University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada

Technical Report Series

CS-93-20

ADVcharts: a Visual Formalism for Highly Interactive Systems

To appear in the book “Software Engineering in Human Computer Interaction”,

edited by M. Harrison and C. J. Johnson

by

L.M.F. Carneiro D.D. Cowan C.J.P. Lucena

December, 93

ADVcharts: a Visual Formalism for Highly Interactive Systems

To appear in the book "Software Engineering in Human Computer Interaction",

edited by M. Harrison and C. J. Johnson

L.M.F. Carneiro-Coffin *
Department of Computer Science &
Computer Systems Group
University of Waterloo
Waterloo - Ont, Canada, N2L 3G1
lmfcarne@neumann.uwaterloo.ca
dcowan@csg.uwaterloo.ca

D.D. Cowan

C.J.P. Lucena
Departamento de Informática
Pontifícia Universidade Católica
22453-900, Rio de Janeiro-RJ, Brazil
lucena@inf.puc-rio.br

Abstract

This paper introduces a new visual formalism, called ADVcharts, for specifying the behavior of interactive systems (including multi-modal interactive systems) using a state-machine-based approach. ADVcharts combine concepts from Abstract Data Views (ADVs), with notations from Objectcharts, Statecharts, and Petri-nets. ADVcharts are part of the ADV specification approach. In this paper, we abbreviate the "ADV specification approach" term to ADVspec. The ADVspec allows the designer to express visually both the relationship among the user-interface objects and the flow of control of an interactive system using a single integrated approach. It is intended that the ADVspec will serve as a foundation for a future design methodology for interactive systems.

In particular, we show some aspects of design specific to interactive systems, such as the association of input and output events with particular Abstract Data Views, the concurrency of the components of a user interface, and the representation of various modes (input and output) in the design of an interactive system.

The semantics of ADVcharts are presented through the specification of some examples, and we demonstrate that ADVcharts can be used as a visual specification language to represent highly interactive systems from the perspective of the user interface.

We conclude this paper by demonstrating that VDM-like specifications can be derived directly from ADVcharts, thus providing the ADV concept with complementary visual and textual formalisms.

1 Introduction

A number of authors including [38, 16, 15, 25, 36, 33, 24, 26] have studied the concept of dialog independence, where interactive systems are designed and implemented with the goal of providing a clear separation between the user interface and the application. These authors and others have discussed the many advantages of dialog independence including ease of maintenance of the existing system, support of alternative visualizations of an application data structure, and design and implementation reuse.

*L.M.F. Carneiro-Coffin holds a doctoral fellowship from CAPES (the Brazilian Research Council)

Recent papers by Cowan et al [9, 7] present a new design concept called Abstract Data Views (ADVs). ADVs clearly express the separation of the application and the user interface at the design level. The concept of ADVs has been applied to interactive systems, where the user-interface part of the system is represented as an ADV, and the non-user-interface part (the application) is represented as an Abstract Data Type (ADT) [43]. The ADV concept allows the association of different user-interface objects with the same ADT, and the composition of user-interface objects through nesting. By allowing various ADVs to view the same ADT, the designer of the ADT can ignore the question of which class of interfaces will be used for the visualization of the ADT. This notion provides strong support to the designer in the creation of general reusable applications.

A design concept, such as Abstract Data Views, is only useful if it can eventually be implemented through some form of reification[1] that maps the design into an operational program. ADVs have been thoroughly tested in this regard as they have been used to design and implement user interfaces for many different software systems. For example, the chess and checkers games used in [7] and a graph editor [8] were designed using the ADV concept and were later implemented in Smalltalk. In addition, a user-interface design system (UIDS) called GUIIZER [34] that was used to create graphical user interfaces for applications in geophysics, was designed using the ADV concept and was implemented in C++. In each case, a systematic approach was used to map the design based on ADVs into an operational software system.

Although the ADV concept appears to be promising, there are several issues that still need to be addressed. Specifically, we need to create an associated formal design model, which ensures that any design incorporating the ADV concept is developed correctly and systematically. Visual formalisms [19] are particularly appealing, since designers often use various graphics to assist in producing a specific design.

We have created a new visual formalism for specifying interactive systems, based on state-machines, called ADVcharts. ADVcharts are part of the *ADVspec* (ADV specification approach). ADVcharts were developed to create design tools based on the ADV concept, and to serve as the foundation for a future design methodology. This notation has a number of advantages over other visual formalisms for specifying interactive systems, since it deals with problems specific to these type of systems.

Our work on ADVcharts was inspired by earlier research on visual formalisms. Two design methods of particular assistance were Statecharts [18] and Objectcharts [6]. Both design models use a state-machine approach for specifying complex systems. Statecharts are an extension of state machines and state diagrams; their purpose is to specify and to design complex reactive systems [21]. Objectcharts are an extension of Statecharts and are used to characterize the behavior of a class for object-oriented systems as state-machines.

In this paper, we present the ADV concept, justify the need for another visual formalism for interactive systems, demonstrate the ADVspec through some examples using the same style as appears in [18] and [6], and illustrate how VDM-like specification can be derived from the ADVcharts. We also describe Statecharts and Objectcharts as they provide similar visual notations for complex systems.

2 The Abstract Data View Concept

Abstract Data Views (ADVs) [9, 7] are Abstract Data Types (ADTs) [43] that have been modified to support the design of user interfaces. The ADV concept cleanly separates the application from the user interface and is intended to promote design reuse. An initial formal semantics of the ADV concept have been given in [7], using an extension of the VDM specification language [30].

A typical system that is based on the ADV concept consists of a collection of Abstract Data Types (ADTs) [27] that manage the data structures and the state of the application—the non-user-interface part of the system—, a collection of ADVs that comprise the perceivable behavior—the user-interface part of the system—, and a mapping from ADVs to ADTs [5]. The independence of the ADTs and ADVs guarantees the property of a clean separation of the application from the user interface.

[1] The Concise Oxford Dictionary defines the verb 'reify' as 'convert (person, abstract concept) into thing, materialize'

In this paper, we view an ADT as an object in the object-oriented sense of program design because we view an ADT as being defined by a private state and a public interface. The public interface comprises services provided by the ADT and attributes (variables and their corresponding values) of the ADT that can be modified by an ADV or another ADT. Only the public interface of an ADT is accessible to external sources. The term ADT is used because we are mostly interested in its properties as a type. Moreover, the ADT is completely independent of the user interface and, in fact, does not have access to any input or output events.

The ADV handles all the input functions, provides all output functions, and fully controls its associated ADT. The ADV implements all decisions concerned with the information exchanged between the user and the user-interface application, and among other ADVs. Figure 1 shows the ADV, ADT, and the mapping from an ADV to an ADT (the mapping is represented as an arrow connecting the ADV to the ADT). The mapping associates the public interface of the ADT with its corresponding ADV. More specifically, the mapping between an ADV and an ADT is represented by a variable in the ADV that we call **owner** (because an ADV is associated with an ADT, we say that the ADT is the owner of the ADV), and therefore the variable **owner** represents the channel between the user-interface part of the application and the non-user-interface part. The variable **owner** provides a notation for binding the ADV to a specific ADT at the design level. Thus, any change in the variable **owner** in the ADV will be reflected in its associated ADT, and vice versa. The strategy used to ensure the consistency of the variable **owner** with the attributes of the associated ADT is not specified by the ADV concept. Several different approaches for implementing the **owner** variable have been studied, but we do not discuss them in this paper. Details of one implementation can be found in [34].

The concept of the **owner** variable can be formalized as:

- Let A = set of elements available in the ADT public interface;
- Let B = set of elements of the **owner** variable;
- B $\subseteq$ A (by definition of the **owner** variable);
- For all elements of B, there is a corresponding element in A (call this element x). We refer to this element as owner.x. Formally:

$$\forall x \in B \cdot \exists y \in A \cdot x = y$$

A user interface can provide many different views of the data stored in an application. For example, an integer—a very simple ADT—could be displayed as a number, a position on a dial, or a position of a slider. For this reason, the ADV concept allows the association of several ADVs with a single ADT, where each ADV can provide a different view of the ADT or a different control functionality. Since an ADT has no knowledge of input or output, it does not need to refer to any ADV. As a consequence, there is *not* a symmetrical arrangement between an ADT and an ADV, that is, the ADV needs to know about its associated ADT, but the ADT does not need to know about its corresponding ADVs—at least at the design level. Since an ADV is associated with one and only one ADT, while an ADT can be associated with several ADVs, there is the property that an ADT can be viewed in different ways, but the different views represented by the multiple ADVs must be consistent with the ADT they represent.

We believe that user interfaces are built by composing behavior, structure, and form. The behavior of a user interface when a user initiates an action can be viewed as the composition of simpler behaviors in just the same way a complex routine is composed of a number of simpler routines. For example, every window under the Microsoft Windows operating system can behave, roughly speaking, in two ways: iconized or expanded. We can usually associate different states with each of these behaviors. Similarly, a user interface can be composed of a number of visual objects. For example, a dialogue box may be composed of a text-field, an editable text-field, radio buttons, check boxes, and a list box. Finally, the different forms of the

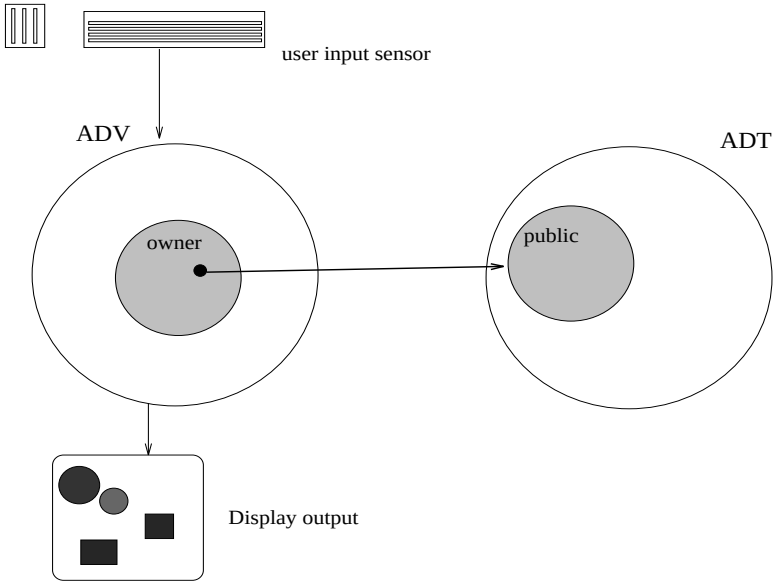


Figure 1: The ADV Architectural Model

objects depicted in a user interface can be classified into groups of related abstractions. For example, human beings can be classified into two groups of related abstractions: men and women. A user interface that displays men behaves in almost the same way as a user interface that displays women, but there are some differences (for example, displaying different faces for men and women). The form of decomposition where the behavior or structure of an object is decomposed into simpler behaviors or structures we call *behavioral* and *structural nesting*, respectively. The form of decomposition where objects are classified into groups of related abstractions with common ancestry we call *decomposition by form* [3, 14]. The ADV concept, as described in [9, 7], supports behavioral and structural nesting. We extended the ADV concept to support decomposition by form [4].

