O conteúdo do presente relatório é de única responsabilidade do(s) autor(es).
(The contents of this report are the sole responsibility of the author(s).)

Programming Dialogue Control of User Interfaces Using Statecharts

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

Relatório Técnico DCC–28/93

Outubro de 1993

Programming Dialogue Control of User Interfaces Using Statecharts

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

Abstract

This paper presents a technique to implement control of complex user interface dialogues. It is based on event-driven control flow specifications described by a deterministic statechart dialect. With the aid of this technique the dialogue control is expressed in terms of a statechart. This statechart is then converted into tables handled by a run-time commented in greater detail. The execution of the run-time driven by those tables is equivalent to the behaviour specified by the underlying statechart. The resulting system calls specific application functions in response to user interactions mapped to events by the presentation component. The application is seen as a set of subroutines which can be invoked during the interaction with the user. The use of the proposed technique frees the programmer from the implementation of complex control aspects. The way of how to construct these statechart dependent tables, their use by the run-time, and the way semantic actions are attached are illustrated by a small example of a reative system which highlights mainly dialogue control aspects.

1 Introduction

The behaviour of interactive programs is often event-driven and cannot be conveniently described in terms of a function which maps input into output data. The behaviour is better captured by an internal state which

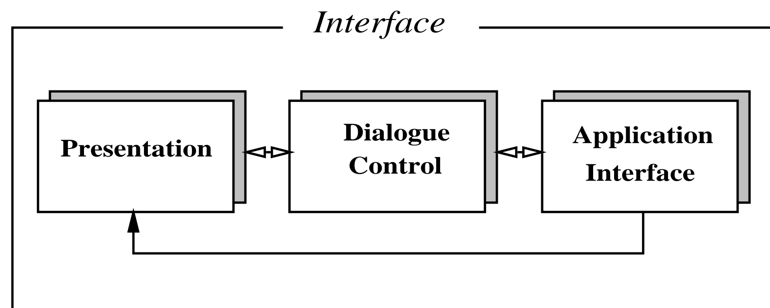

Figure 1: Seeheim model [Gre85].

is affected by asynchronous events.[1] Interactive programs have to fire appropriate actions according to generated events. Under some circumstances there might be many event sources and no specific sequence of event occurrences may be assumed. A same event may affect the state of a program in distinct ways or even have no effect at all depending on the context in which it is eventually handled. From this perspective, the specification of programs of this kind is difficult and the implementation is error-prone. Highly interactive systems based on a direct manipulation interaction style associated with multi-thread dialogues are a good example of hardly tractable specification problems.[2]

Different models and notations have been used to describe the interaction between an user and a computer [Mye89]. State transition diagrams and one of their extensions, namely statecharts, shall be discussed next.

The proposal of this paper is based on the Seeheim's logical user interface model depicted in Figure 1. The Seeheim model divides a user interface into three layers. The presentation layer is responsible for the external presentation of the user interface. This layer defines how

[1] Events are commonly generated by user actions upon input devices like a mouse or a keyboard.

[2] Despite of the apparent chaotic picture, this style of interaction is highly modal [Jac86] and suitable to be described by means of state diagrams.

an interactive system appears and is felt by the end user. It performs low-level input/output processing (lexical aspects). The dialogue control layer receives tokens from surrouding layers and determines how the conversation evolves based on those tokens. This layer defines as well which sequences of such tokens are to be considered or not (syntactic aspects). The dialogue layer has to keep the current state of the user interface and has to have control over this state since the dialogue evolution depends on it. It accepts inputs from the lexical layer and assembles them into commands and data on the one hand, and receives tokens from application requiring data and supplying responses to user requests on the other hand. The last layer represents the functionality (semantics) of an interactive application.

In this paper we propose a technique to derive a control structure underlying a dialogue layer which is described in terms of a statechart. State transition diagrams are very often used to represent this component. These diagrams are commented in more detail later on and their drawbacks in relation to statecharts are pointed out. These shortcomings let the application of statecharts become very interesting in this context. The technique consists of an invariant control run-time which implements the behaviour of a statechart dialect described in Section 4 and of the transformation of statecharts specifications into comprehensible information to the run-time described in Section 5. The implementation of the run-time is discussed in more detail and its code supplied in order to enable readers to implement their own applications. In Section 3 a little more is said about dialogue specification. In the following section a small illustrative example is presented informally. This example is used throughout this paper.

2 The Stopwatch Example

Figure 2 shows the statechart specification of a hypothetical stopwatch. The statechart **Stopwatch** corresponds to the operation of this toy stopwatch example whose display is either switched on (**Alive**) or off (**Dead**). **Alive** is the default substate of **Stopwatch** indicated by a special arrow,

i.e, whenever **Stopwatch** is activated **Alive** becomes activated as well.

The default substate of **Alive** is the state **Reset**. When this state is reached the stopwatch counter value becomes zero and the display blinks for three seconds, for instance. The semantic actions related to states and transitions are particular of the application being modelled. If the event *e4* takes place, then the state is left and revisited and the same related semantic actions are performed once more. If the event *e2* occurs while **Reset** is active, a transition to the state **Operation** is fired.

The activation of the state **Operation** implies the activation of both **Timer** and **Display**, since they represent a concurrent decomposition of **Operation**. If **Operation** is activated for the first time, then the default substates of **Timer** and **Display** become activated. On the contrary the most recently substates are reactivated because of the in-depth history attribute of the state **Alive**.

If the state **On** becomes activated the timer starts to tick away from the current value of the stopwatch counter. This process is interrupted if event *e2* takes place and fires the transition from **On** to **Off**. A subsequent occurrence of *e2* causes the return to state **On**.

The substates of **Display** represent the presentation mode of the stopwatch counter. If **Normal** is active, the value of the stopwatch counter is displayed in terms of minutes and seconds. If, on the other hand, **InSec** is active the display shows the time interval represented by the stopwatch counter solely in seconds. In two cases the **Alive** and all of its active substates are deactivated and **Dead** becomes active: if event *e5* occurs independently of the configuration of the substates of **Alive** or if the state **Off** is active and the event *e4* occurs. In former case, if **On** is active just before the event firing takes place, then the counter keeps ticking away and the counter cannot be stopped while the state **Dead** is active.

The state **Dead** is left under four circumstances: if either the event *e1*, *e2*, *e5* or the event *Exit* occurs. In the first case **Alive** is activated since it represents the default substate of **Stopwatch**. In the second case it becomes active because it is the ancestor of the destination state of the fired transition. In the third case **Dead** is deactivated and reactivated

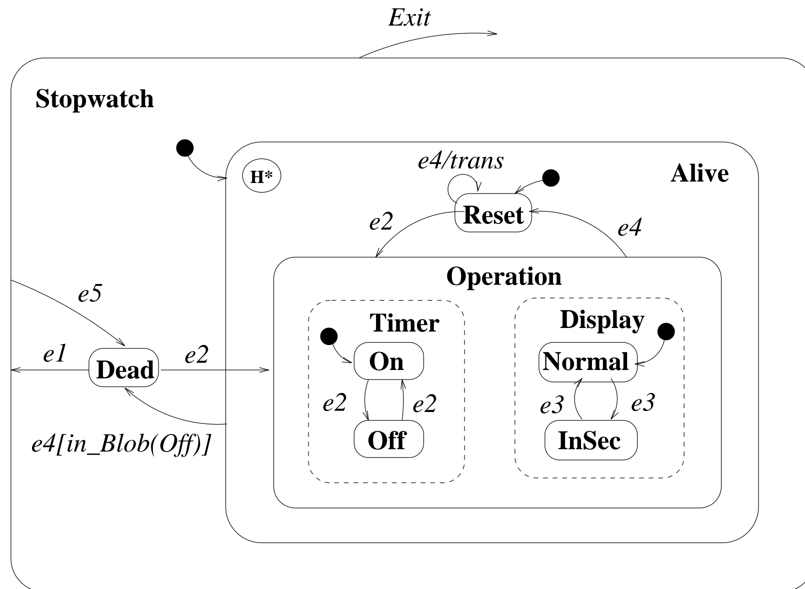

Figure 2: A simple example

and in the last case the animation of **Stopwatch** ceases. Whenever **Alive** or **Operation** is revisited the activation of their substates complies the history enforcements. The **Stopwatch** ceases all its activities, in whatever configuration it might be, if the event *Exit* takes place. In this case a controlled deactivation is performed and the system comes to a halt.

