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

DAVID HAREL, MEMBER, IEEE, HAGI LACHOVER, AMNON NAAMAD, AMIR PNUELI,
MICHAL POLITI, RIVI SHERMAN, AHARON SHTULL-TRAURING,
AND MARK TRAKHTENBROT

Abstract—This paper provides an overview of the STATEMATE® system, constructed over the past several years by the authors and their colleagues at Ad Cad Ltd., the R&D subsidiary of i-Logix, Inc. 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, control and communication systems, and interactive software or hardware. It enables a user to prepare, analyze, and debug diagrammatic, yet precise, descriptions of the system under development from three interrelated points of view, capturing *structure*, *functionality*, and *behavior*. These views are represented by three graphical languages, the most intricate of which is the language of *statecharts* [4], used to depict reactive behavior over time. In addition to the use of statecharts, the main novelty of STATEMATE is in the fact that it "understands" the entire descriptions perfectly, to the point of being able to analyze them for crucial dynamic properties, to carry out rigorous executions and simulations of the described system, and to create running code automatically. These features are invaluable when it comes to the quality and reliability of the final outcome.

Index Terms—Code-generation, executable specifications, functional decomposition, propotyping, reactive systems, statecharts, STATEMATE.

I. Introduction

REACTIVE systems (see [18], [6]) are characterized as owing much of their complexity to the intricate nature of reactions to discrete occurrences. The computational parts of such systems are assumed to be dealt with using other means, and it is their reactive, control-driven parts that are considered here to be the most problematic. Examples of reactive systems include most kinds of real-time computer embedded systems, control plants, communication systems, interactive software of varying nature, and even VLSI circuits. Common to all of these is the notion of *reactive behavior*, whereby the system is not adequately described by specifying the output that results from a set of inputs, but, rather, requires specifying the relationship of inputs and outputs over time. Typically, such descriptions involve complex sequences of events, actions, conditions and information flow, often with explicit timing constraints, that combine to form the system's overall behavior.

It is fair to say that the problem of finding good methods to aid in the development of such systems has not been satisfactorily solved. Standard structured analysis and structured design methods do not adequately deal with the dynamics of reactive systems, since they were proposed to deal primarily with nonreactive, data-driven applications, in which a good functional decomposition and data-flow description are sufficient. Some of these methods have recently been extended to deal with real-time systems (see, e.g., [8], [9], [16], [17], [20]–[22]), and our approach, developed independently,[1] can be viewed as being consistent with many of the ideas in these. See the comparisons in the recent [22]. As to commercially available tools for real-time system design, most are, by and large, but sophisticated graphics editors, with which one can model certain aspects of reactive systems, but with which a user can do little with the resulting descriptions beyond testing them for syntactic consistency and completeness and producing various kinds of output reports. These systems are often helpful in organizing a designer's thoughts and in communicating those thoughts to others, but they are generally considered severely inadequate when it comes to the more difficult task of preparing reliable specifications and designs that satisfy the requirements, that behave over time as expected, and from which a satisfactory final system can be constructed with relative ease.

If we were to draw an analogy with the discipline of conventional programming, there is an acute need for the reactive system's analog of a programming environment that comes complete with a powerful programming language, a useful program editor and syntax checker, but, most importantly, also with a working compiler and/or interpreter, and with extensive debugging facilities. Programs are not only to be written and checked for syntax errors; they must also be run, tested, debugged, and thoroughly analyzed before they are set free to do their thing in the real world.

As it turns out, the problems arising in the design of a typical reactive system are far more difficult than those

Manuscript received May 12, 1988; revised November 13, 1989. Recommended by L. A. Belady. This work was supported in part by the Bird Foundation and the Israel Ministry of Industry and Commerce.

D. Harel and A. Pnueli are with i-Logix Inc., Burlington, MA 01803, Ad Cad Ltd., Rehovot, Israel, and the Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel.

H. Lachover, A. Naamad, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot are with i-Logix Inc., Burlington, MA 01803, and Ad Cad Ltd., Rehovot, Israel.

IEEE Log Number 8933740.

[1] Most of the ideas described in this paper were conceived between 1983 and 1985.

0098-5589/90/0400-0403$01.00 © 1990 IEEE

arising in the preparation of a typical computational or data-processing program. Most reactive systems are highly concurrent and distributed; they fall quite naturally into multiple levels of detail, and usually display unpredictable, often catastrophic, behavior under unanticipated circumstances. More often than not, the development phases of such systems are laden with misunderstandings between customers, subcontractors, and users, as well as among the various members of the design team itself, and their life-cycle is replete with trouble-shooting, modifications, and enhancements.

The languages in which reactive systems are specified ought to be clear and intuitive, and thus amenable to generation, inspection and modification by humans, as well as precise and rigorous, and thus amenable to validation, simulation, and analysis by computers. Such languages ought to make it possible to move easily, and with sufficient semantic underpinnings, from the initial stages of requirements and specification to prototyping and design, and to form the basis for modifications and maintenance at later stages. One of the underlying principles adopted in this paper is that clarity and intuition can be greatly enhanced by the adoption of visual languages for the bulk of the description effort, behavioral aspects included. This, together with the need for precision and rigor, leads naturally to the notion of *visual formalisms* [5], i.e., languages that are highly visual in nature, depending on a small number of carefully chosen diagrammatic paradigms, yet which, at the same time, admit a formal semantics that provides each feature, graphical and nongraphical alike, with a precise and unambiguous meaning. For reactive systems, this means that it should be possible to prepare intuitive and comprehensive specifications that can be analyzed, simulated, and debugged at any stage with the aid of a computerized support system.

This paper describes the ideas behind STATEMATE, a computerized environment for the development of reactive systems, which adheres to these principles. The reader is also referred to additional material about the system, particularly [12]–[14].

II. STATEMATE at a Glance

The underlying premise of STATEMATE is the need to specify and analyze the system under development (SUD in the sequel) from three separate, but related, points of view: *structural*, *functional*, and *behavioral*. The latter two are closely linked, as we shall see later, and constitute together the *conceptual model* of the SUD. See Fig. 1.

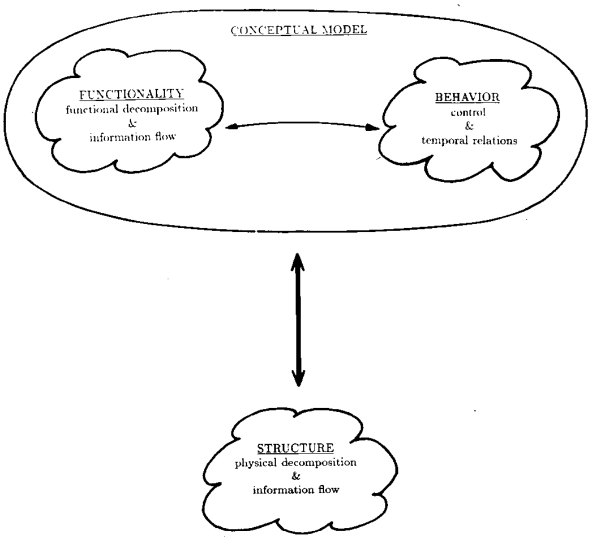


Fig. 1. Structure of a STATEMATE model.