Both ADVs and ADTs can be viewed as objects. In this paper, we concentrate on the specification of the ADV objects. ADVs can be composed of other ADVs—structural nesting—, and also can contain attributes that are shared by their components. This aspect of composition differs from the standard object-oriented paradigm in that objects only contain attributes. We extend this object-oriented view by supporting structural nesting. However, we do not enforce in the ADV concept any implementation-biased approach for implementing the ADV and ADT objects and the nesting mechanism.

In systems based on the ADV concept, the control originates in the user-interface part of the application rather than in the non-user-interface part. We believe that this placement of control avoids the callback spaghetti problem [37] and is appropriate, since in highly interactive systems the flow of control is determined primarily by the user of the system through the user interface. Also, in systems based on the ADV concept, each component of the user interface can be associated with a single ADV because the ADV concept supports structural nesting. Thus, the user interface can be composed of multiple ADVs. The use of multiple ADVs makes the flow of control of each ADV quite simple: each ADV responds to a relatively small number of user-generated events, and examines and manipulates one and only one ADT.

In the ADV concept, ADVs and ADTs often need to be logically related. However, this relationship does not imply anything about the implementation strategy. For example, we might need to specify invariants governing relationships between an ADV and an ADT, but this specification does not imply that the two components—the ADV and the ADT—should reside in the same machine.

In the next section, we briefly discuss Statecharts and Objectcharts.

3 Statecharts and Objectcharts

Statecharts [18] and Objectcharts [6] are visual formalisms for describing the behavior of complex systems. Statecharts are an extension of finite state machines, and have been formulated to avoid many of the disadvantages of a finite state machine representation, such as state explosion and lack of structure [20]. Objectcharts extend Statecharts to allow the specification of object-oriented systems.

Finite state machines are described by an input alphabet, an output alphabet, a set of states (including start and final states), and a set of state transitions [29]. Statecharts extend the finite state machine notation by adding *hierarchy*, *concurrency*, and *broadcast communication*. Hierarchy allows common transitions to be clustered and adds structure to the chart. Hierarchy is shown in Figure 2.a, where state A is composed of two other states, B and C. The system can be in state B or C, but not in both states at the same time. Concurrency is represented by an AND decomposition of states and is an attempt to avoid state explosion. For example, in Figure 2.b, the state D consists of components E AND F. The system can be in states E and F simultaneously. The transitions in the states E and F may be fired synchronously, or independently from each other. In the case the system is in the state $< e_1, f_1 >$, and an event *x* occurs, the system goes to the state $< e_2, f_2 >$. However, if the system is in the state $< e_2, f_2 >$, and an event *y* occurs, the system goes to the state $< e_2, f_1 >$, which means that no transition of the E component is fired.

The communication mechanism used in Statecharts is called broadcast communication. Broadcast communication is the ability to propagate the occurrence of an event through all AND components, and fire simultaneously all transitions caused by the event .

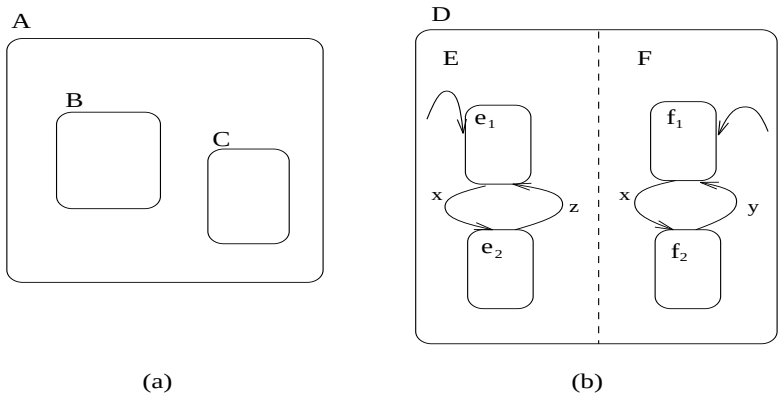


Figure 2: Hierarchy in Statecharts and Concurrency in Statecharts

Objectchart notation, which is a visual formalism for the specification of object-oriented systems, uses two types of diagrams. The instances of objects and the communication among these objects are captured by means of a *configuration diagram*; the class behavior is described by means of Objectcharts. A configuration diagram for the ADVspec is shown in Figure 3 and the one for an Objectchart is similar. The services provided by and required by a class are shown in the diagram.

Objectcharts characterize the behavior of a class as a state machine, with an input and output alphabet comprising the operations provided and required by the class. To differentiate a required serviced from a provided service, the prefix "/" is used whenever a provided service is referenced in a transition. Objectcharts do not use broadcast communication; instead, they consider service request and complete service execution as a single atomic event.

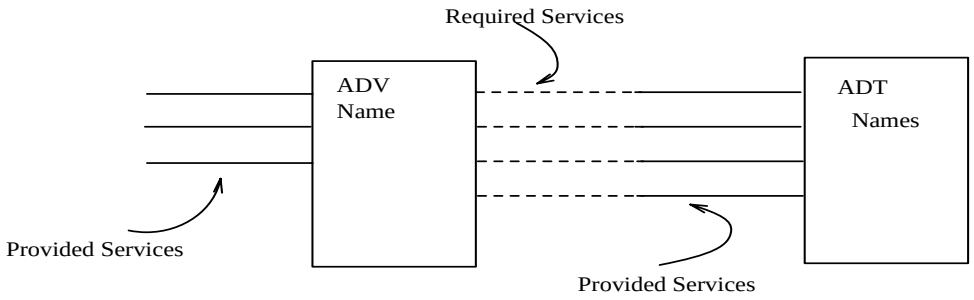


Figure 3: Configuration Diagram for ADV

Objectcharts augment states with *attributes* (variables and their corresponding values) and *observers*. The effect of state transitions on the attributes are specified. Observers allow a provided service to report on the value of attributes without changing the state of an object.

In order to complete the Objectchart, the transitions need to be specified. A transition specification comprises the initial and final state names of the transition, the service name for the transition, the firing condition, and a post-condition. The firing condition is a predicate describing the restrictions over attributes and observers for the transition; the post-condition is a predicate over attributes and observers describing the effect of the transition. We can also add to Objectcharts invariant specifications, comprising a state name and the relation that needs to hold in the state that we are describing.

Every finite-state machine has a start state which is its default state. This state is distinguished from the other states, in Statecharts and Objectcharts, by annotating it with a small arrow without an initial state. We use the same representation in ADVcharts.

4 ADVcharts

ADVcharts are part of the ADVspec. ADVcharts provide a visual schema for specifying interactive systems that are based on the ADV concept. ADVcharts are based on finite-state machines and are an extension of both Statecharts [18] and Objectcharts [6]. ADVcharts also use notations from Petri-nets [41].

The ADVspec is composed of four components: a configuration diagram, ADVcharts, transition specifications, and invariant specifications. The transition and invariant specifications use notations from VDM [32, 11, 1].

As observed by Marshall [35], interaction between a user and a system may be viewed as a flow of control between operations. The ADV concept, as presented in [7], allows only the specification of the operations and the decomposition by structure of the user-interface components. The ADVspec complements the ADV concept by also allowing the specification of the flow of control between operations and the decomposition by form of the user-interface components.

As mentioned in the previous section, the input and output alphabets for Objectcharts are the set of provided and required services. As ADVcharts are based on finite-state machines, we need to specify the input and output alphabets. External (user-initiated) events are modeled as provided services by the ADV, because the ADV provides services to the user that generates the event. ADV-initiated events, such as display, are also modeled as provided services, because ADVs provide services to some media that accepts the event. ADT invocations comprise the required services. In addition, we have noticed a need to associate transitions with the state of the system rather than only to associate them with required and provided services. Therefore, we have associated conditions on attributes (variables and their values) with transitions. In this way, we can specify that a transition fires because the state of the ADV changes, without having to introduce an artificial event for this purpose. Consequently, the input alphabet for

ADVcharts consists of the services provided and required by an ADV, and conditions on attributes of the ADV. The output alphabet for ADVcharts consists of the services provided and required by an ADV and other functions specified inside the ADVs.

Multi-modal interfaces [2] can be specified using ADVcharts by modeling the actions generated by the user from the various media (input modes such as keyboard, mouse, and footpedal) as provided services of an ADV as explained above. The output (ADV-initiated) events (such as sound and display) can be modeled as ADV provided services.

As we mentioned in section 2, the ADV concept supports behavioral and structural nesting, and decomposition by form. Behavioral nesting is represented in ADVcharts through the nesting of states, and is also supported by Statecharts and Objectcharts. Structural nesting is represented in ADVcharts through the nesting of ADVs, in the same style of nesting of states. Structural nesting is not represented in Statecharts and Objectcharts. Decomposition by form is represented in the ADVspec through the inheritance tree (in the same style as Objectcharts) in the configuration diagram, and by shading the ADVcharts.

ADVcharts generalize Statecharts and Objectcharts to handle aspects of design specific to interactive systems, such as using pointing devices to associate events with particular ADVs, and the inherent concurrency between the ADVs that compose the user interface. Structural nesting supports specifying focus of control as part of the design since the components of an object are clearly indicated. Because of the inherent concurrency of ADVs, a mechanism for synchronization of ADVs is needed, as we are specifying user interfaces by means of a collection of independent ADVs. For that purpose, we have introduced some notation from Petri-nets into ADVcharts to represent the synchronization.

As presented in the previous section, ADVs provide services to the user, and require services from ADTs. We assume that ADTs only provide services to simplify the problem (this assumption is reasonable because we can always encapsulate ADTs in such a way that the final ADT will only provide services). The relationship between ADVs and ADTs is shown using a configuration diagram in the same style as the configuration diagram used in Objectcharts. The purpose of the configuration diagram is to specify, and clarify the interface between the ADV and its associated ADT, and to represent the concept of decomposition by form, by representing the ADV inheritance tree. In the configuration diagram we specify the services provided and required by an ADV, show which ADT provides the services required by the ADV, and show the hierarchy tree representing the inheritance of ADVs.

