FROM FINITE STATE MACHINES TO COMPLEX [1] REACTIVE SYSTEMS WITH VISUAL FORMALISMS

Carl W. Steidley, Jeffrey W. Roule
Department of Computer Science
Southeastern Louisiana University
Hammond, Louisiana 70402

Abstract

It is well known, that a digital computer stores information internally in binary form. At any instant, the computer contains certain data, so its internal storage is set in certain patterns of binary digits. We call this the state of the computer at that instant. Since the computer contains a finite amount of storage, there is a finite (although large) number of states that the computer can assume, thus a computer can be formally and abstractly defined and represented with a finite state machine of the form M=[S,I,O,fs,fo] where S is a finite set of states, I is the finite input alphabet, O is the finite output alphabet, fs:SxI->S, and fo:S->O. Another and generally more accessible way to define such machines is the visual formalism of a directed graph called a state graph or more often a state-transition diagram (or state diagram for short).

Generally, a transformational system is specified by a transformation or function, so that an input/output relationship is usually considered a sufficient specification. A reactive system, in contrast with a transformational system, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli.

There has been a major problem in the specification and design of large and complex reactive systems. This problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and rigorous enough to be amenable to detailed computer simulation.

In this paper we will describe a broad extension of the conventional formalism of state machines and their visual formalism, state diagrams, developed by Harel, that is relevant to the specification and design of complex, discrete-event systems, such as multi-computer real-time systems, communications protocols, and digital control units.

Introduction

In a naive sense it may be said that computer scientists are interested in solving two questions: What problems can be solved with a computer, and if a problem is theoretically solvable, what is the most effective way to solve it? Answers to the first question arise through development and study of models of computation. During the undergraduate experience we introduce students to the foundations of computation, which address the first question through abstract models of the computer. This model is frequently the finite state machine or finite state automaton. Finite state

---

[1.] Partial support for this project was provided by the National Aeronautics and Space

machines are of practical interest as well as theoretical interest since they are commonly used in compilers to perform lexical analysis.

We generally introduce the finite state machine in a formal mathematical fashion such as: M=[S,I,O,fs,fo] where S is a finite set of states, I is the finite input alphabet, O is the finite output alphabet, fs:SxI->S, and fo:S->O. Before long however we may find many of our students, if not perplexed, bored by the mathematical abstraction and we find much sagacity in the old student adage that "a picture is worth a thousand mathematical equations", thus, we find ourselves using state transition diagrams to illustrate our finite state machines.

Transformational versus Reactive Systems

Many kinds of data processing systems are transformational systems, that is, the system is defined by a specifying transformation or function. Usually an input/output relation is sufficient. The timing of the inputs and outputs is fairly predictable. Such a system repeatedly accepts inputs , carries out some processing, and outputs the results when the processing is done. While transformational systems can be highly complex, there are a number of methods that allow one to decompose the system's functional or transformational behavior into ever-smaller parts in ways that are both coherent and rigorous.

A reactive system, in contrast with a transformational system, is characterized by being, to a large extent, event driven, continuously having to react to external and internal stimuli. Examples of reactive systems include: automobiles, missile and avionics systems, communication networks, and operating systems. The literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification and design of large complex reactive systems.[see 1,2,3,4,5] The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and sufficiently rigorous to be amenable to detailed computer simulation. The behavior of a reactive system is really the set of allowed sequences of input and output events, conditions and actions, perhaps with some additional information such as timing constraints. The problem is made especially acute in that a set of sequences, usually a very large and complex set, does not seem to lend itself naturally to friendly gradual, level-by-level descriptions, which would fit nicely into a human being's frame of mind.

Visual Formalisms

States and events are a natural medium for describing the dynamic behavior of both transformational and reactive systems. Such systems would be comprised of fragments of the following form "when event x occurs in state X, if condition A is true at the time, the system transfers to state Y." Finite state machines and their corresponding state-transition diagrams (or state diagrams for short) are a visual formalism for collecting all such fragments of a system into a whole. (see Figure 1.)

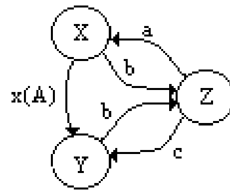


Figure 1

It is difficult, if not impossible, however, to represent a complex system in this fashion due to the unmanageable, exponentially growing multitude of states, all of which are illustrated without the benefit of strata, resulting in a chaotic state diagram without structure. To be useful, a state/event approach, just like a good software system, must be modular, hierarchical, and well-structured.

