Biting the Silver Bullet

Toward a Brighter Future for System Development

David Harel, Weizmann Institute of Science

A "vanilla" approach to modeling, together with powerful notions of executability and code generation, may have a profound impact on the "essence" of developing complex systems.

In an eloquent and thoughtful 1986 article, Frederick Brooks expresses his feelings about the illusions and hopes software engineering offers.[1] He argues that many proposed ideas are not "silver bullets" that will deliver us from the horrors of developing complex systems.

Brooks' article is reminiscent of Parnas' series of minipapers[2] that accompanied his widely publicized resignation from the Strategic Defense Initiative Organization (SDIO) Panel on Computing in 1985. Parnas claims that current proposals are vastly inadequate to build reliable software as complex as that required for the SDI project.

We thus have two rather discouraging position papers, authored by two of the most influential figures in the software world. Neither is a critique of software engineering per se, although both make an effort to dissolve myths of magical power that people have cultivated concerning certain trends in the field.

This article was triggered by those of Brooks and Parnas. It is not a rebuttal. Indeed, I agree with most of the specific points made in both papers. Instead, the goal of this article is to illuminate the brighter side of the coin, emphasizing developments in the field that were too recent or immature to have influenced Brooks and Parnas when they wrote their manuscripts.

The two main aspects of these developments have to do with a carefully wrought "vanilla" approach to system modeling and the emergence of powerful methods to execute and analyze the resulting models. It can be argued that the combined effect of these and other ideas is already showing positive signs and appears to have the potential to provide a truly major improvement in our present abilities — profoundly affecting the essence of the problem. This might take more than the 10 years Brooks focuses on. It will surely be a long time before reliable software for the likes of the SDI project can be built. Such a system remains an order of magnitude too large and too critical to construct today, mainly because of its first-time-must-work nature. But I also believe that we are on the royal (main) road and that the general impression you get from reading the Brooks and Parnas articles is far too bleak.

0018-9162/92/0100-008$03.00 © 1992 IEEE

Past versus present

Brooks' arguments. The main problem, as Brooks rightly sees it, is in specifying, designing, and testing the "conceptual construct" underlying the system being developed, and not in "the labor of representing it and testing the fidelity of the representation."

"The hard thing about building software," he claims, "is deciding what one wants to say, not saying it." In elaborating, he mentions the superlinear growth in the number of system states, the difficulty of comprehending the conceptual construct and communicating it to others, and what he believes to be its inherent unvisualizable character.

Brooks further argues that, in contrast to their apparent appeal, several proposed ideas in the field do not constitute magical solutions to the essential problems. Among the "nonbullets" he discusses are high-level languages, object-oriented programming, artificial intelligence and expert systems, automatic programming, graphical languages, program verification, and hardware improvements.

In his introduction, Brooks says that although he sees no startling breakthroughs in the next decade, "many encouraging innovations are under way," and eventually they will be exploited to "yield an order-of-magnitude improvement."

Brooks mentions two sets of innovations. The first set includes those of the above proposals that he doesn't totally discard (for example, high-level languages and object-oriented programming). However, he claims that they deal only with representation issues, which constitute the accidental part of the problem.

The second set of innovations, the ones Brooks claims *will* influence the essence, include

- buying sufficiently general ready-made software, instead of having it tailor-made;
- refining the requirements iteratively and interactively with the client, using increasingly better prototypes;
- enhancing the design in an iterative, top-down fashion, adding lower-level details at each step; and
- finding, hiring, and cultivating extremely talented designers.

Despite the encouraging way the points are expressed, we come away feeling distinctly uncomfortable. Apart from ideas that deal with the accidental parts of the problem, we are told to buy good software from others, hire better people than we already have, and continue with the well-established practices of prototyping and iterative design. All the rest is marginal.

I have discussed Brooks' article with many people over the past few years. Most stated that while they agree with many of its individual points, the paper presents a far gloomier assessment of the situation than seems appropriate. I feel that this is rooted in some of its underlying adopted themes.

The first is the sharp separation between the accidental and essential aspects of the problem, relegating everything related to representation, language, and levels of abstraction to the former and only the process of thinking about the concepts to the latter.

The second is the treatment of each proposed idea in isolation, with the accompanying claim that most of the proposals address representation, so that they cannot help with the essence.

The third involves concentrating on only 10 years of the future, which is probably too short a period in which to expect any significant improvement. (About half of this period is already behind us.)

Finally, the discussion is presented as a search for a miracle-working silver bullet that will slay the werewolf of constructing complex software. By arguing that current motifs will not bring about that miracle, at least not within the next few years, we are left with the troubling feeling that the werewolf is here to stay.

We've been there before. Since this article takes a longer term point of view, it is instructive to carry out a brief thought experiment. Let's go back, say 40 years. That was the time when instead of grappling with the design of large, complex systems, programmers were in the business of developing conventional one-person programs (which would be on the order of 100-200 lines in a modern programming language) that were to carry out limited algorithmic tasks. Given the technology and methodology available then, such tasks were similarly formidable. Failures, errors, and missed deadlines were all around.

Imagine an article appearing then and claiming the essence of the problem to be deciding what one wants to say, that is, conceiving the algorithm. Writing the program is the accidental part. Such an article might have asked about the availability of a one-stroke solution that deals with the essence. From the way the issue is presented, it would follow that any ideas that relate to representation and levels of detail can be discounted, because they deal with the nonessential parts of the problem. The article could very well go on to argue that ideas like high-level programming languages, compilation, and algorithmic paradigms can be safely set aside, since they do not deal with the essence.

However, while none of these ideas alone has solved the problem, and while it did take more than 10 years for the situation to change, we have indeed witnessed an order-of-magnitude advance in our ability to tackle the very

On biting bullets

There are two opinions about the origin of the phrase "Biting the bullet." One is that it came from the need to bite the top off the paper cartridge prior to firing a certain kind of British rifle used in the mid 19th century. This often had to be done under enemy fire and required keeping a cool head.

The other is that it is an old American phrase, rooted in the folklore of the US Civil War. It supposedly emerged from the practice of encouraging a patient who was to undergo field surgery to bite down hard on a lead bullet to "divert the mind from pain and prevent screaming" (R.L. Chapman, *American Slang*, Harper and Row, New York, 1986).

In more recent years, the phrase has come to signify having to do something painful but necessary, or to undertake an activity despite criticism or opposition, while exhibiting a measure of courage and optimism.

More on the vanilla approach

It is impossible to provide a detailed account of the vanilla approach to modeling in this article. The discussion of it in the text is thus extremely brief. As mentioned, the ideas are based on much early work on the specification and design of nonreactive systems, suitably extended.

