An Experience Using JASMINUM—Formalization Assisting With the Design of User Interfaces

L.M.F. Carneiro-Coffin[1] * D.D. Cowan[1] C.J.P. Lucena[2] D. Smith[1]

[1] Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ont, Canada
[2] Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, RJ, Brazil

Resumo

Descrevemos neste artigo a nossa experiência no uso de um formalismo visual, chamado JASMINUM , que é usado na etapa de design da interface com o usuário de um editor de documentos estruturado chamado Waterloo Rita. JASMINUM combina conceitos de Abstract Data Views com as notações de Objectcharts, Statecharts e redes de Petri. JASMINUM foi motivado pela necessidade de uma notação formal que expressasse diretamente alguns aspectos do design de sistema interativos, especialmente a possibilidade de representação do design do sistema interativo através da interface com o usuário.

Nosso principal objetivo neste artigo é mostrar como o uso de JASMINUM ajudou a apontar certos erros no design do sistema Waterloo Rita , mostrando com isto que JASMINUM pode ser uma notação útil para a re-engenharia de sistemas.

Abstract. This paper describes experience in using a visual formalism, called JASMINUM, in the design of the user interface of a structured document editor called Waterloo Rita. JASMINUM combines concepts from Abstract Data Views, with notations from Objectcharts, Statecharts, and Petri-nets. JASMINUM was motivated by the need for a formal notation to express directly some aspects of the design of interactive systems, especially the design of an interactive system through its user interface. In particular, we show in this paper how the use of JASMINUM helped to find some design flaws in the existing Waterloo Rita system, demonstrating that JASMINUM might be a useful notation for re-engineering systems.

1 Introduction

A number of authors including [32, 14, 13, 19, 29, 24, 18, 20] 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 shown the feasibility of this separation [26] and discussed the many advantages of dialog independence including ease of maintenance of the existing system, support for alternative visualizations of an application data type [30], and design and application reuse.

A new user-interface design approach, called *Abstract Data Views* (ADVs), was presented by Cowan *et al.* [10, 8]. The ADV design approach provides a clear separation

* L.M.F. Carneiro-Coffin holds a doctoral fellowship from CAPES (the Brazilian Research Council).E-mail address: lmfcarne@neumann.uwaterloo.ca

of the user-interface (UI) from the non-user-interface (NUI) part of the system at the design level. The concept of ADVs has been applied to the design of interactive systems, where the UI part of the system is represented as an ADV, and the NUI part (the application) is represented as an Abstract Data Type (ADT) [36].

A design approach such as ADVs is only useful if it can eventually be implemented through some form of reification that maps the design into an operational system. Many different software systems have been designed and implemented based on the ADV design approach. For example, ADVs have been used to support user interfaces for games and a graph editor [9], to interconnect modules in a user interface design system (UIDS) [25], to support concurrency in a cooperative drawing tool, and to implement a ray-tracer in a distributed environment [33]. A research prototype of the VX·REXX [37] system was motivated by the idea of composing applications in the ADV/ADT style.

Although the ADV design approach appears to be promising, a number of issues still needed to be addressed. Specifically, we needed to have an associated formal design model to ensure that any design based on the ADV design approach was developed correctly and systematically, and to incorporate the specification of the flow of control between operations. The use of a graphical or visual formalism [16] to assist in producing a specific design is particularly appealing since designers often use various graphics to assist in producing a specific design.

We have created a new visual formalism called JASMINUM [5] for designing interactive systems. This visual notation is state-machine oriented and is based on earlier research on visual formalisms (Statecharts [15] and Objectcharts [6]). This notation has a number of advantages over other visual formalisms for specifying interactive systems since it handles aspects of design specific to these types of systems, including the association of pointing devices with a particular component of a user interface (focus of control), the specification of concurrent operation of the components of a user interface, and the relationship of the various components of a user interface. Because of these advantages, we believe JASMINUM supports a user-centered design approach to interactive systems in that the appearance of the design strongly resembles the appearance of the user interface.

JASMINUM has been used to design user interfaces for a number of software systems including a graph editor [9] and a structured document editor called Waterloo Rita [11]. The design of the Waterloo Rita user interface was compared with an earlier design, and from that comparison, a number of design flaws in the original system were detected. In particular, we observed that the same functionality was achieved with a simpler and more consistent design. These design flaws were detected because the user-interface design closely resembled the actual user interface, whereas the original design did not clearly specify the relationship and similarity of many of the user-interface components.

In this paper, we introduce the ADV design approach and JASMINUM. In particular, we concentrate on the *ADVcharts*, a key component of JASMINUM. ADVcharts allow the designer to specify the relationship among user-interface components, as well as both concurrent operation and focus of control. We also describe some of the design flaws that were detected when the JASMINUM approach was used to re-design Rita.

2 The Abstract Data View Design Approach

The Abstract Data View (ADV) is a new design approach that clearly separates the UI from the NUI part of an interactive system and promotes design reuse. Further, the ADV design approach promotes reuse of specifications of UI components in that it allows the construction of complex UI components from simpler ones. In the ADV design approach, Abstract Data Types (ADTs) [21] are completely independent of the user interface and, in fact, do not have access to any input or output. The initial formal semantics of the ADV design approach have been given in [8], using an extension of the Vienna development method specification language (VDM-SL)[12, 3, 23].

ADVs and ADTs are types, from which instances can be created. From now on, the word ADV alone refers to an ADV type and the word ADT alone refers to an ADT type.

