Software—Concepts and Tools (1996) 17: 63–69

Software—Concepts and Tools
© Springer-Verlag 1996

Early Fault Detection Tools

Gerard J. Holzmann

Bell Laboratories 2C-521, 700 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
http: //cm.bell-labs.com/who/gerard/, e-mail: gerard@research.bell-labs.com

Abstract. The traditional software development cycle relies mostly on informal methods to capture design errors in its initial phases, and on more rigorous testing methods during the later phases. It is well understood, though, that those bugs that slip through the early design phases tend to cause the most damage to a design. The anomaly of traditional design is therefore that it excels at catching bugs at the worst possible point in a design cycle: at the end.

In this paper we consider what it would take to develop a suite of tools that has the opposite characteristic: excelling at catching bugs at the start, rather than the end of the design cycle. Such *early fault detection* tools differ from standard formal verification techniques in the sense that they must be able to deal with incomplete, informal design specifications, with possibly ill-defined requirements. They do not aim to replace either testing or formal verification techniques, but to complement their strengths.

Proc. 2nd Intern. Workshop. on Tools and Algorithms for the construction and Analysis of Systems, TACAS96, Passau, Germany, March 1996, Lecture Notes in Computer Science, Vol. 1055, pp. 1-13.

Keywords: fault detection, early fault detection, software development, design analysis, message sequence chart

1. Introduction

The goal of this paper is to consider the possibility of developing a suite of tools that can help to improve the reliability of a software design process, specifically for distributed systems, from the very early beginnings of that process. Traditional testing techniques come too late to be of much help in this phase. Formal verification techniques, on the other hand, have the disadvantage that they require a fairly solid insight into the design itself before they can prove to be of value. Users of formal verification tools typically have trouble with two things: (1) producing a completely defined model of the design, when only partial information or insights are available, and (2) formalizing an adequate set of correctness requirements.

Before formal verification and traditional testing techniques become applicable, there is currently a void of tools. Our aim is to consider if this void could be filled. We propose the term *early fault detection* for a tool or technique that can successfully be applied in the earliest phases of a still incomplete and imprecise design, and that can provide the user with some guidance about potential hazards in the design process as it develops.

2. A Paper Problem

To illustrate the main concepts, we will use a simple, made-up, example of a problem to model the interactions in a pseudo-distributed system. We call this *The Paper Problem*. Consider the interactions between authors, editors and referees in handling papers that are submitted for publication. The authors, editors, and referees together form a peer group, where each person from the group could at various times fulfill any one of the three functions mentioned (authoring, editing, refereeing). There is frequently also consultation within the peer group about problems of current interest. The interactions between author, editor and referees can therefore be complex. We will treat the modeling problem here as if it were a true industrial design problem, where we have to discover the intricacies of the situation through a systematic process of refinement. For inspiration, we will consider how design problems of this type, though on a much larger scale, may be handled by an industrial design team.

2.1 Structural View

First we must decide what the main structural components of the new system should be. To make this possible, we draft a very general set of requirements for the new system, as it would operate under ideal circumstances. For our paper problem, one such requirement might look as illustrated in Figure 1.

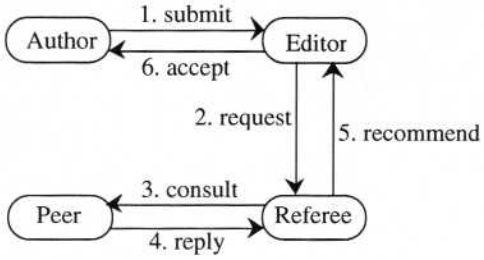

Figure 1. Structural view of the paper problem.

Figure 1 represents the main functional entities from our system: the author, the editor, a single referee and a representative of a peer. One set of possible interactions of these entities is shown. For the time being, we will ignore that the functions can actually overlap, as a referee can take on the functions of an author or a peer, etc.

The labels of the arrows between the structural components suggest a time ordering of typical events in this system. In this case, the referee consults with a peer before returning a recommendation to the editor, and, this being an idealized scenario, the editor informs the author that the paper was accepted.

Many things can throw this simple scenario off course. The referee can refuse to handle the paper or can fail to respond. The author could withdraw the paper, revise it, or become impatient and send additional enquiries to the editor. More to the point, in the case of blind refereeing, the referee might accidentally attempt to consult with the author as a peer, without mentioning the actual reason for doing so. This act could close a dependency cycle in the graph, for instance, when the author decides to delay the peer-response to the referee until the final status of the paper is decided by the editor.