The three independent efforts that led to this approach are described, respectively, in the Ward and Mellor book,[3] the Hatley and Pirbhai book,[4] and in the Harel et al.[5] publication related to the Statemate system.

The latter is less informative on the modeling aspects of the approach than the two books; its main intention was to describe the supporting tool. However, a more detailed description of this modeling framework appears in the following manuscript, which should appear in book form in due time:

Harel, D., and M. Politi, "The Languages of Statemate," Tech. Report, i-Logix, Burlington, Mass., 1991.

The following paper compares and evaluates these three research efforts (as well as a related fourth one). It is quite illuminating and emphasizes the differences between them, particularly those relevant to modeling behavior:

Wood, D.P., and W.G. Wood, "Comparative Evaluations of Four Specification Methods for Real-Time Systems," Tech. Report CMU/SEI-89-TR-36, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1989.

The following book contains interesting discussions and comparisons of these and other modeling approaches. It also features a valuable annotated bibliography of some 600 items:

Davis, A. M., *Software Requirements: Analysis and Specification*, Prentice Hall, Englewood Cliffs, N.J., 1990.

essence of designing one-person programs. There is absolutely no comparison between the process of writing a correct and efficient one-person program 40 years ago and now. (Actually, we need not come all the way to 1992; it suffices to compare 1950 with, say, 1975.) The grand sum of the many innovations that have been suggested and pursued in the interim has worked wonders!

Of course, the situation isn't perfect. There still is a great deal of bad programming around, and there are lots of incompetent programmers, some terrible programming languages, many misleading methods and guidelines, and widespread ignorance of the fundamentals. Nevertheless, most people would agree that the werewolves of one-person programs are gone, never to return.

Vanilla frameworks. Most instrumental in triggering the revolution in one-person programming has been the evolution of a fitting, general-purpose conceptual framework, which we shall call "vanilla." Its main contribution was to free the programmer from having to think on an inappropriate level of detail, enabling him or her to conceive of an idea for solving an algorithmic problem and to map it easily from the mind into an appropriate high-level medium.

The cornerstone of this framework is a collection of fundamental notions and concepts that includes the basic dichotomy between data and control and convenient means for structuring and combining them into an algorithmic whole. Thus, elementary control structures, data types, and data structures were identified, and we learned how to wield them. A rich variety of algorithmic methods was devised, including divide and conquer, dynamic programming, and greedy paradigms; these were adapted to fit a variety of problem sets. Notions of correctness and efficiency were introduced, together with methods for establishing the former and estimating the latter.

In parallel, and based on these concepts, a corresponding set of vanilla high-level programming languages evolved, supported by powerful and sophisticated tools for testing and analyzing. We learned to rely on the theory of computational complexity to help us find efficient algorithms or to detect our stumbling upon an intractable problem; we have begun to understand the great virtues of parallelism, approximation, and randomization in obtaining even better solutions.

Thus, for one-person programs, accidental and essential issues were intimately and unavoidably intertwined.

Of course, as time went by, other flavors, more exotic than vanilla, naturally emerged, such as applicative, functional, and logic programming styles, as well as more esoteric approaches like systolic arrays and neural nets. For each, the basic notions and concepts have had to be redefined, and new languages and tools have been developed. The arsenal has thus grown considerably and has become richer and more varied — a sure sign of healthy evolution.

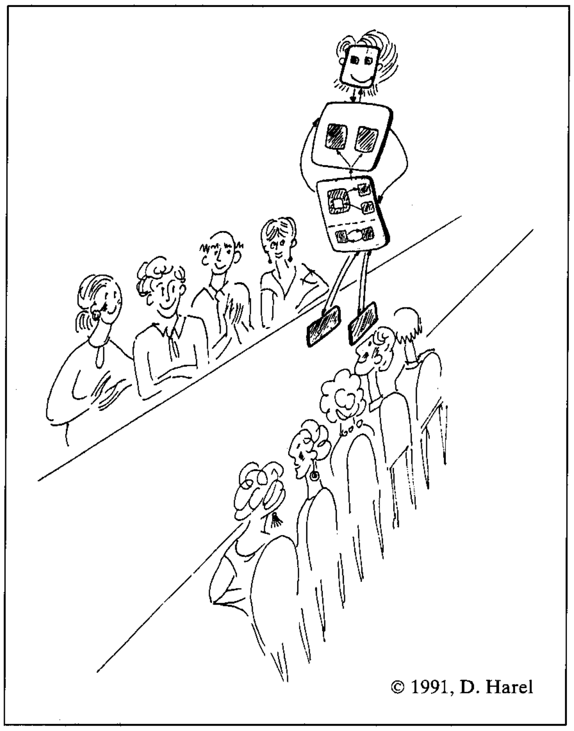
Back to the future. I believe the current situation is similar, except that we are now in the business of developing very complex systems. These systems are to consist of large amounts of software and hardware and are often of a distributed nature. Their size and complexity, as Brooks and Parnas observe, is formidable when compared to one-person programs. By their very nature, they also involve large numbers of technical personnel.

The rest of this article is restricted to a class of systems that has been termed *reactive*.[5,6] This class includes many kinds of embedded, concurrent, and real-time systems, but excludes data-intensive ones such as databases and management information systems. Reactive systems are widely considered to be particularly problematic, posing some of the greatest challenges in this field.

Building on a solid foundation of time-honored work in software and systems engineering, a number of developments have taken place in the past several years. Although not yet universally accepted as such, I submit that they combine to form the kernel of a solid general-purpose vanilla approach (see the sidebar) to the development of complex reactive systems. Moreover, encouraging research is under way in a number of related fields that has fundamental implications regarding these ideas.

The climate suggests that we stand to witness a grand-scale improvement in the process of constructing such systems. It is hard to predict a time frame for this, but the scope of the benefits it will bring about could very well match the striking changes we have witnessed in solving one-person algorithmic problems.

We now discuss the two components of these developments: means for modeling the system, and techniques for inspecting and analyzing the model.



A system model.

Modeling the system

To model systems, we need an underlying set of fundamental concepts and notions — some call them "abstractions" — that, in Brooks' terminology, capture the "conceptual construct" of complex systems. Deciding what they are and how they relate is analogous to the separation of data and control in the vanilla approach to one-person programs and the identification of appropriate ways of structuring, expressing, and combining them. For a nonexotic first cut at the problem, these concepts must be sufficiently general to be widely applicable, even at the expense of being somewhat mediocre. To be amenable to inspection and analysis, they must also be rigorous and precise, with underlying formal semantics.