In the structural view, one provides a hierarchical decomposition of the SUD into its physical components, called *modules* here, and identifies the *information* that flows between them; that is, the "chunks" of data and control signals that flow through whatever physical links exist between the modules. The word "physical" should be taken as rather general, with a module being anything from an actual piece of hardware in some systems to the subroutines, packages and tasks in the software parts of others.

The conceptual model of the SUD consists of a hierarchy of *activities*, complete with the details of the *data items* and *control signals* that flow between them, and, significantly, *control activities* that specify behavior. Let us be a little more explicit here. The activity hierarchy and flow information (without the control activities) constitute our functional view, and are essentially what is often called the *functional decomposition* of the SUD. However, in the functional view we do not specify dynamics; we do not say when and why the activities are activated, whether or not they terminate on their own, and whether they can be carried out in parallel. The same is true of the data-flow; in the functional view we specify only that data *can* flow, and not whether and when it will. For example, if we have identified that two of the subactivities of an automatic teller machine are **identify-customer** and **report-balance,** and that the data item **account-number** can flow from the former to the latter, then no more and no less than that is implied; we still have not specified when that item will flow, how often will it flow, and in response to what, and, indeed, whether the flow will be initiated by the former activity or requested by the latter. In other words, the functional view provides the decomposition into activities and the possible flow of information, but it says virtually nothing about how those activities and their associated inputs and outputs are controlled during the continued behavior of the SUD.

It is the behavioral view, our third, that is used to specify the control activities. These can be present on any level of the activity hierarchy, controlling that particular level. It is these controllers that are responsible for specifying when, how and why things happen as the SUD reacts over time. Among other things, a controlling statechart can start and stop activities, can generate new events, and can change the values of variables. It can also sense whether activities are active or data has flowed, and it can respond to events and test the values of conditions and variables. These connections between activities and control will be seen in Section III to be rather elaborate and multifaceted, so that the conceptual model should be regarded as a closely knit aggregate. The relationship between this conceptual model and the physical view, on the other hand, is far simpler, and consists essentially of specifying which modules implement which activities.

For these three views, the structural, functional, and behavioral, STATEMATE provides graphical, diagrammatic languages, *module-charts, activity-charts*, and *statecharts*, respectively. All three languages are based on a common set of simple graphical conventions (see [5]) and come complete with graphics editors that check for syntactic validity as the specifications are developed, and, more importantly, with formal semantics that are embedded within. The languages are described in some detail in Section III, and in more detail in [12] (statecharts are described in [4]).

Fig. 2 illustrates the overall structure of STATEMATE. The database is central, and obtains much of its input from the three graphics editors, and also from an editor for a *forms language*, in which additional information is specified, as we shall see later.

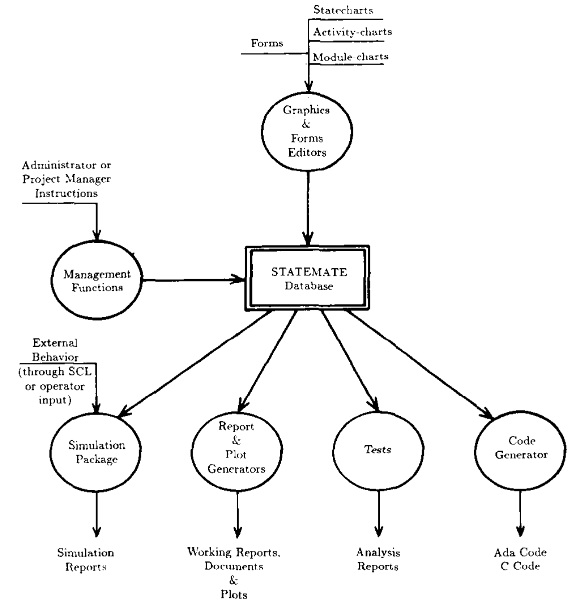


Fig. 2. Overall structure of STATEMATE.

The most interesting parts of STATEMATE, however, are the analysis capabilities, described in Sections IV and V and in [13]. As mentioned, the entire approach is governed by the desire to enable the user to run, debug and analyze the specifications and designs that result from the graphical languages. To this end, the database has been constructed to make it possible to rigorously execute the specification and to retrieve information of a variety of kinds from the description of the SUD provided by the user. Some of the special tools provided for these purposes are 1) a means for querying the database and retrieving information from it; 2) an execution ability with a simulation control language, allowing the user to emulate the SUD's environment and execute the specifications, interactively or in batch or programmed mode, with or without graphic animated response, and using breakpoints if desired; 3) a set of dynamic tests, e.g., for reachability and the detection of deadlock and nondeterminism, which are based on exhaustive executions; 4) an automatic translation of the specification into a high-level programming language, such as Ada or C, yielding code that can be linked to a real or simulated target environment.

STATEMATE has been under development and extension since early 1984, and has been commercially available since late 1987. The currently available versions run on Sun, Apollo and Vax color[2] workstations (or networks of such). Many of the ideas and methods embodied in STATEMATE have been field-tested successfully in a number of large real-world development projects, among which is the mission-specific avionics system for the Lavi fighter aircraft designed by the Israel Aircraft Industries, which was specified in part using statecharts (see [4]). The reader is also referred to [19], a case study of using STATEMATE, to [11], in which an application to process modeling is described, and to the recent comparative evaluation [22].

III. The Modeling Languages of STATEMATE

In this section we present the highlights of the three graphical languages and the forms language that the user of STATEMATE employs to specify the SUD. No formal syntax or semantics are given here, neither are all of the features presented. The reader is referred to [12] for a more comprehensive description, and to [4], [14] for a detailed treatment of the language of statecharts. The languages are described with the help of a simple example of an early warning system (EWS in the sequel), which has the ability to take measurements from an external sensor, compare them to some prespecified upper and lower limits, and warn the user when the measured value exceeds these limits.

The structural view of the SUD is described using the language of *module-charts*, which describe the SUD *modules* (i.e., its physical components), the environment modules (i.e., those parts that for the purpose of specification are deemed to be external to the SUD), and the clusters of data and/or control signals that may flow among them. Modules are depicted as rectilinear shapes and storage modules as rectangles with dashed sides. Encapsulation is used to capture the submodule relationship. Environment modules appear as dashed-line rectangles external to that of the SUD itself. Information flow is represented by labeled arrows or hyperarrows.[3] Various kinds

[2] While color appears to significantly enhance the appeal of STATEMATE, monochrome versions are also available.

[3] A hyperarrow has more than two endpoints.