In the next section, we present the symbols and some rules used in the ADVspec.

5 Symbols and Rules used in the ADVspec

An ADVchart is composed basically of ADVs, states, attributes, and transitions. An ADV is represented in the ADVcharts by a rectangle with the name of the ADV on top of the rectangle, as depicted in Figure 4. A state, shown in Figure 5, is represented by a rounded rectangle containing the name of the state. ADVs must contain one or more states to describe their behavior, and can contain other ADVs to describe their decomposition by structure. A state can stand alone, can enclose a cluster of states or ADVs, or both.

Attributes can be defined in an ADV, or in a state. The scope of an attribute defined in an ADV is the ADV itself; that is, all states and ADVs defined inside this ADV view the attribute. The scope of an attribute defined in a state is the state itself; that is, all states defined inside this state, including states internal to ADVs that are defined inside this state. However, we believe that if we want to define reusable ADVs we should restrict access to attributes of a specific ADV to the ADV itself and the states that are inside the ADV, and not consider the attributes visible by ADVs defined inside this specific ADV.

States are linked with each other by transitions. Transitions are represented by arrows from one state to another state, as depicted in Figure 6. A transition from an ADV to a state, a state to an ADV, or from an ADV to another ADV is not allowed. A transition can be annotated with provided or required services, and conditions on the attributes of the ADV. The format of a transition is similar to the one used in Statecharts.

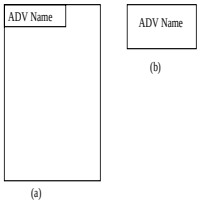


Figure 4: The ADV Representation

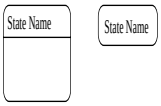


Figure 5: The State Representation

Once the ADVchart is represented, we need to define for each transition in the ADVchart its pre-condition, the event that cause the transition to be fired, and the post-condition. The language used for the specification of the transitions is VDM. Finally, it is also possible to specify invariants over the attributes of the ADVs and states. The specification language used to define the invariants is also VDM.

In the next sections, we present informally the semantics of the ADVcharts through some examples.

6 The First Example

In this section, we illustrate the ADVspec by presenting a solution to the Logon problem. This problem with a solution using VDM and a visual notation based on state-machines (also called statecharts, but different from Harel's work) are presented in [35]. The user interface for the Logon problem consists of a screen composed of two visual objects, as depicted in Figure 7. The system displays the user interface and waits for the user to type a user name into the field with the label *Login:* . Once the user name is provided, the application (ADT) validates the user name by checking whether the user belongs to the set of allowable users. If the user belongs to the set of allowable users, the screen disappears (this assumption is just to simplify the example); otherwise the message "Invalid User" is displayed on the screen, and the system waits for a new user name.

Given the statement of the Logon problem, we can associate the objects that appear on the user interface with ADVs, and specify the configuration diagram. As shown in Figure 8, the whole screen is associated with the ADV Logon, the field where the user will input a user name is associated with the ADV Prompt, and the field where the error message will be displayed is associated with the ADV Error.

In the statement of the problem, we can also identify the services provided and required by the ADV Logon. The two services provided are keyboard entry and displaying of the Logon screen itself; the service required by the ADV Logon is related to the validation of the user name, which is called *Valid User Name*.

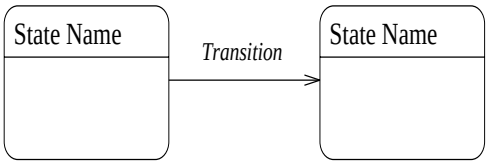


Figure 6: The Transition Representation

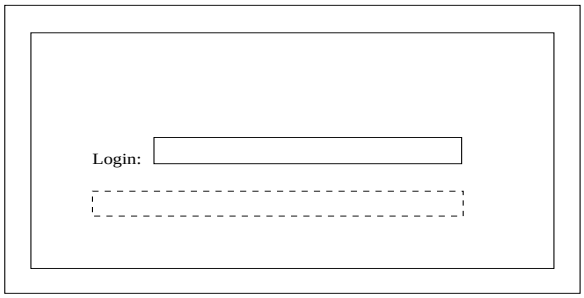


Figure 7: Logon screen

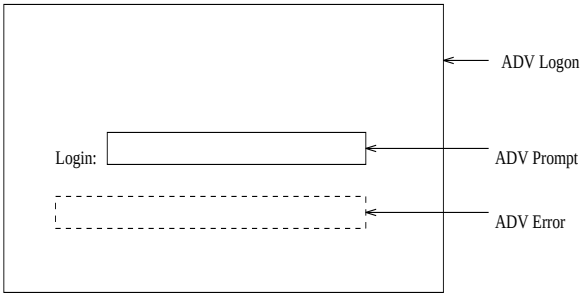


Figure 8: Logon screen with the associated ADVs

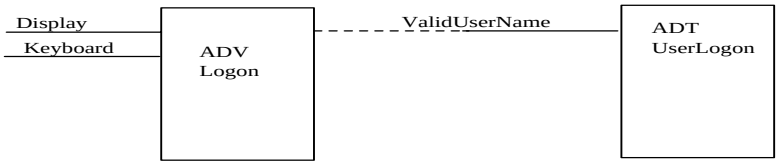
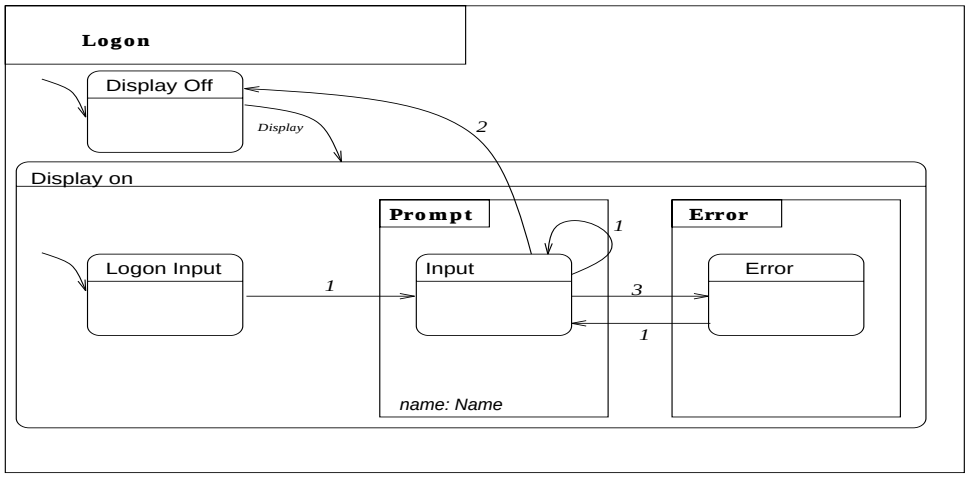


Figure 9: Configuration Diagram for the Simple Logon Example

The configuration diagram is shown in Figure 9.



1. Keyboard (focus & keyboard_input not equal CR)

2. Keyboard (focus & keyboard_input = CR & \Owner.ValidUserName)

3. Keyboard (focus & keyboard_input = CR & not \Owner.ValidUserName)

Figure 10: ADVchart for the Simple Logon Example

As shown in Figure 8, the ADV Logon is composed of two other ADVs: the ADV Prompt, and the ADV Error. The property of allowing an ADV to be composed of two other ADVs is referred to as the ADV structural nesting property, as explained in section 2. In an ADVchart, this property is made visually explicit by showing ADVs drawn inside ADVs as depicted in Figure 10.

The next step is to describe each transition of the ADVchart. We use a VDM-like style to specify the pre-conditions, the post-conditions, and the events used to annotate each transition. This approach is similar to the one used in Objectcharts, but we adopt an approach that is more textual and closer to VDM. The transitions for the ADV Logon are described in Figure 11.

As shown in Figure 11, a transition specification is composed of four fields: the transition, the pre-condition, the event, and the post-condition. The transition is described by means of the name of a state, an arrow, and the name of the next state. For example, the transition from the state "**Display off**" to the state "**Display on**" is written as "**Displayoff** → **Display on**". In the case of a transition in which the first state is the start state, we do not mention the first state. An example is the transition "→ **Display off**".

In the pre-condition field, we specify the conditions that must be true in order to fire the transition. In the event field, we specify the events that must occur in order to fire the transition. In the post-condition field, we specify the conditions that must be true once the transition is fired.

We use the function *focus* in Figure 11. This function can be specified in several ways. One possible approach to define *focus* is to have a variable called **focus** associated with each ADV, and to set this variable on or off, depending whether the ADV is selected. In this case, we need to specify a function that sets and unsets the variable **focus**. Using this approach, *focus(advName)* would be equivalent to *advName.focus*, and we postpone the problem of specifying how this variable will be set and unset. Another possibility is to have a cursor-based approach. Informally, in this case, the *focus* function returns true if the cursor—the focus—is inside the ADV that was given as a parameter. Formally, the function *focus* is described in Figure 12. In the formal description, we are assuming the existence of two state variables: **sm** and **cursor**. The variable **sm** is a mapping from location to ADV, where a location is a set of positions on the screen. We use the function *CursorPos* to describe the function *focus*. The function *CursorPos* returns true if the cursor position

Transitions Logon

- $\rightarrow$ **Display off** :
  event :
  pre-condition : $\{true\}$
  post-condition : $\{screen = empty\}$
- **Display off** $\rightarrow$ **Display on** :
  event : Display
  pre-condition : $\{true\}$
  post-condition : $\{screen = \overleftarrow{screen} + \text{“Login:”}\}$
- **Logon Input** $\rightarrow$ **Input** :
  event : ADV_Prompt.Keyboard
  pre-condition : $\{focus(Prompt)\}$
  post-condition : $\{name = GetUserName() \wedge$
  $screen = \overleftarrow{screen} + name\}$
- **Input** $\rightarrow$ **Error** :
  event :
  pre-condition : $\{\neg \setminus owner.ValidUserName(name)\}$
  post-condition : $\{screen = \overleftarrow{screen} + \text{“Invalid User”}\}$
- **Input** $\rightarrow$ **Display Off** :
  event :
  pre-condition : $\{\setminus owner.ValidUserName(name)\}$
  post-condition : $\{screen = empty\}$
- **Error** $\rightarrow$ **Input** :
  event : ADV_Prompt.Keyboard
  pre-condition : $\{focus(Prompt)\}$
  post-condition : $\{name = GetUserName() \wedge$
  $screen = \overleftarrow{screen} + name\}$

