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).)

A Statechart Engine to Support Implementations of Complex Behaviour

Fábio N. de Lucena
fabio@dcc.unicamp.br

Hans K.E. Liesenberg
hans@dcc.unicamp.br

Relatório Técnico DCC–94-01

Março de 1994

A Statechart Engine to Support Implementations of Complex Behaviour

Fábio N. de Lucena
fabio@dcc.unicamp.br

Hans K.E. Liesenberg
hans@dcc.unicamp.br

Projeto Xchart*
DCC/IMECC/UNICAMP
Caixa Postal 6065
13081-970 Campinas/SP

Abstract

The Statechart notation was deviced for specifying complex behaviour of reactive systems. The gap between a behaviour described in terms of Statecharts and its implementation, however, is still very large due to its intricate semantics and its main purpose as specification language. In order to overcome this gap we propose a new technique which frees the programmer from the subtle and error-prone activity of mapping a specification into an implementation. The technique is centred upon a reactive kernel.

The reactive kernel consists of an invariant *engine* which interprets statechart specifications. Actions are suplied in terms of a set of subroutines in C. The engine invokes these subroutines at proper instants during the application execution as a reaction to particular events. The reactive kernel implementation is the major issue of this paper. In order to exemplify our proposal the behaviour of a toy application is employed.

*Project that deals with the development (specification and implementation) of computer-human interfaces. It is supported partially by CNPq and FAPESP. Just as a curiosity: the 22$^{\text{nd}}$ letter of the greek alphabet ($\mathbf{X}$ e χ) is identified in English by the name CHI which represents the acronym of *Computer-Human Interface.*

1 Introduction

A reactive system is event-driven and it has to react continuously to external and internal stimuli (operating systems, telephone switchers and human-computer interfaces are examples of reactive systems). A reactive program may be seen as a composition of three layers [1]:

- An *interface* that deals with the input reception of physical events and maps them to logical events handled by the reactive kernel. It is also in charge of the converse operation.
- A *reactive kernel* that provides the behavioural pattern of the reactive program. It decides what outputs must be generated in reaction to inputs.
- A *data handling* layer that performs the functionality whenever requested by the reactive kernel.

In a reactive system the reactive kernel represents the most difficult part to be developed. The problem of finding satisfactory methods to describe reactive kernels is taken to be particularly difficult, despite of the existence of specific languages for this purpose [12]. Statecharts [10] have been proposed as an alternative specification notation for this particular domain.

The statechart notation has already been used to specify a large number of real-world systems [14]. It is the central part of the STATEMATE [11] environment which supports the specification and design process of large systems. The Statemaster system [26] provides this kind of support for the construction of user interfaces (a similar use of statecharts is made in [25]). Statecharts are as well used as a graphical notation in object-oriented software development methodologies [4, 24] and in hardware design [5]. A great number of efforts directed towards its formal semantic specification and extension proposals [13, 14, 16, 17, 18, 19, 21, 22] show signs of a vivid interest in this particular notation.

This work sees the statechart notation as a joint specification/programming language for the development of reactive kernels. This language is not a complete programming language. For this reason the interface and the data handling functions of reactive systems must be developed with the aid of a conventional programming language. Tools like STATEMATE and Statemaster mentioned above provide behavioural control based on statechart specifications. Our technique is similar in relation to this aspect. This paper comments in greater detail a software module (engine[1]) that implements a statechart-driven reactive kernel and describes how to use it. Related to our approach we can cite [7], where a technique to implement statecharts is proposed based on the object paradigm. However, in this system (1) the statechart specification must be translated to some object-oriented language (a simple modification requires a change of code, a recompilation and a new batch of tests); (2) there is an overhead at execution time due to the use of virtual functions; and (3) only a small subset of the statechart functionality is properly tackled. In [2] *state tables* are proposed to encapsulate a program control flow based on state transition diagrams (STDs) by means of an array of pointers. State tables are used to handle low level events (e.g., mouse events). In [23] the use of STDs to represent program behaviour is recommended. All these approaches, however, suffer from a common drawback: they are applicable to simple behavioural patterns. Complex control flows cannot be managed adequately by those proposals.