The main purpose of the first design phase, however, is to identify the main building blocks for the system, and for now, we merely need to establish that this goal has probably been reached. In practice, requirements engineers may come up with dozens of sketches of the general flavor of Figure 1, showing different scenarios that illustrate the main lines of the intended operation of a new system. The word *sketch* is to be taken literally: these figures often exist only as pencil sketches on paper or as rough outlines on a white board. They ultimately can reach the status of a figure for reference in a general document for the design rationale, but rarely do they become an integral part of the design process as such.

Our purpose here is to emphasize the very early stages of a design. Our intent is therefore to look very carefully at even the earliest design sketches, to see if they could be amenable to direct tool-based support.

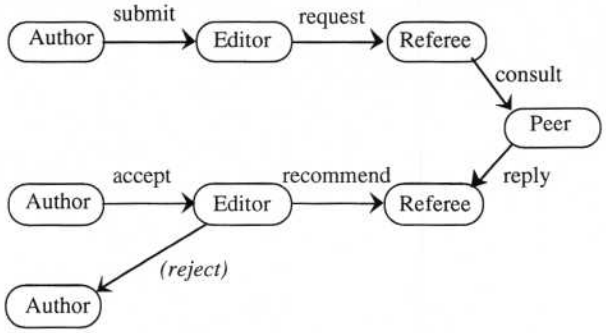

Figure 2. Structural unfolding of an extension of Figure 1.

2.2 Unfolding

A familiar saying is: "You do not fully understand something until you understand it in more than one way." An early fault detection tool, then, can derive one of its strengths from offering a variety of different views of a design while it is under development and still growing. A natural alternate view of the structural view of Figure 1 is obtained by its structural unfolding, as illustrated in Figure 2.

The structural unfolding view, which is reminiscent of the CRC card technique [12], represents the design as a tree, where each path from the root of the tree to a leaf represents a possible scenario. The structural elements are replicated at each node in the tree where they contribute to one of the processing steps. In Figure 2 we have added the possibility of a rejection as the final outcome of the paper submission process. Where Figure 2 differs from the structural unfolding of Figure 1 is indicated in bold. The purely structural view is superior when architectural questions about the system have to be settled. The view obtained by the structural unfolding is better when good insight needs to be obtained about the interdependencies between different, possibly overlapping, scenarios. Since time sequence information is carried in the labels, one way to preserve the information about branching from Figure 2 also in the view of Figure 1 would be to replace the label *6. accept* with *6.1. accept* and to add an arrow from editor to author with label *6.2. reject*. The two views now represent the same design decisions, while emphasizing entirely different aspects of the system.

2.3 Temporal View

Yet another view of the design can be obtained by switching from the structural representation to a purely temporal view using basic message sequence charts (MSCs) [8], as illustrated in Figure 3.

We have again extended the view from Figure 1 slightly to illustrate the differences in strength of the

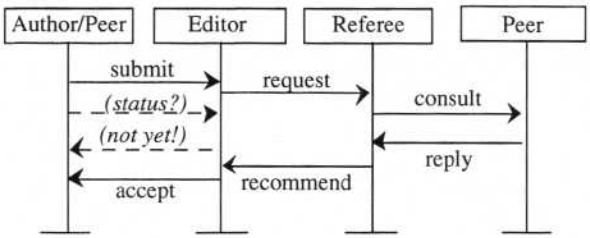

Figure 3. Temporal view of an(other) extension of Figure 1.

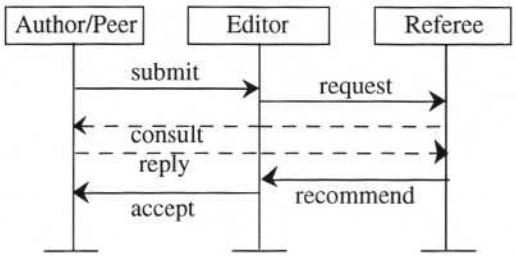

Figure 4. Variation on the scenario in Figure 3.

various views. In this case, we have added (with dashed lines) the possibility of the author interrogating the editor about the status of the paper. The disadvantage of the MSC view is that it does not offer natural support for branching scenarios. To represent the view from Figure 2, therefore, would normally call for two different MSC representations. Such scenarios are often referred to as *sunny day* and *rainy day* scenarios. Figure 3, then, represents a sunny day scenario.

