On the Development of Reactive Systems*

D. Harel and A. Pnueli

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, Israel

January, 1985

Abstract

Some observations are made concerning the process of developing complex systems. A broad class of systems, termed *reactive*, is singled out as being particularly problematic when it comes to finding satisfactory methods for behavioral description. In this paper we recommend the recently proposed statechart method for this purpose. Moreover, it is observed that most reactive systems cannot be developed in a linear stepwise fashion, but, rather, give rise to a two-dimensional development process, featuring behavioral aspects in the one dimension and implementational ones in the other. Concurrency may occur in both dimensions, as *orthogonality* of states in the one and as *parallelism* of subsystems in the other. A preliminary approach to working one's way through this "magic square" of system development is then presented. The ideas described herein seem to be relevant to a wide variety of application areas.

Why Another Paper on System Development?

The literature on software engineering, programming languages, and system and hardware design, is brimming with papers describing methods for specifying and designing large and complex systems. Why then are we

* This research was supported in part by grants from AD CAD Inc. and the Israel Aircraft Industries.

NATO ASI Series, Vol. F13
Logics and Models of Concurrent Systems
Edited by K. R. Apt
© Springer-Verlag Berlin Heidelberg 1985

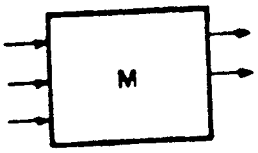
writing yet another one?

In many kinds of computation-oriented or data-processing systems, sometimes characterized as sequential or functional systems, there is, for the most part, consensus as to the basic philosophy for design. For more complex systems, involving many concurrently executing components, which at times integrate software and hardware, there is much less of an agreement. As we argue below, this is due to an essential difference between two kinds of systems that makes the process of developing the more complicated of the two inherently more difficult. Indeed, we would like to think of the present paper as primarily containing an attempt to clarify some of the underlying notions which seem to us to be fundamental. In passing, we shall attempt to address such issues as the kinds of systems that require new ideas, and the gaps such ideas ought to fill.

Which Systems are Problematic?

We would like to state from the start that by "systems" we do not wish to restrict ourselves to ones which are software-based, hardware-based or so-called computer-embedded. The terminology we shall be using is general enough for these and others, and so we shall not be specific about the final form the implementation of the system takes.

In many circles it is common to try to identify the features characterizing "difficult" systems; that is, the ones for which special methods and approaches are needed. Resulting from these efforts are various dichotomies distinguishing the easily-dealt-with systems from the problematic ones. Some people (e.g. in the programming language semantics community) have put forward the deterministic/nondeterministic dichotomy: systems for which the next action is uniquely defined can be easily defined, while nondeterminism requires special treatment. Others (such as certain verification researchers) have suggested that the problems lie in perpetual systems, whereas terminating ones are easy. Additional dichotomies that have been suggested are the synchronous/asynchronous, "lazy" / real-time, and off-line/on-line ones, and what is perhaps the most popular one: the sequential/concurrent dichotomy. Indeed, concurrency gives rise to problems that are quite different from the ones sequential systems present, and there are entire schools of thought devoted to solving the problems raised by the presence of concurrently operating elements.



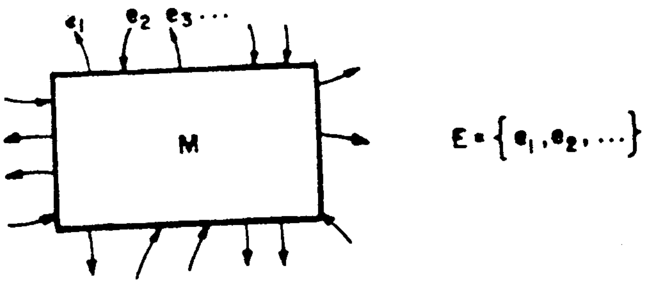


Fig. 1: A transformational system as a black box

Fig. 2: A reactive system as a "black cactus"

As it turns out, all the dichotomies mentioned are real and the problems the more difficult members of each pair present are indeed crucial. We wish, however, to point to another dichotomy, one which we think is the most fundamental of all and the one that seems to us to best distinguish systems that are relatively easy to develop from those that are not. We feel that once the problematic part of this pair is satisfactorily solved, most of the others will yield less painfully too. Our proposed distinction is between what we call *transformational* and *reactive* systems. A transformational system accepts inputs, performs transformations on them and produces outputs; see Fig. 1. Actually, we include in the definition of a transformational system also ones which may ask for additional inputs and/or produce some of their outputs as they go along. The point, however, is that, globally speaking, these systems perform input/output operations, perhaps prompting a user from time to time to provide extra information. Reactive systems, on the other hand, are repeatedly prompted by the outside world and their role is to continuously respond to external inputs; see Fig. 2. A reactive system, in general, does not compute or perform a function, but is supposed to maintain a certain ongoing relationship, so to speak, with its environment.

At this point, the reader should observe that reactive systems are everywhere. From microwave ovens and digital watches, through man/machine based software systems, silicon chips, robots and communication networks, all the way to computer operating systems, complex industrial plants, avionics systems and the like. Common to all of these is the notion of the system responding or reacting to external stimuli, whether normal user-generated or environment-generated ones (such as a lever pulled or the temperature rising), or abnormal ones (such as a power failure). Such systems do not lend themselves naturally to description in terms of functions and transformations. Of course, mathematically speaking, it is always possible to take time itself as an additional input (and output) and turn any reactive system into a transformational one; needless to say, this idea is unrealistic and we shall not adopt it here.

The transformational/reactive dichotomy cuts across all the aforementioned ones: both types of systems can be deterministic or not, terminating or not, on-line or not, and contain concurrently executing components or not. Also, reactive systems can be required to respond in real-time or not, and the cooperation of their components can be required to be synchronous or not. What then is so important about the distinction we are making, and why are we claiming that it is the reactive nature of systems that is problematic?

What is the Problem?

The answer to these questions seems to us to be rooted in the notion of the *behavior* of a system. While the design of the system and then its construction are no doubt of paramount importance (they are in fact the only things that ultimately count) they cannot be carried out without a clear understanding of the system's intended behavior. This assertion is not one which can easily be contested, and anyone who has ever had anything to do with a complex system has felt its seriousness. A natural, comprehensive, and understandable description of the behavioral aspects of a system is a must in all stages of the system's development cycle, and, for that matter, after it is completed too.

