STATECHART VISUAL FORMALISM AS THE EXECUTION LANGUAGE FOR CONTROL OF REAL-TIME SYSTEMS

by

GREGORY BRYON STORM, B.S.C.S.

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

May, 1995

ACKNOWLEDGMENTS

I would like to thank Dr. William M. Marcy for his help and support throughout my graduate study. I also want to thank my mother and my wife whose encouragement have been instrumental to me in this and many other endeavors. I thank FSI International for supporting this research. Thanks above all to God, who has given me my mind, my life, and my joy.

This work is dedicated in memory of my father, Ellery Storm Ph.D., who would have considered this degree a stepping stone toward completing my education. I wish he were alive to know I've gotten this far.

TABLE OF CONTENTS

ABSTRACT

The Statechart visual formalism is an extension to the graphical notation of Finite State Machines. Statecharts have found use in the areas of specification, design, and prototyping of complex systems. This research extends the application of Statecharts by developing both a database capable of holding the Statechart's graphical information, and an execution engine to execute the data. The result is the foundation for a graphical programming environment for real-time systems, and the programming language is the Statechart formalism. The execution engine is generic with respect to the data; therefore, the underlying execution engine remains virtually the same regardless of the nature of the real-world object represented by the original Statechart.

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Research Question

Can the Statechart visual formalism be extended beyond specification, design, and prototyping complex systems to be the execution language for control of real-time systems?

Research Objective

As software increases in complexity, it is becoming the central technological challenge of our time. Processors are getting faster and more powerful, memory is getting bigger and cheaper, and communications speeds are increasing. All of these are making applications, and therefore software, larger, more complex and more difficult to develop. Applications are exceeding our capability to design and implement them promptly and reliably [Williams 90].

Real-time systems have an even greater complexity. These systems face unforgiving requirements, including nonstop reliability, and they face time constraints, where providing even the right answer late is wrong.

The future of computing is graphical. Computer hardware advances, cheaper memory, etc., have made graphics an ever increasing part of computing in general and system development in particular. Complex systems require that humans comprehend them, and humans, in general, comprehend more graphically in a shorter amount of time. For data-oriented applications, graphical and iconic languages are providing a solution to this growing complexity. In putting forth a solution for real-time systems, this research will determine the feasibility of extending the Statechart formalism, which is a Finite State Machine (FSM) based graphical language, beyond the current uses of specification, design, and prototyping real-time systems into the use of Statecharts as the execution language for real-time systems.

Background

A reactive system is characterized as being event driven, continuously having to react to external and internal stimuli in its environment. The system's behavior is the set of allowed sequences of input and output events, conditions, and actions with additional information such as timing constraints. In this definition, real-time systems are reactive systems. The complexity of these systems results from the intricate nature of reactions to discrete events. Literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification, design, and implementation of large complex reactive systems [Harel 87].

Finite State Machines (FSM) and state transition tables have been widely used in computer and engineering applications. They model system behavior and have a formal definition as a derivative of finite state automata. FSMs are suitable for representing the design of a reactive system, for FSMs are beneficial for simplifying the behavioral model of a reactive system. A reactive system responds to events and data in its environment, and the basic fragments of a reactive system description are "when event μ occurs in state A, if condition C is true at the time, the system transfers to state B" [Harel 87, p. 232]. FSMs and state-transition tables are the mechanism for collecting these fragments into a whole.

FSMs provide a graphical view of a system. The main components of a FSM are states (circles) and transitions (arrows). Special instances of the state are the Initial State, and Terminal State. Events are the inputs to the system, and actions are taken in response to the events. The actions specify outputs affecting system behavior, which in turn drives the events.

The two most prominent models of FSMs are the Mealy model and the Moore model. Mealy FSMs attach actions to transitions, whereas actions are attached to states in the Moore model. In either model, transition to the next state is a function of the current state and the event. The past state is not relevant once the new state is reached; response can then be considered present tense.

The state transition table is a translation of the FSM, as it specifies, in row/column fashion, the transition and possible actions for every state and event. The state transition table is preferred over FSMs as the number of states and transitions grows, for the FSM diagram becomes unstructured and loses the ability to meaningfully convey information. State transition tables also suffer from complexity and comprehension problems as the number of states and transitions grow. It is generally agreed that complex reactive systems cannot be beneficially described by FSMs.

Statecharts are a broad extension to FSMs. They extend FSMs in three areas: hierarchy, concurrency, and communication. Used either as a stand-alone behavioral description or as part of a more general design methodology, Statecharts combine the benefits of graphical representation while retaining a formal description. This combination of features is termed "visual formalism" [Harel 87].

Statecharts are not the only type of visual formalism. Another, a combination of Harel Statecharts and the Moore model of FSMs, has been used within standards for the semiconductor industry by SEMI (Semiconductor Equipment and Materials Institute). Hardware advances allow adaptation of representations now being used as presentation mediums to function in the visual programming environment [Jacob 85]. BACCII [Calloni 93] is one such example. Furthermore, empirical observation showed programmers preferred FSMs to Backus-Naur Form front ends as a specification interpreter [Guest 82].

Statecharts, as a visual formalism, are both graphical and formal, providing visual information, not quantitative, of a structural, set-theoretical, and relational nature. Statecharts, furthermore, are a diagrammatic paradigm, topological in nature, not geometric, for shapes, locations, distances, and sizes have no significance. The combination of structural and relational qualities is termed "topovisual" [Harel 88]. The two best known topovisuals are: graphs and Euler circles (Venn diagrams), which originated in the 1700s, and it is in these topovisuals that Statecharts derive their origin.

Statecharts have been used to specify, design, and prototype complex, or reactive, systems. Statecharts move the designer from requirements and specification, through prototyping and design, and form the basis for maintenance and modifications [Harel 90]. Statecharts have even been used in object oriented analysis settings, for they have been used for specification of the behavior of object classes [Coleman 92].

Analysis tools are wide spread. Even though commercial tools for real-time systems have sophisticated editors, these tools do little with the description of the system. They do not provide a "programming environment" with compiler, debugger, etc. The designer often has to decide between describing behavior in ways clear and intuitive or ways precise and rigorous. Many excellent methods exist for transformational (data processing) systems, which allow decomposition of the system into ever-smaller parts. These systems tend to be both intuitive and rigorous. This is not the case for reactive systems tools.

Statecharts are filling the need for tools for reactive systems that are clear and intuitive (graphical) and precise and rigorous (formal). As a formal tool, Statecharts are amenable to validation, simulation, and analysis by computers (execution). Clarity and intuition, furthermore, are greatly enhanced by the visual nature of the Statechart language, which is amenable to generation, inspection, and modification by humans.

Problems arising in the design of a reactive system are far more difficult than those arising in a data-processing program [Harel 90]. Reactive systems are highly concurrent and distributed, consisting of multiple levels of detail. They also display unpredictable, often catastrophic, behavior under unanticipated circumstances. Systems readily understood by non-software engineers, and those which conform to formal methods, will have greater reliability in industry and arrive there more quickly than alternate development approaches. Communication to a human audience with diverse backgrounds, via its graphical nature, gives Statecharts great power for use with reactive systems. Engineers of every discipline, and software developers as well, can quickly understand a system expressed in the form of a Statechart. Based on the diagram and current-state, the system can answer "What can I do next?", "Where am I?", or "How can I do activity x?".

But the advantage afforded by visual formalisms, such as Statecharts, is currently limited, because, when system design ends and system implementation begins, the Statechart must be translated by a human into computer readable form. A significantly improved development process would permit building a system where the design language, the visual formalism, was also the implementation language. Engineers of varied disciplines could understand and modify the design until it was completed, and the design could then be directly executed by the computer without human translation.

Iconic and visual representations, including FSMs, have long been employed to communicate with people, but using visual languages to communicate directly to a computer is in its infancy. Attempts to use visual formalisms to directly describe algorithms to computers are increasing, and these visual models eliminate the need to convert algorithms to the linear strings of symbols traditionally required by most computers. Computer language designers now face the challenge of providing convenient and natural visual programming languages.

The aim of this research is to determine if the Statechart visual formalism is capable of use as a programming language. If the Statechart visual formalism can be executed in its graphical form, the basis for a completely integrated real-time system development tool would exist. Such a tool would contain analysis features common today, but it would contain additional features for compiling the Statechart within an integrated development environment. The tool would also execute the data in simulation or actual execution mode. This research will determine if the Statechart formalism is suitable for use as the foundation for such a tool.

CHAPTER II

LITERATURE REVIEW

Graphical Languages

Graphics facilities are improving. Languages, which encourage visual modes of thinking when tackling systems of ever-greater complexity, will not merely be iconic, but inherently diagrammatic in concept. These languages can exploit and extend the human visual system [Jacob 85].

The intricate nature of a variety of computer-related systems and situations can be represented by visual formalisms. The graphical nature of a visual formalism allows for generation, comprehension, and communication to and from humans. Furthermore, visual formalisms provide manipulation, maintenance, and analysis by computers. Clarity and rigor are provided due to their highly visual nature. A considerable amount of mileage is obtained from such formalisms with a small number of simple diagrammatic notions that are topological, not geometric, in nature, because the small number of carefully chosen diagrammatic paradigms admit formal semantics, providing each feature with a precise and unambiguous meaning.

Visual languages can be thought of in one of two categories [Jacob 85]. In the first category, the object being described is itself a graphical object. These might take the form of a menu, screen layout, engineering drawing, typeset report, or font of type.

Such objects are candidates for direct manipulation, or what might be called "what you see is what you get" programming.

A programming environment supporting this first category of graphical languages need only simulate the appearance of the object and provide direct commands for manipulating it. With respect to the time spent describing a system in this type of language, such systems realize great power from visual communication with the human visual system. But this power is only realized where there is a one-to-one correspondence between a visual programming language and the static visual object.

The second category of graphical languages, which includes Statecharts and BACCII [Calloni 93], represents something that is abstract, such as programming constructs, a time sequence, a hierarchy, conditional statements, or frame-based knowledge. This type of language requires first a suitable graphical representation or visual metaphor. Success of the language depends critically on choosing a good representation. In the absence of an applicable theory of graphical communication, proper use of such representations often requires extensive experimentation.

Visual Formalisms

Two of the best known topo-visual formalisms have their roots in the work of the Swiss mathematician Leonhard Euler (1707-1783) [Harel 88]. The first is graphs and the second is the notion of Euler circles, which later evolved into Venn diagrams.

A graph is simply a set of vertices connected by edges. The graph represents a single set of elements S and some binary relation R on them; restrictions on R yield connected, directed, acyclic, planar, or bipartite graphs. Hypergraphs are an extension of graphs that are less widely used. They represent a graph in which the relation being specified is not necessarily binary, and edges no longer connect a pair of nodes but rather a subset thereof. Information conveyed by graph or hypergraph is nonmetric and captured by the purely topological notion of connectedness. Shapes, locations, distances, and sizes have no significance.

Euler circles are used to represent logical propositions, color charts, etc. They appeal to the 2-D case of the Jordan curve theorem, which establishes that simple closed curves partition the plane into disjoint inside and outside regions. Figure 2.1 shows the plane containing the closed curve A. The plane is divided into the two regions A and $\overline{A}$ (the area outside the closed curve).

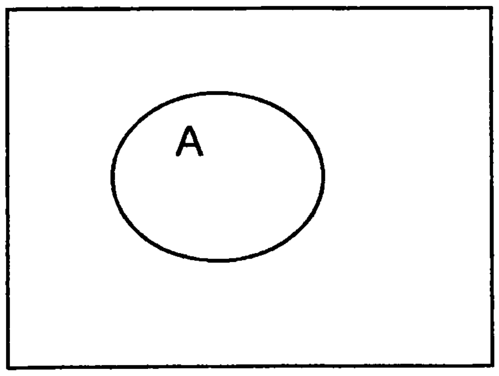


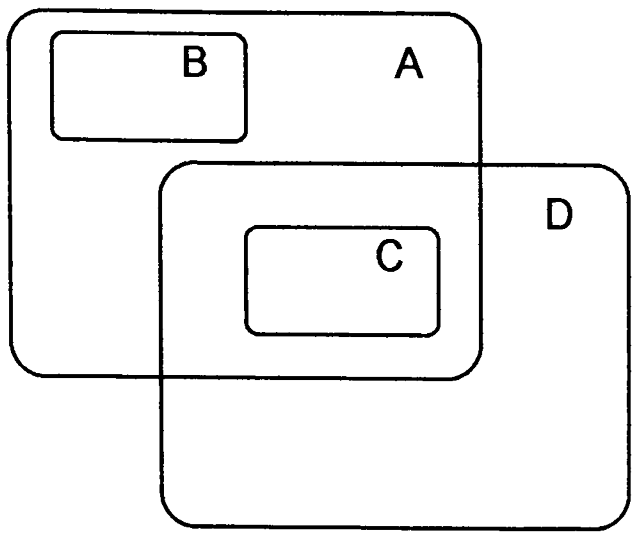
Figure 2.1. Jordan Curve Theorem.

Whereas graphs and Hypergraphs are adequate for representing a set of elements together with some special relation(s) on them, Euler/Venn diagrams are adequate for representing a collection of sets, together with some structural relationships between them. In numerous computer-related applications, the complexity of the objects, systems, or situations under consideration is due in large part to the fact that both capabilities are needed [Harel 88].

There are a large number of sets interrelated in nontrivial, set-theoretic ways, but also related via one or more additional relationships of special nature, depending on the application at hand. It is desirable to identify the Cartesian product of these sets. This can be crucial in preventing certain kinds of representations from growing exponentially in size. In [Harel 88], Euler's two topo-visual formalisms have been extended and combined. The Euler diagrams are extended to represent the Cartesian product, and the resulting curves are connected by edges, which [Harel 88] terms "Higraphs."

