Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour

Fábio Nogueira de Lucena (fabio@dcc.unicamp.br)
Hans Liesenberg (hans@dcc.unicamp.br)

DCC/IMECC/UNICAMP

Relatório Técnico DCC–20/93

Setembro de 1993

Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour

Fábio Nogueira de Lucena (fabio@dcc.unicamp.br)
Hans Liesenberg (hans@dcc.unicamp.br)

DCC/IMECC/UNICAMP

Abstract

The process of software development of human-computer interfaces is complex and expensive. Many tools to automate the interface code generation have been proposed. The majority of these tools defines the behaviour of a system in terms of states and use conventional state transition diagrams (STDs) in order to describe dialogue control aspects. The statechart notation extends STDs to overcome some of their drawbacks. This paper shows the results gained from experiences of the use of statecharts in the development of user interfaces. These experiences led to the identification of improvements and additional constructs which are needed in order to make them more adequate for this specific usage in the even broader domain of distributed environments.

1 Introduction

The past decade has witnessed an intensive research effort to build human-computer interfaces (or *interfaces* for short). According to Myers [14], the code of an interface is huge, complex and its construction is not a trivial task. The goal of ongoing research efforts is to reduce development costs and to improve the quality of interfaces. The majority of these efforts seeks to construct software tools to improve development productivity.

In general, software tools are oriented to a particular development activity: design, implementation and evaluation. If these tools are integrated, then they are said to make up a *user interface management system*(UIMS), a *user interface development system*(UIDS) or even a *user interface development environment*(UIDE). The term "tools" is used in this text to refer to systems of this kind.

A common tool construction approach is to create techniques and/or high level languages to describe the tasks of interest supported by those tools. Many techniques for interface control specification and implementation deal with the syntax of possible user actions, the reactions of interactive systems, and dialogue evolutions. Myers compares the most usual in [13], like state transition diagrams (STDs). These techniques, however, present several shortcomings. This paper analyses the problems of representing and implementing dialogue control components. An STD extension, namely the statechart notation, has been used for this purpose.

Green presents three interface models in [5]: the transition net, the grammar and the event model. Although Green concludes that the event model has a greater expressiveness, each of those models has its own advantages and disadvantages [13]. The problems are in general due to a lack of mechanisms to represent particular features, usage difficulties and, in most cases, restrictions of representing a wide range of interface classes. Statecharts fit in the *transition net model.*

Within the transition net notation class, statecharts hold a comfortable position [1, 19]. This is mainly due to extended features with respect to STDs, which eliminate drawbacks of the latter, but retain their graphic nature [7, 8, 19]. However, there is no agreement at all about what a dialogue control specification language should be. Despite of favourable hints concerning statecharts, little is known about their usage in this particular niche. Wasserman [18] extends STDs in order to make them better suited to the specification process of interfaces based on his particular experience. In a similar way, some extensions of the

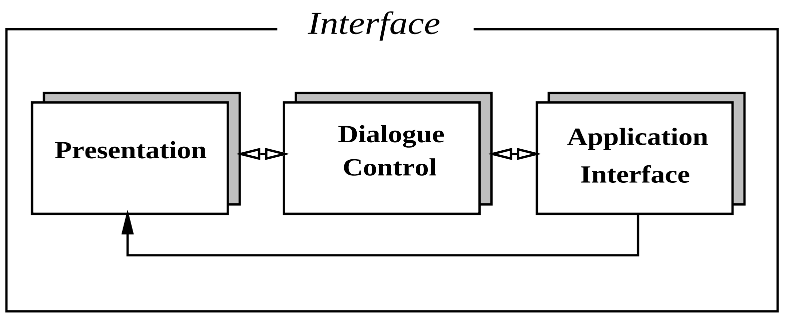


Figure 1: Seeheim Model.

statechart notation are proposed in this text.

Specialized versions of this notation are not a new fact. Johnson **et al.** [11], for instance, emphasize the importance of visual formalisms like the statechart notation and list some already existing versions of this notation. It is taken for granted in this text that the reader is familiar with the basic statechart concepts. A thorough description and others references can be found in [7] and [8].