The vanilla approach is rooted in the early work of Parnas and others on modularization and information hiding,[7] and in that of several researchers on structured analysis and structured design[8,9] that dealt mainly with data-intensive systems. The backbone of the system model should be a hierarchy of *activities*, as we'll call them, that capture the functional capabilities of the system — suitably decomposed to a level with which the designer is happy. (The activities need not be arranged in strict hierarchies. The breakup, or decomposition, may have overlappings, with elements on any level being shared by multiple parent elements. The term "hierarchy" used here thus carries a more flexible connotation.)

Data elements and data stores are also specified therein, and are associated as inputs and outputs that flow between the activities on the various levels. The semantics of this kind of functional description is dynamically noncommitting in that it merely asserts that activities *can* be active, information *can* flow, and so on. It does not contain information about what *will* happen, *when* it will happen, or *why* it will happen. As a consequence, this hierarchy can only serve as *part* of a conceptual model for truly reactive systems — such as control and communication systems or embedded real-time systems, which have a crucial behavioral side that has to be addressed, too.

Some time ago, a number of independent research groups extended these widely accepted ideas to deal with reactive systems.[3-5] Their efforts resulted in a surprisingly similar set of conclusions. Using their own terminology and emphasis, they each recommended that the hierarchy of activities be enriched with behavioral descriptions we'll call *control activities*, which potentially appear on all levels.

Control activities serve as the central nervous system, so to speak, of the model. They are meant to sense and control the dynamics of that portion of the functional description on their level. This includes the ability to activate and deactivate activities, cause data to be read and written, and sense when such things have happened — thus affecting subsequent behavior. The resulting combination is the system's *conceptual model.*

The recommendations also call for a structural, or architectural, description of the system to deal with such notions as subsystems and modules, channels and physical links, and storage components. This description can thus be considered the system's *physical model.* The conceptual and physical models are related by a mapping that assigns implementational responsibility for the various parts of the former to those of the latter.

Modeling behavior. While the functional description is the backbone of the conceptual model, the behavioral descriptions (that is, the control activities) are, in a crucial sense, its heart and soul. Behavior over time is much less tangible than either functionality or physical structure, and more than anything else, this is the aspect that renders reactive systems so slippery and error-prone.

In the realm of dynamic behavior, there is a particularly dire need for approaches that are sufficiently clear and well-structured to enable designers to capture their thinking in a coherent and comprehensive fashion. Moreover, behavioral descriptions must possess rigorous underlying semantics; all too often, insufficient attention has been paid to semantics. The discussion of analysis below will show how important this is.

The aforementioned research groups[3-5] more or less agree that behavioral controllers should be described using modes, or states, together with control elements, such as events and conditions, that trigger transitions between them. Implicitly, they have also adopted a subtle abstraction, termed the *synchrony hypothesis*,[10] according to which everything takes zero time unless explicitly prescribed otherwise. However, there is no agreement as to exactly how this is to be turned into a workable medium for modeling reactive behavior. It is clear that conventional finite-state machines will not do, due to their lack of structure, their verbosity, and the notorious state-explosion phenomenon.

The basic elements of reactive behavior (states, transitions, events, conditions, and time) must be allowed to be properly and naturally conceptualized, structured, and combined so that fundamental patterns of reactive behavior — like sequentiality, concurrency, and synchronization — will mesh smoothly with the functional decomposition.

A number of solutions have been suggested by these groups. They range from variants of communicating finite-state machines,[3,4] through combinational decision tables and other similar means,[4] to a relative newcomer — statecharts.[5,11] Other formalisms, such as Petri nets, temporal logic,[6] or certain languages especially tailored for real-time systems,[10] would be reasonable choices, too.

Modeling data. Although data-intensive systems are not the subject of this article, a few words regarding the issue of incorporating data modeling into the vanilla framework are in order.

Conventional data elements and data structures can be specified and manipulated in standard ways within behavioral descriptions or in bottom-level activities. To deal with large-scale pools of data, such as databases or knowledge-bases, we would have to use a separate data-modeling medium, such as a suitably adapted version of Chen's entity-relationship approach.[3,4,12] The resulting descriptions would then be associated with the data stores.

Incorporating data-modeling techniques into the present framework could serve as an excellent melting pot for combining ideas from the world of data-intensive systems with ones from the world of reactive systems.

Strata of conceptual models. Brooks states that descriptions of software that abstract away its complexity often also abstract away its essence — the complexity itself being part of the essence. Obviously, he is right. Indeed, it is important to use the vanilla approach in a way that does not hide the system's essential complexity. Proper use actually enables harnessing and taming that complexity by allowing the designer to capture the system's inherent conceptual structure in a natural way.

Regardless of how well devised it might be, one conceptual model might not be enough to take us from our initial thoughts to a final working implementation. While it is possible to construct a good functional hierarchy, interweaved with its controlling activities, the mapping we specify between that model and the physical model often turns out to be naive, rarely constituting a satisfactory full-fledged implementation. Consequently, we must often add a new dimension to the modeling process by repeatedly refining the conceptual model.

Using appropriate visual formalisms can have a spectacular effect on engineers and programmers.

This can be done by preparing a new tier, or stratum, of functional hierarchies, one for each of the subsystems appearing in the structural view, and providing a lower-level mapping between these refined models and the subsystems themselves. This process may continue downward until a satisfactory level of design is reached.

These ideas are quite in line with Brooks' sympathetic discussion of top-down design. Of course, two crucial parts of this process concern the methodological issue of providing guidelines and heuristics for actually carrying it out, and the technical issue of showing consistency between the resulting strata. These are briefly discussed below.

Visual representation. Most issues of representation have been skirted above; indeed, some justification could be found in giving them second-class status. However, I believe that convenient media for representing the concepts and structures inherent in a model impact the very thinking that goes into constructing that model. In the one-person programming world, the availability of programming languages such as Pascal and C and even their precursors like Fortran, Algol, and PL/I has had a profound influence on a programmer's ability to conceive of good algorithms. Moreover, good representation is also instrumental in communicating those algorithms and their underlying ideas to others.

I agree with Brooks that flowcharts have become pitiful visualizations of programs, and even with a more general claim concerning the hopelessness of finding a general-purpose visual programming language that could replace conventional languages. But this opinion comes to a screeching halt where complex reactive systems are concerned.

Much of the conceptual construct underlying a complex reactive system is inherently topological in nature, and this is reflected in the vanilla approach outlined above. Hierarchies, with or without overlapping, and multilevel relationships, whether they concern structure, function, or behavior, can be captured naturally by simple, rigorous, and well-known notions from set theory and topology; these, in turn, have natural counterparts as spatial/graphical representations.