ADVs and ADTs are described using the ADV Description Language (ADV-DL). ADV-DL is based on a variant of VDM-SL [22, 8]. We have extended ADV-DL to describe components that are collections and to support inheritance of specifications [5]. A formal semantics for the specification constructors used in ADV-DL, given in terms of textual substitution in a manner similar to that used by Hoare [21], is presented in [1, 2].

In this paper, we view an ADT as being defined by both a type domain whose internal representation is unknown to clients and a set of operations defined on the domain [7] comprising a public interface. Only the public interface of an ADT is accessible to clients. The state of an ADT instance is only modified through operations. The term ADT is used because we are mostly interested in its properties as a type. The ADT is completely independent of the UI part of the system and, in fact, does not specify any input or output.

ADVs [10, 8] are Abstract Data Types (ADTs) [36] that have been modified to support the design of user interfaces. An ADV description contains definitions of all event-driven input and output operations, definitions of all visual objects, and functions for manipulating these visual objects. All design decisions related to passing information between the user and the system are specified in the ADV. These specifications include sequencing of information between the NUI and the UI parts of the system and the connection between these two parts.

Figure 1 shows the relationship between input and output devices, ADVs, and ADTs. The relationship between ADVs and ADTs, shown as an arrow, associates the public interface of the ADT with the corresponding specification in the ADV. The name of the corresponding ADT in the ADV specification is represented by a variable called **owner**,[3] which provides the connection between the UI and NUI parts of the system.

The relation between an ADV and an ADT is not symmetric, that is, there is a variable **owner** in the ADV but there is no equivalent variable in the ADT. Since an ADT has no knowledge of input or output, it does not need to refer to any ADV. As a consequence, the ADT does not need to know about its corresponding ADVs.

A typical design that is based on the ADV design approach consists of a collection of ADTs that manage the data structures and state of the NUI part of the system, a collection of ADVs that comprise the perceivable behavior—the UI part of the system—, and a

[3] Because an ADV is associated with an ADT, we say that the ADT is the owner of the ADV.

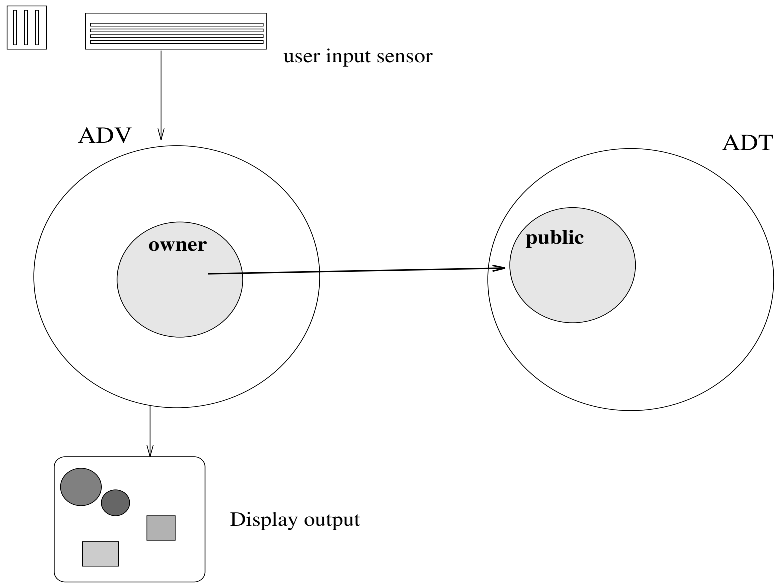

Fig. 1. The ADV Model

mapping from ADVs to ADTs [4]. The independence of ADTs from ADVs guarantees the property of a clean separation of the NUI from the UI parts of the system.

The ADV provides all event-driven input and output handling functions, provides views of the ADT to the user by using the ADT public interface, and fully controls its associated ADT.

A user interface can provide many different views of the data stored in an application. For example, an integer could be displayed as a number or a position on a dial. For this reason, the ADV concept allows the association of several ADVs with a single ADT, where each 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.

We believe that user interfaces are built by composing behavior, structure, and form. The behavior of a user interface can be viewed as the composition of simpler behaviors in just the same way a complex routine is composed from a number of simpler routines. For example, under the Microsoft Windows operating system a window can behave, roughly speaking, in two ways: iconized or expanded. Similarly, a user interface is composed from a number of visual objects. For example, a dialogue box may be composed of a text-field, radio buttons, and check boxes. The different forms of the objects represented in a user interface can be classified into groups of related abstractions. For example, human beings can be classified into two groups: men and women. A user interface that displays human beings will make some distinctions between men and women, such as the display of the face. The form of composition where the behavior or structure of an object is composed of simpler behaviors or structures we call *behavioral and structural nesting*, respectively. The form of composition where the form of the objects are classified into groups we call *decomposition by form*. The ADV concept supports behavioral and structural nesting, and decomposition by form.

3 JASMINUM

JASMINUM stands for **J**oining **A**DVs and **S**tate **M**achines **i**n a **N**otation for **U**ser-Interface **M**odeling. JASMINUM is a state-machine-based visual approach for specifying interactive systems that are modeled using the ADV concept. JASMINUM has a number of advantages over other visual formalisms for specifying interactive systems, since it incorporates several composition mechanisms that promote reusability and it associates explicitly the UI and the NUI parts using the dialogue-independence principle. These mechanisms are incorporated in the ADV design approach and JASMINUM uses them.

