5 Conclusions

There are many obstacles to be overcome before formal methods are accepted as an integral part of the industrial design process. Many papers have been written about the nature of these obstacles. It can be argued convincingly that some of the objections that have been raised are based on misconceptions, or mere myths, about formal methods, e.g., [2]. But beyond these phases of "anger" and "denial," we may have reached a point in the development of formal methods where we "accept" the blame and develop new methods that form a better match with existing practice. This paper reports on our attempts to do so.

One of the objectives of the early fault detection tools described here and in [1, 4] is to ease the transition towards the adoption of more formal design techniques, by accepting that design choices that are made in the early phases of a design are tentative and deliberately incomplete. Demanding that they be rigorous and complete at that stage of the design would be counterproductive. We have argued that this potential handicap can be turned into an advantage.

Acknowledgements: The work on early fault detection tools started with MSC, which was built in a collaborative effort with my colleagues Rajeev Alur, Brian Kernighan, Bob Kurshan, Doron Peled, and Mihalis Yannakakis, and with invaluable feedback from a superb team of requirements engineers: Margaret Eng, Steve Knittel, Margaret Redberg, and Victor Mendoza-Grado.

References

1. Alur, R., Holzmann, G.J., Peled, D.: An analyzer for message sequence charts. *LNCS* **1055**, Springer, (1996), 35–48.
2. Bowen, J., Hinchey, M.G.: Seven more myths of formal methods. *IEEE Software*, **12**(4), (July 1995), 34–41.
3. Holzmann, G.J.: *Design and validation of computer protocols.* Prentice Hall, Software Series, 1991.
4. Holzmann, G.J.: Early fault detection tools. *LNCS* **1055**, Springer, (1996), 1–13.
5. *ITU-T Recommendation Z.120, Message Sequence Chart (MSC)*, March 1993. (MSC96: **http://www.win.tue.nl/win/cs/fm/sjouke/msc.html**)
6. Kernighan, B.W., Ritchie, D.M.: *The C Programming Language.* Prentice Hall, 2nd Ed. 1988.
7. Lee, D., Yannakakis, M.: Principles and Methods for Testing Finite State Machines. *The Proceedings of the IEEE*, August 1996.
8. Koutsofios, E., North, S.C.: *Drawing Graphs with Dot.* Technical Memorandum, Bell Laboratories, 1991.
9. Leue S., Ladkin, P., Implementing and Verifying Scenario-Based Specifications Using Promela/SPIN. *Proceedings of the 2nd Spin Workshop*, Rutgers University, August 5, 1996.
10. Ousterhout, J.: *Tcl and the Tk toolkit.* Addison-Wesley, 1994.
11. Selic, B., Gullekson, G., Ward, P.T.: *Real-time object-oriented modeling.* Wiley, New York, 1994.

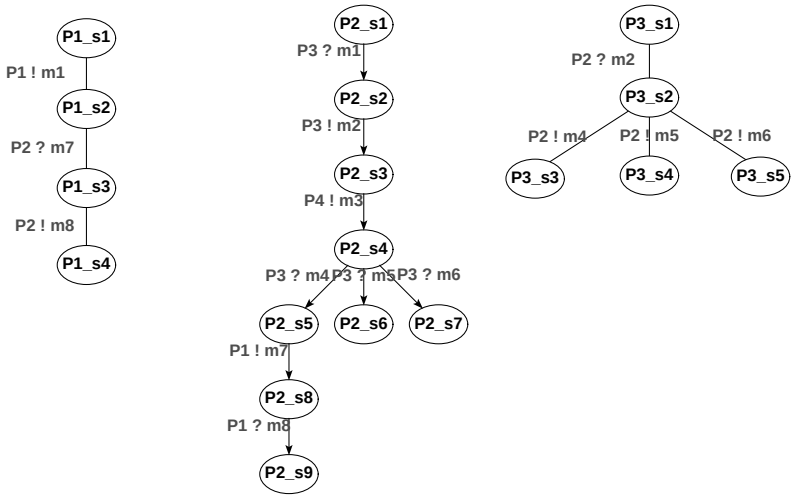


Fig. 8. Three Automata, Derived Manually From The Scenarios Of Figure 7.

to the final product. To complete the design process, it is therefore reasonable to consider if the original design requirements expressed in message sequence charts can be translated into test sequences for the final unit and integration tests of the implemented system.

