Transforming Statecharts into Reactive Systems[1]

Antonio Gonçalves Figueiredo Filho
Hans Liesenberg

Relatório Técnico DCC–01/93

Fevereiro de 1993

Transforming Statecharts into Reactive Systems[†]

Antonio Gonçalves Figueiredo Filho[‡]
Hans Liesenberg[§]

Abstract

This paper describes an environment and associated techniques which support the modelling and generation of reactive systems. The modelling is based on statecharts, an extension of conventional state transition diagrams, which favours event-driven control aspects related to context swappings which can become very complex in non-trivial reactive systems. Contexts are defined incrementally and thus a more structured specification approach is encouraged. Semantics is aggregated in terms of attributes of statechart elements expressed as functions in C code. At a second stage, a translation of a model into a functionally equivalent program in C takes place. Special emphasis is given to the invariant part of the generated code, referred to as the statechart engine, and to the binding of logical events to events recognized by the underlying kernel driving input/output devices. The major advantages brought by the environment are precise description facilities of subtle behavioural aspects, punctual action definitions to be carried out in specific contexts and the capability of automatically transforming specifications into programs.

[†]Supported by CNPq and FAPESP.
[‡]antonio@dcc.unicamp.br
[§]hans@dcc.unicamp.br

1. Introduction

Lehman's PW model [Leh84], which describes processes related to the development and the evolution of software, divides the development process into two stages. At first a formal specification is derived from a concept of an application and, at a second stage, this specification is transformed into an executable program. This paper is concerned with this second stage. The adopted instantiation of the second part of Lehman's model starts from a partial specification which describes event-driven control aspects complemented with lower abstraction level descriptions of actions to be executed in specific contexts. The environment supports the task of describing this kind of hybrid specifications and transforms them automatically into functionally equivalent programs.

Different abstraction levels are generally used in system design in order to enable a better understanding of the artifact under construction as well as to keep its complexity under control. Each abstraction layer generally favours certain characteristics of a system to the detriment of others. Different notations are generally used at distinct levels as a vehicle to convey the reasoning at those levels.

One of the major problems in handling a design at different abstraction levels is maintaining the consistency of different representations of the same object whenever a manual intervention is required to convert one representation at a given abstraction level into another at the immediately lower level. Automatic transformations are in general feasible at the lower end of an abstraction hierarchy. Usually, at higher levels little computational support is provided: either transformations are carried out manually or require manual interventions.

This paper describes a set of tools for the support of modelling and developent of reactive systems. These tools are based on the use of the statechart notation [Har87] for system specification. The supported system development process consists of two stages. In the first stage, the designer specifies the target system using the statechart paradigm, which is backed up by an interactive statechart design editor. In the second stage, the statechart is processed by a transformation tool, which

generates the resultant reactive software.

The statechart notation is very useful in order to incrementally define active contexts of a system as well as conditions which enable controlled context swappings. A statechart specification is complemented with functions in C code which define localized actions to be carried out as contexts are being activated or deactivated. The task of the system designer is eased since now his major concern is to take into account what to do in a specific context without worrying about how the system got into this context, or monitoring the satisfaction of context swapping conditions.

The hybrid specifications (statecharts plus complementary functions in C code) are capable of being transformed automatically into functionally equivalent programs. No manual intervention is required once the specification has been concluded. Thus, the transformation process does not represent a new source of errors and it is carried out rapidly. These are important factors to be considered if the maintenance of a system is expected to occur at the highest abstraction level supported by the environment. The environment provides additional debugging aids based on statechart animations [GSE92], which too intend to encourage corrective actions at the specification level.

This paper is organized as follows. Section 2 gives a brief introduction to statecharts and reactive systems. Section 3 describes the architecture of the transformation tool. Section 4 describes the invariant code resultant of a transformation referred to as the statechart engine. Section 5 describes the binding of events, used in a statechart specification, to events generated by the underlying platform, which the code has been generated for; and section 6 has the conclusions.

2. Statecharts

The statechart notation proposed by D. Harel [Dru89, Har85, Har87] favours control aspects and consists of an extension of conventional state diagrams. Statecharts support the description of hierarchical control specification of event-driven systems composed possibly by concurrent

subsystems whose behaviour might present a certain degree of interdependence. The following description does not adhere in all aspects to the statechart notation and its formal definition as proposed by Harel. A particular dialect has been adopted which is better suited to the domain where statecharts are now being practised.