Our work on JASMINUM was inspired by earlier research on visual formalisms. Three design methods of particular assistance were Statecharts [15], Objectcharts [6], and Marshall's approach to user-interface specification [28, 27]. All these design methods use a state-machine approach for specifying complex systems. Statecharts are an extension of state machines and state diagrams and are used to specify and design complex reactive systems [17]. Objectcharts are an extension of Statecharts and are used to characterize the behavior of classes in object-oriented systems as state-machines. Marshall's approach is a hybrid approach (it uses VDM and a subset of Statecharts) and is used for specifying user interfaces.

JASMINUM is a hybrid model, based on visual and textual formalisms, for specifying interactive systems. The main characteristics of JASMINUM are support for reusability (that is, reuse of user-interface and application (NUI) components) and a smooth integration of visual and textual formalisms.

JASMINUM has four components: a configuration diagram, ADVcharts, transition specifications, and invariant specifications. In the configuration diagram, we specify the interface between the ADV and its associated ADT, and represent the ADV inheritance tree, in the same style as Objectcharts. ADVcharts, a key component of JASMINUM, are an extension of both Statecharts [15] and Objectcharts [6], and use notations from Petri-nets [34]. As ADVcharts are based on finite-state machines, we need to specify the input alphabet. External (user-initiated) and ADV-initiated events are modeled as provided services by the ADV, because ADVs provide services to the user that generates the event and to the 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. Consequently, the input alphabet for ADVcharts consists of the services provided and required by an ADV, and conditions on attributes of the ADV.

Behavior of an ADV is represented by associating states with the ADV and behavioral nesting is supported by nesting of states in a manner similar to Statecharts and Objectcharts. Structural nesting is represented in ADVcharts through the nesting of ADVs. Structural nesting is not supported by Statecharts and Objectcharts. Decomposition by form is represented in JASMINUM through the representation of the several ADVs that compose the design of the UI part of the system.

ADVcharts generalize Statecharts and Objectcharts to deal with 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 interface are clearly indicated. Because we are specifying user interfaces with a collection of independent concurrently operating ADVs, we need a notation to express synchronization. For that purpose, we have introduced some notation from Petri-nets into ADVcharts.

3.1 An Illustrative Example

In this section, we illustrate the use of JASMINUM by presenting the specification of an hour clock. An hour clock, depicted in Figure 2, displays the actual hour on a clock face. For instance, if it is 3am, the hand of the clock points to 3; if it is 3:30am, represented by the number 3.5, the hand of the clock points to between 3 and 4. The user can set the time of the clock by positioning the mouse cursor on the hand of the clock, clicking the mouse button, and positioning the mouse cursor on the time the user wants. A machine (a clock) generates the event **ClockInterrupt** that should be handled by the hour clock.

The design of the hour clock is composed of ADVs only, because there is no other functionality required. That is, in this example, we do not need to specify semantic feedback from an associated application; all the functionality is available in the user-interface part of the system.

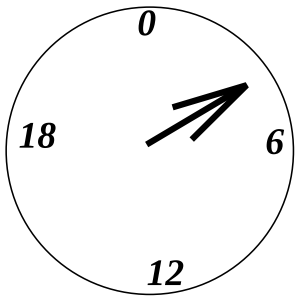

Fig. 2. User Interface of the Hour Clock

Given the statement of the hour-clock problem, we associate the objects that appear on the user interface with ADVs and specify the configuration diagram. As shown in Figure 3, the clock is associated with ADV Clock and the hand of the clock is associated with ADV Hand.

In the statement of the problem, we also identify the services provided and required by the ADV Clock and ADV Hand. ADV Clock provides two services: handling the clock interrupt and displaying the clock. ADV Hand provides four services: displaying the hand of the clock and handling mouse operations, namely Mouse Down, Mouse Move, and Mouse Up. The configuration diagram is shown in Figure 4.

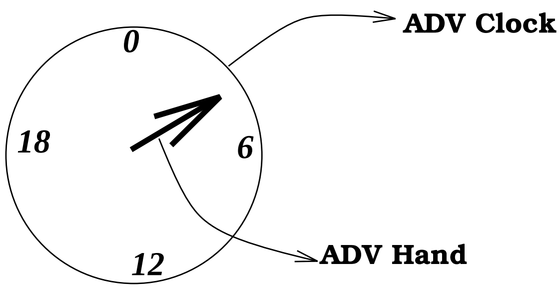

Fig. 3. User Interface of the Hour Clock with the associated ADVs

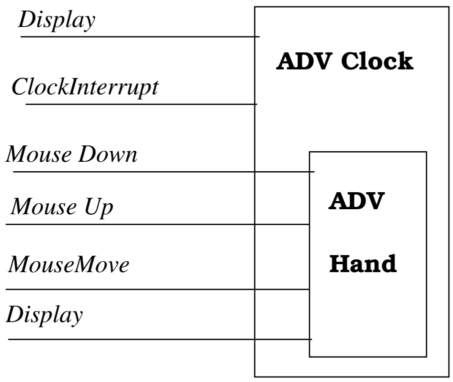

Fig. 4. Configuration Diagram for the ADV Clock and ADV Hand

As shown in Figure 3, the ADV Clock is composed of one other ADV, namely the ADV Hand. The mechanism of allowing an ADV to be composed of two other ADVs is referred to as structural nesting, as explained in Section 2. In an ADVchart, this mechanism is made visually explicit by showing ADVs drawn inside ADVs, as depicted in Figure 5.