3 Dialogue Specification

Two alternatives of how to specify and to structure a dialogue shall now be presented. The first one is based on conventional state transition diagrams. These diagrams are widely used [Was85, Jac86, HH89] despite of their drawbacks pointed out in [Har87]. The second one is based on the statechart notation where the control process is more sophisticated, but it avoids some problems of the former. Both specification alternatives are graphical. This feature represents an advantage over textual descriptions employed by other specification languages. Further information about techniques employed to describe dialogues can be found in [Mye89]. the transition network model class according to [Gre86]. event model is used in textual languages like ALGAE [FB87]

3.1 State Transition Diagrams

One way of representing an event-driven control process is based on state transition diagrams. Specifically in the context of MS-Windows programming, Bertrand and Welch [BW91] have adopted a similar approach referred to as state tables.

This kind of diagrams consists of nodes, which represent particular states of a system, and of directed labelled edges between pairs of nodes which indicate transitions between states fired by the occurrence of an event enrolled in the corresponding transition label. During the execution of a state transition diagram, one particular state is considered to be the current state (at the beginning of an activation process the initial state becomes the current state) and, at each new event occurrence, the

labels of its outgoing transition edges are swept across in an attempt to identify a transition sensible to the current event. If such an edge is identified, then the corresponding transition is fired. The simplicity of the control process is however outweighed by a number of well known disadvantages. These can be inherent [Har87] or specific to dialogue control specifications [BC91, Mye89].

The framework of state transition diagrams is essentially "flat." The lack of means to incrementally define contexts hampers a better structuring of a system specification, i.e., no hierarchical framework is provided. These flat diagrams present another nasty feature. If a new diagram is to be composed of two already existing diagrams, then the states of the new diagram are represented by the cartesian product of the state sets of the two underlying state transition diagrams.[3] This explosive nature can be a great handicap when non-trivial systems have to be specified since those diagrams become very complex and their comprehension difficult. Multi-thread dialogue interfaces belong to this class of systems. Another disadvantage is the lack to explicitly represent concurrency since exactly one state represents the current state of the system. Concurrency has to be hidden and cannot be shown explicitly by the event-driven control flow represented by a state transition diagram.

Some of the specific shortcomings of state transition diagrams to describe dialogue control aspects are as well present in statecharts (e.g., all states must have explicit transitions for all possible erroneous input and commands, if they are to be handled properly). A more heavy use of statecharts in the human-computer interface context, however, is not well documented in the academic literature.

3.2 Statecharts

Harel [Har87] extended state transition diagrams eliminating some shortcomings and named them statecharts. The statechart notation produces more concise specifications. It is also richer in terms of expressiveness

[3] Thus, if one diagram has m and the other n states the resulting diagram is composed of $m \times n$ states!

and supports incremental context definitions related to hierarchical decompositions, control flow based on history and explicit concurrency specifications. The brief description below is not meant to be a tutorial.

Statecharts have been proposed originally as a specification formalism for hardware devices [DH89]. At a later stage the notation has been used to specify software systems [HLN+90] or even user interfaces [Wel89, vZM91]. A subset of the statechart notation and not necessarily the same terminology proposed by Harel has been adopted in this paper.

Statecharts are described in terms of nested contexts represented graphically by non-overlapping rectangles with rounded corners called blobs or states. The nesting reflects successive decompositions. Blobs can be of two kinds: mutually exclusive and concurrent blobs. The former is indicated by a solid and the latter by a dashed contour.[4] Figure 2 depicts one example. If a context is decomposed in terms of mutually exclusive blobs (as state **Stopwatch** in Figure 2), one of those (**Dead**, or **Alive** in the particular case) must be active at a given instance whenever their direct ancestor is active. On the other hand, if a context is decomposed into concurrent blobs, whenever this context becomes active, all of its concurrent blobs become active as well. For example, whenever the state **Operation** is active, all of its direct descendants (**Timer** and **Display**) are active too.

If a given blob is decomposed into mutually exclusive blobs, then one of the subordinated blobs has to be declared as its default descendant which is indicated by a special transition edge. It is the case of the substate **Alive** of the state **Stopwatch** in Figure 2. This default state becomes active whenever its direct ancestor (**Stopwatch**) becomes active and if the activation of no other of its siblings is being enforced by the activation process due to a history condition. The set of active states of a statechart in a stable situation (i.e., at an instance where no transition takes place) is referred to as its configuration.

A transition is represented by a directed edge between two mutually exclusive blobs and, if fired, causes a context swapping. A transition

[4] The use of dashed contours to depict concurrent blobs does not exist in pure statecharts. It makes the blob identification labelling homogeneous.

edge is labelled with an event identifier and is possibly associated with a guarding condition. A transition is fired whenever the origin blob of the corresponding edge is active, the event referred by its label has occurred and its guarding condition (if it exists) is satisfied. For example, the label *e4[in_blob(Off)]* of the transition from **Alive** to **Dead** (Figure 2) will enable this transition whenever the state **Alive** is active, the event *e4* occurs, and if at this instant the guarding condition is (is state **Off** active?) is true.

It is necessary to provide mechanisms other than only control in order to make statecharts useful. We do not only expect from a statechart the parsing of valid event sequences. It is necessary to have a mechanism to produce some output. This mechanism is implemented in terms of function calls in the current proposal. Functions are attached to states and/or transition edges of a statechart. These functions are invoked each time a particular state is entered (activated), left (deactivated) or a specific edge is traversed.

A function associated to a transition is called after the deactivation of the origin blob and before the activation of destination blob. Each blob might as well be associated with two functions: one is executed whenever the state is activated and another when it is deactivated. These outputs are analogous with the outputs of Mealy and Moore machines of the finite automata theory. In Moore machines, however, output is produced only when one state is reached. A statechart output may also be produced upon leaving a state.

4 Run-time Implementation

This section presents more detailed information about a particular implementation of the complex statechart semantics and often refers to code of the listings at the end of this paper. A reference to specific lines of code is made as follows: [<file identifier>, <first line of relevant code>-<last line of relevant code>,...].

The transformation of the control underlying a statechart specification into an executable code can be eased if the control structure of the

resulting program is kept as close as possible to the control process which describes the behaviour of a statechart. This control structure is referred to as the statechart run-time control (or the run-time for short) or even as the statechart engine. The run-time has been implemented in ANSI C.

The run-time represents the invariant code of a system developed according to the proposed technique. It performs, in essence, event sensitivity checks and controls the state activation and deactivation process. In order to get down to details about its implementation, a data structure holding information about a particular statechart topology and its transition lattice shall be described next. The information held by these data structures must be supplied and must reflect the topology and the attributes of the system's statechart specification.

Tables are not the best way to hand over information to the run-time, because their construction process is error-prone. The run-time does not perform any automatic checking. The construction of those tables is better done by a compiler of statechart specifications into the required data structures. An proper environment of this kind is described in [FL93]. The way of how to construct the two tables manually according to a given statechart shall now be described.

It is important to point out that the code presented here implements only a subset of the statechart semantics originally defined by Harel. This subset is illustrated by means of an example given in Section 2. Anything else not mentioned has not been contemplated. The code of the run-time is listed as the contents of the files `engine.h` (Section 8.1) and `engine.c` (Section 8.2).