Taking a very broad and abstract view, we may describe a typical top-down development process as a sequence of transformations:

$$(\mathbf{M}^{(0)}, \mathbf{S}^{(0)}) \rightarrow (\mathbf{M}^{(1)}, \mathbf{S}^{(1)}) \rightarrow \cdots \rightarrow (\mathbf{M}^{(f)}, \mathbf{S}^{(f)})$$

The structure studied at each level is a pair $(\mathbf{M}^{(i)}, \mathbf{S}^{(i)})$ comprising a *specified system* at the i'th level of detail; $\mathbf{M}^{(i)}$ is the i'th level physical, or implementational, description, and $\mathbf{S}^{(i)}$ is the behavioral specification. At the top level $\mathbf{M}^{(0)}$ might be highly underspecified, with $\mathbf{S}^{(0)}$ using such vague terms as "a data-base system responding to conjunctive queries", "a plane for interception", etc. Each level, even the 0'th one, needs some description of the interface of the system with its environment. This can be done by including a list $\mathbf{E}^{(i)}$ in each of the $\mathbf{M}^{(i)}$, containing descriptions of those input and output channels, signals, requests and responses, that constitute the system's interaction with the "outside world". The level of detail in the interface lists can also vary with i, from highly abstract items such as "request communication" or "display target", all the way down to concrete lists of buttons, levers, displays and alarms. The corresponding behavioral specification $\mathbf{S}^{(i)}$ should characterize the desired behavior of the system, in as complete a manner as possible, using the elements of $\mathbf{E}^{(i)}$.

Any development step progressing from level i to level $i + 1$ must include a verification of the consistency of $S^{(i+1)}$ with $S^{(i)}$. This is true regardless of whether it was the refinement of $M^{(i)}$ or of $S^{(i)}$ that prescribed the progress made. At times one can provide a rigid set of possible refinement rules for producing $S^{(i+1)}$ from $S^{(i)}$. In these cases the rules are internally, or locally, consistent, so that a development process that uses them is automatically guaranteed to be globally consistent. A good example of this is in pure software systems where one may use various established program transformations which also prescribe the corresponding transformations on the specification.

To be slightly more specific, one can think of a reactive system **M** as a "black cactus" of sorts (in contrast with a black box). The "thorns" of this cactus are simply the interface elements comprising the set **E**; see Fig. 2. A behavioral description **S** of the system should give rise to a set consisting of the legal sequences of these external input and output events and conditions. Thus, describing the behavior should boil down to defining a subset of the set of finite and infinite words over **E**. Of course, if timing is important, this simplistic definition has to be extended, for example by attaching a time stamp to each element in the sequence, or by specifying the timing constraints separately, but for now specifying the sequences will suffice. This relationship between **M**, **E** and **S**, holds for the level-dependent versions $\mathbf{M}^{(i)}$, $\mathbf{E}^{(i)}$ and $\mathbf{S}^{(i)}$ too.

Now, there have been numerous suggestions for methods, languages and formalisms to be used in the development of complex systems. Many of these are extremely helpful, and in general adopt a stepwise approach such as the one just outlined. They are, for the most part, well-structured and modular, and recommend a gradual step by step development in a top-down, bottom-up or mixed fashion; many of them are visual in nature, or at least have a visual counterpart and are thus easy to grasp; a number of them are based on firm and precise mathematical models which admit certain kinds of formal reasoning.

However, as it turns out, the methods existing for stepwise, well-structured and coherent development of systems are predominantly ***transformational*** in nature. In transformational systems it is possible, actually highly desirable, to decompose the system in a way reflecting the natural "structure of the problem", as it is sometimes referred to. In other words, a high-level description of the problem, in the form of the transformation, or function, that the system is supposed to carry out, is decomposed in these methods into several smaller problems of the same species, in the form of lower level transformations, with the appropriate identifications made among incoming and outgoing items. Each lower level transformation is then considered in its own right, and further decomposed. This is but another way of saying that each $S^{(i+1)}$ consists of a set of transformations, each of which was obtained from a transformation in $S^{(i)}$ by transformational, or functional, decomposition. The system description $M^{(i+1)}$ is then taken to match the transformations described in $S^{(i+1)}$ as closely as possible.

This is admittedly a very crude and sketchy account of such methods, but the point we wish to make is that the procedure it illustrates works nicely for transformational systems because transformations decompose naturally into other transformations, and implementations of transformations decompose into implementations of other transformations. It is therefore a small thing for one to observe that the two decompositions can (and then even recommend that they should) be essentially the same, or at least that they be related via a simple mapping. This is particularly attractive due to the fact that transformational decomposition provides not only static information but also the dynamics necessary for a good behavioral description. For example, a conventional structure diagram or a function tree, two of the kinds of descriptions recurring in the literature, can be given clear operational meanings when considered for transformational systems: inputs (data and/or control) flow into boxes, modules or functions, which proceed to perform their designated transformations, yielding outputs which in turn flow into others, etc. This is a wholly satisfactory behavioral description of a transformational system.

It is for these reasons that in most software engineering views of the life cycle of a system the specification stage is more or less followed by the design one: decompose the problem (=specify), and then use its parts and their interconnections as the basis for planning the implementation (=design). This idea is also one of the implicit mottos of structured programming: let the structure of the program reflect the structure of the problem, or, as one might say, let chunks of implementation (e.g. procedures, blocks, tasks, etc.) be made to correspond to chunks of behavior.

Our main argument here is that this cheerful situation does not apply to reactive systems at all. In a reactive system, even a pure software one, it is not clear if or how complex behavior can at all be decomposed beneficially into chunks, let alone for that decomposition to become the basis for system design. This observation notwithstanding, it is ironical that a breakup of the behavior is precisely what will eventually have to be found, whether one likes it or not: the final system will, if completed, consist of various increasingly more complex actual components (software, hardware or mixed), each of which will, by its very existence, have to have some kind of associated behavior. Moreover, these components will most probably be reactive themselves. And so, developing the system will ultimately have to involve some kind of physical decomposition, which, one way or another, will have to be matched by a behavioral decomposition too.

Let us for now, however, postpone the problem of connecting behavioral descriptions of reactive systems with their implementational ones. Our first concern is with specifying the reactive behavior itself. How does reactive behavior decompose? What can be done to encourage stepwise refinement of the behavioral aspects of a system? How can one cope with the intricacy that the behavior of a complex reactive system presents??