The analysis presented in this paper is a result of building a "testing-bench" interface, which was developed in order to explore the expressiveness of statecharts. Its design was based on the Seeheim model [5]. The Seeheim model (figure 1) identifies three components which conceptually make up an interface. The *presentation* component is responsible for the input of user actions and the output of the system. The *dialogue control* as well as its description in terms of statecharts are the main issues of this paper. The related component co-ordinates user-computer interaction sequences and thus can be seen as the syntactic layer of this interaction. The *application interface* component corresponds to the semantic layer of the interface. It represents the view of the application by the interface and vice versa.

In the following section a short description of the developed interface is given. This interface has been used as a "testing bench" for the experiments with statecharts. In section 3, some results of the use of statecharts to develop this interface are described and some changes of the notation are suggested, which are followed by final remarks in section 4.

2 The "testing bench" interface

An interface for a system which supports music composition was developed in accordance to requirements established by another project. The main framework of this system are trees which hold structural information of a musical piece at intermediate nodes and scores at their leaves. Attributes defined at a given node apply to all of its subtrees. The main goal of the system is to aid the composition process, speed up experimentations as well as encourage the construction and trial of alternatives. A detailed description of the control aspects of the implemented interface and the corresponding application are not essential to the present discussion. Only strictly relevant information is presented.

The communication between the application and its interface is based on message exchanges and is carried out according to a protocol. This strategy leads to a weak binding of the application kernel with the interface module. The interface supports the coexistence of different interaction styles including direct manipulation. The implementation of the interface and the associated application were carried out independently, in order to achieve an independence of the dialogue from the application. The interface supports the edition of tree structures which can be carried out with the mouse. Different commands can be fired by manipulating icons directly or by selecting menu options. The interface furthermore provides a command language for expert users. The interface is sufficiently complex in order to enable a preliminary assessment of the statechart usage to describe interface behaviours.

The behaviour of the interface was described by a statechart which was converted into C code with the aid of a tool described in [3]. The resultant code consists of:

i. two tables representing the topology of the underlying statechart;

ii. invariant code, denominated "statechart engine," which keeps the current configuration of the underlying statechart and reacts to events according to the semantics of the statechart notation and to the particular statechart represented by the tables mentioned above;

iii. actions (functions in C code) executed due to transition firings and to the consequent deactivation/activation of states; as well as

iv. transition guarding conditions (functions in C code).

References to actions and transition guards are held by the two tables. The corresponding code is thus executed by indirect invocations by the "statechart engine." This framework made the change of event-driven control aspects relatively easy. Only changes of the table contents were required.

3 Statecharts in the interface context

This section discusses different issues related to statecharts and interfaces. The major goal is to point out shortcomings of this notation with respect to specification of interfaces. Some of the issues have been derived from facilities and constructs supported by tools which adopt different approaches. Many detected shortcomings are due primarily to the fact that statecharts have originally been proposed to describe the broad class of reactive systems with static architectures, in particular hardware systems. Interfaces represent a particular case of reactive systems, since they have their own peculiarities which are not always found in reactive systems of different kinds. As the statechart notation intends to cover the recurring concepts of reactive systems, it is understandable that special cases are not properly contemplated by the original notation. This section shows that the usage of statecharts in the software domain requires the addition of more flexible constructs in order to keep up with advances in this area.

3.1 The mode concept

A *mode* is a state or a collection of states wherein only a subset of all possible interaction tasks with the user can be carried out. Thus, in distinct modes different tasks can be performed. A modal behaviour furthermore enables the association of different meanings to a same input in accordance to the current mode. Two major issues related to the mode concept have been identified:

- **Are statecharts appropriate to describe dialogue control aspects?**

  According to Myers [13], one difficulty in specifying interfaces which adopt a direct manipulation style is the inexistence of modes, since any command can be given at almost any time. It is important to point out that a statechart models a particular behaviour in terms of a set of states (modes) and a set of transitions between states. Thus, how can this kind of interface be specified if the notion of modes does not exist? Fortunately Jacob [10] states that, in spite of appearances, the direct manipulation style is highly modal! He furthermore identifies a co-routine structure typical of interfaces which adopt this interaction style and shows how to identify states in this kind of interfaces. Lynette [17] and Wellner [19] make some comments about the successful use of statechart for this purpose.