The ADV design approach has scope rules that restrict access to *attributes* of ADVs.[4] Attributes are variables together with their corresponding values [6]. Attributes can be declared in ADVs. The scope of an attribute declared in an ADV is the ADV itself. That is, ADVs that compose another more complex ADV do not have access to the attributes declared in this more complex ADV. This visibility rule promotes reusability because an ADV can only reuse attributes and ADVs that belong to their internal environment. ADVs cannot reuse attributes and ADVs from their external environment because they do not have access to them. The same rules that are applied to ADVs, are applied to ADTs.

The ADV Hand contains one parameter, namely `hour`. We need this parameter to

[4] The variable **owner** of an ADV is also considered an attribute. We do not use the variable **owner** in this example because we do not have an ADT associated with the ADV Clock.

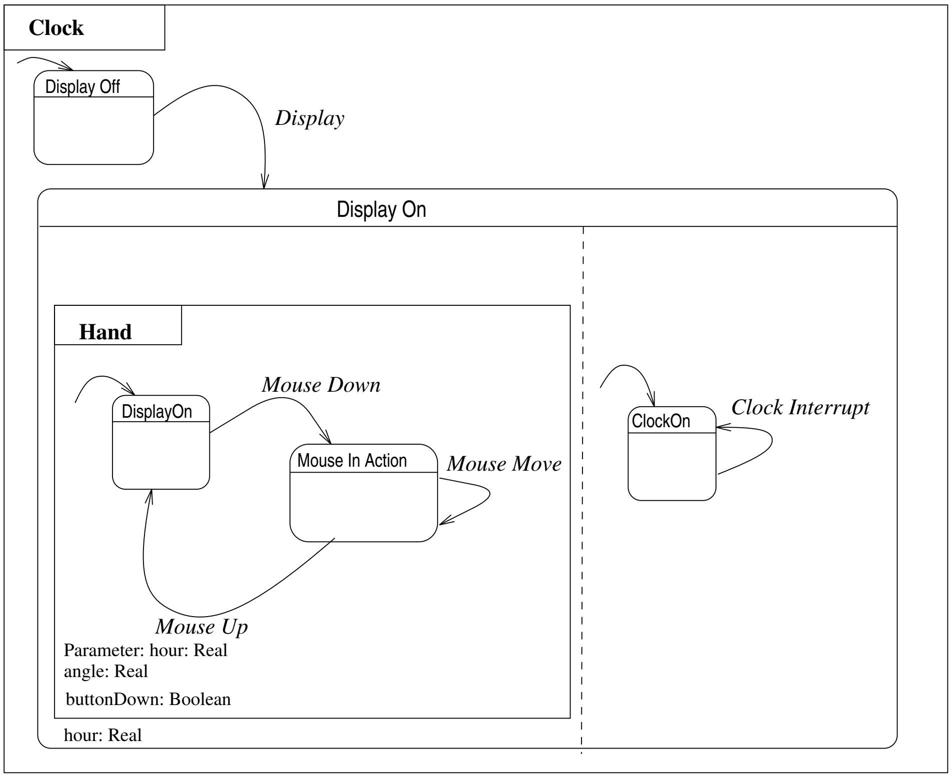

Fig. 5. ADVchart for the ADV Clock

connect the attribute `hour` of the ADV Clock to the angle which the hand of the clock is displayed. In other words, we specify that the parameter `hour` of the ADV Hand is the attribute `hour` of the ADV Clock. In general, an ADV can have a list of parameters. That is, if we specify the ADV Hand separate from ADV Clock, the specification would contain a header like "**ADV Hand (hour:** $\mathbb{R}$)". When reusing this ADV for building a more complex ADV, we need to connect the parameter with an attribute of the more complex ADV.

In Figure 5, the behavior of ADV Clock is specified to have two modes: **Display On** and **Display Off**. When the clock is displayed (including the hand of the clock), the system concurrently handles both the events generated by the mouse device and the clock interrupt. That is, the system is in the state **<Hand.DisplayOn,ClockOn>**. This does not mean that the system is in two states at the same time: this is a method for representing concurrency and is similar to statecharts [15].

Every time the system receives a clock interrupt, the attribute `hour` of the clock is updated. In order to set the time of the clock, the user moves the mouse to position the cursor on the new time. This is done by initially pressing the button of the mouse down, moving the mouse, and, finally, releasing the mouse button. Moving the mouse implies an update of the angle of the hand and, consequently, the time of the clock. (This will be guaranteed by an invariant specified for the ADV Hand.)

The next step is to describe, in detail, 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 closer to VDM. The transitions for the ADVs Clock and Hand are described in Figures 6 and 7, respectively.

Transitions Clock

- $\rightarrow$ **Display off** :
  event :
  post-condition : $\{screen = empty\}$
- **Display off** $\rightarrow$ **Display on** :
  event : Display
  pre-condition : $\{true\}$
  post-condition : $\{\text{The clock is drawn on the screen}\}$
- $\rightarrow$ **ClockOn** :
  event :
  post-condition : $\{\}$
- **ClockOn** $\rightarrow$ **ClockOn** :
  event : $\text{ClockInterrupt}(\text{time})$
  pre-condition : $\{true\}$
  post-condition : $\{time.hour = hour\}$

Fig. 6. Transition Specification for ADV Clock

As shown in Figure 6, a transition specification is composed of four fields: the transition, the event, the pre-condition, and the post-condition. The transition is described by means of a state, an arrow, and the name of the next state. For example, the transition from state **"Display off"** to the state **Display On** is written as "**Display off → 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 event field, we specify the events that must occur in order to fire the transition. In the pre-condition field, we specify the conditions that must hold (after the event happens) in order to fire the transition. In the post-condition field, we specify the conditions that must hold once the transition is fired. For instance, in Figure 7, to fire the transition from state **Display On** to state **Mouse In Action** the event *Mouse Down* must occur and the attribute `buttonDown` must be false. If the pre-condition is satisfied, the attribute `buttonDown` is set to true.