The representation from Figure 3 directly lends itself to some forms of analysis, as illustrated in more detail in a related paper [2] and briefly summarized below.

3. Building EFD Tools

3.1 A First EFD Tool: MSC

The three views we have illustrated so far all represent basically the same information about a design, though in different forms, emphasizing different aspects. A single design tool could easily offer all three views, allowing the user to switch between them at will and revise or add to the growing design from whichever view is most convenient. This (so far) hypothetical tool is the first example of an early fault detection, or EFD, tool. Work is currently underway to extend our MSC tool [2] into this type of tool.

At the time of writing, the MSC tool can represent only the temporal view, as in Figure 3. For designers, the MSC tool, when used in this mode, competes with, and is compared with, pencil-on-paper sketching, despite the additional advantages that the tool may offer for analysis and ease of maintenance. Even on this minimal score, the tool is easily found to be of great value to a designer pressed for time: it takes just twelve mouse actions for the user to define the complete scenario shown in Figure 3 (four mouse actions to define the processes, eight more to add the eight messages), and optionally some typing to override default names assigned to the components. Skilled or unskilled users alike can enter the scenario in under one minute of user time. Pencil and paper will fare no better, especially for larger charts; many word-processing tools will do substantially worse.

Feedback. After reading the scenario, the MSC tool issues warnings for three potential race conditions. The first is a race between the sending of the request message and the arrival of the *status* query from the author. In this case it is harmless, but note that the author could also decide to send a message to *withdraw* or to revise his contribution. The editor may have to take different actions, depending on the outcome of the race. There is also a race between the arrival of the *status* query from the author and the arrival of the *recommend message* from the referee, and, finally, a race between the arrival of *recommend* and the sending of *not yet* at the editor process. Presumably, also a different response from the editor would be called for, depending on the outcome of that race.

If we merge the peer process and the author process into one, to emphasize that the author may be the peer chosen for consultation by the referee in blind refereeing, the diagram from Figure 4 results.

If, further, we would want to define a behavior for the author where the response to the referee's consultation is delayed until after the reception of an *accept* or *reject* message from the editor, we immediately find that this variation cannot be specified without at least one of the message arrows flowing upward in the diagram—against the direction of time. This scenario would create a causal wait cycle that, if it can be specified at all (the MSC tool forbids the creation of message arrows that tilt upwards), can easily be detected and reported as a potential error.

Design Process. Once the design grows, it can be expected that the architectural view illustrated in Figure 1 will stabilize first. The main structural components of the system are identified and can be refined. Some components may be decomposed and further detailed, leading to related changes in the structural unfoldings and in the temporal view of the MSCs. It can also be expected that, as the design matures, the temporal view will grow both in importance and in complexity. Large libraries of related scenarios are likely to be built, and keeping track of the interdependencies between them becomes an overriding concern.

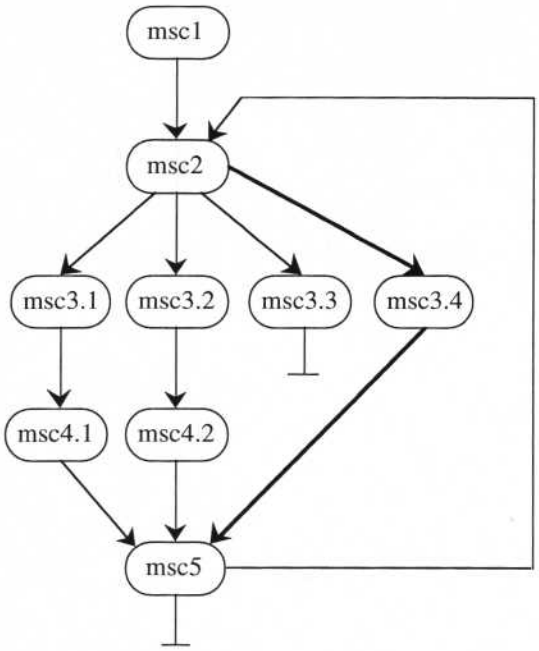

Figure 5. Interdependencies of scenario fragments.

The interdependencies between (perhaps only fragments of) scenarios can be shown graphically in a natural way. Figure 5 shows such a graph, which is based on a similar graph from a real design.