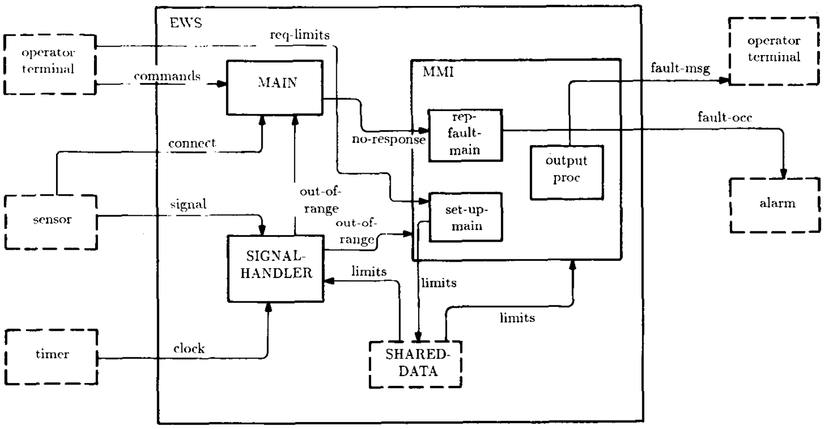


Fig. 3. Module-chart of the early warning system.

of connectors can appear in these charts, both to abbreviate lengthy arrows and to denote compound chunks of data.

Fig. 3 is (part of) the module-chart of our early warning system. It specifies in a self-explanatory fashion that the modules, or subsystems, of the EWS are a **main** component, a **man–machine-interface** (MMI), and a **signal-handler,** and that the **operator-terminal, sensor, timer,** and **alarm** are considered to be external to the system. The MMI is further decomposed into submodules, as shown. There is also a storage module, by the name of **shared-data,** and the information flowing between the modules is specified as well.

Turning to conceptual modeling, the functional decomposition of the SUD is captured by the language of *activity-charts*. Graphically, these are very similar to module-charts, but here the rectilinear shapes stand for the *activities*, or the functions, carried out by the system. Solid arrows represent the flow of data items and dashed arrows capture the flow of control items.[4] See Fig. 4. A typical activity will accept input items and produce output items during its active time-spans, its inner workings being specified by its own lower level decomposition. Activities that are atomic, or *basic* (i.e., they reside on the lowest level) may be described as simple input/output transformations using other means, such as code in a high-level programming language.

Activity-charts may contain two additional kinds of objects: *data-stores* and *control activities*. Data-stores can be thought of as representing databases, data structures, buffers, and variables of various kinds, or even physical containers or reservoirs, and typically correspond to the storage modules in the module-chart. They represent the ability to store the data items that flow into them and to produce those items as outputs upon request.

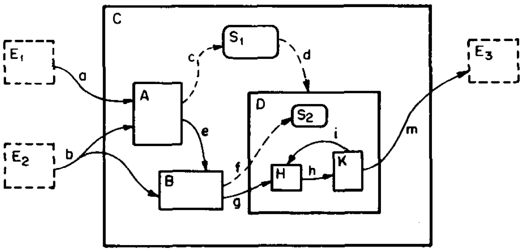


Fig. 4. An activity-chart.

The control activities constitute the behavioral view of the system and they appear in the activity-chart as empty boxes only. A control activity may appear inside an activity on any level, as shown in Fig. 4. The contents of the control activities are described in the third of the graphical languages, *statecharts*, which we discuss below. In general, a control activity has the ability to control its sibling activities by essentially sensing their status' and issuing commands to them. Thus, for example, in Fig. 4 the control activity S_1 can, among other things, perform *actions* that cause subactivities A, B, and D to start and stop, and can sense whether those subactivities have started or stopped by appropriate *events* and *conditions*. Various consequences of such occurrences are integrated into the semantics of the activity-charts language, such as the fact that all subactivities stop (respectively, suspend) upon the stopping (respectively, suspension) of the parent activity.

We now turn to the behavioral view. Statecharts, which were introduced in [4] (see also [5]), are an extension of conventional finite-state machines (FSM's) and their visual counterpart, state-transition diagrams. Conventional state diagrams are inappropriate for the behavioral description of complex control, since they suffer from being flat and unstructured, are inherently sequential in nature, and give rise to an exponential blow-up in the number of states (i.e., small extensions of a system cause unacceptable growth in the number of states to be considered). These problems are overcome in statecharts by supporting

[4] In displaying module-charts and activity-charts on the screen, we employ different conventions regarding color and arrow type, so that a user can distinguish between them quite easily. Thus, for example, the arrows in module-charts are drawn using rectilinear segments parallel to the axes, whereas in activity-charts they are drawn using smooth spline functions.

the repeated decomposition of states into substates in an AND/OR fashion, combined with an instantaneous broadcast communication mechanism. A rather important facet of these extensions is the ability to have transitions leave and enter states on any level.

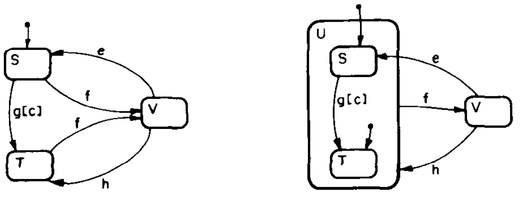
Consider Fig. 5, in which (a) and (b) are equivalent. In Fig. 5(b) states S and T have been clustered into a new state U so that to be in U is to be either in S or in T. The f-arrow leaving U denotes a high-level interrupt, and has the effect of prescribing an exit from U, i.e., from whichever of S or T the system happens to be in, to the new state V. The h-arrow entering U would appear to be underspecified, as it must cause entry to S or T; in fact, its meaning relies on the internal default arrow attached to T, which causes entrance to T.

Turning to AND decomposition, consider Fig. 6, in which, again, Fig. 6(a) and (b) are equivalent. Here, to be in state U the system must be in *both* S and T. An unspecified entrance to U relies on both default arrows to enter the pair $\{V, W\}$, from which an occurrence of e, for example, would lead to the new pair $\{X, Y\}$, and k would lead to $\{V, Z\}$. The meaning of the other transitions appearing therein, including entrances and exits, can be deduced by comparing Fig. 6(a) and (b). It is worth mentioning that this AND decomposition, into what we call *orthogonal* state components, can be carried out on any level of states and is therefore more convenient than allowing only single-level sets of communicating FSM's. Orthogonality is the feature statecharts employ to solve the state blow-up problem, by making it possible to describe independent and concurrent state components; see [4], [5]. Also, orthogonal state decomposition eliminates the need for multiple control activities within a single activity, as is done, e.g., in [9], [21].