Before attempting to answer these questions, let us state the following requisites, which we feel ought to be required from any satisfactory method for behavioral description:

(i) It should provide descriptions that are well-structured, concise, unambiguous, readable, and easy to understand.

(ii) It should be solely descriptive, eliminating, or at least minimizing, dependence on any implementational issues.

Requirement (i) implies that the method must have a simple but rigourous semantics, and (ii) implies that the structuring of a behavioral description should reflect the natural decomposition of the problem rather than that of the implementation.

A Method for Behavioral Description

The ***statecharts*** method was introduced recently[1] as a visual formalism for specifying the behavior of complex reactive systems. The process of preparing statecharts for a system is called the system's ***statification***, and

[1] See D. Harel, "Statecharts: A Visual Approach to Complex Systems", CS84-05, The Weizmann Institute of Science, February 1984 (revised December 1984).

it consists of describing the system's behavior in terms of states, events and conditions, with combinations of the latter two causing transitions between the former. Both states and transitions can be associated in various ways with output events, called *activities*, which can be triggered either by executing a transition or by entering, exiting, or simply being in a state. The system's inputs are thus the (external) events and its outputs are the (external) activities; their union comprises the interface set **E**.

This, as the reader can no doubt see, is a standard and well-known idea, and is actually a simple combination of the Moore and Mealy definitions of finite state automata. The allowed sequences over **E** correspond to the language accepted by the automaton. Moreover, such automata come complete with a standard visual renderation, the transition diagram. This classical state transition method, however, has been all but abandoned as a way of specifying the behavior of complex systems since it provides no modularity or hierarchical structure, and suffers acutely from the exponential blowup in the number of states that need be considered, and hence also in the number of transitions. Indeed, a state/event description seems to have to consider all possible combinations of states in all the components of the system; hence the exponential growth.

The statechart method is rooted in an attempt to revive this old and natural way of thinking about a system's behavior, by extending it in several fundamental ways aimed at overcoming the aforementioned difficulties. The extensions apply to the underlying nongraphical formalism too, but personal preference towards visual descriptions has led us to present the ideas in terms of the graphical version. Some of the extensions are now briefly described, but the reader is strongly advised to consult the original paper for the others, as well as for a detailed example and further discussions.

States in a statechart can be repeatedly combined into higher-level states (or, alternatively, high-level states can be refined into lower-level ones) using AND and OR modes of clustering. Fig. 3 shows a state B whose meaning is "to be in B the system must be in precisely one of D, E or F," and Fig. 4 shows a state A whose meaning is "to be in A the system must be both in B and in C." Notice, however, that in Fig. 4, B and C are themselves OR states, thus the actual possibilities are the state configurations (D,G), (D,H), (E,G), (E,H), (F,G), and (F,H). We say that D, E and F are *exclusive* and B and C are *orthogonal*.

Transitions in a statechart are not level-restricted and can lead from

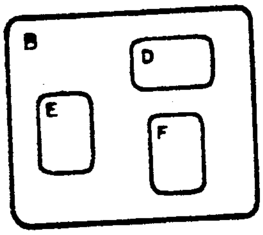


Fig. 3: OR-ing states

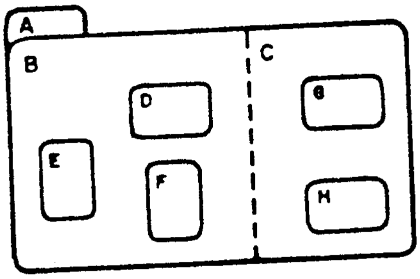


Fig. 4: AND-ing states

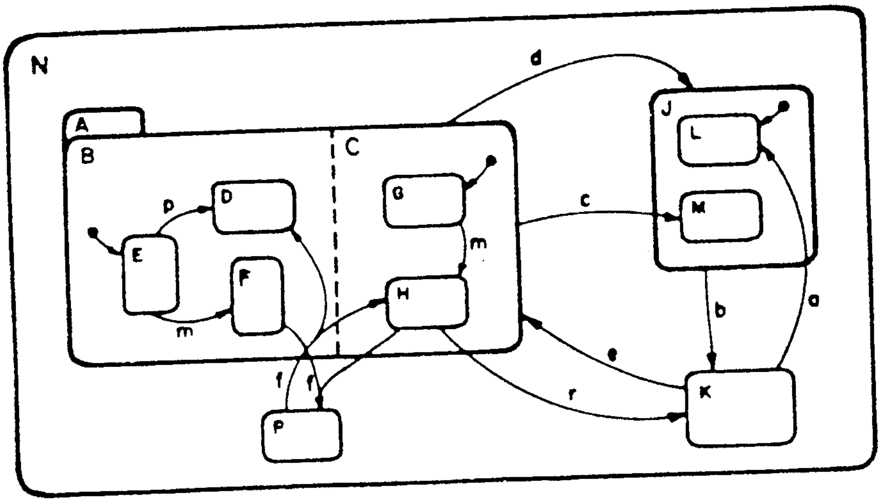


Fig. 5: An output-free statechart

a state on any level of clustering to any other. A significant decision here is to take a transition whose source state is a "superstate" to mean "the system leaves this state no matter which is the present configuration within it." In this way, while event *a* in Fig. 5 causes a simple transition from state *K* to *L*, the event *b* exemplifies a concise way of causing the system to leave *L* or *M*, i.e. any possibility of being in *J*, and to enter *K*. Likewise, *c* causes the system to exit any one of the *A*-configurations listed above and enter *M*. If the target of a transition is a superstate, as in the case of events *d* or *e* in this example, a default arrow must be present indicating which of the lower-level states is actually to be entered (*L* or the combination (E, G) in this example).

Actually, transitions are in general from configurations to configurations, owing to the possibility of orthogonal components in the source and target states. Thus, in Fig. 5 if event *f* takes place in configuration (F, H) the system enters *P* and if the same happens in *P* the system enters (D, H). Concurrency and independence are both made possible by orthogonality: on the one hand event m causes simultaneous transitions in B and C if the configuration is (E, G), and on the other p causes E to be replaced by D regardless of, and with no change to, the present state in C. It is noteworthy that orthogonality (and hence the possibilities it raises) is allowed and encouraged on any level of detail, as the statifier sees fit. Accordingly, a configuration can be layered too, containing orthogonal state components on many levels.