The technique presented here does not impair the development process since it does not require a translation of statechart behaviour to some programming language constructs. It can as well handle behavioural changes at execution time. In order to demonstrate the usefulness of the technique and to show how the responsibility of the implementation of complex behaviour is taken over by an invariant engine, the implementation of a small stopwatch application is discussed in greater detail. The main purpose of this example is to show how a complex behaviour can be specified and implemented and not to show how realistic it can be.

[1] In Section 8 an `ftp` address is provided where from the source code of engine and a descriptive text can be downloaded.

In Section 2 few comments are made about the behavioural specification of interactive applications. It illustrates why applications like our stopwatch example could be difficult to implement. Section 3 points out the difficulties that raise with the STDs notation when complex behaviour is being captured. Section 4 briefly comments on aspects of the statechart notation related to the stopwatch example presented in Section 5. The technique is presented in Section 6 and its use with respect to our example in Section 7. The technique relies on an invariant engine described in Section 8. Extensions in progress and concluding remarks are given in Section 9.

2 Reactive kernels

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 is affected by asynchronous events.[2] Interactive programs have to fire appropriate actions according to incoming 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.

Different models and notations have been used to describe the interaction between an user and a computer [15]. STDs and one of their extensions, namely statecharts, shall be discussed in sections below.

A parallel between the Seeheim interface reference model proposed by Green [9] and a taxonomy [1] for reactive systems is drawn in Figure 1. The model is useful to identify which component of an user interface can be adequately described in terms of statecharts. This is the first

[2] Events are commonly generated by user actions upon input devices like a mouse or a keyboard in unpredictable sequences.

reason of mentioning the model here. The second is to show that there is no loss of generality when we illustrate the proposed technique with an interactive application.

The Seeheim model divides an user interface into three layers. The presentation layer is responsible for the external presentation of the user interface. This layer defines how an interactive system appears and is felt by the end user. It realizes low-level input/output processing (lexical aspects). The dialogue control layer accepts inputs from the lexical layer and assembles them into commands and data on the one hand, and receives tokens from the application which require data and supply responses to user requests on the other hand. The last layer represents the functionality (semantics) of an interactive application from the perspective of the user interface.

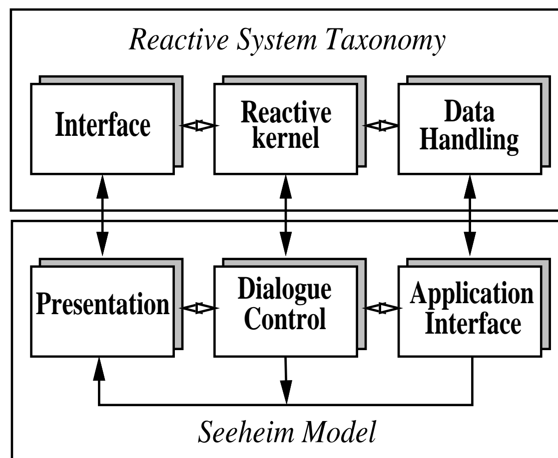

Figure 1: Parallel between the Seeheim model [9] and a taxonomy for reactive systems [1].

A rough correspondence between the components of a reactive system according to the taxonomy presented before and the layers of the Seeheim model can be established quite easily. The interface/presentation layer is concerned with lexical aspects. The reactive kernel/dialogue control

(provided by our engine together with a statechart specification) is responsible for control flow (syntactic) features. The functional/application layer refers to the semantics (functionality) provided by the system.

STDs are used to specify the dialogue control layer. The control process of a statechart is more sophisticated, but it avoids some inherent problems of STDs. Both specification alternatives are graphical. This feature represents an advantage over textual descriptions employed by other specification languages. Further information about languages employed to describe a dialogue control layer can be found in [15].

3 State transition diagrams

One well-known way of representing an event-driven control is based on STDs. During the execution of an STDs, one particular state is considered to be 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.

STDs are essentially "flat." The lack of means to define contexts incrementally 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 STDs.[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. Another disadvantage is the lack of constructs to represent concurrency explicitly since exactly one state represents the current state of the system.

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

4 Statecharts

The statechart notation[4] has been proposed to specify the kernel of reactive systems [10]. It extends STDs and eliminates some of their shortcomings. The statechart notation produces more concise specifications [6] than other notations of similar purpose. It supports incremental context definitions related to hierarchical decompositions, control flow based on history and explicit concurrency specifications. These features on the other hand make the control process rather sophisticated.