Figure 11: Transition Specification for ADV Logon

is inside the ADV that was given as a parameter, and returns false if the cursor position is inside another ADV. The function *CursorPos* is also described in Figure 12. However, we will not enforce any approach to the specification of the *focus* in the ADV concept, but the designer needs to be aware that, eventually, the function *focus* needs to be specified.

$$ScreenMap = Location \xrightarrow{m} ADV$$

$$Location = Position\text{-set}$$

$$\begin{aligned} Position :: \; & x : \{\cdots\} \\ & y : \{\cdots\} \end{aligned}$$

$$\begin{aligned} State :: \; & sm : ScreenMap \\ & cursor : Position \\ & \cdots \end{aligned}$$

$$focus : ADV \rightarrow \mathbb{B}$$

$$\begin{aligned} & focus(adv) \triangleq \\ & \quad \text{if } CursorPos(adv, sm) \\ & \quad \text{then } true \\ & \quad \text{else } false \end{aligned}$$

$$CursorPos : ADV \times ScreenMap \rightarrow \mathbb{B}$$

$$\begin{aligned} & CursorPos(adv, smap) \triangleq \\ & \quad \text{let } i \in smap \text{ in} \\ & \quad \text{if } cursor \in \text{dom } i \\ & \quad \text{then } adv \Leftrightarrow \text{rng } i \\ & \quad \text{else } smap := \text{dom } i \lhd \overleftarrow{smap} \\ & \qquad CursorPos(adv, smap) \end{aligned}$$

Figure 12: A possible formal description of the function *Focus* and *CursorPos*

In the next section, we present other examples using the ADVspec. More specifically, we use AND decomposition and decomposition by form (inheritance).

7 Other Examples

7.1 Example of "And" Decomposition

In this section, we present a solution for an extension of the Logon problem, which was originally described in section 6. We call this problem the *Other Logon* Problem. In this example, we show how to use the ADVspec to specify a user interface composed of ADVs that operate independently from each other, and how to specify a collection of ADVs that have the same behavior without knowing beforehand how many of these ADVs are present in the user interface. We call the latter ADV an *ADV-collection*.

The Other Logon problem consists of allowing two users to login, but each user is assigned different login fields. The description of the user interface for the Other Logon problem consists of a screen composed of several visual objects, as depicted in Figure 13. The problem consists of displaying the user interface with the two login fields, and the system waiting for the two users to input their user names. The user name

can also be selected from a menu that is made available for the user by clicking the mouse button inside the login field (as you see, we are not concerned with security problems !). The list of allowable users is provided by the application (ADT). In the case where the user name is provided by the user, the application will validate the user name for each user by checking whether the user belongs to the set of allowable users. If the user belongs to the set of allowable users, the screen disappears; otherwise the message "Invalid User 1"or "Invalid User 2" is displayed on the screen and the system will wait until one or two users provide new user names.

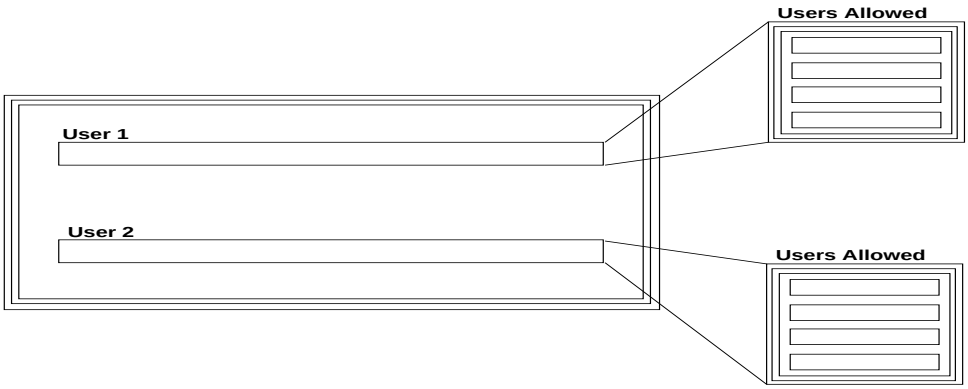


Figure 13: The Other Logon Interface

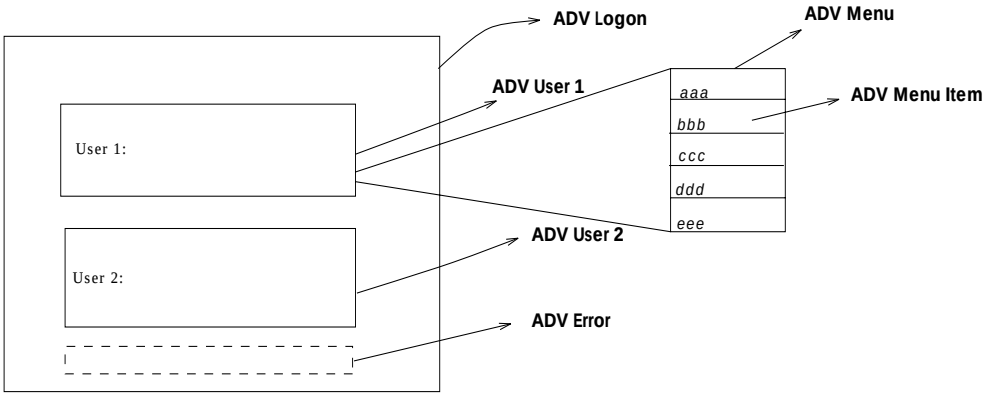


Figure 14: The Other Logon Interface with the Associated ADVs

Given the statement of the Other Logon problem, we can identify several ADVs, as depicted in Figure 14. The whole screen is associated with the ADV Logon, the field where user1 and user 2 input their user names is associated with the ADV User 1 and ADV User 2, respectively; the menu is associated with the ADV Menu, each item of the menu is associated with Menu Item, and the field for the error message is associated with the ADV Error.

In the statement of the problem, we can also identify the existence of three services provided by the ADV Logon (keyboard entry, display of the Logon screen, and mouse click), and the two services required by the ADV Logon (Valid User Name and Get User Names). The configuration diagram is shown in Figure 15.

The ADVchart for the Other Logon Problem is presented in Figure 16 and the annotations for the transitions are presented in Figure 17. In the ADVchart, we can observe three differences with respect

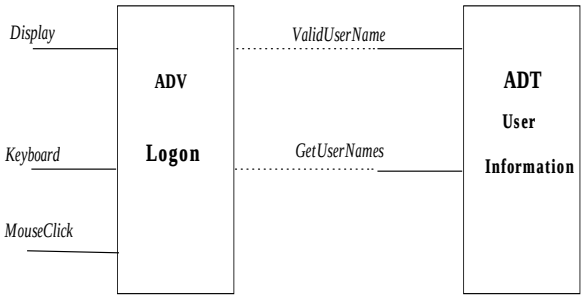


Figure 15: Configuration Diagram for Other Logon

to the previous example: the AND decomposition of the states, the synchronization transition, and the ADV-collection. The AND decomposition is used to model the concurrent aspect of the ADVs User1 and User2. The style of the AND decomposition is the same as the one used in Statecharts [18].

The synchronization transitions are represented by the two different transitions (**input → Display off** and **init → Display off**) shown in Figure 16, which represents the fact that the Logon screen disappears because User 1 and User 2 have input (or chosen from the menu) correct user names. We express the synchronization transition using a notation adopted from Petri-nets: a bar with two or more inputs, and one output. The semantics of this bar notation are an AND of the input transitions, that is, the output will be fired if and only if all the inputs are true, but whenever they are true. A possible representation for the synchronization transition using Statecharts is depicted in Figure 18. The synchronization transition notation is an attempt to save states [2]. In ADVcharts, we assume the interleaving model for the events.

The ADV-collection is used in this example to represent a menu that is composed of several menu items, but we do not know in advance how many items the menu contains. The ***** operator is used with the ADV MenuItem, in Figure 16, to indicate the collection of ADVs with similar behavior. The function for the creation of the ADV components (the ADV MenuItems in this case) is given in the ADVchart and it is specified using the "function:" field. In our example, the creation function is called *CREATE*. The *CREATE* function is referenced in Figure 16 and is described in Figure 19. In the description of the function we use the operator mk-ADV, which has four parameters: the owner of the ADV, the parent of the ADV, the name of the ADV, and an identifier for the ADV—just its number, for instance.

The semantics for the ADV-collection chart of this example, using Statecharts, are depicted in Figure 20. Further discussion of ADV-collection is postponed until Section 8.

The next step is to specify the transitions of the ADVcharts of the Other Logon Problem. These transitions are partially described in Figures 21 to 25. Transitions for the ADV User2 are omitted as they are similar to the transitions for the ADV User1. In Figure 23, we introduce the " $\circ$ " symbol as a notation for the synchronization transition. The synchronization transition is represented as a single transition with two start states and one final state, that is, the system goes from both states **Input** in ADV User 1 and ADV User 2 to the state "**Display Off**" in ADV Logon. However, when specifying pre-conditions, and/or post-conditions, we may need to have information about a particular ADV involved in the transition. In our example, we need to specify a pre-condition establishing that the user names are valid in both ADVs User 1 and User 2. The predicate expressions for both transition specifications have the same format, but the scope of the parameter *name* used in the services is different for each ADV. Therefore, the " $\circ$ " notation differs from a simple AND in that it conveys a correspondence between the arguments in the pre-condition and the state names in the transition. In other words, in the pre-condition predicates, the first predicate refers to the ADV User 1 and the second predicate refers to the ADV User 2.