- **What should a state represent in terms of a dialogue?**

  Another issue about the mode concept is related to the meaning of a *state modelled within a statechart.* The state **User Protocol** in figure 2 (i), for instance, has a substate named **Name Checker**. This substate models a process which verifies the existence of a registered user identifier. The event *Illegal Username* may be taken as the return of a function `NameChecker()` which receives a particular user identifier as its parameter. In this example, a state was used

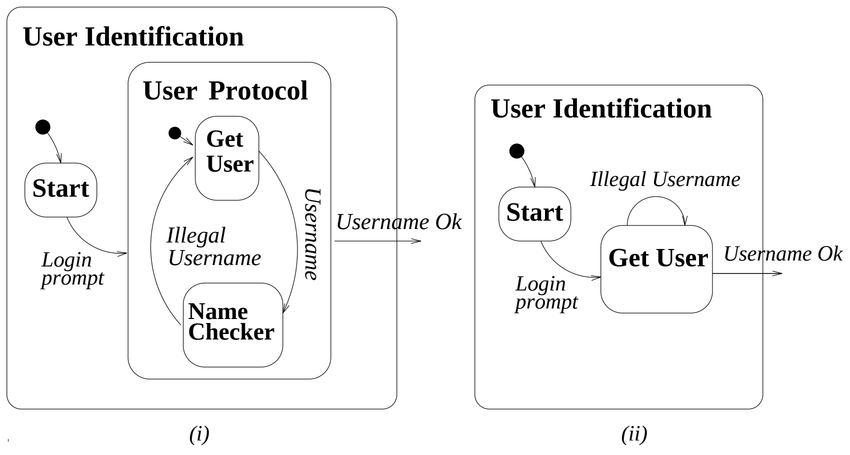


Figure 2: Mode interpretation

to model a process which is hidden from the implementor. Figure 2 (*ii*) represents an alternative modelling and is more meaningful as the former, since it reflects what an user perceives in terms of the interaction. The latter is consistent with the properties considered to be desirable of an interface specification language according to [9]. Both alternatives are possible, but only a more rigorous definition of how states should conceptually be interpreted may lead to more homogeneous specifications.

3.2 Event binding

The "test bench" interface was built on top of the Microsoft Windows System. Events of the keyboard and the mouse are handled by the kernel of this system and semantic actions are carried out by callback functions related to the target interaction objects of the events. This mechanism was used in order to establish the binding of the events generated by the keyboard/mouse ("physical" events) to abstract events ("logical" events) used within the underlying statechart specification. The code of a callback function consisted simply of a request to the "statechart engine" to handle a given logical event identifier.

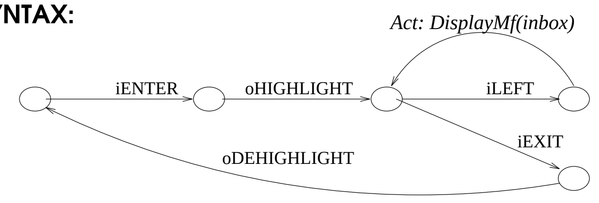
The adopted architecture established a master/slave relation between different components of the interface and it turned out that this solution imposed some limitations on the behaviour capable of being implemented. The implementation of the major components as concurrent processes interacting via message exchanges would be a more flexible approach, since the execution of the application could be carried out in parallel with the execution of the components of its interface.

Figure 3 shows an interaction object described in terms of the language defined by Jacob[10]. The block **TOKENS** in figure 3 indicates the association between logical and physical events. This kind of event binding is part of the role of the presentation component of the Seeheim model. Statechart constructs only support the specification of syntactical aspects of an interface. An evident shortcoming of this notation is due to the impossibility of expressing information to drive the event binding process. Lexical aspect descriptions can, however, coexist with syntactic definitions [16].

Another important point is related to the granularity of events. At the statechart abstraction an event may represent, for instance, a selection of option n of menu A. Events of this kind are of a higher abstraction level than normally supported by platforms like Windows. An abstract event, as illustrated above, has to be derived from a sequence of physical events handled by the underlying window system.

Statecharts could be used as well to recognize patterns in a stream of physical events in order to detect events of a higher abstraction level. The pattern recognizer and the interface control could be modelled as concurrent components of the interface descriptive statechart. Whenever the pattern recognizer detects the occurrence of a higher level event of interest to the interface control, the former would broadcast the event occurrence to the latter.

```
INTERACTION_OBJECT MessageFileDisplayButton IS
IVARS:
position    := { 100,200,64,24 }; --i.e., coordinates of screen rectangle
METHODS:
Draw()         { DrawTextButton(position,"Display"); }
TOKENS:
iLEFT               { click left mouse button }
iENTER              { locator moves inside rectangle given by position }
iEXIT               { locator moves outside rectangle given by position }
oHIGHLIGHT          { invert video of rectangle given by position }
oDEHIGHLIGHT        { same as oHIGHLIGHT }
SYNTAX:
```



```
end  INTERACTION_OBJECT;
```

Figure 3: Interaction object [10].

