DOI:10.1145/1467247.1467274

How avionics work led to a graphical language for reactive systems where the diagrams themselves define the system's behavior.

BY DAVID HAREL

Statecharts in the Making: A Personal Account

WRITING A HISTORICAL paper about something you yourself are heavily involved in is clearly difficult; the result is bound to be personal and idiosyncratic and might well sound presumptuous. I thus viewed an invitation to write about statecharts for the third History of Programming Languages conference in 2007 as an opportunity to put the language in a broader perspective.6 The present article is a greatly abridged version of the resulting conference paper. The implicit claim is that the emergence of the language brought to the forefront a number of ideas that today are central to software and systems engineering, including the general notions of visual formalisms, reactive systems, model-based development, model executability, and full code generation.

In 1979 I published a paper on a tree-like language based on the idea of alternation called "And/Or Programs,"[18] prompting Jonah Lavi of Israel Aircraft Industries (IAI) to contact me. We met in late 1982, at which time I'd been on the faculty of the Weizmann Institute of Science for two years. Lavi, who was a methodologist responsible for evaluating and recommending software engineering methods and tools at IAI, described some problems IAI avionics engineers were having. This was part of the massive effort then under way to build a fighter aircraft, the Lavi (no connection with Jonah's surname). Following that meeting, I began consulting at IAI on a one-day-a-week basis, and for several months Thursday became my IAI day.

The first few weeks were devoted to trying to understand the general issues from Lavi. Then it was time to be exposed to the details of the project and its specific difficulties, since I hadn't yet met the project's engineers. For several weeks, I spent my Thursdays with Lavi's assistant, Yitzhak Shai, and a select group of

experts from the Lavi avionics team, notably Akiva Kaspi and Yigal Livne.

An avionics system is a wonderful example of what my colleague at Weizmann Amir Pnueli and I later identified as a reactive system.[17] The main behavior that dominates such a system is its reactivity, that is, its event-driven, control-driven, event-response nature. The behavior is often highly parallel and includes strict time constraints and possibly stochastic and continuous behavior. A typical reactive system is not predominantly data-intensive or algorithmic in nature. Behavior is the crucial problem in its development—the need to provide a clear yet precise description of what the system does or should do over time in response to both external and internal events.

The Lavi avionics team consisted of extremely talented people, including those involved in radar, flight control, electronic warfare, hardware, communication, and software. The radar people could provide the precise algorithm used to measure the distance to a target. The flight-control people could talk about synchronizing the controls in the cockpit with the flaps on the wings. The communications people could talk about formatting information traveling through the MuxBus communication line. Each had his own idiosyncratic ways of thinking about and explaining the system, as well as his own diagrams and emphases.

I would ask seemingly simple questions, such as: "What happens when this button is pressed?" In response, a weighty two-volume document would be brought out and volume A would be opened to page 389, clause 6.11.6.10, which says that if you press that button such then such a thing would occur. At which point (having by then learned some of the system's buzzwords) I would say: "Yes, but is that true even when an infrared missile is locked on a ground target?" To which someone might say, "Oh no, in volume B, page 895, clause 19.12.3.7, it says that in such a case this other thing happens." These Q&A sessions would continue, and when it would get to the fifth or sixth question the engineers were no longer sure of the answer and would have to call the customer (the Air Force people) for a response. By the time we got to the eighth or ninth question even the customer didn't have an answer.