Unit Tests. For a unit test, one process from the system is selected and marked as the system under test. The inputs to this system can be provided by the test controller, following the scenarios extracted from the POGA dependency graphs, using standard test generation techniques securing maximal coverage of critical test features. By comparing the type, destination, and parameter values generated by the system under test against the test scenarios, a pass or fail verdict can be given.

Integration Tests. In an integration test, only the messages from the environment of the composite system are provided by the test controller, and all other message flows can again be checked against the requirements. Intermediate forms of testing are also feasible: controlling some subset of the processes, and monitoring the others for their responses to the stimuli provided. In collaboration with the first MSC and POGA users, we are exploring which test generation method could be most effective. Based on the finite state machines that can be extracted from the POGA graphs, existing test sequence generation algorithms (e.g., [7]) should be applicable.

The result of "hiding" the eight error handling edges from Figure 5 is shown in Figure 7.

The root node **S1** now shows a subtree of just four nodes. Node **S2** now has three normal successors, and all branch points satisfy the assumptions we have made. There are two additional graph components that can now clearly seen to be reachable only after the execution of one or more error handling edges.

The new view of the graph in Figure 7 raises a suspicion that there could be an error in the original graph. The intention was likely that the choices that we have labeled **a** and **b** can be taken in arbitrary order. The graph shows, however, that there are also spurious paths in the specification: the sequence of edges **a;e**, for instance, can be repeated up to four times in succession. Seeing the new view in Figure 7, the designers of the system depicted here confirmed the anomaly and also spotted one other inconsistency in the design.

The above confirms that a strong support for different views of a design in evolution, even if all views are formally equivalent, can be a powerful design aid. In [4] this observation lead us to propose supporting different views of message sequence charts within the MSC tool: both structural and temporal views. In the POGA tool it leads to the graph structuring techniques illustrated by Figures 5 and 7.

Deriving Automata. From the four nodes in the left-most subgraph in Figure 7, specifying normal processing, it is easy to derive state machine information, either mechanically or manually. As an example, the finite state machines for the first three processes **P1**, **P2** and **P3** are shown in Figure 8.

The initial state of each process corresponds to node **S1** in Figures 5 and 7. In this initial state, process **P1**, shown on the left in Figure 8, holds the *virtual token* to initiate the execution. It does so by sending a message **m1** to process **P2**. Process **P2** receives this message and forwards it as **m2** to process **P3**, and as **m3** to process **P4** (not shown). After all these messages have been sent and received, execution has reached node **S2** in Figures 5 and 7. At this point, process **P3**, shown on the right-hand side of the figure, holds the virtual token, and it is up to that process to resolve the branch condition. It can do so in three different ways, and for each alternative it will send a different message to process **P2**.

4.2 From Automata to Code to Scenarios: Testing.

Scenarios start their life as documentation aids for design requirements. As the design progresses, these requirements can become more detailed, and the collection of requirements becomes more complete. As we have shown in the previous subsection, the requirements specifications may be used to extract even state machine models. These state machine models can in turn evolve into code: the final implemented system. Several commercial CASE tools support the state machine paradigm for code development (e.g., [11]).

The original requirements, once accepted and stabilized, should of course remain valid throughout the entire design cycle. They should be expected to apply

they do not change state.

Unfortunately, the MSC scenario that is linked to node **S3**, the fourth and last successor node of **S2**, specifies an independent send action by process **P4**. Node **S3** also has the largest subtree in the dependency graph of Figure 5, so it appears to be an important branch direction. This seems to spoil the argument we just carefully built, stating that a global choice in the dependency graphs can reasonably be expected to be resolved by a single process in the system. A closer inspection of the scenarios that were specified in the graph reveals, however, that the independent action in the scenario linked to node **S3** is an error recovery action, that is meant to fire only after a timeout (i.e., if for some reason process **P2** fails to make the choice between the other three successor). This error recovery action is also attached to seven other nodes in the graph (**S3**, **S4**, **S5**, **S6**, **S7**, **S8**, and **S9**), contributing to its apparent complexity, and in a sense disguising its true structure.