The notation proposed by Harel is bi-dimensional. A system is represented by a rectangle with rounded corners called blob. Its subsystems are represented as blobs contained within its boundaries. A system can be decomposed in two ways: into mutually excluding subsystems from the behavioural point of view, i.e., at most one of the siblings is active at a given instant; or into concurrent subsystems, i.e., if one of them is active, all of its siblings are active too. Subsystems might be decomposed into even smaller subsystems in the same manner thus enabling hierarchical descriptions.

Dynamic aspects of a statechart are captured in terms of transitions fired by events under possibly enforced restrictions referred to as conditions. A transition can be established between two blobs (the origin and the destination blob) at any level of the hierarchy. A transition is represented by an arrow labelled by the firing event(s) and possibly associated conditions. A transition is fired whenever a) the origin blob is active, b) one of the associated events occurs, and c) the related restricting condition (if such a condition has been declared) is satisfied. Under such circumstances all blobs from an active atomic descendant of the origin blob up to the nearest common hierarchical ancestor of this blob and the destination blob are deactivated (with exception of the nearest common ancestor) as well as all concurrent components and their active descendants along this path.

At a second stage, the blobs on the path from the nearest common ancestor down to the destination blob are successively activated, with exception of the former. Once the destination blob has been reached, the activation process is sustained until an atomic blob is eventually activated. If during an activation process a blob with concurrent components is reached, all of its siblings have to become active. The activation process below the destination blob or of concurrent components

which do not contain the destination blob may be performed basically in two ways: if a history is being enforced, the most recently visited descendant is reactivated; on the contrary or if none of the descendants have yet been activated, then the default descendant is submitted to the activation process. History attributes may be associated either to transitions or to blobs. The history attributes of the former class have precedence over those of the latter. History attributes are of two kinds: flat or in-depth. A flat history applies only to the immediately lower level. In-depth histories apply to all lower levels unless overridden by history attributes of blobs or cancelled at lower levels.

Semantics may be added to a statechart by means of scripts, referred to as actions, to be executed under certain circumstances. A script might be associated to a transition or to a sequential blob. In the first case, the script is executed whenever the corresponding transition is fired. In the second case, up to three scripts may be specified: one to be executed on entry of the corresponding blob (i.e., at the moment of activation), one to be executed on exit (i.e., at the moment of deactivation) and one throughout the interval the blob is active. All actions, except the last one, are supposed to be executed in a very short period of time. The last one may take a longer period of time and is thus referred to as an activity. Harel originally restricted the use of these scripts basically to generate internal events and to manipulate global variables used mainly to verify the satisfaction of conditions related to those variables. Other statechart features are less relevant to the understanding of this text and thus they will not be discussed further.

Reactive systems are characterized as systems which are driven by external and internal events producing results in accordance to these stimuli. Their execution is conceptually infinite. The output of these systems cannot in general be described by a mere function applied to the input of the system. Statecharts are a very appropriate mechanism to model reactive systems because of their essentially event-driven nature.

The statechart notation is used in the present work to specify reactive software systems. Scripts, however, are supplied in the form of functions in C code. In the following sections a transformational tool is described

which converts statechart specifications into functionally equivalent C programs. The code produced consists of two parts: one part is specific and closely related to the structure of the statechart specification to be converted into a C program; the other is invariant and is independent of a particular statechart structure. The second part is called the statechart engine and will be described in greater detail. In essence, it implements the control actions related with the firing of transitions.

Since different contexts are created by the hierarchical structuring of a statechart, the designer of a reactive software can address his attention in a more locally confined way to define scripts associated to blobs. The controlled deactivation and activation process performed at each firing of a transition takes care of more global contexts. Thus, the concern of the designer can be centered on setting up the local context on entering a given blob and on defining the process to be run while this blob is active as well as on the housekeeping tasks to be performed when the blob is left.

3. The Architecture of the Transformational Tool

The front end of the transformational tool consists of a bi-dimensional statechart editor with simulation facilities [Bat91, Can92, Mas91] developed at USP/São Carlos. The back end, i.e., the application generator, performs the actual transformation [Fig91]. The communication between these two ends is performed via a file containing a human-readable textual description of the statechart to be converted into a C program.

