Statecharts in the Making: A Personal Account

David Harel
The Weizmann Institute of Science
Rehovot, ISRAEL 76100
dharel@weizmann.ac.il

Abstract

This paper is a highly personal and subjective account of how the language of statecharts came into being. The main novelty of the language is in being a fully executable visual formalism intended for capturing the behavior of complex real-world systems, and an interesting aspect of its history is that it illustrates the advantages of theoreticians venturing out into the trenches of the real world, "dirtying their hands" and working closely with the system's engineers. The story is told in a way that puts statecharts into perspective and discusses the role of the language in the emergence of broader concepts, such as visual formalisms in general, reactive systems, model-driven development, model executability and code generation.

1. Introduction

The invitation to write a paper on statecharts for this conference on the history of programming languages produces mixed feelings of pleasant apprehension. Pleasant because being invited to write this paper means that statecharts are considered to be a programming language. They are executable, compilable and analyzable, just like programs in any “real” programming language, so that what we have here is not "merely" a specification language or a medium for requirements documentation. The apprehension stems from the fact that writing a historical paper about something you yourself were heavily involved in is hard; the result is bound to be very personal and idiosyncratic, and might sound presumptuous. In addition to accuracy, the paper must also try to be of interest to people other than its author and his friends...

The decision was to take the opportunity to put the language into a broader perspective and, in addition to telling its "story", to discuss some of the issues that arose around it. An implicit claim here is that whatever specific vices and virtues statecharts possess, their emergence served to identify and solidify a number of ideas that are of greater significance than one particular language.

DOI 10.1145/1238844.1238849
http://doi.acm.org/10.1145/1238844.1238849

Some of these ideas are the general notion of a **visual formalism**, the identification of the class of **reactive systems** and the arguments for its significance and special character, the notion of **model-based development**, of which the UML is one of the best-known products, the concept of **model executability** and evidence of its feasibility, whereby high-level behavioral models (especially graphical ones) can and should be executed just like conventional computer programs, and the related concept of full **code generation,** whereby these high-level models are translated — actually, compiled down — into running code in a conventional language. The claim is not that none of these concepts was ever contemplated before statecharts, but rather that they became identified and pinpointed as part and parcel of the work on statecharts, and were given convincing support and evidence as a result thereof.

2. Pre-1982

I am not a programming languages person. In fact, the reader might be surprised to learn that the only programming languages I know reasonably well are PL/I and Basic…. I also enjoyed APL quite a bit at the time. However, even in classical languages like Fortran, PASCAL or C, not to mention more modern languages like C++ and Java, I haven't really done enough programming to be considered any kind of expert. Actually, nothing really qualifies me as a programming language researcher or developer. Prior to statecharts I had published in programming language venues, such as *POPL*, the *ACM Symposium on Principles of Programming Languages*, but the papers were about principles and theory, not about languages…. They mostly had to do with the logics of programs, their expressive power and axiomatics, and their relevance to correctness and verification.

In 1977, while at MIT working on my PhD, I had the opportunity to take a summer job at a small company in Cambridge, MA, called Higher Order Software (HOS), owned and run by Margaret Hamilton and Saydean Zeldin. They had a method for specifying software that took the form of trees of functions — a sort of functional decomposition if you will — that had to adhere to a set of six well-formedness axioms [HZ76]. We had several interesting discussions, sometimes arguments, one of which had to do with verification. When asked how they recommend that one verify the correctness of a system described using their method, the answers usually related to validating the appropriateness of the syntax. When it came to true verification, i.e., making sure that the system does what you expect it to, what they were saying in a nutshell was, "Oh, that's not a problem at all in our method because we don’t allow programs that do not satisfy their

requirements." In other words, they were claiming to have "solved" the issue of verification by virtue of disallowing incorrect programs in the language. Of course, this only passes the buck: The burden of verifying correctness now lies with the syntax checker...

This attitude towards correctness probably had to do with the declarative nature of the HOS approach, whereas had they constructed their method as an executable language the verification issue could not have been side-tracked in this way and would have had to be squarely confronted. Still, the basic tree-like approach of the HOS method was quite appealing, and was almost visual in nature. As a result, I decided to see if the technical essence of this basic idea could be properly defined, hopefully resulting in a semantically sound, and executable, language for functions, based on an and/or functional decomposition. The "and" was intended as a common generalization of concurrency and sequentiality (you must do *all* these things) and the "or" a generalization of choice and branching (you must do *at least* one of these things). This was very similar to the then newly introduced notion of **alternation**, which had been added to classical models of computation in the theory community by Chandra, Kozen and Stockmeyer [CKS81] and about which I recall having gotten very excited at the time. Anyway, the result of this effort was a paper titled *And/Or Programs: A New Approach to Structured Programming*, presented in 1979 at an IEEE conference on reliable software (it later appeared in final form in ACM TOPLAS) [H79&80].

After the presentation at the conference, Dr. Jonah Lavi (Loeb) from the Israel Aircraft Industries (IAI) wanted to know if I was planning to return to Israel (at the time I was in the midst of a postdoctoral position — doing theory — at IBM's Yorktown Heights Research Center), and asked if I'd consider coming to work for the IAI. My response was to politely decline, since the intention was indeed to return within a year or so, but to academia to do research and teaching. This short conversation turned out to be crucial to the statechart story, as will become clear shortly.

3. December 1982 to mid 1983: The Avionics Motivation

We cut now to December 1982. At this point I had already been on the faculty of the Weizmann Institute of Science in Israel for two years. One day, the same Jonah Lavi called, asking if we could meet. In the meeting, he described briefly some severe problems that the engineers at IAI seemed to have, particularly mentioning the effort underway at IAI to build a home-made fighter aircraft, which was to be called the Lavi (no connection with Jonah's surname). The most difficult issues came up, he said, within the Lavi's avionics team. Jonah himself was a methodologist who did not work on a particular project; rather, he was responsible within the IAI for evaluating and bringing in software engineering tools and methods. He asked whether I would be willing to consult on a one-day-per-week basis, to see whether the problems they were having could be solved.

In retrospect, that visit turned out to be a real turning point for me. Moreover, luck played an important part too, since I feel that Jonah Lavi had no particular reason to prefer me to any other computer scientist, except for the coincidence of his happening to have heard that lecture on and/or programs a few years earlier. Whatever the case, I agreed to do the consulting, having for a long time harbored a never-consummated dream, or "weakness", for piloting, especially fighter aircraft.

And so, starting in December 1982, for several months, Thursday became my consulting day at the IAI. The first few weeks of this were devoted to sitting down with Jonah, in his office, trying to understand from him what the issues were. After a few such weeks, having learnt a lot from Jonah, whose broad insights into systems and software were extremely illuminating, I figured it was time to become exposed to the real project and the specific difficulties there. In fact, at that point I hadn't yet met the project's engineers at all. An opportunity for doing so arrived, curiously enough, as a result of a health problem that prevented Jonah from being in the office for a few weeks, so that our thinking and talking had to be put on hold. The consulting days of that period were spent, accompanied by Jonah's able assistant Yitzhak Shai, working with a select group of experts from the Lavi avionics team, among whom were Akiva Kaspi and Yigal Livne.

These turned out to be an extremely fruitful few weeks, during which I was able to get a more detailed first-hand idea about the problem and to take the first steps in proposing a solution. We shall get to that shortly, but first some words about avionics.

An avionics system is a great example of what Amir Pnueli and I later identified as a **reactive system** [HP85]. The aspect that dominates such a system is its reactivity; its event-driven, control-driven, event-response nature, often including strict time constraints, and often exhibiting a great deal of parallelism. A typical reactive system is not particularly data intensive or calculation-intensive. So what is/was the problem with such systems? In a nutshell, it is the need to provide a clear yet precise description of what the system does, or should do. Specifying its **behavior** is the real issue.

Here is how the problem showed up in the Lavi. The avionics team had many amazingly talented experts. There were radar experts, flight control experts, electronic warfare experts, hardware experts, communication experts, software experts, etc. When the radar people were asked to talk about radar, they would provide the exact algorithm the radar used in order to measure the distance to the target. The flight control people would talk about the synchronization between the controls in the cockpit and the flaps on the wings. The communications people would talk about the formatting of information traveling through the MuxBus communication line that runs lengthwise along the aircraft. And on and on. Each group had their own idiosyncratic way of thinking about the system, their own way of talking, their own diagrams, and their own emphases.

Then I would ask them what seemed like very simple specific questions, such as: "What happens when this button on the stick is pressed?" In way of responding, they would take out a two-volume document, written in structured natural language, each volume containing

something like 900 or 1000 pages. In answer to the question above, they would open volume B on page 389, at clause 19.11.6.10, where it says that if you press this button, such and such a thing occurs. At which point (having learned a few of the system's buzzwords during this day-a-week consulting period) I would say: "Yes, but is that true even if there is an infra-red missile locked on a ground target?" To which they would respond: "Oh no, in volume A, on page 895, clause 6.12.3.7, it says that in such a case this other thing happens." This to-and-fro Q&A session often continued for a while, and by question number 5 or 6 they were often not sure of the answer and would call the customer for a response (in this case some part of the Israeli Air Force team working with the IAI on the aircraft's desired specification). By the time we got to question number 8 or 9 even those people often did not have an answer! And, by the way, one of Jonah Lavi's motivations for getting an outside consultant was the bothersome fact that some of the IAI's subcontractors refused to work from these enormous documents, claiming that they couldn't understand them and were in any case not certain that they were consistent or complete.

In my naïve eyes, this looked like a bizarre situation, because it was obvious that someone, eventually, would make a decision about what happens when you press a certain button under a certain set of circumstances. However, that person might very well turn out to be a low-level programmer whose task it was to write some code for some procedure, and who inadvertently was making decisions that influenced crucial behavior on a much higher level. Coming, as I did, from a clean-slate background in terms of avionics (which is a polite way of saying that I knew nothing about the subject matter…), this was shocking. It seemed extraordinary that this talented and professional team *did* have answers to questions such as "What algorithm is used by the radar to measure the distance to a target?", but in many cases did *not* have the answers to questions that seemed more basic, such as "What happens when you press this button on the stick under all possible circumstances?".

In retrospect, the two only real advantages I had over the avionics people were these: (i) having had no prior expertise or knowledge about this kind of system, which enabled me to approach it with a completely blank state of mind and think of it any which way; and (ii) having come from a slightly more mathematically rigorous background, making it somewhat more difficult for them to convince me that a two-volume, 2000 page document, written in structured natural language, was a complete, comprehensive and consistent specification of the system's behavior.

In order to make this second point a little more responsibly, let us take a look at an example taken from the specification of a certain chemical plant. It involves a tiny slice of behavior that I searched for tediously in this document (which was about 700 pages long). I found this particular piece of behavior mentioned in three different places in the document. The first is from an early part, on security, and appeared around page 10 of the document:

Section 2.7.6: Security

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

Later on, in a section on temperatures, which was around page 150 of the document, it says:

Section 9.3.4: Temperatures

"If the system sends a signal hot and $T > 60^0$, then send a message to the operator."

The real gem was in the third quote, which occurred somewhere around page 650 of the document, towards the end, in a section devoted to summarizing some critical aspects of the system. There it says the following:

Summary of critical aspects

"When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when $T < 60^0$."

Despite being educated as a logician, I've never really been able to figure out whether the third of these is equivalent to, or implies, any of the previous two… But that, of course, is not the point. The point is that these excerpts were obviously written by three different people for three different reasons, and that such large documents get handed over to programmers, some more experienced than others, to write the code. It is almost certain that the person writing the code for this critical aspect of the chemical plant will produce something that will turn out to be problematic in the best case — catastrophic in the worst. In addition, keep in mind that these excerpts were found by an extensive search through the entire document to try find where this little piece of behavior was actually mentioned. Imagine our programmer having to do that repeatedly for whatever parts of the system he/she is responsible for, and then to make sense of it all.

The specification documents that the Lavi avionics group had produced at the Israel Aircraft Industries were no better; if anything, they were longer and more complex, and hence worse, which leads to the question of how such an engineering team should specify behavior of such a system in a intuitively clear and mathematically rigorous fashion. These two characteristics, clarity and rigor, will take on special importance as our story unfolds.

4. 1983: Statecharts Emerging

Working with the avionics experts every Thursday for several weeks was a true eye-opener. At the time there was no issue of inventing a new programming language. The goal was to try to find, or to invent for these experts, a means for simply saying what they seemed to have had in their minds anyway. Despite the fact that the simple "what happens" questions get increasingly more complicated to answer, it became very clear that these engineers knew a tremendous amount about the intended behavior of the system. They understood it, and they had answers to many of the questions about behavior. Other questions they hadn't had the opportunity to think about properly because the information wasn't well organized in their documents, or even, for that matter, in their minds. The goal was to

find a way to help take the information that was present collectively in their heads and put it on paper, so to speak, in a fashion that was both well organized and accurate.

Accordingly, the work progressed in the following way. A lot of time was spent getting them to talk; I kept asking questions, prodding them to state clearly how the aircraft behaves under certain sets of circumstances. Example: "What are the radar's main activities in air-ground mode when the automatic pilot is on?" They would talk and we would have discussions, trying to make some coherent sense of the information that piled up.

I became convinced from the start that the notion of a state and a transition to a new state was fundamental to their thinking about the system. (This insight was consistent with some of the influential work David Parnas had been doing for a few years on the A-7 avionics [HKSP78].) They 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 there is no radar locked on a ground target at the time". Of course, for anyone coming from computer science this is very familiar: what we really have here is a finite-state automaton, with its state transition table or state transition diagram. Still, it was pretty easy to see that just having one big state machine describing what is going on would be fruitless, and not only because of the number of states, which, of course, grows exponentially in the size of the system. Even more important seemed to be the pragmatic point of view, whereby a formalism in which you simply list all possible states and specify all the transitions between them is unstructured and non-intuitive; it has no means for modularity, hiding of information, clustering, and separation of concerns, and was not going to work for the kind of complex behavior in the avionics system. And if you tried to draw it visually you'd get spaghetti of the worst kind. It became obvious pretty quickly that it could be beneficial to come up with some kind of structured and hierarchical extension of the conventional state machine formalism.

So following an initial attempt at formalizing parts of the system using a sort of temporal logic-like notation (see Fig. 1)[1], I resorted to writing down the state-based behavior textually, in a kind of structured dialect made up on the fly that talked about states and their structure and the transitions between them. However, this dialect was hierarchical: inside a state there could be other states, and if you were in this state, and that event occurred, you would leave the current state and anything inside it and enter that other state, and so on. Fig. 2 shows an early example, from somewhere in early 1983, of one of these structured **state protocols**, or **statocols**, taken from my messy, scribbled IAI notebook.

As this was going on, things got increasingly complicated. The engineers would bring up additional pieces of the avionics behavior, and after figuring out how the new stuff related to the old, I would respond by extending the state-based structured description, often having to enrich the syntax in real time... When things got a little more complicated, I would doodle on the side of the page to explain visually what was meant; some of this is visible on the right-hand side of Fig. 2. I clearly recall the first time I used visual encapsulation to illustrate to them the state hierarchy, and an arrow emanating from the higher level to show a compound "leave-any-state-inside" transition; see the doodling in Fig. 2 and the more orderly attempts in Fig. 3. And I also remember the first time I used side-by-side adjacency for orthogonal (concurrent) state components, denoted — after playing with two or three possible line styles — by a dashed line; see Fig. 4. However, it is important to realize that, at the time, these informal diagrams were drawn in order to explain what the nongraphical state protocols meant. The text was still the real thing and the diagrams were merely an aid.

After a few of these meetings with the avionics experts, it suddenly dawned on me that everyone around the table seemed to understand the back-of-napkin style diagrams a lot better and related to them far more naturally. The pictures were simply doing a much better job of setting down on paper the system's behavior, as understood by the engineers, and we found ourselves discussing the avionics and arguing about them over the diagrams, not the statocols. Still, the mathematician in me argued thus: "How could these doodled diagrams be better than the real mathematical-looking artifact?" (Observe Fig. 2 again, to see the two options side by side.) So it really took a leap of 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?" And so, over a period of a few weeks the scales tipped in favor of the diagrams. 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; see Figs. 5–9.

Of course, this had to be done in a responsible way, making sure that the emerging pictures were not just pictures; that they were not just doodling. They had to be rigorous, based on precise mathematical meaning. You couldn't just throw in features because they looked good and because the avionics team seemed to understand them. Unless the exact 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 was as of 1983 the only unused combination of "state" or "flow" with "chart" or "diagram".

5. On the Language Itself

Besides a host of other constructs, the two main ideas in statecharts are **hierarchy** and **orthogonality**, and these can be intermixed on all levels: You start out with classical finite-state machines (FSMs) and their state transition diagrams, and you extend them by a semantically meaningful hierarchical substating mechanism and by a notion of orthogonal simultaneity. Both of these are reflected in the graphics themselves, the hierarchy by encapsulation and the orthogonality by adjacent portions separated by a dashed line. Orthogonal components can

[1] Because of the special nature and size of some of the figures, I have placed them all at the end of the text, before the references.

cooperate and know about each other by several means, including direct sensing of the state status in another component or by actions. The cooperation mechanism — within a single statechart I should add — has a broadcasting flavor.

Transitions become far more elaborate and rich than in conventional FSMs. They can start or stop at any level of the hierarchy, can cross levels thereof, and in general can be hyperedges, since both sources and targets of transitions can contain sets of states. In fact, at any given point in time a statechart will be in a vector, or combination, of states, whose length is not fixed. Exiting and entering orthogonal components on the various levels of the hierarchy continuously changes the size of the state vector. Default states generalize start states, and they too can be level-crossing and of hyperedge nature. And the language has history connectors, conditions, selection connectors, and more. A transition can be labeled with an event and optionally also with a parenthesized condition, as well as with Mealy-like outputs, or actions. (Actions can also occur within states, in the Moore style.)

The fact that the technical part of the statecharts story started out with and/or programs is in fact very relevant. Encapsulated substates represent OR (actually this is XOR; exclusive or), and orthogonality is AND. Thus, a minimalist might view statecharts as a state-based language whose underlying structuring mechanism is simply that of classical alternation [CKS81]. Figs. 9 and 10 exemplify this connection by showing a state hierarchy for a part of the Lavi avionics statecharts and then the and/or tree I used to explain to the engineers in a different way what was actually going on.

In order to make this paper a little more technically informative, I will now carry out some self-plagiarism, stealing and then modifying some of the figures and explanations of the basic features of the language from the original statechart paper [H84&87]. However, the reader should not take the rest of this section as a tutorial on the language, or as a language manual. It is extremely informal, and extremely partial. I am also setting it in smaller font, and slightly indented, so that you can skip it completely if you want. For more complete accounts, please refer to [H84&87, HN89&96, HP91, HK04].

In way of introducing the state hierarchy, consider Fig. 11(i). It shows a very simple four-state chart. Notice, however, that event β takes the system to state B from either A or C, and also that δ takes the system to D from either of these. Thus, we can cluster A and C into a new **superstate**, E, and replace the two β transitions and the two δ ones by a single transition for each, as in Fig. 11(ii). The semantics of E is then the XOR of A and C; i.e., to be in state E one must be either in A or in C, but not in both. Thus E is really an abstraction of A and C, and its outgoing β and δ arrows capture two common properties of A and C; namely, that β leads from them to B and δ to D. The decision to have transitions that leave a superstate denote transitions leaving all substates turns out to be highly important, and is one of the main ways statecharts economize in the number of arrows.

Fig. 11 might also be approached from a different angle: first we might have decided upon the simple situation of Fig. 11(iii) and then state E could have been refined to consist of A and C, yielding Fig. 11(ii). Having decided to make this refinement, however, the transitions entering E in Fig. 11(iii), namely, α, δ and γ, become underspecified, as they do not say which of A or C is to be entered. This can be remedied in a number of ways. One is to simply extend them to point directly to A or C, as with the α-arrow entering A directly in Fig. 11(ii). Another is to use multi-level default entrances, as we now explain.

Fig. 11(i) has a start arrow pointing to state A. In finite automata this means simply that the automaton starts in state A. In statecharts this notion is generalized to that of a **default state**, which, in the context of Fig. 11 is taken to mean that as far as the 'outside' world is concerned A is the default state among A, B, C and D: if we are asked to enter one of these states but are not told which one to enter, the system is to enter A. In Fig. 11(i) this is captured in the obvious way, but in Fig. 11(ii) it is more subtle. The default arrow starts on the (topological) outside of the superstate E and enters A directly. This does not contradict the other default arrow in Fig. 11(ii), which is (topologically) wholly contained inside E and which leads to C. Its semantics is that if we somehow already entered E, but inside E we are not told where to go, the inner default is C, not A. This takes care of the two otherwise-underspecified transitions entering (and stopping at the borderline of) state E, those labeled δ and γ, emanating from B and D, respectively, and which indeed by Fig. 11(i) are to end up in C, not in A. Thus, Figs. 11(i) and 11(ii) are totally equivalent in their information, whereas Fig. 11(iii) contains less information and is thus an abstraction.

Besides the default entrance, there are other special ways to enter states, including **conditional** entries, specified by a circled C, and **history** entrances, specified by a circled H. The latter is particularly interesting, as it allows one to specify entrance to the substate most recently visited within the state in question, and thus caters for a (theoretically very limited, but in practice useful) kind of memory. In both of these, the connector's location within the state hierarchy has semantic significance.

So much for the hierarchical XOR decomposition of states. The second notion is the AND decomposition, capturing the property that, being in a state, the system must be in *all* of its components. The notation used in statecharts is the physical partitioning of a state box (called **blob** in [H88]) into components, using dashed lines.

Figure 12(i) shows a state Y consisting of AND components A and D, with the property that being in Y entails being in some combination of B or C with E, F or G. We say that Y is the **orthogonal product** of A and D. The components A and D are no different conceptually from any other superstates; they can have defaults, substates, internal transitions, etc. Entering Y from the outside, in the absence of any additional information (like the τ entrance on the right hand side of Fig. 12(ii)), is actually entering the combination (B,F), as a result of the default arrows that lead to B and F. If event α then occurs, it transfers B to C and F to G simultaneously, resulting in the new combined state (C,G). This illustrates a certain kind of **synchronicity**: a single event causing two simultaneous happenings. If, on the other hand, μ occurs at (B, F) it affects the D component only, resulting in (B,E). This, in turn, illustrates a certain kind of **independence**, since the transition is the same

whether, in component A, the system happens to be in B or in C. Both behaviors are part of the orthogonality of A and D, which is the term used in statecharts to describe the AND decomposition. Later we shall discuss the difference between orthogonality and concurrency, or parallelism.

