Debugging Aids for Statechart-Based Systems[1]

Valéria Gonçalves Soares Elias
Hans Liesenberg

Relatório Técnico DCC–11/92

Novembro de 1992

Debugging Aids for Statechart-Based Systems[†]

Valéria Gonçalves Soares Elias[‡]
Hans Liesenberg[§]

Abstract

This paper describes a software development environment which transforms specifications defined in terms of statecharts into functionally equivalent programs in C. The environment encourages the user to maintain developed systems at the highest supported abstraction level, i.e. at the statechart specification level. This process is aided by a concurrent animation of the statechart specification with the execution of the corresponding functionally equivalent program. The events which drive the execution of the program are handed over to a statechart simulator either in real time or are registered in a log file and later submitted to the simulator. At the latter alternative, the user controls the firing of events which are used to animate the program's statechart at an appropriate pace. Those facilities may be used to validate developed systems as well as to pin-point design errors such as, for instance, infinite loops or deadlocks.

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

Introduction

Debugging facilities have become absolutely essential for the development and the maintenance of software of medium and high complexity. These facilities should be supported at the highest possible abstraction level used by the designer at the development stage. If they are provided only at lower levels, designers tend to take corrective actions at those levels and abandon descriptions at higher levels, which in turn become rapidly obsolete and inconsistent with lower level descriptions. Functional details of lower level descriptions are harder to grasp and thus their maintenance turns out to be more difficult. If C++ programs, for example, had to be debugged in C instead of C++, then the advantages brought by this extension of the C language would probably be outweighed by the debugging facilities the designer would have to depend on.

This paper presents a software development environment. This environment transforms specifications of event-driven software, described in a notation proposed by D. Harel [Har87], into programs in the C programming language. These programs are functionally equivalent to the specifications which they have been derived from. The specification notation favours control aspects, which can become very tricky and hard to manage, if an unstructured approach were adopted.

In order to provide some kind of support to the maintenance at specification level, two special debugging aids are provided by the environment. These aids, supported by a specification simulator, allow an animation of the specification used to generate an executable program. If this animation is carried out in real time, then the program and the specification simulator are run as two independent processes. In this case, events captured by the derived program are handed over to the simulator which, in turn, highlights context swappings due to those event occurrences. Since this animation may be performed too rapidly to be of any use to the designer, the captured events may also be registered in a log file and later on be submitted to the simulator and fired under control of the designer. These facilities are expected to support and encourage corrective actions at the specification level.

Transforming Specifications into Executable Programs

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 then 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 level into another at the immediately lower abstraction level. Automatic transformations are in general feasible at the lower end of an abstraction hierarchy. At higher levels usually little computational support is provided: either transformations are carried out manually or require manual interventions.

Statecharts

The environment under consideration supports the definition of a system in terms of context swappings due to event occurrences. This definition is formulated using two complementary languages: the semantic actions (which define what has to be done whenever such swappings take place) are defined by lower abstraction level descriptions in terms of functions in C code while event-driven control specifications are described via the bi-dimensional statechart notation. This notation consists of an extension of conventional state transition diagrams which supports incremental context definitions due to hierarchical decompositions, a history-based control flow, explicit definitions of concurrent behaviour as well as communication facilities between concurrent components. The description of statecharts presented here does not adhere in all aspects to Harel's formal description. A particular dialect, better suited to the domain where this notation has been practised, has been adopted.

A statechart consists of nested blobs, laid out as rectangles with rounded corners, which describe specific contexts incrementally. In the terminology proposed by Harel, a blob can be decomposed in two ways: into XOR-blobs or into AND-blobs. If the context of a given blob is active and it has been decomposed into XOR-blobs, then only one of its direct descendant blobs is activated as well. If, on the other hand, it has been decomposed into AND-blobs, then all of its direct descendants are activated whenever their ancestor is activated. Thus, AND-blobs are the means of describing concurrent behaviour explicitly. The outermost blob represents the system being described and, for the sake of the blob classification, it is considered to be a XOR-blob.

Context swappings under given restricting conditions at the occurrence of specific events are expressed in terms of transitions represented by directed arcs. The triggers (events and restricting conditions), which fire a given transition as well as specific actions to be carried out whenever a particular transition takes place, are represented as attributes of the corresponding arc. Actions associated to transition firings can be used, for instance, to generate new events which may have some effect on the context of concurrent components. The communication between concurrent components can thus be established by the generation of internal events.

Actions may as well be associated with blobs. In this case, they may be of three kinds: actions executed whenever the corresponding blobs are activated, actions executed whenever related blobs are deactivated, and those whose execution starts at the moment of the activation and are terminated on occasion of the deactivation of the blobs in question. Actions thus allow the aggregation of semantics to a statechart specification.

In addition to actions, a history attribute may also be associated with blobs. History attributes guide the blob activation process and may be of two kinds: flat or in depth. History attributes at lower levels override ones at higher levels of statechart topology. Whenever a blob with a flat history attribute is activated, then its most recently visited direct descendant is reactivated. If this attribute indicates a in-depth history, then this procedure is carried on until an atomic blob is reached or until this attribute is overridden or cancelled. If no history is being enforced at the activation of a blob or it is the first time this blob is being activated, then its default descendant is activated. Whenever a blob is decomposed into XOR-blobs exactly one of its direct descendants must be selected as its default descendant.

If a statechart is to be exercised, then an activation process is started from the outermost blob. Once this process is concluded, the statechart is capable of reacting to events. The set of active blobs of a statechart in a stable situation is referred to as the configuration or as the global state of this statechart. Whenever a transition is fired, the statechart configuration is affected as a consequence of the resultant context swapping.