Higraphs

Higraphs, resulting from the extensions mentioned above, are rectangles that represent set inclusion, as opposed to set membership. Every set of interest is represented by a unique blob. This is a departure from Euler circles, and the reason for departure is to provide every set with its own area. For example, take two intersecting blobs A and D as shown in Figure 2.2.



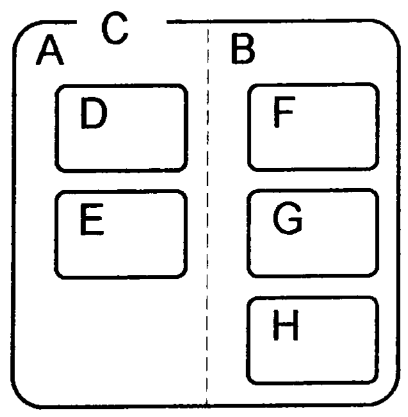
the intersection, $A \cap D$ is C

the difference, A - D, is B

Figure 2.2. Example of a Higraph.

The only identifiable sets are the atomic sets--those represented by blobs residing on the bottom levels of the diagram, containing no wholly enclosed blobs within. Any nonatomic blob denotes the compound set consisting of the union of all sets represented by blobs that are totally enclosed within it. Intersecting blobs mean nothing unless blobs appear within the intersection. Empty space represents nothing.

The Cartesian product is denoted by a blob with dashed lines. Figure 2.3 shows the blob C with orthogonal components A and B. The Cartesian product is unordered, so C is a set of unordered pairs of elements.



$C = A \times B = (D \cup E) \times (F \cup G \cup H)$

$A \times B = B \times A$

Figure 2.3. Example of Higraph Cartesian Product.

Due to the set inclusion property (versus set membership), the product, C, is associative. In this way, if d $\in$ D, g $\in$ G, h $\in$ H, then the unordered triple {d, g, h} would be a legal element of set C, without the need to distinguish it from {d, {g, h}}. We can assume that all atomic sets are pairwise disjoint (no element appears in any two sets), and each component has a unique unambiguous area.

A Higraph is obtained by adding edges, or more generally, hyperedges, to the contour of any blob. Hyperedges connect blobs between any levels. In the set-theoretic spirit, edges simply associate the target blob with the source blob via the particular relationship represented by the edge.

The most beneficial application of Higraphs is extending state-transition diagrams to yield Statecharts.

Statecharts

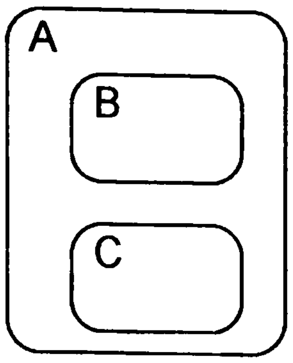
Harel [88] notes a common theme in literature of the existence of a major problem in the specification and design of large and complex reactive systems. The problem is rooted in the difficulty of describing reactive behavior in clear and realistic ways that are at the same time formal and rigorous. Among the solutions to the problem that have been put forth are: Petri nets, communicating sequential processing (CSP), calculus of communicating systems (CCS), sequence diagrams, ESTEREL, temporal logic, and Statecharts.

Statecharts are a Higraph-based extension of standard state-transition diagrams, which represent a broad extension to the conventional formalism of state machines. The extension falls into three categories: hierarchy (depth), concurrency (orthogonality), and communication. Statecharts transform the language of state diagrams into a modular, highly structured, and economical description language [Harel 87]. Conventional state diagrams are extended by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components. Statecharts cater to clustering states into super-states, into orthogonality and independence of states, into more general transitions than the single event-labeled arrow context switch, and into refinement of states. In short:

Statecharts = state diagrams + depth + orthogonality +

broadcast communication.

In the Statechart formalism, rectangles represent states, and arrows represent transitions. Statecharts capture depth and hierarchy with the insideness of states. Statecharts are topological, not geometric, in nature; locations, distances, and sizes have no significance. Take, for example, a super-state A with sub-states B and C, as shown in Figure 2.4.



XOR Relationship:

$A = B \otimes C$

Figure 2.4. Super-states and Sub-states in a Statechart.

Being in state A is being in either B or C but not both.

Another feature of Statecharts is labeled the default state. The default state is analogous to start states of finite-state automata. A default state is represented by a small arrow without an origination state. Figure 2.5 shows an example of super-state A, which contains the sub-state B as the initial state for A.

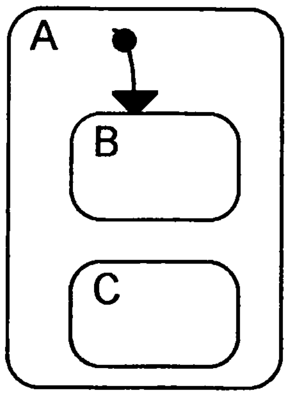


Figure 2.5. Initial State of a Statechart.

Statecharts further modify FSMs to provide a history capability. History affects the entrance of the Statechart into a super-state, and the history defines that the "most recently visited sub-state is entered when entering a super-state" annotated with the history designation. Adding the designation to super-state A above yields Figure 2.6.

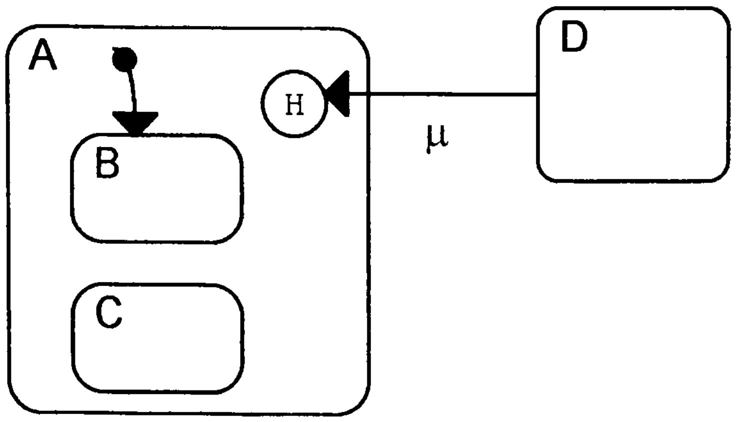


Figure 2.6. History Designation in a Statechart.

In Figure 2.6, a transition from state D to super-state A would cause the most recently visited sub-state, between states B and C, to be entered. If there were no history for state A, if this was the first transition into state A, the initial sub-state, state B, would be entered. The history feature allows Statecharts to achieve effects between one level and all level extremes when transitioning into a state.

As an example of clustering, consider the following conventional FSM shown in Figure 2.7.

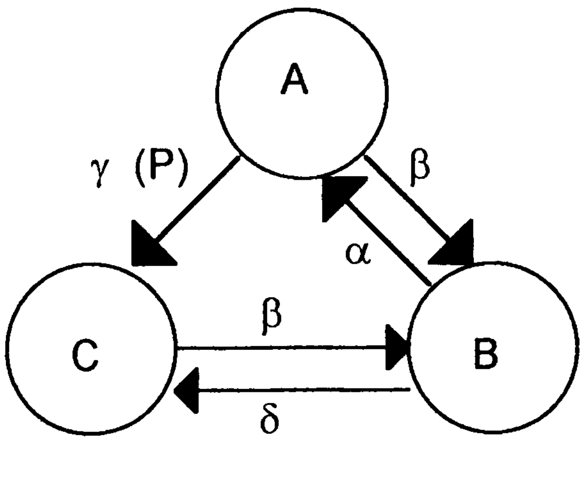


Figure 2.7. A Conventional State Machine

Statecharts cluster states A and C in a super-state named D, and this allows the common transition β to be represented as a single transition, see Figure 2.8.

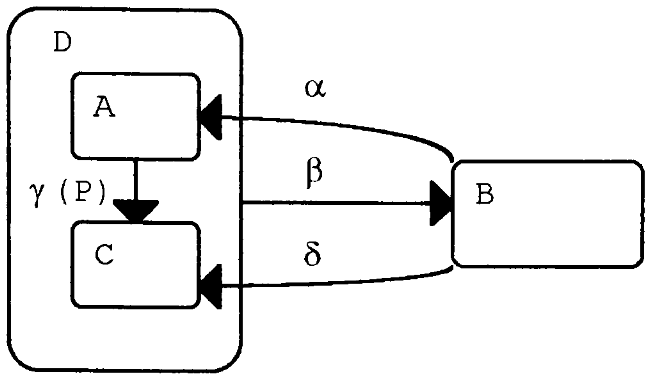


Figure 2.8. A Statechart Demonstrating Clustering.

In capturing the common property, β, the relationship between states A and C is an XOR decomposition, for the system described can be in state A or C but not both. It is said then, that D = A ⊗ C. Any transition leaving the super-state stands for all sub-states.

Abstraction, as seen above, can take another form in Statecharts. Above, the common transition β, which left both states A and C, was combined to leave the super-state D at the boundary of state D. If, however, a transition fails to leave all sub-states within a super-state, say transition ε, this combination cannot be made. When the sub-states are hidden in the abstraction of fine detail, the transition β is shown on the boundary, but the transition ε is shown to originate in the interior of the super state. These examples are illustrated by Figure 2.9.

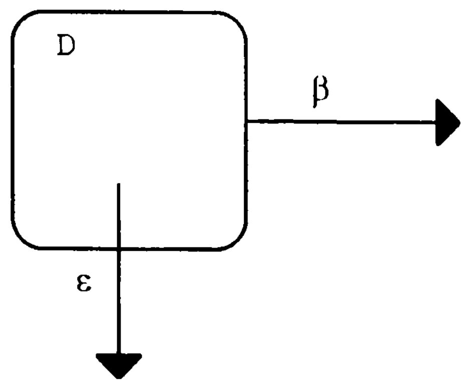


Figure 2.9. Transitions Leaving a Super-state

Time is available through a global clock. Predicates on time can be used for firing conditions on events. For example, an event might translate to "after two minutes in state D, leave state D and transition to state E."

Real-time applications are making more than simple demands for processor speed. They are also demanding more operating system services than are provided by the real-time executives traditionally used for single-board controllers [Williams 90]. Furthermore, in both single board and multiprocessor applications alike, discrete events are happening in parallel, and the modeling system must represent this parallelism (or orthogonality). In Statecharts, orthogonality is the dual of the XOR decomposition of states which equals an AND decomposition. This is the Cartesian Product of Highraphs.

Figure 2.10 shows a state C consisting of AND components A and B. Being in state C, the system must be in all of the AND components. C is the orthogonal product of A and B.

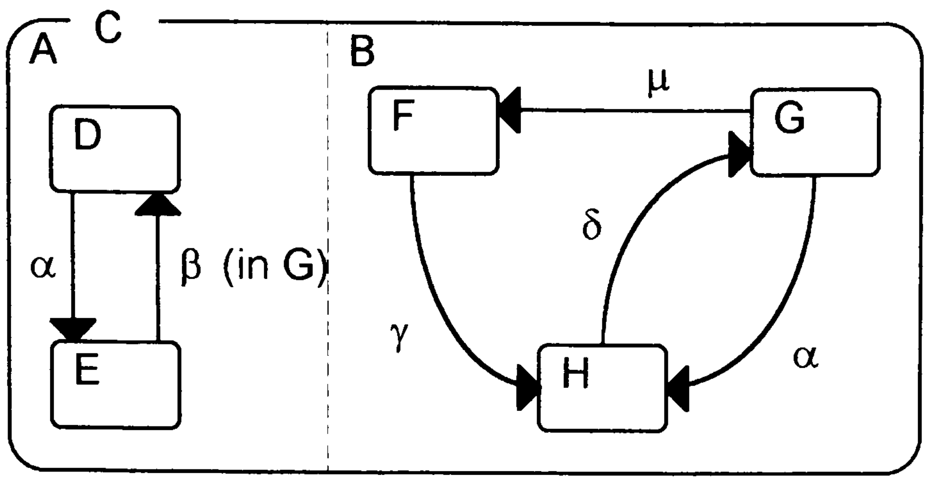


Figure 2.10. Orthogonal Product of states A and B.

If C is in the combined state (G, D), and event α occurs, it transfers G to H and D to E simultaneously, resulting in the new combined state (H, E). If, on the other hand, event μ occurs at state (G, D), it affects only the B component, resulting in the combined state (D, F). If orthogonality is used often and on many levels, the state explosion problem and sequentiality problems are overcome in a reasonable way [Harel 88]. Orthogonal components can synchronize through common events.

Statecharts are viewed as Mealy machines, for actions can be attached optionally to the triggering event along a transition. But unlike Mealy, an action is not merely sent to the "outside" world, it affects behavior of the Statechart in orthogonal components. This broadcast mechanism causes transitions in all relevant components.

Statecharts versus Finite State Machines

Harel [87] points out that in order to be useful, a state/event approach must be modular, hierarchical, well-structured, and solve the exponential blow-up problem of conventional FSMs. Relaxation of the requirement that all combinations of states must be represented explicitly, allowing arrows to originate and terminate at any level are two needs. Statecharts do this.

Furthermore, Statecharts counter many of the objections raised against conventional state diagrams. Among the reasons finite state machines do not work for complex systems are their flat nature. FSMs do not provide a natural notion of depth, hierarchy, or modularity. This leaves FSMs unable to support bottom-up, top-down, and stepwise refinement, which are key elements of functional software development. Furthermore, FSMs are uneconomical in regard to transitions. An event that causes the same transition from a large number of states must be attached to each state separately.

FSMs are further uneconomical when it comes to states. As a system, being described by the FSM formalism, grows linearly, the number of states grows exponentially. FSMs force explicit representation for each state. Another drawback to FSMs are their inherent sequentiality. There is no method for representing concurrency.

These four elements leave FSMs inappropriate for use in specification and design of complex reactive systems. The formal nature of a FSM, however, is desirable.

There have been other attempts to extend FSMs. These have been ineffective in reducing the size of the description. Communicating State Machines, for example, allow only a single set of communicating machines on the highest level. Other attempts at FSM extension are not diagrammatic in nature, and therefore do not cater to the human visual system.