As argued elsewhere,[12] this fact gives rise to *visual formalisms*, in which encapsulation, connectedness, and adjacency play central roles, and lesser features, such as size, shape, and color, can also be exploited. Furthermore, all these graphical features come complete with rigorous mathematical semantics.

Visual formalisms have indeed been proposed for representing the various aspects of vanilla models.[3-5,11,12] From several years of following their application in large real-world projects, I have become convinced that using appropriate visual formalisms can have a spectacular effect on engineers and programmers. (An example of this is the avionics system for the state-of-the-art Lavi fighter at Israel Aircraft Industries, where the visual language of statecharts[11] was used for specification. Although these experiences are too recent to have yielded statistics, some comparisons and evaluations have already appeared.)

Moreover, this effect is not limited to mere accidental issues; the quality and expedition of their very *thinking* was found to be improved. Successful system development in the future will revolve around visual representations. We will first conceptualize, using the "proper" entities and relationships, and then formulate and reformulate our conceptions as a series of increasingly more comprehensive models represented in an appropriate combination of visual languages. A combination it must be, since system models have several facets, each of which conjures up different kinds of mental images.

Of course, the job is far from complete. Some aspects of the modeling process have not been as forthcoming as others in lending themselves to good visualization. Algorithmic operations on variables and data structures, for example, will probably remain textual. In addition, as Brooks aptly observes, some of the less obvious connections between the various parts of system models are not easily visualized. However, for a number of years, we have been doing far, far better than the "several, general directed graphs, superimposed one upon another" Brooks' describes. The graphical languages currently used are still two-dimensional, whereas some of the concepts could definitely do with higher dimensional visualization. This may still happen. In fact, realistic motion-based 3D techniques are rapidly coming into reach. A new aspect of visual languages that will have to be addressed is computerized support for the "nice looking" layout of diagrams. This is a difficult and challenging problem in which only marginal progress has been made.

Regarding hardware, our scopes are currently of limited scope, to use Brooks' captivating phrase, making the extent to which we can comfortably display very large visual models dependent on the availability of dramatically improved graphical hardware. Rather than taking this as a reason to abandon visual approaches, we should find it enlightening. For once, concepts and software ideas are ahead, waiting for the development of matching hardware. If the past record of hardware improvements is any measure, these developments will not be long in coming.

It is our duty to forge ahead to turn system modeling into a predominantly visual and graphical process. I believe this is one of the most promising trends in our field.

It is our duty to forge ahead to turn system modeling into a predominantly visual and graphical process.

Methods and guidelines. In addition to thinking with the "right" concepts and representing the resulting thoughts in appropriate programming languages, a programmer can call on a variety of well-established methods, guidelines, and techniques to help formulate a good solution to a one-person algorithmic problem. These constitute a large reservoir of knowledge accumulated over years, embodying the experience and expertise of generations of programmers, algorithm designers, and computer scientists. As might be expected, there has always been a great deal of cross-fertilization between the world of methods and techniques and the world of concepts and languages.

The story for complex reactive systems is no different, except that it is at a far more embryonic stage. Despite the proliferation of so-called methodologies, it is still too early to see a wide-ranging and well-understood collection of guidelines and techniques for the step-by-step process of system development.

Many of the proposed methodologies are not methodologies at all in that they do not contain recommendations about how to actually do things. For that matter, the vanilla approach described here is not a methodology either. However, what is worse is that many *do* prescribe recipes, but these often suffer from being too restrictive, hard to apply, or downright wrong.

One of the most unfortunate trends has been in presenting a method as exclusive, that is, preaching about its being *the* step-by-step way to develop entire systems. This can be compared in naiveté to someone advocating divide-and-conquer or branch-and-bound as *the* method to write programs.

The availability of a solid, general-purpose framework within which one can conceptualize, capture, and represent a system model seems to be far more important right now. All-encompassing recipes for how to get the work done simply do not exist; guidelines and techniques that work in special cases do exist, and more will surface in time. Obviously, they will be influenced by the choice of the framework and will, in turn, influence that framework and its evolution. And they will draw heavily on our experience in wielding the notions, concepts, languages, and tools.

Among the guidelines suggested are top-down and bottom-up approaches, which prescribe the raw order of things, as well as approaches related to the nature of the elements that are to drive the process, such as state-driven, function-driven, or data-driven techniques.

In principle, all of these can be followed quite smoothly within the vanilla framework, though constructing really good models, as well as choosing which of these guidelines to use for what systems, will obviously remain something of an art.

Some methods are further removed from the vanilla framework, since they advocate a somewhat different set of basic concepts. One of the better-known examples is the object-oriented approach, in which objects and their capabilities take precedence over activities and states. While it *is* possible to follow this approach within the confines of our basic framework, it's perhaps not as smooth-going as one would like. This is an excellent example of a more specialized, or exotic, flavor, which is already resulting in correspondingly specialized advances in languages and tools.

In addition to guidelines for the overall process of development, a number of heuristics have been addressed at the nontrivial process of mapping the conceptual model onto the physical one. They are often based on taking subtle advantage of the cohesion and coupling of activities.[4,7]

For pure software systems, this task is usually less perplexing, since the structure of the final product can be taken to correspond reasonably well with that of the conceptual model. However, embedded systems are different. In them, the physical breakup into components and subcomponents might be acutely orthogonal to the conceptual structure. These cases require more iterations in the design process, giving rise to several strata of physical and conceptual models, as discussed above. The importance of such heuristics stems from such cases. (These heuristics could well find their way into useful expert-system support tools, as envisioned by Brooks.)

Designers would do well to master all of these techniques, guidelines, and heuristics, and to use them to devise models in a manner that they deem most natural. In time, I'm certain we will outgrow the deep convictions we have cultivated around the various methodologies. We will stop trying to get everyone to use them exclusively for all systems, and they will reduce to their proper dimensions — taking their place side by side in our bag of tricks, just as conventional algorithmic methods have for one-person programs.

Analyzing the model

The preceding sections have repeatedly invoked the analogy between conventional algorithms and models of complex systems. When it comes to semantics and analysis, this analogy takes on a particularly interesting twist.

Although the importance of testing and analyzing one-person algorithms has always been acknowledged, the world of complex systems has long suffered from something of an indifference to such needs. By analogy, the situation was as if we were asked to solve one-person algorithmic problems without the possibility of running programs, and hence without being able to test and debug them at all.

Indeed, many past approaches to system development provided no means for capturing behavior, being centered instead on the functional aspects and dataflow. The approaches that did provide such means were informal, lacking the rigorous semantics necessary for even beginning to analyze the dynamics. Hence, it was impossible to predict in early stages how the system would behave if constructed according to the model.