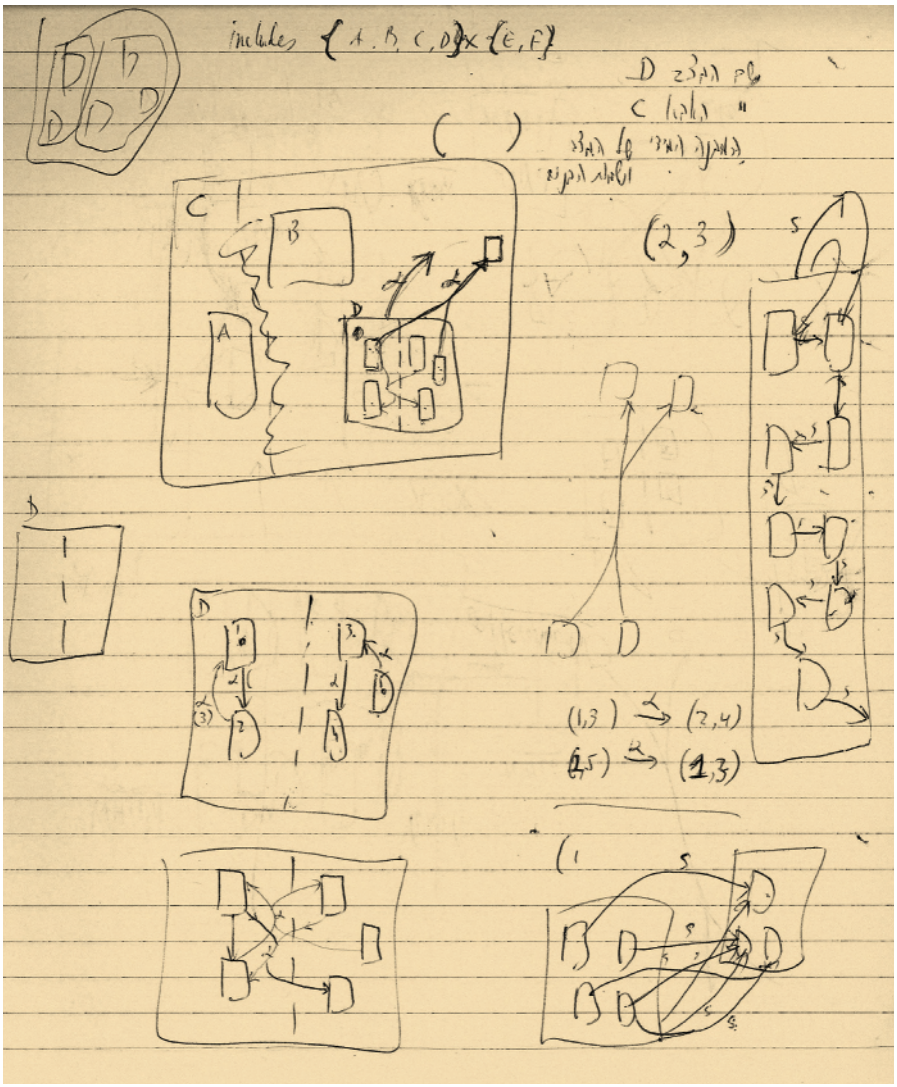

Figure 1: Page from my early IAI notes (1983). Statechart constructs include hyper-edges, nested orthogonality (a kind of concurrency), and transitions that reset a collection of states (chart on right). Note the use of Cartesian products of sets of states (set-theoretic formulation at the top) to capture the meaning of the orthogonality and the straightforward algebraic notation for transitions between state vectors (lower right).

Obviously, someone would eventually have to decide what happens when you press a certain button under a certain set of circumstances. However, this person might turn out to be a low-level programmer assigned to write some remote part of the code, inadvertently making decisions that influenced crucial behavior on a much higher level. Coming, as I did, from a clean-slate background in terms of avionics (a polite way of saying I knew nothing about the subject), this was shocking. It seemed extraordinary that such a talented and professional team knew in detail the algorithm used to measure the distance to a target but not many of the far more basic behavioral facts involving the system's response to a simple event.

To illustrate, consider the following three occurrences of a tiny piece of behavior buried in three totally different locations in a large specification of a chemical manufacturing plant:

"If the system sends a signal hot then send a message to the operator";

"If the system sends a signal hot with T >60° then send a message to the operator"; and

"When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when T <60°."

Despite my formal education in mathematical logic, I've never been able to understand the third item. Sarcasm aside, the real problem is that all three were obviously written by three

different people for three different reasons. It is almost certain that the code for this critical aspect of the chemical plant would be problematic in the best case and catastrophic in the worst. The specification documents the Lavi avionics group had produced at IAI were no better. If anything, they were longer and more complex. Some subcontractors even refused to work from them, claiming they were incomprehensible, inconsistent, and incomplete.

This confusion prompted the question: How should an engineering team specify the behavior of such a complex reactive system in an intuitively clear yet mathematically rigorous fashion? This was what I aimed to try to answer.

How should an engineering team specify the behavior of such a complex reactive system in an intuitively clear yet mathematically rigorous fashion? This was what I aimed to try to answer.

Statecharts Emerging

My initial goal was not to invent a language but to try to recommend, from the toolset of the software and systems engineer, appropriate means for saying what the avionics engineers seemed to already have in mind. It turned out they really did understand the system and could answer many questions about behavior but often hadn't thought through things properly because the information wasn't well-organized in their documents (or heads). I had to spend a lot of the time getting them to talk; I kept asking questions, prodding them to state clearly how the aircraft behaves under certain sets of circumstances. We would then brainstorm, trying to make sense of the information that had piled up.

It was clear from the start that the basic idea of states/modes was fundamental to their way of thinking. (This insight was consistent with the work of David Parnas on the avionics of the A-7 jet fighter.[19]) The IAI avionics engineers would repeatedly say things like, "When the aircraft is in air-ground mode and you press this button, it goes into air-air mode, but only if the radar is not locked on a ground target." This is familiar to anyone in computer science; what we have here is really the likes of a finite-state automaton with its state transition mechanism. Nevertheless, having one big state machine describing what is going on would be fruitless, not only because of the exponentially growing number of states but also because simply listing all possible states and the transitions leading from one to the other is unstructured and nonintuitive; it provides no means for modularity, hiding of information, clustering, and separation of concerns and would not work for highly complex behavior. I was quickly convinced of the need for a structured and hierarchical extension of the conventional state machine formalism.

Following my initial attempt to use temporal-logic-like notation, I resorted to writing down the state-based behavior textually, in a kind of structured state-based dialect of "state protocols" made up on the fly (see the figures in [6]). The dialect was hierarchical; within a state there could be other states, and if you were in this state and the event occurred, you would enter the other state, and so on. As this went on, things would get increasingly complicated. The avionics engineers would bring up more of the system's behavior, and I would respond by extending the state-based structured description, often having to enrich the syntax in real time.

When the multitude of emerging behavioral details caused things to be even more complicated, I would doodle on the side of the page to explain (visually) what was meant. I recall the first time I used visual encapsulation to illustrate for the engineers the state hierarchy and an arrow emanating from the higher level to show a compound "leave-any-state" transition. I also recall the first time I used side-by-side adjacency with a dashed separator line to depict orthogonal (concurrent) state components (see Figure 1).