STATEMATE®: A Working Environment for the Development of Reactive Systems

STATEMATE® is a set of tools, with a heavy graphical orientation, intended for the specification, analysis, design, and documentation of large and complex reactive systems, such as real-time embedded systems [Harel 90].

A reactive system's behavior is not adequately described by specifying the output that results from a set of inputs, rather, the description requires specifying the relationship of inputs and outputs over time. Such descriptions involve complex sequences of events, actions, conditions and information flow, often with explicit timing constraints, that combine to form the overall behavior of the reactive system.

The problem of finding good methods to aid in development has not been solved by research or industry. Structured analysis and design do not adequately deal with the dynamics of reactive systems, since they were designed to deal with non-reactive data-driven applications. In such data-driven applications, functional decomposition is sufficient.

Many commercial tools for real-time system design are sophisticated graphics editors, which model aspects of reactive systems. In this manner, they help to organize thoughts for the designer. But they do little with the resulting descriptions beyond testing them for syntactic consistency and completeness. These tools are inadequate when it comes to preparing reliable specifications and designs that satisfy the requirements, and they are further inadequate in providing construction of a final system.

There is a great need for what is analogous to a programming environment for reactive systems. It must contain a powerful language with a useful editor and syntax checker. There must be capabilities which correspond to compiling or interpreting, and it must further have extensive debugging capabilities. STATEMATE attempts to fill this need with the underlying premise of specification and analysis of the system under development (SUD) from three points of view: the structural view, the functional view, and the behavioral view. The conceptual model is the combination of the functional and behavioral views.

The structural view provides hierarchical decomposition into physical components (modules). This view identifies the information that flows between modules.

The conceptual model consists of a hierarchy of activities, complete with the details of the data items and control signals flowing between modules, and control activities. The control activities specify behavior.

In the functional view, activity hierarchy and flow information are portrayed, and this is the functional decomposition of the SUD. Dynamics are not specified in the functional view, for it does not say when and why the activities are activated. It does not say whether they terminate on their own, and it does not comment on parallel activities. The functional view specifies only that data can flow, not whether and when it will.

The behavioral view specifies the third view of the SUD. This view specifies the control activities. These are present on any level of the activity hierarchy controlling that particular level. These controllers are responsible for specifying when, how, and why things happen. A controller can: start and stop activities, generate new events, change variable values, sense whether activities are active/inactive, respond to events, and test the values of conditions and variables.

Each view has its own diagrammatic language. The language for the structural view is module-charts. The language for the functional view is activity-charts, and the language for the behavioral view is Statecharts.

STATEMATE has analysis capabilities. These enable the user to run, debug and analyze the specifications and designs that result from the graphical languages. A database is constructed to rigorously execute the specification and retrieve a variety of kinds of information for the user.

A number of tools support the analysis capabilities. Among these capabilities are: (1) a means of querying the database to retrieve information, (2) an execution capability with simulation control language that provides interactive or batch execution and breakpoints, (3) dynamic tests for reachability, deadlock, and nondeterminism, and (4) translation of the specification into a high-level language.

STATEMATE has been under development since 1984 and commercially available since 1987. It runs on Sun, Apollo, and VAX workstations. It has been field tested in a number of large real world development projects, including the mission-specific avionics system for the Lavi fighter aircraft designed by the Israel Aircraft Industries.

Objectcharts: Using Statecharts in Object-oriented design

Objectcharts specify the behavior of object classes as state machines, effectively combining object-oriented analysis and design techniques with Harel's Statecharts. The typical first step in object-oriented analysis is to use entity-relationship diagrams to build information models. Subsequent steps include specification of class behavior. This specification is often accomplished by extending state machines with imperative actions.

Objectcharts offer a formal and declarative extension to state machines, for conventional state machines are more abstract and therefore not appropriate for analysis and design. Objectcharts characterize behavior of an object class as a state machine. There is an input alphabet comprising the operations that the class provides, and there is an output alphabet comprising the operations the class requires.

Objectcharts capture the instances of objects and their intercommunication within a system by means of a configuration diagram. The configuration diagram shows the instances of objects in a system and their intercommunication. Objects are represented by boxes containing their instance identifiers and the class to which they belong, solid lines represent provided services, and dashed lines represent required services.

The provided services of an object are the set of services which can be used by other objects in the environment. Required services are those which this object needs from other objects.

Statecharts can capture some aspects of class behavior, and Objectcharts extend Statecharts by augmenting states with attribute information. The effect of state transitions on attributes are specified.

Objects have a life cycle in which they change state as a result of providing services for clients and requiring services from other objects. The life cycle behavior of an object class may be expressed as a Statechart. The states in the Statechart represent the various stages that an object of a class may go through. The transitions are labeled with either state changing services provided by the class or services required of other objects.

Not all services change the state of an object, as some just report on attribute values. Services which report in this manner are Observers.

At a system behavior level, the Objectcharts relate system behavior to the behavior of component objects. There are two steps: first, the behaviors of individual objects are defined, and then these are combined to define the behavior of the overall system.

Since Objectcharts specify class behavior, they can be used to determine whether a proposed inheritance conforms to subtyping. An inheritance is a subtype inheritance if and only if the specification of the parent holds for the descendant. A class inherits by either extending the parent or specializing the parent. The definition of a subtype for Objectcharts is that the descendant class may specialize through: (1) adding an extra state or transition, and (2) strengthening a transition by weakening the firing condition or strengthening post-conditions. The restriction on redefining transitions ensures that a modified transition fires in at least as many situations as the original one.

The procedure for developing Objectcharts begins by drawing object configuration diagrams. The procedure continues by drawing a Statechart and identifying state changing and observing services for each significant class. The Statechart should show the life cycle of state change and required services. For such classes, the Objectchart results by annotating states with observers and attributes, defining firing post condition specifications, and defining invariant specifications for the derived observers and attributes. The procedure concludes by drawing the inheritance tree.

CHAPTER III

DESCRIPTION OF THE RESEARCH

This research successfully extended the Statechart visual formalism beyond previous uses of specification, design, and prototyping complex systems, and it demonstrated that the formalism is capable of use as the programming language for control of real-time systems.

There are several aspects to the research. The initial phase of the research developed a relational database which captured the Statechart diagram in raw data form; the second phase of the research resulted in an execution engine, named the Statechart Executive, which extracts the data from the database and executes it.

The result of the research is a prototype upon which a completely integrated real-time system development and execution environment could be built. The prototype was built using Microsoft® Access™ and Microsoft® Visual Basic™. The remainder of this section contains a discussion of both the database and its design, and of the application and its algorithm.

The Database

A relational database is best suited for storing the Statechart in a form internal to the computer, because the Statechart expresses relations graphically and hierarchically. A flat file representation would be awkward in capturing the visual and relational nature of a Statechart diagram. Additionally, the commercial database application performs many database management system (DBMS) functions, such as query retrieval and file handling. In a flat file representation, complex modifications to the data would be difficult to maintain. By using a commercial database package, the Statechart Executive is free from file details when retrieving information from the database.

Relational database theory treats data as relations and describes how data should be structured and managed [Brackett 87]. The relational model is a way of looking at data; it is a representation of data and data interrelations describing the real-world where the data originates.

Structurally, the relational model is a collection of relations, and a relation is a two-dimensional table consisting of rows and columns of data. A relation, commonly called a table, is synonymous with a flat file representation of data. A row is a line of data in a table, and each row is unique among all other rows and represents a real-world occurrence of a relation. A column represents a specific characteristic of the data.

Each table contains a primary key, and the primary key is an attribute uniquely identifying each row of a table. The primary key is a column of data characterizing the row of data, but it is a column with a special meaning. The key is used to uniquely identify a row among all other rows in a table, and it can be composed of a single column, or multiple columns from the same table.

Tables are interrelated by relationships with other tables. The relationships can be described as one-to-one, one-to-many, many-to-one, or many-to-many depending on the nature of the data. The relationship designates a logical access path between tables.

The Statechart database was designed to be true to the graphical representation of the data. The major components of the Statecharts, states or transitions for example, became the design basis of the database, and there are few aspects to the database design which do not directly relate to the graphical Statechart formalism. Furthermore, the database design lends itself to recreating the Statechart diagram from the data within the database.

Though the database design preserves the graphical nature of the data, the database also presents the data to the execution engine. Balance exists between retaining the graphical nature of the data and between storing the data in a form amenable to execution. The executable form of the data maximizes the inter-relations of the data. For example, all sub-states for a particular state must be retrieved in a timely manner and organized efficiently to be parsed into data structures internal to the Statechart Executive. The database design serves to aid the Statechart Executive in collecting the Statechart data.

The database was normalized in a traditional manner for relational data, and the resulting database is in Third Normal Form (3NF). Normalization theory defines how data tables are formed to capture their inherent meaning, and it defines how data tables are identified to avoid update, insertion, and deletion anomalies. Anomalies occur when there are redundant attributes in the database.

Redundant attributes exist, for example, when an attribute describing a parent table is included in each occurrence of a child table. When an update to the parent attribute is needed, it must be made in each occurrence of the child table rather than just once to the parent table. This not only causes unnecessary processing, but can also result in conflicting parent attributes if one of the child tables is not updated. Normalization is assurance that there is no redundancy of attributes and no anomalies within the Statechart database. Redundancy is limited to primary keys.

The normalization of the Statechart database was accomplished in a series of steps, and the starting point was the non-normalized initial form of the data. The next three forms of the data, First, Second, and Third Normal Form, resolved functional dependencies dealing with single-valued attributes.

A relation, or database table, is in First Normal Form (1NF) if it contains no repeating groups. Repeating groups in the non-normalized data are identified and moved to a new table, and a relation is defined between the original table and the new table.

A relation is in Second Normal Form (2NF) if it is in 1NF and every non-key attribute is fully functionally dependent upon the primary key. A 1NF relation will be in 2NF if one of the following is true:

(1) the primary key is composed of only one attribute,

(2) no non-key attribute exists, or

(3) every non-key attribute is dependent on the entire set of primary key attributes [Brathwaite 88].

Third Normal Form is achieved by removing interattribute dependencies. A relation is in 3NF if it is in 2NF and if no attribute is functionally dependent on another attribute in the same table. If the functional dependency exits, the dependent attribute must be removed and placed in another table [Brackett 87].

The normalization of the Statechart database proceeded as described in the following discussion, which uses the State Table as an example. The initial form of the State Table is listed in Figure 3.1. The information required for a given state includes the identification number, the name, a description, etc. Other necessary information includes the list of all sub-states for this state and the list of transitions which leave this state.

State Name and Identification Number (ID)
State Description
Super State Identification Number (ID)
List of Sub State Identification Numbers (IDs)
Default Sub State Identification Number (ID)
List of Transition Identification Numbers (IDs)
List of Activity Identification Numbers (Ids)
State Type
State History
State Time Out Length

Figure 3.1. The Non-Normalized State Table.

Figure 3.1 is not the exhaustive listing of a state's attributes; the exhaustive and normalized list is found in the Database Dictionary of the Appendix. Figure 3.1 above does not meet the criteria for 1NF, because there are multiple entries in the sub-state, activity, and transition fields. When the repeating groups are removed from the State Table and placed in separate tables, The State and Sub State Table, The State and Activity Table, and the State and Transition Table, the information, as listed below in Figure 3.2, is in 1NF. The key for each table is listed in bold type.

The State and Sub State Table contains one entry for each sub-state of a given state. As shown in Figure 3.3, the state identification number can therefore be listed multiple times in the table. There is no redundancy, because the key for the table is made up of both the State ID attribute and the Sub State ID attribute. The two fields together are required to uniquely identify a given row in the table. Data is stored in a similar manner for each of the State and Transition Table, and the State and Activity Table.

The State Table and its supporting tables are in 2NF. For example, the State Table is in 2NF, because each non-key attribute is dependent on the primary key, which is the State ID. This means that each non-key attribute can be determined by knowing only the State ID key. Furthermore, each of the supporting tables are in 2NF, because there are no non-key attributes in those tables.

State ID
State Name
State Description
Super State ID
Default Sub State ID
State Type
State History
State Time Out Length

Figure 3.2a. The First Normal Form State Table.

State ID
Sub State ID

Figure 3.2b. The First Normal Form State and Sub State Table.

State ID
Transition ID

Figure 3.2c. The First Normal Form State and Transition Table.

State ID
Activity ID

Figure 3.2d. The First Normal Form State and Activity Table.

Figure 3.2. The State Information in First Normal Form.

State ID	Sub State ID
0	1
0	2
2	3
2	4
2	5
2	6
2	7
2	8
3	9
3	10

Figure 3.3. Data in the State and Sub State Table.

The tables listed above in Figure 3.2 are also in 3NF, because there are no attributes in the tables that are functionally dependent on another attribute in the same table.

As shown below in Figure 3.4, the normalized State Table itself does not list the sub-states for a particular state, for the sub-states are in a separate database table. A query, invoked by the Statechart Executive, retrieves all of the information pertaining to a particular state.

Database queries, like those used by the Statechart Executive, are a read-only retrieval operation where the data in the database is not modified. The query specifies the results desired without providing the access paths to the data. The DBMS determines how to access the data [Gardarin 89].

State Name and ID
State Description
Super State ID
Default Sub State ID
State Type
State History
State Time Out Length

Figure 3.4. The State Table.

The queries built within the Statechart database provided the mechanism of retrieving the data and placing it within the Statechart Executive's data structures. The queries, which are listed in their entirety in the Appendix, link each related table into a single operation, and the entire data for the states, including sub-state and transition information, can be retrieved together.

The relational database provides the functionality to link the State Table, the State and Sub State Table, and the State and Transition Table. The Statechart Executive needs only to know the format of the query, and the state data can be read in and placed within the Statechart Executive's data structures. Parsing was required to retrieve the state information from the query, for there is an entry in the query result for each sub-state, transition, and activity combination entered in the database. If for example, a state has three sub-states, two transitions, and four activities, then the query will retrieve 3 x 2 x 4 = 24 entries for that particular state.