The data related to a statechart topology is kept in a tree structure which reflects the hierarchy defined by its topology. The correspondence between blobs and nodes of the tree is an one-to-one relation. Figure 3 shows one tree which is equivalent to the hierarchical structure of the statechart described in Figure 2. This tree is a binary tree by chance. No restriction is imposed on the arity of nodes of this kind of trees.

Nodes are identified internally by values in the range of 1 to n, where n represents the total number of blobs a particular statechart is com-

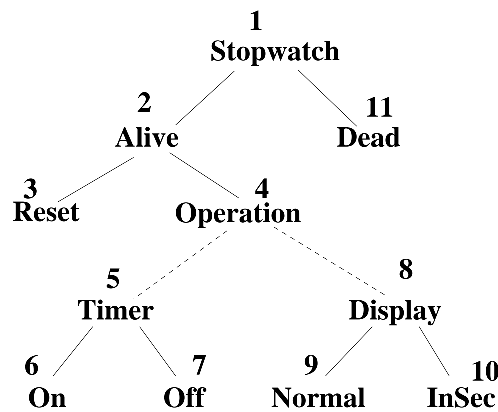

Figure 3: Internal numerical identifiers determined by `StartStEng()`

posed of. The internal blob identification corresponds to the node enumeration of the related tree in an inorder traversal. These values index an array whose elements keep information related to the corresponding blobs and relevant to the tree traversals demanded by the context swapping procedure due to transition firings. The designer is not obliged to relate blob identifiers to numerical values in accordance to the enumeration rule of internal identifiers. The numerical identifiers defined by the designer ("external" identifiers) are converted automatically into "internal" identifiers which adhere to the required ordering relation [`engine.c`, 007-011, 042-090, 117-123]. A closer look at the data structure holding statechart-dependent information shall be given next.

4.1 Data Structure for Statechart Topologies

The definition of an element of the array which describes the topology of a statechart specification is given in Figure 4. The content of this Figure is a transcription of [`engine.h`, 043-050]. It stores the following information for each state: its identifier (`state`), a reference to its default direct descendant (`primogen`), a reference to its next sibling (`sibling` – the sibling list is circular), a reference to its direct ancestor (`ancestor`),

```
typedef struct {
  numBlob state,           /* Numerical Id of the blob given by the designer  */
          primogen,        /* Direct descendant                                */
          sibling,         /* Next sibling in a circular list of siblings      */
          ancestor;        /* Direct ancestor                                  */
  bStatus status;          /* Concurrent/exclusive blob, history, etc.         */
  ptrFunc OnEntry, OnExit; /* Executed whenever a state is entered or left     */
} BNODE;
```

Figure 4: Data structure to hold information about a blob

status information (`status`) and two pointers to functions to be called whenever the corresponding blob is entered (`OnEntry`) or left (`OnExit`) respectively, i.e., whenever it is activated or deactivated. A similar field (`action`) exists in `BTRANS` (Figure 5) which specifies a function to be called whenever the corresponding transition is fired.

The `status` information consists of a sequence of bits which indicates the existence or not of a shallow history condition defined at the corresponding blob, an in-depth history, a history condition cancellation, and tells if the blob in question is a concurrent or a mutually exclusive blob. Their values correspond to those defined at [`engine.h`, 012-024].

4.2 Data Structure for Transitions

The array of type `BTRANS` holds information about the sensitivity of blobs to specific events (see Figure 5, which corresponds to [`engine.h`, 035-041]). One element of this array keeps for each transition the identifiers of the origin (`from`) and the destination blob (`to`), one pointer to a function which returns a boolean value indicating if the transition can take place (`cond`) and to a function which represents a semantic action (`action`).

One element of this type asserts that if state `from` is active and `event` occurs, then the function `cond` is called and its result tested. If it is true,

```
typedef struct {
  numBlob  from;        /* Origin blob                                      */
  numEvent event;       /* Event wich triggers the transition               */
  numBlob  to;          /* Destination blob                                 */
  int (*cond)(void);    /* Condition that guards the transition             */
  ptrFunc action;       /* Action executed whenever the transition takes place */
} BTRANS;
```

Figure 5: Data structure to hold information about a transition

the transition to state `to` takes place. After the deactivation process due to the transition in progress has been concluded and just before the start of the subsequent activation process, the function `action` is called [`engine.c`, 283-295].

Events are as well associated to numerical values, but their enumeration is not critical since it does not affect the operation of the run-time.

4.3 Run-time Interface

A statechart represented by the data structures described above can be exercised by means of four functions. Two of them initialize and deactivate the run-time:

- `int StartStEng(BNODE*,numBlob,BTRANS*,numEvent)`
  This function [`engine.c`, 108-127] receives the tables described above as well as their particular dimensions.

- `void StopStEng()`
  It [`engine.c`, 129-133] informs the run-time that its execution should cease.

The remaining functions are:

- `int stEng(numEvent)`
  It [`engine.c`, 354-366] signals an event occurrence.

- `int in_Blob(numBlob)`
  It [`engine.c`, 308-313] verifies if a given blob is active or not. It is usually invoked by transition guards.

4.4 Run-time Execution

`StartStEng()` [`engine.c`, 108-127] receives the address and the dimensions of the tables which describe the topology and transitions of a particular statechart specification and are used by the run-time in order to make it behave according to this specification. This framework makes it possible to change the behaviour without the need to recompile any code. Internal blob identifiers are generated and the tables are updated accordingly. In order to improve the look-up of the transition table, this table is sorted in increasing order of the internal origin blob identifier [`engine.c`, 095-105].

The array structure enables random access to the information related to a specific blob. The way how blobs are identified internally turn the search for a nearest common ancestor very simple. The ancestor links are followed from the destination or the origin blob, whichever is represented by the greatest internal identifier, until a blob is reached with an internal identifier less than or equal to the identifier of the other blob of the pair of blobs passed as parameters [`engine.c`, 149-163].

The control structure, which guides context swappings, is independent of the topology of a specific statechart and is referred to as the statechart run-time control, or simply the run-time, as stated before. The main function (`stEng(event)` [`engine.c`, 354-366]) describing the functionality of the run-time receives as its argument an event identifier. This function traverses the list that holds the identifiers of active blobs (this list is referred to as the configuration or as the global state of a statechart and is kept by the `status` field of the elements of the `bInfo[]` array) and verifies the sensitivity of those blobs to the current event.

A context swapping due to a transition firing is carried out in two steps. At first all blobs from the atomic blob reachable from the origin

blob, up to the nearest common ancestor of the origin and the destination blob (excluding the latter), have to be deactivated as well as all concurrent siblings and their descendants along this path. Next the blobs of the path between the destination blob and the nearest common ancestor mentioned above (excluding the latter) are activated in reverse order. From this point the activation process is kept up until an atomic blob is eventually reached. All concurrent siblings of activated blobs at this second stage of the context swapping process are activated as well.

The transition firing function (`fromBlobToBlob()` [`engine.c`, 282-294]) identifies an active atomic blob reachable from the origin blob (`activeAtomFrom()` [`engine.c`, 138-147]), it finds out the nearest common ancestor (`nearestCommonAncestor()` [`engine.c`, 149-163]) of the origin and the destination blob, it deactivates all blobs along the path from the active atomic blob found previously up to the nearest common ancestor (excluding the latter) as well as all concurrent components along this path [`engine.c`, 166-195]. It then calls the function representing the semantic action related to the transition (if specified), and starts the activation process (`activatePath()` [`engine.c`, 264-280]).

The first step of the activation process consists of a search of possible history enforcements at the nearest common ancestor and levels above and a demarcation of the path from the destination blob up to the nearest common ancestor (`vhDesc` of the `WrkMemory` is used for this purpose). This path is then followed from the nearest common ancestor down to the destination blob and all blobs along this path (except for the former) are activated. Concurrent siblings are activated according to the history mode being enforced. Once the destination blob is reached, the activation process is sustained, but now complying with history enforcements, until an atomic blob is eventually reached. Concurrent siblings are activated in the same manner at this second stage of the activation process.