Outputs can be associated with transitions as in Mealy automata by writing a/b along an arrow; the transition will be triggered by a and will in turn cause b to occur. Similarly, b can be associated with (entering, exiting, or simply being in) a state, in line with Moore automata. In either case b can be an external event or an internal one, in the latter case triggering perhaps other transitions elsewhere in some orthogonal state.

The statification method is purely behavioral and requires of its user to think in terms of the system's conceptual states and their interconnections. It caters for modular "chunking" of behavior in several ways, most notably by using exclusivity and orthogonality of states, and provides the mechanisms needed for manipulating these modules as separate entities. Statecharts are strongly oriented towards "deep" structured descriptions that are organized into many levels of detail, and permit "zooming" easily in and out of these levels. One can construct them in a disciplined hierarchical way, statifying a system level by level, or use interlevel transitions and vary the depth as deemed appropriate. Statification can proceed by top-down refinement, bottom-up clust ring, or a mixture of both. Note that the exponential blowup in stat s does not arise here at all, as the option of using orthogonality on any level eliminates the need for explicit consideration of all state combinations.

As a truely simple example of the way statechart levels refine reactive behavior, consider the system whose interface set $\mathbf{E}$ is as in Fig. 6. Its behavioral specification is given on two succesive levels in Figs. 7 and 8. The allowed sequences in Fig. 7 can be described by the regular expression

$$B_1^* \cdot (a \cdot B_2^* \cdot a \cdot B_1^*)^\omega \cup B_1^* \cdot (a \cdot B_2^* \cdot a \cdot B_1^*)^* \cdot (B_1^\omega \cup a \cdot B_2^\omega)$$

where $B_1 = (b \cup c \cup d)$ and $B_2 = (b \cup c \cup e)$. The version in Fig. 8, on the other hand, refines the allowed behavior to the sequences given by the same expression, but with $B_1 = (b \cup bcd)$, and $B_2 = (b \cup bce)$.

This, then, is the formalism we wish to recommend for specifying reactive behavior. It is important to observe that the description can be made to reflect the statifier's personal and natural view of the system's

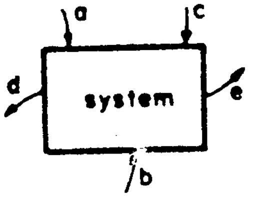
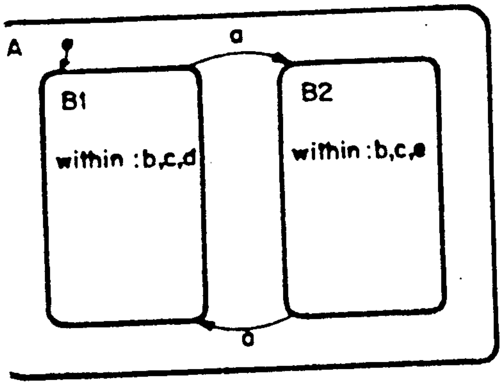


Fig. 6: A reactive system with its interface set



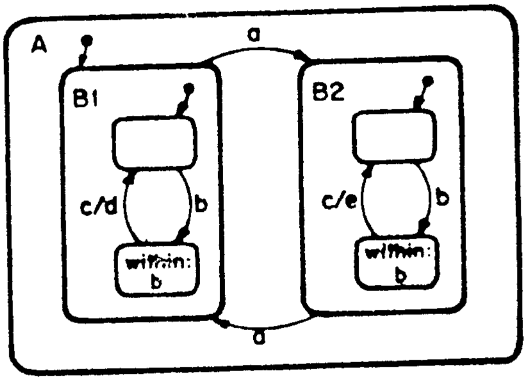


Fig. 7: A statechart

Fig. 8: A refined statechart

behavior, free, if so desired, from implementational details. If lifting a receiver conceptually causes a communication system to enter conversation mode, and this mode entails certain actions, one can say just that, disregarding such questions as which component is responsible for sensing the lifting of the receiver and how the sensed fact is communicated to the others. Of course, the crucial problem here is bringing this conceptual description down to earth; meaning, how does one combine it with the natural implementational process of breaking the system down into its physical parts and their interconnections.

The Magic Square of System Development

Now that we know how to decompose reactive behavior, we can attempt to apply the "problem structure matches system structure" principle. Indeed, if there are no limitations on the implementation at all one can do just that; refine the behavioral specification by adding statechart levels as illustrated above, and then perform the implemenatational refinement to match. While this might sound overly naive, in a pure software system with a sufficiently high-level programming language, the statecharts can be directly encoded into software, with multi-level orthogonality being translated into nested concurrency. This applies to other possible methods for reactive behavior specification, such as Petri nets or languages like CCS.

It is clear, however, that the vast majority of interesting reactive systems feature various preconceived, unavoidable limitations on the structure, distribution, capabilities and interconnections of their implemenational components. Examples include geographical distribution of portions of an airline reservation system, standard components of an avionics system, physical limits on hardware components and their interaction, etc. Even concurrent programs are, in general, not all that pure, as one typically is constrained by a limit on the maximal number of concurrently executing processors. In this way, if the implemenational restricions are to be honored, being overly liberal in the use of orthogonality, for example, can easily cause severe problems in a naive attempt to model the implemenational refinement according to the behavioral one.

What this means is obvious: our recommendation for a method that enables natural behavioral decomposition notwithstanding, the implemantational description has a nasty habit of prescribing, at least in part, its own decomposition, which need not, in general, match our conceived behavioral one.

These observations prepare the ground for a very simple idea: by and large, the development of a reactive system is not a one-dimensional process in which specification and design are two temporally related stages, but rather it is a two dimensional "magic square" in which they play the role of the dimensions themselves. One might label the two dimensions simply "specification" and "design", but we prefer to use those terms as verbs, and so we label the dimensions "behavior" and "implementation". One thus specifies the system's behavior but one designs its implementation. See Fig. 9.