Fig. 12(ii) shows the same orthogonal state Y, but with some transitions to and from it. As mentioned, the τ entrance on the right enters (B,F), but the λ entrance, on the other hand, overrides D's default by entering G directly. But since one cannot be in G alone, the system actually enters the combination (B,G), using A's default. The split ξ entrance on the top of Fig. 12(ii) illustrates an explicit indication as to which combination is to be entered, (B, E) in this case. The γ-event enters a history connector in the area of D, and hence causes entrance to the combination of B (A's default) with the most recently visited state in D. As to the exits in Fig. 12(ii), the ω-event causes an exit from C combined with any of the three substates of D ─ again a sort of independence property. Had the ω arrow been a merging hyper-edge (like the ξ one, but with the direction reversed) with C and, say, G, as its sources, it would have been a direct specification of an exit from (C,G) only. The most general kind of exit is the one labeled χ on the left hand side of the figure, which causes control to leave A×D unconditionally.

Fig. 13(i) is the conventional AND-free equivalent of Fig. 12(i), and has six states because the components of Fig. 12(i) contain two and three. Clearly, if these had a thousand states each, the resulting "flat" product version would have a million states. This, of course, is the root of the exponential blow-up in the number of states, which occurs when classical finite state automata or state diagrams are used, and orthogonality is our way of avoiding it. (This last comment assumes, of course, that we are specifying using the state-based language alone, not embedded in objects or tasks, etc.) Note that the "in G" condition attached to the β-transition from C in Fig. 12(i) has the obvious consequence in Fig. 13(i): the absence of a β-transition from (C,E). Fig 13(ii) adds to this the ω and χ exiting transitions of Fig. 12(ii), which now show up as rather messy sets of three and six transitions, respectively.

Fig. 14 illustrates the broadcast nature of inter-statechart communication. If after entering the default (B,F,J) event φ occurs, the statechart moves to (C,G,I), since the φ in component H triggered the event α, which causes the simultaneous moves from B to C in component A and from F to G in D. Now, if at the next stage a ψ occurs, I moves back to J, triggering β, which causes a move from C to B, triggering γ, which in turn causes a move from G to F. Thus, ideally in zero time (see Section 10), the statechart goes in this second step from (C,G,I) back to (B,F,J).

As mentioned above, the language has several additional features, though the notions of hierarchy and orthogonality are perhaps its two most significant ones. Besides language features, there are also several interesting semantic issues that arise, such as how to deal with nondeterminism, which hasn't been illustrated here at all, and synchronicity. References [HN89&96, HP91, HK04] have lots of information on these, and Sections 6 and 10 of this paper discuss some of them too.

So much for the basics of the language.

6. Comments on the Underlying Philosophy

When it comes to visuality, encapsulation and side-by-side adjacency are topological notions, just like edge connectivity, and are therefore worthy companions to edges in hierarchical extensions of graphs. Indeed, I believe that topology should be used first when designing a graphical language and only then one should move on to geometry. Topological features are a lot more fundamental than geometric ones, in that topology is a more basic branch of mathematics than geometry in terms of symmetries and mappings. One thing being inside another is more basic than it being smaller or larger than the other, or than one being a rectangle and the other a circle. Being connected to something is more basic than being green or yellow or being drawn with a thick line or with a thin line. I think the brain understands topological features given visually much better than it grasps geometrical ones. The mind can see easily and immediately whether things are connected or not, whether one thing encompasses another, or intersects it, etc. See the discussion on **higraphs** [H88] in Section 8.

Why this emphasis on topology, you may ask? Well, I've always had a (positive) weakness for this beautiful branch of mathematics. I love the idea of an "elastic geometry", if one is allowed a rather crude definition of it; the fact that two things are the same if the one can be stretched and squeezed to become the other. I remember being awed by Brouwer's fixed-point theorem, for example, and the Four-Color problem (in 1976 becoming the Four-Color Theorem). In fact, I started my MSc work in algebraic topology before moving over to theoretical computer science. This early love definitely had an influence on the choices made in designing statecharts.

Statecharts are not exclusively visual/diagrammatic. Their non-visual parts include, for example, the events that cause transitions, the conditions that guard against taking transitions and actions that are to be carried out when a transition is taken. For these, as mentioned earlier, statecharts borrow from both the Moore and the Mealy variants of state machines (see [HU79], in allowing actions on transitions between states or on entrances to or exits from states, as well as conditions that are to hold throughout the time the system is in a state. Which language should be used for these nongraphical elements is an issue we will discuss later.

Of course, the hierarchy and orthogonality constructs are but abbreviations and in principle can be eliminated: Encapsulation can be done away with simply by flattening the hierarchy and writing everything out explicitly on the low level, and orthogonality (as Figs. 12 and 13 show) can be done away with by taking the Cartesian product of the components of the orthogonal parts of the system. This means that these features do not strictly add expressive power to FSMs, so that their value must be assessed by "softer" criteria, such as naturalness and convenience, and also by the size of the description: Orthogonality provides a means for achieving an exponential improvement in succinctness, in both upper- and lower-bound senses [DH88, DH94].

A few words are in line here regarding the essence of orthogonality. Orthogonal state-components in statecharts are *not* the same as concurrent or parallel components of

the system being specified. The intention in having orthogonality in a statechart is not necessarily to represent the different *parts* of the system, but simply to help structure its state space, to help arrange the behavior in portions that are conceptually separate, or independent, or orthogonal. The word 'conceptually' is emphasized here because what counts is whatever is in the mind of the "statifier" — the person carrying out the statechart specification.

This motivation has many ramifications. Some people have complained about the broadcast communication mechanism of statecharts because it's quite obvious that you do not always want to broadcast things to an entire system. One response to this is that we are talking about the mechanism for communication between the orthogonal parts of the statechart, between its "chunks" of state-space, if you will, not between the components — physical or software components — of the actual system. The broadcasting is one of the means for sensing in one part of the state space what is going on in another part. It does not necessarily reflect an actual communication in the real implementation of the system. So, for example, if you want to say on a specification level that the system will only move from state *A* to state *B* if the radar is locked on a target, then that is exactly what you'll say, without having to worry about how state *A* will get to know what the radar is doing. This is true whether or not the other state component actually represents a real thing, such as the radar, or whether it is a non-tangible state chunk, such as whether the aircraft is in air-air mode or in air-ground mode. On this kind of level of abstraction you often really want to be able to sense information about one part of the specification in another, without having to constantly deal with implementation details.

A related response to the broadcasting issue is that no one is encouraged to specify a single statechart for an entire system.[2] Instead, as discussed in Sections 7 and 9, one is expected to have specified some breakup of the system itself, into functions, tasks, objects, etc., and to have a statechart (or code) for each of these. In this way, the real concurrency, the real separate pieces of the system, occur on a level higher than the statecharts, which in turn are used to specify the behavior of each of these pieces. If within a statechart the behavior is sufficiently complex to warrant orthogonal components, then so be it. In any case, the broadcast mechanism is intended to take effect only within a single statechart, and has nothing to do with the real communication mechanism used for the system itself.

By the way, some of the people who have built tools to support statecharts have chosen to leave orthogonality out of the language altogether, claiming that statecharts don't need concurrency since concurrency is present anyway between the objects of the system... Notable among these is the ObjectTime tool, which later evolved into RoseRT. My own opinion is that orthogonality is probably the most significant and powerful feature of the language, but also the most complex aspect of statecharts to deal with. So, the decision to leave it out is often made simply to render the task of building a tool that much easier…

Let us return briefly to the two key adjectives used earlier, namely "clear" and "precise", which underlie the choice of the term **visual formalism** [H84&87,H88]. Concerning clarity, the aphorism that a picture is worth a thousand words is something many people would agree with, but it requires caution. Not everything can be visualized; not everything can be depicted visually in a way that is clear and fitting for the brain. (This is related to the discussion above about topology versus geometry.) For some mysterious reason, the basic graphics of statecharts seemed from the very start to vibe well with the avionics engineers at the IAI. They were very fast in grasping the hierarchy and the orthogonality, the high- and low-level transitions and default entries, and so forth.

Interestingly, the same seemed to apply to people from outside our small group. I recall an anecdote from somewhere in late 1983, in which in the midst of one of the sessions at the IAI the blackboard contained a rather complicated statechart that specified the intricate behavior of some portion of the Lavi avionics system. I don't quite remember now, but it was considerably more complicated than the statecharts in Figs. 7–9. There was a knock on the door and in came one of the air force pilots from the headquarters of the project. He was a member of the "customer" requirements team, so he knew all about the intended aircraft (and eventually he would probably be able to fly one pretty easily too…), was smart and intelligent, but he had never seen a state machine or a state diagram before, not to mention a statechart. He stared for a moment at this picture on the blackboard, with its complicated mess of blobs, blobs inside other blobs, colored arrows splitting and merging, etc., and asked "What's that?" One of the members of the team said "Oh, 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". And that was all that was said. The pilot stood there studying the blackboard for a minute or two, and then said, "I think you have a mistake down here, this arrow should go over here and not over there"; and he was right.

For me, this little event was significant, as it really seemed to indicate that perhaps what was happening was "right", that maybe this was a good and useful way of doing things. If an outsider could come in, just like that, and be able to grasp something that was pretty complicated but without being exposed to the technical details of the language or the approach, then maybe we are on the right track. Very encouraging.

So much for clarity and visuality. As to precision and formality, later sections discuss semantics and supporting tools in some detail, but for now it suffices to say that one crucial aspect that was central to the development of the language from day one was **executability**. Being able to actually execute the specification of the Lavi's behavior was paramount in my mind, regardless of the form this specification ended up taking. I found it hard to imagine the usefulness of a method for capturing behavior that makes it possible merely to say some things about behavior, to give snippets of the dynamics, observations about what happens or what could happen, or to provide

[2] This is said despite the fact that in the basic paper on statecharts [H84&87], to be discussed later, I used a single statechart to describe an entire system, the Citizen digital-watch. That was done mainly for presentational reasons.

some disconnected or partially connected pieces of behavior. The whole idea was that if you build a statechart, or a collection of statecharts, everything has to be rigorous enough to be run, to be executable, just like software written in any old (or new…) programming language. Whether the execution is carried out in an interpreter mode or in a compiler mode is a separate issue, one to which we'll return later on. The main thing is that executability was a basic, not-to-be-compromised, underlying concern during the process of designing the language.

This might sound strange to the reader from his/her present vantage point, but back in 1983 system-development tools did not execute models at all, something else we shall return to later. Thus, turning the doodling into a language and then adding features to it had to be done with great care. You had to have a full conception of what each syntactically allowed combination of features in the language means, in terms of how it is to be executed under any set of circumstances.

7. 1984–1986: Building a Tool

Once the basics of the language were there, it seemed natural to want to build a tool that would have the ability not only to draw, or prepare, statecharts but also to execute them. Besides having to deal with the operational semantics of graphical constructs, such a tool would have to deal with the added complication of statecharts over and above classical finite-state machines: A typical snapshot of a statechart in operation contains not just a single state, but rather a vector, or an array, of states, depending on which orthogonal components the chart is in at the moment. Thus, this vector is flexible, given the basic maxim of the language, which is that states can be structured with a mixture of hierarchy and orthogonality and that transitions can go between levels. The very length of the vector of states changes as behavior progresses.

In a discussion with Amir Pnueli in late 1983, we decided to take on a joint PhD student and build a tool to support statecharts and their execution. Amir was, and still is, a dear colleague at the Weizmann Institute and had also been my MSc thesis supervisor 8-9 years earlier. Then, at some point, a friend of ours said something like, "Oh fine, you guys will build your tool in your academic setting, and you'll probably write some nice academic papers about it." And then he added, "You see, if this statechart stuff is just another idea, then whatever you do will not make much difference anyway, but if it has the potential of becoming useful in the real world then someone else is going to build a *commercial* tool around it; they will be the ones who get the credit, they will make the money and the impact, and you guys will be left behind". This caused us to rethink our options, a process that resulted in the founding of a company in Israel in April 1984, by the name of AdCad, Ltd. The two main founders were the brothers Ido and Hagi Lachover, and Amir Pnueli and I joined in too. The other three had previously owned a software company involved in totally different kinds of systems, but in doing so had acquired the needed industrial experience. The company was re-formed in 1987 as a USA entity, called I-Logix, Inc., and AdCad became its R&D branch, renamed as I-Logix Israel, Ltd.[3]

By 1986 we had built a tool for statecharts called **Statemate**. At the heart of a Statemate model was a functional decomposition controlled by statecharts. The user could draw the statecharts and the model's other artifacts, could check and analyze them, could produce documents from them, and could manage their configurations and versions. However, most importantly, Statemate could fully execute them. It could also generate from them, automatically, executable code; first in Ada and later also in C.

Among the other central figures during that period were Rivi Sherman and Michal Politi. In fact, it was in extensive discussions with Rivi, Michal and Amir that we were able to figure out how to embed statecharts into the broader framework that would capture the structure and functionality of a large complex system. To this end, we came up with the diagrammatic language that was used in Statemate for the hierarchical functional structuring of the model, which we called **activity-charts**. An activity-chart is an enriched kind of hierarchical data-flow diagram, where the semantics of arrows is the possible flow of information between the incident functions (which are called activities). Each activity could be associated with a controlling statechart (or with code), which would also be responsible for inter-function communication and cooperation. Statemate also enabled you to specify the actual structure of the system, using **module-charts**, which specify the real 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: structure, functionality and behavior.

Statemate is considered by many to be the first real-world tool to carry out true model executability and full code generation. I think it is not a great exaggeration to claim that the ideas underlying Statemate were really the first serious proposal for **model-driven system development.** These ideas were perhaps somewhat before their time, but were of significance in bringing about the eventual change in attitude that I think permeates modern-day software engineering. The recent UML effort and its standardization by the OMG (see Section 10) can be viewed a subsequent important step in steering software and systems engineering towards model-driven development.

Setting up the links between the statecharts and the activity-charts turned out to be very challenging, requiring among other things that we enrich the events, conditions and actions in the statecharts so that they could relate to the starting and stopping of activities, the use of variables and data types, time-related notions, and much more. After working all this out and completing the first version of Statemate in early 1986, we came across the independent

[3] I-Logix survived as a private stand-alone company for 22 long years, amid the many dips in high-tech. In recent years I maintained a very inactive and low-profile connection with the company, until it was acquired by Telelogic in March 2006. As of the time of writing I have no connection with either.

work of Ward and Mellor [WM85] and Hatley and Pirbhai [HP87], who also linked a functional decomposition with a state-based language (theirs was essentially conventional FSMs). It was very satisfying to see that many of the decisions about linking up the two were common to all three approaches. Several years later, Michal Politi and I sat down to write up a detailed report about the entire Statemate language set, which appeared as a lengthy technical report from I-Logix [HP91]. It took several years more for us to turn this into a fully fledged book [HP96].

Statemate had the ability to link the model to a GUI mockup of the system under development (or even to the real system hardware). Executability of the model could be done directly or by using the generated code, and could be carried out in many ways with increasing sophistication. You 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. Just as one example, you 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 Statemate up to "fly the aircraft" for you, and then come in the following day and find out what had happened. See [H92] for a more detailed discussion of model execution possibilities.

During the two years of the development of Statemate, Jonah Lavi from the IAI and his team were also very instrumental. They served as a highly useful beta site for the tool and also participated in making some of the decisions around its development. Jonah's ideas were particularly influential in the decision to have module-charts be part of Statemate.

Over the years, I-Logix built a number of additional tools, notable among which was a version of Statemate for hardware design, in which the statecharts were translated into a high-level hardware-description language. Much later, in the mid-1990's, we built the **Rhapsody** tool, based on **object-oriented statecharts**, about which we will have more to say in Section 9.

In the early years of I-Logix, I tried hard — but failed — to convince the company's management to produce a cheap (or free) version of Statemate for use by students. My feeling was that students of programming and software engineering should have at their disposal a simple tool for drawing and executing statecharts, connected to a GUI, so that they could build running applications easily using visual formalisms. This could have possibly expedited the acceptance of statecharts in industry. Instead, since Statemate was the only serious statechart tool around but was so very expensive, many small companies, university teachers and students simply couldn't afford it. Things have changed, however, and companies building such tools, including I-Logix, typically have special educational deals and/or simplified versions that can be downloaded free from the internet.

8. The Woes of Publication

In November 1983, I wrote an internal document at the IAI (in Hebrew), titled "Foundations of the State-Based Approach to The Description of System Operation (see Figs. 15-16), which contained an initial account of statecharts. At the time, my take was that this was but a nice visual way to describe states and transitions of more complex behavior than could be done conveniently with finite-state diagrams. I felt that the consulting job at IAI had indeed been successful, resulting, as it were, in something of use to the engineers of the Lavi avionics project. I had given no thought to whether this was indeed particularly new or novel, believing that anyone seriously working with state machines for real-world systems was probably doing something very similar. It seemed too natural to be new. After all, hierarchy, modularity and separation of concerns were engrained in nearly everything people were writing about and developing for the engineering of large systems.

Then one day, in a routine conversation with Amir Pnueli (this preceded the conversation reported above that resulted in our co-founding AdCad/I-Logix), he asked me, out of curiosity, what exactly I was doing at the Aircraft Industries on Thursdays. So I told him a little about the avionics project and the problem of specifying behavior, and then added something about proposing what seemed to be a rather natural extension of finite-state diagrams. He said that it sounded interesting and asked to see what the diagrams looked like. So I went to the blackboard (actually, a whiteboard; see the photo in Fig.17, taken a few months later) and spent some time showing him statecharts. He said something to the effect that he thought this was nice and interesting, to which I said, "Maybe, but I'm certain that this is how everyone works". He responded by saying that he didn't think so at all and proceeded to elaborate some more. Now, although we were both theoreticians, he had many more years of experience in industry (e.g., as part of the company he was involved in with the Lachover brothers), and what he said seemed convincing. After that meeting it made sense to try to tell the story to a broader audience, so I decided to write a "real" paper and try to get it published in the computer science literature.

The first handwritten version was completed in mid-December of 1983 (see Fig. 18). In this early version the word **statification** was used to denote the process of preparing a statechart description of a system. The paper was typed up, then revised somewhat (including the title) and was distributed as Technical Report CS84-05 of our Department at the Weizmann Institute in February of 1984 [H84&87]; see Fig. 19.

The process leading to the eventual publication of this paper is interesting in its own right. For almost two years, from early 1984 until late 1985, I repeatedly submitted it to what seemed to be the most appropriate widely read venues for such a topic. These were, in order, *Communications of the ACM*, *IEEE Computer* and *IEEE Software*. The paper was rejected from all three of these journals. In fact, from *IEEE Computer* it was rejected twice — once when submitted to a special issue on visual languages and once when submitted as a regular paper. My files contain quite an interesting collection of referee reports and editors' rejection letters. Here are some of the comments therein:

> *"I find the concept of statecharts to be quite interesting, but unfortunately only to a small segment of our readership. I find the information presented to be somewhat innovative, but not wholly new. I feel that the use of the digital watch*

example to be useful, but somewhat simple in light of what our readership would be looking for."

"The basic problem […] is that […] the paper does not make a specific contribution in any area."

"A research contribution must contain 'new, novel, basic results'. A reviewer must certify its 'originality, significance, and accuracy'. It must contain 'all technical information required to convince other researchers in the area that the results are valid, verifiable and reproducible'. I believe that you have not satisfied these requirements."

"I think your contribution is similar to earlier contributions."

"The paper is excellent in technical content; however, it is too long and the topic is good only for a very narrow audience."

"I doubt if anyone is going to print something this long."

Indeed, the paper was quite long; it contained almost 50 figures. The main running example was my Citizen Quartz Multi-Alarm digital wristwatch (see Fig. 20), which was claimed in the rejection material by some to be too simple an example for illustrating the concepts and by others to be far too detailed for a scientific article… Some claimed that since the paper was about the much studied finite-state machine formalism it could not contain anything new or interesting…

One must understand that in the mid-1980s there was only scant support for visual languages. Visual programming in the classical sense had not really succeeded; it was very hard to find ways to visualize nontrivial algorithmic systems (as opposed to visualizing the dynamic running of certain algorithms on a particular data structure), and the only visual languages that seemed to be successful in system design were those intended to specify structure rather than behavior. Flowcharts, of course, were a failure in that most people used them to help explain the real thing, which was the computer program. There was precious little real use of flowcharts as a language that people programmed in and then actually executed. In terms of languages for structure, there were structure diagrams and structure charts, hierarchical tree-like diagrams, and so on. The issue of a visual language with precise semantics for specifying behavior was not adequately addressed at all. Petri nets were an exception [R85], but except for some telecommunication applications they did not seem to have caught on widely in the real world. My feeling was that this had mainly to do with the lack of adequate support in Petri nets for hierarchical specification of behavior.

The state of the art on diagrammatic languages at the time can be gleaned from the book by Martin and McClure titled *Diagramming Techniques for Analysts and Programmers* [MM85]. This book discussed many visual techniques, but little attention was given to the need for solid semantics and/or executability. Curiously, this book could have helped convince people that visual languages should *not* be taken seriously as means to actually program a system the way a standard programming language can.

Coming back to the statecharts paper, the inability to get it published was extremely frustrating. Interestingly, during the two years of repeated rejections, new printings of the 1984 technical report had to be prepared, to address the multitude of requests for reprints. This was before the era of the internet and before papers were sent around electronically. So here was a paper that no one wanted to publish but that so many seemed to want to read... I revised the paper twice during that period, and the title changed again, to the final "Statecharts: A Visual Formalism for Complex Systems". Eventually, two and half years later, in July 1987, the paper was published in the theoretical journal *Science of Computer Programming* [H84&87]. That happened as a result of Amir Pnueli, who was one of its editors, seeing the difficulties I was having and soliciting the paper for the journal.[4]

In the revisions of the paper carried out between 1984 and 1987, some small sections and discussions that appeared in earlier versions were removed. One topic that appeared prominently in the original versions and was later toned down, appearing only in a very minimalistic way in the final paper, was the idea of having states contain state protocols, or **statocols**. These were to include information about the behavior that was not present in the charts themselves. The modern term for this kind of information behavior is the **action language**, i.e., the medium in which you write your events, conditions, actions, etc. The question of whether the action language should be part of the specification language itself or should be taken to be a subset of a conventional programming language is the subject of a rather heated debate that we will return to later.