Since no activation path is predetermined for the second part of the activation process or for concurrent siblings come across along this process, the activation in those cases is performed according to one of the following manners: if a history condition is being enforced, then the most

recently visited blob is reactivated; if no such blob exists or no history condition is being enforced, then the default direct descendant is activated. A history condition can be an in-depth (the history condition applies to all lower level contexts of the context where it has been defined unless overridden or cancelled at lower levels) or a shallow history (the history condition applies only to the next lower level context). At the start of the statechart engine, the non-predetermined-path activation process is applied to the outermost blob.

As a result of the deactivation process, the deactivated blobs are removed from the current configuration and, in consequence of the subsequent activation process, the just activated blobs are added at a second stage. In other words, the activated subtree of the nearest common ancestor is replaced by a new one.

The `active` variable is used by the function `_steng()` [`engine.c`, 321-352] to determine the next blob of the original configuration to be submitted to a sensitivity test. Once all blobs of a configuration have been swept across, the function in question can be called again to handle a new event, but now in the context of the resultant configuration from the handling of the prior event.

Since the order for the sensitivity tests is determined by the increasing order of the internal blob identifiers computed by the function `StartStEng()` [`engine.c`, 108-127], the firing of transitions is performed in a deterministic way. Thus, alternative orderings may produce distinct behaviours. It is important to point out that if a given event is found on the event list of two transitions which have as their origin a direct or indirect ancestor and its descendant respectively, then the event has no effect on the latter since the descendant becomes deactivated during the firing of the transition which has its origin at the former.

As already indicated above the algorithm to find the nearest common ancestor is as well heavily dependent on the correct enumeration of blobs. The concept of the nearest common ancestor is degenerated in two cases: if the destination blob is a direct or indirect descendant of the origin blob (e.g., the transition labelled *e5* in Figure 2) and the second by the reverse (e.g., the transition labelled *e1* from state **Dead** to **Stopwatch**).

If however the origin and the destination blob of a transition edge happen to be the same, then the direct ancestor is taken as their nearest common ancestor. This means that the blob in question is activated and immediately afterwards reactivated. This is the case of the transition labelled *e4* at **Reset**. The degenerated cases cause the exclusion of the indirect nearest common ancestor of the deactivation/activation process, i.e., it is kept active all along.

5 Implementing the Stopwatch Behaviour

This section describes the implementation of a small statechart specification with a deterministic behaviour presented in Section 2. The implementation is carried out in the Microsoft Windows environment, but this is not the sole environment which could have been used as the development platform. The main purpose is to show how a complex behaviour can be specified and implemented and not to show how realistic this example can be. It is better taken as a template of the development of complex behaviour, particularly of event-driven interactive applications.

The next two subsections define respectively how transitions and the topology of the given statechart specification (Figure 2) are represented in the tables required by the run-time. The simple example shown before is used for this purpose. The relevant information can be found in two small files: `st.c` (Section 8.4) and `st.h` (Section 8.3).

5.1 Specifying Transitions

Transitions are labelled and usually establish a relation between two blobs, with exception of default transitions. The label of a transition carries an event which might fire the transition and possibly a guarding condition and/or a reference to a function representing a semantic action to be carried out during the transition as described earlier on. The type `BTRANS` in [`engine.h`, 035-041] describes the relevant information.

The enumeration of events [`st.h`, 018-023] is irrelevant to the implemented algorithm of the run-time. An abstract event used within a

```
BTRANS bTrans[] =
{/* state, event,state, condition, action functions */
  { 0,          0,   0,             0, 0     },
  { Stopwatch, e5,   Dead,          0, 0     },
  { Reset,     e2,   Operation,     0, 0     },
  { Reset,     e4,   Reset,         0, trans },
  { Operation, e4,   Reset,         0, 0     },
  { Normal,    e3,   InSec,         0, 0     },
  { InSec,     e3,   Normal,        0, 0     },
  { Off,       e2,   On,            0, 0     },
  { On,        e2,   Off,           0, 0     },
  { Alive,     e4,   Dead,       cond, 0     },
  { Dead,      e1,   Stopwatch,     0, 0     },
  { Dead,      e2,   Operation,     0, 0     }
};
```

Figure 6: Transition description of the statechart in Figure 2

statechart might correspond to a single physical event or a sequence of physical events as in the case of a menu item selection. The event `e1`, for instance, could represent an `WM_LBUTTONDOWN` message sent by the Windows kernel to the application. The responsibility of binding abstract to physical events is of the presentation component discussed earlier. In order to simplify the given example a simple and direct binding is used where each event $\mathtt{e}_k$ is generated by pressing the key k.

The table `bTrans` in [`st.c`, 013-027] shown in Figure 6 describes the transitions of Figure 2. It is important to point out that default transitions are a special case and are described implicitly within the `bInfo` table which shall be commented next.

5.2 Specifying Topology

In this section the representation of a statechart topology is commented. In order to describe a particular topology: one entry for each blob in the table `bInfo` (see Figure 7 and [`st.c`, 029-043]) is used. For each state, its relationships with other states and action functions are listed. The hierarchy of the statechart in Figure 2 is shown in terms of a tree in Figure 3. In this particular case, the tree turned out to be a binary tree. Nodes of this kind of trees, however, can be of any arity, since siblings are kept in a circular list which is reached from the direct ancestor by its default descendant pointer.

For the state **Stopwatch**, for instance, its entry in the table identifies **Reset** as its default descendant. **Stopwatch** has no history attribute nor is it a concurrent component and thus its status is `_noHist`. Whenever **Stopwatch** is entered and left the function `Show()` is called. This function [`stopwatc.c`, 023-032] is only used to trace the blob activation/deactivation paths. For simplicity reasons, none of the specific functionality suggested in Section 2 has been implemented.

6 A Windows-based Application

The architecture of programs for MS-Windows differs from the one of traditional programs. It is briefly commented below since it is used in the implementation of the example and, in general, is similar to how programs are implemented on others Windows Systems.

A Windows program aggregates in general different independent overlapping display areas called windows: one of these represents the main window and each of the remaining windows has to be declared as a subordinated window (referred to as a child window) of some other one. One event handling function is always associated with each window. The Windows kernel captures events and sends them to a queue where from they are successively consumed and passed to the event handling function of the window associated with the particular event. Thus, one part of the presentation layer is encapsulated by the Windows kernel.

```
BNODE bInfo[] = {
/*state, primogen, sibling,ancestor,      status, entry,exit */
{ 0,         0,     0,         0,              0,  0,   0   },
{ Dead,      0,     Alive,     Stopwatch,_noHist,  Show,Show},
{ Stopwatch, Alive, Stopwatch, 0,        _noHist,  Show,Show},
{ Alive,     Reset, Dead,      Stopwatch,_starHist,Show,Show},
{ Reset,     0,     Operation, Alive,    _noHist,  Show,Show},
{ Operation, Timer, Reset,     Alive,    _noHist,  Show,Show},
{ Timer,     On,    Display,   Operation,_concurr, Show,Show},
{ Display,   Normal,Timer,     Operation,_concurr, Show,Show},
{ On,        0,     Off,       Timer,    _noHist,  Show,Show},
{ Normal,    0,     InSec,     Display,  _noHist,  Show,Show},
{ Off,       0,     On,        Timer,    _noHist,  Show,Show},
{ InSec,     0,     Normal,    Display,  _noHist,  Show,Show}
};
```

Figure 7: Topology description of the statechart in Figure 2

The attributes of a window and of visible symbols shown to the user within the bounds of that particular window are defined all over a Windows program. The code of a program based on Windows and a statechart specification consists of:

i. a function `WinMain()` which in general creates an instance of the main window, retrieves events from the event list of the Windows kernel and dispatches them to the appropriate event handling function of the target window;

ii. event handling functions associated to windows defined within the program. These functions are notified of event occurrences whenever the targets of those events are represented by the corresponding windows. They receive the control after the notification of an event occurrence, the event is converted to a logical event and the logical events is passed to the dialogue control layer. This layer, in response, triggers tasks related to changes of the internal state of the interface due to event occurrences. It includes the handling of presentation aspects and function invocations related to the functionality of the application; and finally

iii. functions which represent operations of the application proper.