This loosely coupled architecture resulted from a conservative design strategy which intended to guarantee a certain degree of independence between the two ends of the system. The back end was developed from scratch and the front end had to be adapted to support new features. Among these features we can cite: the facility to specify scripts as C functions; the animation of a specification fed by events occurring during the execution, in debugging mode, of the corresponding program generated from that specification; and the generation of textual descriptions of blobs for the application generator.

The application generator receives a textual description of a statechart which reflects its nested structure. The transformational process is based on conventional compiling techniques which convert a textual specification into an executable code. The specification dependent code generated by the tool consists of a program template customized in relation to the underlying kernel driving the event generating devices as well as to specific operations dependant on the topology and on the attributes of elements of the statechart submitted to the application generator. These operations select transition firings and perform the controlled deactivation and activation of blobs (the latter process is referred to as the statechart engine) due to a transaction firing. Furthermore, functions representing scripts are incorporated literally into the generated code as defined by the system designer. A script associated to a fired transition is executed before the statechart engine is set in motion to handle the exit and the entrance of blobs affected by the fired transition.

4. The Statechart Engine

The blob deactivation and activation process is referred to as the statechart engine. Its operation is based upon a tree structure which reflects the topology of the statechart submitted to the application generator. This tree structure is organized and threaded in such a way that the major operations on this structure are performed efficiently. These operations include the search for the nearest common ancestor, traversing up and down the branches of the tree between two given nodes, and finding out all nodes that descend from a common direct ancestor.

The linear description of a statechart produced by the front end establishes an ordering relation of the blobs and the concurrent components. Textual descriptions of direct descendants are embedded in the description of their ancestor. A default descendant is always the first blob in a sequence of siblings. Thus, a statechart is represented as a nested textual definition. The order of interest for the algorithms presented below is the ordinal number related to the occurrence of the blob and the concurrent component identifiers in this linear description from

left to right. Blob identifiers are associated to integer values corresponding to their ordinal numbers (i.e., they are defined to be equivalent by declaring them as constants of a given value in the generated code) and are referred to as their numerical identifiers. A numerical identifier is used as an index in order to refer an element of the array. This array element contains the information relevant to the corresponding blob and represents the related node of the tree which reflects a statechart topology.

The information kept at each node consists of links used to thread the tree structure, flags indicating history attributes as well as information required to enable the activation of the most recently visited descendant in case of a history enforcement. The size of the tree structure is closely related to the statechart topology submitted to the transformational tool (essentially one entry for each blob and each component) and does not change during the execution of the generated program. The array, where the information about the nodes of the tree are kept, is properly initialized by the application generator and allocated statically.

The tree structure is threaded in the following way: a) an ancestor points to its first direct descendant with the lowest numerical identifier value; b) each node points to its ancestor; c) direct descendants of a given node are inserted in a circular list in increasing order of the values of their numerical identifiers, with exception of the node representing the blob with the highest numerical identifier value, which points to the node of the lowest one. Since the size of the tree structure does not vary during the execution of the generated program and only strictly necessary memory locations are allocated for this structure, the overhead of three pointers at each node to perform the required threading is very small. Furthermore, no recursive algorithms are required to traverse the tree structure, and a traversal can begin at any node because of the random access feature of the array where the tree structure is kept.

The way the tree structure is built and numerical identifiers are associated to blobs turn the algorithm to find the nearest common ancestor of two given blobs into a very simple one. The search for the nearest common ancestor starts from one of the two given nodes. The links of

ancestors are followed until a blob with a lower numerical identifier than the numerical identifiers of the two given blobs is found. The first blob which satisfies the above condition is the blob searched for.

Another important data structure for the understanding of how the statechart engine works is a list which keeps the global state of the system derived from a given statechart. The global state, also referred to as the configuration of a statechart, consists of all active blobs at a given instant. The numerical identifiers of all active blobs are kept in this list in increasing order. Whenever an event is handled (referred to as the current event in this context), this list is traversed to check a possible sensitivity of their elements to the current event. If a blob under consideration is the origin of a transition in condition to be fired, then the active subtree of the nearest common ancestor of the origin and the destination blob is deactivated and a new tree, which includes the destination blob is activated. During this procedure the references to deactivated blobs are removed from the list representing the global state and the references to just activated blobs are inserted. Once stable, the sensitivity checking is continued and applied to blobs belonging to the global state list and located after the last inserted blob reference. The statechart engine thus presents a deterministic behaviour.