The event **ClockInterrupt**, in Figure 6, contains one parameter, namely `time`, with type `Time`. The type `Time`, which is not declared in this specification, is a record composed of two fields, `hour` and `minute`.

In the post-condition of the transition **Mouse In Action → Mouse In Action**, we use

Transitions Hand

- $\rightarrow$ **DisplayOn** :
  event :
  post-condition : $\{\text{The hand is drawn on the screen} \wedge$
  $buttonDown = \mathit{false}\}$
- **Display on** $\rightarrow$ **Mouse In Action** :
  event : $\mathrm{MouseDown}$
  pre-condition : $\{buttonDown = \mathit{false}\}$
  post-condition : $\{buttonDown = \mathit{true}\}$
- **Mouse In Action** $\rightarrow$ **Mouse In Action** :
  event : $\mathrm{MouseMove(mousePosition)}$
  pre-condition : $\{buttonDown = \mathit{true}\}$
  post-condition : $\{post\text{-}atan2(mousePosition.y, mousePosition.x, angle)\}$
- **Mouse In Action** $\rightarrow$ **Display On** :
  event : $\mathrm{MouseUp}$
  pre-condition : $\{buttonDown = \mathit{true}\}$
  post-condition : $\{buttonDown = \mathit{false}\}$

Fig. 7. Transition Specification for ADV Hand

the predicate *post-atan2(mousePosition.y,mousePosition.x,angle)*. When using VDM, since pre- and post-conditions are non-procedural relations, one operation cannot invoke another. Therefore, we use the predicate *post-operation_name(parameters,result)*, where *operation_name* is the name of the operation, *parameters* are the parameters of the operation, and *result* is the result of the operation. The predicate *post-atan2(mousePosition.y,mousePosition.x,angle)* means that we want the post-condition of the function `atan2` to be satisfied and the result of the attribute `angle` to hold the value returned by this function.

The final step is to specify invariants, if needed. In our example, ADV Hand requires an invariant to state that whenever the parameter `hour`, which corresponds to the attribute `hour` of the ADV Clock changes, the attribute `angle` of ADV Hand also changes, and vice-versa. In the invariant formulae, shown in Figure 8, we specify that each hour corresponds to 15 degrees.

Invariant(s) for ADV Hand :
$hour = (angle/15.)$

Fig. 8. Invariant Specification for ADV Hand

4 A Case Study—Waterloo Rita

Waterloo Rita is a structured document editor. The Waterloo Rita system was chosen to be a case study because we consider this software system a fairly large one and we wanted to see how well JASMINUM works for larger systems.

The main screen of the Waterloo Rita system, depicted in Figure 9, consists of a menu bar at the top of the screen, a status bar in the top right corner (not shown in the figure), a list of commands at the bottom of the screen, an area in the middle of the screen that displays the actual document, and an area on the left that displays the structure of the document that is being displayed. The status bar indicates the state of certain control keys.

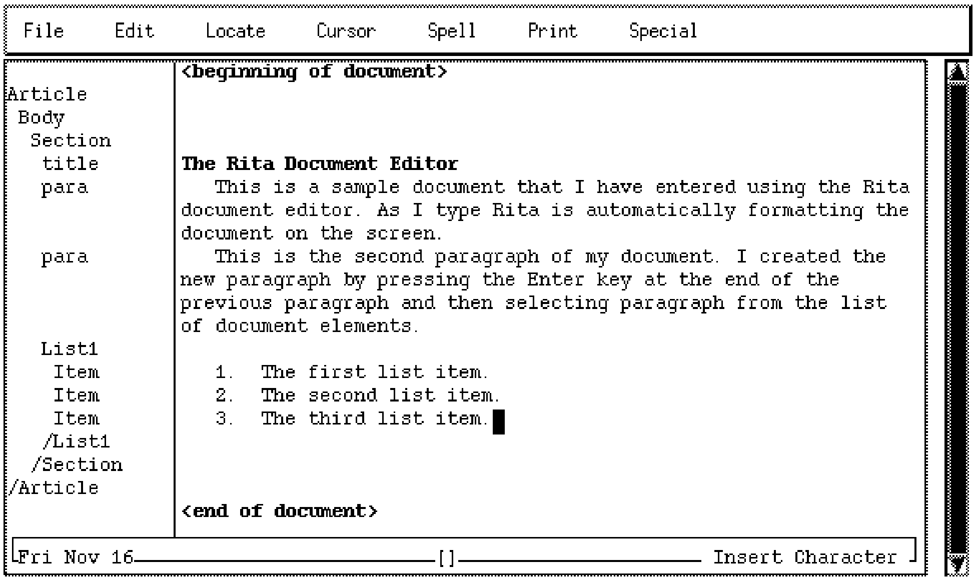

Fig. 9. The Waterloo Rita System User Interface

In the same style that we used in the previous example, each object in the user interface of Waterloo Rita system is associated with an ADV. The association of ADVs with various parts of the Waterloo Rita system user interface is depicted in Figure 10. The ADV StructuredText represents both the document and the structure display. The corresponding ADVchart for the Waterloo Rita system user interface is depicted in Figure 11.