3.3 Reusable behaviours

The specification in terms of statecharts can be carried out in a bottom-up or top-down fashion, concurrently to the specification of other features of a given system (for instance, functional aspects) and by different persons. It would thus be desirable that the effort put into a specific statechart should not be exclusive to a particular specification. Because of the monolithic structure of statecharts, libraries of behaviours captured in terms of statecharts cannot be easily shared by specifications.

Conventional programming languages support the implementation of code fragments encapsulated in terms of procedures/functions which could be kept in libraries for a later use without needing to access their code, but solely their headers ("interfaces"). It would be interesting to admit, in a similar way, the reuse of a given behaviour as a "black box" without the need to know details of its specification, but only of the "interface" of the state at the root of its hierarchy. Reusable behaviours of interest to human-computer interface designers, for instance, are syntactical dialogue specifications related to common interaction objects. Specific examples are, for instance, dialogues to search for files, change parameters of a printer or select background colours. All of these behaviours have well defined goals and are applicable in many situations.

3.4 Dynamic behavioural changes

Interactive prototype development is a widely used interface construction technique [14]. Systems which support the construction of interfaces must give support for dynamic behavioural changes and must reflect the effects of these changes immediately during the animation of a statechart. The term *animation* intends to convey the meaning of simulating a specification or running an implementation with a structure closely related to the hierarchy of the specification. If statecharts are to be used to specify behaviour, the semantics of dynamic changes has to be established. For instance, what happens if an active state is removed as a statechart is being animated?

Prototype construction is not the only application of dynamic behavioural changes. Another application is the development of *event handlers*. Event handlers [5] are the basic elements of the event model and react to specific event occurrences. This kind of modules may be created and destroyed dynamically. Distinct instances of a given event handler may be active at a given instant. In order to capture this kind of behaviour in terms of statecharts, it is necessary to support the creation of multiple instances of a same state. A given state thus could be used as a template for this replication process. Appropriate constructs must be provided to allow replication and customization requests in terms of statechart actions.

For instance, any editor which supports the simultaneous edition of multiple files illustrates the need of this mechanism. In this context, it is more natural to think in terms of multiple instances of an editor statechart handling single files than in terms of a single editor capable of handling a limited number of files at the same time. For example, cut and paste operations on different files under edition could be defined as event broadcasts between different instances of a given state representing the behaviour of an editor capable of operating on a single file.

3.5 Undo operations

Interactive systems generally allow a user to undo some operations. This facility encourages the experimentation and use of a system. Thus, statecharts should have a construct to support the specification of this kind of behaviour. It seems reasonable that after an undo operation a statechart returns to its most recent configuration (set of active states at a given instant) prior to the firing of a transition the effect of which is to be reverted.

An undo operation in general requires that specific actions are to be carried out at the application level (semantics) as well as at the presentation level (lexical aspects). The "output" (response) of statecharts is always specified in terms of actions. Thus, they represent the only means of how statecharts are in contact with the outer world or of how they can exert influence upon it. As a consequence, it seems natural to relate the specification of undo actions to transitions and/or states. Thus, if an undo operation is required, then the corresponding actions are executed while the statechart stabilizes at the prior configuration. A stabilization of a statechart means the controlled blob[1] deactivation/activation process due to a configuration swapping.

In figure 4 the configuration of the statechart prior to the firing of the transition *tr* is highlighted in grey. The next configuration is shown in black. If at this point an undo operation is requested, then it returns to the configuration in grey. This configuration is the same as the one which would have been reached by the transition labelled *tr2* due to the history $H^\star$ condition with exception to history cancellations at lower levels and to effects of specific undo actions.

[1]Alternative term of state.