The view is similar to the unfolded structural view from Figure 2, but there are important differences. First, the nodes in the graph from Figure 2 represented structural elements, e.g. designated logical or physical processes in the distributed system to be. In Figure 5 the nodes represent scenarios: interaction sequences in which *all* structural components of the system can in principle participate. The first node in Figure 5, labeled *msc1*, for instance, can represent the first series of steps in a call setup procedure for a telephone call. A subsequent processing step, *msc2*, representing, say, call routing, could have four different outcomes. The call might have to be rejected (*msc3.3*), or it could proceed in various ways, depending on the specific call features that have been invoked.

As indicated, the meta-view need not be, and in general will not be, acyclic.

Also indicated in Figure 5, with bold lines, is a possible *traversal* of the graph from the root to one of the two possible termination points. The graph traversal identifies one possible complete MSC scenario that can be constructed from the fragments *msc1* ... *msc5*.

3.2 A Second EFD Tool: POGA

The graph shown in Figure 5 has a well-defined meaning and structure. There is a host of algorithms that can be applied to a directed graph to provide feedback to the user about its properties. Many graphical tools already exist or are in preparation for working with generic graphs of this type, e.g., [3]. As part of the investigation of early fault detection tools, we have built a similar tool which ties in directly with several others, as will be discussed below.

The tool, called POGA (Pictures of Graph Algorithms), consists of a generic graphical interface, written in about 1200 lines of Tcl/Tk [9], a background processor called Euler, written in about 850 lines of ANSI C, and links to a number of existing tools, such as Dot for doing graph layout [4], and MSC [2], which has its own background processors for finding race conditions and causal conflicts in sample scenarios.

The background tool Euler provides access to a collection of generic graph algorithms, such as the computation of shortest paths, strongly connected components, or the roots and leaves of a graph. The Euler tool can also drive a visualization of generic search algorithms, such as depth-first and breadth-first searches starting at a user-selected initial node in the graph. As the name suggests, it is likely to be extended further with algorithms for computing covering paths, e.g., Euler tours, to facilitate the construction of test suites.

The view presented by POGA matches what is shown in Figure 5. The user can interactively select a path through the graph to indicate scenarios that require inspection. The user iteratively selects nodes from the graph along the desired path. Whenever two nodes are not directly adjacent, the background process, which encapsulates knowledge of graph algorithms, will compute the shortest path between them. The path itself is highlighted on the screen in much the same fashion as shown in Figure 5 (on color displays, the path is colored red).

POGA derives much of its utility from the links with other tools that it exploits, such as Dot for graph layout and Euler for generic graph algorithms. We will consider three other important links in the next section.

3.3 Tool Integration

Linking POGA with MSC. At the request of the user, a path through a graph that is selected in POGA can be expanded into a message sequence scenario using the reference links stored with the nodes. POGA can then call MSC as an independent background process to allow the user to inspect the specific scenario in more detail. POGA allows the user to provide scenario information in two ways:

- In the *nodes*, as symbolic references to files that contain scenario fragments in ITU standard form, or

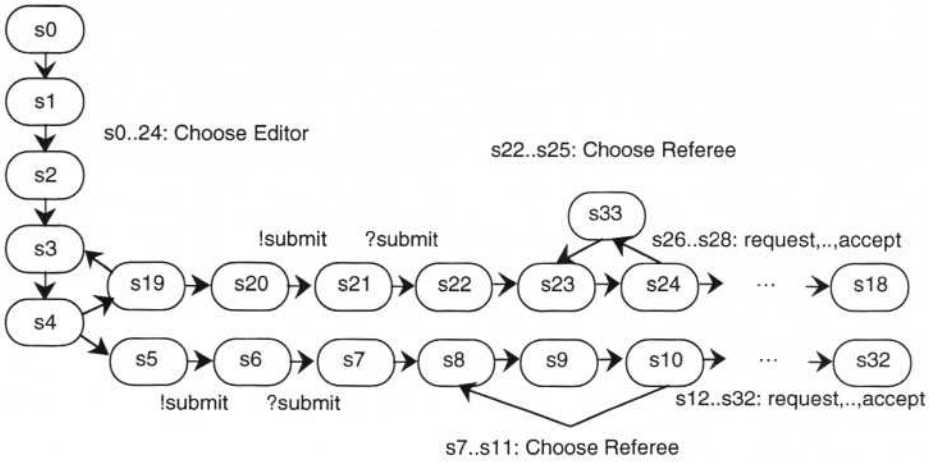

Figure 6. POGA view of a state space generated by SPIN (partial).

- On the *edges*, in a comment field that contains an event description, such as an input or an output action, again in ITU standard form.