We note in the solution of the problem, more specifically in Figure 16, that the specification of User 1 and

[2] Observe that in Figure 18 we have two additional states: User 1 Wait and User 2 Wait

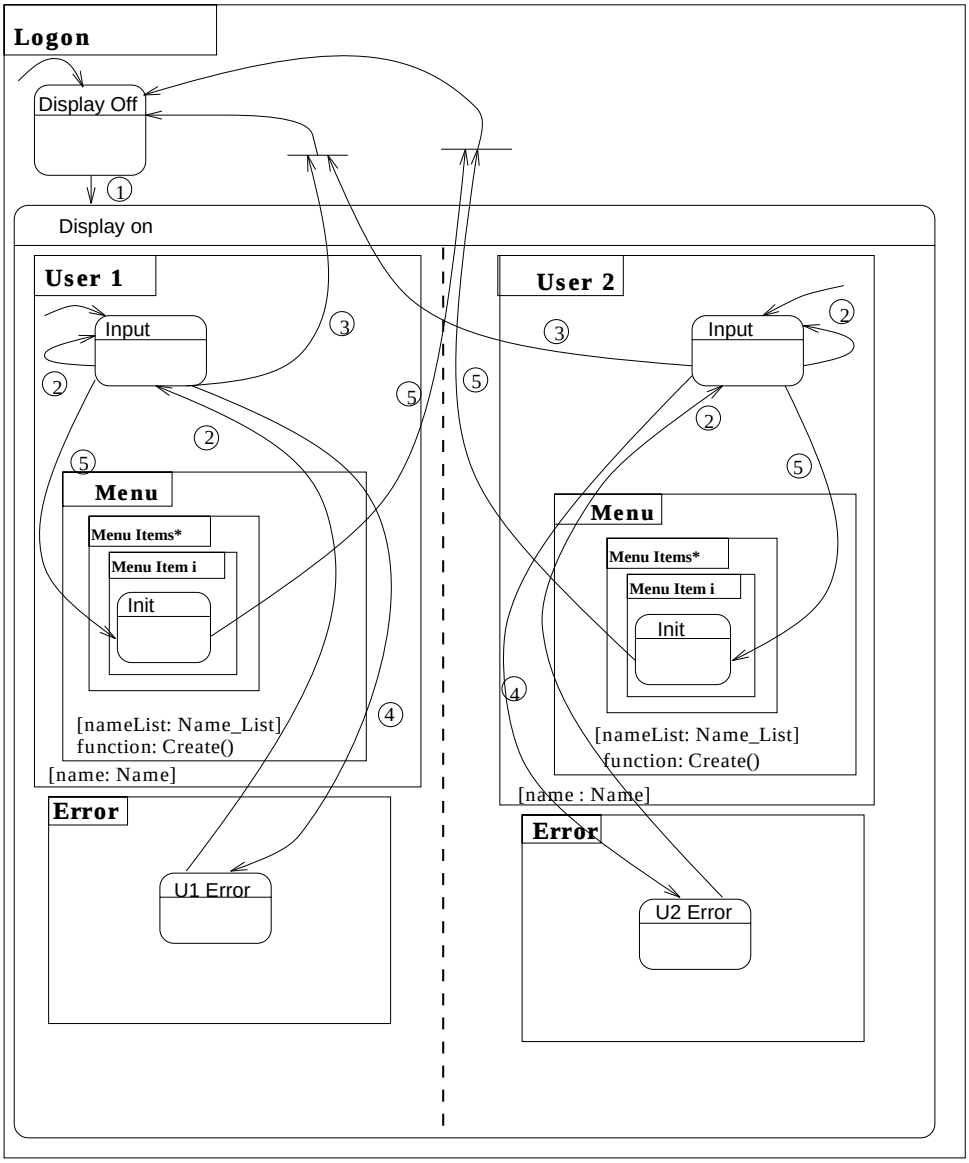


Figure 16: ADVchart for the Other Logon Problem

1. Display
2. Keyboard ($keyboard\_input \neq CR \wedge focus$)
3. Keyboard ($keyboard\_input = CR \wedge focus \wedge \backslash owner.ValidUserName$)
4. Keyboard ($keyboard\_input = CR \wedge focus \wedge \neg \backslash owner.ValidUserName$)
5. MouseClick (focus)

Figure 17: Transitions for the ADVchart of the Other Logon Problem

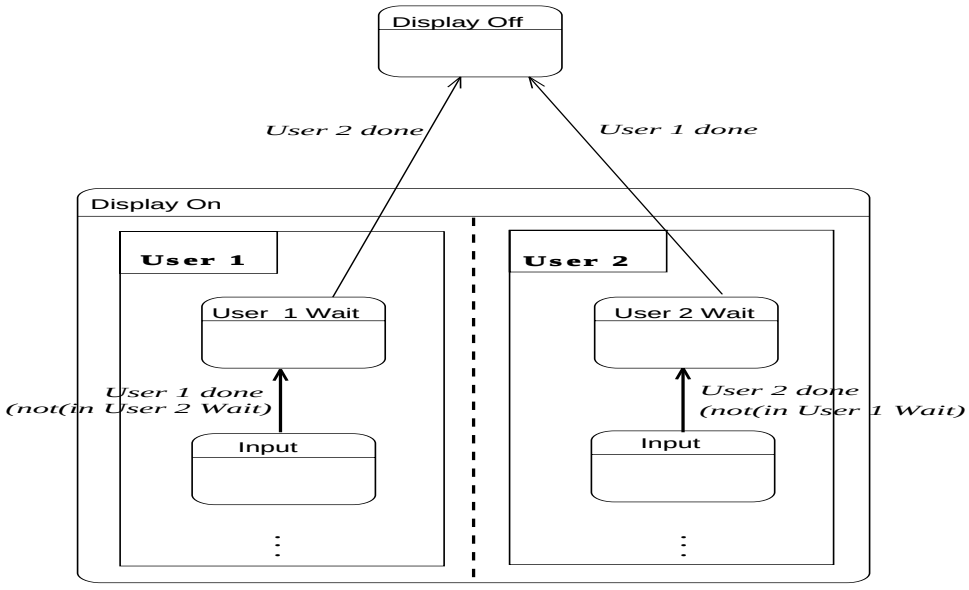


Figure 18: A Possible representation of the Synchronization Transition using Statecharts

Function $Create\ ()$
external: wr $screen$
post-condition: $name\_list = \backslash owner.GetUserNames \wedge$
$\forall i \in name\_list \cdot mk\text{-}ADV(owner, Menu, MenuItem, i)$

Figure 19: The Create Function

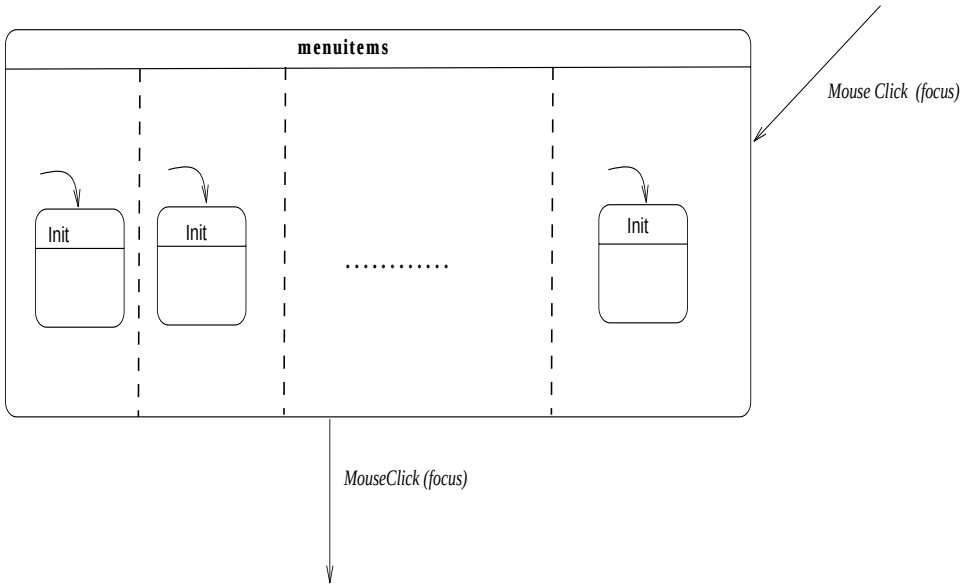


Figure 20: A Possible Representation for the ADV MenuItems using Statecharts

Transitions Logon

- $\rightarrow$ **Display off** :
  event :
  pre-condition : $\{true\}$
  post-condition : $\{screen = empty\}$
- **Display off** $\rightarrow$ **Display on** :
  event : Display
  pre-condition : $\{true\}$
  post-condition : $\{true\}$

Figure 21: Transition Description for ADV Other Logon

Transitions ADV User 1

- $\rightarrow$ **Input** :
  event :
  pre-condition : $\{true\}$
  post-condition : $\{screen = \overleftarrow{screen} +$ "User 1:" $\wedge$
  $name = nil\}$
- **Input** $\rightarrow$ **Input** :
  event : Keyboard
  pre-condition : $\{focus(User1)\}$
  post-condition : $\{name = \overleftarrow{name} + keyboard\_input \wedge$
  $screen = \overleftarrow{screen} + keyboard\_input\}$
- **Input** $\rightarrow$ **U1 Error** :
  event : Keyboard
  pre-condition : $\{focus(User1) \wedge keyboard\_input = CR \wedge \neg \setminus owner.ValidUserName(name)\}$
  post-condition : $\{screen = \overleftarrow{screen} +$ "Invalid User 1" $\wedge$
  $screen = \overleftarrow{screen}$ - name $\wedge$
  $name = nil\}$
- **Input** $\rightarrow$ **Init** :
  event : MouseClick
  pre-condition : $\{focus(User1)\}$
  post-condition : $\{name\_list = \setminus owner.GetUserNames \wedge$
  $\forall i \in name\_list \cdot mk\text{-}ADV(owner, Menu, i)\}$

Figure 22: Transitions for ADV User 1

Transitions ADV User 1.Menu.MenuItem$_i$

- **User 1.Menu.MenuItem$_i$.Init** $\circ$ **User 2.Menu.MenuItem$_i$.Init** $\rightarrow$ **Display off** :
  event : MouseClick $\circ$ Mouseclick
  pre-condition : $\{focus(\text{MenuItem}_i) \circ focus(\text{MenuItem}_i)\}$
  post-condition : $\{screen = empty\}$

Figure 23: Transitions for Menu Item

Transitions ADV Error

- **U1 Error → User 1.Input** :
  event : Keyboard
  pre-condition : $\{keyboard\_input \neq CR \wedge focus(User1)\}$
  post-condition : $\{screen = \overleftarrow{screen} + keyboard\_input \wedge$
  $name = \overleftarrow{name} + keyboard\_input\}$
- **U2 Error → User 2.Input** :
  event : Keyboard
  pre-condition : $\{keyboard\_input \neq CR \wedge focus(User2)\}$
  post-condition : $\{screen = \overleftarrow{screen} + keyboard\_input \wedge$
  $name = \overleftarrow{name} + keyboard\_input\}$

Figure 24: Transitions for ADV Error

Transitions ADV User 1 and ADV User 2

- **User 1.Input ∘ User 2.Input → Display off** :
  event : Keyboard ∘ Keyboard
  pre-condition : $\{focus(User1) \wedge keyboard\_input = CR \wedge \backslash owner.ValidUserName(name) \circ$
  $focus(User2) \wedge keyboard\_input = CR \wedge \backslash owner.ValidUserName(name)\}$
  post-condition : $\{screen = empty\}$

Figure 25: Transitions for ADVs User 1 and User 2

User 2 are identical. Therefore, we could treat them as an ADV-collection and use the ***** operator for their specification. If we use the ***** operator, we need to specify two elements of the set (call it $User_i$ and $User_j$) to show the synchronization problem. The specification of Users using the ***** operator is more general as we allow any number of users to logon.