The general syntax of an expression labeling a transition in a statechart is

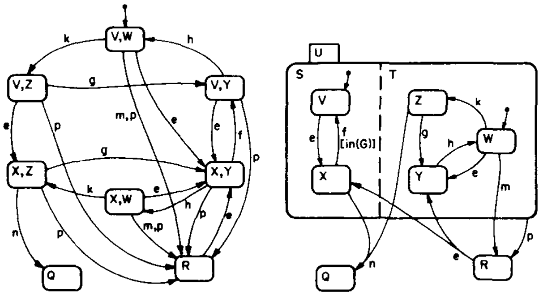
$$\alpha[C]/\beta$$

where α is the event that triggers the transition, C is a condition that guards the transition from being taken unless it is true when α occurs, and β is an action that is carried out if, and precisely when, the transition is taken. All of these are optional. Events and conditions can be considered as inputs, and actions as outputs, except that here this correspondence is more subtle than in ordinary FSM's, due to the intricate nature of the statecharts themselves and their relationship with the activities. For example, if β appears as an action along some transition, but it also appears as a triggering event of a transition in some orthogonal component, then executing the action in the first transition will immediately cause the second transition to be taken simultaneously. Moreover, in the expression α/β, rather than being simply a *primitive* action that might cause other transitions, β can be the special action **start**(A) that causes the activity A to start. Similarly, rather than being simply an external, primitive event, α might be the special event **stopped**(B) that occurs (and hence causes the transition to take place) when B stops or is stopped. Table I shows a selection of the special events, conditions, and actions that can appear as part of the labels along a transition. It should be noted that the syntax is also closed under boolean combinations, so that, for example, the following is a legal label:



(a) (b)

Fig. 5. OR-decomposition in a statechart.



(a) (b)

Fig. 6. AND-decomposition in a statechart.

$$\textbf{entered}(S)\ [\textbf{in}(T) \textbf{ and not active}(C)]/$$
$$\textbf{suspend}(C);\ X := Y + 7$$

Notice that we have incorporated another extension of the FSM approach—the use of conventional variables. The changing of a value is now allowed as an event, standard comparisons are allowed as conditions, and assignment statements are allowed as actions.

Besides allowing actions to appear along transitions, they can also appear associated with the entrance to or exit from a state (any state, of course, on any level).[5] This association is currently specified nongraphically, in the forms language discussed below. Thus, if we associate the action **resume**(A) with the entrance to state S, activity A will be resumed whenever S is entered.

Some of the special constructs appearing in Table I thus serve to link the control activities with the other objects appearing in an activity-chart, and, as such, are part of the way behavior is associated with functionality and dataflow. There are other facets to this association, one of which is the ability to specify an activity A as taking place **throughout** state (S), which is the same as saying that A is started upon entering state S and is stopped upon leaving it. This connection is also stated via forms.

The power to control and sense the status of activities is limited by a scoping rule to the control activity appear-

[5] In this way, statecharts can be seen to generalize both Mealy and Moore automata; see [10].

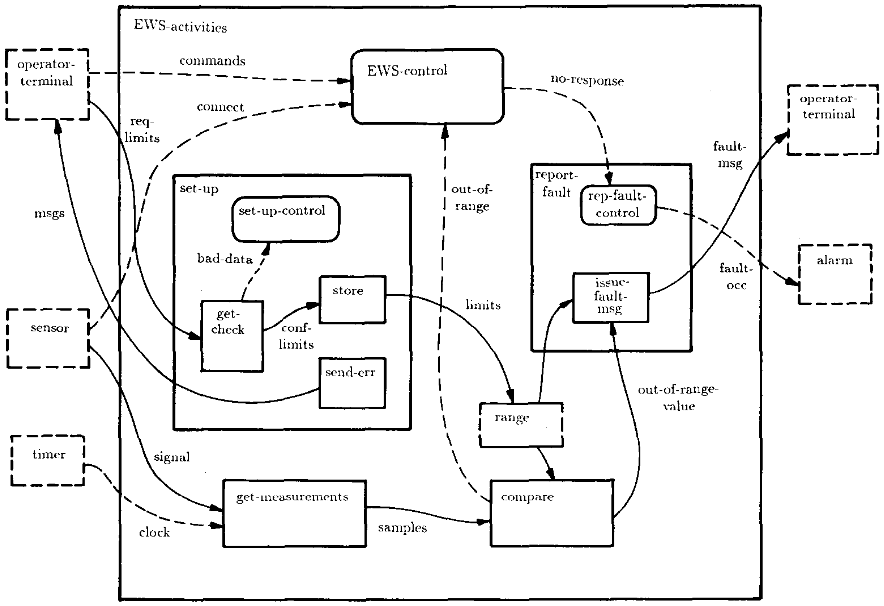


Fig. 7. Activity-chart of the early warning system.

TABLE I
SOME SPECIAL EVENTS, CONDITIONS, AND ACTIONS

REFERRING TO	EVENTS	CONDITIONS	ACTIONS

state
S
	entered(S)
	exited(S)
		in(S)	

activity
A
	started(A)
	stopped(A)
		active(A)
		hanging(A)
			start(A)
			stop(A)
			suspend(A)
			resume(A)

data items
D,F
condition C
	read(D)
	written(D)
	true(C)
	false(C)
		D=F
		D<F
		D>F
		⋮
			D:=exp
			make_true(C)
			make_false(C)

event E action A n-time units	timeout(E,n)		schedule(A.n)

ing on the same level as the activities and flow in question. Thus, in Fig. 4, for example, some of the events and actions that can appear in the statechart S_1 are:[6] **st**(A), **rs!**(B) and **wr!**(d), but ones referring to, say, H and K, such as **st!**(H), cannot, and would appear only in S_2. This scoping mechanism for hiding information is intended to help in making specifications modular and amenable to the kind of division of work that is required in large projects. The scoping rule can easily be overridden by explicitly-flowing events and actions, but we shall not get into the details here.

Fig. 7 shows the activity-chart of the early warning system. The user, via the operator terminal, can send **commands** to the control activity; this is an *information flow*, which, via a form, is specified to consist of **set-up, execute,** and **reset** instructions. The operator can also send the upper and lower required limits to the **get-check** subactivity of **set-up.** These limits can be stored in the data-store **range,** and can subsequently be used by the **compare** and **report-fault** activities. (The item **req-limits** is a compound data item, and stands for the pair containing the required upper and lower limits.) A special activity, **get-measurements,** can receive the **signal** from the sensor and a **clock** reading from the timer, and translates these into a time-stamped digital value sample, which can be sent to the comparing activity. If out of range, a signal and value can be sent to the controller and the **report-fault** activity, respectively. The latter is responsible for sending out an alarm and formatting and sending the user an appropriate message. The second level of Fig. 7 is self explanatory.

It is important to emphasize that Fig. 7 is not required to provide dynamic, behavioral information about the EWS; that is the role of the controlling statecharts. Fig. 8, for example, shows one possible statechart for the high-level control activity of Fig. 7, i.e., EWS-**control,** and the reader should be able to comprehend it quite easily.

The connections between activity-charts and statecharts are rather intricate, resulting in a tightly knit conceptual model. In contrast, the connections between this model and the structural view are more strightforward. What we have to do is to assign implementational responsibility for each part of the former to appropriate parts of the latter. This is done by associating modules with activities, and storage modules with data stores. In our example, some of these associations are that the MAIN module implements the EWS-**control** activity, SIGNAL-HANDLER

[6] Here, and also in Fig. 8, we use abbreviations for the elements appearing in Table I, such as **st** instead of **started, rs!** instead of **resume,** and **tm** instead of **timeout.** STATEMATE recognizes these abbreviations too.