Not until actual code was written — usually at a very late stage in the project by people other than those responsible for and capable of developing the "conceptual construct" and at much greater expense — could one expect to get reliable answers to "what if?" questions. This, of course, has had a deplorable effect on the expedition and quality of development efforts for large and complex systems.

As a consequence, most computerized tools that flourished around such methods (computer-aided software engineering — CASE — tools, as they are often called) concentrated on providing mere graphic-editing capabilities, sometimes accompanied by document generation, version control, and project management facilities. Their proponents heralded the ability of these tools to check model "consistency and completeness," which is really just a grand form of syntax checking.

To use my analogy again, it is like making sure, in a conventional program, that the begins and ends match, that procedure calls have the right number and types of parameters, and that all declared variables are indeed used. In the complex system arena, such checking includes the consistency of level-labeling schemes and of inputs and outputs within the hierarchies, the nonredundancy of flow elements, and so on; it is analogous to the checking carried out in one-person programming environments on-the-fly or in simple precompilation stages.

Since designing a complex reactive system is so much more massive and intricate an undertaking than writing a conventional one-person program, testing for consistency and completeness in system modeling is far more important than syntax checking in programs. Nevertheless, it remains a mere test of the syntactic integrity of the model and has very little to do with that model's conceptual and logical aspects.

Checking that a model is consistent and complete cannot prevent logical errors that cause a missile to fire unintentionally or a stock market system to run amok — exactly the kinds of mishaps that are at the heart of our problem. For this, we need the ability to carry out *real* testing and analysis.

You may feel the following discussion is unrealistically futuristic. Not so. All the possibilities we mention have been implemented in a computerized tool that supports the vanilla approach and that is being used in the development of real systems.[4] Several other tools also support some of these possibilities (see the "Tools for model execution" sidebar).

None of the implementations is perfect; each requires improvements and extensions. However, they do corroborate the feasibility of the ideas summarized below. In fact, since many of these ideas are standard practice in the world of conventional programming, the tools appear to be the first complex-system analogs of useful general-pur-

Tools for model execution

Although model execution is not a new idea, its vast potential has not yet been fully exploited. All the executability features discussed in the text are available in the Statemate tool,[5] the first release of which was developed between 1984 and 1987. It is currently being used mainly in the areas of avionics, telecommunication, and process control.

A number of additional tools support some of these features. A couple of them are commercially available, and others are still in research and development stages. Here are a few additional publications describing techniques and tools for executing models:

Diaz-Gonzalez, J., and J. Urban, "Prototyping Conceptual Models of Real-Time Systems: A Visual Perspective," *Proc. 22nd Hawaii Int'l Conf. System Sciences*, IEEE CS Press, Los Alamitos, Calif., Order No. 1912, 1989, pp. 358-367.

Jensen, K., "Computer Tools for Construction, Modification, and Analysis of Petri Nets," *Advances in Petri Nets, Part II*, W. Brauer, W. Reisig, and G. Rozenberg, eds., Lecture Notes in Computer Science, Vol. 255, Springer-Verlag, New York, 1987, pp. 4-19.

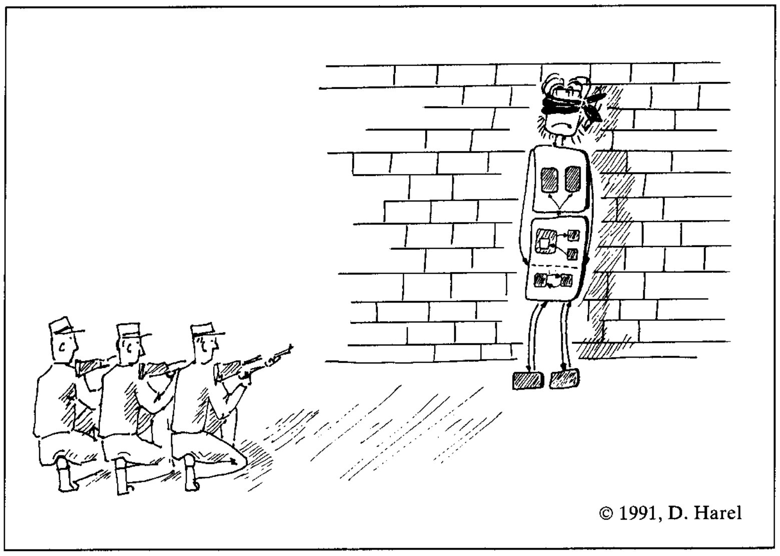
Pulli, P. J., "Pattern-Directed Real-Time Execution of SA/RT Specifications," *Proc. Euromicro Workshop on Real-Time*, IEEE CS Press, Los Alamitos, Calif., Order No. 1956, 1989, pp. 3-9.

Wang, Y., "A Distributed Specification Model and its Prototyping," *IEEE Trans. Software Eng.*, Vol. 14, No. 8, Aug. 1988, pp. 1,090-1,097.

Zave, P., and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," *IEEE Trans. Software Eng.*, Vol. 12, No.2, Feb. 1986, pp. 312-325.

pose programming environments.

Model execution. One of the most interesting notions to come out of recent work in systems engineering is that of *executable specifications* or, to fit in better with the terminology used here, *executable models*. Executing a model is analogous to running a program directly, with the aid of an interpreter. Unfortunately, the term has been erroneously equated with the animation of diagrams only. However, executability is in fact many-sided and far more significant.



Model execution.

A prerequisite to executing complex system models is the availability of a formal semantics for those models — most notably, for the medium that captures the behavioral view. Thus, while the adjective "visual" in the term "visual formalism"[12] was justified earlier on grounds pertaining to model representation, the word "formalism" is justified now on grounds pertaining to model analysis.

The core of model execution is the ability to carry out a single step of the system's dynamic operation, with all consequences taken into account. During a step, the environment can generate external events, change the truth values of conditions, and update variables and other data elements. Such changes then affect the status of the system; they trigger state changes in the controllers, activate and deactivate activities, modify conditions and variables, and so on. In turn, each of these changes can cause many others, often yielding intricate chain reactions.

A semantics for the model must contain sufficient information to capture these ramifications precisely. Given the current status and the changes made by the environment, calculating the effect of a step usually involves complicated algorithmic procedures, which are derived from, and reflect, that semantics.

Interactive and batch execution. The simplest way to execute, or "run," the model using a computerized tool is in a step-by-step interactive fashion. At each step, the user emulates the system's environment by generating events and changing values. The tool, in turn, responds by transforming the system into the new resulting status. If the model is represented visually, the change in status will also be reflected visually, say, by changes in color or emphasis in the diagrams.