A few additional papers on statecharts were written between 1984 and 1987. One was the paper written jointly with Pnueli on **reactive systems** [HP85]. It was born during a plane trip that we took together, flying to or from some conference. We were discussing the special nature of the kinds of systems for which languages like statecharts seemed particularly appropriate. At some point I complained about the lack of a special term to denote them, to which he responded by saying he thought such systems should be termed reactive. "Bingo", I said, "we have a new term"! Interestingly, this paper (which also contained a few sections describing statecharts) was written about two years after the original statecharts paper, but was published (in a NATO conference proceedings) a year earlier…

Another paper written during that period was actually published without any trouble at all, in the *Communications of the ACM* [H88]. It concentrates on the graphical properties of the statecharts language, disregarding the intended semantics of nodes as dynamic

[4] A note on the title page of the published version states that the paper was "Received December 1984, Revised July 1986". The first of these is an error – probably made in the typesetting stage – since submission to Pnueli was made in December 1985. By the way, this story of the repeated rejections of the paper would not be as interesting were it not for the fact that in the 20 years or so since its publication it has become quite popular. According to Citeseer, it has been for several years among the top handful of most widely quoted papers in computer science, measured by accumulated citations since publication (in late 2002 it was in the second place on the list).

states and edges as transitions. The paper defined a **higraph** to be the graphical artifact that relates to a directed graph just as a statechart relates to a finite-state diagram.[5] The idea was to capture the notions of hierarchy, orthogonality, multilevel and multinode transitions, hyperedges, and so on, in a pure set-theoretic framework. It too contains material on statecharts (and a simplified version of the digital-watch example) and since it appeared in a journal with a far broader readership than *Science of Computer Programming* it is often used as the de facto reference to statecharts.

The third paper that should be mentioned here was on the semantics of statecharts [HPSR87]. It was written jointly with the "semantics group" — the people involved in devising the best way to implement statecharts in Statemate — and provided the first formal semantics of the language. However, some of the basic decisions we made in that paper were later changed in the design of the tool, as discussed in Section 10.

Another paper was the one written on the Statemate tool itself [H+88&90], co-authored by the entire Statemate team at Ad-Cad/I-Logix. Its appeal, I think, goes beyond the technical issue of showing how statecharts can be implemented (the lack of which several of the referees of the basic statecharts paper complained about). In retrospect, as mentioned earlier, it set the stage for and showed the feasibility of the far broader concepts of **model-driven development**, true **model executability** and full **code generation**. These are elaborated upon in a later, more philosophical paper, "*Biting the Silver Bullet*" [H92], which also contained a rebuttal of Fred Brooks' famous "*No Silver Bullet*" paper [B87].

To close this section, an unfortunate miscalculation with regards to publication should be admitted. This was my failure to write a book about the language early on. As mentioned in the previous section, it was only in 1996 that the book with Michal Politi on Statemate was published [HP96]. This was definitely a mistake. I did not realize that most engineers out there in the real world rarely have the time or inclination to read papers, and even if they do they very rarely take something in a paper seriously enough to become part of the their day-to-day practice. One has to write a book, a popular book. I should have written a technical book on statecharts, discussing the language in detail, using many examples, describing the tool we already had that supported it, and carefully working out and describing the semantics too. This could have helped expose the language to a broader audience a lot earlier.

9. 1994–1996: The Object-Oriented Version

In the early 1990s Eran Gery, who at the time was (and still is) one of the key technical people at I-Logix, became very interested in object-oriented modeling. As it turned out, several people, including Jim Rumbaugh and Grady Booch, had written about the use of statecharts in object-oriented analysis and design; see, e.g., [B94, RBPEL91, SGW94]. It was Eran's opinion that their work left some issues that still had to be dealt with in more detail; for example, the semantics of statecharts were not worked out properly, as were the details of some of the dynamic connections with the objects. Also, they had not built a tool such as Statemate for this particular, more modern, OO approach. In the terminology of the present paper, their version of the language was not (yet) executable.

Despite being well aware of object-oriented programming and the OO programming languages that existed at the time, I was not as interested in or as familiar with this work on OO modeling as was Eran. Once Statemate had been designed and its initial versions built, the implementational issues that arose were being dealt with adequately by the I-Logix people, and I was spending most of my time on other topics of research. Eran did some gentle prodding to get me involved, and we ended up taking a much closer look at the work of Booch, Rumbaugh and others. This culminated in a 1996 paper, "*Executable Object Modeling with Statecharts*", in which we defined **object-oriented statecharts**, an OO version of the language, and worked out the way we felt the statecharts should be linked up with objects and executed [HG96&97]. One particular issue was the need for two modes of communication between objects, direct synchronous invocation of methods and asynchronous queued events. There were also many other aspects to be carefully thought out that were special to the world of objects, such as the creation and destroying of objects and multithreaded execution. The main structuring mechanism is that of a class in a class diagram (or an object instance in an object model diagram), each of which can be associated with a statechart. A new copy of the statechart is spawned whenever a new instance of the class is created. See Fig. 21 for two examples of statecharts taken from that paper.

In the paper we also outlined a new tool for supporting all of this, which I-Logix promptly started to build, called **Rhapsody**. Eran championed and led the entire Rhapsody development effort at I-Logix, and he still does.

And so we now have two basic tools for statecharts — Statemate, which is not object-oriented and is intended more for systems people and for mixed hardware/software systems, and Rhapsody, which is intended more for software systems and is object-oriented in nature. One important difference between the tools, which we shall elaborate upon in Section 10, is that the semantics of statecharts in Statemate is synchronous and in Rhapsody it 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 that is separate from the code generator. In contrast, the model execution in Rhapsody is carried out solely by running the code generated from the model. Thus, Rhapsody could be thought of as representing a high-level programming language that is compiled down into runnable code. Except, of course, that the statechart language is a level higher than classical programming languages, in that the translation from it was made into

[5] A sub-formalism of higraphs, which contains hierarchy and multi-level transitions has been called **compound graphs** [MS88,SM91].

C++, Java or C, etc. Another important difference is that a decision was made to make the action language of Rhapsody be a subset of the target programming language. So you would draw statecharts in Rhapsody and the events and actions specified along transitions and in states, etc., are fragments of, say, C++ or Java. (The action language in Fig. 21, for example, is C++.) These differences really turn Rhapsody into more of a high-level programming tool than a system-development tool. See also the discussion on the UML in Section 10.

There are now several companies that build tools that support statecharts. There are also many variants of the language. One of the most notable early tools is **ObjecTime**, built by Bran Selic and Paul Ward and others. This tool later became **RoseRT**, from Rational Corp. **StateRover** is another statechart tool, built by my former student, Doron Drusinsky. Finally, **Stateflow** is a statechart tool embedded in Matlab (which is used widely by people interested in control systems); its statecharts can be effortlessly linked to Matlab's other modeling and analysis tools.

It is worth viewing the implementation issue in a slightly broader perspective. In the early 1980s, essentially none of the tools offered for system development using graphical languages were able to execute models or generate running code. If one wants to be a bit sarcastic about it, these so-called CASE tools (the acronym standing for computer-aided software engineering) were not much more than good graphic editors, document generators, configuration managers, etc. It would not be much of an exaggeration to say that pre-1986 modeling tools were reminiscent of support tools for a programming language with lot of nice features but with no compiler (or interpreter). You could write programs, you could look at them, you could print them out, you could ask all kind of queries such as "list all the integer variables starting with D", you could produce documents, you could do automatic indentation, and many other niceties; everything except run the programs!

Of course, in the world of complex systems, tools that do these kinds of things − checking for the consistency of levels and other issues related to the validity of the syntax, offering nice graphic abilities for drawing and viewing the diagrams, automatically generating documents according to pre-conceived standards, and so on − are very important. Although these features are crucial for the process of building a large complex system, I was opposed to the hype and excitement that in pre-1986 years tended to surround such tools. My take was that the basic requirement of a tool for developing systems that are dynamic in nature is the ability not only to describe the behavior, but also to analyze it and execute it dynamically. This philosophy underlies the notion of a visual formalism, where the language is to be both diagrammatic and intuitive in nature, but also mathematically rigorous, with a well-defined semantics sufficient to enable tools to be built around it that can carry out dynamic analysis, full model execution and the automatic generation of running code; see [H92].

10. On Semantics

It is worth dwelling on the issue of **semantics** of statecharts. In a letter from Tony Hoare after he read the 1984 technical report on statecharts, he said very simply that the language "badly needs a semantics". He was right. I was overly naïve at the time, figuring 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, of course, where people invent languages, describe them in the literature and then build compilers for them. That this was naïve is a consequence of the fact that there are several very subtle and slippery issues around the semantics of any concurrent language − statecharts included. These not only have to be decided upon when one builds a tool, something we actually took great pains to do properly when designing Statemate, but they also have to be written up properly for the scientific community involved in the semantics of languages.

In retrospect, what we didn't fully realize in those early years was how different statecharts were from previous specification languages for real-time embedded systems − for better or for worse. We knew that the language had to be both executable and easily understandable by many different kinds of people who hadn't received any training in formal semantics. But at the same time, as a team wanting to build a tool, we also had to demonstrate quickly to our sponsors, the first being IAI, that ours was an economically viable idea; so we were under rather great time pressure. Due to the high level of abstraction of statecharts, we had to resolve several rather deep semantical problems that apparently hadn't been considered before in the literature, at least not in the context of building a real-world tool intended for large and complex systems. What we didn't know was that some of these very issues were being investigated independently, around the same time, by various leading French researchers, including Gérard Berry, Nicholas Halbwachs and Paul le Guernic (who later coined the French phrase *L'approche synchrone*, "the synchronous approach", for their kind of work).

In actuality, during the 1984-6 period of designing Statemate, we did not do such a clean and swift job of deciding on the semantics. We had dilemmas regarding several semantic issues, a couple of which were particularly crucial and central. One had to do with whether a step of the system should take zero time or more, and another had to do with whether the effects of a step are calculated and applied in a fixpoint-like manner in the same step, or are to take effect only in the following one. The two issues are essentially independent; one can adopt any of the four combinations. Here is not the proper place to explain the subtlety of the differences, but the first issue, for example, has to do with whether or not you adopt the pure **synchrony hypothesis**, generally attributed to Berry, whereby steps take zero time [BG92]. Of course, these questions have many consequences in terms of how the language operates, whether events can interfere with chain reactions triggered by other events, how time itself is modeled, and how time interleaves with the discrete event dynamics of the system.

During that period the main people who were sitting around the table discussing this were Amir Pnueli, Rivi Sherman, Janette Schmidt, Michal Politi and myself, and for some time we used the code names Semantics A and B for the two main approaches we were seriously considering. Both semantics were synchronous in the sense of [BG92] and differed mainly in the second issue above. The paper we published in 1987 was based on Semantics B [HPSR87], but we later adopted semantics A for the Statemate tool itself, which was rather confusing to people coming from outside of our group. In 1989, Amnon Naamad and I wrote a technical report that described the semantics adopted in Statemate [HN89&96], i.e., Semantics A, where the effects of a step are accumulated and are then carried out in the following step. At the time, we did not think that this report was worth publishing — naiveté again — so for several years it remained an internal I-Logix document.

In any case, the statecharts of Statemate really constitute a **synchronous language** [B+03], and in that respect they are similar to other, non visual languages in this family, such as Berry's Esterel [BG92], Lustre [CPHP87] and Signal [BG90].

At that time, a number of other researchers started to look at statechart semantics, some being motivated by our own ambivalence about the issue and by the fact that the implemented semantics was not published and hence not known outside the Statemate circle. For example, in an attempt to evaluate the different semantics for statecharts, Huizing, Gerth and de Roever proved one of them to have the desirable property of being fully abstract [HGdR88]. As the years went by, many people defined variants of the statechart language, sometimes dropping orthogonality, which they deemed complicated, and often adding some features or making certain modifications. There were also several papers published that attempted to provide formal, machine-readable semantics for the language, and others that described other tools built around variants thereof.

An attempt to summarize the situation was carried out by von der Beeck, who tried to put some order into the multitude of semantics of statecharts that were being published. The resulting paper [vB94] claimed implicitly that statecharts is not really a well-defined language because of these many different semantics (it listed about twenty such). Interestingly, while [vB94] reported on the many variants of the language with the many varying semantics, it did not report on what should probably have been considered at the time the "official" semantics of the language. This is the semantics we defined and adopted in 1986-7 when building Statemate [HN89&96]; the one I talked about and demonstrated in countless lectures and presentations in the preceding 8 years, but, unfortunately, the only one not published at the time in the widely-accessible open literature…

Around the same time another paper was published, by Nancy Leveson and her team [LHHR94], in which they took a close look at yet another statecharts semantics paper, written by Pnueli and Shalev [PS91]. The Pnueli/Shalev paper provided a denotational fixpoint semantics for statecharts and elegantly showed its equivalence to a certain operational semantics of the language. Leveson and her group did not look at the Statemate tool either and, like von der Beeck, had not seen our then-unpublished technical report [HN89&96]. The Leveson et al paper was very critical of statecharts, going so far as to hint that the language is unsafe and should not be used, the criticism being based to a large extent on anomalies that they claimed could surface in systems based on the semantics of [PS91].

It seems clear that had a good book about statecharts been available early on, including its semantics and its Statemate implementation, some of this could have been avoided. At the very least we should have published the report on the Statemate semantics. It was only after seeing [vB94, LHHR94] and becoming rather alarmed by the results of our procrastination that we did just that, and the paper was finally published in 1996 [HN89&96].

As to the semantic issues themselves, far more important than the differences between the variants of pre-OO statecharts themselves, as reported upon in [vB94], is the difference between the non-object-oriented and the object-oriented versions of the language, as discussed above. The main semantic difference is in synchronicity. Statemate statecharts, i.e., the version of the language based on functional decomposition, is a synchronous language, whereas Rhapsody statecharts, i.e., the object-oriented version thereof, is an asynchronous one. There are other substantial differences in modes of communication between objects, and there are the issues that arise from the presence of dynamic objects and their creation and destruction, inheritance, object composition, multithreading, and on and on. All these have to be dealt with when one devises an object-oriented version of such a language and builds a tool like Rhapsody, which supports both the objects and their structure and the statecharts and code that drive their behavior.

In the object-oriented realm, a similar publication sin was committed, waiting far too long to publish the semantics of statecharts in Rhapsody. Only very recently, together with Hillel Kugler, did we finally publish a paper (analogous to [HN89&96]) that gives the semantics of statecharts as adopted in Rhapsody and describes the differences between these two subtly different versions of the language [HK04].

This section on semantics cannot be completed without mentioning the **unified modeling language**, the **UML**; see [RJB99,UML]. As the reader probably well knows, Rumbaugh and Booch, together with Ivar Jacobson, got together to form the technical core team of the impressive UML effort, which was later standardized by the object management group (OMG). Although the UML features many graphical languages, many of them have not been endowed with satisfactorily rigorous semantics. The heart of the UML − what many people refer to as its driving behavioral kernel − is the (object-oriented variant of the) statecharts language; see Section 9. In the late 1990s Eran Gery and I took part in helping this team define the intended meaning of statecharts in the UML. This had the effect of making UML statecharts very similar to what we had already implemented in Rhapsody.

In fact, currently the two main executable tools for UML-based languages are Rhapsody and RoseRT; the latter, as mentioned above, is a derivative of the earlier ObjecTime tool, and implements a sublanguage of statecharts: for example, it does not support orthogonal

state components.[6] There are other differences between these two tools that the present paper cannot cover. Also, the issue of whether the action language should be the target programming language, as in Rhapsody, or whether there should be an autonomous action language is still raging in full force and the UML jury is not yet in on this issue.

See the recent [HR04], with its whimsical title "*What's the Semantics of 'Semantics'?*", for a manifesto about the subtle issues involved in defining the semantics of languages for reactive systems, with special emphasis put on the UML.

11. Biological Modeling with Statecharts

In terms of usage of statecharts, the language appears to be used very widely in computer embedded and interactive systems, e.g., in the aerospace and automotive industries, in telecommunication and medical instrumentation, in control systems, and so on. However, one of the more interesting developments involves statecharts also being used in non-conventional areas, such as modeling biological systems and health-care processes.

Starting in the mid-1980s I had often claimed that biological systems should be viewed as systems the way we know them in the world of computing, and **biological modeling** should be attempted using languages and tools constructed for reactive systems, such as statecharts. One modest attempt to do so was made by a student in our department, Billie Sandak, around 1989. This work was not carried out to completion, and the topic was picked up about ten years later by another student, Naaman Kam, co-supervised by Irun Cohen, a colleague of mine from the Weizmann Institute's Immunology department. The resulting work (written up in [KCH01]) started a flurry of activity, and by now several serious efforts have been made on using statecharts to model biological systems. This includes one particularly extensive effort of modeling T cell development in the thymus gland, done with our student Sol Efroni [EHC03], and others involving, e.g., the pancreas [SeCH06] and the lymph node [SwCH06]. The thymus model, for example, contains many thousands of complex objects, each controlled by a very large and complicated statechart, and has resulted in the discovery of several properties of the system in question; see the recent [EHC07]. Figs. 22 and 23 show, respectively, the front-end of this model and a schematic rendition of parts of the statechart of a single cell. Figs. 24 and 25 show more detailed parts of some of the statecharts from the thymus model during execution.

One of the notions that we came up with during our work on the thymus model is **reactive animation** [EHC05]. The idea is to be able to specify systems for which the front end requires something more than a GUI — specifically, systems that require true animation. A good example would be a traffic or radar system with many elements and targets, such as cars or aircraft, being created, moving in and out of the scene, traveling around, growing and shrinking in size, changing and getting destroyed, etc. Under normal circumstances, this kind of system would have to be programmed using the script language supported by an animation system. Reactive animation allows one to use a state-of-the-art reactive system tool, such as Statemate or Rhapsody, linked up directly and smoothly with an animation tool. The T cell model of [EHC03, EHC07] was built using statecharts in Rhapsody, linked up with the Flash animation tool, and the two work together very nicely. Reactive animation is used extensively also in the pancreas and lymph node models [SeCH06, SwCH06].

12. Miscellaneous

This section discusses some related topics that came up over the years. One is the notion of **overlapping states**, whereby you want the and/or state hierarchy in statecharts to be a directed graph, not a tree. This possibility, and the motivation for it, was already mentioned in the earliest documents on statecharts; see Fig. 26. In work with an MSc student, H.-A. Kahana, the details of how overlapping could be defined were worked out [HK92]. We found that the issue was pretty complicated since, e.g., overlapping can be intermixed not only with the substate facet of the hierarchy but also with orthogonal components. We actually concluded that the complications might outweigh the benefits of implementing the feature.

Although the basic idea is very natural, it appears that such an extension is not yet supported in any of the statechart tools. Incidentally, this does not prevent people from thinking that overlapping is a simple matter, since it is tempting to think only of simple cases, like that of Fig. 26. Some people have approached me and asked "Why doesn't your tool allow me to draw one state overlapping the other? Why don't you simply tell it not to give me the error message when I try to do this in the graphic editor?" Of course, underlying such questions is the naïve assumption that if you can draw a picture of something, and it seems to make sense to you, then there is no problem making it part of the language… I often use this exchange to illustrate the difference between what many people expect of a visual language and what a real visual formalism is all about; see the discussion on "the doodling phenomenon" in [HR04].

An additional topic is that of **hybrid systems**. It is very natural to want to model systems that have both discrete and continuous aspects to them. In discussions and presentations on statecharts in the 1980s, I often talked about the possibility of using techniques from control theory and differential equations to model the activities occurring within states in a statechart, but never actually did any work on the idea. Many years later the notion of a hybrid (discrete and continuous) system was put forward by several people, and today there is an active community doing deep research in the area. Many models of hybrid systems are essentially finite-state machines, often rendered using statecharts that are intermixed with techniques for specifying continuous aspects of a system, such as various kinds of differential equations.

The other idea I have been trying to peddle for years but have done nothing much about is to exploit the

[6] By the way, Rational's most popular tool, Rational Rose, cannot execute models or produce executable code. In that respect it suffers from the same central weakness afflicting the pre-1986 CASE tools.

structure of the behavior given in statecharts to aid in the **verification** of the modeled system. The philosophy behind this is as follows. We all know that verification is hard, yet there are techniques that work pretty well in practice, such as those based on model checking. However, common verification techniques do not exploit the hierarchical structure or modularity that such models very often have. Now, assume that someone has already made the effort of preparing a statechart-based description of a complex system, and has gone to great pains in order to structure the statecharts nicely to form a hierarchy with multilevel orthogonal components. There should probably be a way to exploit the reasons for the decisions made in this structuring process in carrying out the verification. Perhaps the way to do it is to try to get more information from the "statifier", i.e., the person preparing the statecharts, about the knowledge he or she used in the structuring. For example, just as we expect someone writing a program with a loop to be able to say more about the invariant and convergent properties of that loop, so should we expect someone breaking a system's state space into orthogonal components, or deciding to have a high-level state encompass several low-level states, to be able to say something about the independent or common properties of these pieces of the behavior.

There has actually been quite a lot of work on the verification (especially model checking) of hierarchical state machines, and the availability of various theoretical results on the possibility (or lack thereof) of obtaining significant savings in the complexity of verifying concurrent state machines. There are also some tools that can model-check statecharts. However, my feeling is that the jury is not in yet regarding whether one can adequately formalize this user-provided information and use it beneficially in the verification process.

Finally, I should mention briefly the more recent work with colleagues and students, which can be viewed as another approach, to visual formalisms for complex systems. It has to do with **scenario-based** specification. The statechart approach is **intra-object**, in that ultimately the recommendation is to prepare a statechart for each object of the system (or for each task, function, component, etc., whatever artifacts your system will be composed of). Of course, the statecharts are to also contain information about the communication between the objects, and one could build special controlling statecharts to concentrate on these aspects; however, by and large, the idea of finite-state machines in general, and statecharts in particular, is to provide a way for specifying the behavior of the system per object in an intra-object fashion. The more recent work has to do with scenario-based, **inter-object** specification. The idea is to concentrate on specifying the behavior between and among the objects (or tasks, functions, components, etc.). The main lingua franca for describing the behavior of the system would have to be a language for specifying communication and collaboration between the objects. This became feasible with the advent of **live sequence charts** (or **LSCs**, for short) in work joint with Werner Damm in 1999; see [DH99&01]. Later, with my student Rami Marelly, a means for specifying such behavior directly from a GUI was worked out, called **play-in**, as well as a means for executing the behavior, called **play-out**, and the entire setup and associated methods have been implemented in a tool called the **Play-Engine**; see [HM03].