Statecharts are described in terms of nested contexts represented graphically by non-overlapping[5] rectangles with rounded corners called states. The nesting reflects successive decompositions. States can be of two kinds: mutually exclusive and concurrent states. The former is indicated by a solid and the latter by a dashed contour.[6] Figure 2 depicts one example. If a context is decomposed in terms of mutually exclusive states (as state **Stopwatch** in Figure 2), only one of those (**Dead**, or **Alive** in the particular case) becomes active at a given instance whenever their direct ancestor becomes active. On the other hand, if a context is decomposed into concurrent states, whenever this context becomes active, all of its concurrent states become active as well. For example, whenever the state **Operation** is active, all of its direct descendants (**Timer** and **Display**) are active too. The dashed contours of these states represent this fact.

If a given state is decomposed into mutually exclusive states, then one of the subordinated states 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**. 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 acti-

[4] The brief description below is not meant to be a tutorial and we would like to give notice that a subset of statechart constructs and not necessarily the same terminology proposed in [10] has been adopted in this paper.

[5] Overlapping states are considered in [14].

[6] A dashed contour of states does not exist in the orginal statechart notation. It, however, makes the state identification labelling more homogeneous.

vation process due to a history condition or if it does not represent the destination state of a transition.

A transition is represented by a directed edge between two mutually exclusive states and, if fired, causes a context swapping. A transition edge is labelled with an event identifier and is possibly associated with a guarding condition. A transition is usually fired whenever the origin state 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_State(Off)]* of the transition from **Alive** to **Dead** will enable this transition whenever the state **Alive** is active, the event *e4* occurs, and if at this instant the guarding condition (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 the parsing of valid event sequences from a statechart. There is a need of having a mechanism to produce some output (actions) to control the data manipulation procedures (functional aspect) of the application. This mechanism is implemented in terms of function calls in the current proposal. The Section 6 gives more details about this issue.

5 Stopwatch example

Figure 2 shows the statechart description of a hypothetical stopwatch (a brief explanation of it is given below). It is worthwhile to note that we refer to abstract events like *e1* and *e2*, for instance. As already stated earlier the responsibility of relating physical stimuli to these events is of the interface component. Section 6 discusses this question in more detail.

Stopwatch corresponds to the outermost state 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. If the event *e4* takes place, then the state is left and revisited and the same related semantic actions are performed once more. In particular, the subroutine `trans` is called whenever the transition from **Reset** to **Reset** takes place. 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 (i.e., **On** and **Normal** respectively). On the contrary, the most recently substates are reactivated because of the in-depth history attribute[7] 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.

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* takes place. In the 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. Note that this specification corresponds to an observable behaviour of a stopwatch. It does not describe anything about its functionality, but says what must happen according to the behaviour the user can perceive.

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** which has

[7] The corresponding symbol has been put in the left upper corner of **Alive**.

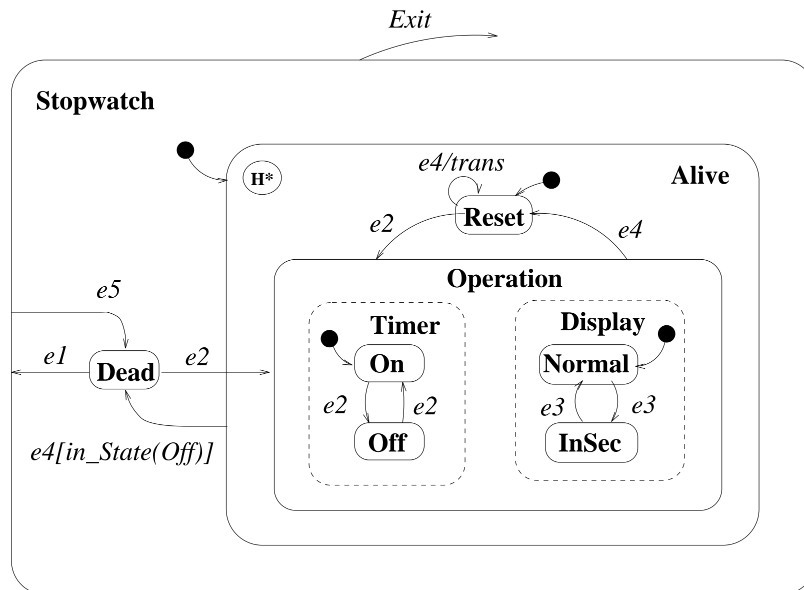

Figure 2: A simple stopwatch example.

no in-depht history attribute. In the second case **Alive** becomes active because it is the ancestor of the destination state of the fired transition. In the third case **Dead** is deactivated and reactivated and in the last case the animation of **Stopwatch** ceases. Whenever **Alive** or **Operation** is revisited the activation of their substates complies with 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 carried out and the system comes to a halt.