State ID	Name	History	Time Out	Super State	Default Sub State	Type	Sub State ID	Transition ID
0	Quartz Multi-alarm	0	0	-1	1	0	1	
0	Quartz Multi-alarm	0	0	-1	1	0	2	
1	dead	0	0	0	-1	0		0
2	alive	0	0	0	-1	1	3	1
2	alive	0	0	0	-1	1	4	1
2	alive	0	0	0	-1	1	5	1
2	alive	0	0	0	-1	1	6	1
2	alive	0	0	0	-1	1	7	1
2	alive	0	0	0	-1	1	8	1

Figure 3.5. The Get All States Query.

As listed in Figure 3.5, state 0 has two entries in the Get All States query. In each of the columns except the "Sub State ID" column, the state's data is repeated. Separate entries are required for each of the two sub-states of state 0 listed in the "Sub State ID" column. State 0 has no transitions, and none are listed under the "Transition ID" column. The -1 entry in state 0's "Super State ID" identifies that state 0 does not have a super state. A blank entry was used for a blank value in a non-required field, and -1 was used for blank entries in required fields. This notation is used throughout the database.

State 1, the dead state, has only a single entry in the Get All States query, because state 1 has zero sub-states and only a single transition which leaves the state.

State 2, shown in Figure 3.5, has six entries in the query. State 2 has six sub-states and only one transition. The design of the Get All States query results in n entries per state, where n = (number of sub-states) * (number of transitions), but at least one entry will exist for each state even if the state has zero transitions and zero sub-states.

The complete database, including other queries, is documented in the Appendix. The queries used by the Statechart Executive are: Get All Conditions, Get All States, Get all Transitions, and Get All Variables. All the queries yield results similar to those shown above for the Get All States query.

Once the database was normalized, the data was entered into the database. Since the scope of this research focused on retrieving and executing the data, the data was manually translated from the Statechart diagram and entered manually into the database. Chapter V includes discussion of future work allowing the data to enter the database as it is developed graphically.

The Statechart Executive

As stated above, this research assumed the existence of a data entry mechanism, and the focus of the research was developing a database to store the Statechart data, retrieving the data from the database, and executing the data. The algorithm and execution engine is named the Statechart Executive, and the following paragraphs describe its development.

As shown in Figure 3.6, the Statechart Executive consists of an execution engine and a graphical user interface. The execution engine interacts with the database to read the

Statechart data, and the engine performs any input and output associated with executing the Statechart. The GUI displays the Statechart data on the Cathode Ray Tube monitor (CRT) as it is executed.

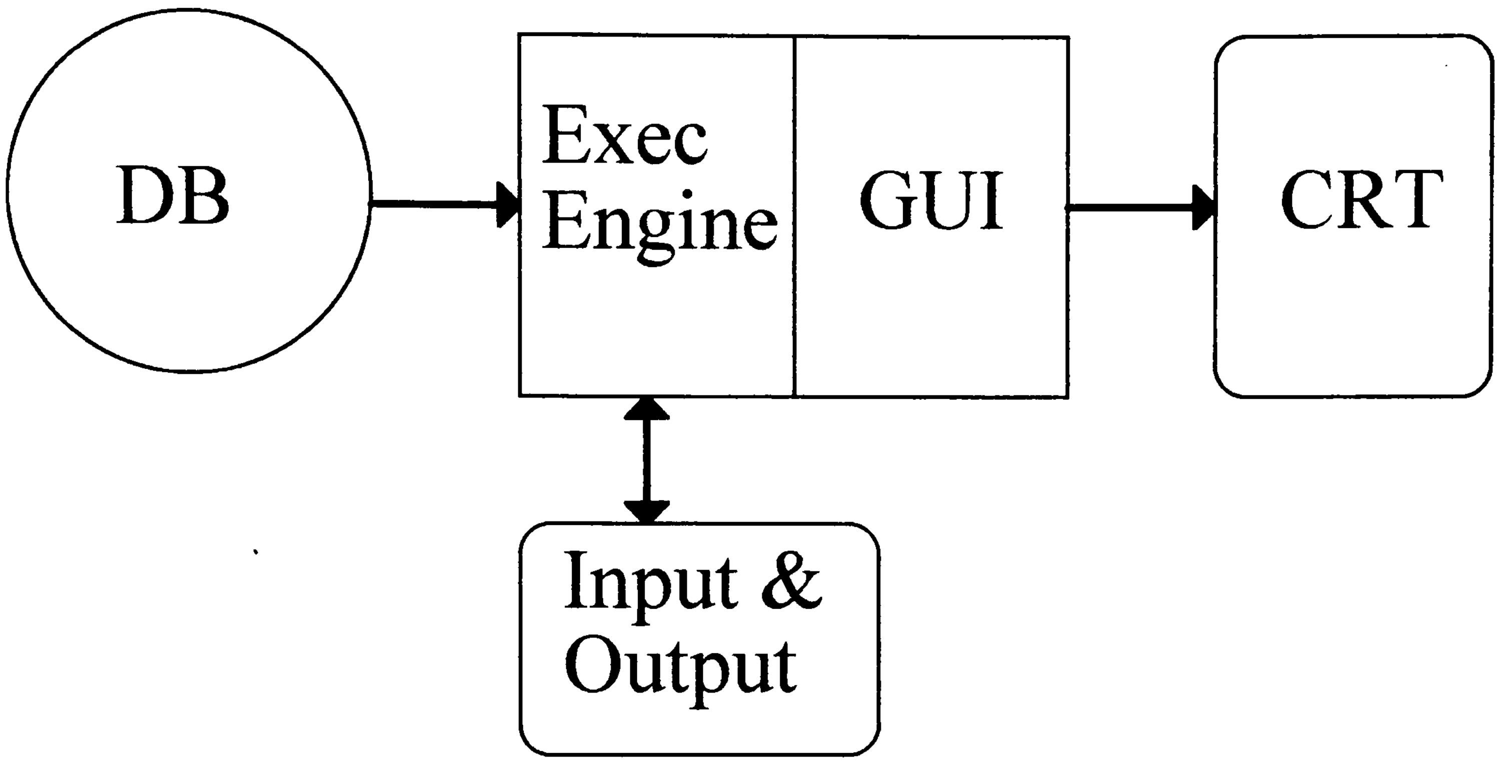


Figure 3.6. Block Diagram of the Statechart Executive.

The execution flow of the application is listed in Figure 3.7. First, the data is loaded into the database, and state 0 is set as the Initial State of the Statechart. When execution is enabled, the Statechart Executive begins execution. State 0 is entered, becoming active, and execution proceeds according to the rules of the formalism.

During execution, the Statechart Executive makes continuous passes through the list of states, and each state is checked during the Statechart Executive's pass. If a state is found active during a pass, then the state's activities are processed.

Activities are outputs from the system, and they correspond to the behavior of a state. Activities take some non-zero amount of time, and there are three varieties of activities: entry, exit, and throughout. Entry activities are performed when the state is entered, exit activities when the state is exited, and throughout activities each pass when the state is active. They modify the system and affect future inputs, and this gives Statecharts the characteristic of Moore FSMs.

Following any activities the state may have, the active state's transitions are checked. If a transition is found whose events and conditions evaluate as true, the transition is taken by exiting the origination state of the transition and entering the destination state of the transition. The state that is exited is marked inactive, and the state entered is marked active.

A transition's events input data into the Statechart, and the data can come from the actual real-world system or from the database. An event might determine if a particular value is open, for example, or it might check the value of internal data, such as a timer.

Conditions augment events, and they place circumstances on when an event causes the transition to be taken.

Events and conditions are very similar, and they are both Boolean, true or false, evaluations. A transition is only made when the event of the transition evaluates as true and the condition, if any, also evaluates as true.

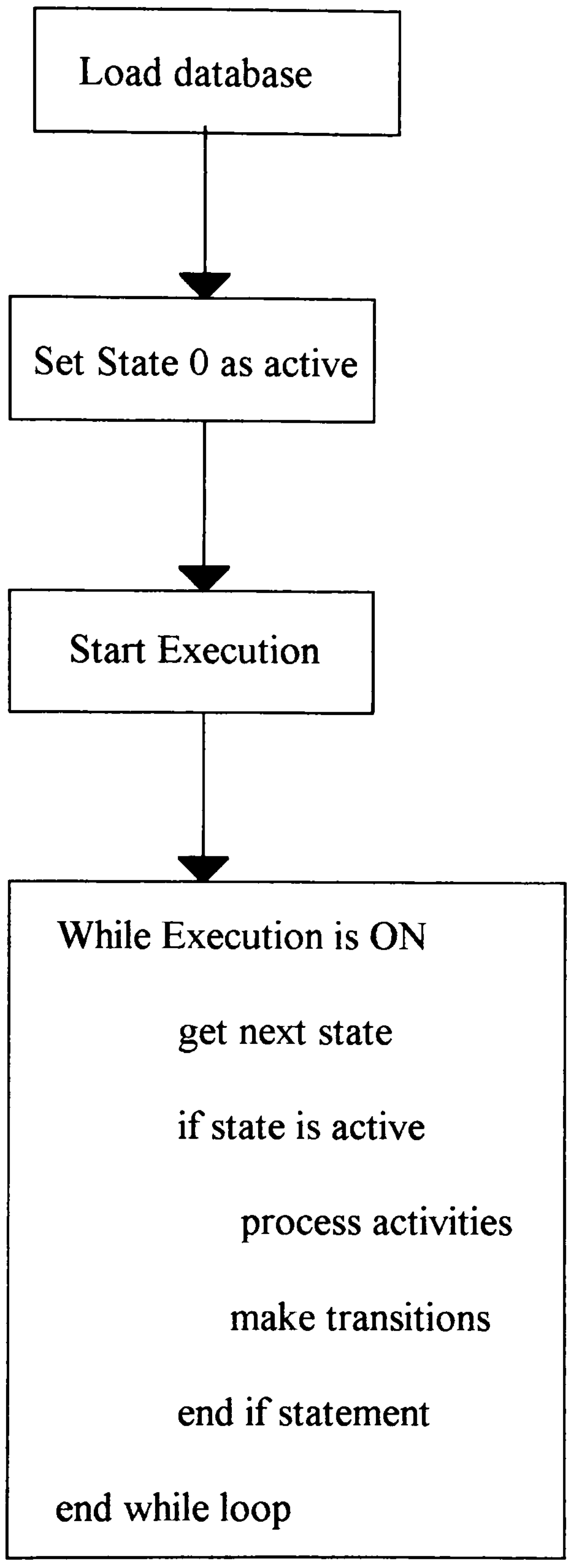


Figure 3.7. The Flow of the Statechart Executive.

If the transition has any actions associated with it, the actions are performed when the transition is made. Actions, in the Statechart formalism, are outputs from the system and similar to the activities of states, but actions give Statecharts the characteristic of Mealy FSMs. Actions are distinct from the activities of states, because actions are instantaneous operations.

The input and output (I/O) drivers are the core of the Statechart Executive's execution loop, for the drivers perform both the inputs and the outputs. Inputs to the system come in the form of events and conditions on transitions, and outputs are the actions of transitions and activities of states. The I/O drivers do not contain logic or control information pertaining to the Statechart; rather, the I/O drivers exist as tools for units of work performed by the Statechart itself. The Statechart dictates what outputs to set, and the output drivers respond to the control of the Statechart. In a similar manner, the Statechart reads the inputs to decide what transitions, if any, are necessary. The list of input and output drivers used in this research's case study are shown in Figure 3.8.

Each input and output driver has attributes associated with it that relate to hardware devices. The address for each driver corresponds to the hardware port address that would be required to access any type of hardware card, such as a digital or analog input card. Additionally, since these types of cards have multiple channels associated with them, each driver has a "Number of Channels" attribute. In the event that these attributes are not required by the driver, the attributes are ignored.