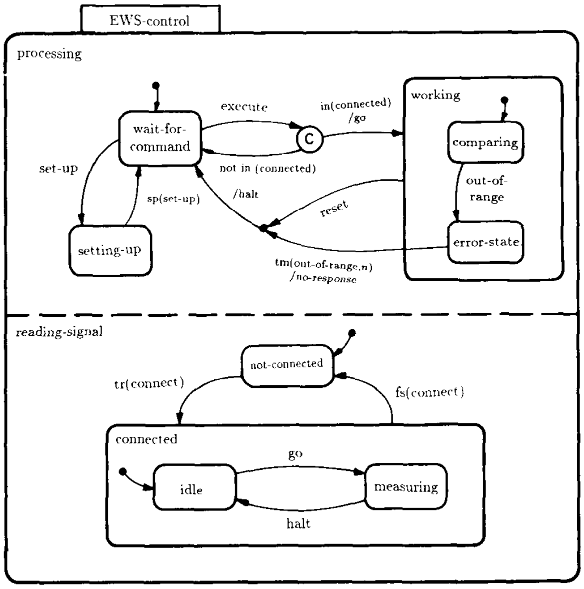
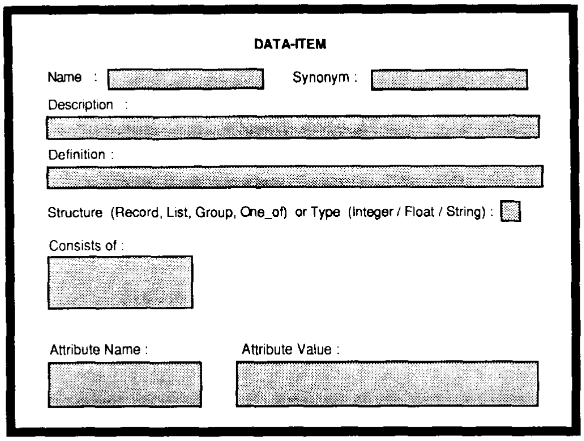


Fig. 8. Statechart for the high-level activity of the warning system.

implements **get-measurements** and **compare,** and MMI implements **set-up** and **report-fault.** Within the MMI association, the **send-err** subactivity is implemented by the **output-proc** submodule and the other three by **set-up-main.** The associations themselves are input in the forms of the activities.

We now turn to the *forms language* itself. A special form is maintained for each of the elements in the description, in which additional information can be input. This includes details that are nongraphical in nature, such as lengthy definitions of compound events and conditions, or the type and structure of data items. Fig. 9 shows an example of the form for a data item, in which most fields are self-explanatory. The "Consists of" field therein makes it possible to structure data items into components, and the "Attribute" fields make it possible to associate attributes with the items (e.g., units and precision for certain kinds of data-items, or the names of the personnel responsible for the specification for certain high-level elements). The attributes are recognizable by the retrieval tools of STATEMATE and are therefore able to play a role in the evaluation and documentation of the model, as we shall see later.

The color graphical editors for all three charts languages continuously check the input for syntactic soundness, and the database is updated as graphical elements are introduced. The editors are mouse- and menu-based, and support a wide range of possibilities, including move, copy, stretch, hide, reveal, and zoom options, all applicable to single or multiple elements in the charts, that can be selected in a number of ways. The form for a selected element can be viewed and updated not only from the special forms editor, but from the appropriate graphical editor as well.



DATA-ITEM

Name : Synonym :

Description :

Definition :

Structure (Record, List, Group, One_of) or Type (Integer / Float / String) : □

Consists of :

Attribute Name : Attribute Value :

Fig. 9. The form for a data item.

Extensive consistency and completeness tests, as well as more subtle *static logic* tests can be carried out during a session. Examples include checking whether information flow in the module-chart is consistent with that in the activity-chart, listing modules that have no outputs, or activities that are never started, and identifying cyclic definitions of nongraphical elements (e.g., events and conditions).

IV. Queries, Reports, and Documents

In this section, we describe some of the tools that are available for retrieving and formatting information from the model.

STATEMATE provides a querying tool, the *object list generator*, with which the user can retrieve information from the database. It works by generating lists of elements that satisfy certain criteria. At all times it keeps a *pending list* that gets modified as the user refines the criteria or asks for a list of elements of another type. For example, starting with an empty pending list, one can ask for all states in the controlling statechart of activity *A*, and the resulting list promptly becomes the new pending list. This list might then be refined by asking for those states therein that have an attribute named "**responsibility-of**" whose value is "**Jim Brown.**" Then one might ask for all activities that are started within any of those states, and so on. This query language, on the face of it, might appear to be bounded in its expressive power by that of the conjunctive queries of [2]. However, since it actually supports certain kinds of transitive closures (such as the ancestor and descendent relationships between states or activities), it is not directly comparable with the conjunctive queries, and can be shown to be a subset of the more general fixpoint queries (see [1]).

The charts that constitute the SUD's description can be plotted. The user can control the portion of the chart to be plotted, as well as its size and depth. In addition, the user can ask for several kinds of fixed-format **reports** that are compiled directly from the description of the SUD in the database, and which can be displayed on the workstation screen or output to an alphanumeric terminal or printer. Each of these can be projected, so to speak, on any part of the description that is retrievable by the query language. In other words, the user may first use queries to capture, say, a set of activities of particular interest, and then request the report; it will be applied only to the activities in the list. Among the reports currently implemented are *data dictionaries* of various kinds, textual *protocols* of states or activities that contain all the information relevant to them, *interface diagrams*, *tree versions* of various hierarchies, and so-called N^2-diagrams of [15]. Using certain parameters, the user can control various aspects of the reports produced, such as the depth of the trees in the tree reports, and the keys by which the dictionaries are to be sorted.

In addition to fixed-format reports, STATEMATE has a *document generation language* with which users can tailor their own documents. Programs can be written in this language to produce documents with particular structure, contents and appearance. One uses the language to design a document template, containing formatting commands for one's desired word processor,[7] interleaved with instructions to incorporate information from the model. These instructions activate queries in the query language to retrieve information, or routines to extract graphical charts, and then format these according to the template. A document generation program can therefore be prepared once, in advance, and can then be run whenever the document is needed. The templates for some particular documents have been prepackaged, and are available ready-made to the user. They include the main parts of the US DoD Standards 2167 and 2167A. Programmed documents too can be generated at any stage of the development, and for the complete model or portions thereof.

V. Executions and Dynamic Analysis

We now turn to the analysis capabilities of STATEMATE, which constitute one of its main novelties. In [13] we have tried to set out the underlying philosophy in some detail, emphasizing the analysis capabilities.

The heart of these is the ability to carry out a *step* of the SUD's dynamic behavior, with all the consequences taken into account. A step, briefly, is one unit of dynamic behavior, at the beginning and end of which the SUD is in some legal *status*. A status captures the system's currently active states and activities, the current values of variables and conditions, etc. During a step, the environment activities can generate external events, change the truth values of conditions, and update variables and other data items. Given the potentially intricate form that a STATEMATE description of the SUD might take on, such changes can have a profound effect on the status, triggering transitions in statecharts, activating and deactivating activities, modifying other data items, and so on. Clearly, each of these changes, in turn, may cause many others.

The portion of STATEMATE that is responsible for calculating the effect of a step contains involved algorithmic procedures, which reflect the formal semantics that have been defined mathematically for the modeling languages. The particular semantics of statecharts that has been adopted is described in [14], and is somewhat different from that described in [7], although on most standard examples the two are equivalent.

The most basic way of "running" the SUD is in a step-by-step interactive fashion. At each step the user generates external events, changes conditions and carries out other actions (such as changing the values of variables) at will, thus emulating the environment of the system. All of these are assumed to have occurred within a single step, the most recent one. When the user then gives the "go" command, STATEMATE responds by transforming the SUD into the new resulting status. Typically, there will be one or more statecharts on the screen while this is happening, and often also an activity-chart. The currently active states and activities will be highlighted with special coloring.[8]

This ability to run through dynamic scenarios has obvious value as a debugging mechanism in the specification stage. If we find that the system's response is not as expected we go back to the model, change it (by modifying a statechart, for example), and run the same scenario again.

At times, however, we want to be able to see the model executing noninteractively, and under circumstances that we do not care to spell out in detail ourselves. We would like to see it perform under random conditions, and in both typical and less typical situations. This more powerful notion of executing the model is achieved by the idea of *programmed executions*. To that end, a specially tailored *simulation control language* (SCL) has been designed and incorporated into STATEMATE, enabling the user to retain general control over how the executions proceed, yet exploit the tool to take over many of the details.

Programs in SCL look a little like conventional programs in a high-level language; they employ variables and support several control structures that can be combined and nested. They are used to control the simulation by reading events and changes from previously prepared files, and/or generating them using, say, random sampling from a variety of probability distributions. Several kinds of *breakpoints* can be incorporated into the program, causing the execution to stop and take certain actions when particular situations come up. These actions can range from incrementing counters (e.g., to accumulate statistics about performance), through switching to interactive mode (from which the user can return to the programmed execution by a simple command), and all the way to executing a lengthy calculation constituting the inners of a basic activity that was left unspecified when modeling the SUD.

[7] Several standard word processors are supported.

[8] Actually, the system will highlight only those states and activities that are on the lowest level visible.

Executions can thus be stopped and restarted, and intervening changes can be made; the effects of events generated with prescribed probabilities can be checked, and the computational parts of the SUD and its environment can be emulated. Moreover, during such simulated executions a *trace database* is maintained, which records changes made in the status of the SUD. The trace database can later be reviewed, filed away, printed or discarded, and, of course, is important for inspecting the execution and its effects off-line. A variety of *simulation reports* can be produced, in which parts of the information are gathered as the execution proceeds, via instructions in the SCL program, and other parts are taken from the trace database after the execution ends. Moreover, we may view the progress of a programmed execution graphically just as in the interactive case; the same color codes are used to continuously update the displayed charts. The result is a visually pleasing discrete animation of the behavior of the SUD.[9]

The part of the SUD that is simulated (in either interactive or programmed mode) can be restricted in scope. For example, one can simulate an activity and its inners, and the rest of the STATEMATE specification is considered to be nonexistent for the duration of that simulation. Moreover, there is no need to wait until the entire SUD is specified before initiating executions and simulations; a user can start simulating, or running, a description from the moment the portion that is available is syntactically intact (and this can be checked by the static tests). In the simulation the user will typically provide those events and other items of information that are external to the specified portion, even though later they might become internal to the complete specification.

In general, then, a carefully prepared SCL program can be used to test the specification of the SUD under a wide range of test data, to emulate both the environment and the as-of-yet unspecified parts of the SUD, to check the specification for time-critical performance and efficiency, and, in general, to debug it and identify subtle run-time errors. Needless to say, the kinds of errors and misconceptions that can be discovered in this way are quite different from the syntactic completeness and consistency checks that form the highlights of most of the other available tools for system design, and which STATEMATE carries out routinely.

It is important to keep in mind that the role of the SCL programs is to oversee the execution of the model; they are not intended to replace it. Thus, SCL is not a modeling language but a meta-language that serves as a vehicle for some of our analysis capabilities. It should not be compared with simulation languages in the sense that term is often used, where the programs themselves constitute the model.

Now, since STATEMATE can fully execute steps of dynamic behavior, and since SCL programs can be written to control the execution of many scenarios, it becomes tempting to provide the ability to execute *all* scenarios—as long as the number of possibilities is manageable—in order to test for crucial dynamic properties. STATEMATE has been programmed to provide a number of these dynamic tests, all of which proceed essentially by carrying out exhaustive, brute-force, sets of executions. They include *reachability, nondeterminism, deadlock,* and *usage of transitions*. For the first of these, given an initial configuration and a target condition, the test seeks sequences of external events and other occurrences that lead from the initial status to one that satisfies the condition, producing these sequences if they exist and stating that there are none otherwise. It is important to stress that this is run-time, dynamic, reachability, not merely a test for whether two boxes in some diagram are connected by arrows. The same applies to the other dynamic tests too.

One must realize, however, that even if we limit the values of variables to finite sets, the number of scenarios that have to be tested in an exhaustive execution quickly becomes unmanageable. This means that unless the portion of the model that we are testing is sufficiently small and has only a few external connections, we will not always be able to complete our exhaustive test. Indeed, these dynamic tests should be used only on very critical, well isolated parts of our model. When larger parts require exhaustive testing, we may limit the scope of the test by instructing it, for example, to ignore some of the external events, or to avoid simulating the details of certain activities. We have used the reachability test successfully on a number of occasions. In one real-world situation, when analyzing part of the specification of the trigger mechanism in a certain deployed missile system, our reachability test discovered a new sequence of events (that was unknown to the design team!) leading to the firing of a missile.

The reachability test can be used in a more sophisticated way, by attaching *watchdog* statecharts to the model being tested. Thus, we can test whether it is possible to reach situations of temporal, dynamic nature, by adding a watchdog statechart that enters a special state S when the situation in case arises. A reachability test is then run on the original statechart with the new one added as an orthogonal component, and the condition being sought for is specified to be **in**(S).

An additional feature that is planned for a future version is the ability to verify a STATEMATE specification against a formula in temporal logic.

VI. Code-Generation and Rapid Prototyping

Once a model of the SUD has been constructed, and has been executed and analyzed to the satisfaction of the designer, STATEMATE can be instructed to translate it automatically into code in a high-level programming language. This is analogous to the compilation of a program in a high-level language, whereas the executability of the model is analogous to its interpretation. Currently, trans-

[9] There are other visual tools that support animated executions. See, for example, [3].

lations into Ada and C are supported. Technically, any activity-chart (together with its controlling statecharts) can be translated, which, again, means that one need not wait until the entire model is ready but can produce code from portions thereof. If code was supplied by the bottom-level basic activities, it can be appropriately linked to the generated code, resulting in a complete running version of the system.