During a transition the following happens conceptually. First, all blobs, starting from an atomic blob reachable from the origin blob up to the nearest common ancestor of the origin and the destination blob (excluding the last one), have to be deactivated as well as all concurrent siblings and their descendants along this path. Next the blobs on the path from the nearest common ancestor down to the destination blob, excluding the first one, have to be activated. The activation process is sustained from this point onwards until an atomic blob is eventually reached. However, below the destination blob possible history enforcements are now taken into account. The same happens to all concurrent siblings come across along the activation process.

The Development Environment

These follows a very short description of the event-driven software development environment. The environment consists of three major tools: a graphic statechart editor, a statechart simulator and an application generator. The first two are closely coupled and share the same data structure in main memory. Both are encapsulated by a same module. The third one is loosely coupled with the prior ones, i.e. the module which encapsulates the first two tools hands over information to the third tool via a file. This file contains a textual description of the statechart edited by the first tool, possibly validated by the second one, to be transformed eventually into a program in C by the third one.

The structure of the architecture is due to historical reasons. The first two tools were developed originally at USP/São Carlos [Bat91, Can92, Mas91]. The application generator [Fig91] was developed from scratch at the same time as the imported tools were adapted and extended to take over the role of the front end of the environment. In order to avoid conflicting situations it was decided that an intermediate, human-readable code would be used to ship over all information relevant to the transformation process.

The application generator uses conventional compiling techniques and produces code which is partially dependent on the particular statechart topology submitted to the application generator. The invariant part of the generated code consists of the implementation of the event-driven control process which is kept as closely as possible to the conceptual behaviour of the deactivation/activation process which describes what is going on at a transition firing. The code can be generated enabling or not a debugging mode option. Further details of the application generator and the editor/simulator are beyond the scope of this paper.

Debugging Aids

As already stated earlier, some debugging aids had to be provided in order to encourage an effective use of the statechart specification during the live cycle of a system whose development has been backed up by the environment described. If no support is given at the statechart level, then the designer has to resort to the C debugging facilities. In this case, the designer would be tempted to fix design faults at the C code level since he would have been forced to understand the behaviour of the system at this level in order to be capable of finding errors. Under those circumstances the fault fixing at the statechart level would still require an additional effort of tracing back the errors found in the C code to the corresponding point at the statechart specification level.

To solve this problem, the environment provides two debugging aids: a real time animation of the statechart specification (which can be performed concurrently with the execution of the corresponding code produced by the application generator), and an animation controlled by the user based on events occurred and logged during a run of the corresponding code. These animation facilities are very useful to detect among others the contexts where a system got stuck in an infinite loop or where a deadlock happened.

The animation is carried out by the statechart simulator, either in real time or from a log file. The first alternative may result in an hectic pace of the animation, which may give the designer no chance to comprehend fully what is going on. The second alternative overcomes this difficulty and allows a "slow-motion" animation controlled by the designer. If this debugging facility is selected, then the events captured and handled during a execution of the generated code are written into a log file. At a next stage the log file is submitted to the statechart simulator which sets up the corresponding statechart and reacts to the registered events under the control of the user. The events are consumed in the same order in which they have been registered in the log file.

The first alternative required a more sophisticated implementation. If this alternative is selected, then the generated code as well as the simulator run as two independent processes. The processes are created by the application generator. The simulator is run as a server and the generated code as a client. The communication between these two processes is accomplished via sockets [NPG90]. When the designer selects the real time animation mode, the simulator creates its socket and it remains blocked until the connection with its client is requested. In order to avoid that unrelated processes become connected to this particular socket, the simulator only accepts connection requests which carry the identifier of the statechart to be animated. When this happens, the simulator loads the specific statechart topology from the corresponding file and activates it by default. If, on the other hand, a connection request does not carry the proper "password", then it is rejected by the server.

The generated program, i.e. the client, makes use of a function called `SendEvent()` to ship off event identifiers to the simulator. The first time this function is called, the socket of the client is created and the "password" is sent to the server. After the connection request has been accepted, the established connection can be used to pass event identifiers to the server as soon as they are handled by the client.

Whenever the server receives a blob identifier, it sweeps across the blobs belonging to the current configuration of the statechart being animated. If those blobs may be taken as origins of transitions capable to be fired, then the deactivation/activation procedure is carried out. The highlights of deactivated blobs are removed while just activated blobs are put in highlights. The designer thus receives feedback information about the configuration evolution along the running of the corresponding program. This kind of information is of great value to understand subtle nuances of the dynamic behaviour of the program derived from a statechart and makes the precise identification of a fault at the specification level easier.

Conclusions

The availability of proper debugging facilities at the highest supported abstraction level may be one of the decisive factors to choose a software development environment. If such an enviroment lacks those facilities, it might even so be selected because of other outstanding qualities. In this latter case, however, higher level descriptions tend to be gradually neglected since an additional effort has to be made to keep those descriptions consistent with what is going on at lower levels where more adequate debugging facilities are available.

Two debugging aids of a software development environment based on specification animations have been outlined. In this particular case, specifications are described in terms of statecharts and are complemented with lower level action descriptions. These aids intend to encourage designers to maintain their applications at the highest supported application description level, i.e. the statechart specification.

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.

[Fig91] Figueiredo Filho, A.G. *Um Processo de Síntese de Sistemas Reativos*, MSc Thesis, IMECC/UNICAMP, 1991.

[Har87] Harel, D. & Pnueli, A. *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.

[NPG90] *Network Programming Guide*, Sun Microsystems, Inc., 1990.

Relatórios Técnicos