We observe from the ADVchart in Figure 11 that the Waterloo Rita system user interface is comprised of the ADV Status Bar and the ADV Rita Editor. The ADV Rita Editor is composed of the ADVs StructuredText, Menu Bar, and Commands. In the design the ADV Status Bar is considered to operate independently from the ADV Rita Editor. Thus these two ADVs are represented using AND decomposition. In other

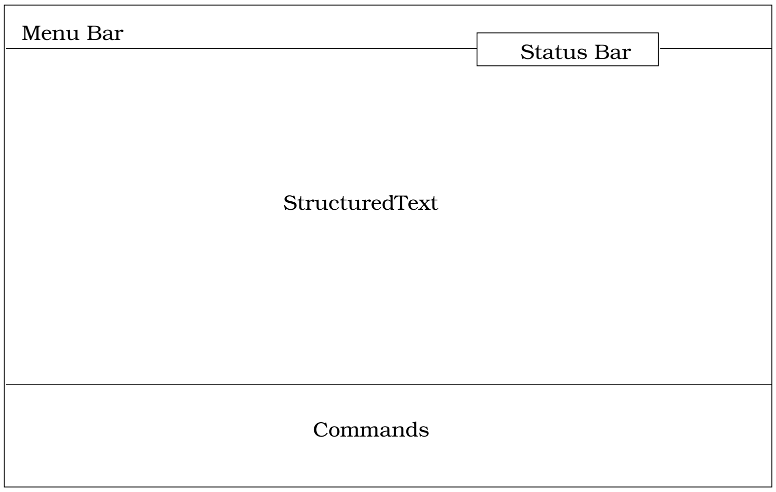

Fig. 10. The Association of the Waterloo Rita System User Interface with ADVs

words, the behavior of the control keys is independent of the editing of the text.

The transition **Exit Program** in Figure 11 indicates that there is a state within the ADV Menu Bar that has a transition annotated with the event **Exit Program**. The possibility of having a transition specified in this manner, that is within an ADV, was a consequence of this case study: it is important to specify the flow of control at a high level of abstraction.

The complete specification of the Waterloo Rita system is found in [35]. In this section, we focus on the specification of some of the ADVcharts for the Waterloo Rita system, more specifically the ADV Menu Bar, since we encountered a problem in an earlier design in specifying this ADV in a simple way.

The ADVchart for the ADV MenuBar, depicted in Figure 12, is composed of a set, called `menus`. The elements of the set `menus` have some states in common, that is, they behave in a similar, but not identical manner. The set `menus` is composed of specializations of the ADV Menu, namely ADVs ControlMenu, FileMenu, EditMenu, LocateMenu, CursorMenu, SpellMenu, and PrintMenu. All these ADVs, from ADV ControlMenu to ADV PrintMenu, have the same behavior with respect to starting in state **Inactive** and going from state **Inactive** to state **DropDown**. We expect that the specification of state **DropDown** differs for some or all the elements that compose the set `menus`.

In this section, we only describe, in more detail, the ADV EditMenu to illustrate how ADVcharts helped us locate a design flaw in the original Waterloo Rita system user interface.

The ADV EditMenu is composed of several ADVs (it is a pull-down menu). These ADVs are called Attributes, Insert, Transform, Clear, Cut, Paste, CopytoClipboard, PasteFromClipboard, and SplitBlock. The way these ADVS were described in the original design, their behavior was not consistent and, consequently, could not be described