Statecharts

According to Harel, [6] a good state/event approach should also cater naturally to more general and flexible statements, such as

(1) "in all airborne states, when yellow handle is pulled seat will be ejected"
(2) "gearbox change of state is independent of braking system"
(3) "when selection button is pressed enter selected mode"
(4) "display-mode consists of time-display, date-display, and stopwatch-display"

In keeping with the "one picture is worth a thousand words" (or for that matter a thousand formal equations) aphorism, Harel offers statecharts as a way to extend the state/event formalism in ways that will satisfy the needs expressed above , and at the same time retain and improve the visual appeal of state diagrams. Statecharts constitute a visual formalism for describing states and transitions in a modular fashion, providing for clustering, orthogonality (concurrency), and refinement. Statecharts also encourage zoom capabilities for moving easily back and forth between levels of abstraction. That is,

statecharts = state-diagrams + depth + orthogonality + broadcast communication.

where: depth = hierarchy
orthogonality = independence or concurrency

Clustering and Refinement

Generally we teach the concept of hierarchy using trees or other such line graphical representations. There is, however, a real disadvantage in utilizing these representations. Since lines and points have no width, no advantage is taken of the area or location of the diagram.

Statecharts depict states as boxes and utilize encapsulation to depict hierarchy. Transition arrows are allowed to originate and terminate at any level and can be labeled with state change events and/or conditions. (see Figure 2)

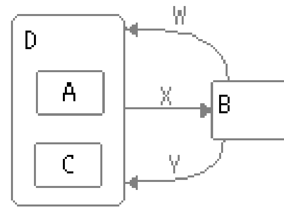


Figure 2

The semantic implication of the superstate depicted in Figure 2 is the exclusive-or of the two substates. Thus, the superstate depicted is an abstraction or clustering of the two substates. The outgoing transition arrow captures the common, and highly important property of statecharts, the two substates' transition arrows are replaced by the single transition arrow from the superstate indicating that the transition is from all substates. The statechart of Figure 2 could have been designed from another approach. That is, we may have arrived at the situation depicted in Figure 3.

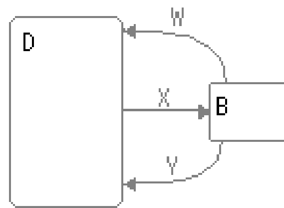


Figure 3

The superstate may then have been refined to include the two substates depicted in Figure 4.

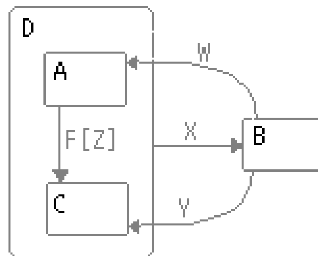
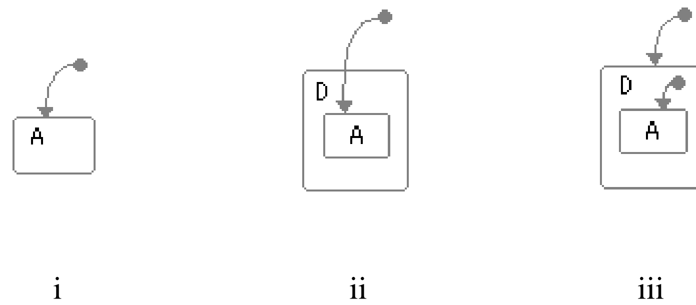


Figure 4

Of course the underspecification of the transitions and conditions would be realized and included in the diagram yielding the original statechart of Figure 2. Thus, clustering or abstraction is a bottom-up concept and refinement is a top-down concept and both depict the or-relationship between a state's substates.

Suppose that as far as the "outside world" is concerned, the default start state of a system is state A. Three methods of depicting this in statecharts are illustrated in Figure 5. Figure 5i depicts a direct entry to an open state. Figure 5ii depicts entry into a substate. Figure 5iii depicts a two-step form of Figure 5ii.



i ii iii

Figure 5

Independence and Concurrency

The concept of concurrency, that is the concept of two processes brought together to interact, where the interactions may be regarded as events that require simultaneous participation of both processes, can be difficult to model. For example, consider the diagram of Figure 6.