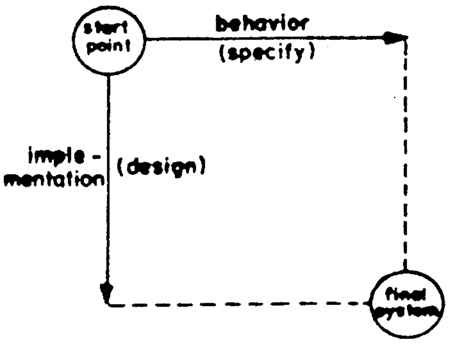


Fig. 9: The magic square

Along both dimensions making progress amounts to supplying more detail, but the two axes involve fundamentally different kinds of detail. Proceeding vertically (downwards), one is bringing the system closer to its final form by supplying more information about its implementation, and proceeding horizontally (rightwards), one is fine-tuning the system's performance by providing more information about its behavior. In either case, and this explains in part our use of the term "magic square", every line or column amounts to a full, stratified, description of the system along one axis, but at a fixed level of detail along the other.

Ideally, the development process starts at the upper leftmost point, with nothing known about the system's intended behavior or its desired implementation, and ends at the lower rightmost point, with everything known about both. The more subtle side of the term "magic square" is rooted in the many possible ways of traversing the square from its initial point to its final one; following any of these correctly should result in the system being fully specified and fully designed.

It seems almost an ac·dent that for the easier systems in our dichotomy, i.e. transformational ones, or reactive ones with no implementational constraints, this process can actually be linearized by, in essence, mapping one dimension onto the other relatively easily as discussed earlier. In our opinion this accident is the heart of many misconceptions and difficulties encountered in the development of complex systems, and to some extent also in pure concurrent programming.

We are aware of the fact that even with concrete formalisms in mind any discussion of such a general model for system development is bound to seem naively idealistic when considered for real-world applications. Nevertheless, we are determined to describe our model in the simplest possible terms, and for that purpose, besides adopting simple statecharts, free of global constraints, for the behavioral axis, we adopt a very simple decomposition method for the vertical, implementational axis.

At level 0 of the vertical axis the system resides as an unspecified entity $\mathbf{M}^{(0)}$, together with its interface set $\mathbf{E}^{(0)}$. Progressing down the vertical axis is characterized by a step of implementational decomposition. For a typical descent from level i to level $i+1$, a subsytem M on level i, with interface set E, might be decomposed into its constituent components $M_1, \ldots, M_n$, with their interface sets $E_1, \ldots, E_n$; see Fig. 10. For this decomposition to be acceptable certain obvious properties must be satisfied. For exampie, each element of E, the interface set of M, must appear in at least one set E_j, or at least be refined into more concrete elements, each of which appears in at least one E_j.

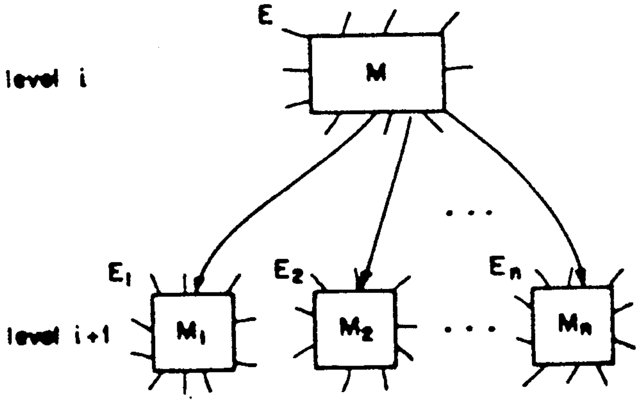


Fig. 10: Implementational decomposition

In addition to interface elements that are external to the whole system, the interface set E_j of M_j may contain additional elements which are external to M_j but internal to M. These elements provide the components with the ability to communicate, synchronize, and influence one another. Thus, each element in $E_j - E$ must be tagged with some indication as to its source or target subsystem(s) from among the others. We shall not go into more detail here so as not to detract attention from the underlying issues. Besides, our presentation of this decomposition method is highly simplified anyway, and many standard methods can be readily adopted for a satisfactory treatment of the implementational axis.

In contrast to making progress down the vertical axis by refining the implementational design of the system, progressing along the horizontal axis refines and structures its behavior, and as discussed above can be thought of as adding levels to the appropriate statecharts. Having reached some fixed vertical level i, one has essentially decided upon a set of implementational modules, say $M_1, \ldots, M_k$, and now proceeding horizontally at this vertical level amounts to statifying each of these. The outcome, of course, is a set $S_1, \ldots, S_k$ of statecharts whose interface sets are simply those of $M_1, \ldots, M_k$.

As a side remark, note that the magic square is not a square at all. First, there is no reason whatsoever for the levels of implementational detail to be equal in number to those of behavioral detail, so that at the very best the development model should be called a magic rectangle. Secondly, and more significantly, the implementational decomposition, even using the simple model above, forms a tree, and one whose branches are not necessarily of equal length, so that the outcome is more like a tree with

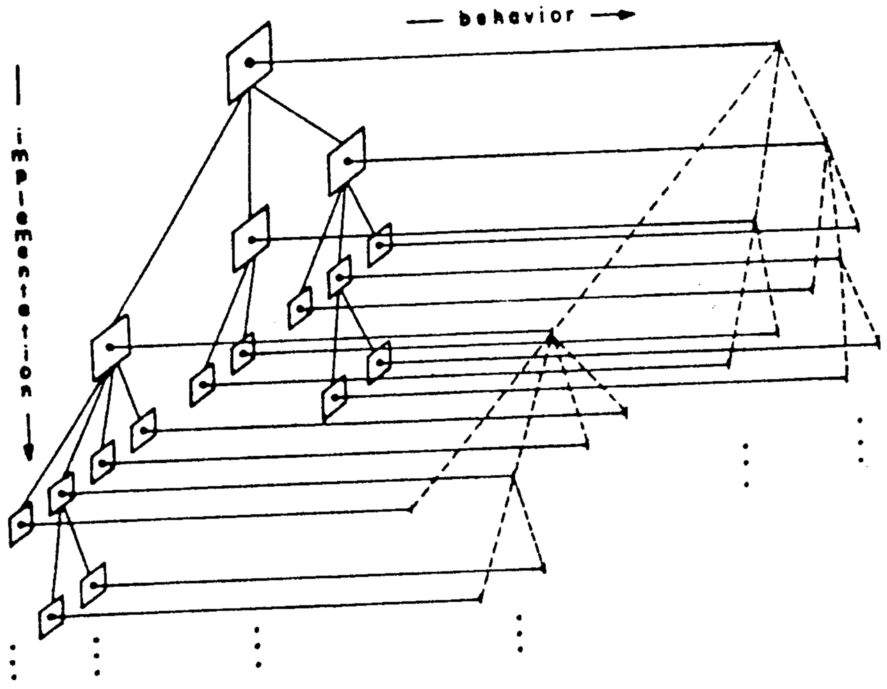


Fig. 11: The magic square as a spiky tree

long spikes; see Fig. 11. Thirdly, and most significantly, for any given node in the implementation tree (=system component) the behavioral description itself, even using simple statecharts on their own, is actually a tree or worse, again with no uniform depth existing either amongst or within each other. For a fixed system, therefore, the actual development creature is far more complicated, rather like a multidimensional tangled tree (and the reader must forgive us for not supplying a picture of one here). Nevertheless, we stick to the term "square", emphasizing its two dominating dimensions.