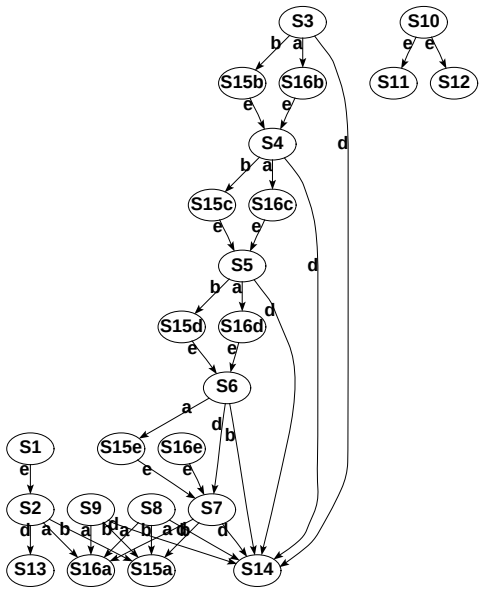


Fig. 7. The POGA Graph From Figure 5, With Error Handling Edges Hidden.

At this point it is a reasonable question to ask what the structure of the dependency graph would be in the absence of the error recovery actions, i.e., for normal executions. Can we make this structure visible? It turns out to be easy to do. Based on this example, we added an option to the POGA tool to optionally *hide* specific types of edges from the graph, and to ask the background layout tool (in our case **dot**) to redraw the graph, as if the hidden edges did not exist.

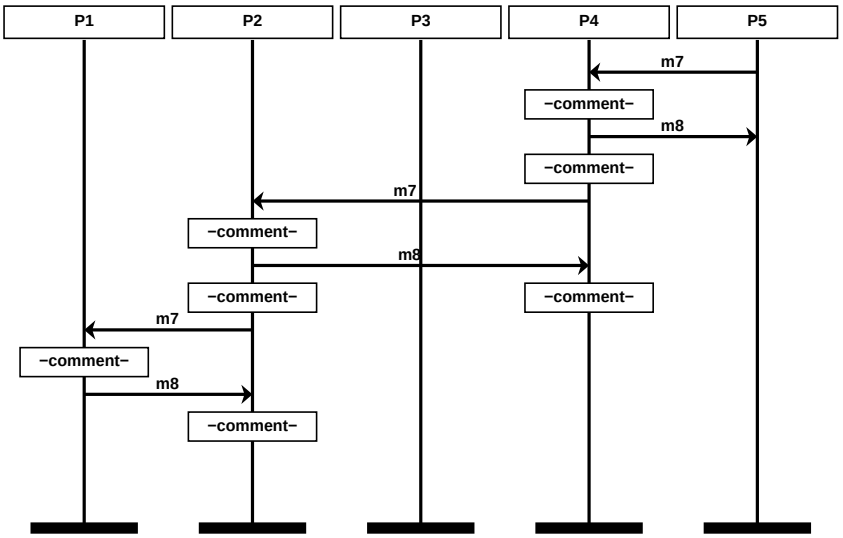


Fig. 6. MSC Link from Node **S10** in Figure 5.

condition, and initiate the execution that corresponds to one of the outgoing edges of the current node of the POGA graph.

A Reality Check. The graph from Figure 5 is a dependency graph that was produced in an industrial trial of the MSC and POGA tools in our company. (We have changed all label and node names.) We will use it here as a quick reality check for the above suppositions.

With one exception (node **S11** at the bottom of the graph) all nodes in this graph contain a link to an MSC scenario fragment. There are no subgraphs. Figure 6 shows an arbitrary one of these scenarios, the one that is linked to node **S10**. (The names of processes and messages are again changed, and the comments have been omitted.) There are five concurrent processes in all scenarios.