A modification of the Other Logon problem would be more useful and more realistic. Suppose that instead of having two user fields on the same screen, they are on different screens. For example, in order to run a certain program (such as a chess game), the two users (or players) need to give their names. In this new example, the ADV Logon becomes a virtual ADV and, therefore, the specification of the user interface of the logon screen of this new problem is similar to the one presented in this section.

7.2 Modeling Inheritance in ADVcharts

In this section, we demonstrate the semantics and the modeling capability of ADVcharts by presenting a solution to the "Human Identification" problem—HumanId problem, for short. In this example, inheritance and nesting properties are used to illustrate the combination of the two design strategies. The reasons for both kinds of decomposition (form and structure) are discussed in [3, 14]. We use the same semantics for inheritance as the one used in [6].

The HumanId problem consists of displaying a person, given his/her name, and displaying the information about the person's family. Depending on the sex of the person, we need to display a different type of face, that is, a woman or a man's face. The person's family information will be obtained by positioning the cursor on the different parts of the person's body, and clicking the mouse. If we click the mouse on the head, the system displays the names of the person's parents; if we click the mouse on an arm, the system displays the names of the person's siblings; if we click the mouse on the body, the system displays the name of the person's partner; and, finally, if we click the mouse on a leg, the system displays the names of the person's children.

We have divided humans into two classes: men and women. In other words, Man *Is-a* human and Woman *Is-a* human. The Is-a relation is represented by using decomposition by form (inheritance). Humans are *Composed-of* legs, arms, body, and head. The *Composed-of* relation is represented by using the nesting property.

Men and women, in our problem, differ from each other only in that they have a different type of face. Therefore, most information can be shared by men and women, and we want to represent and make this fact explicit at the design level.

By analyzing the statement of the problem, we find we need an ADT that manipulates a data structure for a genealogy tree and that provides the following services: *getPerson*, *getParents*, *getSiblings*, *getChildren*, and *getPartner*. All the functions have the name of the person as a parameter. The function *getPerson* locates the person in the genealogy tree, and returns the person's sex; the functions *getParents*, *getSiblings*, *getChildren*, and *getPartner* traverse the genealogy tree and return the names of the person's parents, siblings, children, and partner, respectively.

The Configuration diagram for the HumanId problem is shown in Figure 26. The ADV Human provides the following services: *MouseClick* and *Display*. The ADV Man and ADV Woman modify the service *Display* and inherit the service *MouseClick* from the ADV Human. The ADV Human requires the following services: *GetPerson*, *GetParents*, *GetPartner*, *GetSiblings*, and *GetChildren*. These services are provided by the ADT Genealogy.

The ADVchart of the ADV Human is shown in Figure 27. We notice in Figure 27 that the ADVs Leg, Body, Arm, and Head are independent. This fact is shown using the AND decomposition notation in the same style presented in [18]. In the initial state, which is the "**Display off**" state, the screen is blank. When a *Display* event occurs, given the name of the person we want to display, the system displays the leg, the arm, the body, and the head of a person. The head is displayed according to the sex of the person. Therefore, after the *Display* event occurs, the system must be in different states and the initial state for the ADV Head

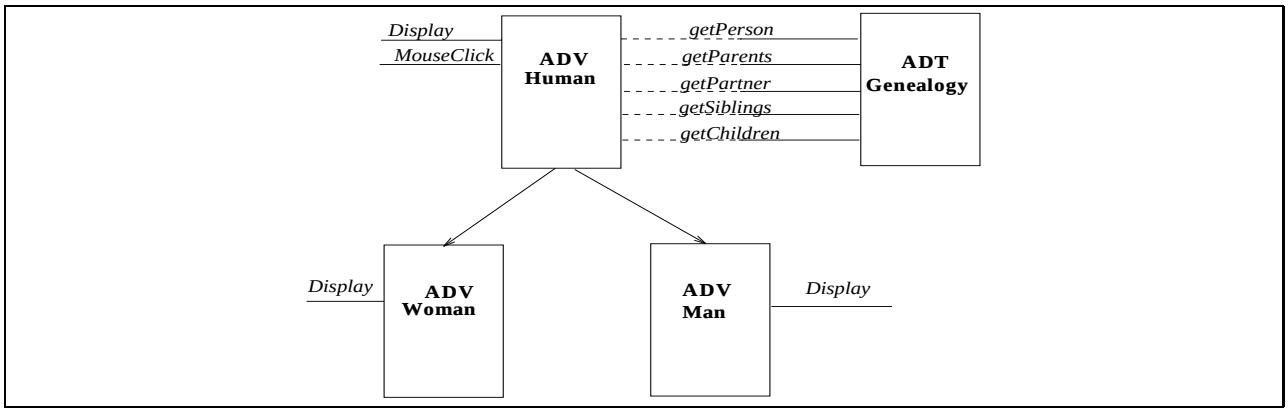


Figure 26: Configuration Diagram for the Human Problem

must be different for the ADV Woman and Man. This difference is shown in Figure 28 for the ADV Woman.

In Figure 28, we note that we have shaded some states and some ADVs. This visual effect is chosen to show that there are differences between the subtypes of the ADV Human and which states and ADVs are inherited. In the ADV Woman, we added one more state, to reflect the fact that the face of a woman is different from the face of a Man. The same approach is used for the ADV Man.

Some of the transition specifications for the HumanId problem are shown in Figures 29 and 30.

We observe in this example that decomposition by form, AND decomposition, and structural nesting are useful when specifying even a simple user interface. We believe that the "AND" decomposition and structural nesting simplify the specification of individual components of the user interface as they will be usually represented by small Statecharts. We also believe that decomposition by form simplifies the specification of the user interface as we can reuse part of a specification already defined.

In the next section, we discuss the semantics of the ADV-collection operator used in the ADVcharts.

8 ADV-Collection

The ADV-collection notation, using the * operator, can be used to represent two different situations: all the components of an ADV have the same behavior and operate independently or all the components of the ADV have the same behavior but do not operate independently. We call these two different situations case 1 and case 2, respectively.

Case 1 is the situation of the example in section 7.1—the Other Logon problem—where all the components of the Menu have the same behavior and operate independently.

Case 2 is used when we need to represent some relation (usually sequencing) among the components. One situation that illustrates this case is a menu problem (more realistic than the one presented in the Other Logon problem, in section 7.1), where we are allowed to move up and down through the menu items (either using the keyboard or using the mouse). To simplify the problem, we will leave out the first and the last items of the menu. A possible ADVchart for this problem is depicted in Figure 31. When the **down** event happens, we go from one state of the item that has the focus to one state of the next item of the menu, which will now have the focus. If the **up** event happens, the reverse operation occurs. So, there is a relation among the components of the menu and, therefore, the components are not independent. The semantics for this case, using Statecharts, are depicted in Figure 32.

We can conclude that the ***** operator can have different semantics and the semantics are connected to the type of relationship among the components that form the ADV-collection.