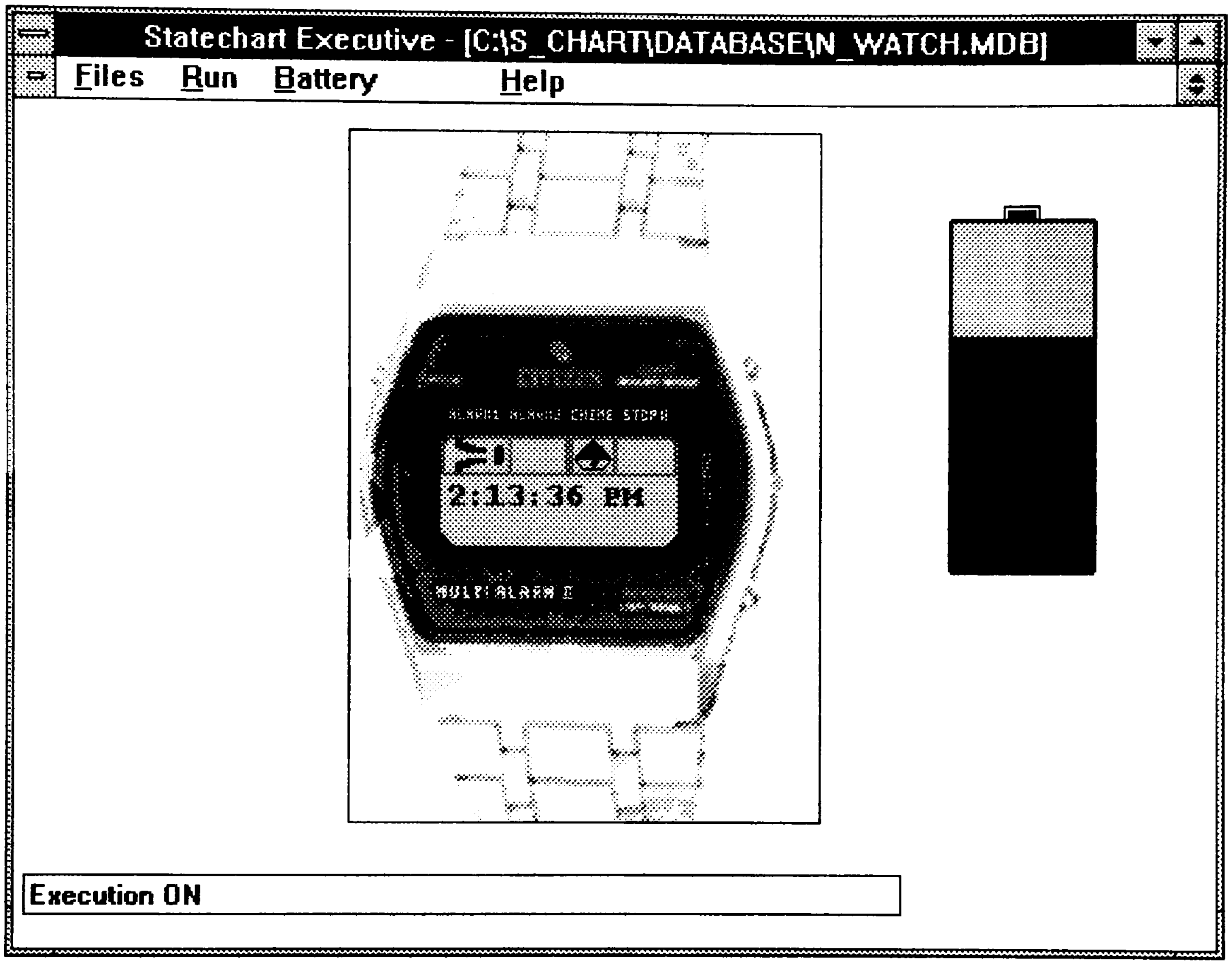
Driver ID	Name	Description	Address	Number of Channels
0	keyboard	input: the keyboard	0	0
1	timeout	input: timeout of current state	0	0
2	beep	output: beeping sound	0	0
3	light	output: light the display	0	0
5	alarm1	output: alarm1 sound, set, and enable	0	0
6	alarm2	output: alarm2 sound, set, and enable	0	0
7	time	input: current time	0	0
8	time	output: current time	0	0
9	date	input: current date	0	0
10	date	output: current date	0	0
11	battery	input: battery status	0	0
12	clear history	output: clear state's history	0	0
13	chime	input: chime status	0	0
14	chime	output: turn chime on or off	0	0
15	alarm1 status	input: alarm1 status	0	0
16	alarm2 status	input: alarm2 status	0	0
17	stopwatch	input: stopwatch status	0	0
18	stopwatch	output: turn stopwatch on or off	0	0
19	display mode	output: display OK of blinking	0	0
20	whole hour	input: is time whole hour	0	0
21	In State?	input: in a particular state?	0	0
22	changes?	input: changes made while in a state	0	0
23	reset timeout	output: reset timeout for a state	0	0
24	key status	input: status of a key	0	0
25	toggle time	output: toggles hour, min., or sec	0	0
26	toggle date	output: toggles Mon., day, or year	0	0
27	toggle alarm1	output: toggle alarm1 hour and min.	0	0
28	toggle alarm2	output: toggle alarm2 hour and min.	0	0

Figure 3.8. The List of Input and Output Drivers.

Microsoft® Visual Basic™ 3.0 Professional provided an environment where the concept of the Statechart Executive could be prototyped. The resulting application is discussed in further detail in the following section, but Figure 3.9 shows the Statechart Executive executing the wristwatch Statechart.

While the Statechart Executive itself is intended to be generic with respect to the data it executes, the GUI must be specific to the data. The case study for this research executes the Statechart of a wrist watch, and the GUI for that example, shown in Figure 3.9, below, includes a bitmap of a wristwatch, a bitmap of a battery (created with Paint

Brush™), and a text information box. All these graphics objects are specific to the example, and would change as the Statechart data changed.



Execution ON

Figure 3.9. The Statechart Executive Executing the Watch Statechart.

Visual Basic's greatest strength is providing easy access to standard Windows features. The file menu, title bar, and other common windows features are provided with little programming effort.

The professional version of Visual Basic™ was chosen to develop the Statechart Executive, for it provides libraries which perform file open operations via the traditional

File Open dialog box. Here again, the programmer is required to perform very few operations to include this feature in the application. Though the underlying development tool based on this research would be programmed in C++, these capabilities of Visual Basic™ make it a strong candidate for GUI development for systems controlled by the Statechart database and Statechart Executive in their final form.

The bitmap used in the case study described in the following section was scanned from literature obtained from Nobila Timepieces. The image was scanned into a bitmap with the equipment of and the help from graduate students in the Vision and Robotics Lab of Texas Tech University.

CHAPTER IV

CASE STUDY

The case study for this research is taken from [Harel 87], where the author developed a Statechart for the Citizen Quartz Multi-Alarm III wrist watch. The watch, shown in Figure 4.1, has a main display area (MAIN) and four smaller ones (1, 2, 3, and 4), a two-tone beeper, and four control buttons denoted here A, B, C, and D. It displays the time or the date, it has a chime, which beeps on the hour if enabled, two independent alarms, a stopwatch, a light, and a beeping test.

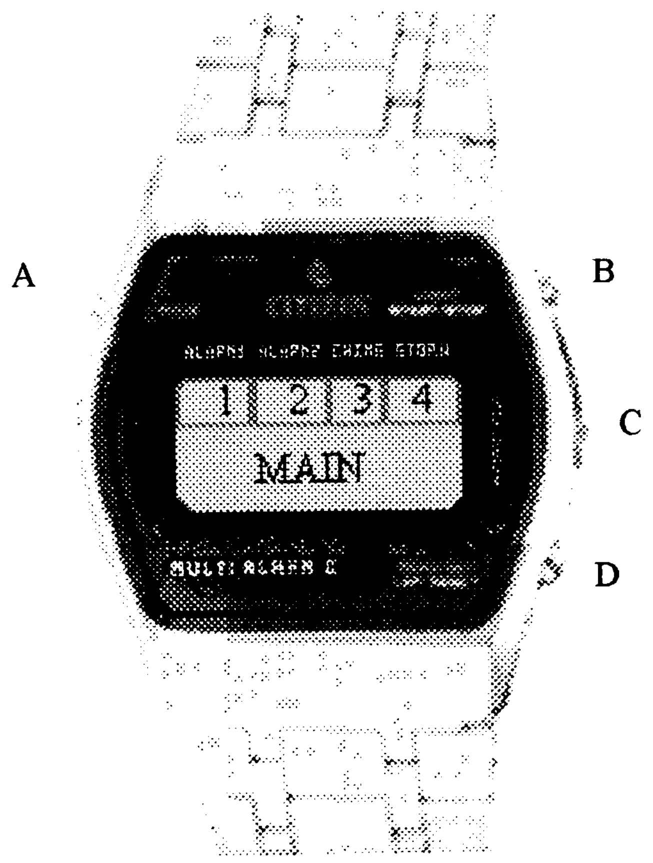


Figure 4.1. The Citizen Quartz Multi-Alarm III Wristwatch.

The external events, or inputs into the watch, are (1) depressing and releasing any of the control buttons and (2) inserting and removing the battery. The outputs are the display and sounds of the watch. The Statechart describing the watch's behavior was developed in [Harel 87], and is shown in Figure 4.2. It is intended to be as faithful to the actual watch as possible.

The Statechart for the watch was given in [Harel 87] does not exhaustively describe all of the watch's behavior. Several sub-states were added during this research which accommodate full functionality of the watch. Those added states are not shown in Figure 4.2, but they were added to the sub-states, shown in Figure 4.2, which update time, date, and alarm times.

When the data representing the Statechart is loaded into the Statechart Executive, state 0 is the Initial State. Once execution begins, state 0 is entered, and the default sub-state of state 0, state 1, is also entered.

As shown in Figure 4.3, there is only one transition out of state 1, and it is a transition to the alive state, state 2. State 2, in turn, has two transitions out, both of which enter the dead state, state 1. Figure 4.3 demonstrates an example of the hierarchical nature of Statecharts, for all of the detail of state 2 can be abstracted to remove the detail from view. One transition, the "weak battery dies" transition, emerges from the interior of state 2, because the transition actually originates in a sub-state of state 2.

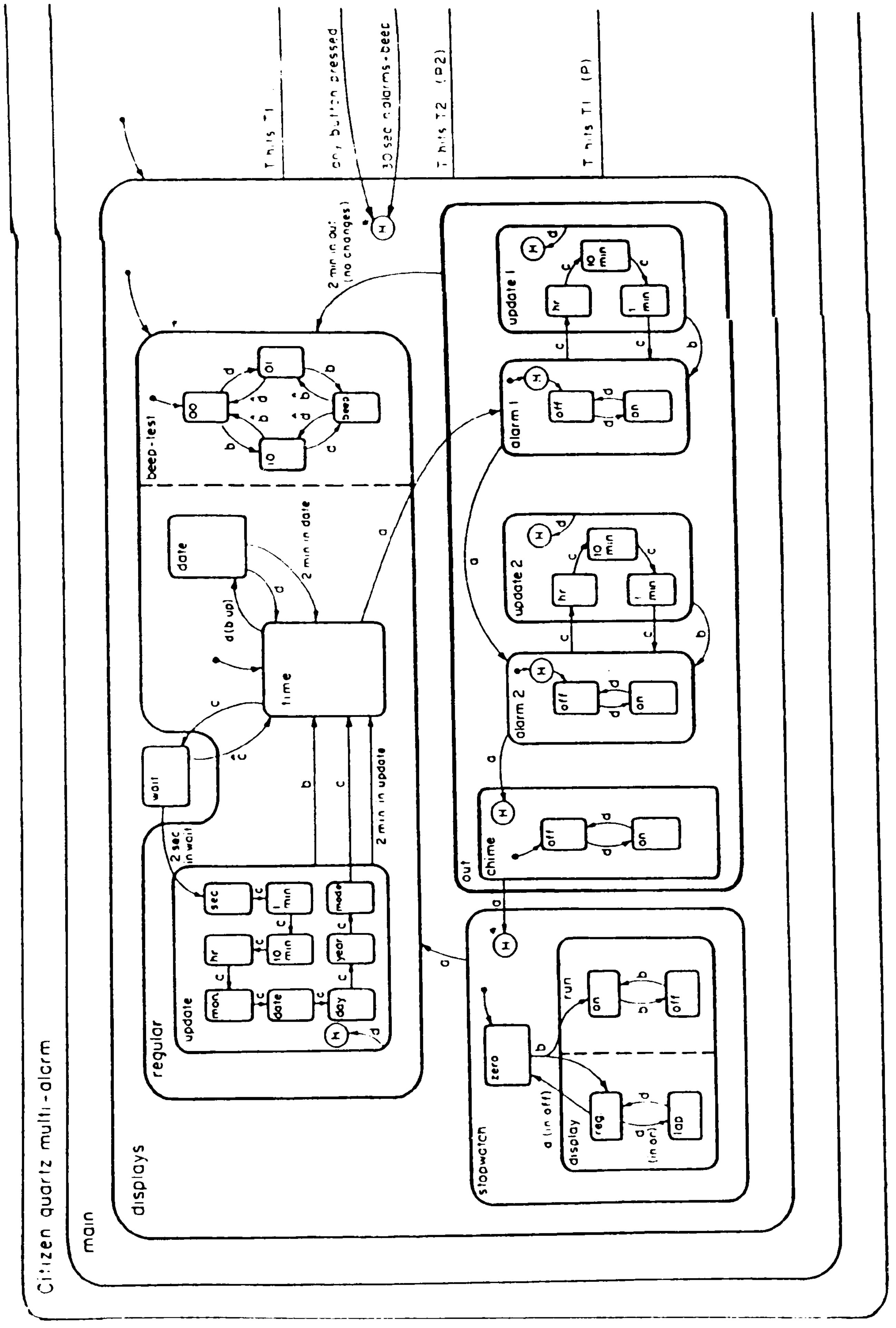


Figure 4.2. Statechart of the Quartz Wrist Watch.