The root of the dependency graph, node **S1** at the top of Figure 5, has only one successor. This successor, node **S2**, has four outgoing edges to **S3**, **S16a**, **S15a**, and **S13**. Is the choice between these four branches made by a single process or by all processes in parallel?

In three of the four successor nodes (**S16a**, **S15a**, and **S13**) process **P3** decides the outcome of the branching. It will send one of three different messages to indicate its choice. The messages are all sent to process **P2**, and from there they may be forwarded to some of the other processes to influence also their decisions. The remaining processes are unaffected by the choice made here, and

tracking, and analysis of early design decisions. The tool does not attempt to force greater rigor than the designer can provide at this point in the design process. It is meant to be a characteristic of all early fault detection tools that, despite the apparent lack of formality and rigor, the tools can still provide useful feedback to the user about potential problems in the design as it evolves.

The framework that is sketched above for scenario based design allows naturally for extension towards more rigorous formal specification of a design in terms of labeled transition systems, or finite state machines [3], by exploiting the information that is provided in the MSC dependency graphs maintained by the POGA tool. In effect, while working with graphs in POGA, and time sequence charts in MSC, the designer is unwittingly defining series of communicating finite state machines, that may, in a later design phase, be verified mechanically for their desired or undesired properties. To complete a formalization into state machines, the designer will have to provide some additional information, for instance about the underlying systems architecture (e.g., synchronous or asynchronous communications, the number and type of message buffers, lossy or ideal channels, and the like).

While the additional information is unavailable, either some reasonable defaults can be used for analysis purposes, or the user can be allowed to select from a short list of standard choices. The latter is the choice that was made in the MSC tool. Once the design is nearing completion, the selections can be formalized more explicitly, so that more aggressive forms of verification become possible, e.g., as supported by the SPIN model checker [3].

4.1 From Scenarios to Automata.

We have suggested above that there is likely to be a mapping from scenario dependency graphs to standard state machines (cf. also [9]). Let us consider what would be involved to establish the existence of such a mapping, and, if one exists, to extract it from the graphs.

To see what the potential problem here is, note that MSC dependency graphs, like High-level MSCs, formalize the actions of *multiple* asynchronous processes in a distributed system. The concurrency is not apparent in the dependency graph as such, but is hidden in its nodes. When a branch point is reached, we will suppose that one of two cases always applies:

- All processes evaluate the branch condition individually, and are known to arrive at matching results. For this to work, we must not only be able to guarantee that each process can be provided with consistent information needed to evaluate the branch condition, we must also be able to guarantee that all processes are somehow synchronized to reach the branch point at the same time (unless this can be shown to be irrelevant to the consistent evaluation of the branch condition).
- One process can be considered to have a virtual *token* to proceed, and the other processes wait to respond to its choice at the branch point. The process with the "token" can make a purely local choice. It can evaluate a local

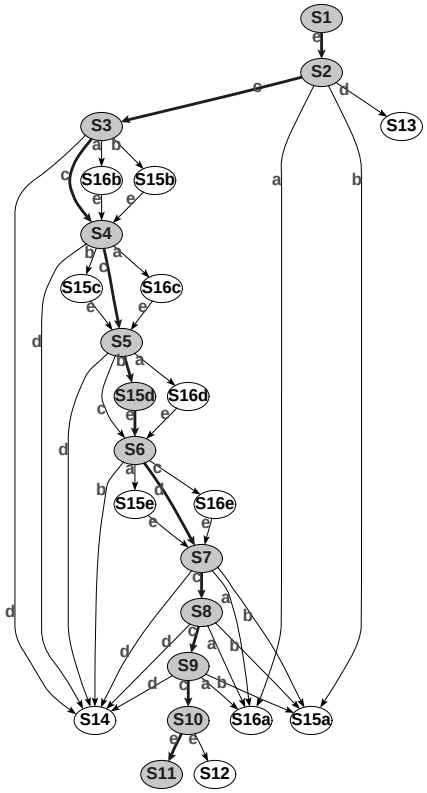


Fig. 5. POGA Path Through An MSC Dependency Graph.

and contain additional constraints, e.g., about the connectivity of the graph, or at which sides of a node incoming or outgoing edges (called *flow lines*) may be connected. High-level MSCs also allow for a richer set of symbols, and a richer semantics. Special symbols can be used to indicate, for instance, branch points, start and termination nodes, and parallel composition of MSC fragments. There is no strict reason to prefer dependency graphs as supported by the POGA tool over High-level MSCs as described in [5]. When the new draft is adopted, POGA will likely be modified to support the standardized form for these graphs.