I drew these informal diagrams in order to explain what the nongraphical text-based state protocols meant. The text was still the real thing, however, and the diagrams were merely an aid. But after a while it dawned on me that everyone around the table seemed to understand these back-of-the-napkin diagrams much better, relating to them far more naturally. The pictures simply did a much better job of setting down on paper the system's behavior, as understood by the avionics engineers, and we found ourselves discussing the diagrams and arguing about the avionics over them, not over the state protocols. Still, the mathematician in me found this difficult to accept; I told myself that doodled diagrams could not really be better than a real mathematical-looking artifact. So it really took a leap of my own faith to be able to think: "Hmmm...couldn't the

pictures be turned into the real thing, replacing, rather than supplementing, the textual structured-programming-like formalism?" So I gradually stopped using the text or used it only to capture supplementary information inside the states or along transitions, and the diagrams became the actual specification we were constructing.

This process of turning the diagrams into the specification language had to be done in a disciplined way, making sure the emerging pictures were not just pictures. You couldn't just throw in features because they looked good and the avionics team seemed to understand them. Unless the exact mathematical meaning of an intended feature was given—in any allowed context and under any allowed set of circumstances—it simply couldn't be considered. This was how the basics of the language emerged. I chose to use the term "statecharts" for the resulting creatures, which at the time was the only unused combination of "state" or "flow" with "chart" or "diagram" (see Figure 2).

The pilot stood there studying the blackboard for a couple of minutes, then said, "I think you have a mistake down here; this arrow should go over here and not over there." He was right.

Comments on the Language

Besides a host of secondary constructs, the two main ideas in statecharts—hierarchy and orthogonality—can be intermixed on all levels (see the figure). The language can be viewed as beginning with classical finite-state machines and their diagrams and extending them through a semantically meaningful hierarchical substating mechanism and through a notion of orthogonal simultaneity. Both extensions are reflected in the graphics—hierarchy by encapsulation of the blobs depicting states and orthogonality by partitioning a blob using dashed separator lines. Orthogonal components of the state space cooperate in several ways, including direct sensing of the state status in another component or through actions. The mechanism within a statechart thus has a broadcasting flavor whereby any part of the (same) statechart can sense what is happening in any other part.

As a result of the new constructs for hierarchy and orthogonality and their graphical renditions, transitions become far more elaborate and rich than in conventional state machines. They are still labeled with their triggering events and conditions but can now start or stop at any level of the hierarchy, cross levels, and in general be richer than standard source-target; they can be full hyperedges, since both sources and targets of transitions can be sets of states. At any given point in time a statechart is in a combination (vector) of states, the length of which is not fixed, since entering and exiting orthogonal components on various levels of the hierarchy changes the size of the state vector dynamically (see the nongraphical portions of the figure). Default states generalize start states, and the small arrows leading to them can be level-crossing and hyperedge in nature. In addition, statecharts also have special history connectors, conditions, output events, selection connectors, and more.

The fact that the technical part of the statechart story started out with And/Or Programs[18] is interesting and relevant. Encapsulated substates represent OR (actually XOR, exclusive or), and orthogonality is AND. Thus, a minimalist might view statecharts as a state-based language with an underlying structuring mechanism of classical alternation.

As for the graphic renditions, the two novel visual constructs in statecharts—blob encapsulation and partitioning—are both topological in nature and therefore worthy companions to edges in graphs. Indeed, when designing a graphical language, topology should be used whenever possible, since it is a more basic branch of mathematics than geometry. Being inside something is more fundamental and robust than being smaller or larger or than being a rectangle or a circle. Being connected to something is more basic than being green or yellow or being drawn with a thick line or a thin line. The human visual system notices and comprehends such things immediately.

Still, statecharts are not exclusively visual/diagrammatic. Transitions can be labeled not only with the events that cause the transitions but also with the conditions that guard against taking them and the actions (output events) that are to be carried out when they are taken. Moreover, statecharts borrow from both the Moore and the Mealy variants of state machines, allowing actions on transitions between states, as well as on entrances to or exits from states. Statecharts also allow "throughput" conditions attached to a state and are to hold through the entire time the system is in that state.