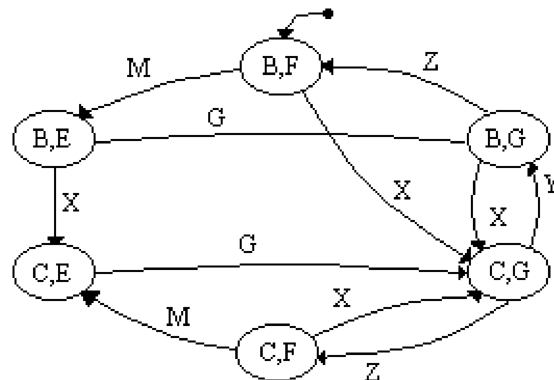


Figure 6

In the study of automata we may refer to the situation depicted in Figure 6 as the orthogonal product of states. That is, where we considered the OR of states in the last section, now we will consider the decomposition of the AND condition. The concept of orthogonal product of states captures the notion that to be in a given state a system must be in all of its AND components.

A statechart implementation of the system depicted in the figure above is given in Figure 7.

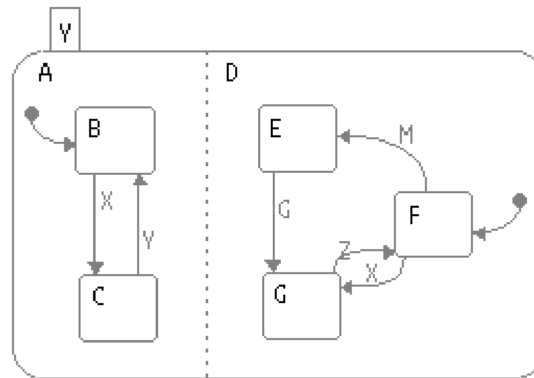


Figure 7

An orthogonal product statechart is depicted by a dashed line to split a state box into components. Figure 7 depicts a super-superstate Y consisting of two AND component superstates A and D with the property that being in the super-superstate entails being in some combination of one of the two states of superstate A with one of the three states of superstate D. In this case, a super-superstate is the orthogonal product of the two super states. Without further information, entering the super-superstate from the outside is equivalent to entering through the default arrow to both of the states indicated. If the first event occurs, it transfers to the next two states simultaneously. This illustrates a type of synchronization: a single event causing two simultaneous happenings.

We have considered the hierarchy, orthogonality, and broadcast features of statecharts since these are the features necessary to clearly specify and model complex reactive systems. However, while beyond the scope of this paper, statecharts include condition and selection entrances, delays and timeouts, and unclustering features as well. A full semantic description of statecharts can be found in [9,10].

Modeling With Formal Methods

The main stages in the development life cycle of a system are requirements analysis and specification, design, implementation, testing, and maintenance. In both the requirements analysis and specification, and design stages developers specify the system under development in terms of its functional capabilities and its behavior over time, and determine the main subsystems

that implement these capabilities. The resulting specification is called the system model.

Since correcting specification errors and misconceptions discovered during later stages of the system's life cycle is extremely expensive, it is commonly agreed that a thorough comprehension of the system and its behavior must be achieved in the specification stage and that extensive analysis should be carried out as early as possible. This is important for all the participants in the development of the system. If a clear and comprehensive model is constructed early, the customer can become acquainted with and can approve of the functionality and behavior of the system before investing heavily in the implementation stages. Precise and detailed models are also in the best interests of the designers and the testers of the system. Moreover, the specifiers themselves use modeling as the main medium of expressing their ideas, and exploit the resulting models in analyzing feasibility and operational issues. This is particularly true for reactive systems, whose behavior can be complex, causing the specification problem to be particularly elusive and error prone. Thus, building a model can be considered a transition from concepts and ideas to concrete descriptions. The concepts underlying a system are captured in three views: functional, behavioral, and structural.

The functional view captures the “what”. It describes the system’s functions and processes, also called activities, thus pinning down its capabilities. This view also includes the inputs and outputs of the activities, i.e., the flow of information to and from the external environment of the system and among the internal activities. The behavioral view captures the “when”. It describes the system’s behavior over time, including the dynamics of the activities, their control and timing behavior, the states and modes of the system, and the conditions and events that cause modes to change and other things to happen, thus, it also provides answers to questions about causality, concurrency, and synchronization. The structural view captures the “how”. It describes the subsytems and modules constituting the real system, and the communication between them. The functional and behavioral views provide the conceptual model while the structural view is considered to be the physical model.