We have also built mechanisms to bridge between the two approaches, so that one can connect one or more Play-Engines with other tools, such as Rhapsody (see [BHM04]). In this way, one can specify part of the behavior of the system by sequence charts in a scenario-based, inter-object, fashion, and other objects can be specified using statecharts, or even code, in an intra-object fashion.

13. Conclusions

In summary, it would seem that one of the most interesting aspects of this story of statecharts in the making is in the fact that the work was not done by an academic researcher sitting in his/her ivory tower, inventing something and trying to push it down the engineers' throats. Rather, it was done by going into the lion's den, so to speak, working in industry and with the people in industry. This is consistent with the saying that "the proof of the pudding is in the eating".

Other things crucial to the success of a language and an approach to system-development are good supporting tools and excellent colleagues. In my own personal case, both the IAI engineers and the teams at AdCad/I-Logix who implemented the Statemate tool and then the Rhapsody tool were an essential and crucial part of the work. And, of course, a tremendous amount of luck is necessary, especially, as in this case, when the ideas themselves are not that deep and not that technically difficult.

I still believe that almost anyone could have come up with statecharts, given the right background, being exposed to the right kinds of problems and being surrounded by the right kinds of people.

Acknowledgments

Extensive thanks are due to my many colleagues at the Israel Aircraft Industries, at Ad-Cad/I-Logix and at The Weizmann Institute. Some of these have been mentioned and credited in the text itself, but I'd like to express particularly deep gratitude to Jonah Lavi, Amir Pnueli, Eran Gery, Rivi Sherman and Michal Politi. In addition, Moshe Vardi, Willem de Roever and the HOPL III referees made valuable comments on early versions of the paper. The process of writing of this paper was supported in part by the John von Neumann Center for the Development of Reactive Systems at the Weizmann Institute, and by grants from Minerva and the Israel Science Foundation.

(Note: the References section appears after the figures)

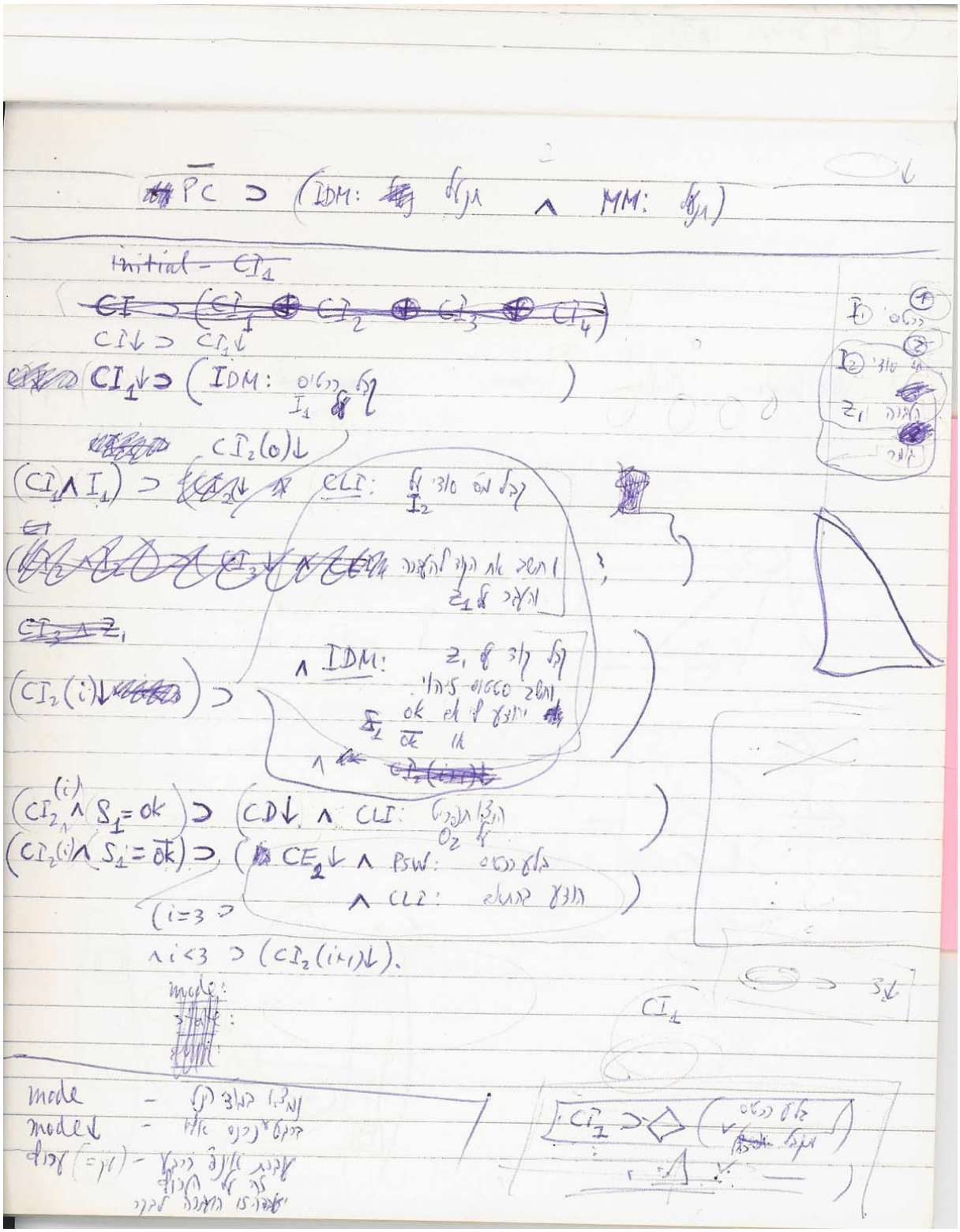


Figure 1. Page from the IAI notes (early 1983; with some Hebrew) showing the first attempt at helping specify the Lavi avionics, using a temporal logic-like formalization.

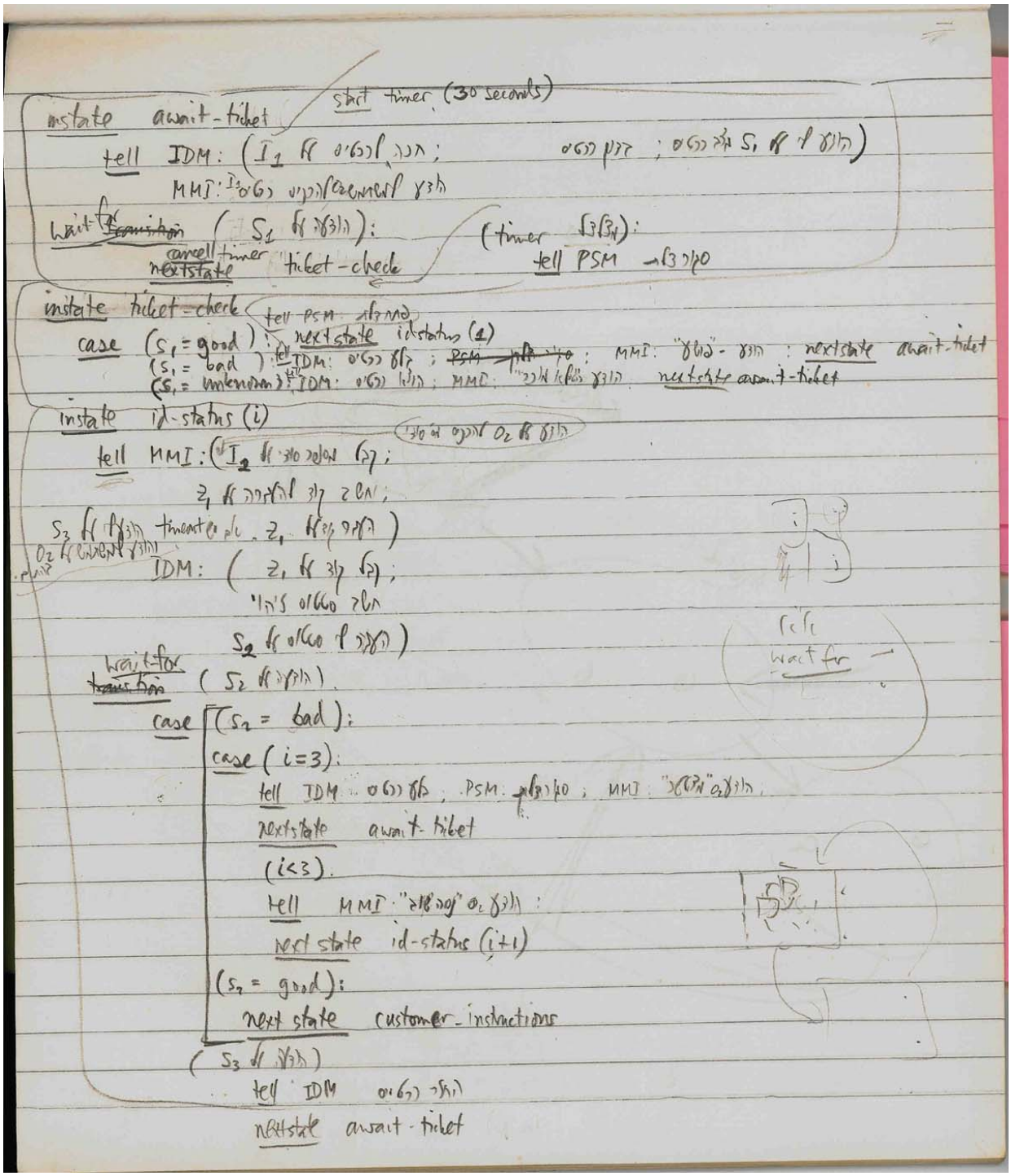


Figure 2. Page from the IAI notes (early 1983; with some Hebrew) showing parts of the Lavi avionics behavior using "*statocols*", the second attempt — a kind of structured state-transition protocol language. Note the graphical "doodling" on the right hand side, which was done to help clarify things to the engineers, and which quickly evolved into statecharts.

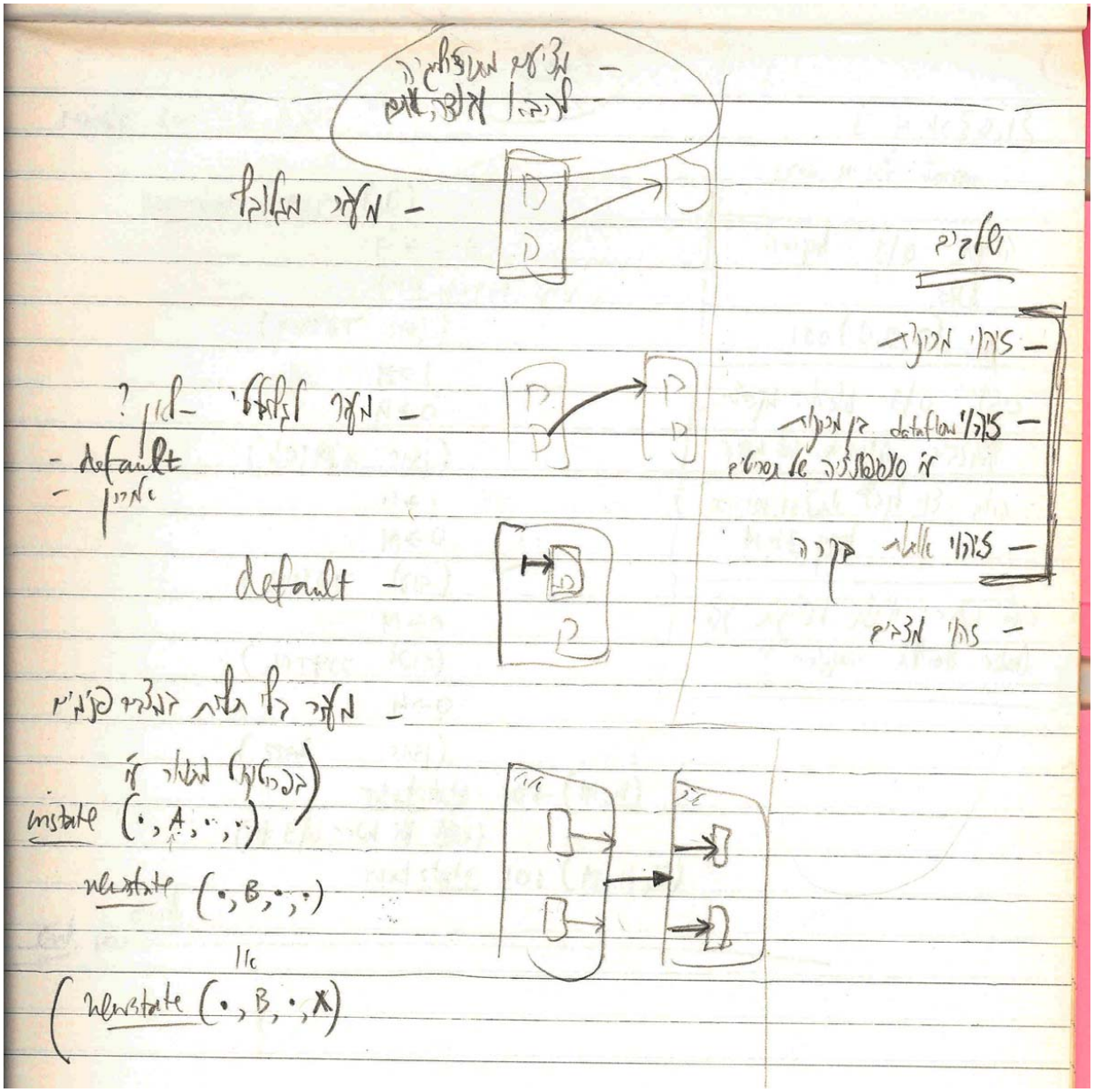


Figure 3. Page from the IAI notes (mid-1983; in Hebrew) showing a first attempt at deciding on graphical/topological elements to be used in the hierarchy of states. Note the use of the term *default* as a generalization to hierarchical states of the notion of a start state from automata theory.

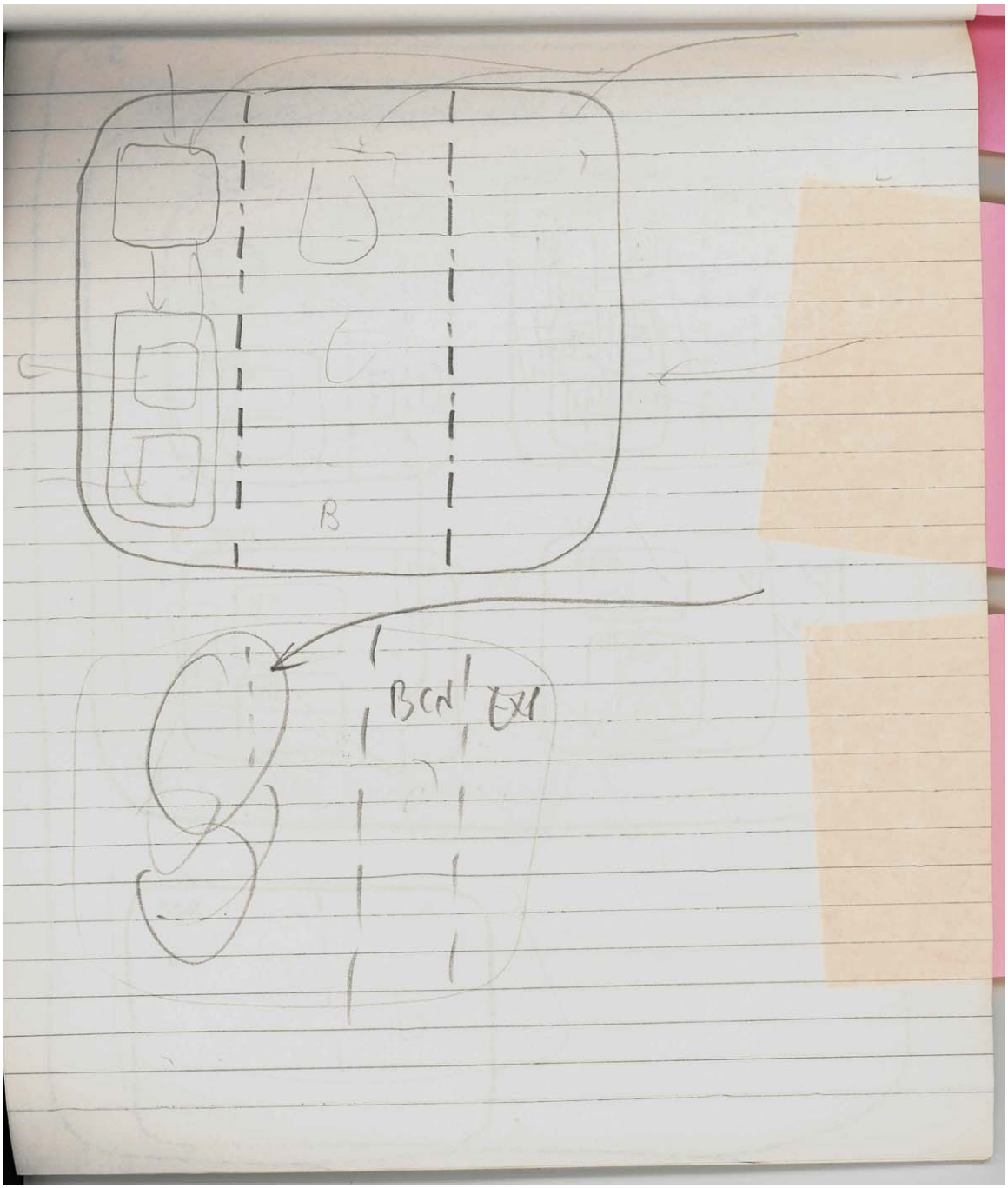


Figure 4. Page from the IAI notes (mid-1983) showing the first rendition of orthogonal state components. Note the hesitation about what style of separation lines to use.

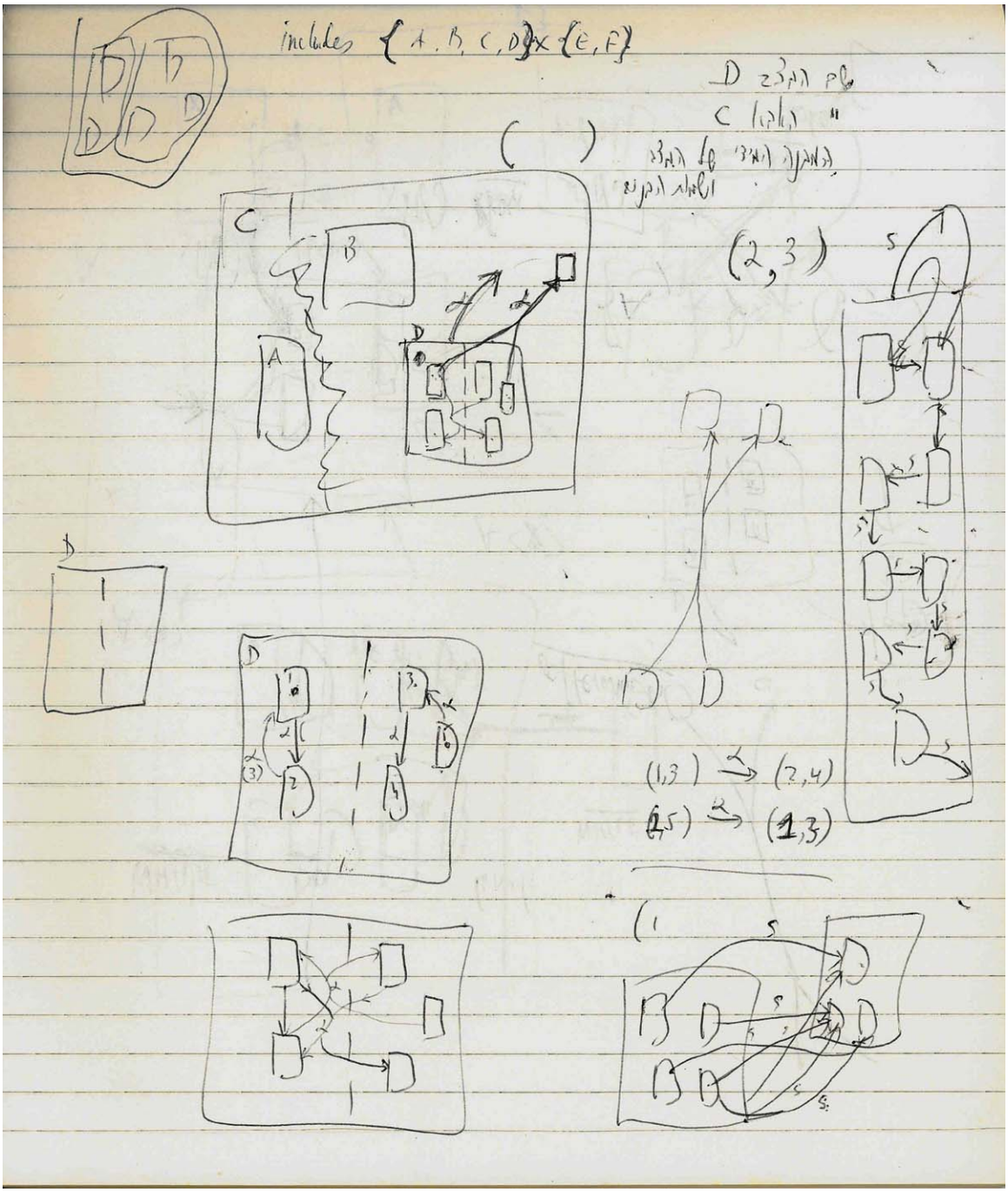


Figure 5. Page from the IAI notes (mid-1983). Constructs shown include hyper-edges, nested orthogonality, transitions that reset a collection of states (chart on right). Note the use of Cartesian products of sets of states (top) to capture the meaning of orthogonality, and the straightforward algebraic notation for transitions between state vectors (bottom third of page, on right).