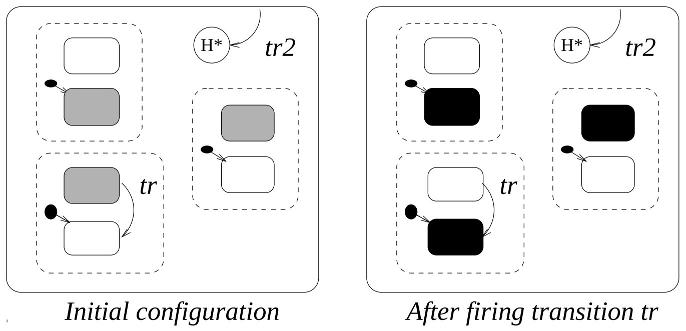


Figure 4: Possible semantics of the undo operation (undo(tr) $\approx$ tr2 + specific actions).

3.6 Event scopes related to transition firings

Figure 5 represents part of a statechart specification. If the configuration **A**, **B**, **C** and **D** is assumed at a given instant and if the user requests a help information at this point (i.e., the user generates the event *F1* possibly by pressing the key F1), then a transition of state **D** to the state **Help** occurs. The corresponding actions would generally be implemented in terms of a dialogue box which would exhibit relevant information. In order to carry on with the execution of the system (i.e., return to the prior state and remove the dialogue box from the screen) it is necessary that the user leaves the state **Help**. In the example, this is accomplished by generating the event *ESC*. However, another transition labelled *ESC* exists as well, from the state **B** to the state **End**. These two transitions conflict, since one or the other should be fired at the occurrence of the event *ESC*, depending on the context, but not both or one chosen at random. Both transitions, however, should be kept due to the consistency principle. The design of interfaces does not yet have any set of "golden rules," but some principles [15] are generally accepted like the consistency principle which tries to handle related activities in a homogeneous manner.

In the particular case above, the end of the help operation as well as

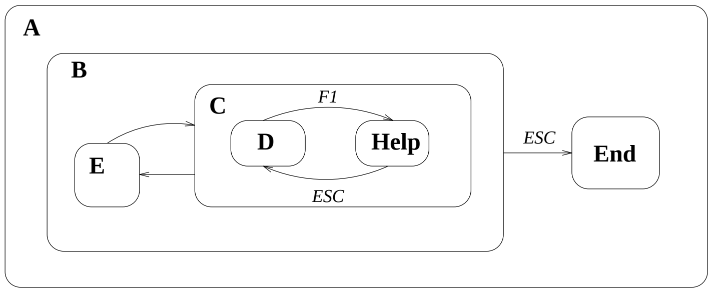


Figure 5: Event scopes

the exit of the system are signalled by the same event *ESC*. The exit of any dialogue box, as well as of menus and others interaction objects, is expected to occur in the same way.

A scope can be defined for the handling of events in order to eliminate ambiguities. Whenever a conflicting situation occurs, it could be established that the enabled transition with its origin at the lowest hierarchical level would be fired. In the particular case illustrated above, this rule determines that it is the transition which originates at the state **Help** which should be fired and not the one which originates at state **B**.

This strategy can as well be useful in other situations. Another example is the definition of states whose sole purpose would be to disable events in specific contexts.

3.7 The "complementary" and the pseudo-transition

The need to specify all possible interactions in terms of transitions represents an inconvenience due to the syntactic rules of the statechart notation. One way of reducing this problem is to permit the use of transitions which are fired whenever a given event does not affect the current configuration. This kind of transition is referred to as a "complementary" transition. Figure 6 illustrates a "complementary" transition shown as a dashed arrow. Whenever the state **A** is active and the current event does not fire any transition, then the dashed transition is fired and it

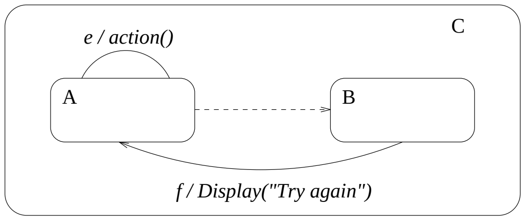


Figure 6: “Complementary” and pseudo-transitions.

causes the deactivation of the state **A** and the activation of state **B**.

There are many situations where no context swapping is desired (i.e., deactivations followed by activations due to a transition fired by a particular event), but where a particular event is expected to be handled since the side effects of the associated action are of interest. For instance, an operation “redraw” does not, in general, cause any change in the current configuration of a system. Only a given drawing is expected to be redone. In the original statechart notation, no means are provided to capture this kind of behaviour unless a transition is fired. In figure 6, the unoriented edge which connects state **A** to itself allows to reflect such a situation. In this particular case, whenever state **A** is active and the event e occurs, then the action `action()` is executed without generating a context swapping, i.e., without causing a deactivation followed by an activation of state **A**.