4 The Next Step

Together, the tools MSC and POGA provide an intuitive, and attractive access to the database that documents an initial design. The tools can be understood quickly, and they can be used productively by designers virtually from the first day of a new design cycle. The anticipated use of the tools is the documentation,

conditional branching or iteration. This constraint forces the designer to specify many different variations of each basic scenario, one for each combination of the conditions that may be encountered. This approach breaks down when the number of scenarios increases to several hundreds or thousands of sets of similar scenario fragments.

To structure a library of scenarios, the designers can take each basic scenario and split it, at the potential branch points, into several smaller fragments. The common parts of a set of alternatives can now be specified once, and the relation between the various segments can be indicated with a flow graph, much like the one shown in Figure 4. We will call such graphs *MSC dependency graphs*. The graph editing tool POGA has proven a valuable aid to construct, and to manipulate precisely these types of graphs.

MSC Dependency Graphs. A node in an MSC dependency graph can represent a named MSC scenario fragment. An edge connecting two nodes represents a possible catenation of the fragments that correspond to the source and the target node. When a node has two outgoing edges, there are two variations of scenarios sequences. The condition that must be satisfied to select one of the successor nodes can be explicitly named or it can remain unnamed. If unnamed the choice represented is interpreted to be non-deterministic. The condition name can initially be an informal descriptive string. In a later design phase this string can be replaced with a more formally expressed condition, perhaps given in the target programming language for the system.

By selecting a node in an MSC dependency graph, the user can request the corresponding scenario fragment to be displayed. To do so, POGA can start an independent session of the MSC tool, with the contents of the scenario fragment displayed. A more powerful option is the capability of catenating a series of scenario fragments into a composite scenario. When the user selects a series of nodes along a path through the dependency graph, not necessarily adjacent, POGA will construct the shortest path through those nodes (as illustrated in Figure 5), an catenates the scenario fragments along this path. The user can now start an MSC session with the thus created composite path, for a more detailed analysis.

POGA graphs can also be hierarchical, by allowing each node to define, apart from a scenario fragment also a subgraph, specified again in POGA format. The subgraph can be created, inspected, or edited, by invoking the POGA tool recursively on the subgraph that is referenced in the node. The sessions are independent. POGA graphs can thus be nested arbitrarily deep. They may even be recursive (e.g., the subgraph may be equal to the graph in which it is referenced). Both the MSC scenario references and the subgraph references are interpreted as the names of files in a standard hierarchical file system.

HMSC Specifications. A recent MSC draft for the Z.120 standard [5] contains a definition of *High-level MSC*s that is comparable to MSC dependency graphs as described above. High-level MSCs are defined with new graphical symbols,

statistical techniques, simulations, empirical tests).

The timing verification option in MSC is also sensitive to this phenomenon. Its saving grace can be that it deals only with designs that are still abstract and tentative, for which even approximate answers can be of value to a designer.

3.3 The POGA Tool.

POGA, Pictures Of Graph Algorithms, is a graphical tool for constructing directed labeled graphs. The background processor of the tool provides implementations of a small set of standard graph algorithms, e.g. to find shortest paths, strongly connected components, roots and leaves, etc. POGA relies also on the background processor **dot**, an existing graph layout tool, for computing the size of nodes and the routing of edges [8]. An example of a POGA display of a labeled graph is given in Figure 4.