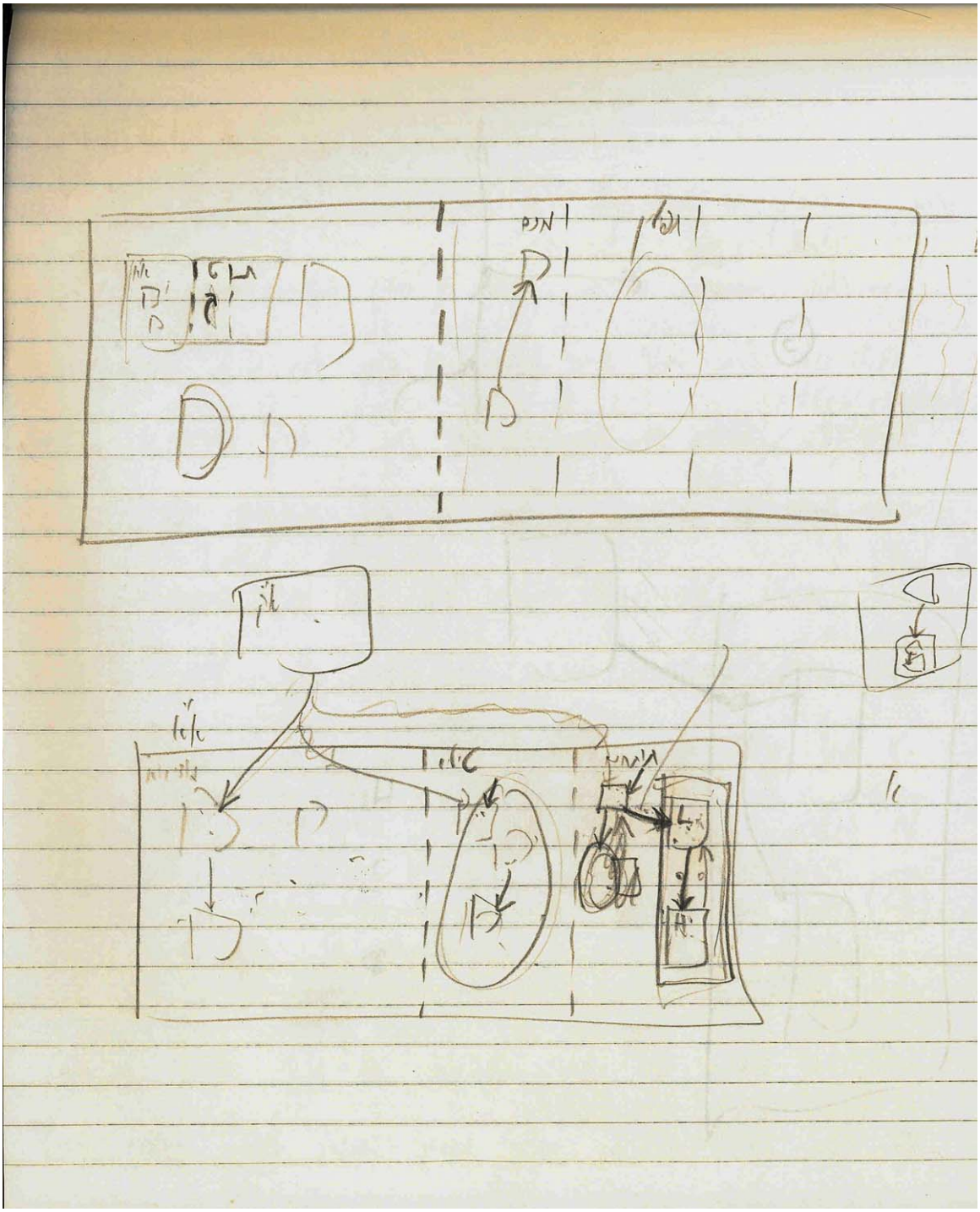


Figure 6. Page from the IAI notes (mid-1983) showing some initial statechart attempts for the Lavi avionics. Note the nested orthogonality (top left) and the inter-level transitions (bottom).

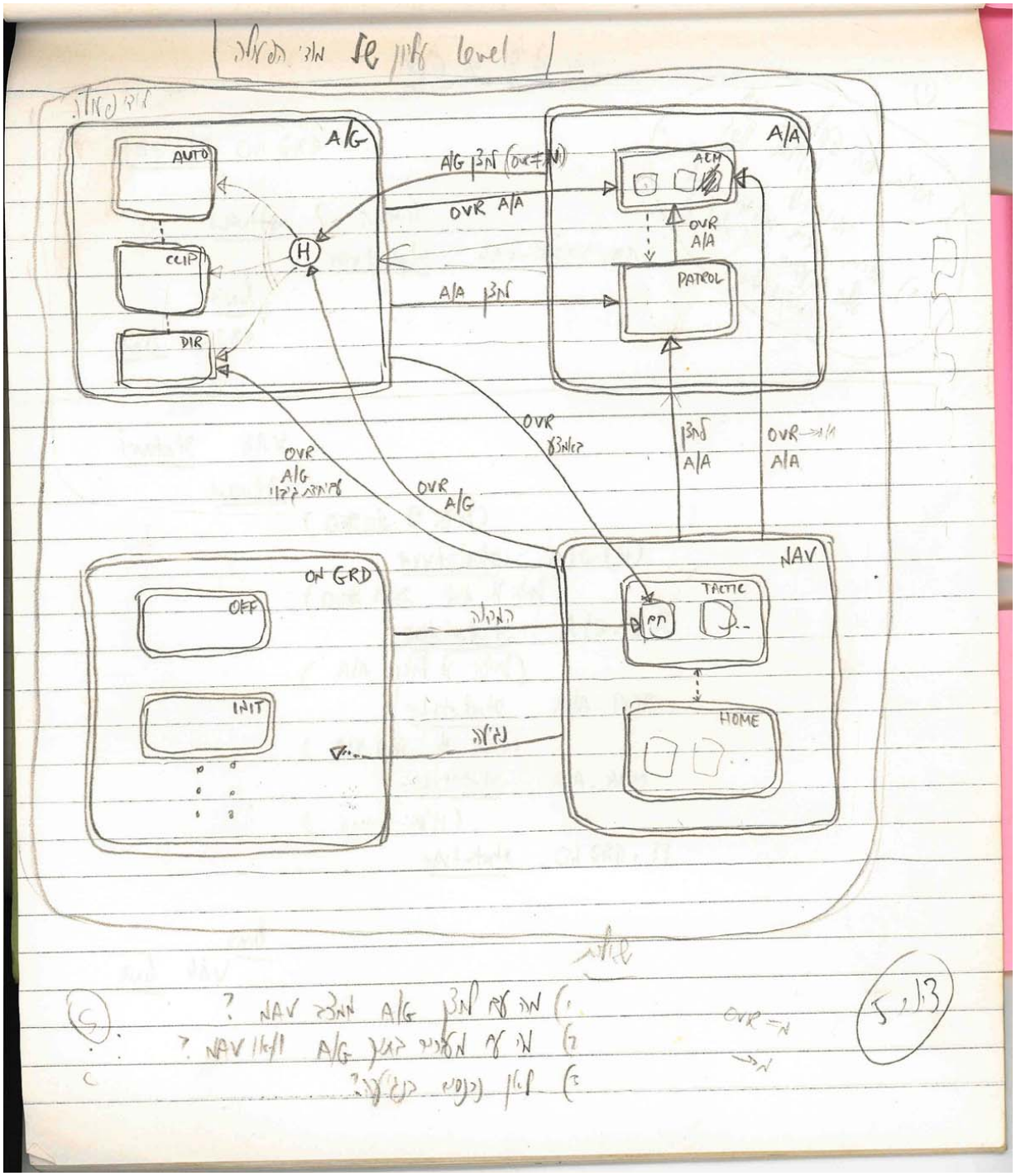


Figure 7. Page from the IAI notes (mid-1983; events in Hebrew) showing a relatively "clean" draft of the top levels of behavior for the main flight modes of the Lavi avionics. These are A/A (air-air), A/G (air-ground), NAV (automatic navigation) and ON GRD (on ground). Note the early use of a *history* connector in the A/G mode.

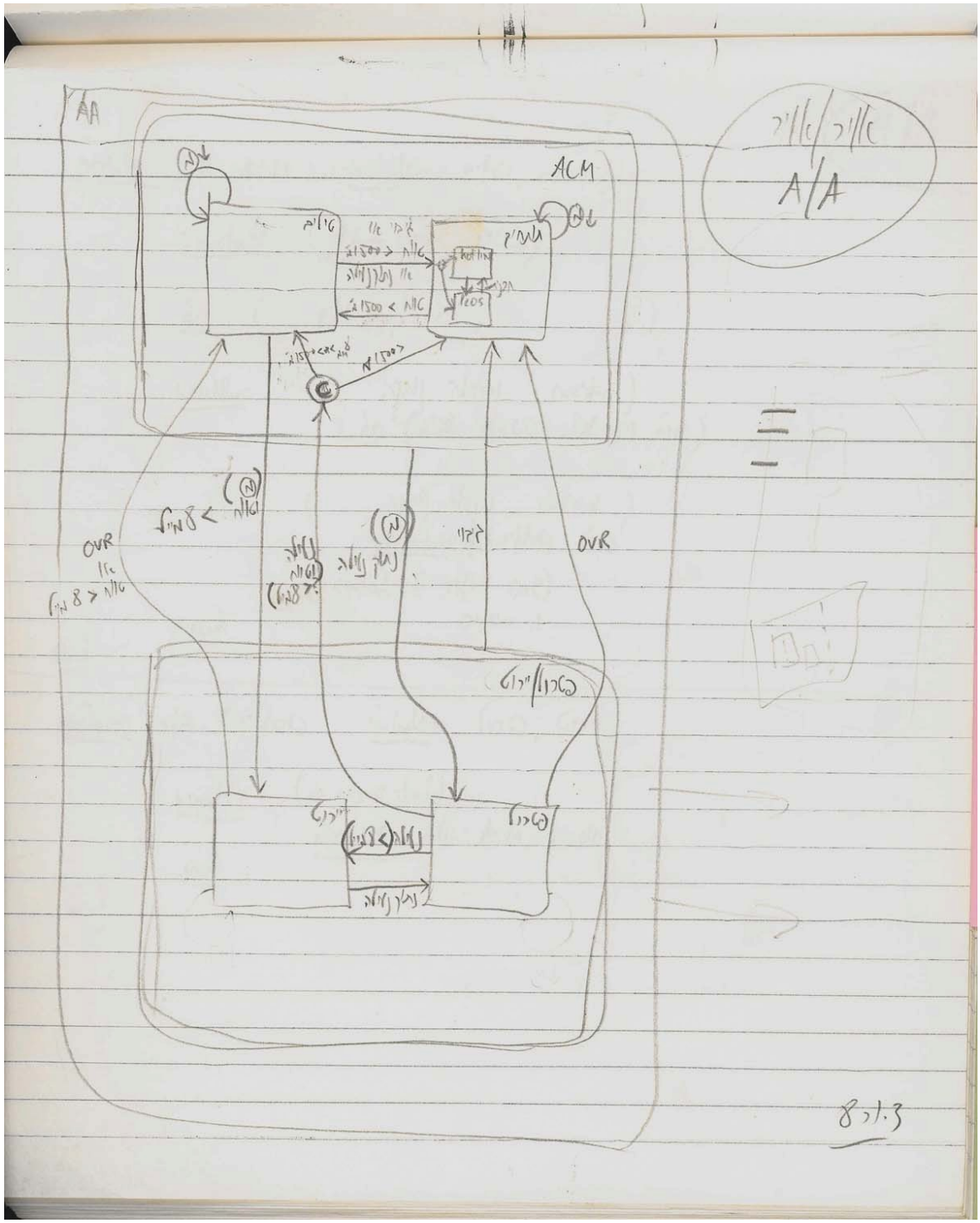


Figure 8. Page from the IAI notes (mid-1983) showing the inner statechart specification of the A/A (air-air) mode for the Lavi avionics.

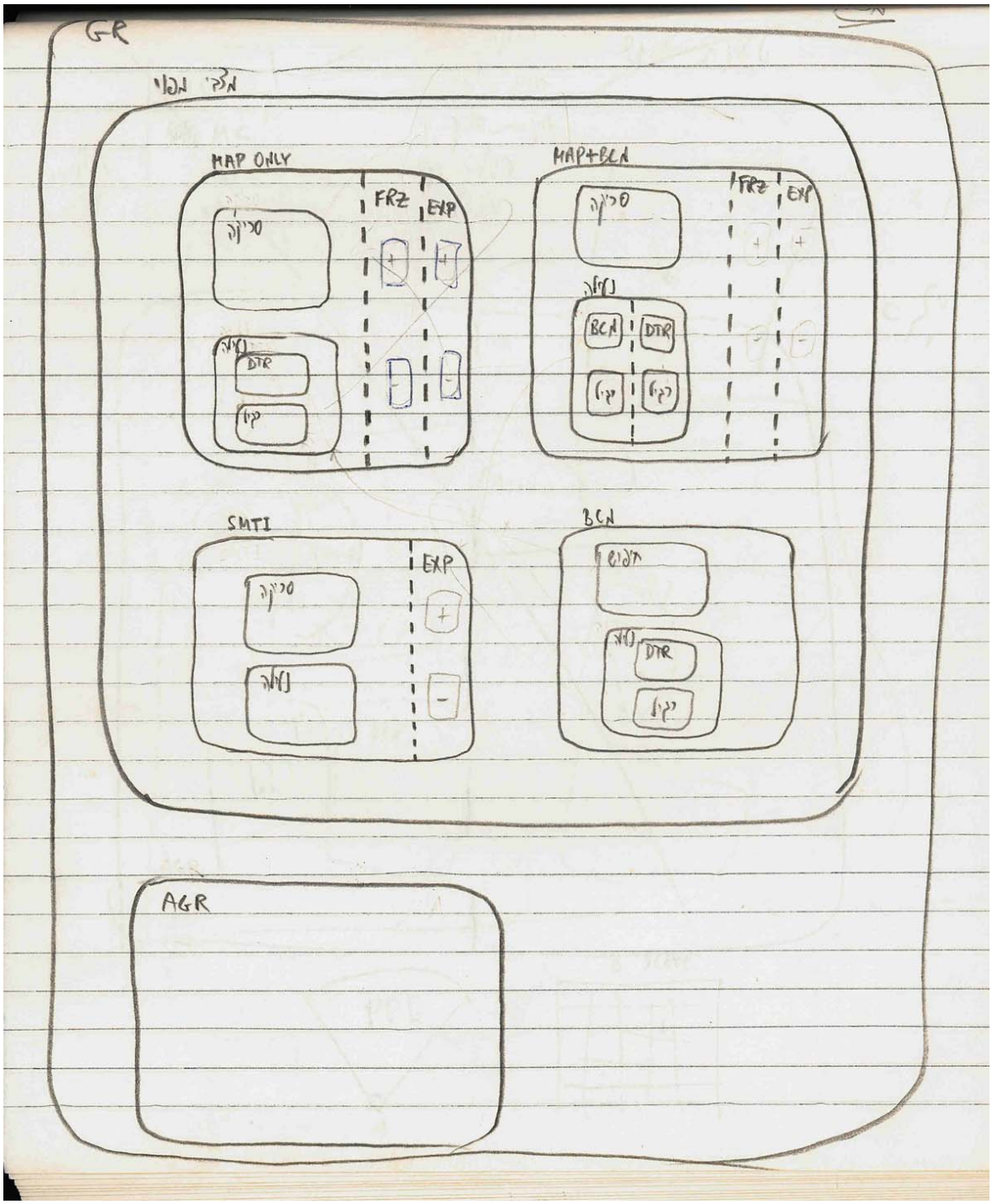


Figure 9. Page from the IAI notes (late 1983) showing multiple-level orthogonality in a complex portion of the Lavi avionics. Most of the orthogonal components on all levels here are not tangible components of the system, but rather exhibit a natural way of conceptually breaking up the state space.

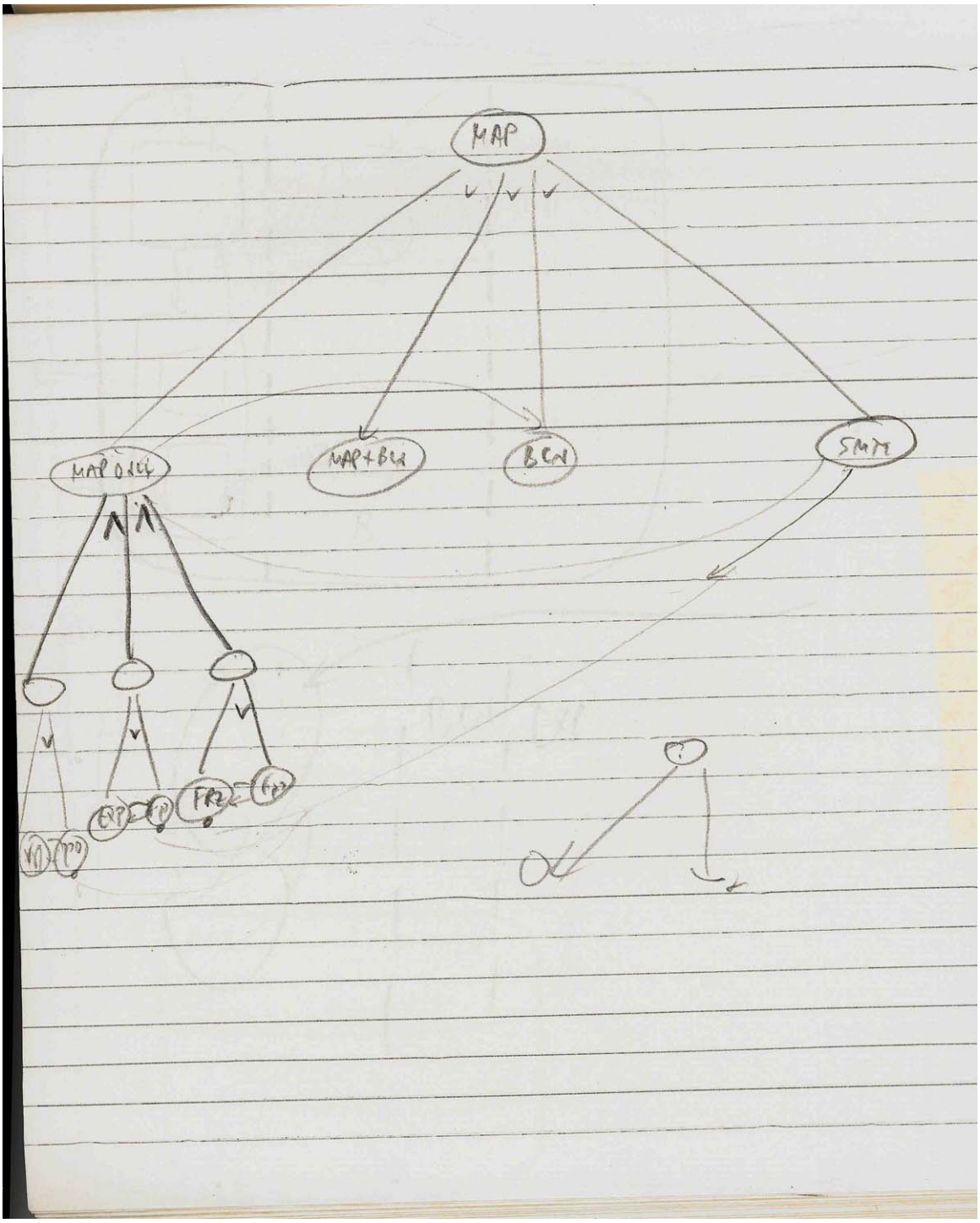


Figure 10. Page from the IAI notes (late 1983) showing an and/or tree rendition of (part of) the state hierarchy in Fig. 9.

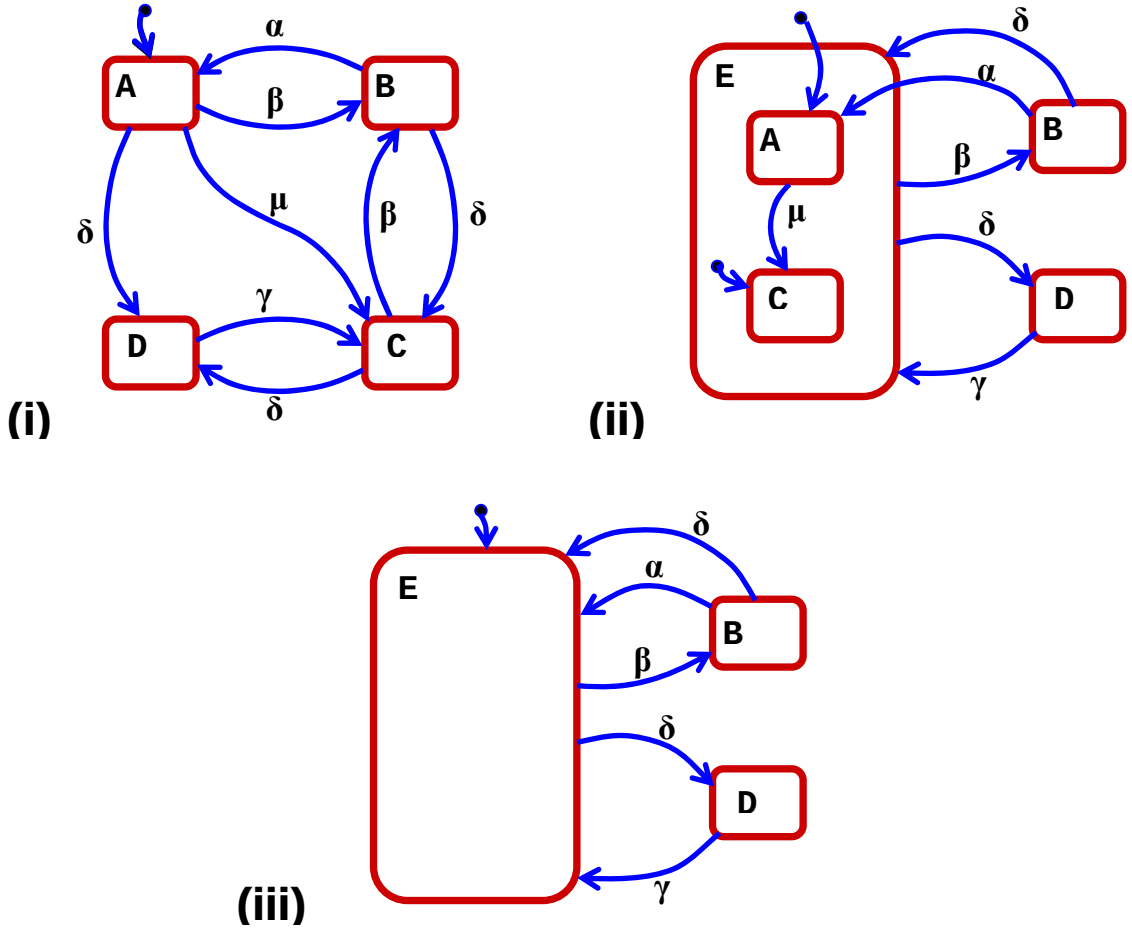


Figure 11. Illustrating hierarchy in statecharts: multi-level states, transitions, default entrances, refinement and abstraction.