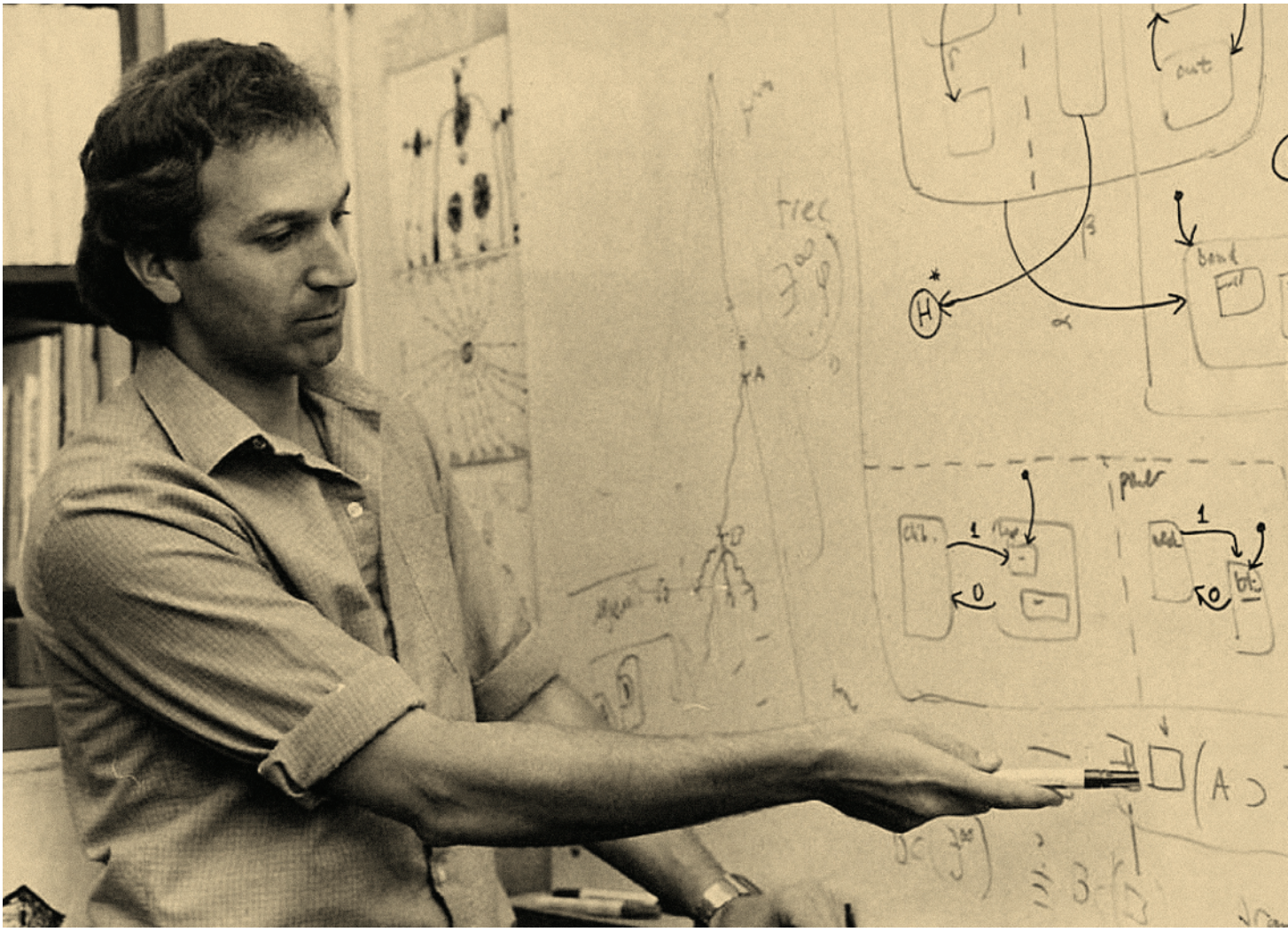

Figure 2: Explaining statecharts (1984); note the temporal logic bottom right.

Speaking in the strict mathematical sense of power of expression, hierarchy and orthogonality are but helpful abbreviations and can be eliminated; the hierarchy can be flattened, writing everything out explicitly on a low level, and orthogonality can be removed by taking the Cartesian product of the components (as in the top of the figure). Thus, these features do not add raw expressive power, and their value is reflected mainly in additional naturalness and convenience. However, they also (in general) provide great savings in size; for example, orthogonality can yield an exponential improvement in succinctness in both upper- and lower-bound senses.[3]

Incidentally, orthogonal state-components in statecharts do not necessarily represent concurrent or parallel components of the system being specified. They need not represent different parts of the system at all but can be introduced to help structure its state space and arrange the behavior in portions that are conceptually and intuitively separate, independent, and orthogonal. I emphasize the word "conceptually" because what counts is whatever is in the mind of the person doing the specification.

This motivation has many ramifications. I chose the broadcast communication mechanism of statecharts not because it is preferred for actual communication between a system's components. It is merely one way to coordinate the orthogonal components of the statechart, between its "chunks" of state-space, if you will; these will often not be the components—physical or software—of the system itself. Broadcasting is a way to sense in one part of the state space what is going on in another part and does not necessarily reflect actual communication between tangible aspects of the actual system. On certain levels of abstraction one often really wants to be able to sense properties of a part of the specification in another without worrying about implementation details. I definitely do not recommend having a single statechart for an entire system. Rather, as I discuss later, there will almost always be a decomposition of the system into functions, tasks, objects, and the like, each endowed with its own behavior (described by, for example, a statechart). In this way, the concurrency occurs on a higher level.

I return now to the two adjectives discussed earlier—"clear" and "precise"—behind the choice of the term "visual formalism."[14,16] Concerning clarity, the fact that a picture is worth a thousand words demands special caution. Not everything is beneficially depicted visually, but the basic topology-inspired graphics of statecharts seemed from the start to jibe well with the IAI avionics engineers; they quickly grasped the hierarchy and orthogonality, high- and low-level tran-