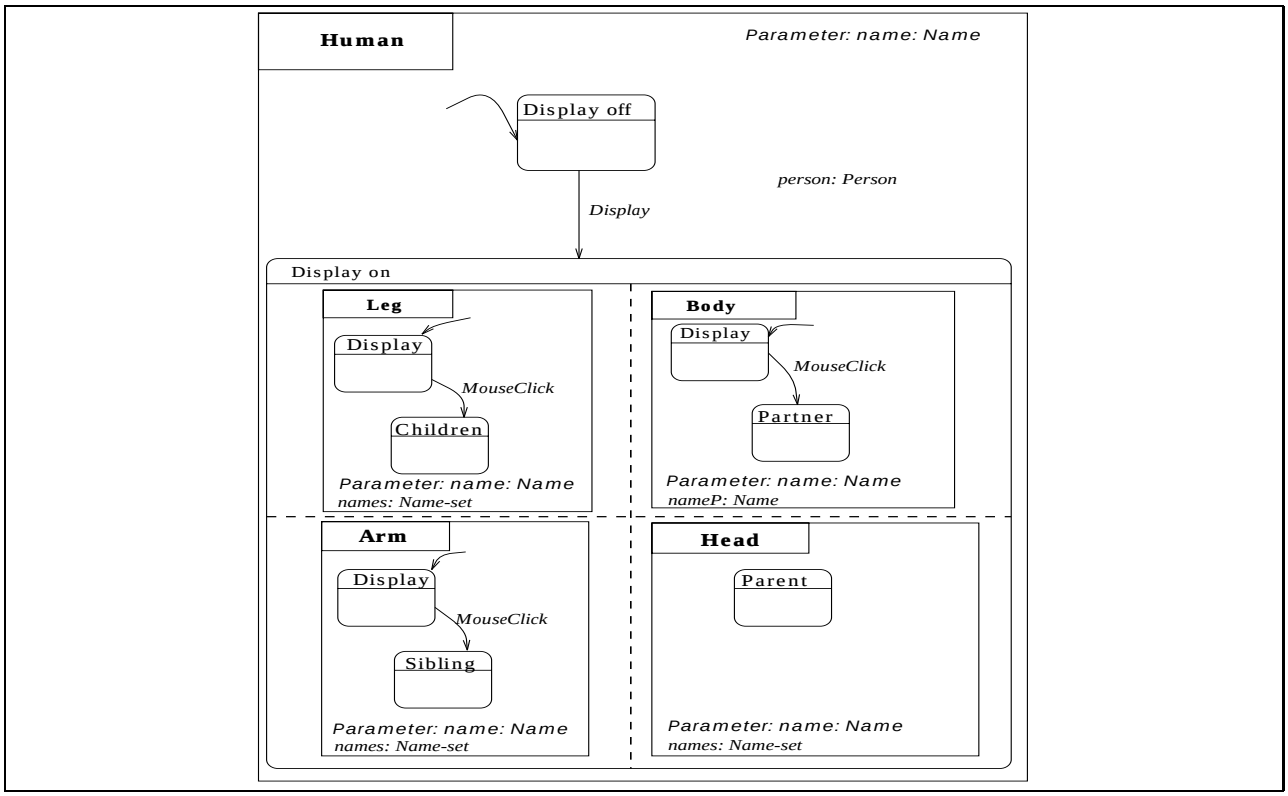


Figure 27: ADV Human

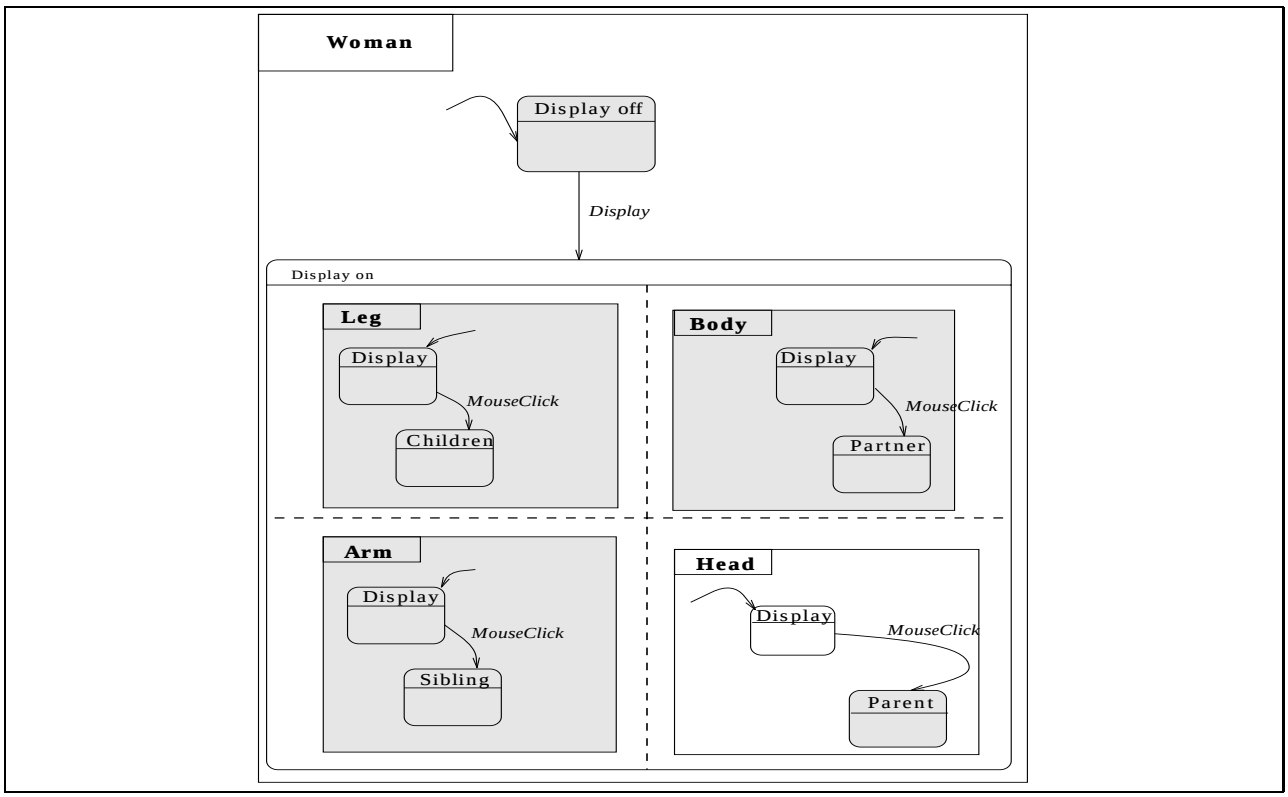


Figure 28: ADV Woman

Transitions Human

- → **Display off** :
  event :
  pre-condition : $\{true\}$
  post-condition : $\{screen = empty\}$
- **Display off** → **Display on** :
  event : Display(name)
  pre-condition : $\{true\}$
  post-condition : $\{person = \backslash owner.getPerson(name)\}$

Figure 29: Transitions for ADV Human

Transitions Woman

Transitions Head

- → **Display** :
  event :
  pre-condition : $\{person.sex = \text{“woman”}\}$
  post-condition : $\{screen = \overleftarrow{screen} + \text{“woman head”}\}$
- **Display** → **Parent** :
  event : MouseClick
  pre-condition : $\{true\}$
  post-condition : $\{names = \backslash owner.getParent(name) \wedge$
  $screen = \overleftarrow{screen} + names\}$

Figure 30: Transitions for ADV Woman

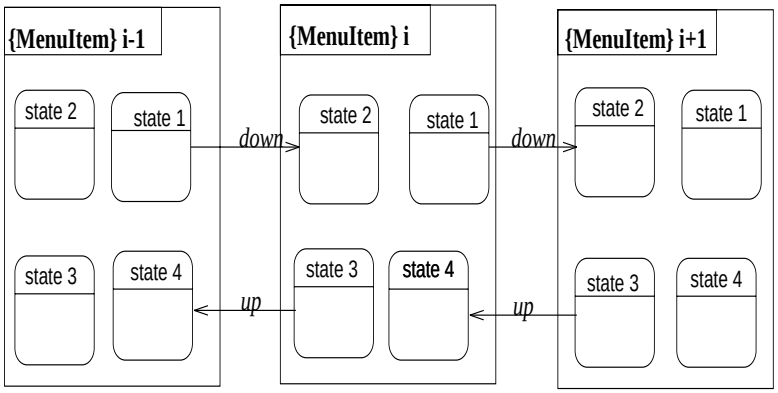


Figure 31: ADVchart for the a Menu

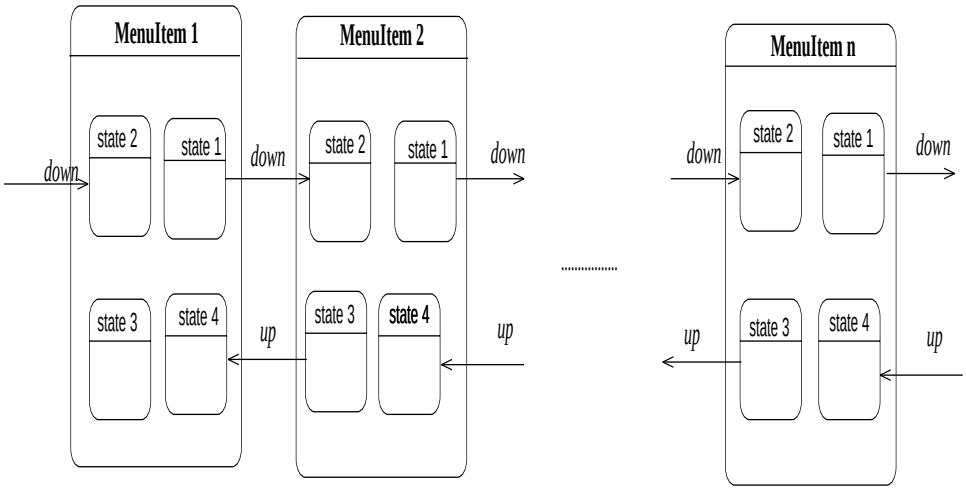


Figure 32: Statechart for a Menu

9 Mapping ADVcharts to a VDM-like specification

In this section we show that using some informal rules and ADVcharts and by proceeding in an incremental fashion, we can directly generate a VDM-like specification of ADVs in the same style presented in [7]. We use the example discussed in section 6 for purposes of illustration.

ADV *Logon* **For ADT** *UserLogon*
Declaration: $screen: Screen$
⋮
ADV *Prompt*
Declaration: $name: Name$
EVENT $KeyBoard\ (p: Point)$
⋮
End Prompt
ADV *Error*
⋮
End Error
EVENT *Display* ()
End Logon

Figure 33: First Phase of the VDM for ADV Logon

The specification of ADVs, as discussed in this paper, is composed of a configuration diagram, ADVcharts, specification of invariants over states, and specification of transitions. The first step in producing a VDM-like specification consists of transforming the configuration diagram, and the ADVcharts, into a template for the VDM specification. The template contains general information, such as ADV names, their associated ADTs, the event names (services provided), and the attribute names. From the configuration diagram, we get the information related to the associated ADT (the **For ADT** field); and from the ADVcharts, we get the nesting composition of the ADVs and the event names.

By examining the ADVcharts for the Logon example (Figure 10), we see that the ADV Logon is composed of two other ADVs: ADV Prompt and ADV Error. The events are taken from the services provided by the ADV specified in the configuration diagram and from the transitions in the ADVcharts. In this example, we see in the configuration diagram of Figure 9 that the ADV provides two services: Keyboard and Display. From the ADVcharts of the ADV Logon (Figure 10), we locate the position of these events (services). The event Display must be described in the ADV Logon, since it is an event assigned to a transition that ends in a state that belongs to the ADV Logon, namely the "**Display on**" state. Similarly, the event Keyboard must be described in the ADV Prompt, since it is an event assigned to a transition that ends in a state that belongs to the ADV Prompt.

The information in the declaration field of the VDM-like specification is taken from the ADVcharts, using the attributes declared inside the ADV boxes. The attribute "screen" is defined inside the ADV Logon box, and the attribute "name" is defined inside the ADV Prompt box. A template generated from analyzing the configuration diagram and the ADVcharts of the Logon problem is shown in Figure 33.

The second step is to specify the events, and to determine if there are any functions that need to be described in the ADVs. This information is in the transition specification. In the transition specification, we look at the event field to find the event we want to describe. The description of the event is in the pre-

ADV $Logon$ **For ADT** $UserLogon$
Declaration: $screen$: $Screen$
ADV $Prompt$
Declaration: $name$: $Name$
EVENT $KeyBoard\ (p: Point)$
external: wr $screen$
pre-condition:
$focus(Prompt)$
post-condition:
$name = GetUserName() \wedge$
$screen = addstr(\overleftarrow{screen}, name)$
End Prompt
ADV $Error$
Function $Display\ ()$
external: wr $screen$
post-condition: $screen = addstr(\overleftarrow{screen}, \text{“User Invalid”})$
End Error
EVENT $Display\ ()$
external: wr $screen$
post-condition: $screen = addstr(\overleftarrow{screen}, \text{“Login: ”})$
Function $Initialize\ ()$
external: wr $screen$
post-condition: $screen = empty$
End Logon

Figure 34: Second Phase ADV Logon

and post-condition fields of the transition specification. In our example, the event Display is assigned to the transition "**Display off → Display on**". We copy the pre- and post-condition fields of the transition specification of the event to the pre- and post-condition fields of the VDM-like specification, respectively. The same procedure is used for the event Keyboard. Functions are created for transitions that do not have events assigned to them. In our example, we have three transitions that belong to this category, and two of them have the same predicates defined in the post-condition. Therefore, we only need to define two functions, which we call *Initialize* and *Display*. The function Initialize corresponds to the two transitions that have the same post-condition, namely the transitions "**→ Display off**", and "**Input → Display off**", and it will be defined in the ADV Logon, since these transitions end in states that belong to the ADV Logon. The function Display corresponds to the transition "**Input → Error**", which ends in a state that belongs to the ADV Error. The post-conditions of these functions are simply a copy of the post-condition field of the transition specifications. The result of this second and last phase is shown in Figure 34.