Once the statechart engine has been put in action the following steps are performed: a) it locates the nodes representing the origin (A) and the destination blob (B) in the array holding the tree structure; b) it determines the nearest common ancestor (C) of those two blobs; c) it navigates from node A down to an active atomic node (D) reachable from node A; d) it traverses the tree from node D up to node C deactivating all blobs and all concurrent components along this path, except for the blob associated to node C; e) it then traverses the tree from node C to node B and activates all blobs represented by the corresponding nodes on that path as well as all concurrent siblings of blobs along this path; f) in order to get the statechart into a stable state, the activation process has to be continued until an atomic blob is reached descending via default sub-blobs or via the most recently visited sub-blobs in case a history attribute is being enforced.

Concurrent siblings are activated according to the order in which they occur in the circular sibling list. The implemented behaviour is thus deterministic mainly because of efficiency reasons, since the target machines of the generated code by the environment are sequential processors.

Activation of a blob means executing the script (i.e., calling the corresponding function) specified as its "on-entry" action and starting its "throughout" activity. On the other hand, deactivation means interrupting the associated activity and executing the "on-exit" action. Statecharts provide controlled deactivation and activation of contexts defined by blobs. Other considerations, like how processes exchange information, are left to the designer. The designer must be careful in order to avoid potential deadlocks. Strategies like temporarily suspending the insertion of new events in the event list have to be implemented too by the designer if, for example, an action demands a greater period of time to be concluded.

5. Event Binding

Many toolkits are available to support the design of human-computer interfaces and event-driven systems. The tool here described has been used in conjunction with the X Window System [Nye90]. The major problem is how to bind events handled by the underlying environment provided by the X toolkit with the event identifiers recognized by the code of generated programs. In general these toolkits keep the control of the system and take care of the event list.

For a better understanding of what part of an interactive program is subject to be modelled by a statechart, the Seeheim's logical user interface model [Gre86] is briefly described. This model divides a user interface into three layers: a presentation layer, a dialogue control layer and a layer which establishes the interface with the application proper. Ideally those layers should be loosely coupled to guarantee a certain degree of independence between successive layers and between the application and its user interface.

The user interacts with the presentation layer which represents visible symbols capable of being manipulated by the user via input devices. The presentation layer can be seen as the lexical level of the user interface, since its major role is to convert external stimuli into abstract internal representations (tokens) and tokens into visible or audible response reactions.

The dialogue control layer defines patterns of how a conversation between the user and an application takes place. This layer receives input tokens from the presentation layer and output tokens from the application via the third layer, i.e., the interface between the application and the entire user interface. Based on those streams the dialogue control layer determines how the conversation evolves possibly generating tokens for the surrounding layers in response to a change of the internal state of the user interface. The dialogue layer has to keep the current state of the user interface and has to have control over this state since the dialogue evolution depends on it. The dialogue control layer can be seen as the syntactic level of a user interface.

The third layer establishes the interface between the application and its front end, i.e., its user interface. It calls procedures of the application, passes the appropriate parameters, receives results and converts them into tokens recognizable by the dialogue control layer. From the user interface's viewpoint this layer represents the functionality of the application and from the application's viewpoint it represents the whole user interface. Thus, the third layer represents the semantic level of a user interface.

An "application" in the context of human-computer interfaces is fragmented by the designer into functions which implement the scripts associated to blobs and transitions. The code generated by the transformational tool stripped of these functions represents an intermediate dialog layer between the presentation and the application layer and its role is to set up and dismantle contexts in a controlled manner. Because the generated code takes over this role the architecture of the final system becomes better structured and easier to be maintained.

In X Window the interaction of the user with the application takes

place via widgets (predefined interaction elements like buttons, scroll bars and dialog boxes). An interaction with a widget generates an event. Once an event is handled by the kernel of the X toolkit, a specific function associated to this widget and referred to as a callback function is called. The system designer defines the code of this kind of function.