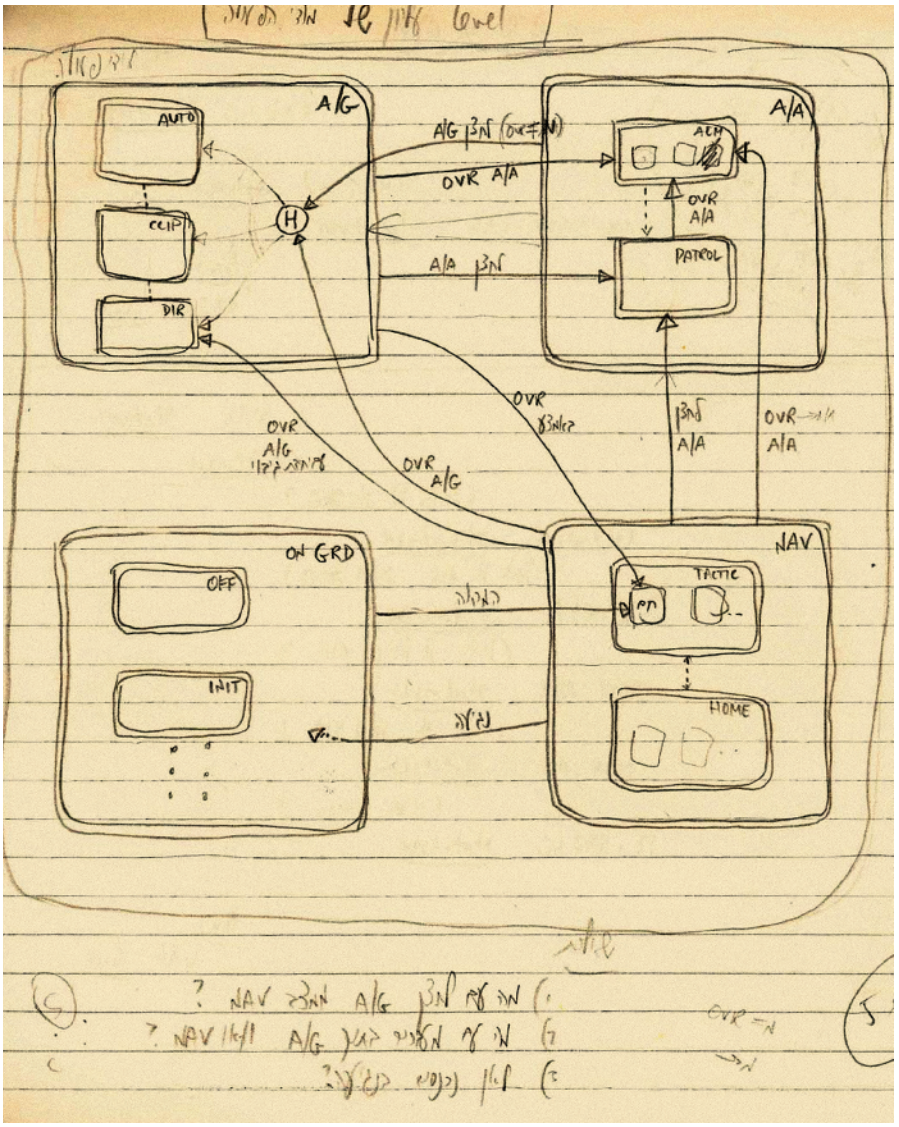

Figure 3: Page from the IAI notes (1983, events in Hebrew) showing a relatively "clean" draft of the top levels of behavior for the main flight modes of the Lavi avionics, including A/A (air-air), A/G (air-ground), NAV (automatic navigation), and ON GRD (on ground). Note early use of a history connector in the A/G mode.

sitions, default entries, and more.

Interestingly, the same quick comprehension applied to nonexperts outside the avionics group. I recall an anecdote from late 1983 in which in the midst of one session the blackboard showed a complicated statechart specifying the behavior of some intricate portion of the Lavi's avionics. A knock on the door brought in an Air Force pilot from the "customer" team who knew a lot about the aircraft being developed and its desired behavior but had never seen a state machine or a state diagram before, not to mention a statechart. I remember him staring at this intricate diagram (the statechart) on the blackboard, with its complicated mess of blobs inside other blobs, arrows splitting and merging, and asking, "What's that?" One of the engineers said, "That's the behavior of the so-and-so part of the system, and, by the way, these rounded rectangles are states, and the arrows are transitions between states." The pilot studied the blackboard for a couple of minutes, then said, "I think you have a mistake down here; this arrow should go over here and not over there." He was right.

For me, this little event indicated that we might be doing something right, that maybe what I was proposing was a good and useful way of specifying reactive behavior. If an outsider could come in, just like that, and grasp something that complicated without being exposed to the technical details of the language or the approach, then maybe we were on the right track (see Figure 3). Very encouraging.

So much for clarity. As for precision and formality, full executability was always central to the development of the language. I found it difficult to imagine the usefulness of a method that merely makes it possible to say things about behavior, give snippets of the dynamics or observations about what happens or what could happen, or provide some partially connected pieces of behavior. The whole idea was that if one builds a statechart-based specification everything must be rigorous enough to be run (executed) just like software written in a programming language. Executability was a basic, not-to-be-compromised, underlying concern during the process of designing the language. It might sound strange to a reader 26 years later, but in 1983 system-development tools did not execute models at all. Thus, turning doodles like those in the figure into a real language could be done only with great care.

Building a Tool

Once the basics of the language were established, it seemed natural to want a tool that could be used not only to prepare statecharts but also to execute them. So in April 1984, three colleagues (the brothers Ido and Hagi Lachover and Amir Pnueli) and I founded a company, Ad Cad, Ltd., later (1987) reorganizing it as I-Logix, Inc., with Ad Cad as its R&D branch. By 1986, we had built a tool for statecharts called Statemate.

In extensive discussions with the two most senior technical people associated with the company, Rivi Sherman and Michal Politi, along with Amir Pnueli, we were able to figure out during the Ad Cad period how to embed statecharts into a broader framework that was capable of capturing the structure and functionality of a large complex system. To this end, we proposed a diagrammatic language to structure a model that we called activity-charts, an enriched kind of hierarchical data-flow diagram whereby arrows represent the possible flow of information between the incident functions (activities). Each activity can be associated with a controlling statechart (or code) that would also be responsible for interfunction communication and cooperation.