Once we have the basic ability to execute a step, our appetite grows. We might now want to see the model executing noninteractively. To check, for example, that a telephone call connects when it should, we can prepare the relevant sequence of events and signals in a batch file, set up the model to start in the initial status, and ask our tool to execute steps iteratively, reading in the changes from the file. The graphic feedback from such a batch execution becomes an (often quite appealing) animation of the diagrams.

By executing scenarios that reflect the way we expect our system to behave, we are able to verify that it will indeed do so — long before final implementation. If we find that the system's response is not as expected, we may go back to the model, change it, and run the same scenario again. This is analogous to single-step — or batch — debugging of conventional programs.

It should be emphasized that scenarios can be run at any time in the development effort, as long as the portion of interest is syntactically legal. During an execution, the user plays the role of all parts of the model that are external to the portion being executed, even if those parts will eventually be specified and thus become internal.

Again, from several years of seeing such execution capabilities used, mainly in large aerospace and electronic industries, I have become convinced of their value (again, no statistics are yet available to quantify this impression, although a couple of preliminary case studies have appeared). These execution capabilities appear to introduce an entirely new and powerful dimension into the task of verifying and debugging system models.

I have seen model executions uncovering hitherto unknown patterns of behavior, when the members of the development team thought they had covered everything. As a result, these people were able to discuss deep behavioral issues that would otherwise have been swept under a rug of enormous unreadable specification documents.

I have seen engineers use executability to tackle crucial problems and, very early in the project, correct subtle conceptual errors — ones that could otherwise go undiscussed or undetected until it was too late. And typically, all these phenomena start to take place as soon as the first executions are run.

Customer representatives are often involved in these stages, which further supports what Brooks and others have urged: extensive prototyping and simulation of the system early on with the client.

Programmed execution. Our appetite now becomes even greater. We now might ask ourselves: If the tool can execute the model in detail, reading events in from a file, why should we be satisfied with merely witnessing the run in action and inspecting the final status? We would like to be able to incorporate breakpoints, causing the execution to suspend and the tool to take certain actions when particular situations come up. These actions can range from temporarily entering interactive mode (in order to monitor careful step-by-step progress) to executing a piece of ready-made code that describes a bottom-level activity.

In fact, we need not restrict ourselves to running self-devised scenarios. We might want to see the model executing under circumstances that we do not care to specify in detail. We might like to see its performance under *random* conditions and in both typical and less-than-typical situations. Such a capability gets to the heart of the need for an executable model: to minimize the unpredictable in the development of complex systems.

This more powerful notion of inspecting a model is achieved by the idea of *programming executions*, using a special metalanguage supported by the tool. Programs in this language (which might be appropriately termed an *execution control language*) can be set up to look out for predefined breakpoints and accumulate information regarding the system's progress as it takes place.

As a simple example, in a typical flight of an aircraft we are specifying, we might want to know how many times the radar loses a locked-on target. Since it might be difficult for the engineer to put together a typical flight scenario, we can tap the power of our tool by instructing it to run many typical scenarios, using the accumulated results to calculate average-case information.

The tool follows typical scenarios by generating random numbers to select new events according to predefined probability distributions. The statistics are then gathered using appropriate breakpoints and simple arithmetical operations. The ideas behind these techniques are, of course, well known. However, the point is to extend them to conceptual models of complex systems, long before the costly final-implementation stages.

In a similar vein, we can use programmed executions to apply other, more powerful kinds of dynamic tests to system models. For example, we might set up an execution control program to carry out performance analysis. If we want to check whether an operating system we are modeling will ever require more main memory than some maximum allowed value, we can associate with the relevant activities in the functional view values that represent our knowledge about their memory consumption. We can then program the tool to run many typical scenarios, calculating the maximum memory consumption of all activities that are active simultaneously.

We can use programmed executions to apply other, more powerful kinds of dynamic tests to system models.

Despite its being applied to a system model, and not to a final implementation, this approach to analysis is far more informative than the extraction of worst-case estimates from simple graphs of process dependencies. The model being analyzed will (hopefully) be realistic and detailed, and executing it reflects precisely what would have happened had we run the real system instead.

If our analysis shows that the memory limit might indeed be exceeded, the tool can support that prediction by supplying the actual sequence of events that would cause it. Clearly, by replacing memory values with time information, similarly meaningful timing analysis can be carried out as well.

In general, then, carefully programmed executions can be used to inspect and debug the system model under a wide range of test data to emulate both the environment and the as-yet-unspecified parts of the system and to analyze the model for performance and efficiency.

Exhaustive executions. When executing the model, we might detect such unpleasant anomalies as deadlocks or behavioral ambiguities (nondeterminism). However, finding and eliminating these in the cases that we happen to encounter does not ensure they will never occur in the lifetime of the system. It would be extremely useful to be able to run through all possible scenarios in search of such situations, by generating all possible external events and all changes in the values of conditions and variables.

We might also be interested in *reachability tests*, which would determine whether — when started in some given initial situation — the system can ever reach a situation in which some specified condition becomes true. This condition can be made to reflect desired or undesired situations. Moreover, we could imagine the test's being set up to report on the first scenario it finds that leads to the specified condition, or to report on all possible ones, producing the details of the scenarios themselves. We thus arrive at the idea of *exhaustive executions*.

Are such tests realistic? Could we subject the model to an exhaustive reachability test, for example, after which we will know for sure whether there is any possibility of its occurring under any possible circumstances? The answer, in principle, is yes, but with serious reservations. The number of possibilities that might have to be considered in an exhaustive execution can easily become incredibly large, even if we ensure that it is finite by limiting the possible values of the variables.

To get a feel for the sizes involved, a behavioral model that contains about 40 concurrent components, each with about 10 states, has more state configurations and, hence, might have more possible scenarios than the number of elementary particles in the entire universe. There can never be a language, method, or tool with which one can, in general, consider all of these in any reasonable amount of time.

This doesn't mean, however, that such tests are a bad idea.

First, the above numbers denote worst-case asymptotic estimates; a real system might very well have far fewer scenarios that can actually happen, and a careful process of considering only those that are feasible will take far less time than the worst-case estimate.

In fact, just such a reachability test was recently applied to a model of the firing mechanism of a certain, already deployed, ballistic missile system. (The Statemate system[5] was used for this. The main part of the underlying model consisted of a statechart[11] with about 80 states. However, since these included parallel state components, the real number of states was much larger.) In less than three hours on a standard workstation, the test terminated, in the process discovering a new sequence of events, unknown to the design team, that leads to the firing of the missile!