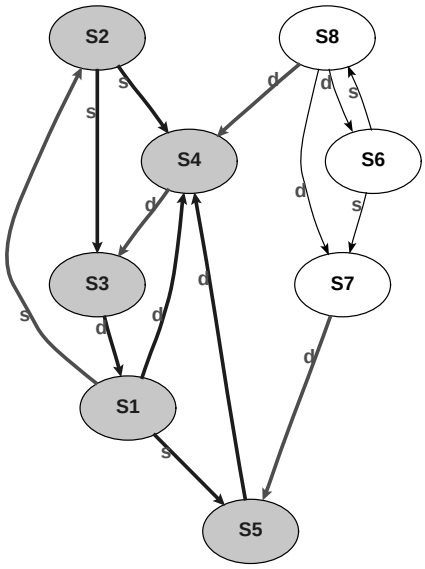


Fig. 4. POGA View Of A Labeled Graph. One of Three Strongly Connected Components is Shaded.

Although not originally designed for this purpose, POGA has proven to be an attractive tool for capturing the dependency relations in large sets of message sequence charts. A recurring complaint from users of ITU standard basic message sequence charts is that they do not allow for the specification of

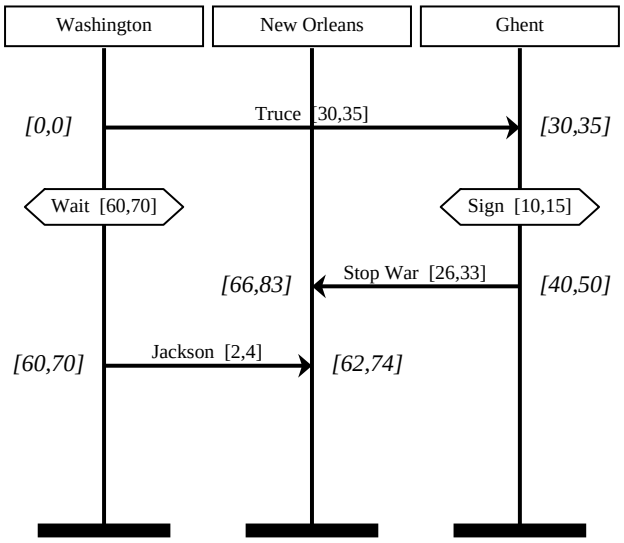


Fig. 3. Inconsistency Between Stated and Implied (*italic*) Time Bounds.

For the ITU standard, these annotations have no semantics. The MSC tool can interpret them, however, to analyze a scenario for possible inconsistencies in the assumptions about time bounds in the various processes. In the example of Figure 3, the MSC tool indicates an inconsistency in the calculated relative time of occurrence of the arrival at "New Orleans" of the "Stop War" message and of (General Andrew) "Jackson." In the scenario these events appear in an order that is inconsistent with the combined time bounds on the three messages and the two tasks. (The stated time bounds imply that the event labeled *[66,83]* must always occur *after* the one labeled *[62,74]*, but it appears *before* it in the chart.)

How Real Is "Real-Time" Really? There is a danger in the use of timing verification, though, that should be mentioned here. It can be very hard to provide meaningful upper- and lower-bound estimates for the duration of tasks (such as message transfer, or local computation) in a distributed system. Moreover, any bounds that exist at the time of the design, are almost guaranteed to change during the lifetime of the system. Unless at least the *relative* values of the bounds remain unchanged, the validity of the timing verification is jeopardized by such changes. It is therefore usually better to prove the soundness of a design *independently* of any assumptions about the passage of time or the duration of tasks, and to settle a performance evaluation of a system in other ways (with

Semantics. It should be noted here that the semantics of basic message sequence charts do not in general allow one to produce information about the existence of even race conditions. The feedback from the MSC tool is based on default choices for the missing information. The designer can override these defaults, or can make them more specific, once sufficient additional information about a system has become available.

For instance, to determine which event orders are enforceable (e.g., locally by each participating process) one needs information about the underlying architecture of the system in which the scenario is to be executed. The number and type of message buffers determines how messages from different sources may be interleaved at a common destination.