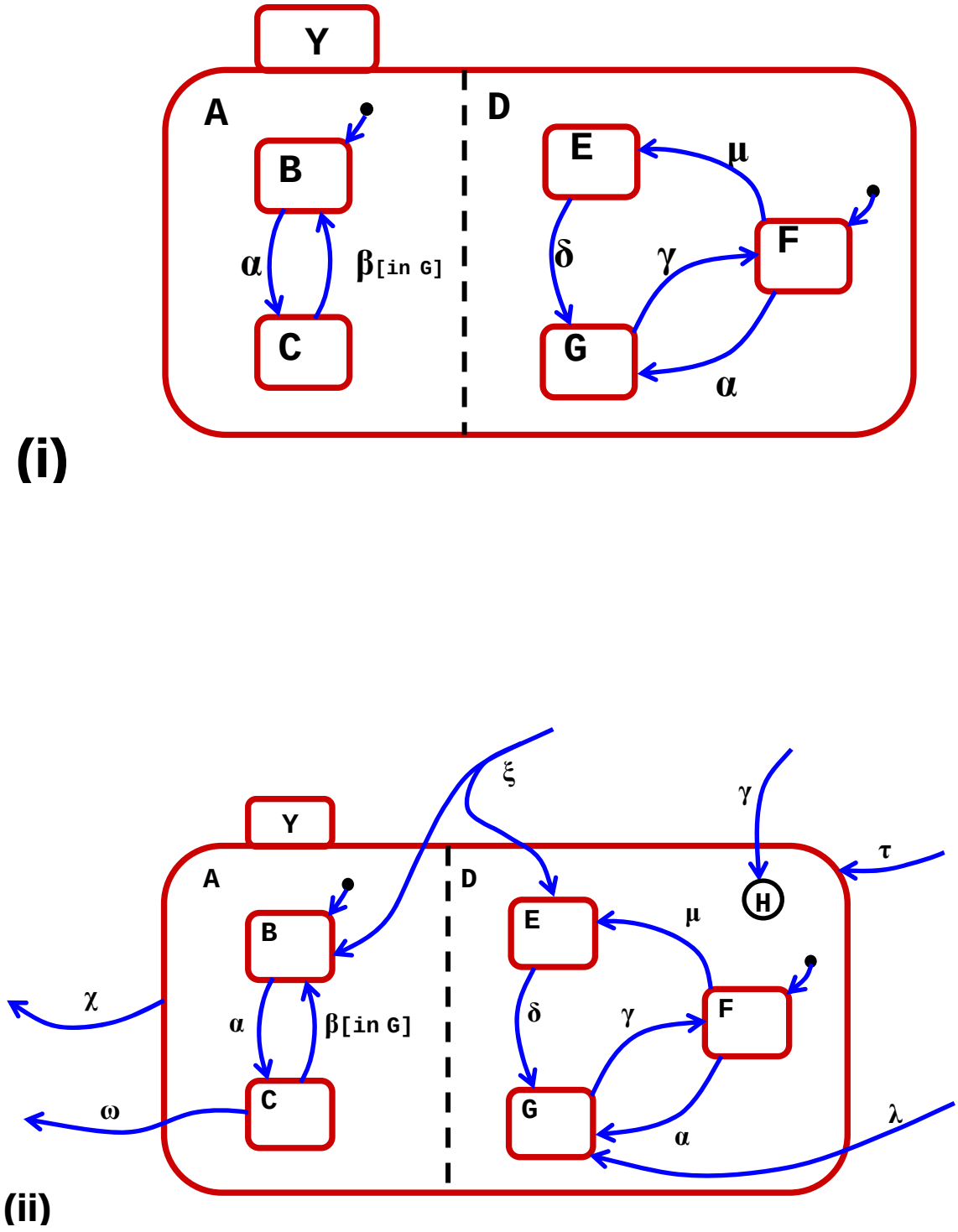


Figure 12. Orthogonality in statecharts, with and without out exits from and entrances to other states.

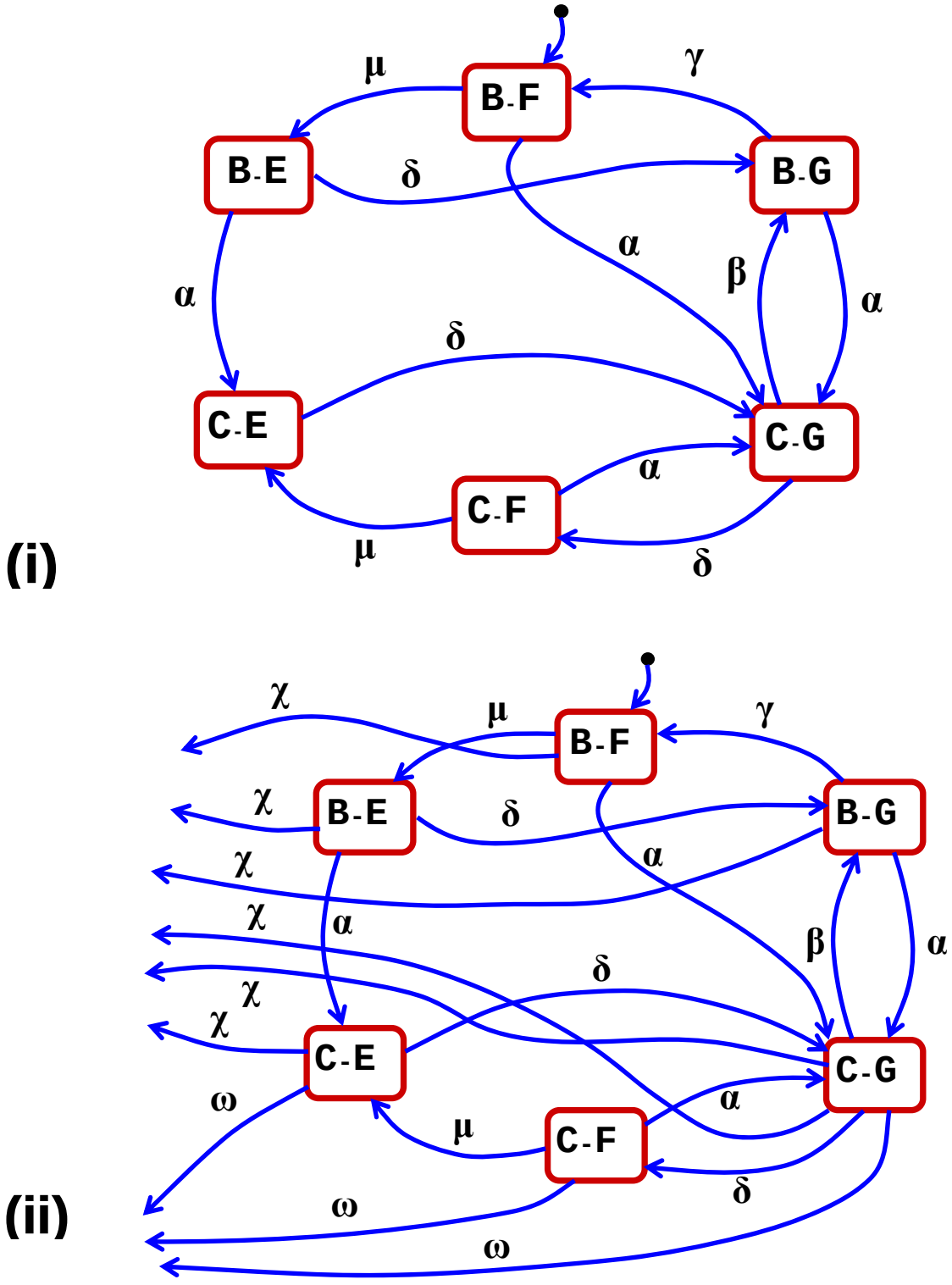


Figure 13. "Flattened" orthogonality-free versions of the two parts of Fig. 12, minus the external entrances in 12(ii). (These are really the Cartesian products.)

.

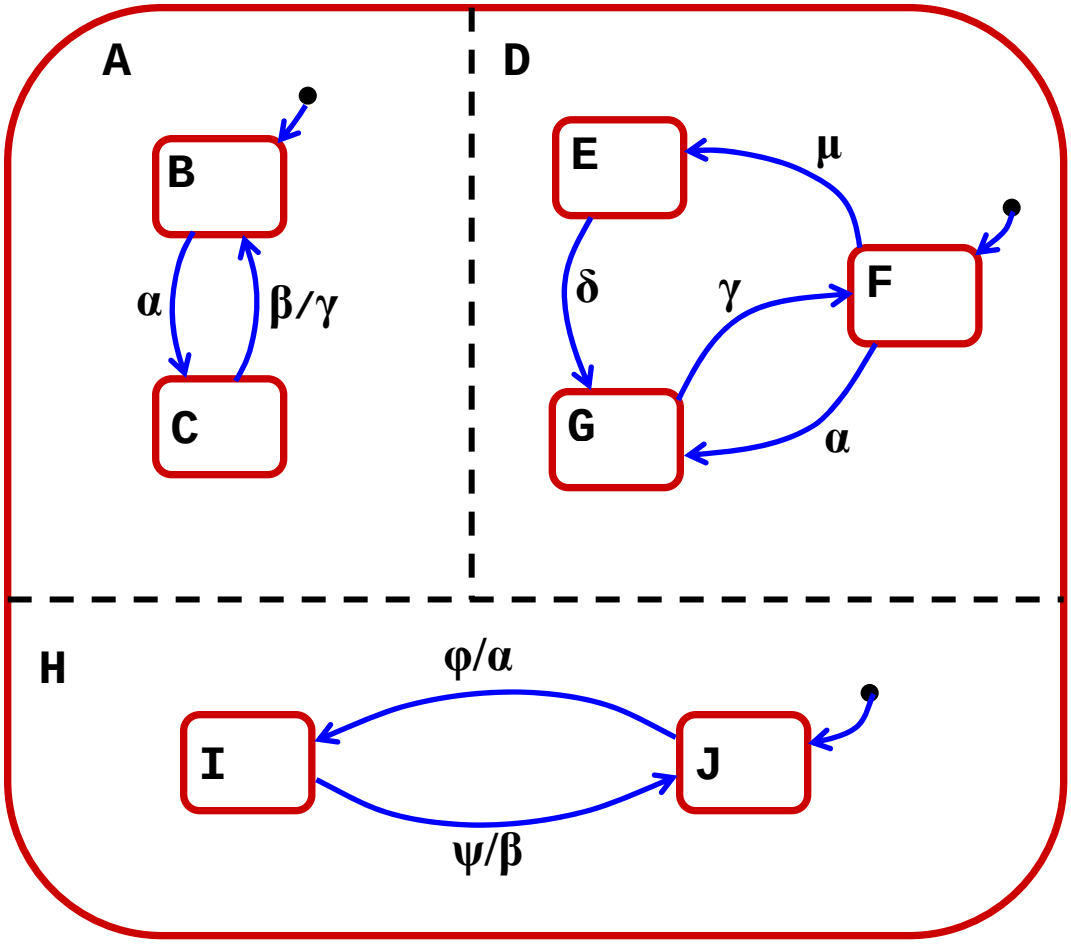


Figure 14. Broadcasting within a single statechart.

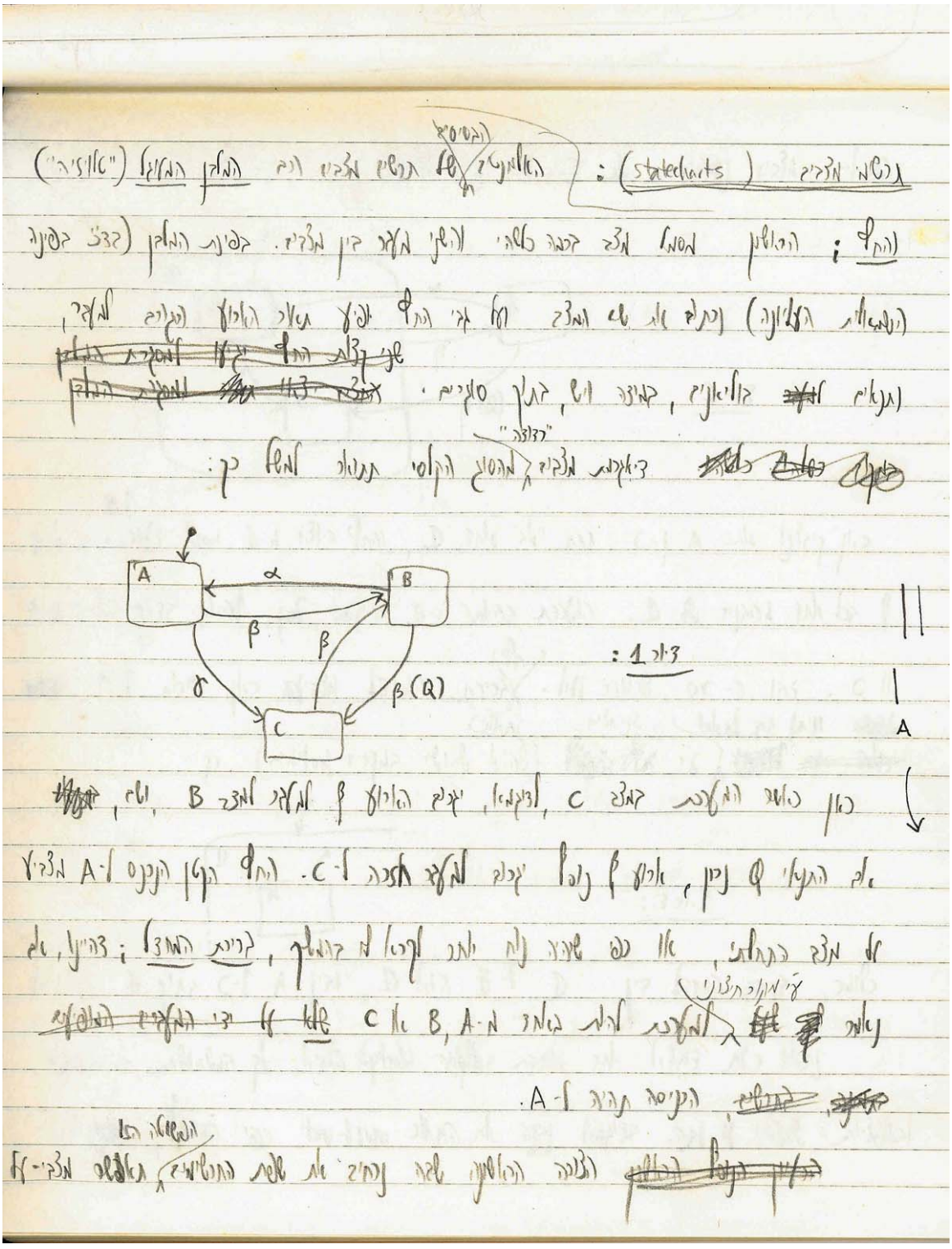
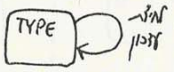


Figure 15. Page from the IAI notes (late 1983; in Hebrew) showing part of the draft of the internal IAI document reporting on the results of the consulting. Note the first use of the term *statecharts* (top line).

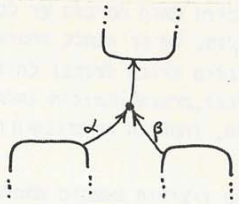
הסמון המשובה של כניסה עצמית, כאילו, ל-H בציור 11 מורה על העובדה שהעדכון משפיע אומנם בכל זאת מצב של UPDATE אך אינו מתואר כאן ע"י מעבר למצב חדש. כמובן, הכוונה היא שארוע העדכון מוציא את המערכת מתת מצב כלשהו ומחזירה מידית שוב ל-UPDATE ע"פ ההיסטוריה, דהיינו, שוב לאותו תת מצב שבו היתה לפני העדכון. למותר לציין, ניתן היה לתאר ענין זה ע"י ארבע חצים מהסוג המופיע בציור 12.

ציור 12:



נרשה קיצורי דרך גרפיים שונים להקלה על העומס בתרשימים, כגון כתיבת שני ארועים מופרדים ע"י OR על חץ אחד במקום שני חיצים עם ארוע על כל אחד, או ניקוז שני חצים או יותר לאחד ע"י צמת ניקוז כמו בציור 13.

ציור 13:



האופציה הנוספת שעדין לא תארנו היא של הפיצול האורתוגונלי למרכיבים, המתאים ל-AND. הדוגמא (החלקית) שניתנת כאן בציור 14 מתארת מקצת מן הרמה העליונה ביותר של מערכת האוויוניקה הכללי. האוויוניקה לבניא, ובה רואים כמה קוארדינטות אורתוגונליות של מצב

ציור 14:

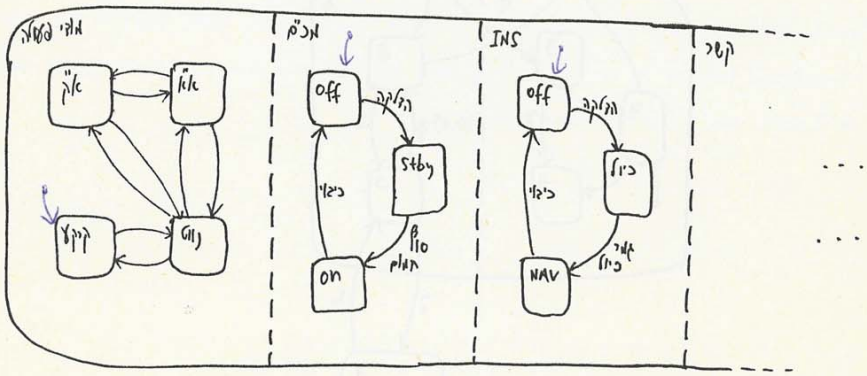


Figure 16. Page 10 of the internal IAI document (December 1983; in Hebrew). The bottom figure shows some of the high-level states of the Lavi avionics, including on the left (in Hebrew…) A/A, A/G, NAV and GRD.