At the request of the user, the tool can build a scenario from the fragments that are specified through the nodes and edges along the selected path and start up the MSC tool in the background for independent processing of the scenario thus created.

The MSC tool can be used to modify the scenario, to repair faults, and update the scenario files without disturbing or modifying the view of the interdependencies offered by POGA.

The view offered by POGA can readily be extended to support hierarchical views of a still larger structured collection of scenarios as well. Each node can then represent either a scenario fragment or a subgraph. By double-clicking a node, the user brings up either MSC for a closer view of the scenario fragment stored at the node, or a second copy of POGA for those nodes that point to subgraphs of scenarios. Scenarios can be nested to arbitrary depth in this fashion, and the design space can be navigated with relative ease.

Linking with SPIN. Early fault detection tools, by their very nature, do not allow for a very thorough verification of a design. Once the early design phases have been completed and both the basic structure and the correctness requirements for the design have been settled, it becomes possible to build precise *verification models*. Critical design requirements can then be proven rigorously with specialized tools, such as the model checker SPIN [6]. Of course, it is attractive if the early fault detection tools can somehow be integrated with the formal verification methods.

To explore this integration, we have built an experimental interface between POGA and the verifiers that are generated by SPIN. With this interface, POGA can be used to visualize the reachable state space of small verification problems and can offer visual feedback on the precise effect of various types of search algorithms, for instance, SPIN's partial order reduction techniques [7].

Information on actions, input, and output events can be made available on the edges of the graph by postprocessing the information generated by SPIN with the help of a small Awk filter [1].

The example shown in Figure 6 is part of the reachable state space for a SPIN verification model of our *Paper Problem*, for simplicity restricted to the *sunny day* scenario, in which the author will not query the status of the submitted paper and the referee will not consult with a peer before returning a recommendation to the editor. The state numbers assigned by SPIN indicate the depth-first search order in which the state space is explored.

Provided with this graph, POGA again offers the user the choice of selecting an arbitrary path, which can then be converted into a message sequence chart in ITU standard form [8] and handed off to MSC as before. The reachability graph can also be inspected for the presence of strongly connected components and the like.

The verification model that generated the graph from Figure 6 is 45 lines of PROMELA [6]. A slightly more sophisticated version of 59 lines allows for random peer consultation by referees and allows the author to defer responding to peer consultations until after receiving final word on the status of a paper. We can now attempt to prove the benign liveness property, expressed in the syntax of Linear Temporal Logic [10], that each paper submitted is eventually either accepted or rejected:

$$\Box\,(\mathit{submitted} \rightarrow \Diamond\,(\mathit{accepted} \vee \mathit{rejected})).$$

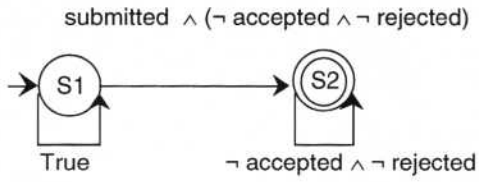

Figure 7. SPIN's never claim for verifying the paper problem.

In SPIN we can attempt to verify this desirable property by claiming that its negation (a decidedly undesirable property) cannot occur. This negated property is formalized as the language accepted by a two-state Büchi automaton shown in Figure 7.

A Büchi automaton is an automaton over infinite words, known as a never claim in PROMELA parlance. (The automaton is generated automatically by SPIN from the LTL formula using the algorithm described in [5].) The initial state of the automaton is *S1*. The transitions of the automaton are labeled with the state properties that must be true for the transition to be enabled. The automaton, then, can remain in state *S1* for an arbitrary length of time. Whenever the property

$$(submitted \wedge (\neg\, accepted \wedge \neg\, rejected)$$

becomes true, the automaton can nondeterministically choose to switch to the accepting state *S2*, where it can remain infinitely long only if the property

$$(\neg\, accepted \wedge \neg\, rejected)$$

remains true infinitely long. For the extended verification model, SPIN will report a match of this undesirable property within a fraction of a second and thus prove that the original liveness requirement can be violated.

Despite the ease with which the potential flaw in the interactions of authors, editors and referees can be found with a formal verification model, the visual formalism of the message sequence charts can be considered superior in this case. In the MSC tool, it is simply impossible to specify a scenario that could include a causal wait cycle, and hence the system designer may be alerted at an early stage to the possible side-effects of some design decisions. More subtle correctness properties, however, can easily escape attention and will require more general error checking capabilities.