6 Proposed approach

The approach presented here assumes that there is a specification in terms of statechart which describes the behaviour of a reactive system to be implemented. At this point the programmer must know a little more about the architecture of reactive programs developed with the aid

of this approach. Figure 3 helps us in this respect.

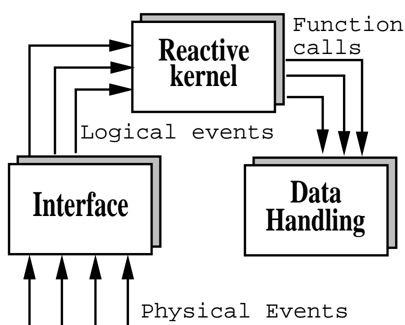

Figure 3: Reactive program architecture based on the proposed approach.

The components in Figure 3 were commented earlier on in this text. The interface establishes a correspondence between physical events and logical events, which are used in the specification. The reactive kernel is provided by the engine described in Section 8. In essence, the engine reacts to events signalled by the interface by calling functions which carry out operations (actions) responsible for the functionality (data handling functions) of the reactive program. In order to pass events to the engine, turn it on/off and get information from it, the programmer must use the protocol described in the following section.

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.[8] A function associated to a transition is called after the deactivation of the origin state and before the activation of destination state.[9] So, whenever the engine is notified about an event occurrence the firing capability of transitions is checked. If some transition is capable of being traversed, the

[8]These outputs are analogous with the outputs of Mealy and Moore machines [3]. In Moore machines, however, output is produced only when one state is reached. A statechart output may also be produced upon leaving a state.

[9]The deactivation/activation process is described in more detail in section 8.

proper function associated with every state entered/exited as a result of the transition is called.

In order to react properly to events the engine must be fed with statechart specification and after that it has to wait for incoming events. A specification is passed on to the engine in terms of two manually constructed tables.[10] The way of how to construct these two tables according to a given statechart shall now be discussed.

The data related to a statechart topology is kept in a tree structure which reflects the hierarchy defined by its topology. The correspondence between states and nodes of the tree is an one-to-one relation. Figure 11 shows a tree which is equivalent to the hierarchical structure of the statechart described in Figure 2. (This tree is a binary tree by sheer chance. It can be of any order.)

The engine controls basically the transition firings and the state activation and deactivation process through function calls. This code is used for whatever statechart specification we might have. So, the pertinent code can be compiled once and linked with other system components as often as required.

6.1 Engine protocol

The functionality provided by the engine is exercised by means of four functions. We omit the complete signature (prototype) of these functions for the sake of simplicity. Two of them start and stop the operation of the engine. They essentially carry out some housekeeping functions.

▪ `StartStEng`

This function receives the tables which describe a particular statechart specification and submits them to some transformations. Only after a call of this function the engine is capable to react properly to events.

[10] Tables are not the best way to hand over information to the engine, because their manual construction process is error-prone. The engine does not realize any automatic consistency check. The construction of those tables is better done by a compiler of statechart specifications into the required data structures. A proper environment of this kind is described in [8].

■ `StopStEng`
It informs the engine that its execution should cease. The animation of a specification begins after a call of the previous function and it comes to an end when this one is called. In a program one may have more than one unnested `StartStEng/StopStEng` pair.

The remaining functions are:

■ `stEng`
It signals an event occurrence to the engine, i.e., an external stimulus is sensed by the interface module, converted to a numerical value understandable by the reactive kernel and handed over to the engine. This function should be called only after a successful call of `StartStEng`.

■ `in_State`
It verifies if a given state is active or not. It is usually invoked by a transition guard function.

6.2 Topology data structure

The topology of a specification in terms of statechart is handed over to the engine by means of an array. The definition of each element of the array is given in Figure 4. 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`), status information (`status`) and two pointers to functions to be called whenever the corresponding state is entered (`OnEntry`) or left (`OnExit`) respectively, i.e., whenever it is activated or deactivated. A similar field (`action`) exists in `BTRANS` (Figure 6) which specifies a function to be called whenever the corresponding transition is fired.