An observation worth emphasizing is the presence, indeed highly desirable presence, of concurrency along both dimensions of the square. In the implemenational axis concurrency appears as the obvious coexistence of physical entities, usually termed the *parallelism* of system components, and in the behavioral axis concurrency appears as the coexistence of modes of behavior, which in statechart terminology is simply the *orthogonality* of states. Both, either directly or indirectly, can cause simultaneity of activities in the final system. In fact, orthogonality seems to us to be a more natural manifestation of concurrency in the specification than certain suggestions in the literature, such as specifying concurrency by Boolean conjunction.

With the story we have told so far, two obvious ways of traversing the magic square come immediately to mind: the L-shaped all-the-way-down then all-the-way-to-the-right traversal, and its dual. The first corresponds to a practice common in certain kinds of concurrent programs: obtain information as to the number, type and interconnections of available processors, and then specify the behavior of each, which is tantamount to programming them. Whether the programming, which can itself proceed in a stepwise disciplined manner, is regarded here as specification or design is irrelevant; the main point is that one is programming per processor. Actually, one usually has some high level description of the intended behavior in mind even when one proceeds in this L-shaped way, but rather than being used in a rigorous way in the development process it is more often simply referred to at the end for the purpose of verifying the final product against it.

The dual traversal calls for a complete behavioral specification prior to any implementational decomposition, and was hinted at earlier. Neither of these traversals of the magic square can be particularly recommended for complex systems. Actually, neither of them makes essential use of the available two-dimensionality at all, and as a consequence neither requires that behavioral descriptions be projected in a nontrivial way from one vertical level to the next. This kind of behavioral projection, however, is one of the most crucial aspects of our magic square, as we now set out to show.

A Consistency Criterion for The Magic Square

In general, we wish to argue, a healthy development process prescribes some horizontal progress prior to each significant progress made on the vertical level. That is, at vertical level i system component M is given a behavioral specification S of certain depth, that is, extending to some horizontal point. One then decomposes M (or is provided with a decomposition of M) into its subcomponents $M_1, \ldots, M_n$ and somehow specifies the behavior of each, to the *same* horizontal depth, yielding the preliminary $S'_1, \ldots, S'_n$. The S'_j are then refined as discussed above, yielding the final behavioral descriptions $S_1, \ldots, S_n$ of vertical level $i+1$. Of course, this set of behavioral descriptions of the M_j has to be consistent with

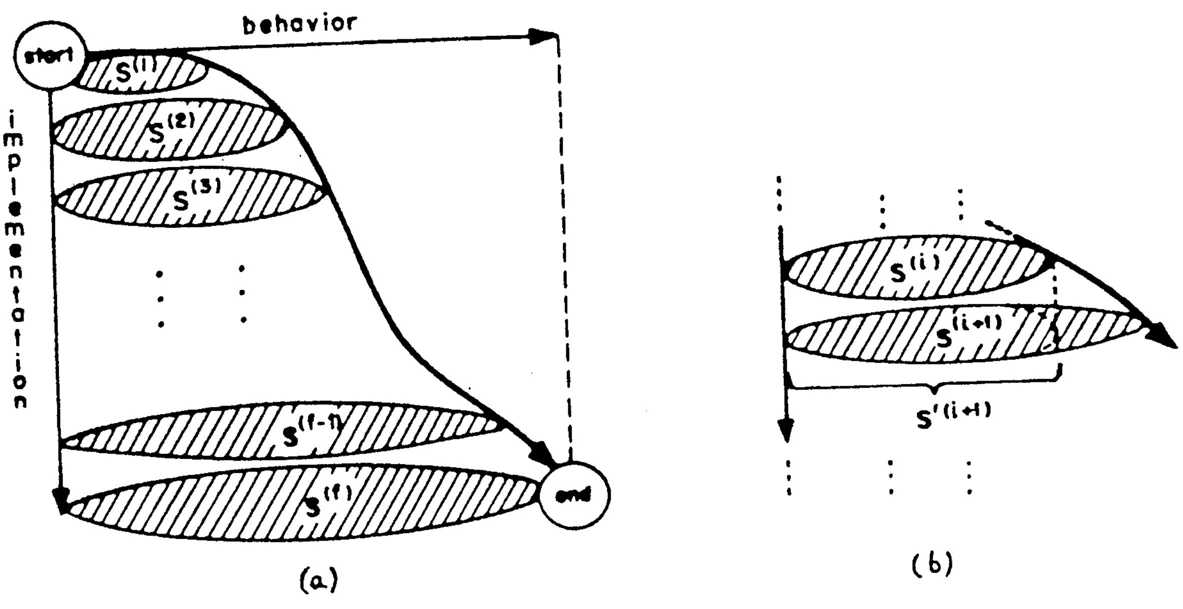


Fig. 12: Traversing the magic square

the behavioral description S of the higher-level system M, and we define the nature of this consistency below. The general progress can thus be schematically described as in Fig. 12, with the progress arrow aiming and hitting the final point corresponding to the fully-specified, fully-designed system. Each shaded area $S^{(i)}$ in Fig. 12 thus represents the collection of behavioral descriptions of all subsystems relevant to level i of the implementation.

At the expense of sounding repetitious we remind the reader of Fig. 11 and its more realistic, hence far more complicated version of our simple "square", meaning that the S of each M on any level is given at the depth of behavioral detail appropriate to it and its "neighbor" subsystems. However, as mentioned earlier, for our purposes the issues can be discussed under the pretensions implicit in Fig. 12.