Most of the facilities provided by the Windows environment are of a very low abstraction level and correspond basically to the facilities of a presentation layer. One extra layer is often provided by libraries with a procedure-oriented or object-oriented interface on top of the Windows API. These tools (generally referred to as toolboxes and toolkits, respectively) provide some facilities to support simple interaction operations (e.g., scrolling) and provide a higher degree of abstraction to the program developer. Little support is given to the designer to structure the dialogue evolution of a system. In this paper the presented technique is used for this purpose.

7 Concluding Remarks

Statecharts are an extension of conventional state transition diagrams. Due to the hierarchical nature of the specification notation, distinct contexts can be defined incrementally at different levels of a hierarchy. These contexts come into existence and are destroyed in a controlled manner by firing events. Designs based on this technique turn out to be better structured in general. The task of the designer becomes lighter since greater efforts can now be put on what has to be performed in specific contexts without having to pay greater attention on what is happening in terms of context swappings. This focus concentration makes the design task easier.

The way how a complex specification in terms of a statechart could be implemented using a run-time in the Microsoft Windows environment has been described. The present work reflects an evolution from a previous proposal [BW91]. There are some restrictions of the run-time which inhibit the implementation of the whole statechart functionality. Nevertheless the restricted behaviour implementation is believed to be useful to implement interface dialogues, and new version with additional funcitonality are planned.

Care must be taken since the tables for the run-time are built manually. A proper environment [FL93], however, may solve this difficulty.

It is easy to see, by means of the presented example, that the underlying control of a complex behaviour can be trivially implemented by adopting the proposed technique. Complexity does not simply disappear, but it is transferred to the run-time. Changes of behaviour require changes in only two tables and can even be made at run time. If the programmer makes use of the proposed technique, the existence of sophisticated control mechanisms can be taken for granted and only lexical and semantic aspects have to be taken into consideration. The programmer is released from the error-prone task of developing a complicated code segment which exist in all interactive system: the dialogue control component.

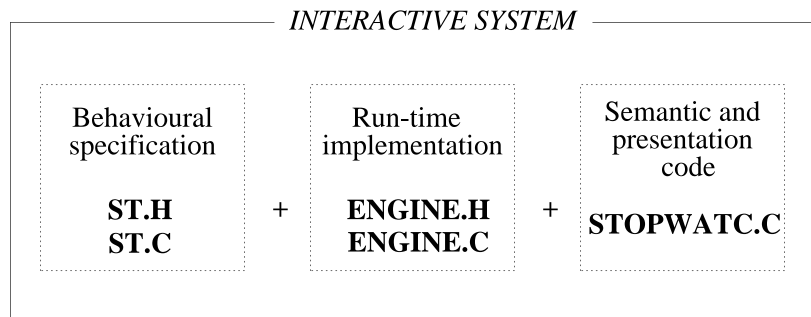

Figure 8: Relationship among files

8 Listings

Figure 8 illustrates the relations between the files which implement the adopted example. The first module stands for the tables holding the relevant data of a given statechart. The second module represents the invariant code (named run-time or statechart engine) which controls basically the transition firings and the state activation and deactivation process. The last module contains semantic actions (i.e., the application proper in terms of a set of functions) and some code responsible for the presentation of information to the end-user. The complete relevant listings follow.

8.1 engine.h — run-time header

```
001  /* engine.h -- The Statechart Engine (HEADER) */
002
003  #ifndef ENGINEH
004  #define ENGINEH
005
006  typedef unsigned short numBlob;   /* engine data types */
007  typedef unsigned short numEvent;
008  typedef char bStatus;
```

```
009  typedef char history;
010  typedef void (*ptrFunc)(int,int);
011
012  /* status information operations
013     bit 6: 0(blob)  1(concurrent component)
014     bit 5: activated
015     bit 4: mark blob visited
016     bit 2: cancel history enforcement
017     bit 1: star history (* history)
018     bit 0: standard history (h history) */
019
020  #define _noHist      0x00  /* masks */
021  #define _hHist       0x01
022  #define _starHist    0x02
023  #define _cancelHist  0x04
024  #define _concurr     0x08
025
026  #define _hist        0x07 /* masks of internal use */
027  #define _activ       0x20
028  #define _visited     0x10
029
030  #define ERR_NBLOBS          2 /* Error codes */
031  #define RECURSIONNOTALLOWED 3
032  #define FINISHED            4
033  #define OUTOFMEMORY         5
034
035  typedef struct {
036    numBlob  from;    /* Origin blob                                          */
037    numEvent event;   /* Event wich triggers the transition                   */
038    numBlob  to;      /* Destination blob                                     */
039    int (*cond)(void);/* Condition that guards the transition                 */
040    ptrFunc  action;  /* Action executed whenever transition takes place */
041  } BTRANS;
042
```

```
043 typedef struct {
044   numBlob state,    /* Numerical Id of the blob given by the designer */
045           primogen, /* Direct default descendant                      */
046           sibling,  /* Next sibling in a circular list of siblings    */
047           ancestor; /* Direct ancestor                                */
048   bStatus status;   /* Concurrent, history, etc.                      */
049   ptrFunc OnEntry, OnExit;/* Callbacks called when entry/left a state */
050 } BNODE;
051
052 /* Interface to the run-time */
053
054 extern int  stEng(numEvent);  /* Pass event to the run-time    */
055 extern int  in_Blob(numBlob); /* Return true if blob is active */
056
057 /* Activate and deactivate the statechart run-time */
058 extern int  StartStEng(BNODE*,numBlob,BTRANS*,numEvent);
059 extern void StopStEng(void);
060
061 #endif
062
```