This mechanism provided by the X toolkit was used to establish the binding of a widget interaction with a specific event identifier, referred to as its logical identifier, i.e., its numerical identifier, used in a statechart specification. The binding is carried out by a call inside a callback function of the function which puts the statechart engine into action which carries as its actual parameter the corresponding logical identifier of the event to be used in the sensitivity checking of the blobs referred in the global state list.

5. Concluding Remarks

The statechart paradigm has been used in a particular class of reactive systems: the domain of human-computer interfaces. Because of efficiency reasons a deterministic statechart behaviour has been adopted. The paradigm proved capable of precise specification expressiveness of many event-driven control aspects as well as it required only more locally confined semantic action definitions due to the incremental context specification facilities provided by the statechart notation.

The use of the paradigm is backed up by a development environment, which transforms hybrid specifications (statecharts plus C functions) into functionally equivalent programs. The automatic transformation facility as well as debugging aids based on statechart animations carried out concurrently with the run of the corresponding programs intend to encourage the development and the maintenance at the highest abstraction level supported by the environment, i.e., the specification level.

At the moment possible extensions of the statechart notation are being investigated. The incorporation of some specific control features of human-computer interfaces into statechart specifications, like "undo" operations for instance, proved to be very tricky or impossible to be

modelled. The support of basic control frames of such recurrent features would turn the specification of such operations into a more "natural" task.

References

[Bat91] Batista, J.E.S. *Um Editor Gráfico para Statecharts*, MSc Thesis, ICMSC-USP, São Carlos, 1991.

[Can92] Cangussu, J.W.L. & Masiero, P.C. *Uma Linguagem para Execução Programada de Statecharts*, XIX Seminário Integrado de Software e Hardware, Rio de Janeiro, 1992, pp. 229-241.

[Dru89] Drusinsky, D. & Harel, D. *Using Statecharts for Hardware Description and Synthesis*, IEEE Transactions on Computer-Aided Design, Vol. 8, No. 7, July 1989, pp. 798-807.

[GSE92] Gonçalves Soares Elias, V. & Liesenberg, H. *Debugging Aids for Statechart-Based Systems*, Technical Report DCC-11/92, Department of Computer Science, IMECC/Unicamp, 1992.

[Fig91] Figueiredo Filho, A.G. & Liesenberg, H. *Geração de Gerenciadores de Sistemas Reativos*, V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, 1991, pp. 31-44.

[Gre86] Green, M. *A Survey of Three Dialog Models*, ACM Transactions on Graphics, Vol. 5, No. 3, July 1986, pp. 244-275.

[Har85] Harel, D. & Pnueli, A. *On the Development of Reactive Systems*, K.R. Apt, Ed., Logics and Models of Concurrent Systems (Springer, New York, 1985), pp. 477-498.

[Har87] Harel, D. *STATECHARTS: A Visual Formalism for Complex Systems*, Science of Computer Programming, Vol. 8, No. 3, June 1987, pp. 231-274.

[Leh84] Lehman, M.M. *A Further Model of Coherent Programming Processes*, IEEE Proc. Software Process Workshop, UK, February 1984.

[Mas91] Masiero, P.C., Fortes, R.P.M. & Batista, J.E.S. *Edição e Simulação do Aspecto Comportamental de Sistemas de Tempo Real*, XVIII Seminário Integrado de Software e Hardware, Santos, 1991, pp. 45-61.

[Nye90] Nye, A. & O'Reilly, T. *The Definitive Guides to the X Window System*, O'Reilly & Associates, Inc., 1990.

Relatórios Técnicos

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in d-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An (l, u)-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

08/92 **Maintaining Integrity Constraints across Versions in a Database,** *C. B. Medeiros, G. Jomier, W. Cellary*

09/92 **On Clique-Complete Graphs,** *C. L. Lucchesi, C. P. Mello, J. Szwarcfiter*

10/92 **Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms,** *T. Kowaltowski*

11/92 **Debugging Aids for Statechart-Based Systems,** *V. G. S. Elias, H. Liesenberg*

12/92 **Browsing and Querying in Object-Oriented Databases,** *J. L. de Oliveira, R. de O. Anido*

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

reltec@dcc.unicamp.br