I-Logix Corporation of Andover, Massachusetts has created a software tool which incorporates these three views of a system model. The system, called STATEMATE, uses three graphical languages: the first called activity-charts is for modeling the functional view, the second incorporates Harel’s statecharts visual formalism for modeling the behavioral view, and the third, called module-charts, is for modeling the structural view.

Some of the basic ideas that make up STATEMATE languages have been adapted from other modeling languages, such as data-flow diagrams, state-transition diagrams, data dictionaries and mini-specs. However the languages of STATEMATE include many extensions that increase the expressive power and simplify and clarify the model. The languages of STATEMATE have formal semantics. The general visual style as well as many of the conventions and syntax rules are common to all three of the STATEMATE languages.

At this point the reader may ask the question, “What can we get out of this model beyond the benefits of careful thinking that come implicitly from the process of its construction?” The STATEMATE package provides for analyzing the model by syntax checking, the ability to

execute or run the model, and to view graphically the system's response using its simulation feature. A simulation control language is part of the system providing the user with control over how the executions proceed, yet allowing for the exploitation of the power of the tool to take over many of the details. The simulation feature thus provides for debugging, deeper analysis, and exhaustive scenario testing.

Finally, one of the most far reaching benefits of a model that is precise and comprehensive is the ability to translate it in its entirety into runnable code. In STATEMATE this is done using its prototyper package, which can be instructed to automatically translate an Activity-chart with all of its descendants, including their controlling Statecharts, into a high level language such as Ada or C. The main use of the resulting code is in observing the system performing under circumstances that are as close to the real world as one wants. For example, the prototype code can be executed in a full-fledged simulator of the target environment or in the final environment itself. The code produced should be considered to be prototypical, and not necessarily production or final code. Consequently, it might not always reflect accurate real-time performance of the intended system. Nevertheless, it runs much faster than the animated simulations, and hence is useful for testing the system's performance in close to real circumstances.

Example

For demonstration purposes we will include two examples given by Hoare in his book Communicating Sequential Processes (CSP)[12]. For comparison and contrast we will implement the examples first in FDR and then in statecharts. (The automated formalism Failures Divergence Refinement (FDR)[13] was developed from CSP.)

The first of these examples is a simple vending machine which will deliver a chocolate for each coin of proper denomination deposited in the machine. The formal FDR expression for this example is as follows:

VM = coin -> choc -> VM

A statechart implementation of this example follows:

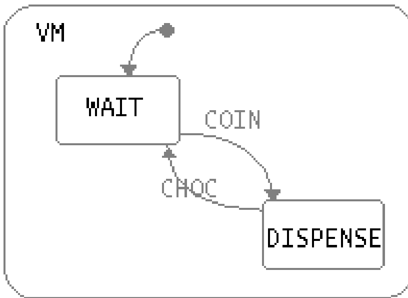


Figure 8

While certainly more illustrative, the statechart implementation is probably no more informative than the FDR expression for the mathematically literate.

However, the FDR implementation of a slightly more complicated machine, one offering the use of a choice of coins to insert and a selection of candies to dispense, is much more complicated using more specialized notation and requiring a slightly more sophisticated understanding of mathematical notions.

```
VMC = (in2p -> (large -> VMC) [] (small -> out1p -> VMC))
    [] (in1p -> (small -> VMC) [] (in1p -> large -> VMC) []
     (in2p -> out1p -> large -> VMC) [] (in1p -> STOP))
```

This example, definitely of British origin, models a more complex vending machine (VMC). In the example, VMC waits for a coin to be input. Upon insertion of a one penny coin, a large or small chocolate is selected. If a small chocolate is selected, it is dispensed and VMC returns to a waiting state. If a large chocolate is selected VMC waits for another coin to be input. If the second coin is a one penny coin, VMC dispenses a large chocolate and returns to a waiting state. If the second coin is a two penny coin, VMC returns a one penny coin in change, dispenses a large chocolate, and returns to a waiting state.

Upon insertion of a two penny coin, if a small chocolate is selected, VMC returns a one penny coin in change, dispenses a small chocolate, and returns to a waiting state. If a large chocolate is selected, VMC dispenses a large chocolate and returns to a waiting state.

The statechart implementation of this more complicated vending machine is, we believe, much more intuitive.