8.2 engine.c — run-time implementation

```
001 /* engine.c -- The Statechart Engine */
002
003 #include <stdlib.h>
004 #include <string.h>
005 #include "engine.h"
006
007 typedef struct WorkMemory {
008   numBlob vhDesc;        /* It keeps track of paths in the tree           */
009   numBlob start, last;   /* Holds the range (btrans) relevant to a blob   */
010   numBlob blobId, backId;/* backId(k) = user id equivalent to run-time id */
011 } WrkMem;                /* k. blobId(k) = run-time id to k user id.      */
```

```
012
013  static WrkMem *ctlMem;
014  static numBlob active;
015  static numBlob nBlob;
016  static numEvent nTran;
017  static BNODE *bInfo;
018  static BTRANS *bTrans;
019  static numBlob k;       /* General purpose                              */
020  static char finished = 0;/* Mark for the end of execution of the run-time */
021
022  /* declarations to legibility */
023  #define _vhDesc(x)     ctlMem[(x)].vhDesc
024  #define _start(x)      ctlMem[(x)].start
025  #define _last(x)       ctlMem[(x)].last
026  #define _blobId(x)     ctlMem[(x)].blobId
027  #define _backId(x)     ctlMem[(x)].backId
028  #define _concurrent(x) (bInfo[(x)].status & _concurr)
029  #define _history(x)    (history)(bInfo[(x)].status & _hist)
030  #define _active(x)     (bInfo[(x)].status & _activ)
031  #define _activate(x)   (bInfo[(x)].status |= _activ)
032  #define _deactivate(x) (bInfo[(x)].status &= (~_activ))
033  #define _ancestor(x)   (bInfo[(x)].ancestor)
034  #define _sibling(x)    (bInfo[(x)].sibling)
035  #define _primogen(x)   (bInfo[(x)].primogen)
036  #define _exit(x)       bInfo[(x)].OnExit
037  #define _entry(x)      bInfo[(x)].OnEntry
038  #define _state(x)      bTrans[(x)].to
039  #define _status(x)     bInfo[(x)].status
040  #define _event(x)      bTrans[(x)].event
041
042  /* Functions to compare elements of arrays passed to quicksort */
043  int CompBlobs(const void *el1, const void *el2)
044  {
045    return (((BNODE*)el1)->state - ((BNODE*)el2)->state);
```

```
046  }
047
048  int CompTrans(const void *el1, const void *el2)
049  {
050    return (((BTRANS*)el1)->from - ((BTRANS*)el2)->from);
051  }
052
053  /* Get internal identifiers to blobs.
054     The relation between internal and external identifiers is hold in
055     blobId and backId elements of the structure WrkMem
056  */
057  void _intBlobId(numBlob root)
058  {
059    _blobId(root) = ++k;
060    _backId((numBlob)k) = root;
061    if (_primogen(root))
062      _intBlobId(_primogen(root));
063    if (_sibling(root) && !_blobId(_sibling(root)))
064      _intBlobId(_sibling(root));
065  }
066
067  /* Number appropriately each node of the tree (root) starting
068     from a given initial value (idInit).
069  */
070  void intBlobId(numBlob root, numBlob idInit)
071  {
072    k = (numBlob)(idInit - 1);
073    _intBlobId(root);
074    for (k=1; k<=nBlob; k++) {      /* Adjust tables to new values */
075      bInfo[k].state = _blobId(k);
076      _primogen(k)   = _blobId(_primogen(k));
077      _sibling(k)    = _blobId(_sibling(k));
078      _ancestor(k)   = _blobId(_ancestor(k));
079    }
```

```
080   for (k=1; k<=nTran; k++) {
081     bTrans[k].from = _blobId(bTrans[k].from);
082     bTrans[k].to   = _blobId(bTrans[k].to);
083   }
084 }
085
086 numBlob Outermost(void)  /* Get outermost blob */
087 {
088   k=1; while (k<=nBlob && _ancestor(k)) k++;
089   return (k);
090 }
091
092 /* Sort transitions by origin state and get the range of relevant
093    transitions to each state.
094 */
095 void handleTransitions(void)
096 {
097   numBlob lastState = 0;
098   qsort(bTrans,nTran+1,sizeof(BTRANS),CompTrans);
099   for (k=1;k<=nTran;k++)
100     if (lastState != bTrans[k].from) {
101       _last(lastState) = (numBlob)(k - 1);
102       _start(lastState = bTrans[k].from) = k;
103     }
104   _last(bTrans[k-1].from) = (numBlob)(k - 1);
105 }
106
107 extern void inflate(numBlob,history);
108 int StartStEng(BNODE *blnd,numBlob qtBlobs,BTRANS *trn,numEvent qtTrans)
109 {
110   nTran  = qtTrans;
111   nBlob  = qtBlobs;
112   bInfo  = blnd;
113   bTrans = trn;
```

```
114    ctlMem = (WrkMem*)malloc((nBlob+1)*sizeof(WrkMem));
115    if (!ctlMem) return (OUTOFMEMORY);
116    memset(ctlMem,0,sizeof(WrkMem)*(nBlob+1)); /* clear */
117    qsort(bInfo,nBlob+1,sizeof(BNODE),CompBlobs);
118    /* Sort binfo by user's identifiers of blobs. This is necessary
119       when changing to new values. A generic entry l of the table has de
120       value l, then we can set internal id with a simple atribution state
121    */
122    intBlobId(Outermost(),1);
123    qsort(bInfo,nBlob+1,sizeof(BNODE),CompBlobs);
124    handleTransitions();
125    inflate(1,_noHist); /* Activate the outermost blob without history */
126    return (0);         /* returns OK!                                */
127  }
128
129  void StopStEng(void)  /* Cease the execution of the run-time */
130  {
131    if (ctlMem) free((char*)ctlMem);
132    finished++;         /* Inhibit the execution of any further action */
133  }
134
135  /* Identify leaf of tree whose ancestors are all active.
136     This leaf is used by the deactivation process.
137  */
138  numBlob activeAtomFrom(numBlob blob)
139  {
140    numBlob blob0;
141    do {
142      blob = _primogen(blob0 = blob);
143      if (blob)
144        while (!_active(blob)) blob = _sibling(blob);
145    } while (blob);
146    return(blob0);
147  }
```

```
148
149 /* Due to the node numeration the following property is always true:
150     if n1 and n2 are two identifiers of nodes in one tree, if we go up in
151     the tree from the higher number (say n2), the first ancestor whose
152     id is less or equal to min(n1,n2) represents the "nearest"
153 */
154 numBlob nearestCommonAncestor(numBlob blob1,numBlob blob2)
155 {
156   if (blob1 > blob2) {
157     numBlob blob0 = blob2;
158             blob2 = blob1;
159             blob1 = blob0;
160   }
161   do { blob2 = _ancestor(blob2); } while (blob2 > blob1);
162   return(blob2);
163 }
164
165 extern void collapse(numBlob);
166 void deactivate(numBlob prior,numBlob blob)
167 {
168   _deactivate(blob);
169   if (_exit(blob)) (_exit(blob))(_backId(blob),0);
170   _vhDesc(blob) = prior;
171   /* deactivate concurrent blobs if exist */
172   if (_concurrent(blob)) collapse(_sibling(blob));
173 }
174
175 void deactivatePath(numBlob blobFrom,numBlob blobTo,int inclusive)
176 {
177   numBlob prior = 0;
178   while (blobFrom != blobTo) {
179     deactivate(prior,blobFrom);
180     blobFrom = _ancestor(prior = blobFrom);
181   }
```

```
182   if (inclusive) deactivate(prior,blobFrom);
183 }
184
185 /* Deactivate concurrent blobs */
186 void collapse(numBlob concurrBlob)
187 {
188   numBlob liveAtom;
189   while (_active(concurrBlob)) {
190     liveAtom = activeAtomFrom(concurrBlob);
191     deactivatePath(liveAtom,concurrBlob,1/*TRUE*/);
192     concurrBlob = _sibling(concurrBlob);
193   }
194 }
195
196 numBlob stepDown(numBlob blobFrom,history hist)
197 {
198   if (hist == _noHist) return(_primogen(blobFrom));
199   if ((hist == _hHist) || (hist == _starHist))
200     return (numBlob)(_vhDesc(blobFrom)?_vhDesc(blobFrom):_primogen(blobFrom));
201   return(0);
202 }
203
204 void activate(numBlob blob,history hist,history* hist1)
205 {
206   _activate(blob);
207   if (_entry(blob)) (_entry(blob))(_backId(blob),1);
208   *hist1 = _history(blob);
209   if (*hist1 == _cancelHist) *hist1 = _noHist;
210   else
211     if ((*hist1 == _noHist) && (hist == _starHist))
212       *hist1 = _starHist;
213 }
214
215 void inflate(numBlob blobFrom,history hist)
```

```
216 {
217   history hist1 = hist;
218   numBlob blob = blobFrom;
219   if (!_active(blobFrom)) { /* FALSE - all concurrent siblings activated */
220     do {
221       blobFrom = blob;
222       hist = hist1;
223       activate(blobFrom,hist,&hist1);
224       blob = stepDown(blobFrom,hist1);
225       if (_concurrent(blobFrom)) {
226         inflate(blob,hist1);
227         inflate(_sibling(blobFrom),hist);
228         blob = 0;
229       }
230     } while (blob);
231     if (active < blobFrom) active = blobFrom;
232   }
233 }
234
235 void activPath(numBlob blobFrom,numBlob blobTo,history hist)
236 {
237   history hist1;
238   numBlob blob;
239   while (blobFrom <= blobTo) {
240     activate(blobFrom,hist,&hist1);
241     if (blobFrom == blobTo) {
242       blob = stepDown(blobFrom++,hist1);
243       if (blob)
244        inflate(blob,hist1);
245       else
246         if (active < blobTo)
247           active = blobTo;
248     }
249     else {
```

```
250       blob = _vhDesc(blobFrom);
251       if (_concurrent(blobFrom)) {
252         activPath(blob,blobTo,hist1);
253         inflate(_sibling(blobFrom),hist);
254         blobFrom = (numBlob)(blobTo+1);
255       }
256       else {
257         blobFrom = blob;
258         hist = hist1;
259       }
260     }
261   } /* while */
262 }
263
264 void activatePath(numBlob blobFrom,numBlob blobTo)
265 {
266   history hist;
267   numBlob blob0,blob = blobFrom;
268   do {        /* sensing history enforcement */
269     hist = _history(blob);
270     blob = _ancestor(blob);
271   } while (blob && (hist == _noHist));
272   if ((hist == _cancelHist) || (!blob))
273     hist = _noHist;
274   blob = blobTo; /* tracing the activation Path */
275   do {
276     blob0 = blob;
277     _vhDesc(blob = _ancestor(blob)) = blob0;
278   } while (blob > blobFrom);
279   activPath(_vhDesc(blobFrom),blobTo,hist);
280 }
281
282 void fromBlobToBlob(numBlob origin,numBlob destination,ptrFunc act)
283 {
```

```
284   numBlob liveAtom,commonAncestor;
285   liveAtom = activeAtomFrom(origin);
286   commonAncestor = nearestCommonAncestor(origin,destination);
287   deactivatePath(liveAtom,commonAncestor,0/*FALSE*/);
288   if (act) (*act)(_backId(origin),_backId(destination));/* trigger action */
289   if (commonAncestor == destination) {
290     destination = stepDown(destination,_history(destination));
291     inflate(destination,_history(destination));
292   }
293   else activatePath(commonAncestor,destination);
294 }
295
296 int _gtact(numBlob root)
297 {
298   bInfo[root].status |= _visited;
299   if (_active(root))
300     if (root == k) return (1); /* k = blob desired. */
301   if (_primogen(root))        /*See function below */
302     if (_gtact(_primogen(root))) return (1);
303   if (_sibling(root) && !(_status(_sibling(root)) & _visited))
304     if (_gtact(_sibling(root))) return (1);
305   return (0);
306 }
307
308 int in_Blob(numBlob blob)
309 {
310   for (k=1; k<=nBlob; ++k) bInfo[k].status &= ~_visited;
311   k = _blobId(blob);    /* k is used in next call!! */
312   return (_gtact(1));
313 }
314
315 numBlob nextActive(numBlob blob)
316 {
317   do { if(++blob>nBlob) return(0); } while (!_active(blob));
```

```
318   return(blob);
319 }
320
321 void _steng(numEvent event)
322 {
323   unsigned int count,begin,end,dif;
324   if (!event) {
325     deactivatePath(activeAtomFrom(1),1,1);
326     free((char*)ctlMem);
327     return;
328   }
329   active = 1;
330   do {
331     begin = _start(active); end = _last(active);
332     dif   = end - begin; dif++; count = 0;
333     while (count < dif) /* There are events from this state*/
334     {                   /*pass every event of this state   */
335       if (event == _event(begin+count))
336         if (bTrans[begin+count].cond) {
337           if ((bTrans[begin+count].cond)()) {
338             fromBlobToBlob(active,_state(begin+count),
339     bTrans[begin+count].action);
340             return; /* outermost transition fired */
341           }
342         }
343         else {
344           fromBlobToBlob(active,_state(begin+count),
345   bTrans[begin+count].action);
346           return;
347         }
348       count++; /* get next event */
349     }
350     active = nextActive(active);
351   } while (active > 0);
```

```
352  }
353
354  int stEng(numEvent event) /* Return 0 if event handled appropriately */
355  {
356    static int called = 0;
357    ++called;
358    if (called > 1) {   /*Avoid recursive call*/
359      called--;
360      return (RECURSIONNOTALLOWED);
361    }
362    if (finished) return (FINISHED);
363    _steng(event);
364    if (!event) finished++;
365    return (--called);
366  }
```