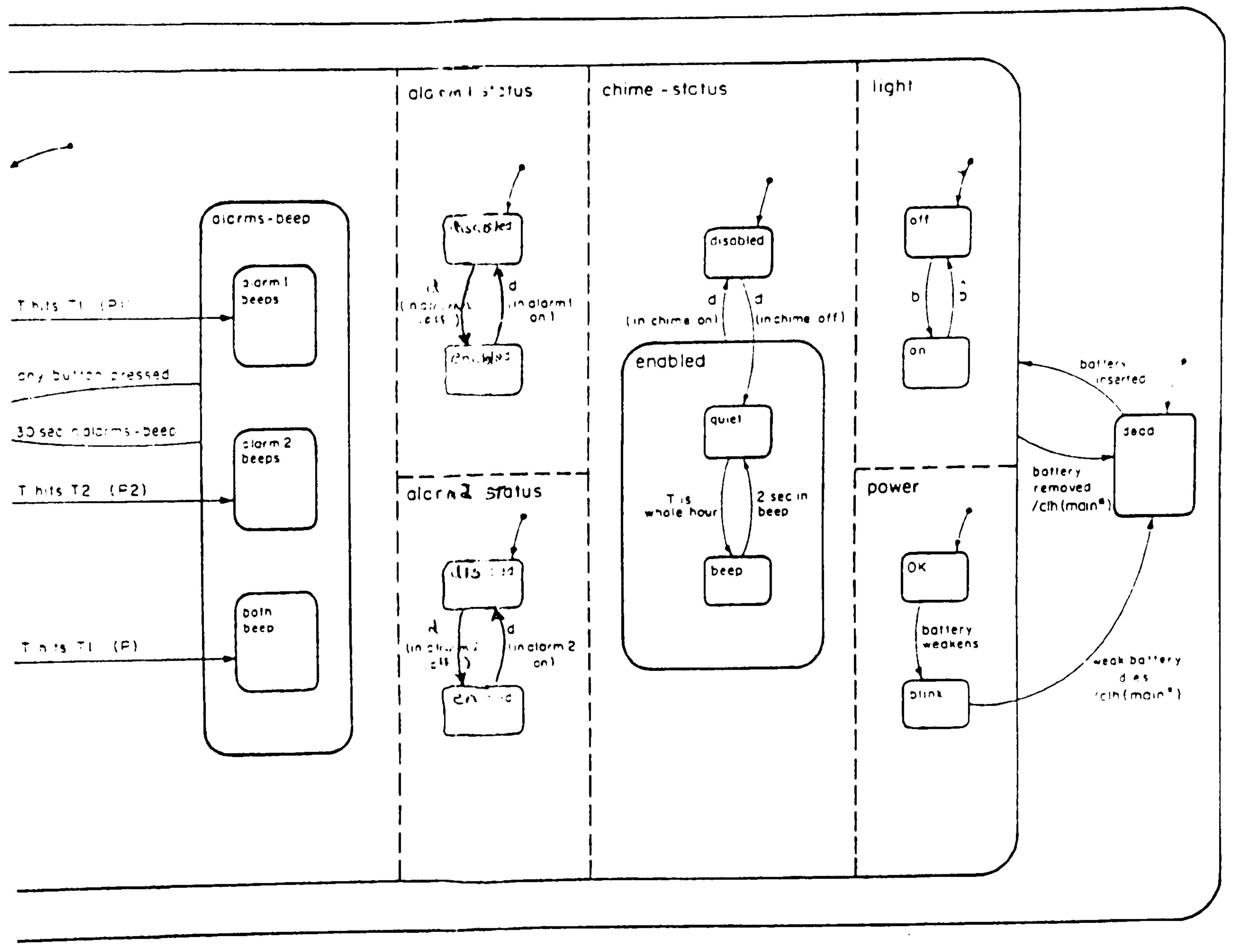


Figure 4.2. Continued.

The Statechart Executive will remain executing only states 0 and 1, neither of which have activities, until the battery is inserted. Referring to Figure 3.9, the battery is inserted via the Battery menu item on the Statechart Executive's menu bar.

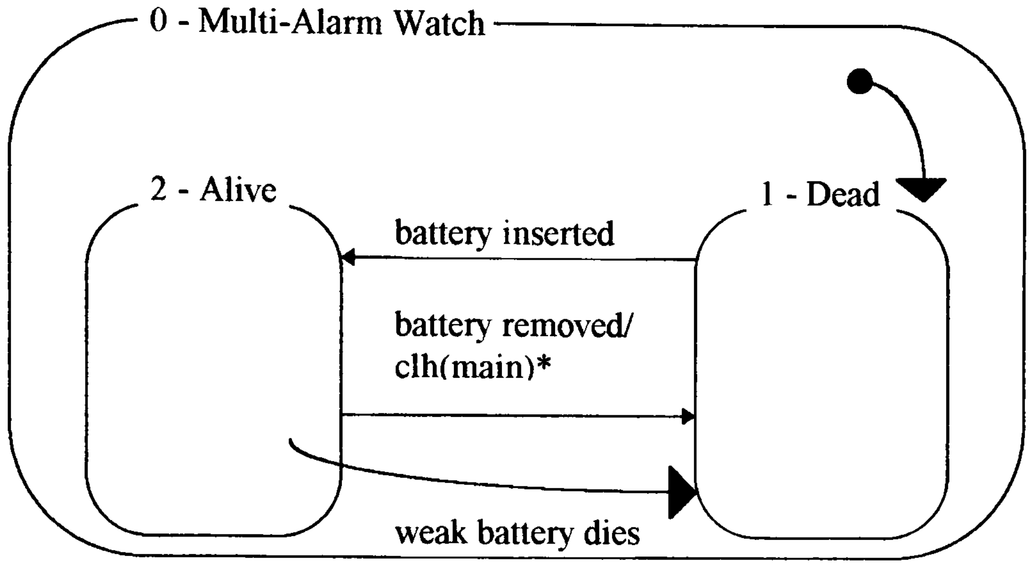


Figure 4.3. Statechart of States 0, 1, and 2.

Inserting the battery causes the Statechart to transition from state 1 to state 2 via the "battery inserted" transition. State 1 is exited and set inactive; state 2 is entered and set active. Many of the Statechart's properties are visible as state 2, shown in Figure 4.4, is entered. State 2 is an AND state, which causes the Statechart Executive to execute in parallel each of state 2's sub-states. The six sub-states of state 2 encompass the simultaneous activities of the wrist watch, and those activities are: displaying information on the watch face (state 3), enabling and disabling the alarms (states 4 and 5), enabling and disabling the chime (state 6), turning the light on and off (state 7), and monitoring for the battery to weaken (state 8).

Another property of Statecharts, seen when state 2 is entered, is the chain reaction of entering state 2's sub-states. States 3, 4, 5, 6, 7, and 8 are entered, because those states are the AND component sub-states of the alive state. The chain reaction continues in each of these sub-states, because they in turn have sub-states. For example, state 9, the displays state shown in Figure 4.5, is the default sub-state of state 3, and state 9 is entered when state 3 is entered. State 21 is the default sub-state of state 9, and it is entered when state 9 is entered. When the chain reaction of entering state 2 is finished, the complete list of states that will be set active is: 0, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17, 19, 21, 29, 31, 40, and 42.

As mentioned above, the parallel execution characteristic of Statecharts is seen as the Statechart Executive executes the AND component sub-states of state 2, which are states 3, 4, 5, 6, 7, and 8. These sub-states are separated by dotted lines in Figure 4.4. The algorithm is a round robin execution, where each state and its active sub-states are executed in turn. States are executed the same number of times, for there is no weighting or priority of active states within the Statechart formalism.

The most complex of state 2's sub-states is state 3, the state named main. State 9, the displays state, is the default sub-state for state 3, and state 9 is entered when state 3 is entered. State 9 has several sub-states, and the default sub-state, state 21, is itself an AND state. The displays state is shown in Figure 4.5.

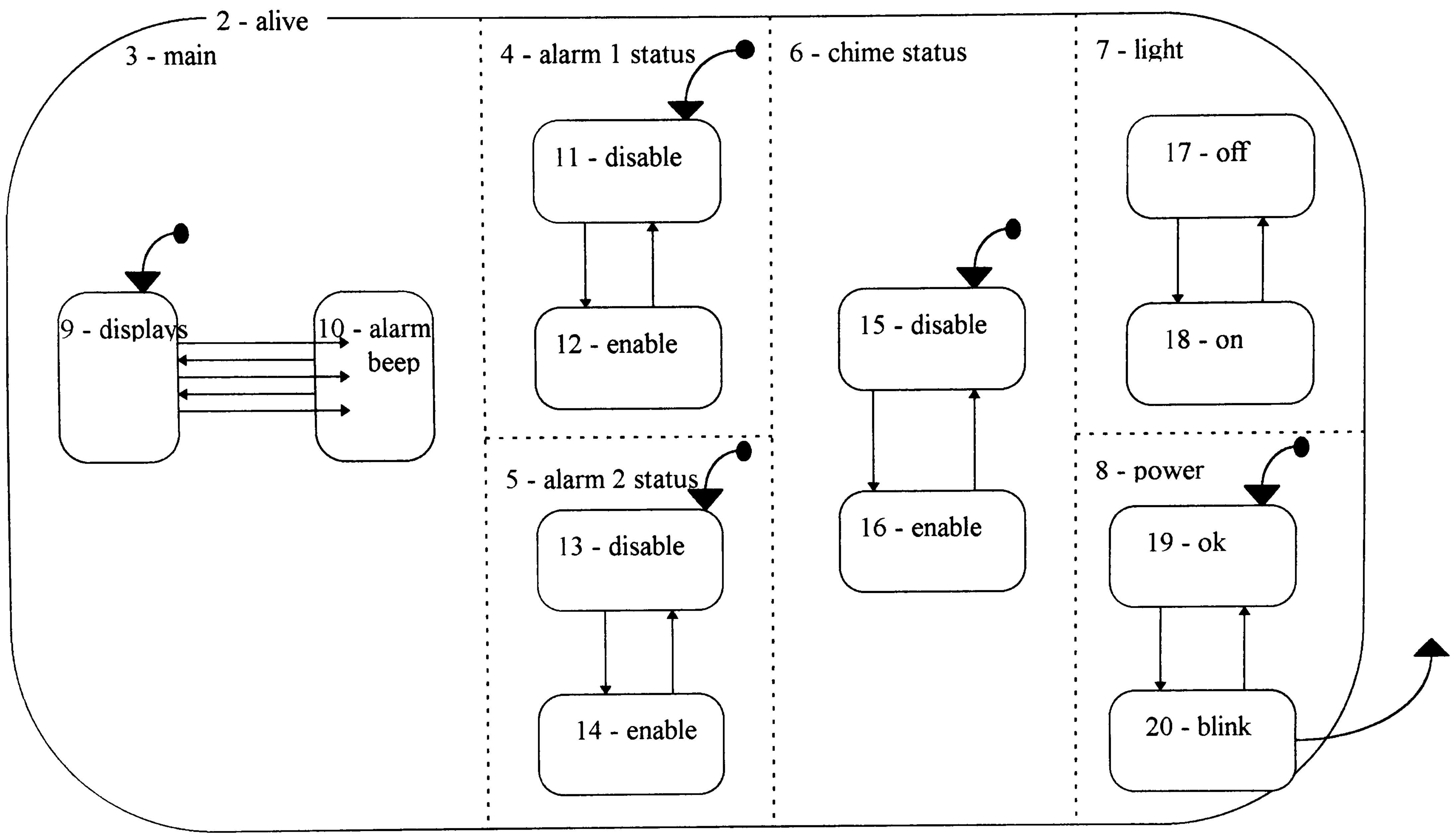


Figure 4.4. Statechart of the Alive State (State 2).

The displays state is the core of the watch's execution mode. The watch will be displaying the time via an activity, while in state 40, and waiting for the control button sequences leading to the beep test. The a, b, c, and d buttons cause transitions out of the time state, but these transitions lead back to the time state after completing operations such as setting the alarm times or performing stopwatch functions.

All of the watch's functionality is described within state 2. The Statechart Executive remains in state 2 until the battery is removed or dies, and the battery is removed from the Statechart Executive's menu bar.

The two transitions leaving state 2 which enter state 1 illustrate further properties of Statecharts. The "battery removed" transition leaving the boundary of state 2, see Figure 4.3, can be made from any point in the execution of state 2. A transition leaving the boundary of a super-state causes all sub-states of that super-state to also be exited. Leaving state 2, therefore, causes every active sub-state of state 2 to be exited, and it will leave only states 0 and 1 active once state 1 is entered.

The "battery removed" transition is labeled with the action "clh(main)*". This action will clear the history of the main state, state 3, when the transition is taken; clearing the history of state 3 causes the next entry into state 3 to act as if it were the first entry into state 3.

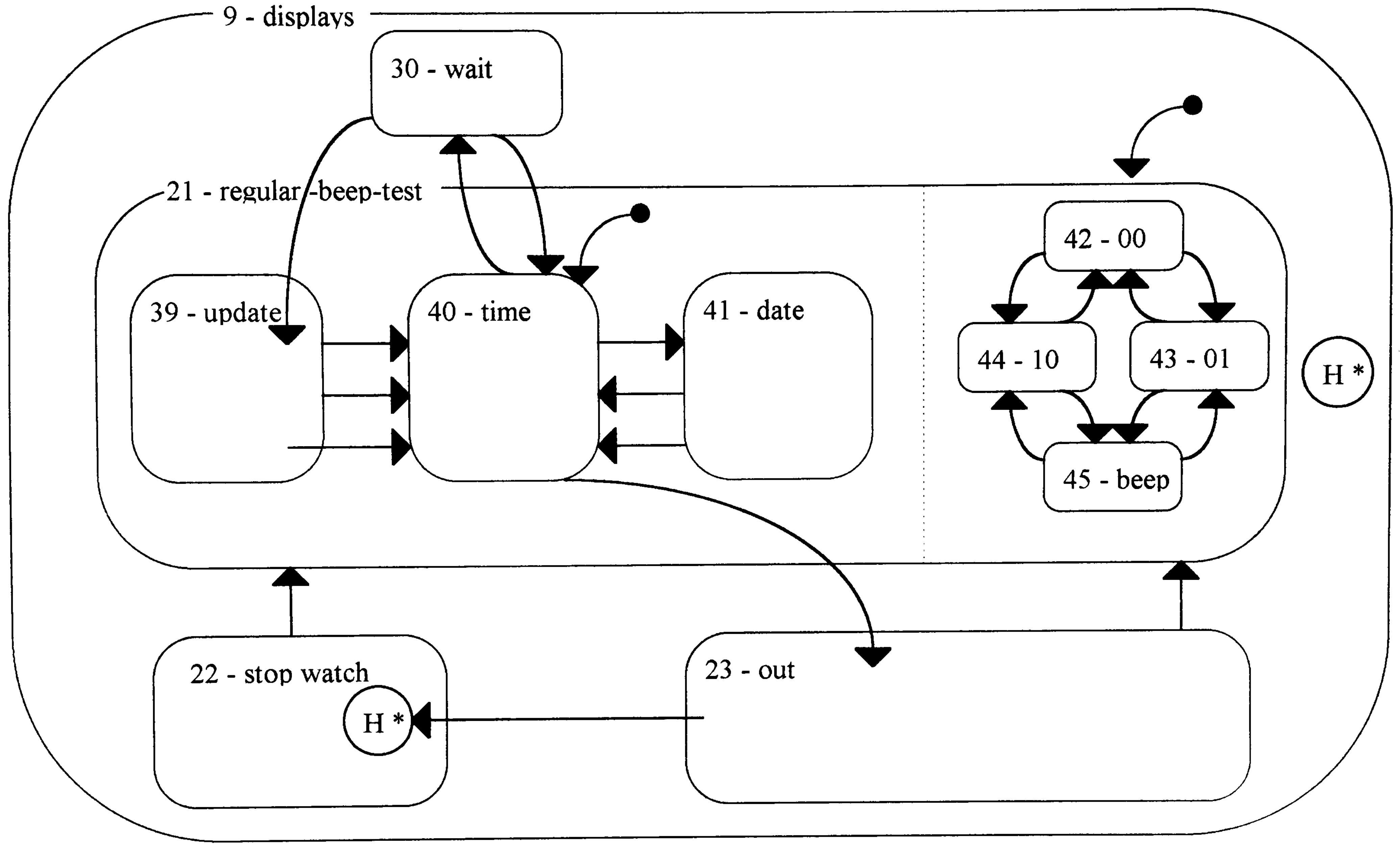


Figure 4.5. Statechart of the Displays State (State 9).