3.8 The in-breadth history mechanism

History mechanisms provided by statecharts are one of the major improvements of statecharts upon STDs. Very sophisticated behaviours can be captured in a clear and concise manner by means of history mechanisms. Two mechanisms are provided by statecharts referred to as *flat* and *in-depth* histories which are of interest during a blob activation process. A flat history applies only to the immediately lower level while a in-depth history applies to all lower levels unless overridden by new history attributes or cancelled at lower levels. If a history is being enforced, the most recently visited descendant is reactivated. If none of the descendants has yet been activated, then the default descendant is submitted to the activation process.

During the development of an interface, an alternative history mechanism was suggested and named *in-breadth* history. Whenever an exclusive blob (i.e., an OR-blob) decomposed in terms of subblobs is active and an in-breadth history is being enforced, then the exit of a given subblob and the return to the most recently visited "sibling" of this subblob occurs. A typical case where this history alternative is useful is the navigation through menus.

Whenever a blob with this kind of history attribute is activated, then the following happens from the conceptual point of view: an empty stack of limited capacity to hold blob identifiers is created. While this blob is active, all identifiers of its visited direct descendants, except for those activated due to the in-breath history, are pushed onto the stack. An identifier of a direct subblob is added to the stack whenever it is deactivated and a sibling is activated. If the stack is full and a new identifier is added, then the identifier at the bottom of the stack overflows and it is discarded in order to give way to the just added identifier. The stack is destroyed whenever the related blob is deactivated.

The following takes place whenever a transition is fired which has the in-breath history symbol as its destination:

i. The current blob and all its active descendants are deactivated, the identifier at the top of the stack is removed and the blob represented by this identifier is activated.

ii. The prior step can be carried out as often as desired by firing repeatedly the transition in question. The system represented by the statechart recedes the navigation route within the in-breadth history record hold by the stack. If the stack becomes empty, the system remains in the current blob, i.e., no transition takes place.

3.9 Intrinsic $\times$ attributed semantics

The statechart semantics defined by Harel is intrinsic, i.e., it is the same for all statecharts. The behaviour of a statechart is non-deterministic as formally described in [8]. A more flexible approach would be to consider the operational semantics of the statechart notation as an attribute of a given statechart. This would mean that a syntactically correct statechart could be interpreted in different ways according to the value of its semantic attribute. This more flexible approach would allow a more finely behaviour tuning of statechart specifications. Non-determinism makes the implementation on single-processor platforms harder and less efficient. From this point of view, different dynamic statechart models would be interesting. These models, however, have to be well defined since they represent abstractions statechart specifications are built upon. The major disadvantage of this approach is that unexpected effects may be obtained if the semantics attribute of a specification is changed without a prior careful thought. On the other hand, the designer of a system can take full advantage of some peculiarities of the adopted dynamic model and thus gain performance increases of the resulting system.

3.10 Describing distributed applications

As stated by the quotation bellow of Morse and Reynolds[12], the recent technology evolution requires new solutions to handle increasing complexity of the interface construction process:

> *During the eight years since the introduction of the Seeheim model the underlying technology has evolved into distributed, heterogeneous environments, which must be managed by the interface, and the requirements imposed by the application domains have become far more complicated. ... This model has to some extent been realized, but real progress has been achieved only in the area of the presentation component.*

The statement above presents a new challenge and it leads to the following question: **How can the statechart notation be extended in order to describe more explicitly peculiarities of systems**

envisaged by Morse and Reynolds? The work of the authors of the present paper is directed towards this matter. The extended notation proposal is denominated *Xchart. The major goal is to keep the visual nature of the notation and to introduce as few as possible new constructs in order to describe the desired features properly.*

The Xchart notation has to be based on an abstract model of distributed,[2] possibly heterogeneous, environments. The distribution units and the communication model have to be defined precisely in order to present to the designer an abstraction layer on top of which Xchart specifications can be built. This abstraction layer has to be implemented in terms of run-time systems to be executed at each site of a distributed environment capable of animating Xchart applications.