Statemate also enabled one to specify the actual structure of the system itself, using module-charts that specify the components in the implementation of the system and their connections. In this way, the tool supported a three-way model-based development framework for systems consisting of structure, functionality, and behavior. The user could draw the statecharts and the model's other artifacts, link them together rigorously, check and analyze them, produce documents from them, and manage their configurations and versions. Most important, Statemate could fully execute them and generate from them, automatically, executable code in, say, Ada and C and later also in appropriate hardware description languages.

Even then, more than 20 years ago, Statemate could link the model to a GUI mockup of the system under development (or even to real hardware). Executability of the model could be done directly or by using the generated code and carried out in many ways with increasing sophistication. Verification wasn't in vogue in the 1980s, so analysis of the models was limited to various kinds of testing offered by Statemate in abundance. One could execute the model interactively (with the user playing the role of the system's environment), in batch mode (reading in external events from files) or in programmed mode. One could use breakpoints and random events to help set up and control a complex execution from which you could gather the results of interest. In principle, you could thus set up Statemate to "fly the aircraft" for you under programmed sets of circumstances, then come in the following day and find out what had happened. These capabilities, allowing us to "see" the model in operation, either via a GUI or following the statecharts as they were animated on the fly, were extremely useful to Statemate users. The tool was an eye-opener for software and systems engineers used to writing and debugging code in the usual way and was particularly beneficial for real-time and embedded systems.

Statemate is considered by some to be the first real-world tool to carry out true model executability and full code generation. The underlying ideas were the first serious proposal for model-driven system development. They might have been somewhat before their time but were deeply significant in bringing about the change in attitude that permeates modern-day software engineering, as exemplified by such efforts as the Unified Modeling Language. A decade after Statemate, we built the object-oriented Rhapsody tool at I-Logix (discussed later).

Woes of Publication

I wrote the first version of a paper describing statecharts in late 1983.[16] The process of trying to get it published was long and tedious but interesting in its own right. The details appear in the full version of the present article,[6] but I can say that the paper was rejected by several leading journals, including *Communications* and *IEEE Computer*. My files contain an interesting collection of referee comments and editor rejection letters, one of which asserted: "The basic problem [...] is that [...] the paper does not make a specific contribution in any area." It was only in July 1987 that the paper was finally published, in *Science of Computer Programming*.[16] The full version of the present article[6] also contains information (and anecdotes) about other publications on statecharts, including a paper I wrote with Pnueli defining reactive systems,[17] a *Communications* article on visual formalisms and higraphs,[14] an eight-author paper on Statemate,[13] the definitive paper on the Statemate semantics of statecharts,[13] and a Statemate book with Politi.[10]

Object-Oriented Statecharts

In the early 1990s, Eran Gery from I-Logix became interested in the work of James Rumbaugh and Grady Booch on the use of statecharts in an object-oriented framework. Gery did some gentle prodding to get me interested, with the ultimate result being a 1997 paper[11] in which we defined object-oriented statecharts and worked out the way we felt they should be linked up with objects and executed. In particular, we proposed two modes of communication between objects: direct synchronous invocation of methods and asynchronous queued events. The paper considered other issues, too, including creation and destruction of objects and multithreaded execution. The main structuring mechanism involved classes and objects, each of which could be associated with a statechart.

We also outlined a new tool for supporting all this, and I-Logix promptly built it under the name Rhapsody.[11] One important difference between the function-oriented Statemate and the object-oriented Rhapsody is that the semantics of statecharts in Statemate is synchronous and in Rhapsody is (by and large) asynchronous. Another subtle but significant difference is reflected in the fact that Statemate was set up to execute statecharts directly in an interpreter mode separate from the code generator. In contrast, the model execution in Rhapsody is carried out by running the code generated from the model. A third difference is our decision to make the action language of Rhapsody a subset of the target programming language; for example, the events, conditions, and actions specified along state transitions are fragments of, say, C++ or Java. In any event, the statechart language may be considered a level higher than classical programming languages in that the code generated from it is in, say, C++, Java, or C; we thus might say that statecharts are high above C level.

Several software vendors have since developed tools based on statecharts or its many variants, including RoseRT (which grew out of ObjecTime), StateRover, and Stateflow (the statechart tool embedded in Matlab).

The implementation and tool-building issue can also be viewed in a broader perspective. In the early 1980s, no system-development tool based on graphical languages was able to execute models or generate full running code. Such CASE tools essentially consisted of graphic editors, document generators, and configuration managers and were thus like programming environments without a compiler. In contrast, I have always felt that a tool for developing complex systems must have the ability to not only describe behavior but also to analyze and execute it in full. This philosophy underlies the notion of a visual formalism, which must come endowed with sufficiently well-defined semantics so as to enable tools to be built around it that can carry out dynamic analysis, full model execution, and the automatic generation of running code.

On Semantics

It is worth dwelling further on the issue of semantics, which is a prerequisite for

understanding the true meaning of any language, particularly executable ones. In a letter to me in 1984, Tony Hoare said that the statecharts language "badly needs a semantics." He was right. Being overly naïve at the time, I figured that writing a paper that explained the basics of the language's operation and then building a tool that executes statecharts and generates code from them would be enough. This approach took its cue from programming language research, where developers invent languages and then simply build compilers for them.