Weakness and Strength. The omission of this information from the scenario is not accidental. Rather than a flaw, this is one of the strengths of the scenario technique. By not requiring that *all* the information about a new design be available and agreed upon before a message sequence chart is composed and analyzed, some hurdles in the design process are removed. To still be able to provide feedback about the quality of an early design, the MSC tool provides a small number of predefined choices for reasonable underlying system architectures. For each such choice, the tool can perform the analysis of enforceable and non-enforceable orders, race conditions, and timing conflicts. The rational for this choice is that, rather than placing an extra burden on the designer, by asking for extra choices that may be irrelevant or distractive at an early point in the design, the tool can offer feedback on the consequences of *possible* choices that a designer may choose to make later.

3.2 Timing Verification.

In offering the user a choice of possible underlying system architectures, the MSC tool goes slightly beyond the ITU standard. Similarly beyond the standard is a facility for annotating scenarios with information about the minimum and maximum duration of time between events. The user can annotate a message, for instance, with an upper- and lower-bound estimate for message transmission delay, and the user can annotate local tasks (or *conditions* in ITU terminology) with an upper and lower-bound estimate for computational delays. Both cases are illustrated in Figure 3, where the time bounds are enclosed in square brackets of the form: [lower,upper].

Pointed boxes in Figure 3 represent the tasks.[2] The time bounds attached to tasks and messages are user-provided, the remaining bounds are calculated by the tool.

[2] The scenario depicted refers to the Battle of New Orleans, a famous event in American history. The battle was fought on 8 January 1815, two weeks after the signing of a peace treaty in Ghent between the British and the Americans. Trans-Atlantic news traveled slowly in those days.

tion of message sequence charts. From the designer's point of view, editing and documenting these charts is the primary attraction. An added benefit is that these tools make it possible to warn the designer about logical inconsistencies in the growing design as well. By formalizing message sequence charts, we further make it possible to link the early design techniques seamlessly with more powerful mechanical verification techniques later in the design cycle.

The ITU is in the process of standardizing the visual and textual elements of basic message sequence charts (and a number of extensions to these) [5]. By conforming to this standard, we can help the designers produce unambiguous documents that can safely be stored, processed, and exchanged with others. At the same time, by referring to the ITU Z.120 documents, any disagreements between tool users (about which visual elements or concepts should be supported) can be settled objectively.

3 Two Light-weight Design Tools: MSC and POGA

MSC is a simple graphical tool to support online construction and editing of ITU standard basic message sequence charts, described in more detail in [1, 4]. The graphical interface is written in about 2,000 lines of Tcl/Tk [10]. Additional information about a scenario can be provided by a background process, written in about 900 lines of ANSI standard C [6]. POGA, similarly, has a graphical interface written in Tcl/Tk and a background process written in ANSI C, each of roughly the same size as the corresponding part in MSC. POGA is used to define and formalize the dependencies between message sequence charts.

3.1 The MSC Tool.

MSC's background process provides information about the enforceable and non-enforceable parts of the visual order that is shown in a message sequence chart, and it can warn about possible race conditions, and timing conflicts. MSC will flag two race conditions for the chart shown in Figure 2. The first one is the race between the first two messages to the "Patent Office." Because the messages are sent from two independent sources, there can be no guarantee about their relative order of arrival at the destination. The second race is between the next two messages sent by the inventors to the "Patent Office." The example illustrates that the existence of some races can be critical, while others are quite harmless. Only the designer of the distributed system can tell in which category each specific case belongs. The MSC tool will provide the designer with a menu of all race conditions and other logical inconsistencies it detects (e.g., see Timing Verification below). In considering the feedback from the tool, the designer has the option to define so-called *co-regions* [5] to indicate event sequences that require not particular ordering. The remaining inconsistencies must be addressed by changes in the design itself. Usually, roughly half the number of warnings produced by the MSC tool in this way are found to reveal true errors in a specification, that require corrections to the design itself.

The Sketches Designers Make. Time sequence diagrams have been invented and reinvented by countless people. For better or for worse, they provide an irresistible informal technique for sketching sample behaviors of distributed systems. