PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 17/27, 17/30

A1

(11) International Publication Number: **WO 98/29817**

(43) International Publication Date: 9 July 1998 (09.07.98)

(21) International Application Number: PCT/IL97/00418

(22) International Filing Date: 22 December 1997 (22.12.97)

(30) Priority Data:
119914 25 December 1996 (25.12.96) IL

(71) Applicant *(for all designated States except US)*: EMULTEK LTD. [IL/IL]; Beit Rabin, Teradion Industrial Park, 20179 Misgav (IL).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: ROTBART, Frederick, C. [IL/IL]; 142 Yuvalim Street, 20142 Misgav (IL).

(74) Agents: COLB, Sanford, T. et al.; Sanford T. Colb & Co., P.O. Box 2273, 76122 Rehovot (IL).

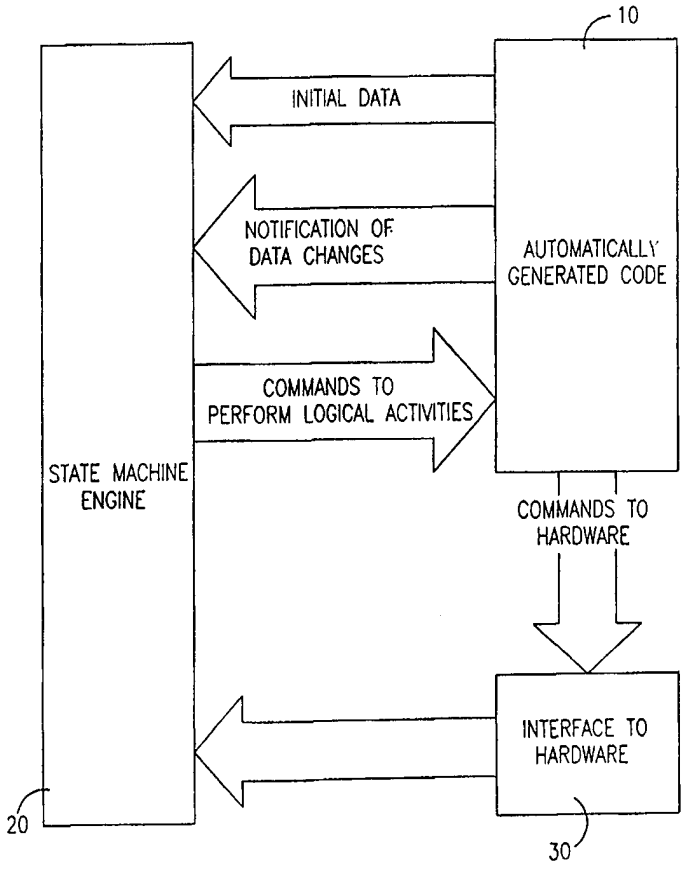
(81) Designated States: AL, AM, AT, AT (Utility model), AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, CZ (Utility model), DE, DE (Utility model), DK, DK (Utility model), EE, EE (Utility model), ES, FI, FI (Utility model), GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SK (Utility model), SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: DEVICE FOR IMPLEMENTING HIERARCHICAL STATE CHARTS AND METHODS AND APPARATUS USEFUL THEREFOR

(57) Abstract

An apparatus allows users to implement hierarchical state charts. The apparatus includes a state machine engine (SME) (20) operative to carry out repertoires of behaviors of a system, a state chart encoder (10) operative to replicate, in computer code form, a user-defined hierarchical state chart describing a repertoire of behaviors of the system, the encoder being operative to preserve the state chart's hierarchical structure intact in a first data structure, an event list generator operative to generate, for each individual event from among a plurality of events within the repertoire, a list of at least one transition between states of the state chart which are associated with the individual event, the event list generator being operative to preserve the list in a second data structure, and a behavior implementer operative to activate the system to carry out each of the behaviors in its repertoire.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

DEVICE FOR IMPLEMENTING HIERARCHICAL STATE CHARTS AND METHODS AND APPARATUS USEFUL THEREFOR

FIELD OF THE INVENTION

The present invention relates to computer code and to apparatus and methods for computer code generation.

BACKGROUND OF THE INVENTION

Hierarchical state charts are described in "Statecharts, a visual approach to complex systems", David Harel, Science of Computer Programming, 1987.

The disclosures of all publications mentioned in the specification and of the publications cited therein are hereby incorporated by reference.

SUMMARY OF THE INVENTION

The present invention seeks to provide improved computer code and improved apparatus and methods for computer code generation.

There is thus provided in accordance with a preferred embodiment of the present invention apparatus implementing hierarchical state charts and including a state machine engine (SME) operative to carry out repertoires of behaviors of a system, a state chart encoder operative to replicate, in computer code form, a user-defined hierarchical state chart describing a repertoire of behaviors of the system, the encoder being operative to preserve the state chart's hierarchical structure intact in a first data structure, an event list generator operative to generate, for each individual event from among a plurality of events within the repertoire, a list of at least one transition between states of the state chart which are associated with the individual event, the event list generator being operative to preserve the list in a second data structure, and a behavior implementer operative to activate the system to carry out each of the behaviors in its repertoire.

Further in accordance with a preferred embodiment of the present invention the system includes at least one of hardware and software.

Still further in accordance with a preferred embodiment of the present invention the first data structure's size increases linearly in direct proportion to the number of branches in the hierarchical structure.

Additionally in accordance with a preferred embodiment of the present invention the second data structure's size increases linearly in direct proportion to the number of transitions in the list of at least one transition.

Moreover in accordance with a preferred embodiment of the present invention the order of the at least one transition of each of the lists corresponds to the order of evaluation of the transitions as defined by the hierarchical state chart.

Further in accordance with a preferred embodiment of the present invention the plurality of events for which the event list generator generates lists includes at least one external input event.

Still further in accordance with a preferred embodiment of the present invention the plurality of events for which the event list generator generates lists includes at least one internal data value change event, the apparatus also including a data value change event notifier operative to generate a notification to the state machine engine of each data value change event.

There is additionally provided in accordance with a preferred embodiment of the present invention a method implementing hierarchical state charts, the method including providing a state machine engine (SME) operative to carry out repertoires of behaviors of a system replicating, in computer code form, a user-defined hierarchical state chart describing a repertoire of behaviors of the system, the replicating step being operative to preserve the state chart's hierarchical structure intact, generating, for each individual event from among a plurality of events within the repertoire, a list of at least one transition between states of the state chart which are associated with the individual event, and activating the system to carry out each of the behaviors in its repertoire.

There is additionally provided in accordance with a preferred embodiment of the present invention an automatic code generation method including automatically generating code which forms, in memory, a replica of a hierarchical state chart which preserves the chart's hierarchical structure, automatically generating code which forms, for each of a first plurality of input events, a list of transitions affected by that input event, automatically generating code

which forms, for each of a second plurality of internal data variables, a list of transitions affected by changes in that internal data variable, automatically generating code which notifies the state machine engine of changes in internal data variables, and automatically generating code which evaluates conditions and implements actions and activities.

Further in accordance with a preferred embodiment of the present invention the method includes the step of providing a state machine engine operative, upon receipt of notification regarding occurrence of an event, to evaluate a list of transitions affected by the event so as to select transitions to be performed, and to perform the transitions.

There is additionally provided in accordance with a preferred embodiment of the present invention a method for computing a state for a hierarchical state machine, the method including providing a state machine engine operative to receive notification regarding occurrence of at least one event, providing a hierarchical state chart describing a repertoire of behaviors of a system, providing a list of at least one transition between states of the state chart for each of the events with which the at least one transition is associated, evaluating the list of at least one transition so as to select a transition to be performed in response to the at least one event, and evaluating the hierarchical state chart to compute the state that corresponds to the selected transition.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be understood and appreciated from the following detailed description, taken in conjunction with the drawings in which:

Fig. 1 is a simplified block diagram of a software system constructed and operative in accordance with a preferred embodiment of the present invention which includes automatically generated code which interfaces with a state machine engine (SME);

Fig. 2 is a simplified flowchart illustration of a preferred method of operation for the state machine engine of Fig. 1;

Fig. 3 is a simplified flowchart illustration of a preferred method for generating high level computer code from hierarchical state charts;

Fig. 4 is a pictorial illustration of a computer screen generated in the course of automatically generating computer code from hierarchical state charts in a particular example;

Fig. 5 is a pictorial illustration of a computer screen, generated in the course of processing the example of Fig. 4, which shows all modes making up the state chart of the example;

Fig. 6 is a pictorial illustration of a computer screen, generated in the course of processing the example of Fig. 4, which shows an alternative presentation of the state chart of Fig. 5; and

Fig. 7 is a simplified flowchart illustration of an automatic code generation technique operative in accordance with a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Reference is now made to Fig. 1 which is a simplified block diagram of a software system constructed and operative in accordance with a preferred embodiment of the present invention which includes automatically generated code 10 which interfaces with a state machine engine (SME) 20. An optional interface 30 to underlying hardware (not shown) is also provided.

The state machine engine 20 is a pre-built object code module that is linked, either dynamically or statically, to the automatically generated code and the interface library.

The state machine engine implements the behavior of the software system preferably as defined solely by the automatically generated code that is linked with it. That being the case, the state machine engine code is typically provided only once in object code form, and does not need to be changed per software system.

The state machine implements this behavior by converting input events and changes of data values into transitions within the hierarchical state chart, and then implementing these transitions. It does this by computing the new state associated with this transition, deactivates the relevant parts of the old state and finally activates the relevant parts of the new state.

A preferred method of operation for the state machine engine (SME) 20 is described by the flow chart of Fig. 2.

The SME begins operation by building initial data structures based on data provided by the automatically generated code 30. These data structures typically include:

- a data structure that is a one to one representation of the hierarchical state chart,

- a data structure that couples together any input events with the transitions that are triggered by these input events,
- a data structure that couples together any data variables in the application with transitions that are triggered by changes in the value or state of these data variables, and with any expressions that must be evaluated when these data variables change, and
- a pointer to all logic and activity functions within the automatically generated code.

The SME then enters its normal operating state, which is to wait for any input events or any changes in data values. It is appreciated that input events may originate from a hardware device, such as a lamp as described in greater detail hereinbelow with reference to Fig. 4, or from a software process, such as an interprocess message.

When an input event notification arrives or a data change notification arrives from the automatically generated code, SME 20 checks if the event or data structure that triggered the notification has a list of transitions or logic expressions associated with it. If not it discards the notification and returns to its normal operating state.

If the source of the notification does have such a list associated with it, the SME then checks each entry in the list and selects those entries that belong to a state that is currently active. All other entries are ignored.

The SME then iterates over the selected list and checks each entry.

If an entry is a transition, and the source state of that transition is still active, the SME performs the transition by computing the new state associated with the transition, deactivating those parts of the current state that will no longer be active in the new state, and then activating all those elements in the new state that were not active in the current state. The new state then becomes the current state.

If the entry is a logic expression, rather than a transition, the SME evaluates this expression by calling the corresponding functions in the generated code.

Once the SME has processed all entries in the list, it returns to its normal operating mode to await the next notification.

Referring back to Fig. 1, the automatically generated code 10 is a translation of a hierarchical state chart description of a system, in addition to all data elements in the system, which, when compiled and run together with the SME and the interface library, implements the behavior of the original system.

The automatically generated code 10 typically comprises some or all of the following components:

- code to generate in memory, on a one to one basis, a replica of the original state chart, including the transitions and activities as defined,
- code to generate the lists that link sources of input events with the transitions to be evaluated when notification of these events are received by the SME,
- code to generate the lists that link data variables with the transitions and the logical expressions to be evaluated when notification of changes to these data variables are received by the SME,
- code to generate the notifications to the SME whenever a data variable changes,
- code to evaluate the conditions and to implement actions and activities that describe the behavior of the system described by the original state chart, including code to activate elements of hardware or software as specified by the state chart,
- code to deliver to the SME any of the above as is needed by it.

The object code interface library 30 to the underlying hardware is optional pre-built object code that allows one or both of the state machine and the automatically generated code to interact with the hardware on which these elements operate.

This object code interface library is specific to the hardware on which the system is to be run.

Conventional technology for generating high level code from state charts has entailed generating a static representation of all the possible transitions and states as described by the state chart, as well as generating code for the state machine that activates these various states. In general, the state and the state machine are inseparable in such generated code.

The two main techniques available for accomplishing this have been: state machines built of nested branch statements and state machines built around state transition tables.

In the first technique, code generation using branch statements, the state chart formalism is converted into a large number of typically deeply nested branch statements, where each branch statement represents one of the possible transitions. The depth of nesting is at least as deep as the deepest hierarchy in the state chart and increases non-linearly as a function of the depth of all branches in the state chart, the width of the branches and the number of

possible input events.

Branch statements are statements of the type "if-else" or "switch-case".

For simple programs this may be adequate, but in general, because of the large amount of code needed to code the nested branch statements, this technique produces overly large programs.

Every possible input event is typically represented by some sort of flag. When an input event occurs, the corresponding flag is set to indicate this. The state machine mechanism involves continuously cycling through all the branch statements in the search for any branches that involve flags which may have been set before the current cycle as well as any condition statements that may have become true due to a variable changing value. This has the inevitable result that such code is very slow, especially since every cycle has to go through the deeply nested branch points that make up the code. This makes this type of code big and too slow for any application that requires a fast response time or modest memory requirements.

In the second technique, code generation using state transition tables, a procedure or function is generated for every possible transition and state, together with a state transition table that is used by the generated state machine for choosing which procedure to call for every possible external input. This state transition table is, in effect, a two-dimensional array whose indices are "input event" and "current state" and whose value is the new state.

The state here is not a hierarchical state, but rather a state in a normal linear state diagram, which is derived from the state chart by "flattening out" the original state chart.

When an input event arrives at the state machine, the state machine uses it, together with the current state to look up the next state. This use of tables results in much better response times than the nested branch statements to input. On the other hand, condition statements must still be evaluated by cycling through all such statements in search of any that evaluates to "true." This, of course, is still very slow.

In addition, because of the need to "flatten out" or linearize the original state chart, the size of the table explodes exponentially as a function of the number of possible input events, of depth of the state chart and the width of each branch. Hence, programs generated this way are normally so large as to make the generated code impractical for commercial products.

In contrast, the code generation technique of the present invention preferably has

some or all of the following features:

- It does not try to reproduce the hierarchical state machine in the generated code, but rather generates code that, after compilation and linkage, is run by a pre-built, optimized state machine engine.
- It does not generate a static representation of all possible states and the transitions in the state chart, but rather generates code that reproduces the actual hierarchical state structure of the original state chart. The pre-built state machine engine then dynamically computes the required new state in response to every input event, or change in conditions.
- It is completely event driven, even in the case of conditions. Every source of events and every variable can have a list of logic associated with it. This list points to any statement that must be evaluated if and when that event is triggered or that variable changes value. In general, the statements in the list represent transitions, and the order of the statements expresses the order of evaluation of these transitions. As soon as an input event arrives, or a variable changes value, the transition dependent upon it is immediately identified and evaluated.
- The size of the generated code is linear in the number of branches and the number of transitions between the branches. The size is independent of the width of the branches and the number of possible input events.

The code generation technique of the present invention and the code structure of the present invention preferably has some or all of the following advantages:

- The resultant generated code, together with the pre-built state machine engine, is very small, and has memory requirements which are comparable to equivalent code developed manually by programmers.
- Whereas response time of the generated code to input events is as good as conventional automatically generated code, the response time to changes in variables is better than that for automatically generated code, and in fact, is similar to that for input events.
- The code generated is guaranteed to reflect the state chart from which it was derived, rather than being a transformed version of it. As a result, the generated code behaves as described by the original state chart. In addition, because of the clarity of the state chart formalism itself, the generated code is more likely to behave as required, compared to the

equivalent manually developed code.

EXAMPLE: Fig. 3 illustrates incorporation of a system for generating the code shown and described herein into the Rapid S/W tool, commercially available from Emultek Ltd., Misgav 20179, Israel.

The steps of Fig. 3 are now described:

Build Interface Library: The interface library is used by the automatically generated code to interact with the hardware on which the system to be developed runs. For example, the library code may be used to request memory allocation from the operating system, or, if the hardware controls a device such as a lamp, the library might contain the code to switch the lamp on and off, e.g., by accessing certain addresses on the hardware. The library can be written in any convenient language and is preferably pre-tested before development with Rapid begins.

Map Components to Simulation Objects: In Rapid, applications are built by assembling simulations of the hardware and/or software components on the screen and building logical expressions from functions supplied with the object simulations. These supplied functions animate the behavior of the object simulations. For example in the case of a Lamp object, the function "switchOn" would cause the simulated lamp on the screen to change to the "on" color. In order for the logical expressions be able to interact with the actual components, a mapping between the supplied functions and interface library is provided. This mapping may be hard coded into the simulated objects code. Alternatively, an external textual mapping file may be used.

Open a New Rapid Application: This is the first step in building Rapid applications, providing a "clean slate" on which to build the application.

Build Simulation of the System: The simulation is built by dragging objects from a tool palette in the Layout Editor and positioning them as required on the graphic representation of the device being simulated. Each object can be graphically customized and, to some extent, functionally customized.

Rapid State Charts: The logic in a Rapid application is described by a state chart. State charts in Rapid are built from units called "modes". A mode is a functional state of the system and any mode can be hierarchically nested within any other mode. Therefore, the modes describe a hierarchical tree. Modes come in two "flavors": exclusive and concurrent. For

exclusive modes, only one sibling can be in the active state at any given time, while for concurrent modes all siblings must be active together. When a particular mode is active, this implies that all its ancestors are active. If the mode's children are exclusive modes, this also implies that at least one of its children are active, while if the mode's children are concurrent modes, this implies that all of the children are active. A state in a state chart is the list of all currently active modes. A mode can have lists of activities and transitions. Activities describe what the system does when it is a particular state. Activities can be entry activities, exit activities or mode activities. Entry activities are activities that are performed when the mode that contains them becomes activate. Exit activities are activities that are performed when the mode that contains them becomes inactive. Mode activities are activities that are performed when the mode becomes active and that, as long as the mode is still active, are automatically re-evaluated whenever data variables that describe them change value. Transitions define how the system evolves from one state to another. Transitions typically have a destination mode. Transitions must also have a list of input events, or a condition expression or both. Transitions may also have a list of actions. If notification of one of the input events held by a transition is received, and that transition belongs to a currently active mode, and the condition expression, if any evaluates to true, then the mode belonging to the transition is deactivated (together with its children and any relevant ancestors) and the destination mode of the transition is activated (together with its children etc.). Between the deactivation of the original mode and the activation of the destination mode, any actions in the transition are performed.

Build the State Chart: The simulation is typically built using two tools in Rapid: the Mode Tree Editor and the Logic Editor. The Mode Tree Editor allows the user to define and add modes to the hierarchical tree of modes by selecting the parent of the new mode, specifying the type of the new mode, and giving a name to the new mode. During this process, the transitions and activities can be added and edited using the Logic Editor. As each logic expression is added to the state chart it is compiled into byte codes for a virtual machine that is used to run a simulation of the system.

Run Simulation: At any time during the development process, the simulation of the system under development can preferably be run. This allows incremental development of the system and verification that the system behaves as desired.

Generate Code: Once the developer is satisfied with the system, Rapid

automatically generates C++ code for that system having the characteristics described above. It is appreciated that programming languages other than C++ may be used for the automatically generated code.

Compile the Generated Code: The automatically generated code can be compiled to produce object code.

Link with State Machine and Interface Library: The object code produced by compiling the automatically generated code is then linked with the SME object code and the interface library to produce executable code for the target system.

This executable code can be downloaded and run on the target system.

Sample Application:

The following is a sample application in Rapid and the code automatically generated for it:

Fig. 4 is a screen capture of the application in the Layout Editor, showing the object layout.

As can be seen the application contains five lamps, and five push buttons:

Objects

RAPID APPLICATION: TRANSITN 12/12/96 16:09:08
OBJECT DATA REPORT

Parameters:

Scope: Subtree.

Order: Hierarchy.

Content: Active objects, Nongraphic objects, Object parameters, Object properties.

1. OBJECT: transitn

   Parent: none Type: Root Object

   Parameters:

   Position (pxl): '0 @ 0' Size (pxl): '393 @ 403' Dynamic: 'false'

Drag 'n Drop: 'false'

2. OBJECT: LampH

Parent: transitn Type: Round Lamp

Parameters:

Center (pxl): '300 @ 139' Radius (pxl): '15 @ 15' Dynamic: 'false'

Drag 'n Drop: 'false'

Properties:

2.1. Name: blinkPeriod Type: Data

Parameters:

Value: 500

3. OBJECT: LampG

Parent: transitn Type: Round Lamp

Parameters:

Center (pxl): '243 @ 139' Radius (pxl): '15 @ 15' Dynamic: 'false'

Drag 'n Drop: 'false'

Properties:

3.1. Name: blinkPeriod Type: Data

Parameters:

Value: 500

4. OBJECT: LampA

Parent: transitn Type: Round Lamp

Parameters:

Center (pxl): '72 @ 140' Radius (pxl): '15 @ 15' Dynamic: 'false'

Drag 'n Drop: 'false'

Properties:

4.1. Name: blinkPeriod Type: Data

Parameters:

Value: 500

5. OBJECT: LampE

Parent: transitn Type: Round Lamp

Parameters:

Center (pxl): '129 @ 139' Radius (pxl): '15 @ 15' Dynamic: 'false'

Drag 'n Drop: 'false'

Properties:

5.1. Name: blinkPeriod Type: Data

Parameters:

Value: 500

6. OBJECT: LampF

Parent: transitn Type: Round Lamp

Parameters:

Center (pxl): '186 @ 139' Radius (pxl): '15 @ 15' Dynamic: 'false'

Drag 'n Drop: 'false'

Properties:

6.1. Name: blinkPeriod Type: Data

Parameters:

Value: 500

7. OBJECT: BToA

Parent: transitn Type: Flat Pushbutton

Parameters:

Position (pxl): '57 @ 199' Size (pxl): '40 @ 40' Dynamic: 'false'

Drag 'n Drop: 'false' Momentary: 'true' Autorepeat: 'false'

Autorepeat period: 500

8. OBJECT: Default

Parent: transitn Type: Flat Pushbutton

Parameters:

Position (pxl): '131 @ 199' Size (pxl): '40 @ 40' Dynamic: 'false'

Drag 'n Drop: 'false' Momentary: 'true' Autorepeat: 'false'

Autorepeat period: 500

9. OBJECT: HIstory

Parent: transitn Type: Flat Pushbutton

Parameters:

Position (pxl): '206 @ 199' Size (pxl): '40 @ 40' Dynamic: 'false'

Drag 'n Drop: 'false' Momentary: 'true' Autorepeat: 'false'

Autorepeat period: 500

10. OBJECT: DeepHistory

Parent: transitn Type: Flat Pushbutton

Parameters:

Position (pxl): '276 @ 199' Size (pxl): '40 @ 40' Dynamic: 'false'

Drag 'n Drop: 'false' Momentary: 'true' Autorepeat: 'false'

Autorepeat period: 500

11. OBJECT: History

Parent: transitn Type: Label

Parameters:

Position (pxl): '198 @ 244' Size (pxl): '58 @ 15' Dynamic: 'false'

Drag 'n Drop: 'false' Text: 'History' Orientation: 'Horizontal'

Font: 'Fixedsys' Line width: 1

12. OBJECT: Next

Parent: transitn Type: Flat Pushbutton

Parameters:

Position (pxl): '167 @ 282' Size (pxl): '40 @ 40' Dynamic: 'false'

Drag 'n Drop: 'false' Momentary: 'true' Autorepeat: 'false'

Autorepeat period: 500

Fig. 5 is a screen capture of the Mode Tree Editor showing all the modes that make up the state chart of this sample application. As can be seen there are nine modes in this application.

An alternative view of the state chart can be seen in the screen capture of the Logic Chart tool shown in Fig. 6. Fig. 6 shows each mode as a rectangle and each transition as an arc. The hierarchical nature of the state is clearly seen.

Reference is now made to Fig. 7 which is a simplified flowchart illustration of a automatic code generation technique operative in accordance with a preferred embodiment of

the present invention. The resulting code is useful in conjunction with a state machine engine which performs the following functions, inter alia:

a. Upon receipt of notification that an input event has occurred, the SME accesses the list formed for that input event and evaluates each transition on that list to determine, depending on the current state, which transitions are to be performed.

b. Upon receipt of notification that an internal data variable has been changed, the SME accesses the list formed for that internal data variable and evaluates each transition on that list to determine, depending on the current state, which transitions are to be performed.

To perform a transition, the SME computes the new state associated with the transition by going to the destination of the transition and looking up and activating all ancestors. All children are also looked up and activated as appropriate, e.g. in accordance with predefined defaults. Those portions of the current state that are no longer active in the new state are deactivated. Elements in the new state that were not active in the current state are activated. The new state then becomes the current state and is stored in the SME.

Alternatively, one or both of the above lists may not be provided. For example, the system may poll periodically for internal events rather than being driven thereby. Alternatively or in addition, the system may poll periodically for external events rather than being driven thereby.

Logic in State Chart

RAPID APPLICATION: TRANSITN 12/12/96 16:07:32
MODE DATA REPORT

Parameters:
Scope: Subtree.
Order: Hierarchy.
Includes: User Functions, Activities, Transitions, Triggers, Actions.

USER FUNCTIONS

1. MODE: transitn

   Parent: none Type: root

   Note:

2. MODE: A

   Parent: transitn Type: exclusive

   Entry Activities:

   a. LampA on

   Exit Activities:

   a. LampA off

   Transitions:

   a. Destination: B Entry type: Default

      a1. Trigger: Default in &

   b. Destination: B Entry type: History

      b1. Trigger: HIstory in &

   c. Destination: B Entry type: Deep history

      c1. Trigger: DeepHistory in &

3. MODE: B

   Parent: transitn Type: exclusive

   Transitions:

   a. Destination: A Entry type: Default

      a1. Trigger: BToA in &

4. MODE: C

   Parent: B Type: exclusive

5. MODE: E

Parent: C Type: exclusive

Entry Activities:

a. LampE on

Exit Activities:

a. LampE off

Transitions:

a. Destination: F Entry type: Default

a1. Trigger: Next in &

6. MODE: F

Parent: C Type: exclusive

Entry Activities:

a. LampF on

Exit Activities:

a. LampF off

Transitions:

a. Destination: G Entry type: Default

a1. Trigger: Next in &

7. MODE: D

Parent: B Type: exclusive

8. MODE: G

Parent: D Type: exclusive

Entry Activities:

a. LampG on

Exit Activities:

a. LampG off

Transitions:

a. Destination: H Entry type: Default

a1. Trigger: Next in &

9. MODE: H

Parent: D Type: exclusive

Entry Activities:

a. LampH on

Exit Activities:

a. LampH off

Transitions:

a. Destination: E Entry type: Default

a1. Trigger: Next in &

Generated Code

Header File

```
#ifndef __include_transitn
#define __include_transitn

#include "applic.h"
#include "task.h"
```

```
#include "object.h"
#include "mode.h"

class Application;
class myApplication;

typedef bool (myApplication:: *MyApplicationMethod)() ;

#define cNumberDependencies     8
#define cNumberDependencyIndices 6
#define cNumberChildren         18
#define cNumberObjects          20
#define cNumberTriggers         5
#define cNumberTransitions      8
#define cNumberModes            9
#define cNumberActivities       10

class myApplication: public Application
 {
 private:
  // Objects
  enum objectID
   {
   cOB_RootObject = 0,
   //PushButtons
   cOB_Default,    //1
   cOB_History,    //2
   cOB_DeepHistory, //3
   cOB_BToA,       //4
   cOB_Next,       //5
   //Lamps
```

```
        cOB_LampA,      //6
        cOB_LampE,      //7
        cOB_LampF,      //8
        cOB_LampG,      //9
        cOB_LampH,      //10
        //Modes
        cOB_RootMode,   //11
        cOB_ModeA,      //12
        cOB_ModeB,      //13
        cOB_ModeC,      //14
        cOB_ModeD,      //15
        cOB_ModeE,      //16
        cOB_ModeF,      //17
        cOB_ModeG,      //18
        cOB_ModeH       //19
        };

    enum modeID
        {
        cMO_RootMode = 0,
        cMO_ModeA,    //1
        cMO_ModeB,    //2
        cMO_ModeC,    //3
        cMO_ModeD,    //4
        cMO_ModeE,    //5
        cMO_ModeF,    //6
        cMO_ModeG,    //7
        cMO_ModeH,    //8
        };

    enum ActivityIndices
```

```
        {
        cAC_ModeA_entry,
        cAC_ModeA_exit,
        cAC_ModeE_entry,
        cAC_ModeE_exit,
        cAC_ModeF_entry,
        cAC_ModeF_exit,
        cAC_ModeG_entry,
        cAC_ModeG_exit,
        cAC_ModeH_entry,
        cAC_ModeH_exit,
        };

    enum ConcreteIndices
        {
        cConcrete1 = 0,
        cConcrete2,
        cConcrete3,
        cConcrete4,
        cConcrete5,
        cConcrete6,
        cConcrete7,
        cConcrete8,
        cConcrete9,
        cConcrete10,
        };

    public:

    void init (RapidTask *t);
```

```
// Objects:
RootObject      rootObject;
PushButton      pbDefault;
PushButton      pbHistory;
PushButton      pbDeepHistory;
PushButton      pbNext;
PushButton      pbBToA;
RapidLamp       LampA;
RapidLamp       LampE;
RapidLamp       LampF;
RapidLamp       LampG;
RapidLamp       LampH;

// Modes:
RootMode        rootMode;
XMode           ModeA;
XMode           ModeB;
XMode           ModeC;
XMode           ModeD;
XMode           ModeE;
XMode           ModeF;
XMode           ModeG;
XMode           ModeH;

// Logic Methods:
bool            ModeA_entry();
bool            ModeA_exit();
bool            ModeE_entry();
bool            ModeE_exit();
bool            ModeF_entry();
bool            ModeF_exit();
```

```
bool        ModeG_entry();
bool        ModeG_exit();
bool        ModeH_entry();
bool        ModeH_exit();

// Dynamic State Machine Data:
RapidObject     *FrameTable[cNumberObjects];
ModeP           buffer1[cNumberModes];
ModeP           buffer2[cNumberModes];
ModeP           buffer3[cNumberModes];

unsigned char _ModeFlags[3*cNumberModes][1];

// Static State Machine Data:

// Each object has a list of dependency records, sorted
// by property and event, and referencing pairs of
// (modes, activity/condition, index), or a transition index
static DependencyRecord dependencies[cNumberDependencies];

// For each object/property pair, there is a dependency index record
// that includes the first & last dependency record pertaining to the
// object/property pair.
static DependencyIndexRecord dependencyIndicies[cNumberDependencyIndices];

// Each transition has an index and an associated record that includes
// the indices of the source and destination, type, condition, and action
static TransitionRecord transitionRecords[cNumberTransitions];

// Each transition has an index and an associated record that includes
// the indices of the source and destination, type, condition, and action
```

```
    static TriggerRecord triggerRecords[cNumberTriggers];

    // Activity methods are indexed through the following array
    // subroutine methods are NOT indexed here.
    static MyApplicationMethod activityRecords[cNumberActivities];

    // Each mode record includes  entry and exit activities, if any,
    // and the start index and length of the mode activities. Can the mode activities
    // be merged?  Also, the number of transitions, to facilitate flag allocation.
    static ModeRecord modeRecords[cNumberModes];

    // Each object record includes
    static ObjectRecord objectRecords[cNumberObjects];

    // Array to hold all of the children for the objects
    static int childrenArray[cNumberChildren];

    // General Data on this APplication
    static ApplicationRecord applicationRecord;
};

class myTask : public RapidTask
{
public:
  myTask();
  ~myTask()
    {
    removeContext(&_mainApp);
    };

private:
```

```
    myApplication _mainApp;
  };
#endif
```

Body File

```
/*
```

Rapid Application:

Objects:

```
    Transitn
    |-->BToA (Square Pushbutton)
    |-->Default (Square Pushbutton)
    |-->History (Square Pushbutton)
    |-->DeepHistory (Square Pushbutton)
    |-->Next (Square Pushbutton)
    |-->LampA (Round Lamp)
    |-->LampE (Round Lamp)
    |-->LampF (Round Lamp)
    |-->LampG (Round Lamp)
    |-->LampH (Round Lamp)
```

Modes:

```
    Transitn
    |-->A (D)
    |
    |---B
        |-->C(D)
        |       |-->E(D)
        | |
        |       |---F
        |
```

|---D

|-->G(D)

|

|---H

Logic:

1. MODE: Transitn

Parent: none Type: root

Note:

2. MODE: A

Parent: Transitn Type: exclusive

Entry Activities:

a. LampA on

Exit Activities:

a. LampA off

Transitions:

a. Destination: B Entry type: Default

a1. Trigger: Default in &

b. Destination: B Entry type: History

b1. Trigger: History in &

c. Destination: B Entry type: Deep History

c1. Trigger: DeepHistory in &

3. MODE: B

Parent: Transitn Type: exclusive

Transitions:

a. Destination: A Entry type: Default

a1. Trigger: BToA in &

4. MODE: C

Parent: B Type: exclusive

5. MODE: D

Parent: B Type: exclusive

6. MODE: E

Parent: C Type: exclusive

Entry Activities:

a. LampE on

Exit Activities:

a. LampE off

Transitions:

a. Destination: F Entry type: Default

a1. Trigger: Next in &

6. MODE: F

Parent: C Type: exclusive

Entry Activities:

a. LampF on

Exit Activities:

a. LampF off

Transitions:

a. Destination: G Entry type: Default

a1. Trigger: Next in &

6. MODE: G

Parent: D Type: exclusive

Entry Activities:

a. LampG on

Exit Activities:

a. LampG off

Transitions:

a. Destination: H Entry type: Default

a1. Trigger: Next in &

6. MODE: H

Parent: D Type: exclusive

Entry Activities:

a. LampH on

Exit Activities:

a. LampH off

Transitions:

a. Destination: E Entry type: Default

a1. Trigger: Next in &

Generated Code:

```
*/

#include "transitn.h"

#include "emtrace.h"

#include "ekernel.h"

/*
struct DependencyRecord
 {
 DependencyClass dependencyClass;  // event, condition, mode activity
 int mode;                  // Frame Table index
 int eventID;                // eventID, if any
 int index;                 // Index of transition or mode Activity
 };
*/

DependencyRecord myApplication::dependencies[cNumberDependencies] =
 {
 {cEvent,        cOB_ModeA, cPushButtonIn, 0}, // 0 A->B on Default in
 {cEvent,        cOB_ModeA, cPushButtonIn, 1}, // 1 A->B on History in
 {cEvent,        cOB_ModeA, cPushButtonIn, 2}, // 2 A->B on DeepHistory in
 {cEvent,        cOB_ModeB, cPushButtonIn, 0}, // 3 B->A on BToA in
 {cEvent,        cOB_ModeE, cPushButtonIn, 0}, // 4 E->F on Next in
 {cEvent,        cOB_ModeF, cPushButtonIn, 0}, // 5 F->G on Next in
 {cEvent,        cOB_ModeG, cPushButtonIn, 0}, // 6 G->H on Next in
 {cEvent,        cOB_ModeH, cPushButtonIn, 0}, // 7 H->E on Next in
 };

/*
```

```
struct DependencyIndexRecord{
    PropertyEntry _propertyId;
    int flags = 0;
    unsigned int _firstElement;
}
*/

DependencyIndexRecord myApplication::dependencyIndicies[cNumberDependencyIndices] =
{
  {cSelf, 0, 0} ,          // pbDefault self
  {cSelf, 0, 1} ,          // pbHistory self
  {cSelf, 0, 2} ,          // pbDeepHistor self
  {cSelf, 0, 3} ,          // pbBToA self
  {cSelf, 0, 4} ,          // pbNext self
  {cNone, 0, cNumberDependencies}    // Sentinel for last index record
};

// children listed grouped by common parent
int myApplication::childrenArray[cNumberChildren] =
 {
    cOB_Default,    // 0
    cOB_History,    // 1
    cOB_DeepHistory, // 2
    cOB_BToA,           // 3
    cOB_Next,       // 4
    //Lamps
    cOB_LampA,      // 5
    cOB_LampE,      // 6
    cOB_LampF,      // 7
    cOB_LampG,      // 8
    cOB_LampH,      // 9
```

```
    //Modes
    cOB_ModeA,      // 10
    cOB_ModeB,      // 11
    cOB_ModeC,      // 12
    cOB_ModeD,      // 13
    cOB_ModeE,      // 14
    cOB_ModeF,      // 15
    cOB_ModeG,      // 16
    cOB_ModeH       // 17
  };

/*
struct ObjectRecord
  {
  ObjectIdentifier name;
  int parent;
  int id;  // Id within type...e,g,  modes are numbered 1-n, concretes are numbered 1-m
  int numChildren;
  int firstChild;
  int firstDependentIndex;
  int numDependentIndex;
  };
*/

ObjectRecord myApplication::objectRecords[cNumberObjects]=
  {
                                    cNone,     10,        0,              0,
    0},
                cOB_RootObject,        cConcrete1,     0,        cNone,
    0,          1},
```

```
                cOB_RootObject,     cConcrete2,     0,      cNone,
    1,      1},
                                    cConcrete3,     0,      cNone,
    2,      1},
                cOB_RootObject,     cConcrete5,     0,      cNone,
    3,      1},
            cOB_RootObject,     cConcrete4,     0,      cNone,      4,
    4},
                                    cConcrete6,     0,      cNone,
    0,      0},
                                    cConcrete7,     0,      cNone,
    0,      0},
                                    cConcrete8,     0,      cNone,
    0,      0},
                                    cConcrete9,     0,      cNone,
    0,      0},
                                    cConcrete10,    0,      cNone,
    0,      0},
                                cMO_RootMode,   7,      11,     0,
    0},
                                    cMO_ModeA,      0,      cNone,
    0,      0},
                                    cMO_ModeB,      6,      10,     0,
    0},
                                cMO_ModeC,  2,      14,     0,
    0},
                                cMO_ModeD,  2,      16,     0,
    0},
                                cMO_ModeE,  0,      cNone,      0,
    0},
```

```
                                    cMO_ModeF,    0,        cNone,        0,
        0},
                                    cMO_ModeG,    0,        cNone,        0,
        0},
                                    cMO_ModeH,    0,        cNone,        0,
        0}
    };

    ApplicationRecord myApplication::applicationRecord=
    {
    cNumberModes,
    cNumberActivities
    };

    /*
    struct TriggerRecord
    {
    int objectID;
    int propertyID;
    int eventID;
    };
    */

    // NOTE: Could get sophisticated, and reuse trigger records...
    TriggerRecord myApplication::triggerRecords[cNumberTriggers] =
    {
    {cOB_Default, cSelf, cPushButtonIn},     // 0
    {cOB_History, cSelf, cPushButtonIn},     // 1
    {cOB_DeepHistory, cSelf, cPushButtonIn}, // 2
    {cOB_BToA, cSelf, cPushButtonIn},        // 3
    {cOB_Next, cSelf, cPushButtonIn},        // 4
```

```
  };
/*
struct TransitionRecord
  {
  int triggerStart;    // index of first trigger record
  int triggerLength;   // number of trigger records
  int sourceMode;      // object index of source mode
  int destinationMode; // object index of destination mode
  int condition;       // index of condition method
  int action;          // index of action method
  unsigned int type:3;
  unsigned int IsConditionOnly:1;
  unsigned int toAncestor:1;
  unsigned int toDescendant:1;
  unsigned int sourceEqualsDestination:1;
  };
*/

TransitionRecord myApplication::transitionRecords[cNumberTransitions]=
  {
  {0, 1, cOB_ModeA, cOB_ModeB, cNone, cNone, cDefaultTransition,      false, false, false,
false},// A->B  #1
  {1, 1, cOB_ModeA, cOB_ModeB, cNone, cNone, cHistoryTransition,      false, false, false,
false},// A->B  #2
  {2, 1, cOB_ModeA, cOB_ModeB, cNone, cNone, cDeepHistoryTransition, false, false, false,
false},// A->B  #3
  {3, 1, cOB_ModeB, cOB_ModeA, cNone, cNone, cDefaultTransition,      false, false, false,
false},// B->A  #1
  {4, 1, cOB_ModeE, cOB_ModeF, cNone, cNone, cDefaultTransition,      false, false, false,
false},// E->F  #1
```

```
 {4, 1, cOB_ModeF, cOB_ModeG, cNone, cNone, cDefaultTransition,      false, false, false,
false},// F->G  #1
 {4, 1, cOB_ModeG, cOB_ModeH, cNone, cNone, cDefaultTransition,      false, false, false,
false},// G->H  #1
 {4, 1, cOB_ModeH, cOB_ModeE, cNone, cNone, cDefaultTransition,      false, false, false,
false},// H->E  #1
 };

/*
struct ModeRecord
 {
 ObjectIdentifier name;
 int defaultChildID;       //ObjectID of default child, cNone if none
                        // For parents of AndModes, first child
 int entry;
 int exit;
 int modeActivityStart;
 int numModeActivities;    // N/A, condition index, activity index
 int transitionStart;      // First transition record pertaining to this mode
 int numTransitions;       // Number of transitions
 };
*/

ModeRecord myApplication::modeRecords[cNumberModes]=
          //def child   entry                exit                modeStart numMode    transSart
     numTrans
 {"RootMode", cOB_ModeA,  cNone,        cNone,        cNone,     0,      cNone,    0},
                                                                                0,      0,
     3},
 {"ModeB",   cOB_ModeC,  cNone,        cNone,        cNone,     0,      3,      1},
 {"ModeC",   cOB_ModeE,  cNone,        cNone,        cNone,     0,      cNone,    0},
```

```
{"ModeD", cOB_ModeG, cNone, cNone, cNone, 0, cNone, 0},
                    cAC_ModeE_entry, cAC_ModeE_exit, cNone, 0, 4,
    1},
                    cAC_ModeF_entry, cAC_ModeF_exit, cNone, 0, 5,
    1},
                    cAC_ModeG_entry, cAC_ModeG_exit, cNone, 0, 6,
    1},
                    cAC_ModeH_entry, cAC_ModeH_exit, cNone, 0, 7,
    1}
};

MyApplicationMethod myApplication::activityRecords[cNumberActivities] =
{
&myApplication::ModeA_entry,
&myApplication::ModeA_exit,
&myApplication::ModeE_entry,
&myApplication::ModeE_exit,
&myApplication::ModeF_entry,
&myApplication::ModeF_exit,
&myApplication::ModeG_entry,
&myApplication::ModeG_exit,
&myApplication::ModeH_entry,
&myApplication::ModeH_exit,
};

bool myApplication::ModeA_entry()
{
LampA.turn_on();   // LampA on
return false;
}
```

```
bool myApplication::ModeA_exit()
 {
 LampA.turn_off();   // LampA off
 return false;
 }

bool myApplication::ModeE_entry()
 {
 LampE.turn_on();   // LampA on
 return false;
 }

bool myApplication::ModeE_exit()
 {
 LampE.turn_off();   // LampA off
 return false;
 }

bool myApplication::ModeF_entry()
 {
 LampF.turn_on();   // LampA on
 return false;
 }

bool myApplication::ModeF_exit()
 {
 LampF.turn_off();   // LampA off
 return false;
 }

bool myApplication::ModeG_entry()
```

```
{
LampG.turn_on();   // LampA on
return false;
}

bool myApplication::ModeG_exit()
{
LampG.turn_off();   // LampA off
return false;
}

bool myApplication::ModeH_entry()
{
LampH.turn_on();   // LampA on
return false;
}

bool myApplication::ModeH_exit()
{
LampH.turn_off();   // LampA off
return false;
}

void myApplication::init (RapidTask *t)
{
setTask(t);
Application::dependencies = dependencies;
Application::dependencyIndicies = dependencyIndicies;
Application::transitionRecords = transitionRecords;
Application::triggerRecords = triggerRecords;
Application::activityRecords = (ApplicationMethod *)activityRecords;
```

```
Application::objectRecords = objectRecords;
Application::modeRecords = modeRecords;
Application::objects = FrameTable;
Application::applicationRecord = &applicationRecord;
Application::children = childrenArray;

rootObject.init(cOB_RootObject, this);

pbDefault.init(cOB_Default,this);
pbHistory.init(cOB_History,this);
pbDeepHistory.init(cOB_DeepHistory,this);
pbBToA.init(cOB_BToA,this);
pbNext.init(cOB_Next,this);
LampA.init(cOB_LampA,this);
LampE.init(cOB_LampE,this);
LampF.init(cOB_LampF,this);
LampG.init(cOB_LampG,this);
LampH.init(cOB_LampH,this);

rootMode.init(cOB_RootMode, this,
        _ModeFlags[(3*cMO_RootMode)],
        _ModeFlags[(3*cMO_RootMode)+1],
        _ModeFlags[(3*cMO_RootMode)+2]);
ModeA.init  (cOB_ModeA, this,
        _ModeFlags[(3*cMO_ModeA)],
        _ModeFlags[(3*cMO_ModeA)+1],
        _ModeFlags[(3*cMO_ModeA)+2]);
ModeB.init  (cOB_ModeB, this,
        _ModeFlags[(3*cMO_ModeB)],
        _ModeFlags[(3*cMO_ModeB)+1],
        _ModeFlags[(3*cMO_ModeB)+2]);
```

```
ModeC.init  (cOB_ModeC, this,
        _ModeFlags[(3*cMO_ModeC)],
        _ModeFlags[(3*cMO_ModeC)+1],
        _ModeFlags[(3*cMO_ModeC)+2]);
ModeD.init  (cOB_ModeD, this,
        _ModeFlags[(3*cMO_ModeD)],
        _ModeFlags[(3*cMO_ModeD)+1],
        _ModeFlags[(3*cMO_ModeD)+2]);
ModeE.init  (cOB_ModeE, this,
        _ModeFlags[(3*cMO_ModeE)],
        _ModeFlags[(3*cMO_ModeE)+1],
        _ModeFlags[(3*cMO_ModeE)+2]);
ModeF.init  (cOB_ModeF, this,
        _ModeFlags[(3*cMO_ModeF)],
        _ModeFlags[(3*cMO_ModeF)+1],
        _ModeFlags[(3*cMO_ModeF)+2]);
ModeG.init  (cOB_ModeG, this,
        _ModeFlags[(3*cMO_ModeG)],
        _ModeFlags[(3*cMO_ModeG)+1],
        _ModeFlags[(3*cMO_ModeG)+2]);
ModeH.init  (cOB_ModeH, this,
        _ModeFlags[(3*cMO_ModeH)],
        _ModeFlags[(3*cMO_ModeH)+1],
        _ModeFlags[(3*cMO_ModeH)+2]);

rootMode.initActiveChild();
ModeA.initActiveChild();
ModeB.initActiveChild();
ModeC.initActiveChild();
ModeD.initActiveChild();
ModeE.initActiveChild();
```

```
ModeF.initActiveChild();
ModeG.initActiveChild();
ModeH.initActiveChild();

_currentState.init(buffer1,cNumberModes);
_nextState.init(buffer2,cNumberModes);
_tempState.init(buffer3,cNumberModes);
Application::rootMode = &rootMode;
Application::rootObject = &rootObject;
}

myTask::myTask()
{
_mainApp.init(this);
setMainApplication(&_mainApp);
setupKernelTask (this);
};

// declare one task
myTask GlobalTask;
```

It is appreciated that the software components of the present invention may, if desired, be implemented in ROM (read-only memory) form. The software components may, generally, be implemented in hardware, if desired, using conventional techniques.

It is appreciated that various features of the invention which are, for clarity, described in the contexts of separate embodiments may also be provided in combination in a single embodiment. Conversely, various features of the invention which are, for brevity, described in the context of a single embodiment may also be provided separately or in any suitable subcombination.

It will be appreciated by persons skilled in the art that the present invention is not limited to what has been particularly shown and described hereinabove. Rather, the scope of the present invention is defined only by the claims that follow:

CLAIMS

We claim:

1. Apparatus implementing hierarchical state charts and comprising:

a state machine engine (SME) operative to carry out repertoires of behaviors of a system;

a state chart encoder operative to replicate, in computer code form, a user-defined hierarchical state chart describing a repertoire of behaviors of said system, said encoder being operative to preserve said state chart's hierarchical structure intact in a first data structure;

an event list generator operative to generate, for each individual event from among a plurality of events within the repertoire, a list of at least one transition between states of said state chart which are associated with said individual event, said event list generator being operative to preserve said list in a second data structure; and

a behavior implementer operative to activate said system to carry out each of the behaviors in its repertoire.

2. Apparatus according to claim 1 wherein said system comprises at least one of hardware and software.

3. Apparatus according to claim 1 wherein said first data structure's size increases linearly in direct proportion to the number of branches in said hierarchical structure.

4. Apparatus according to claim 1 wherein said second data structure's size increases linearly in direct proportion to the number of transitions in said list of at least one transition.

5. Apparatus according to claim 1 wherein the order of said at least one transition of each of said lists corresponds to the order of evaluation of said transitions as defined by said hierarchical state chart.

6. Apparatus according to claim 1 wherein said plurality of events for which said event list generator generates lists comprises at least one external input event.

7. Apparatus according to claim 1 wherein said plurality of events for which said event list generator generates lists comprises at least one internal data value change event, the apparatus also including a data value change event notifier operative to generate a notification to the state machine engine of each data value change event.

8. A method implementing hierarchical state charts, the method comprising:

providing a state machine engine (SME) operative to carry out repertoires of behaviors of a system;

replicating, in computer code form, a user-defined hierarchical state chart describing a repertoire of behaviors of said system, said replicating step being operative to preserve said state chart's hierarchical structure intact;

generating, for each individual event from among a plurality of events within the repertoire, a list of at least one transition between states of said state chart which are associated with said individual event; and

activating said system to carry out each of the behaviors in its repertoire.

9. An automatic code generation method comprising:

automatically generating code which forms, in memory, a replica of a hierarchical state chart which preserves said chart's hierarchical structure;

automatically generating code which forms, for each of a first plurality of input events, a list of transitions affected by that input event;

automatically generating code which forms, for each of a second plurality of internal data variables, a list of transitions affected by changes in that internal data variable;

automatically generating code which notifies the state machine engine of changes in internal data variables; and

automatically generating code which evaluates conditions and implements actions and activities.

10. A method according to claim 9 and comprising the step of providing a state machine engine operative, upon receipt of notification regarding occurrence of an event, to

evaluate a list of transitions affected by said event so as to select transitions to be performed, and to perform said transitions.

11. A method for computing a state for a hierarchical state machine, the method comprising:

providing a state machine engine operative to receive notification regarding occurrence of at least one event;

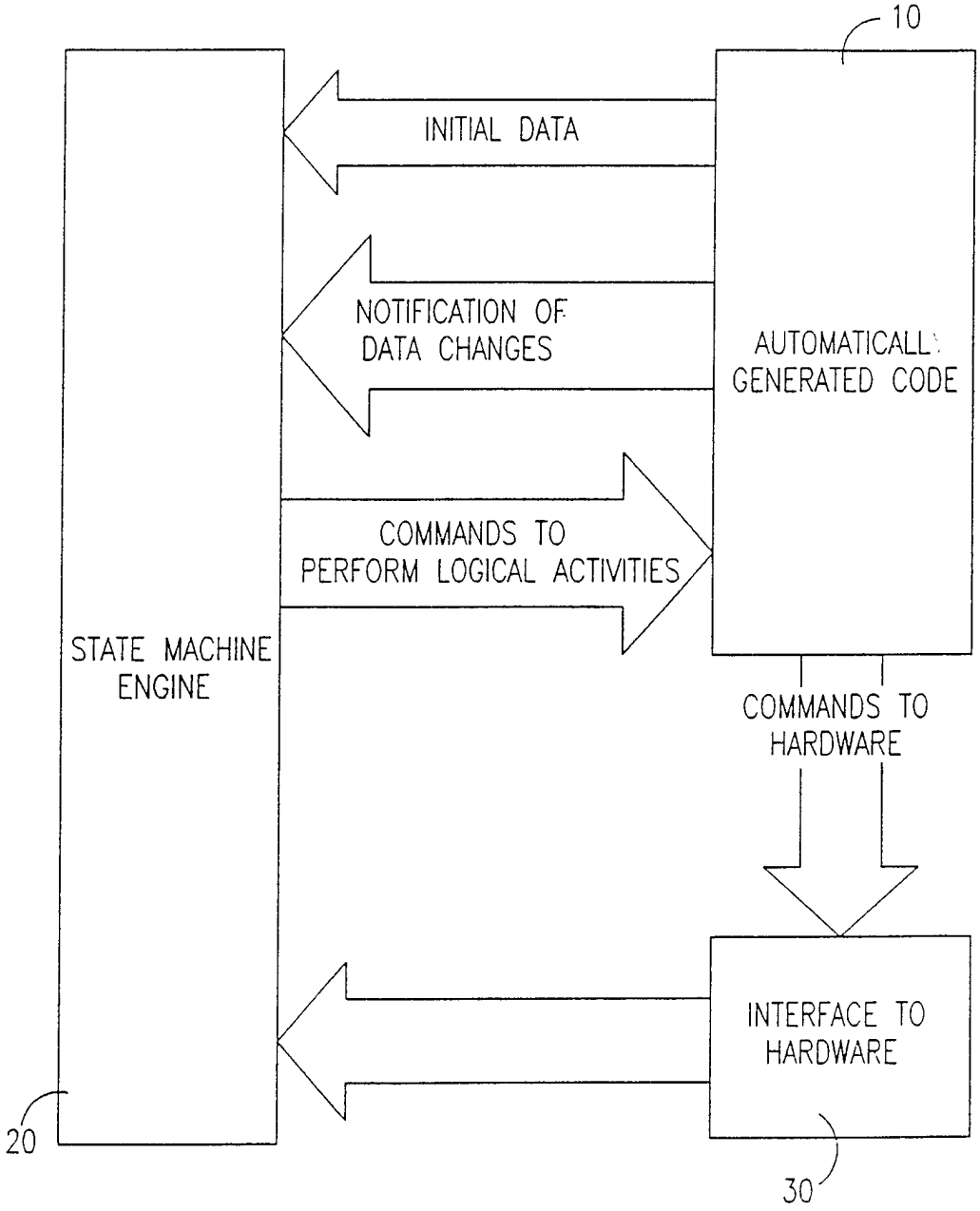
providing a hierarchical state chart describing a repertoire of behaviors of a system;

providing a list of at least one transition between states of said state chart for each of said events with which said at least one transition is associated;

evaluating said list of at least one transition so as to select a transition to be performed in response to said at least one event; and

evaluating said hierarchical state chart to compute the state that corresponds to said selected transition.

FIG. 1


10
INITIAL DATA
NOTIFICATION OF DATA CHANGES
AUTOMATICALLY GENERATED CODE
COMMANDS TO PERFORM LOGICAL ACTIVITIES
STATE MACHINE ENGINE
COMMANDS TO HARDWARE
INTERFACE TO HARDWARE
20
30

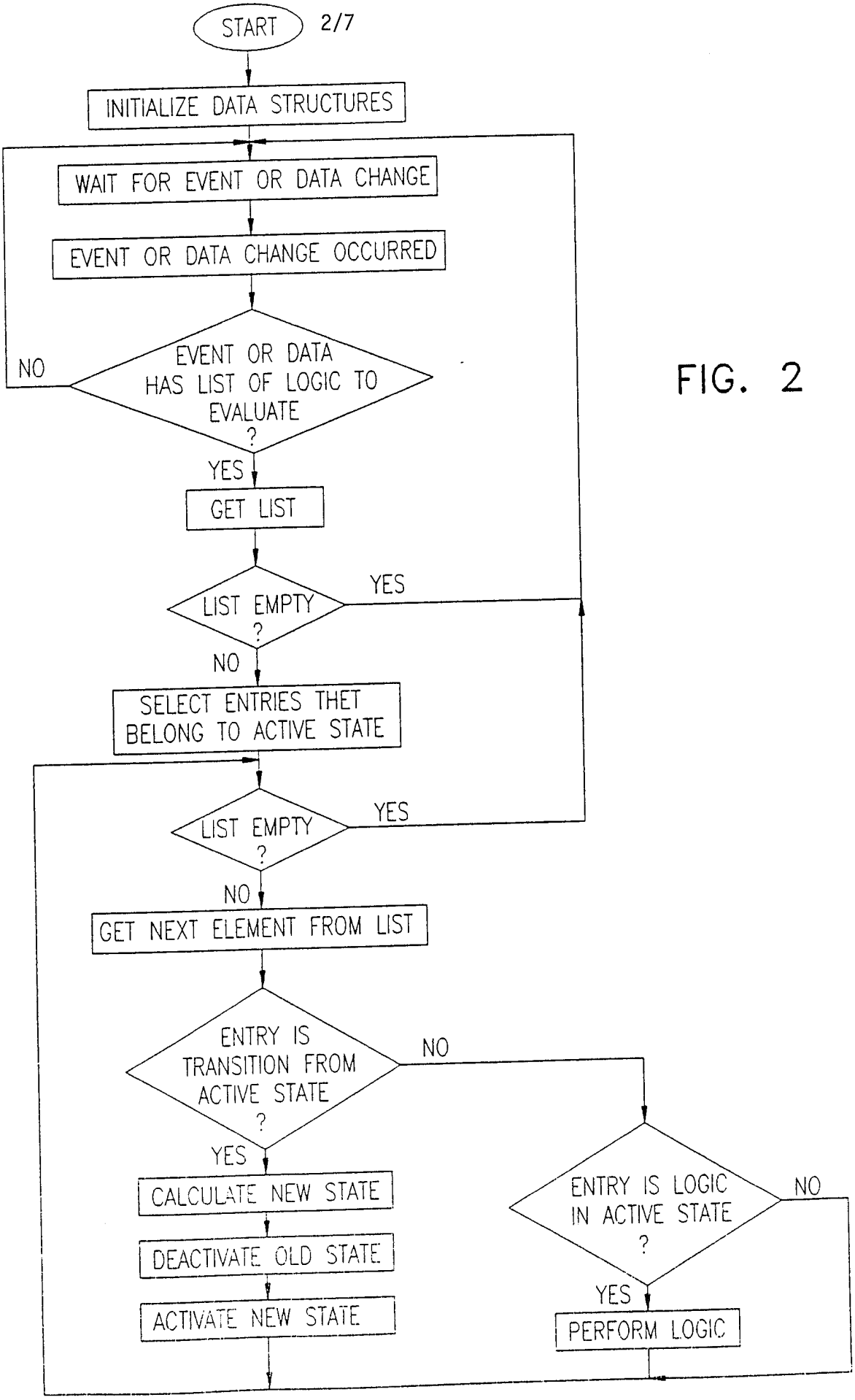

START
INITIALIZE DATA STRUCTURES
WAIT FOR EVENT OR DATA CHANGE
EVENT OR DATA CHANGE OCCURRED
EVENT OR DATA HAS LIST OF LOGIC TO EVALUATE ?
NO
YES
GET LIST
LIST EMPTY ?
YES
NO
SELECT ENTRIES THET BELONG TO ACTIVE STATE
LIST EMPTY ?
YES
NO
GET NEXT ELEMENT FROM LIST
ENTRY IS TRANSITION FROM ACTIVE STATE ?
NO
YES
CALCULATE NEW STATE
DEACTIVATE OLD STATE
ACTIVATE NEW STATE
ENTRY IS LOGIC IN ACTIVE STATE ?
NO
YES
PERFORM LOGIC

FIG. 2

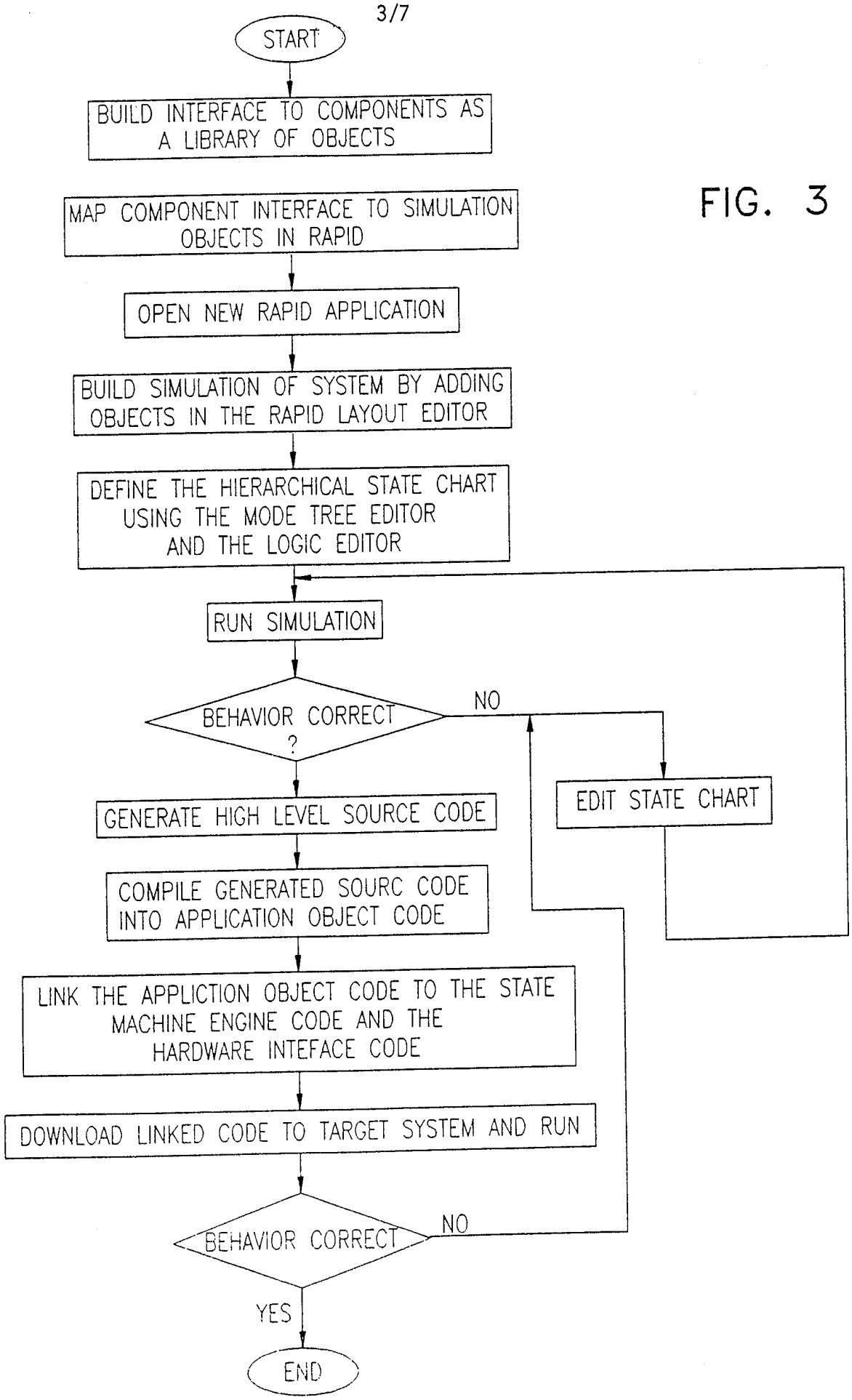

START
BUILD INTERFACE TO COMPONENTS AS A LIBRARY OF OBJECTS
MAP COMPONENT INTERFACE TO SIMULATION OBJECTS IN RAPID
OPEN NEW RAPID APPLICATION
BUILD SIMULATION OF SYSTEM BY ADDING OBJECTS IN THE RAPID LAYOUT EDITOR
DEFINE THE HIERARCHICAL STATE CHART USING THE MODE TREE EDITOR AND THE LOGIC EDITOR
RUN SIMULATION
BEHAVIOR CORRECT ?
NO
EDIT STATE CHART
GENERATE HIGH LEVEL SOURCE CODE
COMPILE GENERATED SOURC CODE INTO APPLICATION OBJECT CODE
LINK THE APPLICTION OBJECT CODE TO THE STATE MACHINE ENGINE CODE AND THE HARDWARE INTEFACE CODE
DOWNLOAD LINKED CODE TO TARGET SYSTEM AND RUN
BEHAVIOR CORRECT
NO
YES
END

FIG. 3

FIG. 4

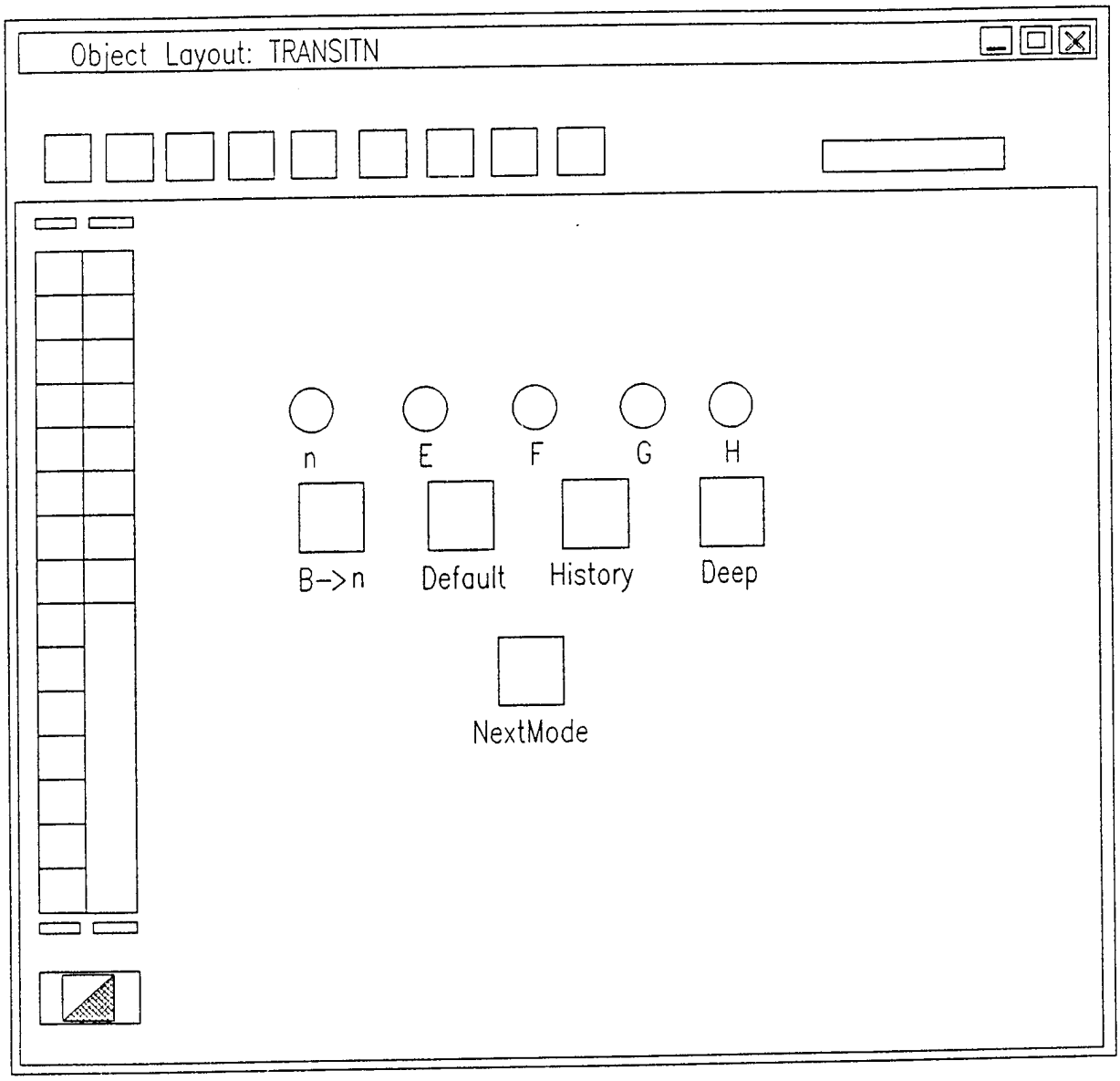

Object Layout: TRANSITN
n
E
F
G
H
B->n
Default
History
Deep
NextMode

FIG. 5

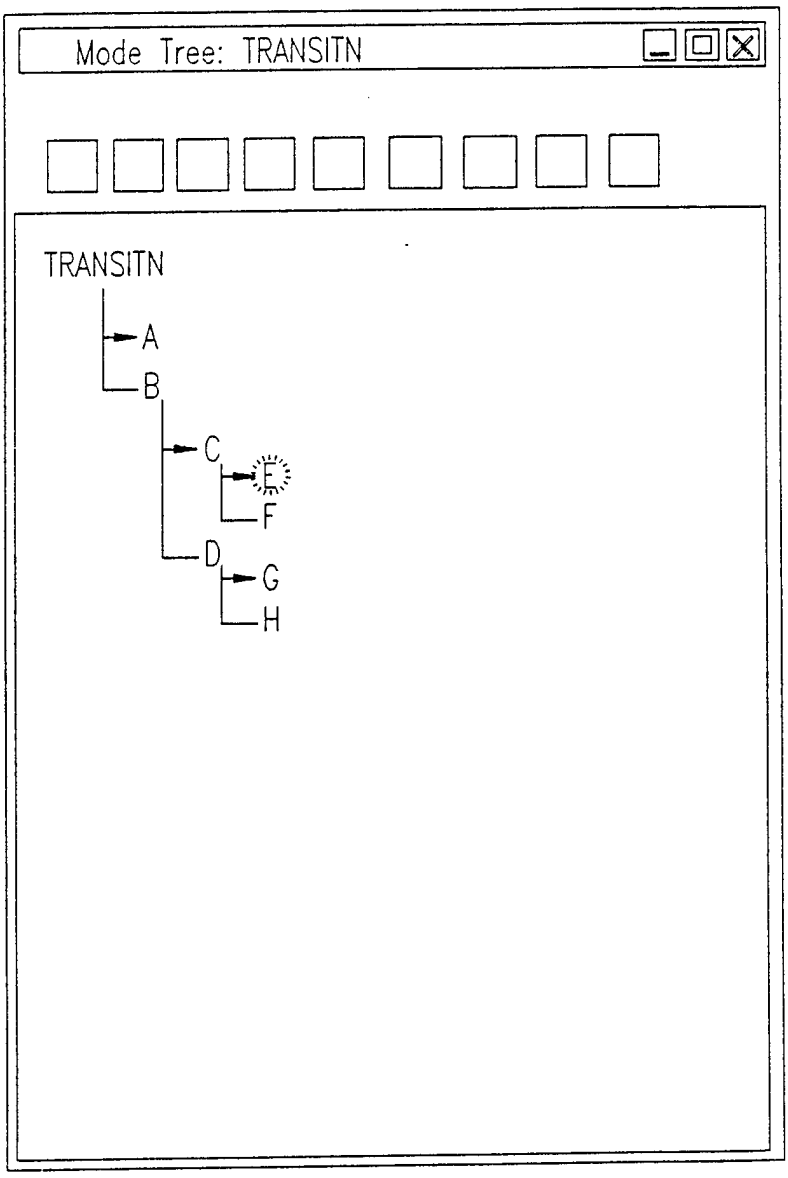


FIG. 6

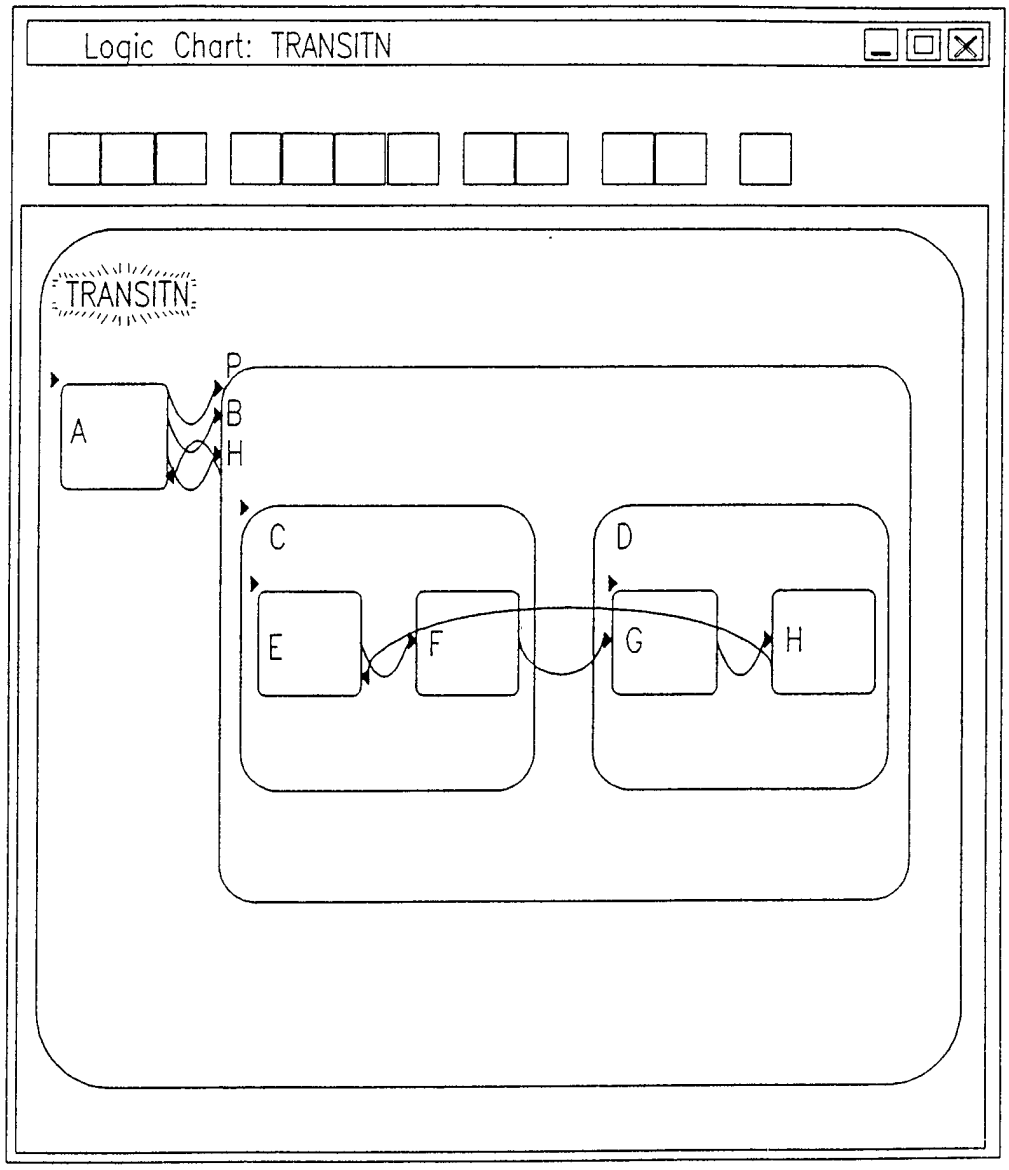
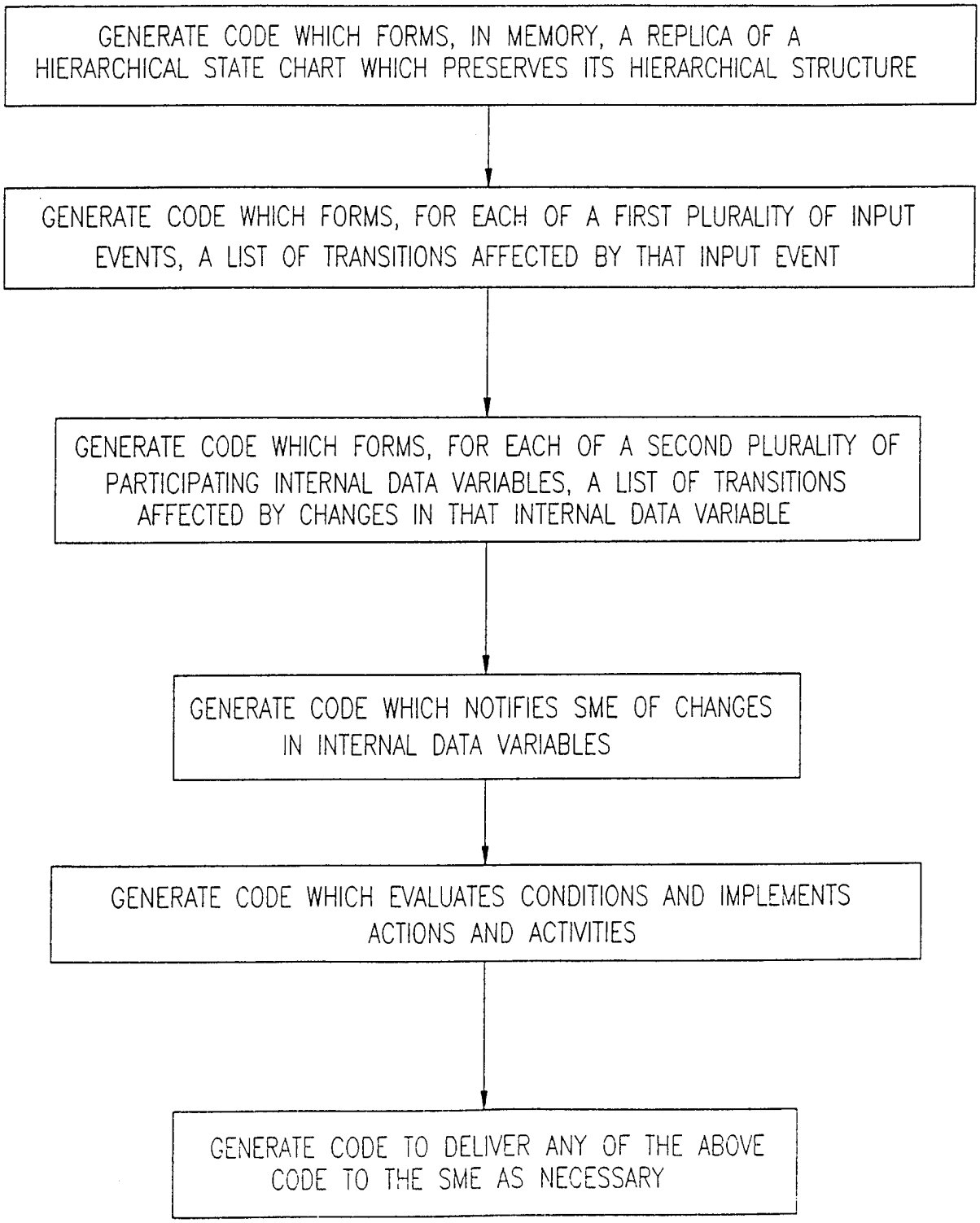


FIG. 7


GENERATE CODE WHICH FORMS, IN MEMORY, A REPLICA OF A HIERARCHICAL STATE CHART WHICH PRESERVES ITS HIERARCHICAL STRUCTURE
GENERATE CODE WHICH FORMS, FOR EACH OF A FIRST PLURALITY OF INPUT EVENTS, A LIST OF TRANSITIONS AFFECTED BY THAT INPUT EVENT
GENERATE CODE WHICH FORMS, FOR EACH OF A SECOND PLURALITY OF PARTICIPATING INTERNAL DATA VARIABLES, A LIST OF TRANSITIONS AFFECTED BY CHANGES IN THAT INTERNAL DATA VARIABLE
GENERATE CODE WHICH NOTIFIES SME OF CHANGES IN INTERNAL DATA VARIABLES
GENERATE CODE WHICH EVALUATES CONDITIONS AND IMPLEMENTS ACTIONS AND ACTIVITIES
GENERATE CODE TO DELIVER ANY OF THE ABOVE CODE TO THE SME AS NECESSARY

INTERNATIONAL SEARCH REPORT

International application No.
PCT/IL97/00418

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 17/27, 17/30
US CL :Please See Extra Sheet.
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/500, 676, 680, 682, 685, 701; 364/468.24, 474.24; 707/100, 101, 104

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS, IEEE TRANSACTIONS/JOURNALS, COMPUTERS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,485,600 A (JOSEPH ET AL.) 16 JANUARY 1996, col. 11, col. 12, col. 13, line 26 to col. 14, line 10, col. 18.	1-11
Y	US 5,257,363 A (SHAPIRO ET AL.) 26 October 1993, cols. 9-19.	1-11
Y	US 5,163,016 A (HAR'EL ET AL.) 10 November 1992, entire document.	1-11
Y	US 5,493,680 A (DANFORTH) 20 February 1996, cols. 8-27.	1-11

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search
03 APRIL 1998

Date of mailing of the international search report
01 JUN 1998

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer
THAI PHAN
Telephone No. (703) 305-3900

Form PCT/ISA/210 (second sheet)(July 1992)★

INTERNATIONAL SEARCH REPORT

International application No.
PCT/IL97/00418

A. CLASSIFICATION OF SUBJECT MATTER:
US CL :

395/500, 676, 680, 682, 685, 701; 364/468.24, 474.24; 707/100, 101, 104