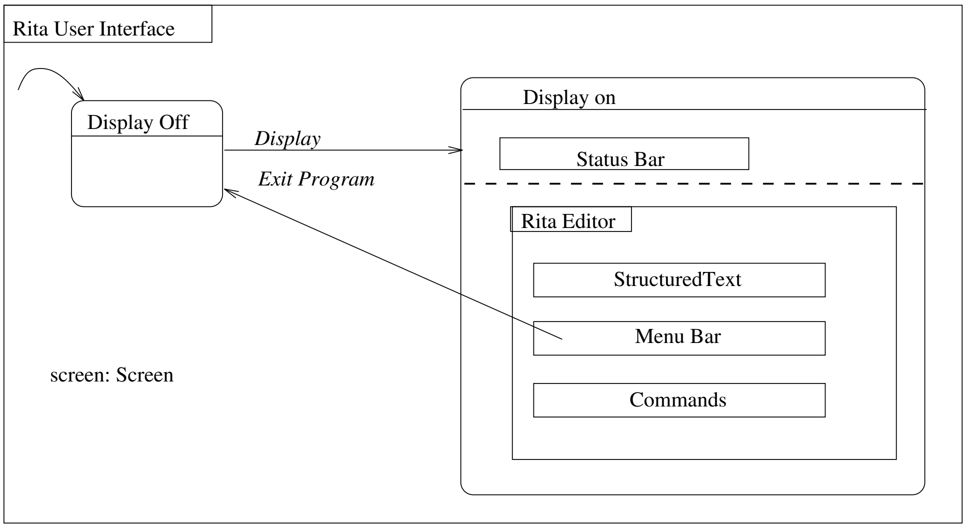

Fig. 11. The ADVcharts for the Waterloo Rita System User Interface

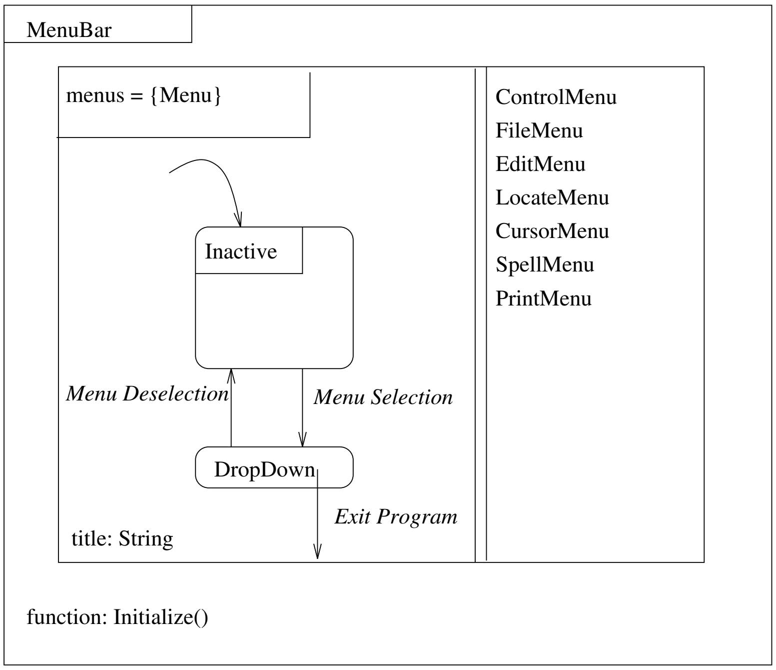

Fig. 12. ADVcharts for ADV MenuBar

using the set operator. For example, the ADV CopytoClipboard was always bright, and if activated without selecting a text block, an error message was displayed. On the other hand, the ADV Clear was dimmed until some text was selected. We consider this inconsistent behavior a design flaw, as the user might expect the same bright-dim behavior for all menu items.

We believe this design fault was easily noticed because we used JASMINUM. With JASMINUM, the designer uses the AND decomposition, and decomposition by behavior, structure, and form in a natural design process of successive refinement. The existence of the set operator also plays a significant role in the specification, as this operator will be the first choice for the designer when modeling a situation where an ADV is composed of several other ADVs with similar behavior.

ADVcharts led us to re-design the ADV EditMenu by considering all menu items associated with the ADV EditMenu dimmed until a text block is selected. This specification is depicted in Figure 13.

As a result of this case study, we can conclude that the re-design of Waterloo Rita system using JASMINUM highlighted some design flaws in the original interface and some simple enhancements that should be made to ADVcharts to support the design of larger software systems. In addition, because JASMINUM provides mechanisms to represent decomposition by behavior, structure, and form, it naturally supports successive refinement. These mechanisms, especially decomposition by structure, tend to identify collections of components with similar behavior in the design process. The representation of several components with similar behavior is very simple when using JASMINUM: we use the set and sequence notations.

5 Conclusions

In this paper, we have shown how to use JASMINUM to describe some aspects of the operation of a user interface. In particular, we show the specification of two software systems. One of these systems, the Waterloo Rita, is a fairly large software system and appears to be a good test of the JASMINUM notation.

We presented ADVcharts, a component of the JASMINUM notation, that is a state-machine-based visual design formalism for specifying interactive systems. The re-design of the user interface for Rita using JASMINUM highlighted some design flaws in the original user interface.

Because JASMINUM provides decomposition by behavior, structure, and form, it naturally supports successive refinement, which tends to identify early in the design process, among other things, collections of components with similar behavior . Furthermore, the specification of Waterloo Rita demonstrated that JASMINUM can be used to specify fairly large software systems, and that it helps in making design decisions because it is a formalism that reflects the structure of the part of the system it is aiming to describe.

One of the problems we found when using JASMINUM for specifying large software systems, was the lack of an automated tool for assisting in the drawing of charts and for verifying the design against the syntactic rules of JASMINUM. A first attempt to solve this problem is discussed in [31].