The `status` information indicates the existence or not of a shallow history condition, an in-depth history, a history condition cancellation defined at the corresponding state, and tells if the state in question is a concurrent or a mutually exclusive state. This information is set by an `or` composition of the relevant values in Figure 5.

```
typedef struct {
  numState state,
           primogen,
           sibling,
           ancestor;
  bStatus status;
  ptrFunc OnEntry, OnExit;
} BNODE;
```

Figure 4: Data structure for a state.

```
#define _noHist      0x00
#define _hHist       0x01
#define _starHist    0x02
#define _cancelHist  0x04
#define _concurr     0x08
```

Figure 5: Masks for status description.

6.3 Transitions data structure

Transitions are as well handed over to the engine in terms of an array. Every transition is associated with an element of the `BTRANS` type (Figure 6). For each transition the origin (`from`) and the destination state (`to`), the identifier of the event (`event`) that possibly triggers the transition, one pointer to a guarding condition function (`cond`) and another to a function which represents a semantic action (`action`) must be specified.

One element of this type asserts that if state `from` is active and `event` occurs, then the function `cond` is called and its boolean result tested. If it is true, the transition to state `to` takes place and function `action` is called during this process.

```
typedef struct {
  numState  from;
  numEvent event;
  numState  to;
  int (*cond)(void);
  ptrFunc action;
} BTRANS;
```

Figure 6: Data structure for a transition.

7 Example implementation

This section describes the implementation of a small statechart specification presented in Section 5. It is not a very realistic example, but it can be used as a template of the development of complex behaviour, particularly of event-driven applications based on the proposed reactive kernel, i.e. the engine.

The next subsections describe respectively how transitions and the topology of a given statechart specification (Figure 2) are represented in the tables required by the engine. First we must identify states and events in order to build the required tables.

7.1 Identifying states and events

Each state in a specification has to be assigned to a numerical value. The Figure 7 is a possible association of states and identifiers in C. This numerical value represents its identifier recognized by the engine.

Events are as well associated to numerical values. The implemented engine algorithm does not depend on a particular enumeration process of events. An abstract event used within a statechart might correspond to a single physical event or a sequence of physical events. The event `e1`, for instance, could represent a message sent by some window manager to the application. The responsibility of binding physical to abstract events is of the interface component mentioned earlier on. In order to simplify our reactive program a simple and direct binding shall be used

```
#define Timer     1 /* ASSIGNMENT RULE   */
#define Alive     2 /* Each numerical id */
#define InSec     3 /* to be assigned to */
#define Dead      4 /* a state must be   */
#define Stopwatch 5 /* in range [1..n],  */
#define Reset     6 /* where n           */
#define Operation 7 /* represents the    */
#define Display   8 /* total number of   */
#define Normal    9 /* states of the     */
#define On       10 /* underlying        */
#define Off      11 /* specification     */

#define Exit 0 /* The way numerical event*/
#define e1   1 /* id assignments are made*/
#define e2   2 /* has no effect on the   */
#define e3   3 /* engine operation.Except*/
#define e4   4 /* for the event to stop  */
#define e5   5 /* the engine operation   */
            /* identified by 0       */
```

Figure 7: Defining states and events.

where each event $\mathtt{e}_k$ is generated by pressing the key k on the keyboard. Thus the logical event that corresponds to the physical action of the user pressing the key 5 is the event *e5*, for instance.

7.2 Describing transitions

Transitions are labelled and establish a relation between two states, 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.

Figure 8 describes the transitions of Figure 2. It is important to point out that default transitions are a special case and are described implicitly by the topology tree (Figure 11). The leftmost descendant of

```
BTRANS bTrans[] =
{/* from,  event, to,        cond,action */
{ 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 8: Transition description.

a state is taken as its default direct substate.

Two particular transitions shall be commented. The transition from **Alive** to **Dead** has a guard condition (Figure 2). This is represented by a pointer to a function returning a boolean value in the corresponding field of transition table (Figure 8). In this case the proper field is occupied by the address of the function `cond`, which calls the function `in_State` provided by the engine (Figure 9).