Three questions come immediately to mind:

(1) What is, or should be, the precise consistency criterion relating $S^{(i)}$ to $S^{(i+1)}$?

(2) Given a satisfactory answer to (1), can one recommend a recipe for obtaining $S^{(i+1)}$ from the decompositions carried out when progressing downwards from level i to level $i+1$, together with $S^{(i)}$?

(3) Whatever the answers to questions (1) and (2) are, can one recommend a "good curve" for traversing the square?

Question (1) is purely technical, and without answering it satisfactorily the whole magic square model collapses. There must be a firm and precisely defined connection between the specified behavior of a portion of the system and that of its constituent components, one which then transcends to become a global connection between the initial and final stages of the entire development process.

The answer to question (1) can be stated informally as follows:

> The external behavior implicit in $\mathbf{S}^{(i)}$ must be equivalent to that implicit in $\mathbf{S}'^{(i+1)}$, the preliminary behavioral description on level $i+1$ which is of the same horizontal detail as $\mathbf{S}^{(i)}$.

This simple answer contains some subtlety, since apart from the haziness of "external", "implicit" and "equivalent", $\mathbf{S}^{(i)}$, in our chosen framework of formalisms, is but a collection of statecharts, one for each component on level i, and $\mathbf{S}'^{(i+1)}$ is a different collection, assoc ated with a different level and different components. Nevertheless, the answer can be made precise quite naturally, as illustrated in Fig. 13.

Take a typical level i component M and its subsystems $M_1, \ldots, M_n$ on level $i+1$. In $\mathbf{S}^{(i)}$ there will be a statechart S for M and in $\mathbf{S}'^{(i+1)}$ statecharts $S'_1, \ldots, S'_n$ for the M_j. As discussed earlier, each pair (M_j, S'_j)

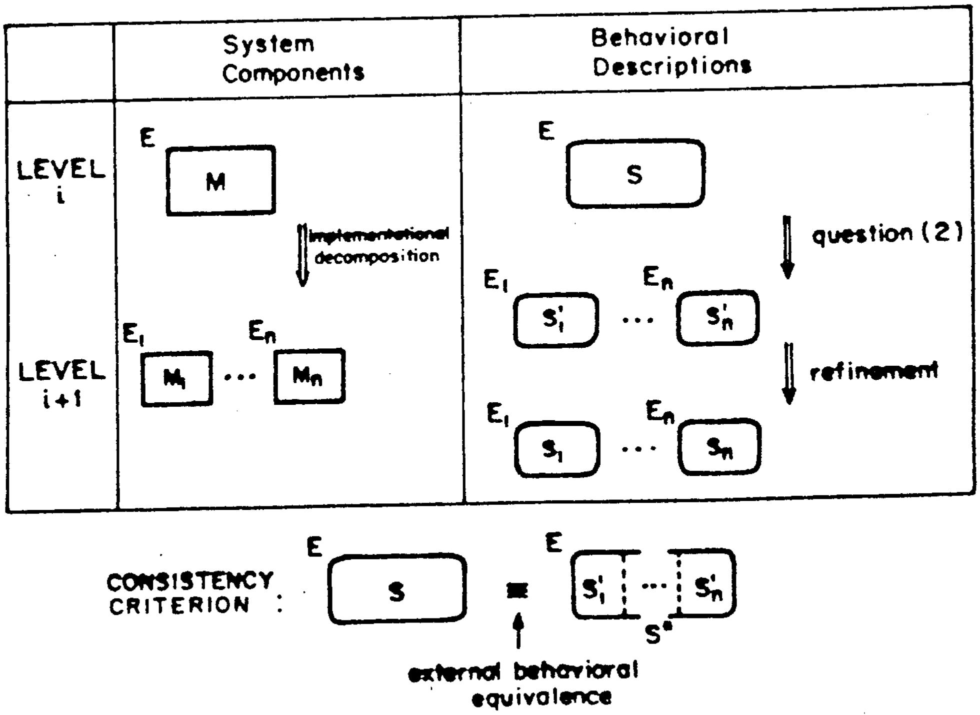


Fig. 13: The consistency criterion for the magic square

has an interface set E_j, only part of which is external to $\{M_1, \ldots, M_n\}$ by corresponding directly to the interface set E of (M, S). For the sake of simplicity let us assume that the elements in E have not been themselves refined in the transition to the E_j, but, as discussed above, each E_j consists of elements in E and possibly some new elements, internal to the set $\{M_1, \ldots, M_n\}$. Now form a new statechart S^* by simply placing $S'_1, \ldots, S'_n$ side by side as orthogonal components. Their interconnections via the internal interface sets $E_j - E$ will be taken care of by the statechart formalism itself. In this way, S^* will be a single statechart whose interface set consists of all interface elements which are not internal to the set $\{M_1, \ldots, M_n\}$, that is, to the union over j of $E_j - (E_j - E)$, which is simply E itself. We now require S and S^*, compatible by virtue of their common interface set E, to be actually ***equivalent***, i.e., to define the same set of sequences over E.

If this equivalence holds when applied to each and every implementational component on level i, we say that $\mathbf{S}^{(i)}$ and $\mathbf{S}'^{(i+1)}$ are equivalent, and, more importantly, that $\mathbf{S}^{(i)}$ and $\mathbf{S}^{(i+1)}$ are ***consistent***. The latter term is justified by the former, together with the fact that $\mathbf{S}^{(i+1)}$, the final behavioral description on implementational level $i+1$, is obtained from $\mathbf{S}'^{(i+1)}$ by adding levels of detail to the statecharts therein, thereby refining the behaviors they define.

In the case where E is indeed refined in making the vertical transition, the notion of equivalence, and accordingly the notion of consistence, has to be appropriately modified to account for the matching of interface elements. Also, if the statecharts are accompanied by a set of additional global constraints $\mathbf{G}$, a possibility mentioned earlier, the refinement of $\mathbf{S}'^{(i+1)}$ to $\mathbf{S}^{(i+1)}$ must adhere to those, and hence might require a separate, specially tailored, consistency criterion, stating, roughly, that $\mathbf{S}'^{(i+1)} \cap \mathbf{G}$ is equivalent to $\mathbf{S}^{(i)} \cap \mathbf{G}$. Furthermore, one might be interested in the possibility of refining, and hence making more concrete, the global constraints too in the process of making vertical progress. This progress will then relativize to the interface sets of each level, just as the statecharts do, resulting in a $\mathbf{G}^{(i)}$ for level i, and, again, the notions of equivalence and consistency will have to change too. These, and numerous other possible complications might surface in specific attempts to use the magic square, but they can be dealt with by extending the basic model in a natural way, and with the underlying principles being the same.