An Xchart consists of a hierarchical blob composition capable of being represented by a static diagram and subject to dynamic structural changes, as well as of transitions fired by events. Blobs are subdivided into two categories: exclusive and concurrent blobs. In the first case only one directly descendant blob of an active blob is active, while in the second case all direct descendants are active during the whole activation interval of their ancestor. A particular blob may be declared to be capable of being distributed. In this case it might be animated at a different site of the site of its direct ancestor.

Xchart is a visual language to describe control aspects of event-driven distributed systems. Semantic actions are provided by the programmer in terms of C++ code fragments. An Xchart specification plus the related code fragments are capable of being transformed automatically into functionally equivalent code. *Weasel*[4] is a system that provides a language and the corresponding support to the creation of applications in which multiple users perform a related task in a distributed context. In this system, e.g., all issues of network communication and concurrency are handled automatically based on specifications in an appropriate language. *Weasel* however adopts a client/server architecture,

[2]Distributed systems are understood as a system consisting of multiple autonomous processors that do not share primary memory but cooperate by sending messages over a communication network.

instead of a peer-to-peer architecture where all participants are of a same "rank." The latter represents an evolutionary step with respect to the client/server framework, which in turn is an evolution of the centralized computing paradigm[6].

It is important to note that in [2], among various problems that have inhibited the widespread use of a distributed application to support cooperative work, the problem of heterogeneous environments is mentioned. It would be unwise to expect a distributed application to run in all potential environments, but the definition of a language encompassing the "major" environments to manage the communication between distributed processes is a feasible goal. The Xchart notation is being devised to support the implementation of distributed applications among Macintoshes, personal computers running MS-Windows and UNIX with X11.

Since blobs capable of being distributed may be animated at different sites, each site represents a particular event domain wherein blobs might be inserted. Events sensed at one site are not detected by other sites unless explicitly notified via message exchanges over the communication network. An event occurrence might thus be perceived at a given site as a message reception. By this means an event of one site might cause a configuration swapping of a blob at a different site. Blobs animated at the same site sense the same events and thus no message exchanges between those blobs have to take place. The programmer does not have to take care of this kind of details. Appropriate code shall be generated by a transformation tool. From this point of view Xcharts represent a smooth extension of Statecharts.

4 Concluding remarks

This paper analysed the suitability of statecharts to capture dialogue control aspects. Specific situations were described where the expressiveness of existing constructs fell short. As a result of this analysis, a set of extensions of the statechart notation to support interactive human-computer interface development is suggested. The intended extensions require very few changes at the syntactic level and thus do not affect the graphic nature and the simplicity of the original notation.

Present work involves the formal description of the suggested extensions which shall drive the development of tools to aid the construction of interfaces with a sophisticated behaviour as well as the construction of an abstraction layer. This abstraction layer implements the distributed system model which provides the required services to the animation of the extended statecharts.

References

[1] Len Bass and Joëlle Coutaz. *Developing Software for the User Interface*. SEI Series in Software Engineering. Addison-Wesley Publishing Company, Inc., 1991.

[2] Nathaniel S. Borenstein. Computational Mail as Network Infrastructure for Computer-Supported Cooperative Work. *CSCW 92 Proceedings*, pages 67–74, November 1992.

[3] Antonio Gonçalves Figueiredo Filho and Hans Liesenberg. Transforming Statecharts into Reactive Systems. In *XIX Conferência Latinoamericana de Informática*, volume 1, pages 501–509, Buenos Aires, AR, August 1993.

[4] T. C. Nicholas Graham and Tores Urnes. Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications. *CSCW 92 Proceedings*, pages 59–66, November 1992.

[5] Mark Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[6] Michael Guttman, James A. King, and Jason Matthews. A Methodology for Developing Distributed Applications. *Object-Magazine*, pages 55–59, January 1993.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[9] Robert J. K. Jacob. Using Formal Specifications in the Design of a Human-Computer Interface. *Communications of the ACM*, 26(4):259–264, April 1983.

[10] Robert J. K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[11] Jeff A. Johnson, Bonnie A. Nardi, Craig L. Zarmer, and James R. Miller. Ace: Building Interactive Graphical Applications. *Communications of the ACM*, 36(4):41–55, April 1993.

[12] Alan Morse and George Reynolds. Overcome Current Growth Limits in UI Development. *Communications of the ACM*, 36(4):73–81, April 1993.

[13] Brad A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software*, pages 15–23, January 1989.