```
int cond(state) {
    return in_State(state);
}
```

Figure 9: Guarding condition function.

The other transition goes from **Reset** to **Reset**. In the field `action` an address to a function is specified. As already mentioned, this function is called whenever event *e4* occurs and **Reset** is activated.

7.3 Describing topology

In this section the representation of a statechart topology is commented. In order to describe a particular topology one entry for each state in the table `bInfo` (Figure 10) 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 11.

In the particular case of the state **Stopwatch**, for instance, its entry in the table identifies **Alive** 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 `foo()` is called by the engine.

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

Figure 10: Topology description.

8 More about the engine

This section presents more detailed information about a particular implementation of the complex statechart semantics, referred to as engine

in this text. The engine has been implemented in ANSI C and has been used in different experimental developments. It is available (full source code) electronically by anonymous `ftp` at `obelix.unicamp.br` in the `/pub/dcc/Xchart/engine` directory.

The engine represents the invariant code which can be linked with any program developed according to the proposed technique. It performs, in essence, event sensitivity checks and controls the state activation and deactivation process. The engine interprets a statechart specification according to a data structure (Section 6.1) holding information about a particular statechart topology and its transition lattice. The information held by these data structures must be supplied and must reflect the topology and the attributes of the system's statechart specification.

It is important to point out that the engine presented here implements only a subset of the statechart features originally defined in [10]. This subset is illustrated by means of an example presented in Section 5. Anything else not mentioned in this text has not been contemplated.

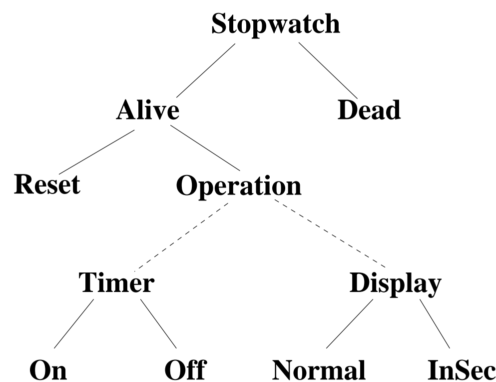

Figure 11: Tree representing the topology.

8.1 Engine execution

This section gives more details about the engine at execution time. A technical presentation of it can be found in [20] available electronically

by anonymous `ftp` at `obelix.unicamp.br` in the `/pub/dcc/RT` directory. It is important to point out that the operational semantics for a complex language like statecharts is not adequately presented in a natural language. A formal description can be found in [13]. The description below gives only a little flavour of the statechart semantics.

A program implemented based on the proposed technique must call `StartStEng` to signal its intent to use the engine. `StartStEng` receives the addresses and the dimensions of the tables which describe the topology and transitions of a particular statechart specification to be used by the engine in order to make it behave according to this specification. This approach makes it possible to change the behaviour without the need to recompile any code.

The control structure, which guides context swappings, is independent of the topology of a specific statechart, so sometimes we use the term invariant engine. The main function `stEng` receives as its argument an event identifier. This function traverses the list that holds the identifiers of active states (this list is referred to as the configuration of a statechart) and verifies the sensitivity of those states to the current event.

A context swapping due to a transition firing is carried out in two steps. At first all states from the atomic state reachable from the origin state, up to the nca (nearest common ancestor[11]) of the origin and the destination state (excluding the latter), have to be deactivated as well as all concurrent siblings and their descendants along this path Next the states of the path between the destination state and the nca mentioned above (excluding the latter) are activated in reverse order. From this point the activation process is kept up until an atomic state is eventually

[11] The concept of the nca is degenerated in two cases: the first case is characterized if the destination state is a direct or indirect descendant of the origin state (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**). In both cases the state which represents the ancestor of the second one of the pair (destination,origin) is taken as the nca. If however the origin and the destination state of a transition edge happen to be the same as the case of the transition labelled *e4* at **Reset**, then the the direct ancestor is taken as the nca.

reached. All concurrent siblings of activated states at this second stage of the context swapping process are activated as well.

The transition firing function[12] identifies an active atomic state reachable from the origin state,[13] it finds out the nca of the origin and the destination state,[14] it deactivates all states along the path from the active atomic state found previously up to the nca (excluding the latter) as well as all concurrent components along this path. It then calls the function representing the semantic action related to the transition (if specified), and starts the activation process.[15]

The first step of the activation process consists of a search of possible history enforcements at the nca and levels above and a demarcation of the path from the destination state up to the nca. This path is then followed from the nca down to the destination state and all states along this path (except for the former which after all has not been deactivated) are activated. Concurrent siblings are activated according to the history mode being enforced. Once the destination state is reached, the activation process is sustained, but now complying with history enforcements, until an atomic state 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 state is reactivated; if no such state 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 global 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 con-