Linking with Case Tools. The tools described so far can, in principle, span the early phases of the design process. There already exist several competitive design tools that target the later phases of design, for instance, CASE tools such as OBJECTIME [11]. Just as it is desirable to establish links between formal verification tools and the early fault detection tools MSC and POGA, it can be beneficial to establish a smooth connection with existing CASE tools.

Links can be established in several ways.

- We can convert a SPIN verification model into, for instance, an initial OBJECTIME model to provide a sound starting point for the development of the final code.
- It is also possible to derive verification models from OBJECTIME models, at various stages of the development, to make sure that essential design properties are correctly preserved throughout the design.
- It can also be attractive to use a library of scenarios (in ITU standard form) and synthesize state machine models, one for each process in the system, that captures all the behavior specified. The synthesis procedure would then have to rely on a judicious use of state names (part of the ITU recommendation) to indicate common points of reference in the various scenarios. Note that each separate scenario defines a path through the state machine of each process that is part of the system. The state machines generated in this way can be used either as an initial verification model, to be fed into SPIN, or as an initial OBJECTIME model, to serve as the starting point of further development.
- Simulation trails produced by OBJECTIME can be converted into ITU standard format and provided to the MSC tool for analysis. If race conditions are detected, variations of the scenario can be fed back into OBJECTIME to make sure that the original model is not sensitive to the ordering of events that could be involved in the race.

We are actively exploring all the above options.

4. Conclusions

The initial interest in the early fault detection tools described in this paper has surprised even their authors. We can speculate that the tools derive their appeal not only because of the new checking capabilities they offer, but also because they provide a means to edit and maintain important design documents online as formal objects. The availability of the new tools makes it unnecessary to maintain online representations of scenarios and their dependencies in a fragile word-processing tool format. Conformance to the ITU standards is an attractive bonus, though not considered essential by many of the tool users.

At the time of writing, the MSC tool is in active use and the POGA tool is in a prototyping stage. At least one CASE tool provider is in the process of

extending their tool so that a direct connection can be made with these early fault detection tools.

Clearly, though, both MSC and POGA are only initial attempts to develop early fault detection tools. There can be a suite of similar tools, each offering different views of (aspects of) the design process. Perhaps the best result we can hope for would be that an improved understanding of the paradigm of early fault detection will help us to render the tools discussed in this paper obsolete.

Acknowledgements: I am grateful to Rajeev Alur, Doron Peled, Brian Kernighan, and Mihalis Yannakakis for many helpful discussions on the topic of this paper. The development of the early fault detection tools is a collaborative effort to which all the above have made critical contributions.

References

1. Aho AV, Kernighan BW, Weinberger PJ (1988) The AWK Programming Language. Addison-Wesley
2. Alur R, Holzmann GJ, Peled D (March 1966) An Analyzer for Message Sequence Charts. Proc. TACAS96, Passau, Germany, Lecture Notes in Computer Science, Vol. 1055, 35–48
3. Berry J, et al. (June 1955) Link: A Combinatorics and Graph Theory Workbench for Applications and Research. DIMACS, Technical Report, 95–15
4. Gasner ER, et al. (May 1993) A Technique for Drawing Directed Graphs. IEEE Trans. on Software Engineering, Vol. 19, No. 3, 214–230
5. Gerth R, et al. (1995) Simple on-the-fly Automatic Verification of Linear Temporal Logic. PSTV95, Protocol Specification Testing and Verification, Warsaw, Poland, Chapman & Hall, Germany, 173–184
6. Holzmann GJ (1991) Design and Validation of Computer Protocols. Prentice Hall, Software Series
7. Holzmann GJ, Peled D (1994) An Improvement in Formal Verification. Proc. 7th Int. Conf. on Formal Description Techniques, Berne, Switzerland, 177–194
8. ITU-T (previously CCITT) (March 1993) Criteria for the Use and Applicability of Formal Description Techniques. Recommendation Z.120, Message Sequence Chart (MSC), 35 pgs.
9. Ousterhout J (1994) Tcl and the Tk Toolkit. Addison-Wesley
10. Pnueli A (1977) The Temporal Logic of Programs. Proc. of the 18th IEEE Symp. on Foundation of Computer Science, 46–57
11. Selic B, Gullekson G, Ward PT (1994) Real-time Object-oriented Modeling. Wiley, New York
12. Wilkinson NM (1995) Using CRC Cards: An Informal Approach to Object Oriented Development. SIGS Books, New York, Advances in Object Technology