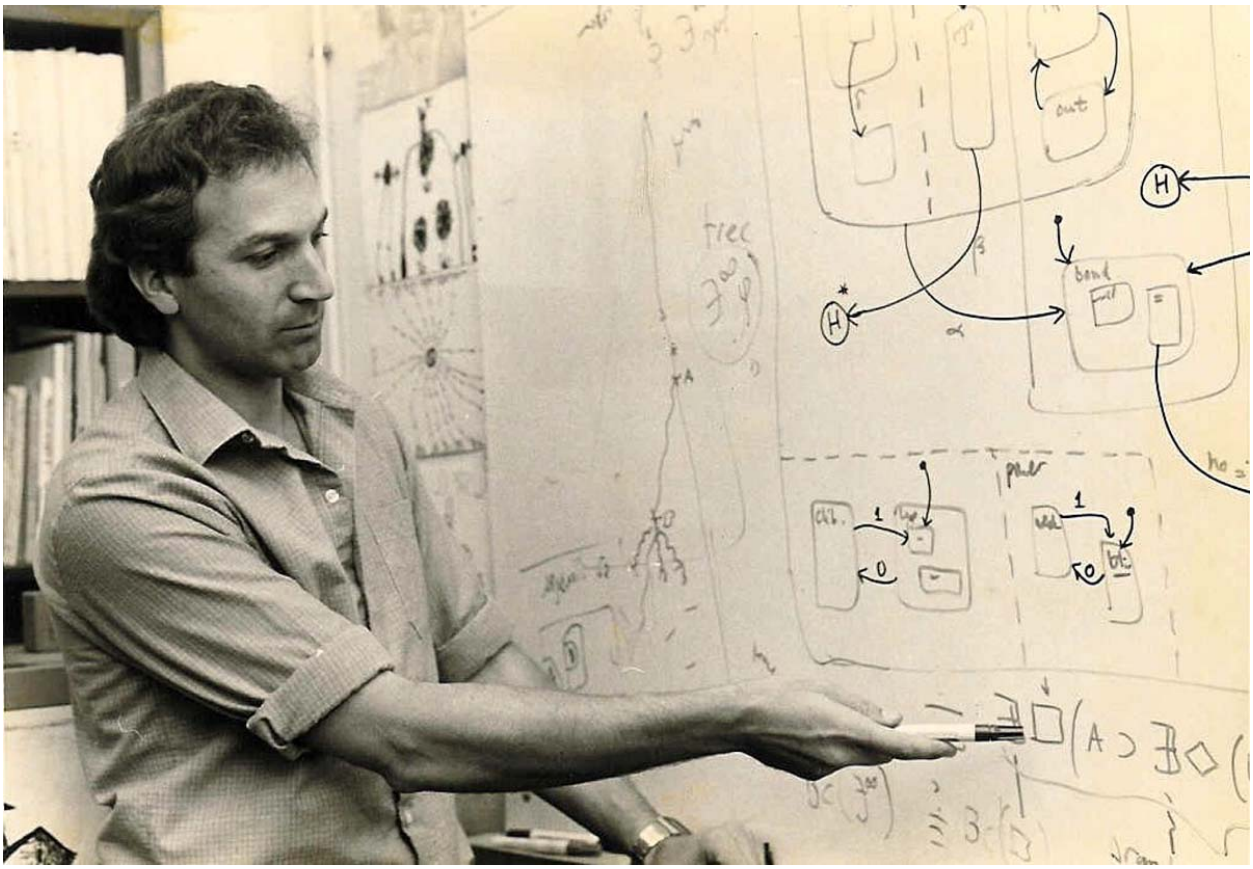
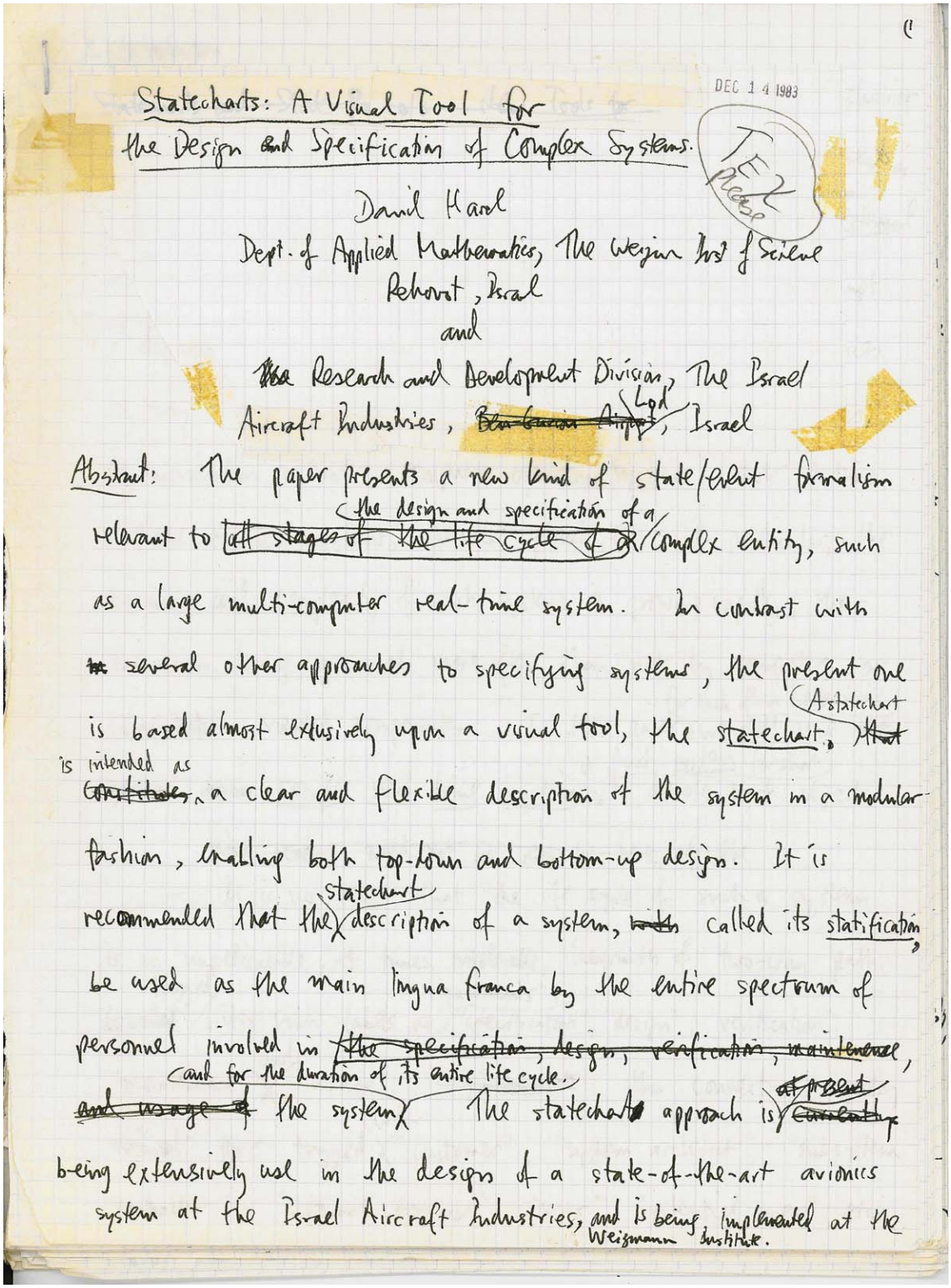


Figure 17. Explaining statecharts (early 1984). Note the temporal logic on the bottom right.



Statecharts: A Visual Tool for the Design and Specification of Complex Systems.

DEC 14 1983

TEX please

David Harel
Dept. of Applied Mathematics, The Weizmann Inst. of Science
Rehovot, Israel
and
Research and Development Division, The Israel Aircraft Industries, Lod, Israel

Abstract: The paper presents a new kind of state/event formalism relevant to the design and specification of a complex entity, such as a large multi-computer real-time system. In contrast with several other approaches to specifying systems, the present one is based almost exlusively upon a visual tool, the statechart. A statechart is intended as a clear and flexible description of the system in a modular fashion, enabling both top-down and bottom-up design. It is recommended that the statechart description of a system, called its statification, be used as the main lingua franca by the entire spectrum of personnel involved in the system, and for the duration of its entire life cycle. The statecharts approach is being extensively used in the design of a state-of-the-art avionics system at the Israel Aircraft Industries, and is being implemented at the Weizmann Institute.

Figure 18. Front page of the first draft of the basic statecharts paper (Dec. 14, 1983). Note the "TEX please" instruction to the typist/secretary; the original title (later changed twice), the use of the word "*stratification*" for the act of specifying with statecharts (later dropped), and the assertion that the language "is being implemented at the Weizmann Institute" (later changed).

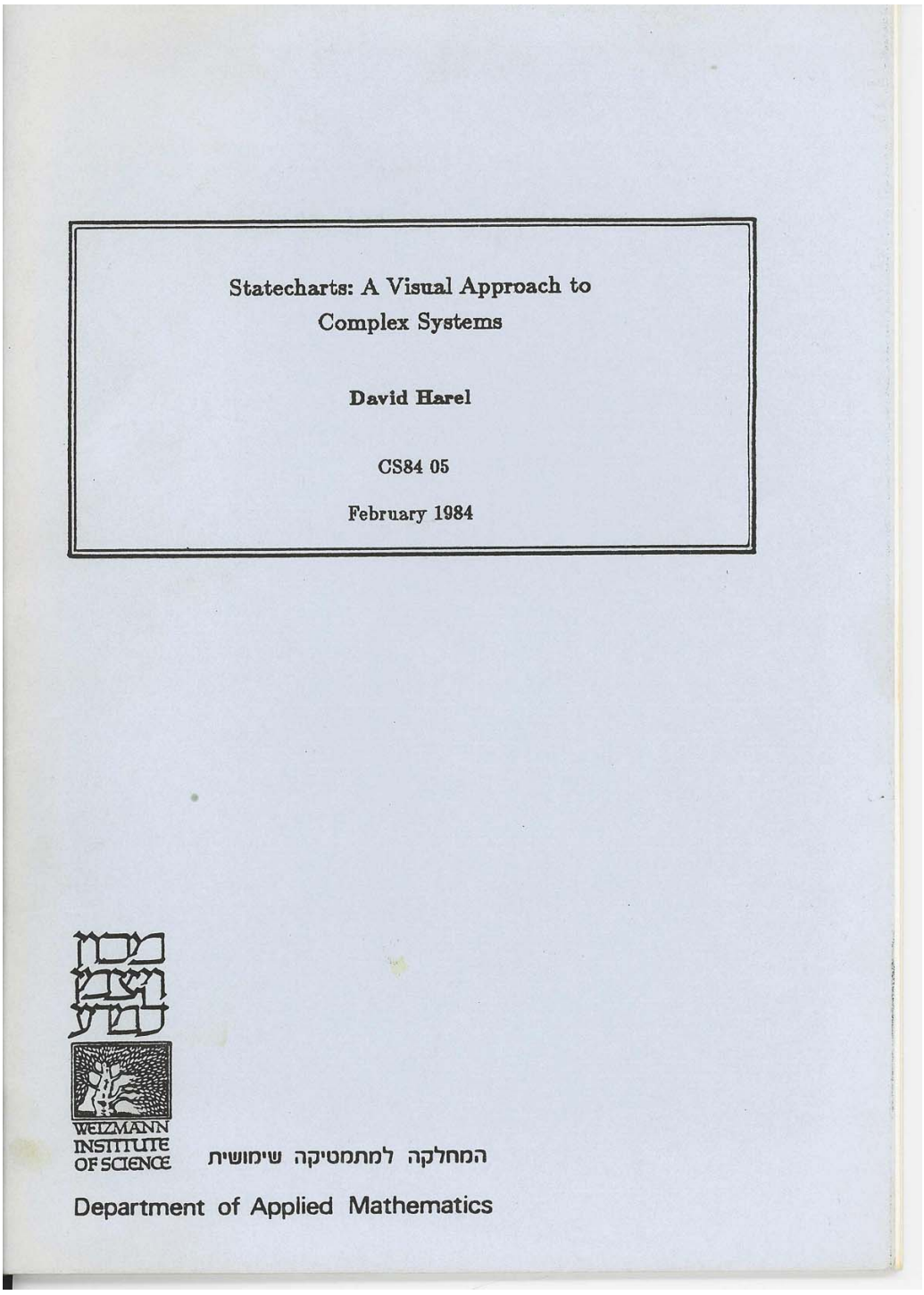


Figure 19. Front page of the technical report version of the basic statecharts paper (February 1984). Note the revised title (later changed again…).

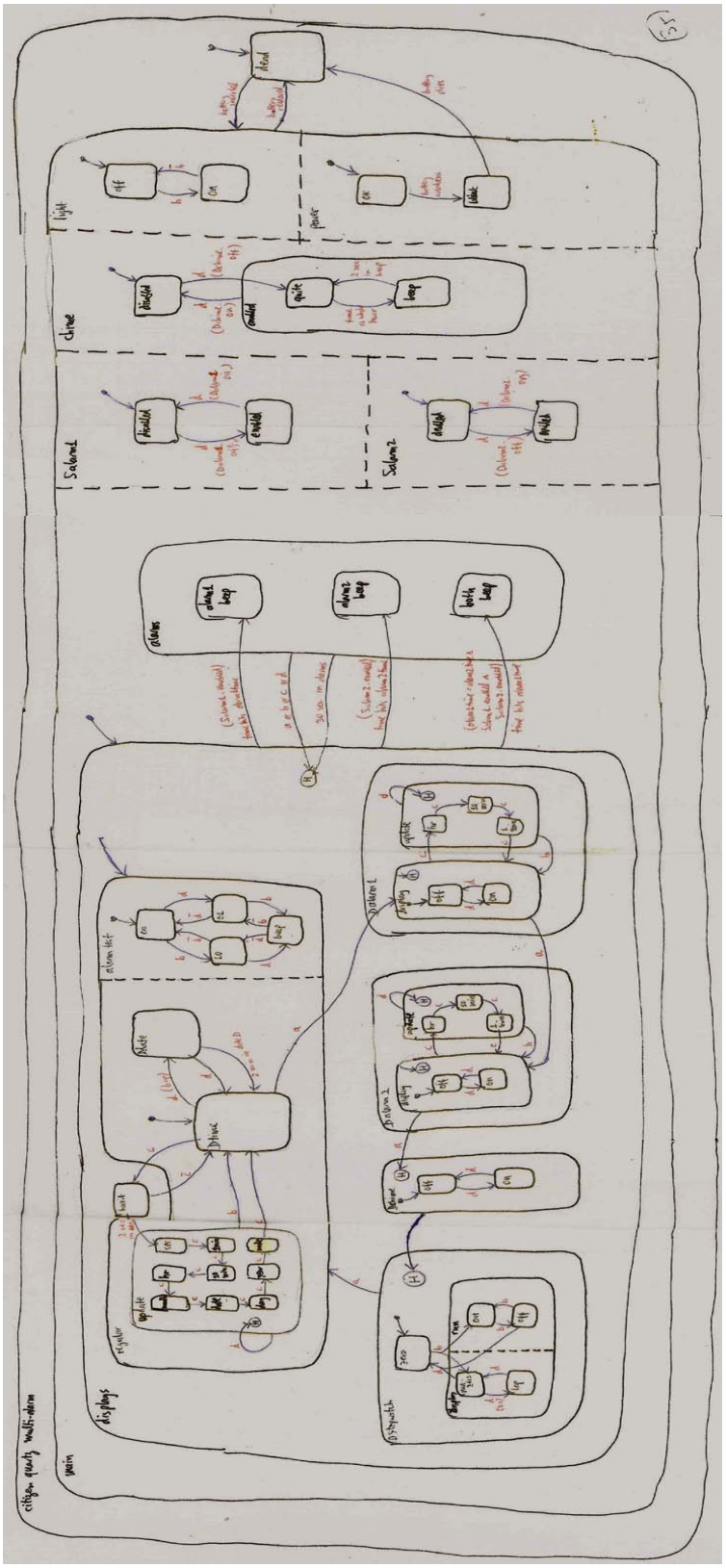
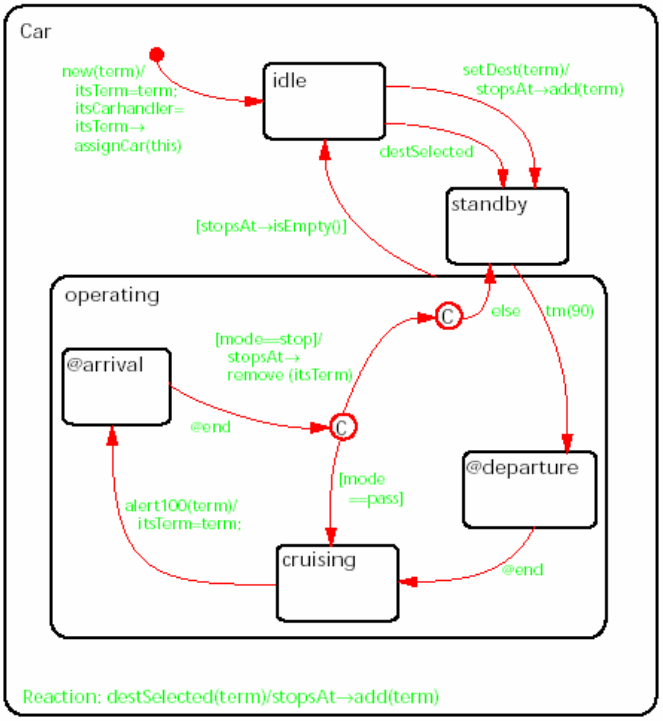


Figure 20. Hand-drawn draft of the Citizen watch statechart (late 1984), as sent to our graphics draftsman for professional drawing.



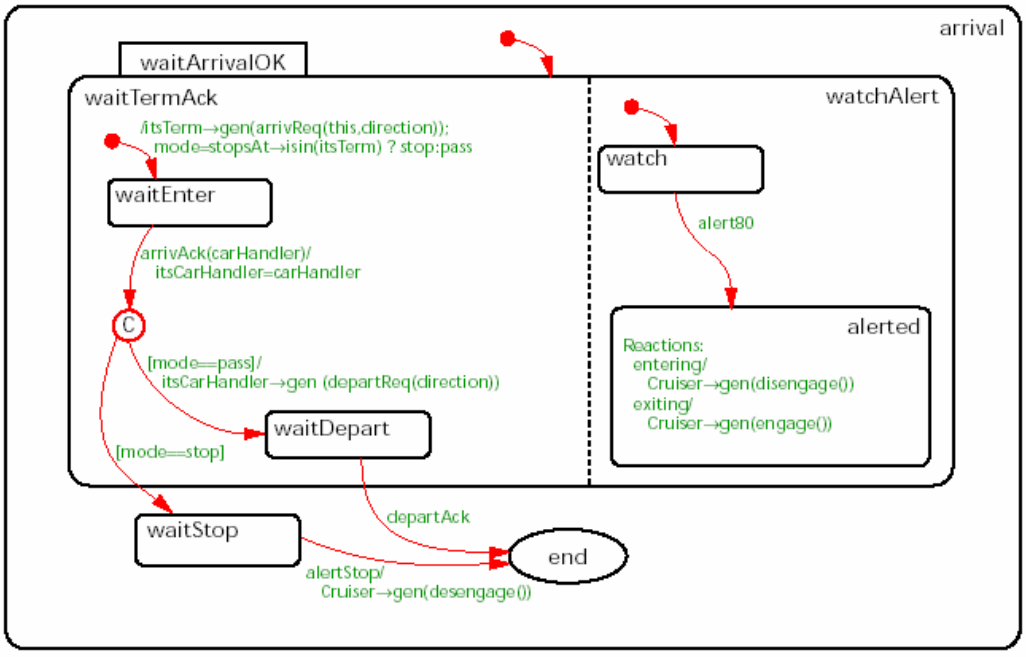


Figure 21. Two object-oriented statecharts for a railcar example, taken from [HG96&97]. Note the C++ action language.

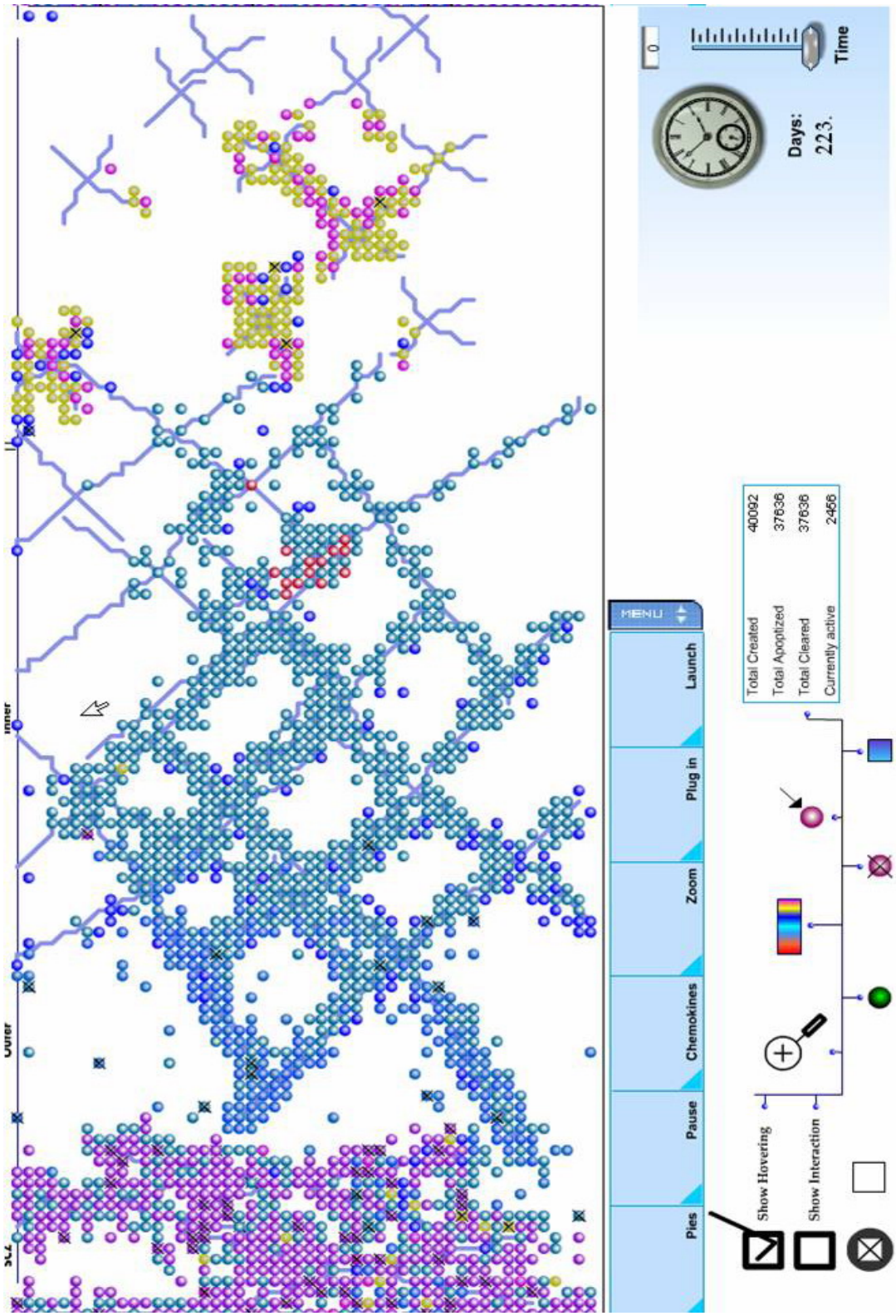


Figure 22. Front end (in Flash) of a model of T cell development in the thymus (from [EHC03]).