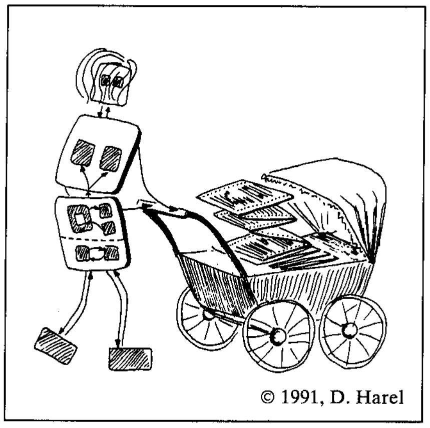
Second, exhaustive tests can be run on small, critical, and well-isolated parts of the model. We can instruct the tool to ignore some of the external events or to avoid simulating the details of certain activities. Clearly, this can cause it to overlook crucial situations, but the advantage is that the set of scenarios it considers is greatly reduced. To maximize the test's effectiveness, such limiting constraints should be prepared very carefully, using as much knowledge of the modeled system as possible. This is another place where expert systems might come in handy.

Third, even if exhaustive tests cannot always be completed in reasonable amounts of time, it would be wise to have them run in the background, perhaps at night or on weekends, for as long as we can afford. There is nothing wrong with routinely submitting large system models to powerful supercomputers for exhaustive testing, even nonexhaustively. Since the tool can be set up to report on phenomena as they are discovered, the more time we observe such tests running without surprises the more confidence we have in the integrity of our model.

Watchdogs and temporal verification. Often, we are interested in establishing properties of the model that are of a global nature but are more involved than reachability or freedom from deadlock. Suppose we want to make sure that a certain party in a communication protocol never sends two consecutive messages unless a special item, say, an acknowledgment, is sent in the interim.

Although seemingly more complicated, this query can be cast in the form of a reachability test in the following way. First, construct a small special-purpose "piece" of behavioral specification that is carefully set up to enter a special state if and when the offending situation occurs. Next, attach this *watchdog*, as it is called, to the original model as a concurrent behavioral component and run a reachability test on the extended model to find out whether the special state can ever be entered. Since the watchdog runs in parallel with the rest of the model, the effect will be as desired.

Watchdogs can be used to verify the model against a wide variety of properties. *Temporal logic*,[6] one of the most useful and well-known media for specifying global constraints on the behavior of a system, nicely complements modeling approaches that specify behavior in a more local, operational fashion. Under certain technical conditions, any temporal logic formula can be systematically translated into a watchdog, reducing the problem of verifying the complicated formula to that of establishing a much simpler property, such as reachability. The watchdog is then attached to the high-level controlling activity of the original model, resulting in a modified model, and an appropriate exhaustive test is run.


© 1991, D. Harel

Code generated from model.

Actually, such verification need not be based solely on exhaustive executions. Research into the theory and technology of automatic verification of very large (but finite-state) systems against properties in temporal logic is already showing promising results. The techniques being developed in this area are far more subtle and efficient than brute-force exhaustive executions, and I believe that they will eventually find their way into system analysis tools. This direction of work might very well bring true system verification into the living room, so to speak.

Code generation. Even in its most advanced forms, executability is analogous to running conventional programs using *interpretation*. Complex systems are also amenable to the analog of *compilation* — that is, translating a model into runnable code in a lower level language. We call this ability *code generation*, although the term is often used to denote the more humble ability to recast an unadorned functional description as a template of code that contains empty-body procedures for the controllers and the bottom-level activities. However, since the behavioral view does not exist (or is not covered) in such cases, the resulting code is but a scaffold that has to be enriched by handwritten code for the most crucial parts — notably, those that depict the dynamics. It is thus like an automobile without an engine. The notion we have in mind here is far stronger. (Indeed, new terms, such as *codifier* and *codification*, might be more appropriate than *code generator* and *code generation*.)

Using the vanilla approach terminology discussed earlier, we are talking about the translation of an entire conceptual model, that is, an activity hierarchy with all of its add-ons, including the controlling statecharts, into a programming language such as C, Modula 2, or Ada.

If the model contains bottom-level activities that were left unspecified, but have library routines or specially prepared code supplied by the user, this code can be linked to the code generator's output, thus completing the picture. Since the behavioral aspects are an integral part of the conceptual model, they too are included in the translation. Hence, the resulting code can be run as is and, in terms of its dynamic semantics, is equivalent to the model itself. Needless to say, as in model execution, code generation can be carried out on any syntactically legal portion of the model and at any stage of its development.

Generated code is sometimes referred to as *prototype code*, since it reflects only the design decisions made in the process of preparing the conceptual model, and not decisions driven by implementation concerns. In many real-time applications, this code is not as efficient as the required real-time code. Nevertheless, it runs much faster than executions of the raw model itself, just as compiled code usually runs faster than an interpreted program.

Using generated code. One of the main uses of code-generator output is in observing the system performing under close-to-real-world circumstances. For example, the code can be ported to, and executed in, the actual target environment or, as is often the case in earlier stages, in a simulated version of the target environment.

The code can be linked to "soft" panels — graphical mock-ups of control boards, complete with images of display screens, switches, dials, and gauges — that represent the actual user interface of the final system. These panels appear on the screen and can be manipulated with mouse and keyboard.

In the past few years, a number of companies have used this approach in design reviews involving customers and contractors, and it has proved to be extremely helpful — much more so than the typical documentation that accompanies such reviews.

It's important to point out that these system interface panels are not driven by hastily written code prepared especially for prototype purposes, but by code that was generated automatically from a model that is typically thoroughly tested and analyzed before being subjected to code generation. Moreover, when parts of the real target environment are available, they too can be linked to the code, and the runs become even more realistic.

Code generation is thus to be used for goals that go beyond the development team, in that code-driven mock-ups can be used as part of the standard communication between customer and contractor or contractor and subcontractor. It is not unreasonable that such a running version of the system model be a required deliverable in certain development stages.

A good code-generation facility would also have a debugging mechanism, with which the user can trace the executing parts of the code back up to the system model. Breakpoints can be inserted to stop the run when specified events occur, at which point the model's status can be examined and elements can be modified on-the-fly before resuming the run.

If substantial problems arise, changes can be made in the original model, which is then recompiled down into code and rerun. As in executions, trace files can be requested, recording crucial information for future inspection. Carrying the analogy between compilation and code generation a step further, this ability is tantamount to source-level debugging.

In addition to compiling, or codifying, the model itself, we can automatically produce code from specially prepared segments of behavior, such as watchdogs or test suites that are not part of the model but are used to execute and analyze it. For these, of course, the code generator output is actually final code.

As more and more code becomes final production code, the entire system comes closer to its final form.