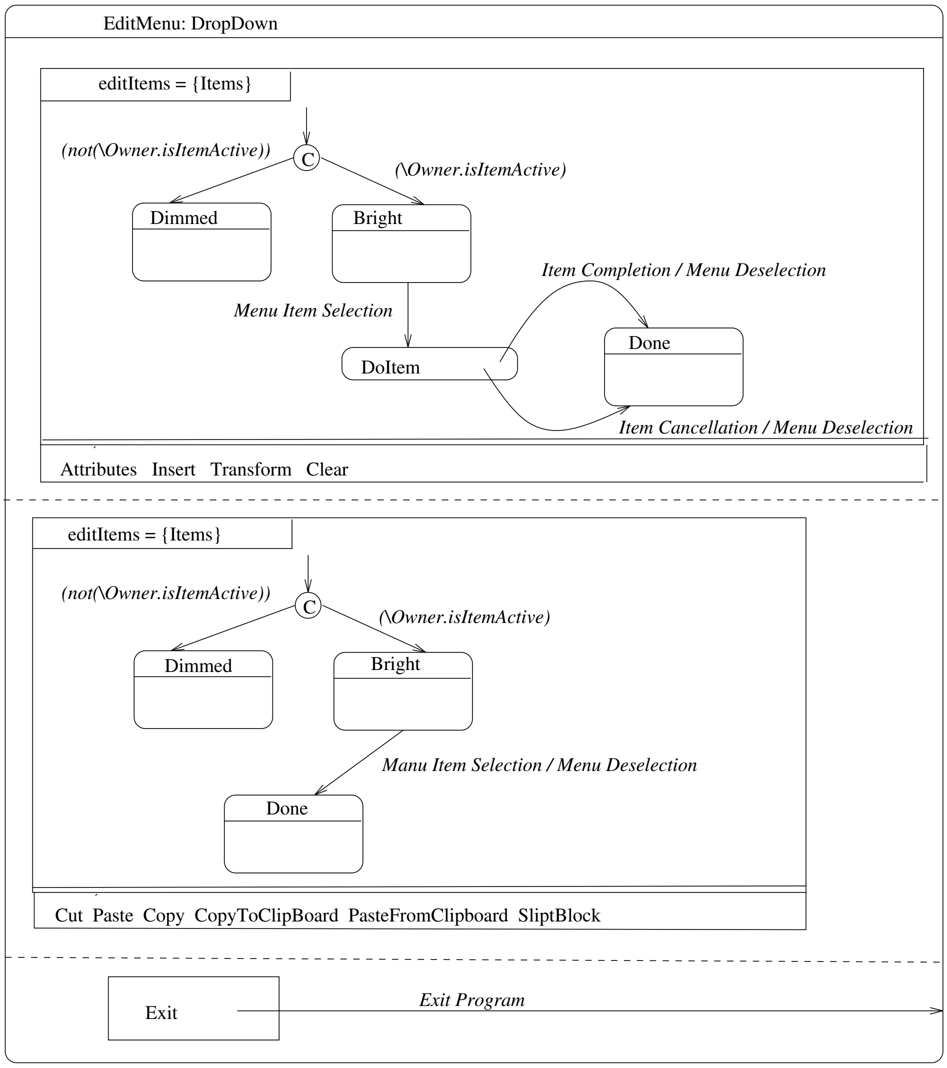

Fig. 13. ADVcharts for the ADV Edit Menu

We believe that a formalism that reflects the structure of the part of the system it is aiming to describe is helpful in the design, and, consequently, in the development of that part of the system. We believe JASMINUM is one of these formalisms.

References

1. P. Alencar, L. M. F. Carneiro-Coffin, D. D. Cowan, and C. J. P. Lucena. The Semantics of Abstract Data Views: A Design Concept to Support Reuse-in-the-Large. In *The 62nd*

Congress of l'Association Canadienne Française pour l'Avancement des Sciences (ACFAS), pages 49–60, Montreal, Canada, May 1994.
2. P. Alencar, L. M. F. Carneiro-Coffin, D. D. Cowan, and C. J. P. Lucena. Towards a Formal Theory of Abstract Data Views. Technical Report 94–18, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, April 1994.
3. D. Andrews and D. Ince. *Practical Formal Methods with VDM*. McGraw-Hill Software Engineering Series, 1991.
4. 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.
5. L. M. F. Carneiro, D. Cowan, and C. J. P. Lucena. ADVcharts: a Graphical Specification of Abstract Data Views. In A. Gawman, W. M. Gentleman, E. Kidd, P.-A. Larson, and J. Slonim, editors, *CASCON'93: Working Together*, pages 84–96, Oct. 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. W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. M. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 151–178. Springer Verlag, 1990.
8. D. Cowan, R. Ierusalimschy, C. Lucena, and T. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, Jan. 1993.
9. 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.
10. 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.
11. 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.
12. J. Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.
13. 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.
14. 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.
15. D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
16. D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.
17. 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.
18. R. Hartson. User-Interface Management Control and Communication. *IEEE Software*, pages 62–70, Jan. 1989.
19. 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.
20. R. D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI'92 Conference Proceedings*, May 1992.

21. C. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.
22. R. Ierusalimschy. A Method for Object-Oriented Specifications with VDM. Monografias em Ciência da Computação, PUC - RJ, Rio de Janeiro, RJ, Brazil, Feb. 1991.
23. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.
24. 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.
25. 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.
26. J. M. Manheimer, R. C. Burnett, and J. A. Wallers. A Case Study of User Interface Management System Development and Application. In *Proceeding of CHI'89*, pages 127–132. ACM Press, 1989.
27. L. S. Marshall. *A Formal Description Method for User Interfaces*. PhD thesis, University of Manchester, Department of Computer Science, Oct. 1986.
28. 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.
29. 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.
30. I. A. Newman. Personalised User Interfaces to Computer Systems. In *Proceedings of the European Computing Congress*, pages 473–486, London, 1978.
31. L. Nova, A. von Staa, D. D. Cowan, and C. J. P. Lucena. On the Automation of Code Generation for User-Interface Models. Technical Report 94–18, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, July 1994.
32. 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.
33. A. B. Potengy, C. J. P. Lucena, and D. D. Cowan. A Programming Approach for Parallel Rendering Applications. Technical Report 93–62, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 1993.
34. W. Reisig. *Petri Nets : an Introduction*. Springer-Verlag, 1985.
35. D. Smith. Abstract Data Views: A Case Study Evaluation. Technical Report 94–19, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, April 1994.
36. J. Uhl and H. A. Schmid. *A Systematic Catalogue of Reusable Abstract Data Types*. Springer-Verlag, 1990.
37. Watcom, Waterloo, Ontario, Canada. *WATCOM VX·REXX for OS/2 Programmer's Guide and Reference*, 1993.

This article was processed using the LaTeX macro package with LLNCS style