In retrospect, I didn't fully realize in those early years how different statecharts are from previous specification languages for real-time embedded systems. I knew, of course, that the language had to be executable, as well as easily understandable, even by people with no training in formal semantics. At the same time, as a tool-building team, we also had to demonstrate quickly to our sponsors, the first being IAI, that our efforts were economically viable. Due to the high level of abstraction of statecharts, we had to make decisions regarding rather deep semantic problems that apparently hadn't been adequately considered in the literature, at least not in the context of building a real-world tool intended for large, complex systems. Moreover, some of these issues were then being investigated independently by leading French researchers, including Gérard Berry, Nicholas Halbwachs, and Paul le Guernic, who coined the French phrase L'approche synchrone—the synchronous approach—for this kind of work.[1] Thus, when designing Statemate from 1984 to 1986, we did not do such a good job of deciding on the semantics.

We had to address a number of dilemmas regarding central semantic issues. One had to do with whether a step of the system should take zero time or more; another had to do with whether the effects of a step should be calculated and applied in a fixpoint-like manner in the same step or take effect only in the following step. The two issues are independent. The first concerns whether or not one adopts Berry's pure synchrony hypothesis,[1] whereby each step takes zero time. Clearly, these questions have many consequences in terms of how the language operates, whether events might interfere with chain reactions triggered by other events, how time itself is modeled, and how time interleaves with the system's discrete event dynamics.

At the time, we used the terms Semantics A and B to refer to the two main approaches we were considering. Both were synchronous in the sense of Benveniste et al.,[1] differing mainly in the second issue—regarding when the effects of a step take place. In Semantics A all events generated in the current step serve as inputs to the next step, whereas in Semantics B the system responds to all events generated internally in the current step until no further system response is possible. We called this chain of reactions a "super-step." The paper we published in 1987 was based on Semantics B,[15] but we later adopted semantics A for the Statemate tool itself.[10,13] Thus, Statemate statecharts constitute a synchronous language[1] and in that respect are similar to other, nonvisual languages in that family, including Esterel, Lustre (in commercial guise, known as Scade), and Signal.

We decided to implement Semantics A mainly because calculating the total effects of a step and carrying them out in the following step was easier to implement; we were also convinced that it was easier to understand for a typical systems engineer. Another consideration was related to the semantic level of compositionality; Semantics B strengthens the distinction between the system and its environment or between two parts of the system. If at some point in the development a system developer wants to consider part of the system to serve as an environment for the other part, the behaviors under Semantics B will be separated (as they should be), because chain reactions that go back and forth between the two halves are no longer all contained in a single super-step.

A number of other researchers had also begun looking into statechart semantics, often severely limiting the language (such as by completely dropping orthogonality) so the semantics are easier to define. Some of this work was motivated by the fact that our implemented semantics had not been published yet (we published in 1996[12]) and was not known outside the Statemate circle. This pre-object-oriented situation was summarized by Michael von der Beeck who tried to impose some order on the multitude of semantics of statecharts that were then being published. His resulting paper[22] claimed implicitly that statecharts are not well defined due to these many different semantics (it listed approximately 20). Interestingly, while [22] reported on the many variants of the language with their semantics, it did not report what should probably have been considered the language's "official" semantics, the one we defined and adopted in 1986 when building Statemate[13] but unfortunately also the only semantics not published at the time in the open public literature.

As to the semantic issues themselves, von der Beeck[22] discussed only the differences between variants of pre-object-oriented statecharts, but they are far less important than the differences between the non-object-oriented and the object-oriented versions of the language. The main semantic difference between Statemate and Rhapsody semantics is in synchronicity. In Statemate, the version of the statecharts language is based on functional decomposition and is a synchronous language, whereas the object-oriented-based Rhapsody version of statecharts is asynchronous. There are other substantial differences in modes of communication between objects; there are also issues arising from the presence of dynamic objects and their creation and destruction, inheritance, composition, and multithreading. The semantics of object-oriented statecharts was described in my 2004 paper with Hillel Kugler[8] (analogous to [12]) describing the differences between these two versions of the language.

Meanwhile, the Unified Modeling Language, or UML, which was standardized by the Object Management Group in 1997, featured many graphical languages, some of which are still not endowed with satisfactorily rigorous semantics. The heart of UML—its driving behavioral kernel—is the object-oriented version of statecharts. In the mid-1990s Eran Gery and I took part in helping the UML design team define the intended meaning of statecharts, resulting in UML statecharts being similar to those in [11] that we implemented in Rhapsody. For a manifesto about the subtle issues involved in defining the semantics of languages for reactive systems, see [7], with its whimsical subtitle "What's the Semantics of 'Semantics'?"

Biology, Hybrid Systems, Verification, Scenarios

Statecharts today are widely used in such application areas as aerospace, automotive, telecommunication, medical instrumentation, hardware design, and control systems. An interesting development also involves the language being used in such unconventional areas as modeling biological systems.[4,20,21]

Another important topic is hybrid systems. Statecharts can include probabilities, thus supporting probabilistic and stochastic behavior, but their underlying basis is discrete. It is very natural for a software or systems engineer to want to model systems with mixed discrete and continuous behavior, and it is not difficult to imagine using mathematics geared for continuous dynamics (such as differential equations) to model the activities within states in a statechart. An active community today is carrying out research on such systems.

Another topic involves exploiting the structuring of behavior in statecharts to aid verification of the modeled system. We all know how difficult program verification is, yet a number of techniques work well in many cases. While most common verification techniques do not exploit the hierarchical structure or modularity models often have, this structure can be used beneficially in verifying statecharts. Work has indeed been done on the verification of hierarchical state machines, though much more remains to be done.