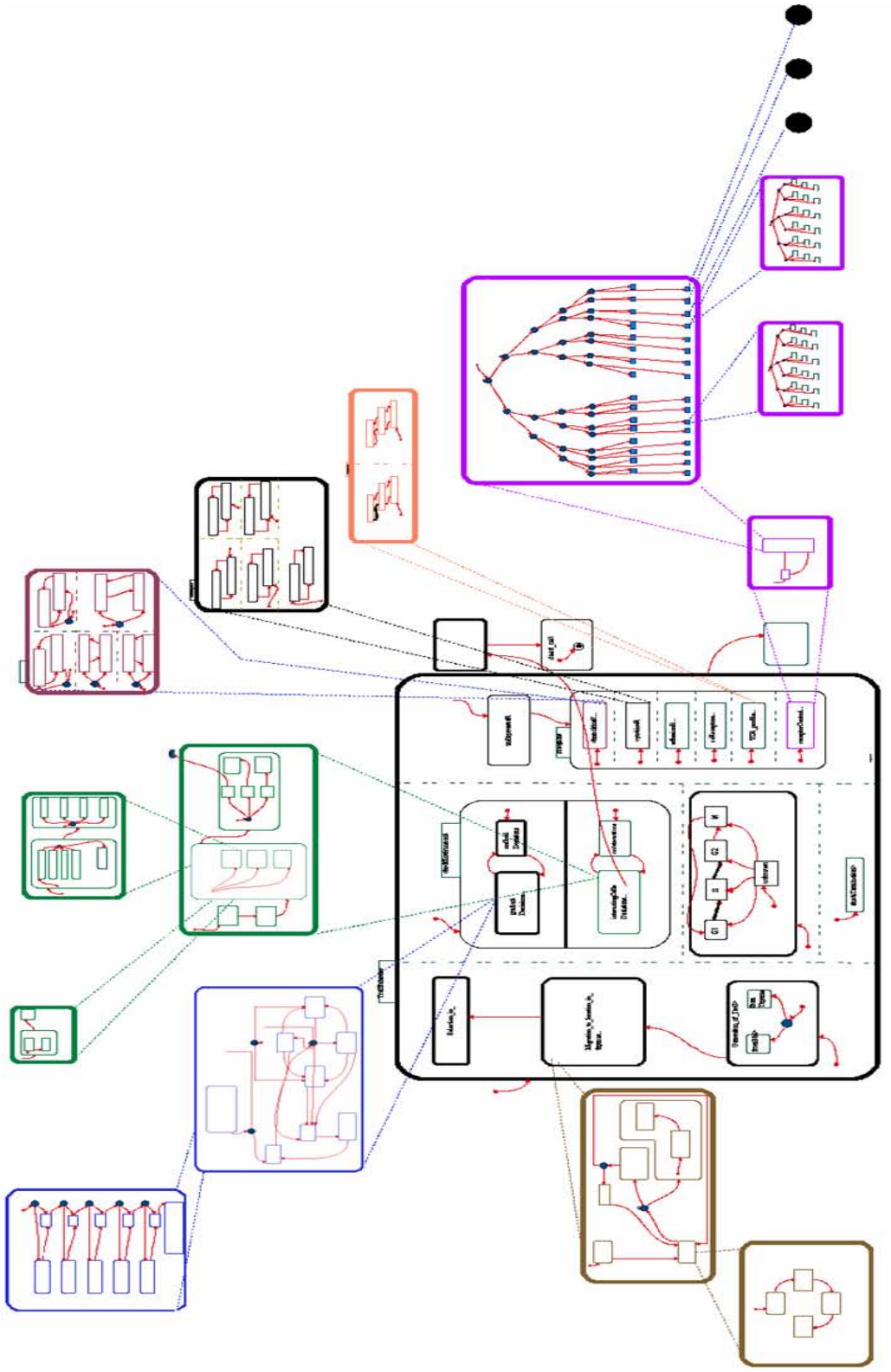


Figure 23. Schematics of parts of the statechart of a single cell from the T cell model shown in Fig. 21 (from [EHC03]).

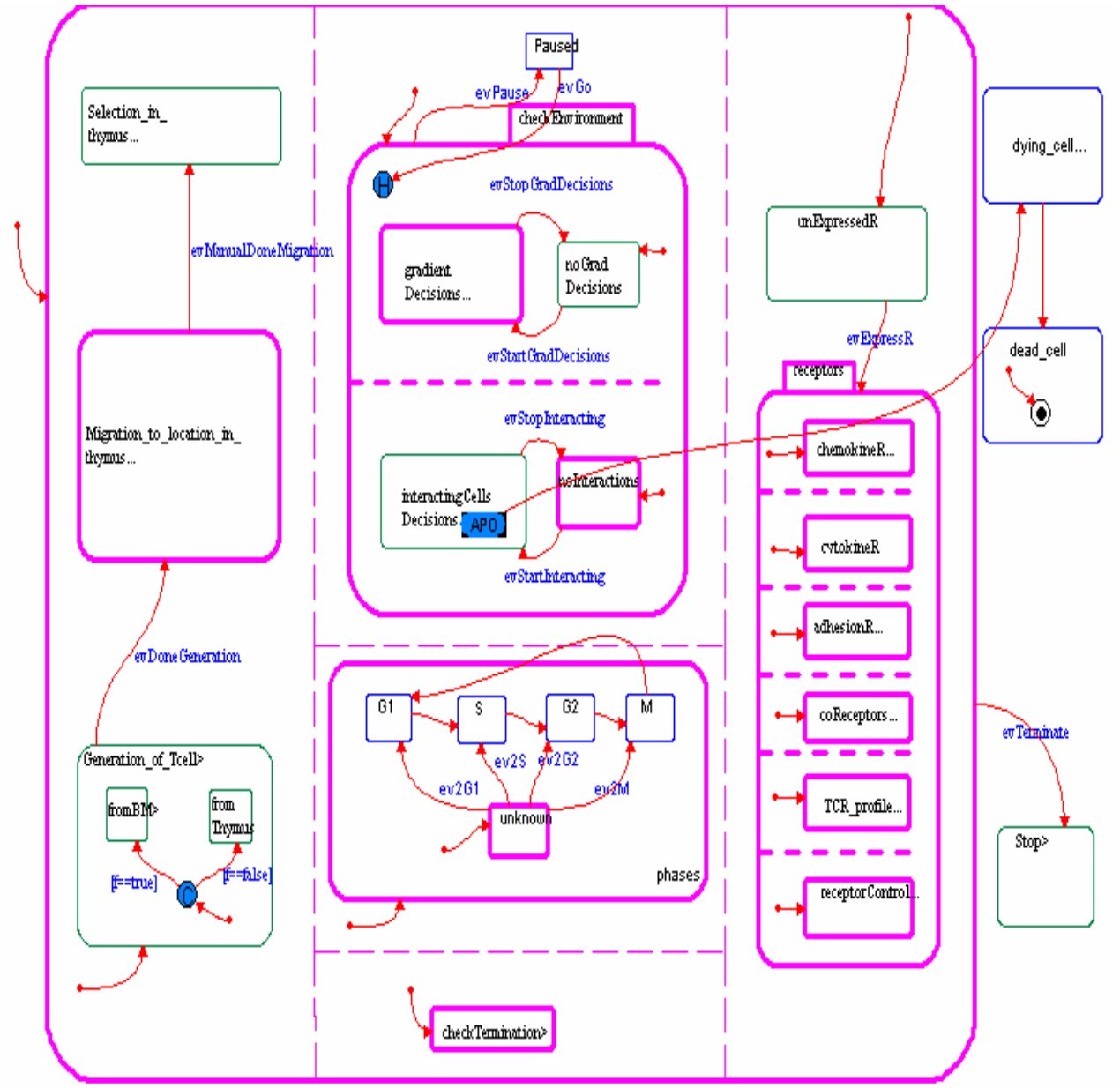


Figure 24. Snapshot of the first few levels of the statechart of a single cell from the T cell model of [EHC03], shown during execution on Rhapsody. The purple states (thick-lined, if you are viewing this in B&W) are the ones the system is in at the moment.

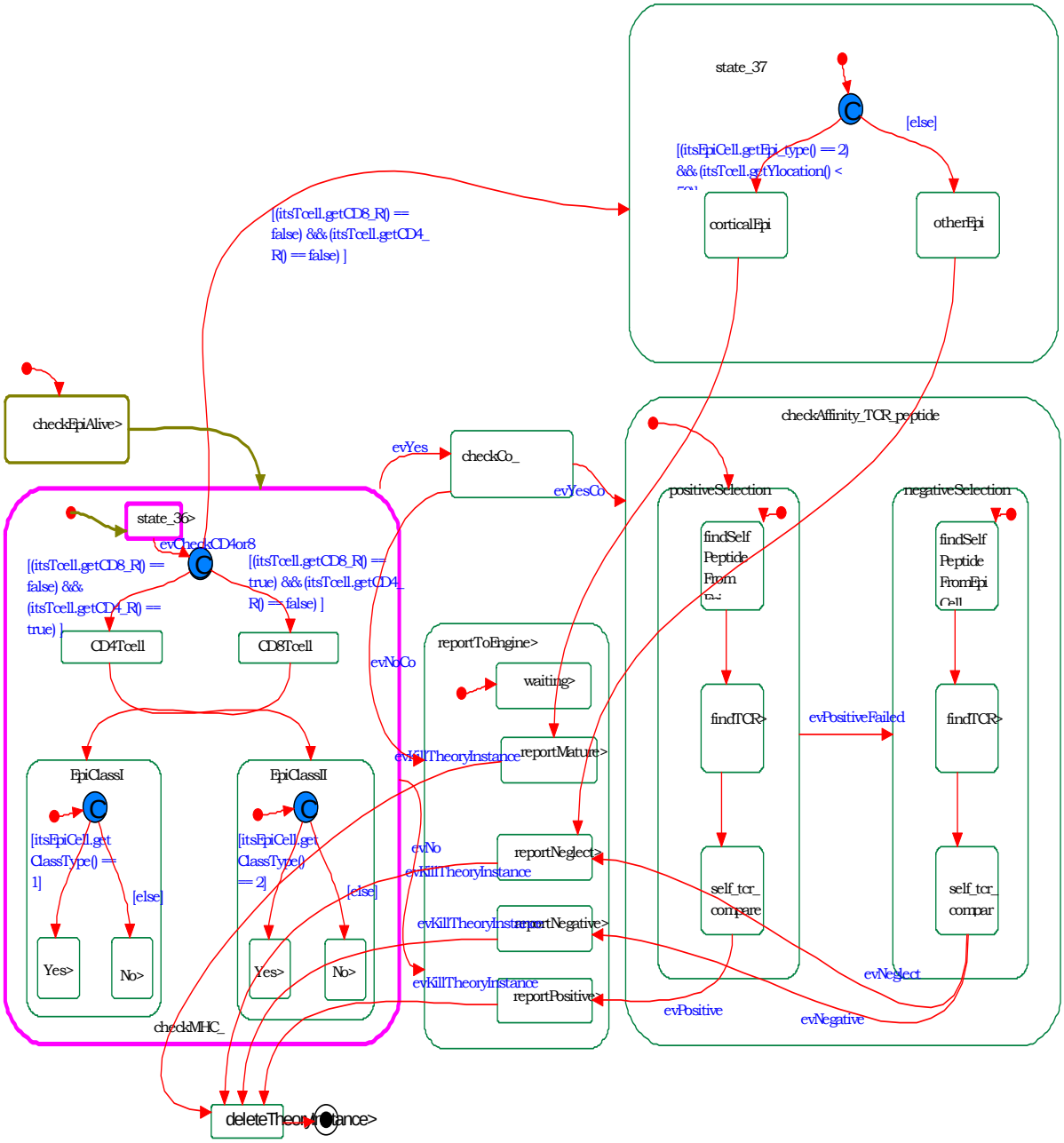


Figure 25. Snapshot of the statechart of the object dealing with the interaction between a potential T cell and an epithelial cell in the model of [EHC03], shown during execution on Rhapsody. The purple states (thick-lined, if you are viewing this in B&W) are the ones the system is in at the moment.

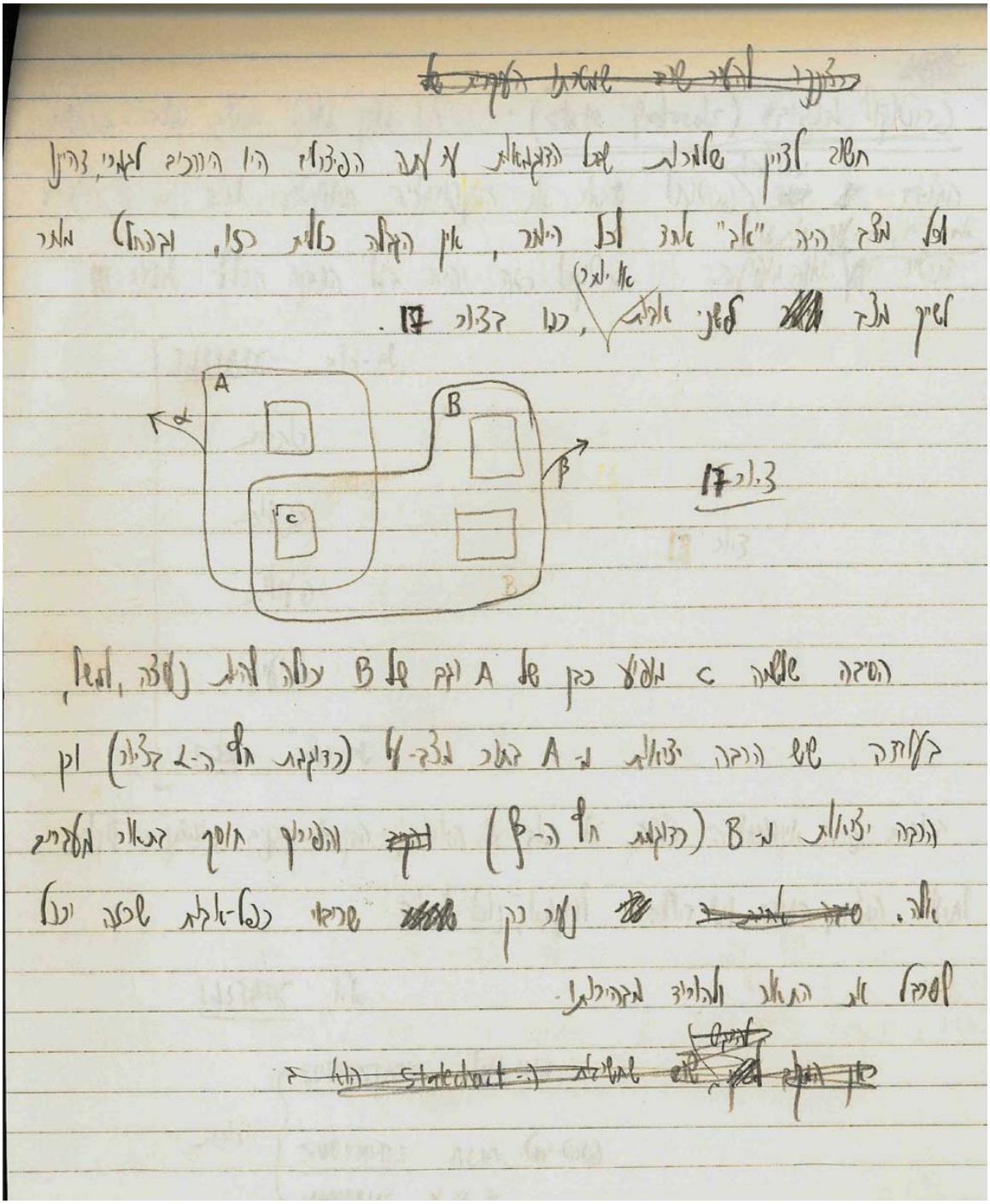


Figure 26. Page from the IAI notebook (late 1983) showing the use of overlapping states, which were never implemented.

References

[BHM04] D. Barak, D. Harel and R. Marelly, "InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming", *Lectures on Concurrency and Petri Nets* (J. Desel, W. Reisig and G. Rozenberg, eds.), Lecture Notes in Computer Science, Vol. 3098, Springer-Verlag, 2004, pp. 66–86.

[B+03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, "The Synchronous Languages Twelve Years Later", *Proc. of the IEEE* **91** (2003), 64–83.

[BG90] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the Signal Language", *IEEE Trans. on Automatic Control*, AC-**35** (1990), 535–546.

[BG92] G. Berry and G. Gonthier, "The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation," *Science of Computer Programming* **19**:2 (1992) 87–152.

[B94] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, California, 1994.

[B87] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* **20**:4 (1987) 10–19.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A Declarative Language for Programming Synchronous Systems", *Proc. 14th ACM Symp. on Principles of Programming Languages*, ACM Press, 1987, pp. 178–188.

[CKS81] A.K Chandra, D.C. Kozen and L.J. Stockmeyer, "Alternation", *Journal of the ACM* **28**:1 (1981), 114–133.

[DH99&01] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design* **19**:1 (2001), 45–80. (Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (FMOODS '99) (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.)

[DH88] D. Drusinsky and D. Harel, "On the Power of Cooperative Concurrency", *Proc. Concurrency `88*, Lecture Notes in Computer Science, Vol. 335, Springer-Verlag, New York, 1988, pp. 74–103.

[DH94] D. Drusinsky and D. Harel, "On the Power of Bounded Concurrency I: Finite Automata", *J. Assoc. Comput. Mach.* **41** (1994), 517–539.

[EHC03] S. Efroni, D. Harel and I.R. Cohen, "Towards Rigorous Comprehension of Biological Complexity: Modeling, Execution and Visualization of Thymic T Cell Maturation", *Genome Research* **13** (2003), 2485–2497.

[EHC05] S. Efroni, D. Harel and I.R. Cohen, "Reactive Animation: Realistic Modeling of Complex Dynamic Systems", *IEEE Computer* **38**:1 (2005), 38–47.

[EHC07] S. Efroni, D. Harel and I.R. Cohen, "Emergent Dynamics of Thymocyte Development and Lineage Determination", *PLOS Computational Biology* **3**:1 (2007), 127-136.

[HZ76] M. Hamilton and S. Zeldin, "Higher Order Software: A Methodology for Defining Software", *IEEE Transactions on Software Engineering* **SE-2**:1 (1976), 9–36.

[H79&80] D. Harel, "And/Or Programs: A New Approach to Structured Programming", *ACM Trans. on Programming Languages and Systems* **2** (1980), 1–17. (Also *Proc. IEEE Specifications for Reliable Software Conf.*, pp. 80–90, Cambridge, Massachusetts, 1979.)

[H84&87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* **8** (1987), 231–274. (Preliminary version: Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)

[H88] D. Harel, "On Visual Formalisms", *Comm. Assoc. Comput. Mach.* **31**:5 (1988), 514–530. (Reprinted in *Diagrammatic Reasoning* (Chandrasekaran et al., eds.), AAAI Press and MIT Press, 1995, pp. 235–271, and in *High Integrity System Specification and Design* (Bowen and Hinchey, eds.), Springer-Verlag, London, 1999, pp. 623–657.)

[H92] D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development", *IEEE Computer* **25**:1 (1992), 8–20.

[HG96&97] D. Harel and E. Gery, "Executable Object Modeling with Statecharts", *IEEE Computer* **30**:7 (1997), 31–42. (Also in *Proc. 18th Int. Conf. on Software Engineering*, IEEE Press, 1996, pp. 246–257.)

[HK92] D. Harel and H.-A. Kahana, "On Statecharts with Overlapping", *ACM Trans. on Software Engineering Method.* **1**:4 (1992), 399–421.

[HK04] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)", *Integration of Software Specification Techniques for Applications in Engineering*, (H. Ehrig et al., eds.), Lecture Notes in Computer Science, Vol. 3147, Springer-Verlag, 2004, pp. 325–354.

[HKV02] D. Harel, O. Kupferman and M.Y. Vardi, "On the Complexity of Verifying Concurrent Transition Systems", *Information and Computation* **173** (2002), 143–161l.

[H+88&90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Trans. on Software Engineering* **16**:4 (1990), 403–414. (Early version in *Proc. 10th Int. Conf. on Software Engineering*, Singapore, April 1988, pp. 396–406. Reprinted in *Software State-of-the-Art: Selected Papers* (DeMarco and Lister, eds.), Dorset House Publishing, New York, 1990, pp. 322–338, and in *Readings in*

Hardware/Software Co-design (De Micheli, Ernst and Wolf, eds.), Morgan Kaufmann, 2001, pp. 135–146.)

[HM03] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.

[HN89&96] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Trans. on Software Engineering Method*. **5**:4 (1996), 293–333. (Preliminary version appeared as Technical Report, I-Logix, Inc., 1989.)

[HP85] D. Harel and A. Pnueli, "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, pp. 477–498.

[HPSR87] D. Harel, A. Pnueli, J. Schmidt and R. Sherman, "On the Formal Semantics of Statecharts", *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, NY, 1987, pp. 54–64.

[HP91] D. Harel and M. Politi, *The Languages of STATEMATE*, Technical Report, I-Logix, Inc., Andover, MA (250 pp.), 1991.

[HP96] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998. (This book is no longer in print, but it can be downloaded from my web page.)

[HR04] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of 'Semantics'?", *IEEE Computer* **37**:10 (2004), 64–72.

[HP87] D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1987.

[HKSP78] K.L. Heninger, J.W. Kallander, J.E. Shore and D.L. Parnas, "Software Requirements for the A-7E Aircraft", *NRL Report 3876*, November 1978, 523 pgs.

[HU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[HGdR88] C. Huizing, R. Gerth and W.P. de Roever, "Modeling Statecharts Behaviour in a Fully Abstract Way", *Proc. 13th Colloquium on Trees in Algebra and Programming* (CAAP '88), Lecture Notes in Computer Science, Vol. 299, Springer-Verlag, 2004, pp. 271–294.

[I-L] I-Logix, Inc. Products Web page, http://www.ilogix.com.

[KCH01] N. Kam, I.R. Cohen and D. Harel, "The Immune System as a Reactive System: Modeling T Cell Activation with Statecharts", *Bull. Math. Bio.*, to appear. (Extended abstract in *Proc. Visual Languages and Formal Methods* (VLFM'01), part of *IEEE Symp. on Human-Centric Computing* (HCC'01), 2001, pp. 15–22.)

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process-Control Systems", *IEEE Transactions on Software Engineering* **20**:9 (1994), 684–707.

[MM85] J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, 1985.

[MS88] K. Misue and K. Sugiyama, "Compound graphs as abstraction of card systems and their hierarchical drawing," *Inform. Processing Soc.*, Japan, Research Report 88-GC-32-2, 1988, (in Japanese).

[PS91] A. Pnueli and M. Shalev, "What Is in a Step: On the Semantics of Statecharts", *Proc. Symp. on Theoret. Aspects of Computer Software*, Lecture Notes in Computer Science, Vol. 526, Springer-Verlag, 1991, pp. 244–264.

[R85] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice-Hall, New York, 1991.

[SeCH06] Y. Setty, I.R. Cohen and D. Harel, in preparation, 2006.

[SM91] K. Sugiyama and K. Misue, "Visualization of Structural Information: Automatic Drawing of Compound Digraphs", *IEEE Trans. Systems, Man and Cybernetics* **21**:4 (1991), 876–892.

[SwCH06] N. Swerdlin, I.R. Cohen and D. Harel, "Towards an *in-silico* Lymph Node: A Realistic Approach to Modeling Dynamic Behavior of Lymphocytes", submitted, 2006.

[SGW94] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

[UML] Documentation of the Unified Modeling Language (UML), available from the Object Management Group (OMG), http://www.omg.org.

[vB94] M. von der Beeck, "A Comparison of Statecharts Variants", *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, Lecture Notes in Computer Science, Vol. 863, Springer-Verlag, 1994, pp. 128-148.

[WM85] P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, vols. 1–3, Yourdon Press, 1985.