[12] `fromBlobToBlob`, in the engine source code
[13] `activeAtomFrom`
[14] `nearestCommonAncestor`
[15] `activatePath`

text). At the start of the statechart engine, the non-predetermined-path activation process is applied to the outermost state.

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

Once all states of a configuration have been swept across and the control flow goes back to the program, this 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 numerical order of the state identifiers, the firing of transitions is performed in a deterministic way. Thus, alternative numerical identifiers assignments 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.

9 Concluding remarks

Due to the hierarchical nature of statecharts, 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 done in specific contexts without having to pay greater attention on what is happening in terms of context swappings which are handled by the engine. This focus concentration makes the design task easier for applications with complex behaviour. The proposed technique also reduces the reactive kernel implementation to a simple translation from a specification into two tables.

The way how a complex behaviour specification in terms of a statechart could be implemented has been described. The present work reflects an evolution from others proposals [2, 7, 23]. There are some restrictions imposed by the engine architecture which inhibit the implementation of the whole statechart functionality. Nevertheless the restricted behaviour implementation is believed to be useful and a new version with additional functionality is now in progress. This new version removes as well the naive function calls to application code, and introduces a queue of output tokens and another for input events. In this approach the engine can be executed concurrently with other modules of the system, working as a "behaviour server." These and other changes resulted from our experience of using statechart in the computer-human interface domain [20].

Care must be taken since the tables for the engine are built manually. A proper environment [8], 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 implemented quite trivially provided that the proposed approach is adopted. Complexity does not simply disappear, but it is transferred to the engine. Changes of behaviour require changes in only two tables and can even be made at execution 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 reactive program: the reactive kernel.

References

[1] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

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

[3] John Carroll and Darrell Long. *Theory of Finite Automata*. Prentice Hall, Inc., 1989.

[4] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing ObjectCharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

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

[6] Doron Drusinsky and David Harel. On the Power of Cooperative Concurrence. *LNCS*, 335, 1988.

[7] Ted Faison. Object-Oriented State Machines. *Software Development*, 1(3):37–50, September 1993.

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

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

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

[11] 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.

[12] D. Harel and A. Pnueli. On the Development of Reactive Systems. Technical Report, The Weizmann Institute of Science — Department of Applied Mathematics, Rehovot 76100, Israel, 1985.

[13] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. *Proceedings of 2nd. IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[14] David Harel and Chaim-Arie Kahana. On Statecharts with Overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, October 1992.

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

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

[17] C. Huizing and W. P. de Roever. Introduction to Design Choices in the Semantics of Statecharts. *Information Processing Letters*, 37:205–213, 1991.

[18] C. Huizing, R. Gerth, and W. P. de Roever. Modelling Statecharts Behaviour in a Fully Abstract Way. In *Lecture Notes in Computer Science*, pages 271–294. Springer-Verlag, 1988. Number 299.

[19] Y. Kesten and A. Pnueli. Timed and Hybrid Statecharts and their Textual Representation. *Lecture Notes in Computer Science*, 571:591–620, 1992.

[20] Fábio N. Lucena and Hans K.E. Liesenberg. Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour. Technical Report DCC-20/93, DCC/IMECC/UNICAMP, 1993.

[21] Bonnie E. Melhart, Nancy G. Levison, and Matthew S. Jaffe. Analysis Capabilities for Requirements Specified in Statecharts. *ACM SIGSOT Engineering Notes*, 14(3):100–102, May 1989.

[22] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Lecture Notes in Computer Science*, pages 244–264. Springer-Verlag, 1991. Number 526.

[23] Bob Rodini. Crafting WinApps with STD. *Computer Language*, 7(3):45–50, March 1990.

[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[25] 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.

[26] Pierre D. Wellner. Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *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*

28/93 **Programming Dialogue Control of User Interfaces Using Statecharts,** *Fábio Nogueira de Lucena e Hans Liesenberg*

29/93 EGOLib – **Manual de Referência,** *Eduardo Aguiar Patrocínio e Pedro Jussieu de Rezende*

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

reltec@dcc.unicamp.br