenarios can be carried out in a similar manner.

The design of the component behavior of the ATM controller is conducted in three phases:

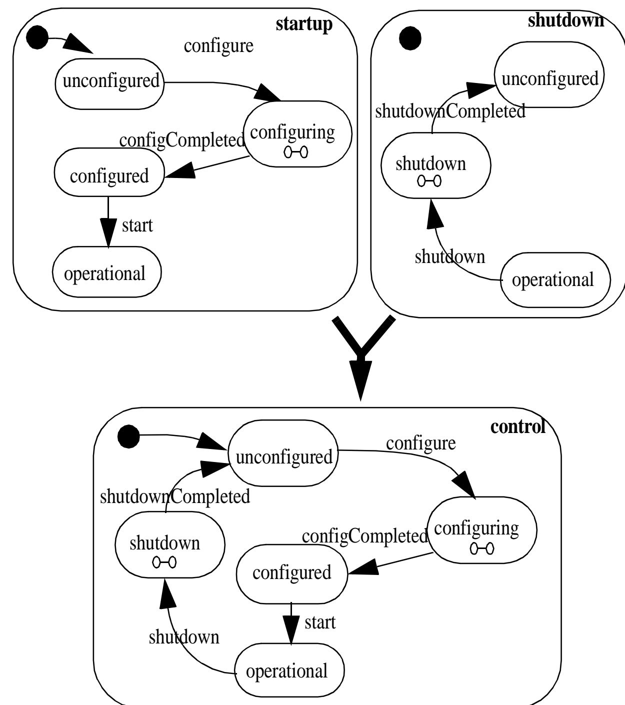
1. Definition of the control state machine.
2. Definition of the operational state machine.
3. Integration of the two state machines in a single one.

Integration of Control Scenarios

First, we integrate the two role state machines associated with the control scenarios: the *startup* state machine and the *shutdown* state machine. These state machines are illustrated in the top part of Figure 4. Both of these state machines are hierarchical state machines. In the *startup* state machine, the details of the startup configuration are encapsulated in the *configuring* composite state. Similarly in the *shutdown* state machine, the details of the shutdown scenario are encapsulated in the *shutdown* composite state.

In this case, the integration of the state machines is rather simple since the end state of one scenario is the initial state of the other scenario. The result of the integration of the two state machines is illustrated in the bottom part of Figure 4. The resulting state machine is a very general one that could be used for different types of systems.

Figure 4. Integration of Control Scenarios



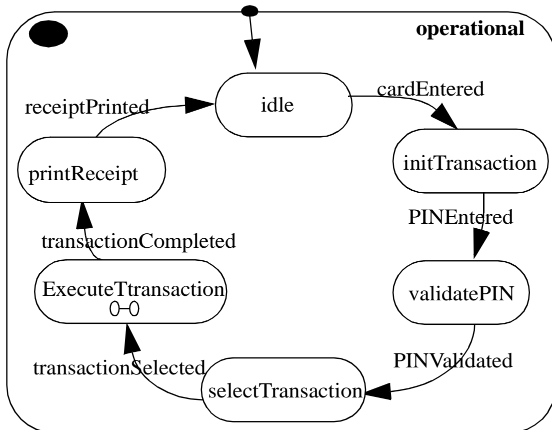
ATM Transactions

Second, we build a state machine for the operational aspect of the ATM controller. This state machine encapsulates all the ATM transaction scenarios. In this case, the general transaction scenario is a high-level scenario that uses the other transaction scenarios for the purpose of specific transactions. For this reason, we establish a "uses" relationship between the general transaction scenario and the other transaction scenarios. This is reflected at the hierarchical state machine level by the definition of a *transaction* composite state in the general transaction state machine. This composite state encapsulates the whole set of transaction scenarios.

The general transaction scenario could be described by the *operational* state machine given in Figure 5. This state machine describes the steps that are common to all transactions. We observe that this state machine is defined inde-

pendently of any particular transactions. This allows reusing this state machine in different versions of the ATM that offer different types of transactions.

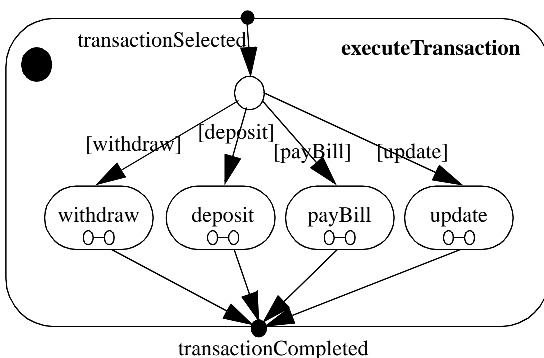
Figure 5. Operational State



At the level of the transactions, the state machine of Figure 6 could be defined. This state machine, called *executeTransaction*, contains a set of composite states that correspond to the different types of transactions offered by the system. Each of these composite states encapsulates a state machine that describes the steps required for a transaction. The *executeTransaction* state machine is designed using the *Mutually Exclusive Scenario* pattern. Recall that this pattern ensures that the scenarios can only be executed one at a time. Once defined, this state machine (Figure 6) is placed in the *executeTransaction* composite state of the *operational* state (of Figure 5).

The result is a concrete *operational* state machine that encapsulates all the transaction related behavior of an ATM. This state machine is completely independent of the *control* state machine previously defined. Therefore, it could be used with different control level state machines.