The main consequence of the definition of local consistency between vertical levels is in its transitivity. By this we mean that consistency is propagated down (or up) the vertical axis, resulting in the following fact:

> If a complete development process is carried out in the magic square model, using any desired traversal, while checking local consistency, the final behavioral description of the system, $\mathbf{S}^{(f)}$, is consistent with the initial one, $\mathbf{S}^{(0)}$. In other words, if one constructs the entire system using the "atomic" implementational components prescribed by the final vertical decomposition level f and connects them as prescribed by the interface sets on that level, and if one then convinces oneself that each of these low-level components behaves as prescribed by its behavioral description in $\mathbf{S}^{(f)}$, then the entire system is *correct* with respect to its initial specification $\mathbf{S}^{(0)}$.

This, by the way, should remind the reader of classical methods for program verification, where global correctness follows from the correctness of the consituent modules.

Precisely how one convinces oneself of the behavioral correctness of the atc nic components is of no concern here. It might follow from a manufacturer's documentation, a programmer's verification, or be regulated to mere belief. Just as in any typical verification process, the soundness of one's use of the magic square in this fashion is a doubly-relative concept; it relies on an accepted initial specification, against which one verifies what one has constructed (in this case this is the initial behavioral specification $\mathbf{S}^{(0)}$), and it relies on an "axiomatic" acceptance of the fact that the atomic elements, which constitute the building blocks with which one has carried out that construction, are correctly specified (in this case this amounts to accepting the final vertical level $\mathbf{M}^{(f)}$, $\mathbf{S}^{(f)}$, as is).

At this point, one might be tempted to ask the following question: if the S and S^* appearing at a certain stage in the process (see Fig. 13) are required to be two equivalent descriptions of the same part of the final system, given in the same formalism, and over the same interface set E, why then is S^* not constructed directly? Why was S^* not given as the level i behavioral description of component M, especially since it is already conveniently decomposed in accordance with the decomposition $M_1, \ldots, M_n$ of M? It goes without saying that this would eliminate any necessity for checking the equivalence of behavioral descriptions. The answer, of course, embodies the main issue here: S^* is not necessarily a natural behavioral description of M *because* it is composed according to $M_1, \ldots, M_n$. A good behavioral description of an airline reservation system which uses ten geographically distributed computers and a thousand terminals might be one which decomposes in ways that cut across this physical decomposition, just as a good behavioral description of a VLSI chip need not be given in terms which even mention its breakup into design-related blocks. Whatever the case, the "unnatural" S^*, which consists of the preliminary behavioral descriptions $S'_1, \ldots, S'_n$ of the components $M_1, \ldots, M_n$, has to be prepared somehow; but how? This is precisely question (2) above.

How to Traverse the Magic Square?

We are in no position to present detailed answers to questions (2) and (3), and even less so to claim that we know of answers that can or should be used universally. Quite to the contrary, different kinds of systems present different kinds of problems in behavioral specification, and the entire development process can be regarded as an art, with many facets and many possibilities.

Moreover, more often than not, a complex development effort does not start at the initial point of a totally blank magic square. It usually starts with various portions of the square already filled in. That is, certain system components, and even certain chunks of behavior, are already prescribed to the developer in advance, and the development process must accomodate them as it proceeds. Therefore, as far as question (3) is concerned, we can only say that, when viewed as a function that plots horizontal progress against vertical progress, the development curve should be monotonically increasing (see Fig. 12); it makes very little sense to provide a subcomponent with less behavioral detail than its parent component. However, other than that, and other than observing that the curve should probably be nontrivial (i.e., not L-shaped or its dual), developers should be free to define their own preferred curves for traversing the square.

As to question (2), there is one general point to be made here. Let M be a component on vertical level i, decomposed on level $i+1$ into $M_1, \ldots, M_n$, and let S be the behavioral specification of M. If E and $E_1, \ldots, E_n$ are the interface sets of the components (no refinement in the E's; as above), then one can proceed as follows: for each $1 \leq j \leq n$, start with S itself as a first approximation to S_j, the behavioral specification of M_j on level $i+1$.

Now, clearly S is not a legal statification of S_j in general, since it refers to elements from $E - E_j$. However, this can be fixed as follows. Output elements in $E - E_j$ are simply eliminated, and for each input element e therein one finds a k with $e \in E_k$, and modifies the two copies of S, that of E_k and that of E_j, so that the former, whenever e is sensed, outputs a new item e' to M_j, in whose copy of S each e is replaced by e'. In this way M_j can sense the input event e even though it can be directly sensed only by M_k. The resulting statecharts, call them $S''_1, \ldots, S''_n$, are now legel behavioral specifications of the M_j, in the sense that their orthogonal product is equivalent to S. Moreover, the S''_j are on the same horizontal level of detail as S. These S''_j, therefore, conform to the definition of the preliminary description of $S'^{(i+1)}$, see Fig. 12, and hence, in principle, we have answered question (2). However, we clearly do not recommend that one stop here; the result will be an exponentially growing behavioral specification, not to mention its complete detachment from any naturalness involved in the implementation refinement of M into the M_j. Rather, one should work on the S''_j, simplifying and changing them to reflect the intended behavior of the M_j. In this sense, the only useful role S''_j can play, is that of a starting point leading to the desired S_j, the final behavioral specification of M_j.

Actually, we feel that it is worthwhile to search for good transformations for weeding out portions of S''_j not really relevant to M_j, using the difference between E and E_j, and S itself as directives. Such transformations, certain simple examples of which come immediately to mind, should be required to preserve equivalence of the behavior, modulu the transition from E to E_j.

The equivalence problem for reactive behavioral specifications, and in particular equivalence-preserving transformations of statecharts, seems to be an important area for future research. Satisfactory and useful results in this direction can help turn the magic square from a *model* of system development, which is the way we have tried to portray it here, into a detailed *methodology*, or prescription, a line we have not yet tried to pursue.

Acknowledgement

The contents of this paper owe much to many colleagues and authors. In particular, we have been influenced by several ideas found in the work of M. Alford, L. Lamport and D. Parnas.