We term the result *prototype code*, since it is generated automatically, and reflects only those design decisions made in the process of preparing the conceptual model. It may thus not be as efficient as final real-time code, though it runs much faster than the executions of the model itself, just as compiled code runs faster than interpreted code. Future plans call for enhancing the code generator with the ability to incorporate decisions made interactively by the human designer, as well as with various further optimization features. We might add that an interesting way to further exploit STATEMATE for analyzing the model is to construct special statecharts, which are not part of the model itself, and whose role is to test the model. Of course, for these test suites (and also for the watchdog statecharts described earlier) the output from the code-generation is actually final code.

One of the main uses of the prototype code is in observing the SUD performing in circumstances that are close to its final environment. The code can be ported and executed in the actual target environment, or—as is more realistic in most cases—in a simulated version of the target environment. Often we have linked the prototype code with "soft" panels, graphical mock-ups of control panels, dials, gauges, etc., that represent the actual user interface of the final system. These panels appear on the screen and are manipulated with the mouse and keyboard. Unlike conventional prototypes of such systems, however, here the soft panels are not driven by hastily-written code prepared especially for the prototype, but by code generated automatically from the STATEMATE model, which typically will have been thoroughly tested and analyzed before being subjected to code-generation. The idea is to use this feature for goals that go beyond the development team. We envision mock-ups of the SUD driven by our prototype code being used as part of the communication between customer and contractor or contractor and subcontractor. It is not unreasonable to require such a running version of the system to be one of the deliverables in certain development stages, such as the preliminary design review.

Associated with the code-generation facility is a debugging mechanism, with which the user can trace the executing parts of the code back up to the STATEMATE model. Breakpoints can be inserted to stop the run of the code when chosen events occur, at which point one may examine the model's configuration (states, activities, etc.) and modify elements (conditions, data-items, etc.), prior to resuming the run. Of course, if substantial problems arise in the running of the code, changes can be made in the STATEMATE model itself, which is then recompiled down into Ada or C, and rerun. As in simulations, trace files can be requested, in which the changes in desired elements can be recorded. Continuing the analogy between conventional compilation and our generation of code from a STATEMATE model, this debugging facility might be termed *source-level* debugging.

Finally, the code-generation facility can be used for bringing the model gradually closer to a final software implementation. This is done by *incremental substitution*, whereby increasingly larger parts of the system are replaced by code, the process being interleaved with the making of design decisions. This procedure, which we hope to discuss more fully in a subsequent paper, is different from conventional integration in that the medium is changed (from conceptual model to code) as the integration is being carried out. As a consequence, there is a need for testing and validation in intermediate steps, much of which can be carried out in STATEMATE.

In the future, we plan to enrich the code-generator with the ability to yield VHDL code. This will enable hardware designers to use STATEMATE not only for the specification and early design stages, but also for the later stages. Silicon compilation would then be carried out from code that is generated automatically from a STATEMATE specification.

VII. Conclusions

In conclusion, we might say that the STATEMATE system combines two principles, or theses, that we feel should guide future attempts to design support tools for system development. The first is the long-advocated need for *executable specifications*, and the second is the advantage of using *visual formalisms*.

As far as the first of these goes, the development of complex systems must not be allowed to progress from untested requirements or specifications. Rather, ways should be found to model the SUD on any desired level of detail in a manner that is fully executable and analyzable, and which allows for deep and comprehensive testing and debugging, of both static and dynamic nature, prior to, and in the process of, building the system itself. We might add that the dynamic analysis capabilities of STATEMATE go far beyond what is normally taken as the meaning of the term executable specification, i.e., the simple ability to animate a diagram in a step-by-step fashion.

As to the second principle, we believe that visual formalisms will turn out to be a crucial ingredient in the continuous search for more natural and powerful ways to exploit computers. It is our feeling that the progress made in graphical hardware, combined with the capabilities of the human visual system, will result in a significant change in the way we carry out many of our complex engineering activities. The surviving approaches will be, we believe, of diagrammatic nature, yet will be formal and rigorous, in both syntax and semantics.

Acknowledgment

We would like to thank J. Lavi and his group at the Israel Aircraft Industries for their suggestions, their time, and their constructive criticism during the lengthy period in which the STATEMATE system was being developed. In a way, this project would not have gotten started had Dr. Lavi not lured the first-listed author into consulting for IAI in early 1983. This action led to the invention of statecharts in mid-1983, and to the decision to form Ad-Cad Ltd. and to start work on STATEMATE in early 1984.

We are grateful to all the technical staff members of Ad-Cad Ltd., past and present, who were indispensible in turning the ideas described here into a real working system. They include R. Arnan, E. Bahat, S. Barzilai, A. Bernstein, M. Cohen, R. Cohen, D. Falkon, A. Farjou, N. Fogel, L. Gambom, E. Gery, O. Hay, R. Heiman, M. Hirsch, R. Kazmirski, D. Levin, H. Libreich, R. Livne, A. Maimon, L. Maron, Y. Partosh, S. Pnueli, Y. Pnueli, A. Polyack, J. Prozan-Schmidt, Y. Rubinfain, A. Sarig, R. Shaprio, A. Sharabi, I. Shimshoni, M. Trachtman, Y. Yochai, and B. Yudowitz. In addition, I. Lachover deserves special thanks for being the most pleasant manager imaginable, contributing his experience and expertise to all phases of the work.

We would like to thank the Bird Foundation, and the office of the chief scientist of Israel's Ministry of Industry and Commerce for financial help. One of the referees provided many helpful comments on the penultimate version of this paper.

REFERENCES

[1] A. K. Chandra and D. Harel, "Structure and Complexity of Relational Queries," *J. Comput. Syst. Sci.*, vol. 25, pp. 99-128, 1982.

[2] A. K. Chandra and P. Merlin, "Optimal implementation of conjunctive queries in relational databases," in *Proc. 9th ACM Symp. Theory of Computing*, Boulder, CO, 1977, pp. 77-90.

[3] M. Graf, "Building a visual designer's environment," in *Principles of Visual Programming Systems*, S.-K. Chang, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1990, pp. 291-325.

[4] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231-274, 1987 (appeared in preliminary form as Rep. CS84-05, Weizmann Inst. Sci., Rehovot, Israel, Feb. 1984).

[5] ——, "On visual formalisms," *Commun. ACM*, vol. 31, no. 5, pp. 514-530, 1980.

[6] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. New York: Springer-Verlag, 1985, pp. 477-498.

[7] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *Proc. 2nd IEEE Symp. Logic in Computer Science*. New York: IEEE Press, 1987, pp. 54-64.

[8] D. J. Hatley, "A structured analysis method for real-time systems," in *Proc. DECUS Symp.*, Dec. 1985.

[9] D. J. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*. New York: Dorset, 1987.

[10] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

[11] W. S. Humphrey and M. I. Kellner, "Software process modeling: Principles of entity process models," in *Proc. 11th Int. Conf. Software Eng.*, Pittsburgh, PA. New York: IEEE Press, 1989, pp. 331-342.

[12] "The languages of STATEMATE," i-Logix Inc., Burlington, MA, Tech. Rep., 1987.

[13] "The STATEMATE approach to complex systems," i-Logix Inc., Burlington, MA, Tech. Rep., 1989.