Finally, we need to describe the flow of control. The flow of control can be specified using Statecharts, which already have a precise semantics [28, 22, 40]. We will also need some transformation rules to adapt the ADVcharts to pure Statecharts. Some of these rules were discussed in previous sections, such as the description of the synchronization transition in section 7.1, and the ADV-collection in sections 7.1 and 8. However, we still need rules to abstract the ADV from the ADVcharts in order to generate only states.

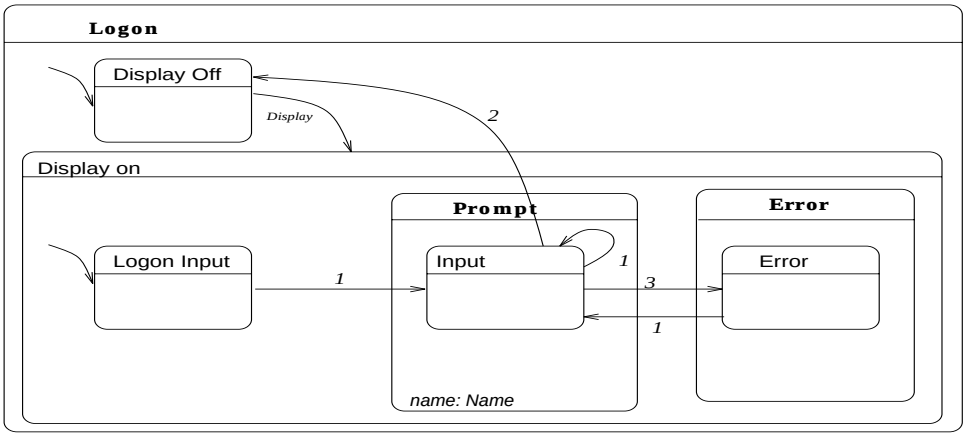


Figure 35: Statechart of Logon screen

We assume for the specification of the flow of control that an ADV is a clustering of states, and each transition is associated with a condition that connects the transition to a particular component of the user interface. The result of the application of this rule is shown in Figure 35.

The rules applied for the transformation of ADVcharts into a VDM specification and into pure Statecharts are quite simple. Thus, they guarantee the consistency of the specification of the operations and the flow of control with the ADVspec.

10 Conclusions

In this paper, we have introduced the ADVspec, which provides a visual formalization of the ADV concept and a notation for specifying the flow of control between the operations in the ADV. We expect the ADVspec to be used as the foundation for a future design methodology based on ADVs.

We showed some design issues that are important to interactive systems, such as the composition and concurrent operation of the components of the user interface. Furthermore, we showed that these aspects of design can be specified directly in ADVcharts using decomposition by structure, decomposition by form, and AND decomposition. We believe that other state machine based models for specifying user interfaces [39, 17, 35, 45, 42, 12, 44, 13, 31, 23] do not address all these design issues.

The semantics of ADVcharts have been presented through a set of examples in a style similar to the one used in [18] and [6].

We show how to represent similar components of a user interface, that is, to represent a collection of ADVs. Usually, we do not know how many components will be in the user interface. We presented the semantics associated with this kind of representation using Statecharts and also some examples to indicate the importance of this representation.

We also show how ADVcharts can be directly converted using simple informal rules to a VDM-like specification and to "pure" Statecharts. The VDM specification provides semantics for the operations and the Statechart specification provides semantics for the flow of control between operations.

There are a number of issues associated with ADVcharts that we will be examining in our future research. An important issue is the representation of recursion to allow ADV-based designs in situations such as the Graph Editor discussed in [8].

The ADVspec is being used to design interactive systems with complex user interface. For example, the structured editor Rita [10] is being re-designed using the ADVspec.

References

[1] D. Andrews and D. Ince. *Practical Formal Methods with VDM*. McGraw-Hill Software Engineering Series, 1991.

[2] M. M. Blattner and R. B. Dannenberg. CHI'90 Workshop on Multimedia and Multimodal Interface Design. *SIGCHI Bulletin*, 22(2):54–58, Oct. 1990.

[3] P. A. Buhr and C. R. Zarnke. Nesting in an Object Language is not for The Birds. In *ECOOP'88 – European Conference on Object-Oriented Programming*, 1988.

[4] L. M. F. Carneiro. A Specification-based Approach to User-Interface Design. Internal report, University of Waterloo, Dec. 1992.

[5] L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, and C. J. Lucena. User Interface High-Order Architectural Models. Technical Report 93–14, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, Feb. 1993.

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

[7] D. Cowan, R. Ierusalimschy, C. Lucena, and T. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, Jan. 1993.

[8] D. D. Cowan, L. Barbosa, R. Ierusalimschy, C. J. P. Lucena, and S. B. de Oliveira. Program Design Using Abstract Data Views–An Illustrative Example. Technical Report 92–54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, Dec. 1992.

[9] D. D. Cowan, R. Ierusalimschy, and T. M. Stepien. Programming Environments for End-Users. In *Proceedings of IFIP 92, Volume III*, pages 54–60, 1992.

[10] D. D. Cowan, E. W. Mackie, G. M. Pianosi, and G. d. V. Smit. Rita—An Editor and User Interface for Manipulating Structured Documents. *Electronic Publishing*, 4(3):125–150, 1991.

[11] J. Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.

[12] F. N. de Lucena and H. Liesenberg. Reflections on Using Statecharts to Capture Human-Computer Interface Behavior. Technical Report DCC-20/93, DCC/IMECC/UNICAMP, Campinas, SP, Brazil, 1993.

[13] E. Denert. Specification and Design of Dialogue Systems with State Diagrams. In *Proceedings of the International Computing Symposium, Liege*, pages 417–423. Elsevier, North Holland, 1977.

[14] S. W. Draper and D. A. Norman. Functional Refinement and Nested Objects for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 15, Jan. 1989.

[15] M. Green. Design Notations and User Interface Management Systems. In G. E. Pfaff, editor, *User Interface Management Systems – Proceedings of the Workshop on User Interface Management Systems held in Seeheim,FRG, November 1-3, 1983*. Spriger-Verlag, 1985.

[16] M. Green. Report on Dialogue Specification Tools. In G. E. Pfaff, editor, *User Interface Management Systems – Proceedings of the Workshop on User Interface Management Systems held in Seeheim,FRG, November 1-3, 1983*. Spriger-Verlag, 1985.

[17] M. Grochtmann. WINDOWNET—A Formal Notation for Window-Based User Interfaces. In H.-J. Bullinger and B. Shackel, editors, *Human-Computer Interaction—INTERACT'87*, pages 437–442. North-Holland, 1987.

[18] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[19] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[20] D. Harel. Biting the Silver Bullet — Toward a Brighter Future for System Development. *Computer*, pages 8–20, Jan. 1992.

[21] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On The Development of Reactive Systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[22] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *Proceedings of the Symposium on Logic in Computer Science*, pages 54–64, Ithaca, New York, June 1987.

[23] M. Harrison and A. Dix. A State Model of Direct Manipulation in Interactive Systems. In M. Harrison and H. Thimbleby, editors, *Formal Methods in Human-Computer Interaction*, pages 129–152. Cambridge University Press, 1990.

[24] R. Hartson. User-Interface Management Control and Communication. *IEEE Software*, pages 62–70, Jan. 1989.

[25] R. D. Hill. Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction—The Sassafras UIMS. *ACM Transaction on Graphics*, 5(3):179–210, July 1986.

[26] R. D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI'92 Conference Proceedings*, May 1992.

[27] C. Hoare. Proof of Correctness of Data Representations. In C. Hoare and C. Jones, editors, *Essays in Computer Science*, pages 103–115. Prentice Hall, 1989.

[28] J. J. M. Hooman and W. P. Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science*, 2(101):289–335, July 1992.

[29] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[30] R. Ierusalimschy. A Method for Object-Oriented Specifications with VDM. Monografias em Ciência da Computação, PUC - RJ, Rio de Janeiro, RJ, Brazil, Feb. 1991.

[31] R. J. K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transaction on Graphics*, 5(4):283–317, Oct. 1986.

[32] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.

[33] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *JOOP*, pages 26–49, Aug. 1988.

[34] C. J. P. Lucena, D. D. Cowan, and A. B. Potengy. A Programming Model for User Interface Compositions. In *Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, SIBGRAPI'92*,Águas de Lindóia, SP, Brazil, Nov. 1992.

[35] L. S. Marshall. Formally Describing Interactive Systems. In C. B. Jones and R. Shaw, editors, *Case Studies in Systematic Software Development*, pages 293–336. Prentice Hall, 1990.

[36] J. McCormack and P. Asente. An Overview of the X Toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 46–55, Oct. 1988.

[37] B. A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *UIST – Fourth Annual Symposium on User Interface Software and Technology*, pages 211–220, 1991.

[38] D. R. Olsen, Jr. Presentational, Syntactic and Semantic Components of Interactive Dialogue Specifications. In G. E. Pfaff, editor, *User Interface Management Systems – Proceedings of the Workshop on User Interface Management Systems held in Seeheim,FRG, November 1-3, 1983*. Spriger-Verlag, 1985.

[39] D. L. Parnas. On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System. In *Proceedings of the 24th National ACM Conference*, pages 379–385, 1969.

[40] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In T. Ito and A. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 244–264. Springer-Verlag, 1991.

[41] W. Reisig. *Petri Nets : an Introduction*. Springer-Verlag, 1985.

[42] J. Rumbaugh. State Trees as Structured Finite State Machines for User Interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 15–29. ACM SIGGRAPH, ACM Press, Oct. 1988.

[43] J. Uhl and H. A. Schmid. *A Systematic Catalogue of Reusable Abstract Data Types*. Springer-Verlag, 1990.

[44] A. J. Wasserman. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, 8(11):699–713, Aug. 1985.

[45] P. D. Wellner. Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *Human Factors in Computing Systems and Graphics Interface—CHI'89*, pages 177–182, May 1989.