8.3 st.h — blobs and events id definition

```
001  /* st.h ---  Header for tables that specify an statechart */
002
003  #ifndef STH
004  #define STH
005
006  #define Timer      1 /* Rule for the user to define external  */
007  #define Alive      2 /* numerical blob id. Each id must be    */
008  #define InSec      3 /* in the range 1..n, where n represents */
009  #define Dead       4 /* the total number of blobs of the      */
010  #define Stopwatch  5 /* statechart                            */
011  #define Reset      6
012  #define Operation  7
013  #define Display    8
014  #define Normal     9
015  #define On        10
016  #define Off       11
```

```
017
018  #define Exit 0 /* The values associated to events are irrelevant to */
019  #define e1   1 /* the run-time algorithm.                          */
020  #define e2   2
021  #define e3   3
022  #define e4   4
023  #define e5   5
024
025  extern BTRANS bTrans[];/* transitions                              */
026  extern BNODE  bInfo[]; /* topology. The programmer must pass these */
027          /* tables through call to StartStEng       */
028
029  #endif
```

8.4 st.c — specification tables

```
001  /* st.c --- Statechart specification tables */
002
003  #include "engine.h"
004  #include "st.h"
005
006  int cond(void);
007
008  /* Sole function (for simplicity) called whenever */
009  /* entry or exit a state                          */
010  extern void Show(int,int);
011  extern void trans(int,int);
012
013  BTRANS bTrans[] =
014  {/* state, event,state, condition, action functions */
015    { 0,         0,   0,           0, 0     },
016    { Stopwatch, e5,  Dead,        0, 0     },
017    { Reset,     e2,  Operation,   0, 0     },
018    { Reset,     e4,  Reset,       0, trans },
```

```
019   { Operation, e4,   Reset,        0, 0     },
020   { Normal,    e3,   InSec,        0, 0     },
021   { InSec,     e3,   Normal,       0, 0     },
022   { Off,       e2,   On,           0, 0     },
023   { On,        e2,   Off,          0, 0     },
024   { Alive,     e4,   Dead,      cond, 0     },
025   { Dead,      e1,   Stopwatch,    0, 0     },
026   { Dead,      e2,   Operation,    0, 0     }
027  };
028
029  BNODE bInfo[] = {
030  /*state, primogen, sibling,ancestor,      status, entry,exit */
031  { 0,          0,     0,          0,               0,  0,   0   },
032  { Dead,       0,     Alive,      Stopwatch,_noHist,  Show,Show},
033  { Stopwatch, Alive, Stopwatch, 0,          _noHist,  Show,Show},
034  { Alive,      Reset, Dead,       Stopwatch,_starHist,Show,Show},
035  { Reset,      0,     Operation, Alive,     _noHist,  Show,Show},
036  { Operation, Timer, Reset,      Alive,     _noHist,  Show,Show},
037  { Timer,      On,    Display,    Operation,_concurr, Show,Show},
038  { Display,    Normal,Timer,      Operation,_concurr, Show,Show},
039  { On,         0,     Off,        Timer,     _noHist,  Show,Show},
040  { Normal,     0,     InSec,      Display,   _noHist,  Show,Show},
041  { Off,        0,     On,         Timer,     _noHist,  Show,Show},
042  { InSec,      0,     Normal,     Display,   _noHist,  Show,Show}
043  };
```