[14] Brad A. Myers and Mary Beth Rosson. Survey on User Interface Programming. In *Human Factors in Computing Systems CHI'92 Conference Proceedings*, pages 195–202, Monterey, California, May 1992.

[15] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Company, Inc., 1987.

[16] Gurminder Singh. *Automating the Lexical and Syntactic Desigh of Graphical User Interfaces.* PhD thesis, University of Alberta, Edmonton, Alberta, 1989.

[17] Lynette van Zijl and Deon Mitton. Using Statecharts to Design and Specify a Direct-Manipulation User Interface. *Proceedings of the Southern African Computer Symposium*, pages 51–68, 1991.

[18] Anthony I. Wasserman. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, SE-11(8):699–713, August 1985.

[19] Pierre D. Wellner. Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *Human Factors in Computing Systems, Proceedings SIGCHI'89*, pages 177–182, Austin,TX, April 1989.

Relatórios Técnicos – 1992

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

08/92 **Maintaining Integrity Constraints across Versions in a Database,** *C. B. Medeiros, G. Jomier, W. Cellary*

09/92 **On Clique-Complete Graphs,** *C. L. Lucchesi, C. P. Mello, J. L. Szwarcfiter*

10/92 **Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms,** *T. Kowaltowski*

11/92 **Debugging Aids for Statechart-Based Systems,** *V. G. S. Elias, H. Liesenberg*

12/92 **Browsing and Querying in Object-Oriented Databases,** *J. L. de Oliveira, R. de O. Anido*

Relatórios Técnicos – 1993

01/93 **Transforming Statecharts into Reactive Systems,** *Antonio G. Figueiredo Filho, Hans K. E. Liesenberg*

02/93 **The Hierarchical Ring Protocol: An Efficient Scheme for Reading Replicated Data,** *Nabor das C. Mendonça, Ricardo de O. Anido*

03/93 **Matching Algorithms for Bipartite Graphs,** *Herbert A. Baier Saip, Cláudio L. Lucchesi*

04/93 **A lexBFS Algorithm for Proper Interval Graph Recognition,** *Celina M. H. de Figueiredo, João Meidanis, Célia P. de Mello*

05/93 **Sistema Gerenciador de Processamento Cooperativo,** *Ivonne. M. Carrazana, Nelson. C. Machado, Célio. C. Guimarães*

06/93 **Implementação de um Banco de Dados Relacional Dotado de uma Interface Cooperativa,** *Nascif A. Abousalh Neto, Ariadne M. B. R. Carvalho*

07/93 **Estadogramas no Desenvolvimento de Interfaces,** *Fábio N. de Lucena, Hans K. E. Liesenberg*

08/93 **Introspection and Projection in Reasoning about Other Agents,** *Jacques Wainer*

09/93 **Codificação de Seqüências de Imagens com Quantização Vetorial,** *Carlos Antonio Reinaldo Costa, Paulo Lício de Geus*

10/93 **Minimização do Consumo de Energia em um Sistema para Aquisição de Dados Controlado por Microcomputador,** *Paulo Cesar Centoducatte, Nelson Castro Machado*

11/93 **An Implementation Structure for RM-OSI/ISO Transaction Processing Application Contexts,** *Flávio Morais de Assis Silva, Edmundo Roberto Mauro Madeira*

12/93 **Boole's conditions of possible experience and reasoning under uncertainty,** *Pierre Hansen, Brigitte Jaumard, Marcus Poggi de Aragão*

13/93 **Modelling Geographic Information Systems using an Object Oriented Framework,** *Fatima Pires, Claudia Bauzer Medeiros, Ardemiris Barros Silva*

14/93 **Managing Time in Object-Oriented Databases,** *Lincoln M. Oliveira, Claudia Bauzer Medeiros*

15/93 **Using Extended Hierarchical Quorum Consensus to Control Replicated Data: from Traditional Voting to Logical Structures,** *Nabor das Chagas Mendonça, Ricardo de Oliveira Anido*

16/93 $\mathcal{LL}$ – **An Object Oriented Library Language Reference Manual,** *Tomasz Kowaltowski, Evandro Bacarin*

17/93 **Metodologias para Conversão de Esquemas em Sistemas de Bancos de Dados Heterogêneos,** *Ronaldo Lopes de Oliveira, Geovane Cayres Magalhães*

18/93 **Rule Application in GIS – a Case Study,** *Claudia Bauzer Medeiros, Geovane Cayres Magalhães*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`