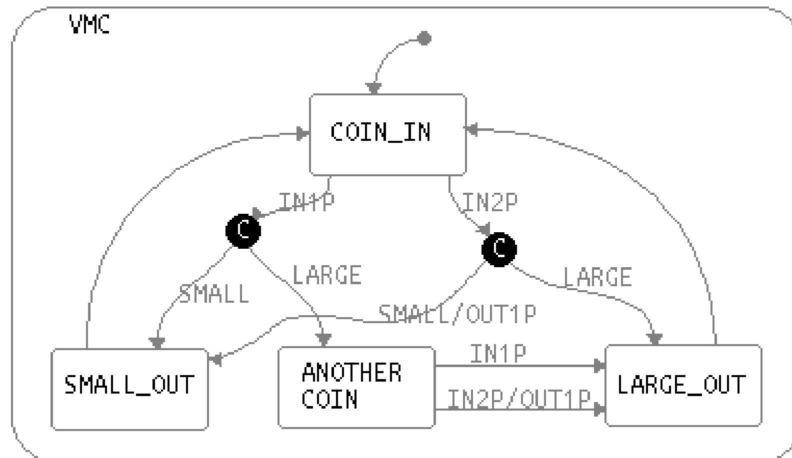


Figure 9

Conclusions

The broad use of the state/event approach, in the form of finite state machines or state transition diagrams, for the specification of systems is attested to by the literature. This approach has been recommended for the specification of the following: user interface of interactive software [4,5], data processing systems [2], hardware system description [1], communication protocols [8,9], and computer aided instruction [3].

One of our jobs as educators is to teach students that there is no magic solution to the problem of designing large and complex systems, except to use carefully fashioned methods, languages, and formalisms that enable the work to be carried out gradually, module by module and level by level[11]. Further, students should understand that the descriptions resulting from these practices ought to be sufficiently coherent and precise to be both useful to an observer and amenable to computer simulation, thus enabling as much computerized support in formal verification and analysis as possible.

References

1. M.D. Edwards and D. Aspinall, The synthesis of digital systems using ASM techniques, in: T. Uehara and M. Barbacci, Eds, Computer Hardware Description Languages and their Applications, North-Holland, Amsterdam, 1983, pp 55-64.

2 A.B. Ferrentino and H. D. Mills, State machines and their semantics in software engineering, Proc. IEEE COMPSAC'77 Conference (1977) pp 242-251.

3. S. Feycock, Transition diagram-based CAI/HELP systems, International Journal Man-Machine Studies, 9, pp 399-413, 1977.

4. R.J.K. Jacob, Using formal specifications in the design of a human-computer interface, Communications of the ACM, 26, pp 259-264, 1983.

5. D.L. Parnas, On the use of transition diagrams in the design of a user interface for an interactive computer system, Proc. ACM Conference, pp 379-385, 1969.

6. D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming, North-Holland, pp 231-274, 1987.

7. C.A. Sunshine et al, Specification and verification of communication protocols in AFFIRM using state transition models, IEEE Transactions on Software Engineering, Vol 8, pp 460-489, 1982.

8. A.S. Tannebaum, Computer Networks, Prentice Hall, Englewood Cliffs, NJ, 1981.

9. D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman, On the formal semantics of statecharts, Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, 1987.

10. The Semantics of Statecharts, iLogix, Inc. Andover, MA.

11. Steidley, Carl W. , Roule, Jeffrey, A Tool for Teaching Visual Formalisms, submitted Journal for Computing in Small Colleges.

12. C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall Internationla Ltd., New York, 1985.

13. Failures Divergence Refinement, User Manual and Tutorial, Formal Systems (Europe) Ltd., 1995.

Biographical Information

CARL STEIDLEY is a Professor of Computer Science. He earned his Ph.D. at the University of Oregon. His interests are in the applications of artificial intelligence and robotics. Prior to joining the faculty at Southeastern La. Univ. he taught computer science at Central Washington Univ., and Austin Peay State Univ. Before that he was a member of the faculty of mathematics and physics at Oregon Institute of Technology. He has had recent research and development appointments at NASA Ames Research Center and Oak Ridge Natl. Labs.

JEFFREY ROULE is a senior majoring Computer Science at Southeastern Louisiana University. He has been Steidley's research assistant, funded by the NASA JOVE program, for one year.