8.5 stopwatc.c — code of Windows program example

```
001  /* stopwatc.c -- Program Example of Statechart + Windows */
002
003  #include <windows.h>
004  #include "engine.h"
005  #include <string.h>
006  #include "st.h"
```

```
007
008  LRESULT FAR PASCAL WinProc(HWND,UINT,WPARAM,LPARAM);
009  static HWND mHwnd;
010  RECT retang;
011  int  cxCarac,cyCarac;
012
013  int cond(void) {        /* Condition function that guards the transition */
014     return in_Blob(Off); /* labelled e4 from Reset to Reset.              */
015  }
016
017  void trans(int state, int inOut) {
018     /* call presentation code */
019     ShowBehaviour(mHwnd,"Transiton from Reset to Reset");
020  }
021
022  extern void ShowBehaviour(HWND,LPSTR);
023  void Show(int state,int inOut)
024  {
025    char sChar[80];
026    static LPSTR names[] = { "Timer","Alive","InSec","Dead",
027       "Stopwatc","Reset","Operation","Display","Normal","On","Off"
028    };
029    static LPSTR op[] = { "OnExit ", "OnEntry" };
030    wsprintf((LPSTR)sChar,"%s %s",op[inOut],names[state-1]);
031    ShowBehaviour(mHwnd,sChar);
032  }
033
034  void ShowBehaviour(HWND hWnd, LPSTR str)
035  {
036    char szBuffer[80];
037    HDC hdc;
038    ScrollWindow(hWnd,0,-cyCarac,&retang,&retang);
039    hdc = GetDC(hWnd);
040    TextOut(hdc,cxCarac,retang.bottom-cyCarac,szBuffer,
```

```
041   wsprintf((LPSTR)szBuffer,"%-60s",(LPSTR)str));
042   ReleaseDC(hWnd,hdc);
043   ValidateRect(hWnd,NULL);
044 }
045
046 int PASCAL WinMain(HINSTANCE hInst,HINSTANCE hprvInst,
047     LPSTR lpszLinhaCmd,int nCmdMostrar)
048 {
049   char szNomeAplic[] = "Stopwatch";
050   MSG msg;
051   lpszLinhaCmd = NULL;
052   if (!hprvInst) {
053     WNDCLASS wC;
054     wC.style         = CS_HREDRAW | CS_VREDRAW;
055     wC.lpfnWndProc   = WinProc;
056     wC.cbClsExtra    = 0;
057     wC.cbWndExtra    = 0;
058     wC.hInstance     = hInst;
059     wC.hIcon         = LoadIcon(NULL,IDI_APPLICATION);
060     wC.hCursor       = LoadCursor(NULL,IDC_ARROW);
061     wC.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
062     wC.lpszMenuName  = NULL;
063     wC.lpszClassName = szNomeAplic;
064     RegisterClass(&wC);
065   }
066   mHwnd = CreateWindow(szNomeAplic,"StopWatch Example",
067           WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,CW_USEDEFAULT,
068           CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,hInst,NULL);
069   ShowWindow(mHwnd,nCmdMostrar);
070   UpdateWindow(mHwnd);
071
072   StartStEng(bInfo,14,bTrans,14); /* Put the statechart engine in motion */
073   while (GetMessage(&msg,NULL,0,0)) {
074     TranslateMessage(&msg);
```

```
075     DispatchMessage(&msg);
076   }
077   StopStEng();                          /* Stop the statechart engine */
078   return msg.wParam;
079 }
080
081 LRESULT FAR PASCAL WinProc(HWND hWnd,UINT mensagem,
082     WPARAM wParam,LPARAM lParam)
083 {
084   char szBuffer[80];
085   HDC hdc;
086   PAINTSTRUCT ps;
087   TEXTMETRIC tm;
088   LPCSTR cab = "Transitions            Events";
089
090   switch (mensagem)
091   {
092     case WM_CREATE:
093       hdc = GetDC(hWnd);
094       GetTextMetrics(hdc,&tm);
095       cxCarac = tm.tmAveCharWidth;
096       cyCarac = tm.tmHeight;
097       ReleaseDC(hWnd,hdc);
098       retang.top = 2*cyCarac;
099       return 0;
100
101     case WM_SIZE:
102       retang.right  = LOWORD(lParam);
103       retang.bottom = HIWORD(lParam);
104       return 0;
105
106     case WM_PAINT:
107       hdc = BeginPaint(hWnd,&ps);
108       TextOut(hdc,cxCarac,cyCarac/2,cab,lstrlen(cab));
```

```
109        EndPaint(hWnd,&ps);
110        return 0;
111
112      case WM_CHAR:
113        wParam -= '0';  /* get logic events */
114        wsprintf(szBuffer,"%35d",wParam);
115        ShowBehaviour(hWnd,szBuffer);
116        if (stEng((numEvent)wParam)==FINISHED)
117          DestroyWindow(hWnd);
118        return 0;
119
120      case WM_DESTROY: PostQuitMessage(0);
121      return 0;
122    }
123    return DefWindowProc(hWnd,mensagem,wParam,lParam);
124  }
```

8.6 Definition file for Windows

```
001  NAME        Stopwatch
002  DESCRIPTION 'Example Stopwatch'
003  EXETYPE     WINDOWS
004  STUB        'winstub.exe'
005  CODE        PRELOAD MOVEABLE DISCARDABLE
006  DATA        PRELOAD MOVEABLE MULTIPLE
007  HEAPSIZE    1024
008  STACKSIZE   8192
```

8.7 Makefile

```
001  #dos.mak -- Makefile of the Stopwatch example for DOS (MS C/C++)
002
003  COMP = /c /W4 /WX /Od
004
005  .c.obj:
```

```
006     cl $(COMP) /Tp$*.c
007
008  stopwatc.exe: engine.obj st.obj stopwatc.obj
009    link /co $**,stopwatc.exe;
010
011  engine.obj: engine.c engine.h
012  st.obj: st.c st.h engine.h
013  stopwatc.obj: stopwatc.c engine.h
```

References

[BC91] Len Bass and Joëlle Coutaz. *Developing Software for the User Interface*. SEI Series in Software Engineering. Addison-Wesley Publishing Company, Inc., 1991.

[BW91] Michael A. Bertrand and Willian R. Welch. Programming Windows Using State Tables. *Supplement to Dr. Dobb's Journal*, December 1991.

[DH89] Doran Drusinsky and David Harel. Using Statecharts for Hardware Description and Synthesis. *IEEE Transactions on Computer Aided Design*, 8(7):798–807, July 1989.

[FB87] Mark A. Flecchia and R. Daniel Bergeron. Specifying Complex Dialogs in ALGAE. In *Proceedings of the ACM CHI+GI'87 Conference*, pages 229–234, Toronto, Canada, April 1987.

[FL93] 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.

[Gre85] Mark Green. Report on Dialogue Specification Tools. In Günther E. Pfaff, editor, *User Interface Management Systems*, pages 9–20. Springer-Verlag, 1985.

[Gre86] Mark Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HH89] H. Rex Hartson and Deborah Hix. Human-Computer Interface Development: Concepts and Systems for its Management. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, April 1990.

[Jac86] Robert J. K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[Mye89] Brad A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software*, pages 15–23, January 1989.

[vZM91] 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.

[Was85] 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.

[Wel89] 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*

19/93 **Modelamento, Simulação e Síntese com VHDL,** *Carlos Geraldo Krüger e Mário Lúcio Côrtes*

20/93 **Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour,** *Fábio Nogueira de Lucena e Hans Liesenberg*

21/93 **Applications of Finite Automata in Debugging Natural Language Vocabularies,** *Tomasz Kowaltowski, Cláudio Leonardo Lucchesi e Jorge Stolfi*

22/93 **Minimization of Binary Automata,** *Tomasz Kowaltowski, Cláudio Leonardo Lucchesi e Jorge Stolfi*

23/93 **Rethinking the DNA Fragment Assembly Problem,** *João Meidanis*

24/93 EGOLib — **Uma Biblioteca Orientada a Objetos Gráficos,** *Eduardo Aguiar Patrocínio, Pedro Jussieu de Rezende*

25/93 **Compreensão de Algoritmos através de Ambientes Dedicados a Animação,** *Rackel Valadares Amorim, Pedro Jussieu de Rezende*

26/93 GeoLab: **An Environment for Development of Algorithms in Computational Geometry,** *Pedro Jussieu de Rezende, Welson R. Jacometti*

27/93 **A Unified Characterization of Chordal, Interval, Indifference and Other Classes of Graphs,** *João Meidanis*

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

reltec@dcc.unicamp.br