[14] "The semantics of statecharts," i-Logix Inc., Burlington, MA, Tech. Rep., 1989.

[15] R. J. Lano, *A Technique for Software and Systems Design* (*TRW Series on Software Engineering*). Amsterdam, The Netherlands: North-Holland, 1979.

[16] J. Z. Lavi and E. Kessler, "An embedded computer systems analysis method," Manuscript, Israel Aircraft Industries, Nov. 1986.

[17] J. Z. Lavi and M. Winokur, "ECSAM—A method for the analysis of complex embedded systems and their software," in *Proc. Structured Techniques Assoc. Conf. STA5*, Univ. Chicago, Chicago, IL, May 1989, pp. 50-63.

[18] A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends," in *Current Trends in Concurrency (Lecture Notes in Computer Science*, vol. 224), de Bakker *et al.*, Eds. Berlin: Springer-Verlag, 1986, pp. 510-584.

[19] S. L. Smith and S. L. Gerhart, "STATEMATE and cruise control: A case^2 study," in *Proc. COMPAC '88, 12th Int. IEEE Comput. Software and Applicat. Conf.* New York: IEEE Press, 1988, pp. 49-56.

[20] P. Ward, "The transformation schema: An extension of the data flow diagram to represent control and timing," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 198-210, 1986.

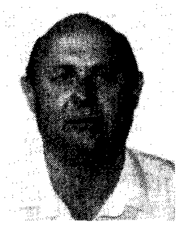
[21] P. Ward and S. Mellor, *Structured Development for Real-Time Systems*. New York: Yourdon, 1985.

[22] D. P. Wood and W. G. Wood, "Comparative evaluations of four specification methods for real-time systems," Software Eng. Inst., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU/SEI-89-TR-36, Dec. 1989.

David Harel (M'84) received the B.Sc. degree in mathematics from Bar-Ilan University in 1974, the M.Sc. degree in computer science from Tel-Aviv University in 1976, and the Ph.D. degree in computer science from the Massachusetts Institute of Technology in 1978.

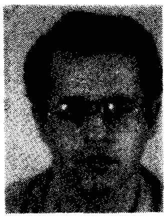
He is currently a Professor at the Weizmann Institute of Science in Israel, and Chairman of its Department of Applied Mathematics and Computer Science. He is also a co-founder and Chief Scientist of i-Logix, Inc., Burlington, MA. He has been on the research staff of IBM's Research Center at Yorktown Heights, NY, and a Visiting Professor at Carnegie-Mellon University's Department of Computer Science. His research interests include logics of programs, computability theory, systems engineering, and visual languages, topics in which he has published widely. He is on the editorial boards of *Information and Computation* and the ACM-Press/Addison-Wesley book series. His most recent book, *The Science of Computing: Exploring the Nature and Power of Algorithms*, was published by Addison-Wesley in 1989.

Dr. Harel is a member of the Association for Computing Machinery and the IEEE Computer Society.



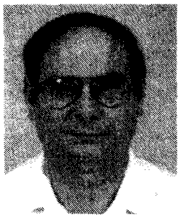
Hagi Lachover received the B.Sc. degree in applied mathematics from Tel-Aviv University in 1967.

In previous positions, he developed operating systems at the Weizmann Institute of Science and was Vice President for product development at Mini Systems, Ltd., a company that developed the software for Scitex, Inc., Herzelia, Israel. He is a co-founder and Vice President for Operations at Ad Cad, Ltd., the R&D subsidiary of i-Logix, Inc., Burlington, MA.



Amnon Naamad received the B.Sc. degree in mathematics and the M.Sc. degree in computer science from Tel-Aviv University in 1976 and 1979, respectively, and the Ph.D. degree in computer science from Northwestern University in 1981.

He is currently a Project Leader at Ad Cad, Ltd., the R&D subsidiary of i-Logix, Inc., Burlington, MA. He was responsible for the development of the simulation and dynamic analysis tools of the STATEMATE system, and is currently developing a translator from STATEMATE descriptions into hardware specification languages such as VHDL.



Amir Pnueli received the Ph.D. degree in applied mathematics from the Weizmann Institute of Science, Rehovot, Israel, in 1967.

He is currently a Professor of Computer Science at the Weizmann Institute of Science. He is also a co-founder and Chief Scientist of i-Logix, Inc., Burlington, MA. He has been a Chief Scientist at Mini Systems, Ltd., and a Visiting Professor at the Departments of Computer Science of Stanford University, Harvard University, and Brandeis University. His research interests include specification and verification of reactive systems, with a special emphasis on temporal logic, which he introduced into computer science in 1977. He has published widely on these topics. He is on the editorial boards of *Science of Computer Programming* and Springer-Verlag's Lecture Notes in Computer Science series.

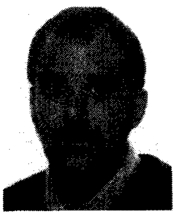
Dr. Pnueli is a member of the Association for Computing Machinery and the IEEE Computer Society.

Michal Politi received the B.Sc. degree in mathematics and physics from the Hebrew University in 1969, and the M.Sc. degree in computer science from the Weizmann Institute of Science.

She has had many years of experience in developing complex real-time systems in various places. Since 1986 she has been Product Manager for the STATEMATE system at Ad Cad, Ltd., the R&D subsidiary of i-Logix, Inc., Burlington, MA. Her interests are in methods and languages for the specification and design of real-time systems.

Rivi Sherman received the B.Sc. degree in mathematics from Tel-Aviv University in 1974, and the M.Sc. degree and Ph.D. degrees in computer science from the Weizmann Institute of Science in 1978 and 1984, respectively.

She is currently on the technical staff of Orbot, Inc., Yavne, Israel, and has spent three years as a researcher at the University of Southern California/Information Sciences Institute in Los Angeles. She was the first project leader at Ad Cad, Ltd., the R&D subsidiary of i-Logix, Inc., Burlington, MA, from 1984 to 1986, and was responsible for the development of the first version of the STATEMATE system.



Aharon Shtull-Trauring received the B.A. degree in urban studies from Columbia University in 1975, and the M.Sc. degree in public management from Carnegie-Mellon University in 1979.

For many years he worked in software development, particularly of database systems. From 1984 to 1988 he was one of the project leaders at Ad Cad, Ltd., the R&D subsidiary of i-Logix, Inc., Burlington, MA, where he was responsible for the database and systems aspects of the STATEMATE system. He is now involved in the international marketing of STATEMATE.

Mark Trakhtenbrot received the M.Sc. degree in computer science from the University of Novosibirsk in 1971, and the Ph.D. degree from the Kiev Institute of Cybernetics in 1978.

He has been a Project Manager at Mayda, Ltd., Rehovot, Israel, developing and implementing an Ada-based PDL. He is currently a project leader at Ad-Cad, Ltd., the R&D subsidiary of i-Logix, Inc., Burlington, MA, where he was responsible for the development of the prototyping and code-generation capabilities of STATEMATE. His interests are in methodologies and tools for software and systems engineering.