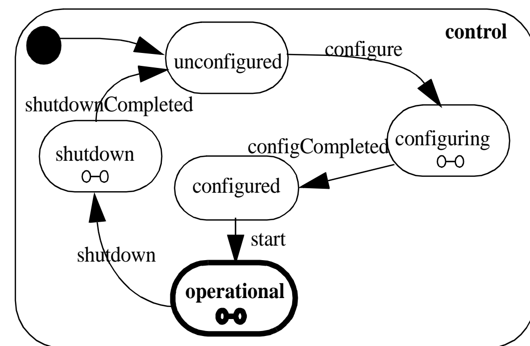
Figure 6. Transaction State



Integration of ATM Transactions in the Top Level Control State Machine

Finally, we place the *operational* state machine defined in Figure 5, in the *operational* state of the top level *control* state machine (lower half of Figure 4). As a result, the *operational* state of the *control* state machine is modified to become a composite state. The resulting control state machine is illustrated in Figure 7.

Figure 7. ATM Controller Component Behavior



Consequences

- This form of component behavior design promotes the reuse of existing state machines. First, designing complex component behavior is taken to proceed from assembling together simpler state machines, as illustrated above. Second, these simpler state machines ideally become reusable behavioral components. For example, a generic *control* state machine (lower half of Figure 4) can be used in the design of components that significantly differ at the functional level. Similarly, a state machine that defines the steps required to establish a telephone connection (e.g., with the CBS) could be reused in many systems that need to establish such a connection.
- In turn, this approach encourages the development of libraries of specialized (and well-documented) state machines to be used for the definition of complex component behavior. The main advantage of this strategy is that designers can benefit from the existence of state machines that have already been designed and tested by others in different contexts. In other words, designers benefit from the experience of other designers.
- Reuse of existing state machines also facilitates customization. For example, in the context of the ATM system, a “fast cash” ATM machine (i.e., a machine that only allows for withdraw and account update) can easily be assembled from existing state machines, with only minor modifications.

- Because it associates a role state machine with each scenario, the State Machine Integration pattern allows establishing a strong traceability relation [7,11] between the structure of a hierarchical state machine and the scenarios that it must satisfy. Also other patterns defined in [2] (see Figure 1) allow establishing a traceability relationship between specific types of inter-scenario relationships and state machine structure. Such traceability has two advantages. First, it facilitates maintaining consistency between hierarchical state machines and scenario models as the system is modified and extended. Second, it simplifies maintenance. For example, in the ATM system, if an error is found in the execution of the *shutdown* scenario, then this error must be localized in that particular composite state.

5. Summary

In this paper, we have presented one of the behavior integration patterns we have defined (see Figure 1) to provide a systematic approach to the design of hierarchical state machines [2]. A two-step strategy subsumes these patterns. First, define the state machines associated with individual scenarios. Second, integrate such state machines into more complex hierarchical state machines.

Industrial experience with such patterns suggests important benefits, including:

- 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 system traceability, maintainability and extensibility

6. Acknowledgments

NSERC, TRIO, Nortel Technology, and ObjecTime provided financial assistance. Bran Selic, Carmine Ciancibello, Don Cameron, Stan Gedrysiak and Luigi Logrippo provided many helpful comments. Fahim Sheik, Julian Kung, and students from Carleton's 94.586 course tried out some of these ideas and provided feedback.

7. References

[1] F. Bordeleau, R.J.A. Buhr. "UCM-ROOM Modeling: From Use-Case Maps to Communicating State Machines", *Proceedings of IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS'97)*. March 24-28 1997, Monterey, California.

[2] F. Bordeleau. *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)

[3] R. J. A. Buhr, R. S. Casselman, *Use Case Maps for Object-Oriented Systems,* Prentice Hall, 1996

[4] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. A System of Patterns. Wiley, 1996.

[6] W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray. Anti Patterns, Refactoring Software, Architectures, and Project in Crisis. Wiley, 1998.

[7] J.-P. Corriveau, "Traceability for Large Projects", *IEEE Computer*, September 1996, pp.63-68.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns-Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[9] D. Harel. "StateCharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Vol. 8, pp. 231-274, 1987.

[10] ITU (1996). *Message Sequence Charts ('96)*. Recommendation Z.120. Geneva.

[11] I. Jacobson et al. *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.

[12] K. Koskimies, T. Mannistö, T. Systä, J. Tuomi. *On the Role of Scenarios in Object-Oriented Software Design*, Technical Report A-1996-1, Dept. of Computer Science, University of Tampere, Tampere, Finland.

[13] K. Koskimies, E. Makinen. "Automatic Synthesis of State Machines from Trace Diagrams". *Software-Practice and Experience*, vol.24, No. 7, pp. 643-658 (July 1994).

[14] D. Lea. Doug Lea's Home Page. http://g.oswego.edu/dl/.

[15] D. Lea. Concurrent Programming in Java, Design Principles and Patterns. Addison-Wesley, 1996.

[16] S. Leue, L. Mehrmann, M. Rezai. Synthesizing ROOM Models From Message Sequence Charts Specifications. TR98-06, Department of Electric and Computer Engineering, University of Waterloo, Waterloo, Canada, 1998.

[17] A. Miga. Use Case Map Navigator. http://www.usecasemaps.org/.

[18] Rational Corporation. Rational Rose for Real-Time Toolset. http://www.rational.com

[19] D. Schmidt. http://www.cs.wustl.edu/~schmidt/patterns-info.html.

[20] B. Selic, J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. http://www.objectime.com, 1999.

[21] B. Selic. *An Architectural Pattern for Real-Time Con-*

trol Software. Internal Report, ObjecTime Limited, Kanata, Ontario, Canada, 1996.

[22] B. Selic, G. Gullickson and P.T. Ward. *Real-time Object-Oriented Modeling*. Wiley, 1994.

[23] J.M. Vlissides, J.O. Coplien, N.L. Kerth. Pattern Languages of Program Design. Addison-Wesley, 1996.

[24] B. Wilhelm. *Designing for Concurrency.* http://www.objectime.com, 1999.