An interesting variation calls for replacing high-level programming languages as the target medium for generated code by hardware description languages. A particular example is VHDL (which stands for VHSIC hardware description language, with VHSIC the abbreviation for very high-speed integrated circuit). In this way, hardware designers can also benefit from the virtues of the modeling and analysis techniques discussed above, and then translate their models into VHDL code, which can be subjected to silicon compilation or other appropriate procedures.

Verifying consistency between levels. Recall the process of preparing tiers, or strata, of conceptual models according to the physical model of the system. How can we establish the consistency of one level with the next?

There are a number of ways in which model-analysis techniques can help. The basic idea is to redirect the efforts from the task of inspecting and debugging a single model to the task of comparing two models. This applies to all manner of analysis and verification: interactive, programmed, and exhaustive execution; watchdogs and temporal verification; and code generation.

For example, we may execute the conceptual model prepared for a subsystem under the same conditions used to execute the original model of the entire system, and compare the results. One way to do this involves preparing scenarios for executing the new model directly from trace files of executions run on the original model. Clearly, this is not as simple as it sounds, and much research on this topic is still needed.

As far as code generation goes, we can often replace parts of the code generated from the original model by code generated from the newly designed subsystem models. If the final system is to be implemented in software, this has the effect of gradually bringing the original prototype code down toward a real implementation.

As subsystems are remodeled, their generated code is incorporated into the code that was generated one level higher. As more and more of the code becomes final production code, the entire system comes closer to its final form.

It is not out of the question that this process will also become amenable to computerization. We can envision a user making restructuring decisions in the design stages (perhaps aided by an expert system) and the tool taking over from there, reorganizing the generated code in new, more efficient ways that reflect those decisions.

Combined with optimization procedures, which are badly needed and will hopefully be developed in the future, code generation has a chance to go far beyond prototyping, further justifying its role as the true complex system analog of conventional compilation.

The vanilla framework for system modeling outlined above is far from being universally accepted. Many of its facets are rooted in well-established and familiar ideas, but others are more recent and immature and require further work and experience.

On some issues, there is little agreement among researchers and practitioners, such as how to best approach the specification of behavior. I believe that the general framework is a good one and that there are also adequate proposals for behavioral specification. However, even the overall mold could easily turn out to be inadequate.

If it does not become the accepted analog of good old vanilla programming, then some other approach will. The precise form the winning effort takes on will be secondary, though I am fully convinced that reactive behavior will be one of its most crucial and delicate components, rigorous semantics included, and that visuality will play center stage. From this basic framework will evolve more specialized and exotic ones, for which appropriate modeling languages will be designed and implemented and methods and guidelines conceived and mastered.

Things are far clearer in the analysis realm, where most of the abilities we have discussed are, to some extent, independent of the idiosyncrasies of the particular modeling approach. I believe that system development tools that lack powerful execution and code-generation capabilities will all but disappear.

Whichever approach people ultimately use to conceptualize and model their systems, the ability to thoroughly execute the resulting models and to compile them down into conventional high-level code will become indispensable. In a way, this too is vanilla. I believe that in time more exotic kinds of executability features will emerge, such as ones tailored to carry out timing and performance analysis, gather statistics, or compare the behavioral aspects of separate models.

A number of research directions present themselves, and in some there is already a promising body of work. Among the most important are

The current situation and prospects for significant improvement indicate that we are at the start of a new and exciting era.

(1) improving the techniques for generating high-quality code from conceptual models, and providing (semi)automated help to make design decisions in the process, and
(2) enabling truly useful computerized verification of conceptual models against global constraints.

One of the crucial ingredients for success in these areas is extensive cooperation and collaboration of researchers in software and systems engineering with those in compilers, optimization, and heuristics for item (1) and in logic, semantics, and verification for item (2).

The current situation and the prospects for significant improvement indicate that we are at the start of a new and exciting era. ■

Acknowledgments

Discussions over the years with Amir Pnueli, Michal Politi, Rivi Sherman, and Moshe Cohen were extremely beneficial in helping me form the opinions voiced in this article. I am indebted to Derek J. Hatley, C.A.R. Hoare, Daniel Jackson, Ray Moritz, David L. Parnas, and the anonymous referees for commenting on a preliminary version of this material. This research was supported in part by grants from the Gutwirth Foundation and the Yeda Foundation for Applied Research.

References

1. F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, Apr.

1987, pp. 10-19. Also appeared in *Information Processing 86*, H.-J. Kugler, ed., Elsevier Science Publishers B.V., North-Holland, 1986, pp. 1,069-1,076.

2. D.L. Parnas, "Software Aspects of Strategic Defense Systems," *Comm. ACM*, Vol. 28, No. 12, 1985, 1,326-1,335. Also in *Am. Scientist*, Vol. 73, No. 5, 1985, pp. 432-440.

3. P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, Vols. 1-3, Yourdon Press, New York, 1985.

4. D.J. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1987.

5. D. Harel et al., "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.*, Vol. 16, No. 4, Apr. 1990, pp. 403-414. Preliminary version in *Proc. 10th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 849 (microfiche only), 1988, pp. 396-406.

6. A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," *Current Trends in Concurrency*, de Bakker et al., eds., Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, 1986, pp. 510-584.

7. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems," *Comm. ACM*, Vol. 15, No. 5, 1972, pp. 1,053-1,058.

8. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

9. L.L. Constantine and E. Yourdon, *Structured Design*, Prentice Hall, Englewood Cliffs, N.J., 1979.

10. G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," to appear in *Science of Computer Programming*, North-Holland. Also in INRIA Research Report 842, 1988.

11. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, North-Holland, Vol. 8, No. 3, 1987, pp. 231-274. Preliminary version appeared as Tech. Report CS84-05, Weizmann Inst. of Science, Rehovot, Israel, 1984.

12. D. Harel, "On Visual Formalisms," *Comm. ACM*, Vol. 31, No. 5, 1988, pp. 514-530.

David Harel is the William Sussman Professor of Mathematics at the Weizmann Institute of Science in Israel and chair of its Department of Applied Mathematics and Computer Science. He cofounded i-Logix, Burlington, Massachusetts. Harel received a best paper award at the 10th International Conference on Software Engineering in 1988, and his book *Algorithmics: The Spirit of Computing* (Addison-Wesley, 1987) was the spring 1988 main selection of the Macmillan Library of Science.

Harel received the BSc degree from Bar-Ilan University in 1974, the MSc from Tel Aviv University in 1976, and the PhD from the Massachusetts Institute of Technology in 1978. He is a senior member of the IEEE, and a member of the IEEE Computer Society and the ACM.

Readers can contact Harel at the Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel. His e-mail address is harel@wisdom.weizmann.ac.il.