The “weak battery dies” transition does not leave the boundary of state 2. It causes side effects similar to the “battery removed” transition, because the transition crosses the boundary of state 2. Since the “weak battery dies” transition leaves from an AND component of state 2, the event on the transition is monitored by the Statechart Executive as long as state 2 is active. As soon as the battery dies, the transition is made. Each active sub-state of state 2 is exited, because the transition causes state 2 to exit.

CHAPTER V
CONCLUSIONS

Since its introduction in [Harel 87], the Statechart formalism has proven its capabilities of representing and describing the intricate nature of complex reactive systems. As a broad extension to traditional FSMs, Statecharts counter the objections raised to using FSMs as a graphical description language for real-time systems, for the extensions of hierarchy, concurrency, and communication combine the benefits of graphical representation while retaining the rigor of a formal description.

Prior to this research, Statecharts had been used to specify, design, and prototype reactive systems. They had been applied to requirements and specification, had moved a system through prototyping and design, and had formed the basis for maintenance and modifications. But when system development turned to implementation, the Statechart was translated from its graphical human readable form to a computer readable form.

This research answers the need for a real-time system development tool that is used in all phases of real-time systems development. This research sets forth the basis for a graphical development tool, which can provide solutions in the face of ever increasing complexity and unforgiving requirements. As in the past, the system can be specified and designed in the Statechart notation, but this research provides a database that captures the graphical representation of the Statechart without the requirement of human translation. The data held in the Statechart database can be presented to an execution engine without modification, for the specification of the system is also the implementation of the system.

By creating a database capable of storing the graphical representation of a Statechart, this research extends the Statechart formalism to actually become a programming language for complex systems. The Statechart database communicates the control algorithms directly to the computer, and this eliminates the need to convert control algorithms from their graphical representation to strings of symbols traditionally required by computers. The graphical representation is naturally descriptive and communicates the nature of the control algorithm to humans. The graphical representation of the data is also descriptive and precise to communicate the control algorithm directly to the computer.

Statecharts as a programming language can be readily understood by non-software engineers. The Statechart database and Statechart Executive provide a tool for building systems which are more reliable, because they can be developed by the engineers who understand the complex system in its real-world form. When software is developed for a complex system, the system can be developed more quickly than current methods, for the engineer will not have to translate to and rely upon software engineers to program the control algorithm in machine readable form.

Quick development and reliability will result when the engineer recognizes any design mistakes that manifest themselves as erroneous behavior of the system. These will be recognized during programming as well as during execution of the system, and the problems will be tracked quickly back to the Statechart. The result will be a more reliable system than produced by alternate development approaches.

This research further applies to companies which are currently unable to afford the cost of software development. Software development and maintenance costs have become one of the largest components of new product development. Many companies cannot support new software development costs, even in the face of sophisticated software requirements. In cases where software development costs are prohibitive, companies are looking for a means of cutting those costs.

A integrated real-time system development tool based on this research is a solution for those companies. Existing engineering staffs would program the control of systems. The company would then employ a unified team of developers to maintain the development tool as necessary. Engineers would program Statecharts out of their expertise, and programmers would maintain the tool within their realm of expertise. The cost of software development would shrink dramatically, for much of the required software development cycle would disappear. The core of the control system, the development tool, would be reused without modification from system to system.

Future Research

This research prototypes the benefits described above, and several areas of enhancement are left for future research. The future research can focus on (1) a development environment for reactive systems, (2) a compiler for the Statechart formalism, and (3) a generic input and output sub-system for interacting Statecharts with the real-world.

A Statechart programming environment should provide a full featured editing and debugging environment comparable to the integrated development environments (IDE) currently available for C and C++ programming. The IDE would allow stepping through a simulated or actual execution of the Statechart; additional features, such as break point execution should be permitted. Sophisticated features for cut, copy, and paste of graphical components would be supported.

Compiling a graphical notation is another area for future research. Harel [90] describes many features that a Statechart compiler should contain. A compiler will provide the benefits of correcting human error and human oversight in significant details. The compiler must be based on formal semantics, like those described for FSMs in finite state automata, and the accompanying automata for Statechart's extension to FSMs must be developed.

If the Statechart Executive was to be fully developed, it would have to follow trends and precedents in the mainstream of computing. Current trends include hardware independent solutions. For example, any manufacturer producing a modem for a personal computer could make the modem compatible with any variety of mother board. In this spirit, the success of the Statechart database and Statechart Executive would require a sophisticated strategy for compatibility with the spectrum of computer interface hardware. The Statechart Executive should be able to remain unchanged as hardware changed.

To accomplish hardware independence, the Statechart Executive would be distinct from the hardware specific routines. The routines would be accessed through a defined software interface into the Statechart Executive. The facility for this interface would be similar to Windows® dynamic link libraries (DLLs). The names of the library routines would be data loaded into the Statechart Executive at run-time, rather than object code linked in at compile time.

REFERENCES

Avnur, A. Finite State Machines for Real-time Software Engineering. Computing and Control Engineering Journal, v1, no 6, pp. 275-8, November 1990.

Brackett, M. Developing Data Structured Databases. Prentice Hall, Inc., Englewood Cliffs, New Jersey 1988.

Brathwaite, K. Analysis, Design, and Implementation of Data Dictionaries. McGraw Hill, New York 1988.

Brathwaite, K. Relational Theory: Concepts and Application. Academic Press, Inc., San Diego, California 1991.

Calloni, B; Bagert, D. BACCII: An Iconic Syntax-Directed System for Teaching Procedural Programming. Proceedings of the ACM South East Region 31st Annual Conference, pp. 177-183, Birmingham, Alabama, April 14-16, 1993.

Codd, E. The Relational Model for Database Management. Addison-Wesley Publishing Co., Inc., Reading, Massachusetts 1990.

Coleman, D; Hayes, F; Bear, S. Introducing Objectcharts or how to use Statecharts in object-oriented design. IEEE Transactions on Software Engineering, v 18, pp. 9-18, January 1992.

Gardarin, G.; Valduriez, P. Relational Databases and Knowledge Bases. Addison-Wesley Publishing Co., Inc., Reading, Massachusetts 1989.

Guest, S. The Use of Software Tools for Dialogue Design. International Journal of Man-Machine Studies, v16, no 3, pp. 263-285, March 1982.

Harel, D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, pp. 231-274, June 1987.

Harel, D. On Visual Formalisms. Communications of the ACM, v 31, no 5, pp. 514-530, May 1988

Harel, D; Lachover H.; Naamad A.; Pnueli; Politi; Sherman; Shtull-Trauring; Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, v 16, no 4, pp. 403-414, April 1990.

Jacob, R. A State Transition Diagram Language for Visual Programming. Computer, v 18, pp. 51-9, August 1985.

Joseph, M. , Goswami, A. Formal Description of Realtime Systems: a Review. Information and Software Technology, v 31, no 2, pp. 67-76, March 1989.

Kagan, H. Objects in Real Time. Byte, pp. 187-190, August 1992.

Microsoft® Visual Basic™ Version 3.0 Language Reference. Microsoft Corporation, United States of America 1993.

Microsoft® Visual Basic™ Version 3.0 Programmer's Guide. Microsoft Corporation, United States of America 1993.

Spence, L.; Vanden Eynden, C.; Gallin, D. Applied Mathematics for the Management, Life, and Social Sciences. Scott, Foresman/Little, Brown Higher Education, Glenview, Illinois 1990.

Williams, T. Real-time Operating Systems Struggle with Multiple Tasks. Computer Design, pp. 92-108, October 1, 1990.

APPENDIX

THE STATECHART DATABASE DICTIONARY

1. Statechart Database

This data base is the Statechart database for real-time system control.

1.1 Tables

All tables are located under this heading. Each table contains a text description of the table and lists the records contained in the table. The key for each table is highlighted in bold print and underlined.

1.1.1 Action Table

1.1.1.1 Description

The Action Table defines Statechart actions. Actions are associated with transitions, for the action is performed when the event, and corresponding conditions, for a particular transition evaluate to true. The action is performed prior to entering the TO State.

1.1.1.2 Records

1.1.1.2.1 **Action ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.1.2.2 Name

Text	32
Required	Yes
Index	No

1.1.1.2.2 Description

Text	100
Required	No
Index	No

1.1.1.2.3 Action Type

Text	50
Required	No
Index	No

1.1.1.2.4 Output ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.1.2.5 Variable ID

1.1.1.2.5 Variable ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.1.3 Associated Tables

Transition and Action Table

1.1.2 Activity Table

1.1.2.1 Description

The Activity Table contains the activities associated with a Statechart state. Activities are performed when the state becomes active, or is "entered", and activities are performed in one of the following three modes: the activity is performed once when the state is entered, the activity is performed once when the state is exited, or the activity is performed throughout the period when the state is active.

1.1.2.2 Records

1.1.2.2.1 **Activity ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.2.2.2 Name

Text	32
Required	Yes
Index	No

1.1.2.2.3 Description

Text	100
Required	No
Index	No

1.1.2.2.4 Activity Type

Integer	(0=Entry, 1=Exit, 2=Throughout)
Default Value	0
Required	Yes
Index	No

1.1.2.2.5 Output ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.2.2.6 Variable ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.2.3 Associated Tables

State and Activity Table

1.1.3 Condition Table

1.1.3.1 Description

The Condition Table contains conditions under which a Statechart event must evaluate to true. The condition augments the event, for the transition is not taken unless both the event and the condition evaluate to true. Conditions are formed by combining one or more expressions with zero or more operators.

1.1.3.2 Records

1.1.3.2.1 **Condition ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.3.2.2 Name

Text	32
Required	Yes
Index	No

1.1.3.2.3 Description

Text	100
Required	No
Index	No

1.1.3.3 Associated Tables

Condition and Expression Table
Condition and Operator Table

1.1.4 Condition and Expression Table

1.1.4.1 Description

The Condition and Expression Table lists the expressions that compose each condition, for a single condition can be made up of numerous expressions. The evaluation order is explicitly specified, for each condition/expression pair in this list contains the order, 1st, 2nd, 3rd, etc., in which it will be evaluated.

1.1.4.2 Records

1.1.4.2.1 **Condition ID**

Integer	>=0
Default Value	0
Required	Yes
Index	Yes (Duplicates OK)

1.1.4.2.2 **Expression ID**

Integer	>=0
Default Value	0
Required	Yes
Index	Yes (Duplicates OK)

1.1.4.2.3 **Order**

Integer	>=1
Default Value	1
Required	Yes
Index	No

1.1.4.3 Associated Tables

Condition Table
Expression Table

1.1.5 Condition and Operator Table
These operators are applied to expressions within conditions.

1.1.5.1 Description

The Condition and Operator Table contains the operators that compose each condition. Each condition/operator pair is ordered explicitly to define which operator to use as the condition is evaluated.

1.1.5.2 Records
1.1.5.2.1 **Condition ID**

Integer	>=0
Default Value	0
Required	Yes
Index	Yes (Duplicates OK)

1.1.5.2.2 **Operator ID**

Integer	>=0
Default Value	0
Required	Yes
Index	Yes (Duplicates OK)

1.1.5.2.3 **Order**

Integer	>=1
Default Value	1
Required	Yes
Index	No

1.1.5.3 Associated Tables

Condition Table
Operator Table

1.1.6 Conditional Entry Table
1.1.6.1 Description

The Conditional Entry Table contains the conditional entry information associated with transitions.

1.1.6.2 Records
1.1.6.2.1 **Conditional ID**

Integer	
Default Value	0
Required	Yes

Index	Yes (No Duplicates)

1.1.6.2.2 Name

Text	32
Required	Yes
Index	No

1.1.6.2.3 Description

Text	100
Required	No
Index	No

1.1.6.2.4 Transition ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.6.3 Associated Tables

Transition Table

1.1.7 Conditional Entry and Sub State Table

1.1.7.1 Description

The Conditional Entry and Sub State Table associates a conditional entry attribute with the sub-state that is entered conditionally.

1.1.7.2 Records

1.1.7.2.1 **Conditional ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.7.2.2 **Sub State ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.7.3 Associated Tables

State Table
Conditional Entry Table

1.1.8 Conversion Table

1.1.8.1 Description

The Conversion Table contains the conversion types associated with variables.

1.1.8.2 Records

1.1.8.2.1 **Conversion ID**

Integer	
Default Value	0

Required	Yes
Index	Yes (No Duplicates)

1.1.8.2.2 Name

Text	32
Required	Yes
Index	No

1.1.8.2.3 Description

Text	100
Required	No
Index	No

1.1.8.3 Associated Tables

1.1.9 Driver Table

1.1.9.1 Description

The Driver Table defines the input and output drivers used in Statechart activities and actions.

1.1.9.2 Records

1.1.9.2.1 **Driver ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.9.2.2 Name

Text	32
Required	Yes
Index	No

1.1.9.2.3 Description

Text	100
Required	No
Index	No

1.1.9.2.4 Address

Integer	
Default Value	-1
Required	No
Index	No

1.1.9.2.5 Number of Channels

Integer	
Default Value	-1
Required	No
Index	No

1.1.9.3 Associated Tables

Variable Table
Output Table
Input Table

1.1.10 Event Table

1.1.10.1 Description

The Event Table defines the events which determine when a transition is made. A transition from one state to another is made when the event evaluates to true.