Finally, I should mention some recent work my colleagues and I have carried out on a new approach to visual formalisms for complex systems. It involves a scenario-based specification method, rather than the state-based approach of statecharts. The idea is to concentrate on specifying the behavior between and among the objects (or tasks, functions, and components), not within them—inter-object rather than intra-object. The language we have proposed for this—Live Sequence Charts—was worked out jointly with Werner Damm.[2] The associated play-in and play-out programming techniques were developed later with my Ph.D. student Rami Marelly.[9] I published a paper last year describing a long-term vision on how this could be made much more general.[5]

Conclusion

If asked about the lessons to be learned from the statecharts story, I would definitely put tool support for executability and experience in real-world use at the top of the list. Too much computer science research on languages, methodologies, and semantics never finds its way into the real world, even in the long term, because these two issues do not get sufficient priority.

One of the most interesting aspects of this story is the fact that the work was not done in an academic tower, inventing something and trying to push it down the throats of real-world engineers. It was done by going into the lion's den, working with the people in industry. This is something I would not hesitate to recommend to young researchers; in order to affect the real world, one must go there and roll up one's sleeves. One secret is to try to get a handle on the thought processes of the engineers doing the real work and who will ultimately use these ideas and tools. In my case, they were the avionics engineers, and when I do biological modeling, they are biologists. If what you come up with does not jibe with how they think, they will not use it. It's that simple.

Looking back over the past 26 years, the main mistakes I made during the early years of statecharts concerned getting the message out to real-world software and systems engineers. This involved the confusing process of deciding on a clear semantics for the language and publicizing the chosen semantics promptly in the public literature, as well as not recognizing how important it was to quickly publish a book to acquaint engineers in industry with, and get them to use, a new language, method, and tool.

Nevertheless, despite the effort that went into developing the language (and later the tools to support it) I am convinced that almost anyone could have come up with statecharts, given the right background, exposure to the right kinds of problems, and right kinds of people.

Acknowledgment

Many people influenced this work, but my deepest gratitude goes to Jonah Lavi, Amir Pnueli, Eran Gery, Rivi Sherman, and Michal Politi. C

References

1. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91* (2003), 64–83.
2. Damm, W. and Harel, D. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design 19*, 1 (2001), 45–80.
3. Drusinsky, D. and Harel, D. On the power of bounded concurrency I: Finite automata. *J. ACM 41* (1994), 517–539.
4. Efroni, S., Harel, D., and Cohen, I.R. Towards rigorous comprehension of biological complexity: Modeling, execution, and visualization of thymic T cell maturation. *Genome Research 13* (2003), 2485–2497.
5. Harel, D. Can programming be liberated, period? *IEEE Computer 41*, 1 (Jan. 2008), 28–37.
6. Harel, D. Statecharts in the making: A personal account. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, CA, June 9-10). ACM Press, New York, 2007.
7. Harel, D. and Rumpe, B. Meaningful modeling: What's the semantics of 'semantics'? *IEEE Computer 37*, 10 (2004), 64–72.
8. Harel, D. and Kugler, H. The Rhapsody semantics of statecharts (or, on the executable core of the UML). In *Integrations of Software Specification Techniques for Applications in Engineering, LNCS, Vol. 3147*, H. Ehrig et al., Eds. Springer-Verlag, New York, 2004, 325–354.
9. Harel, D. and Marelly, R. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, New York, 2003.
10. Harel, D. and Politi, M. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, New York, 1998.
11. Harel, D. and Gery, E. Executable object modeling with statecharts. *IEEE Computer 30*, 7 (1997), 31–42.
12. Harel, D. and Naamad, A. The statemate semantics of statecharts. *ACM Transactions on Software Engineering Methods 5*, 4 (Oct. 1996), 293–333.
13. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtul-Trauring, A., and Trakhtenbrot, M. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering 16*, 4 (1990), 403–414.
14. Harel, D. On visual formalisms. *Commun. ACM 31*, 5 (May 1988), 514–530.
15. Harel, D., Pnueli, A., Schmidt, J., and Sherman, R. On the formal semantics of statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science* (Ithaca, NY, 1987), 54–64.
16. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.
17. Harel, D. and Pnueli, A. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, K.R. Apt, Ed. NATO ASI Series, Vol. F-13, Springer-Verlag, New York, 1985, 477–498.
18. Harel, D. And/Or programs: A new approach to structured programming. *ACM Transactions on Programming Languages and Systems 2*, 1 (Jan. 1980), 1–17.
19. Heninger, K.L., Kallander, J.W., Shore, J.E., and Parnas, D.L. *Software Requirements for the A-7E Aircraft, NRL Report 3876*. Washington, D.C., Nov. 1978.
20. Setty, Y., Cohen, I.R., Dor, Y., and Harel, D. Four-dimensional realistic modeling of pancreatic organogenesis. In *Proceedings of the National Academy of Science 105*, 51 (2008), 20374–20379.
21. Swerdlin, N., Cohen, I.R., and Harel, D. Toward an in-silico lymph node: A realistic approach to modeling dynamic behavior of lymphocytes. In *Proceedings of the IEEE, Special Issue on Computational System Biology 96*, 8 (2008), 1421–1443.
22. von der Beeck, M. A comparison of statecharts variants. In *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS, Vol. 863*. Springer-Verlag, New York, 1994, 128–148.

David Harel (dharel@weizmann.ac.il) is the William Sussman Professorial Chair in the Department of Computer Science and Applied Mathematics at The Weizmann Institute of Science, Rehovot, Israel.