1.1.10.2 Records

1.1.10.2.1 **Event ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.10.2.2 Name

Text	32
Text	32
Required	Yes
Index	No

1.1.10.2.3 Description

Text	100
Required	No
Index	No

1.1.10.2.4 Event Type

Integer	(0=Regular)
Default Value	0
Required	Yes
Index	No

1.1.10.2.5 Variable1 ID

Integer	>=0
Default Value	0
Required	Yes
Index	No

1.1.10.2.6 Operator ID

Integer	>=0
Default Value	0
Required	Yes
Index	No

1.1.10.2.7 Variable2 ID

Integer	>=0
Default Value	0
Required	Yes
Index	No

1.1.10.3 Associated Tables

Transition and Event Table

1.1.11 Expression Table

1.1.11.1 Description

The Expression Table defines the expressions which are used to form Statechart conditions. The expression is made up of two variables and an operator, and one or more expressions combine with zero or more operators to form conditions.

1.1.11.2 Records

1.1.11.2.1 **<u>Expression ID</u>**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.11.2.2 Name

Text	32
Required	Yes
Index	No

1.1.11.2.3 Description

Text	100
Required	No
Index	No

1.1.11.2.4 Variable1 ID

Integer	>=0
Default Value	0
Required	Yes
Index	No

1.1.11.2.5 Operator ID

Integer	>=0
Default Value	0
Required	Yes
Index	No

1.1.11.2.6 Variable2 ID

Integer	>=0
Default Value	0
Required	Yes
Index	No

1.1.11.3 Associated Tables

Condition and Expression Table
Operator Table
Variable Table

1.1.12 Input Table

1.1.12.1 Description

The Input Table defines inputs into the Statechart. Inputs are used with events and conditions on transitions to determine whether state changes are to take place.

The Select value contains the data associated with this input which is specifically related to the particular input driver.

1.1.12.2 Records

1.1.12.2.1 **Input ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.12.2.2 Name

Text	32
Required	Yes
Index	No

1.1.12.2.3 Description

Text	100
Required	No
Index	No

1.1.12.2.4 Driver ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.12.2.5 Select Value

Integer	
Default Value	0
Required	Yes
Index	No

1.1.12.3 Associated Tables

Driver Table

1.1.13 Operator Table

1.1.13.1 Description

The Operator Table contains each operator used in Expressions and Events.

1.1.13.2 Records

1.1.13.2.1 **Operator ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.13.2.2 Name

Text	32
Required	Yes
Index	No

1.1.13.2.3 Description

Text	100
Required	No
Index	No

1.1.13.3 Associated Tables

Condition and Operator Table
Event Table

1.1.14 Output Table

1.1.14.1 Description

The Output Table defines the outputs of the Statechart. Outputs are used in Activities and Actions to change the state of the Statechart.

The select value contains the data associated with this output.

1.1.14.2 Records

1.1.14.2.1 **Output ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.14.2.2 Name

Text	32
Required	Yes
Index	No

1.1.14.2.3 Description

Text	100
Required	No
Index	No

1.1.14.2.4 Driver ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.14.2.5 Select Value

Integer	
Default Value	0
Required	Yes
Index	No

1.1.14.3 Associated Tables

Variable Table

1.1.15 Range Table

1.1.15.1 Description

The Range Table defines valid ranges for Statechart variables. The range applies only to numeric (double precision) values.

1.1.15.2 Records

1.1.15.2.1 **Range ID**

Integer	
Default Value	0

Required	Yes
Index	Yes (No Duplicates)

1.1.15.2.2 Name

Text	32
Required	Yes
Index	No

1.1.15.2.3 Description

Text	100
Required	No
Index	No

1.1.15.2.4 LoValue

Double	
Default Value	0.0
Required	Yes
Index	No

1.1.15.2.5 UpValue

Double	
Default Value	0.0
Required	Yes
Index	No

1.1.15.3 Associated Tables

Variable Table

1.1.16 Selection Entry Table

1.1.16.1 Description

The Selection Entry Table contains entries that effect how a state is entered.

1.1.16.2 Records

1.1.16.2.1 **Selection ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.16.2.2 Name

Text	32
Required	Yes
Index	No

1.1.16.2.3 Description

Text	100
Required	No
Index	No

1.1.16.2.4 Transition ID

Integer	
Default Value	0
Required	Yes
Index	No

1.1.16.3 Associated Tables

State Table

1.1.17 Selection Entry and Sub State Table

1.1.17.1 Description

The Selection Entry and Sub State Table contains the selection entry and sub-state pairs that can augment how a state is entered.

1.1.17.2 Records

1.1.17.2.1 **Selection ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.17.2.2 **Sub State ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.17.3 Associated Tables

State Table
Selection Entry Table

1.1.18 State Table

1.1.18.1 Description

The State Table contains the Statechart states.

1.1.18.2 Records

1.1.18.2.1 **State ID**

Integer	
Default Value	0
Required	Yes
Index	Yes (No Duplicates)

1.1.18.2.2 Name

Text	32
Required	Yes
Index	No

1.1.18.2.3 Description

Text	100
Required	No
Index	No

1.1.18.2.4 Super State ID

Integer	>=-1
Default Value	-1
Required	No
Index	No

1.1.18.2.5 Default Sub State ID

Integer	>=-1
Default Value	-1
Required	No
Index	No

1.1.18.2.6 History

Integer	(0=No, 1=Yes, 2=Yes*)
Default Value	0
Required	Yes
Index	No

1.1.18.2.7 Type

Integer	(0=XOR, 1=AND)
Default Value	0
Required	Yes
Index	No

1.1.18.2.8 Time Out Length

Integer	
Default Value	0
Required	Yes
Index	No

1.1.18.3 Associated Tables

State and Activity Table
State and Sub State Table
State and Transition Table
Selection Entry and Sub State Table

1.1.19 State and Activity Table

1.1.19.1 Description

The State and Activity Table contains state and activity pairs. States can have multiple activities associated with them.

1.1.19.2 Records

1.1.19.2.1 **State ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.19.2.2 **Activity ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.19.3 Associated Tables

State Table
Activity Table

1.1.20 State and Sub State Table

1.1.20.1 Description

The State and Sub State Table contains state and sub-state pairs. States can have multiple sub-states.

1.1.20.2 Records

1.1.20.2.1 **State ID**

Integer

Default Value	0
Required	Yes
Index	No

1.1.20.2.2 **Sub State ID**

Integer

Default Value	0
Required	Yes
Index	No

1.1.20.3 Associated Tables

State Table

1.1.21 State and Transition Table

1.1.21.1 Description

The State and Transition Table contains state and transition pairs. The state represents the FROM State of the transition, which is the origin state of the transition.

1.1.21.2 Records

1.1.21.2.1 **State ID**

Integer

Default Value	0
Required	Yes
Index	No

1.1.21.2.2 **Transition ID**

Integer

Default Value	0
Required	Yes
Index	No

1.1.21.3 Associated Tables

State Table
Transition Table

1.1.22 Transition Table

1.1.22.1 Description

The Transition Table contains the Statechart transitions.

1.1.22.2 Records

1.1.22.2.1 **Transition ID**

Integer

Default Value	0

Required	Yes
Index	Yes (No Duplicates)

1.1.22.2.2 Name

Text	32
Required	Yes
Index	No

1.1.22.2.3 Description

Text	100
Required	No
Index	No

1.1.22.2.4 FROM State ID

Integer	
Default Value	0
Required	Yes
Index	Yes (Duplicates OK)

1.1.22.2.5 TO State ID

Integer	
Default Value	0
Required	Yes
Index	Yes (Duplicates OK)

1.1.22.2.6 Transition Type

Integer	
Default Value	0
Required	Yes
Index	No

1.1.22.2.7 Conditional ID

Integer	
Default Value	-1
Required	No
Index	No

1.1.22.2.8 Selection ID

Integer	
Default Value	-1
Required	No
Index	No

1.1.22.3 Associated Tables

State Table
Transition Table

1.1.23 Transition and Action Table

1.1.23.1 Description

The Transition and Action Table contains the transition and action pairs.

1.1.23.2 Records

1.1.23.2.1 **Transition ID**

Integer

Default Value	0
Required	Yes
Index	No

1.1.23.2.2 **Action ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.23.3 Associated Tables

Action Table
Transition Table

1.1.24 Transition and Event Table

1.1.24.1 Description

The Transition and Event Table contains the transition and event pairs.

1.1.24.2 Records

1.1.24.2.1 **Transition ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.24.2.2 **Event ID**

Integer	
Default Value	0
Required	Yes
Index	No

1.1.24.2.3 Condition ID

Integer	
Default Value	-1
Required	No
Index	No

1.1.24.3 Associated Tables

Event Table
Transition Table

1.1.25 Variable Table

1.1.25.1 Description

The Variable Table contains the Statechart variables. Variables hold the values, whether they be inputs or outputs, associated with the Statechart. Variables can be one of two distinct types: either string (character) or double precision (numeric).

1.1.25.2 Records

1.1.25.2.1 **Variable ID**

Integer

```
   Default Value  0
   Required       Yes
   Index          Yes (No Duplicates)

1.1.25.2.2 Name
   Text           32
   Required       Yes
   Index          No

1.1.25.2.3 Description
   Text           100
   Required       No
   Index          No

1.1.25.2.4 Conversion ID
   Integer
   Default Value  -1
   Required       No
   Index          No

1.1.25.2.5 Range ID
   Integer
   Default Value  -1
   Required       No
   Index          No

1.1.25.2.6 Units ID
   Integer  2
   Default Value  -1
   Required       No
   Index          No

1.1.25.2.7 Value Type
   Integer        (0=String, 1=double)
   Default Value  0
   Required       Yes
   Index          No

1.1.25.2.8 Default String Value
   Text           100
   Required       No
   Index          No

1.1.25.2.9 String Value
   Text           100
   Required       No
   Index          No

1.1.25.2.10 Default Double Value
   Double
   Default Value  0.0
   Required       No
   Index          No

1.1.25.2.11 Double Value
   Double
   Default Value  0.0
```

Required	No
Index	No

1.1.25.2.12 Snapshot Variable

Yes/No

Default Value	No
Required	Yes
Index	No

1.1.25.2.13 Trace Variable

Yes/No

Default Value	Yes
Required	No
Index	No

1.1.25.2.14 Input ID

Integer

Default Value	-1
Required	No
Index	No

1.1.25.2.15 Output ID

Integer

Default Value	-1
Required	No
Index	No

1.1.25.3 Associated Tables

Input Table
Output Table
Expression Table

1.2 Queries

This heading defines the queries of the Statechart database. Each query definition contains a text description of the query, the list of tables used in the query, and records of the query.

1.2.1 Get All Conditions
1.2.1.1 Description

The Get All Conditions query will retrieve all the pertinent condition information. The result of the query will be all the expressions and operators associated with a given condition, and each of the condition information will be sorted in ascending order by the following three fields: Condition ID, Expression Order, and Operator Order.

1.2.1.2 Tables
1.2.1.2.1 Condition Table

1.2.1.2.2 Condition and Expression Table

1.2.1.2.3 Condition and Operator Table

1.2.1.3 Records
1.2.1.3.1 Condition ID
Sorted Ascending

1.2.1.3.2 Condition Name

1.2.1.3.3 Expression ID

1.2.1.3.4 Expression Order
Sorted Ascending

1.2.1.3.5 Operator ID

1.2.1.3.6 Operator Order
Sorted Ascending

1.2.2 Get All Drivers
1.2.2.1 Description
1.2.2.2 Tables
1.2.2.2.1 Drivers Table

1.2.2.3 Records
1.2.2.3.1 Driver ID
Sorted Ascending

1.2.2.3.2 Driver Name

1.2.2.3.3 Address

1.2.2.3.4 Number of Channels

1.2.3 Get All States
1.2.3.1 Description

```
   1.2.3.2 Tables
      1.2.3.2.1 State Table

      1.2.3.2.2 State and SubState Table

      1.2.3.2.3 State and Transition Table

   1.2.3.3 Records
      1.2.3.3.1 State ID
         Sorted   Ascending
         Joined   State Table -> State and SubState Table
         Joined   State Table -> State and Transition Table

      1.2.3.3.2 State Name

      1.2.3.3.3 History

      1.2.3.3.4 Time Out Length

      1.2.3.3.5 Super State ID

      1.2.3.3.6 Default SubState ID

      1.2.3.3.7 Type

      1.2.3.3.8 SubState ID
         Sorted   Ascending

      1.2.3.3.9 Transition ID
         Sorted   Ascending

1.2.4 Get All Transitions
   1.2.4.1 Description
   1.2.4.2 Tables
      1.2.4.2.1 Transition Table

      1.2.4.2.2 Transition and Action Table

      1.2.4.2.3 Transition and Event Table

   1.2.4.3 Records
      1.2.4.3.1 Transition ID
         Sorted   Ascending
         Joined   Transition Table -> Transition and Action Table
         Joined   Transition Table -> Transition and Event Table

      1.2.4.3.2 Transition Name

      1.2.4.3.3 FROM State ID

      1.2.4.3.4 TO State ID

      1.2.4.3.5 Transition Type

      1.2.4.3.6 Action ID
         Sorted   Ascending
      1.2.4.3.7 Event ID
         Sorted   Ascending

      1.2.4.3.8 Condition ID
         Sorted   Ascending

1.2.5 Get All Variables
   1.2.5.1 Description
   1.2.5.2 Tables
      1.2.5.2.1 Variable Table

   1.2.5.3 Records
      1.2.5.3.1 Variable ID
         Sorted Ascending

      1.2.5.3.2 Variable Name

      1.2.5.3.3 Input ID

      1.2.5.3.4 Output ID
```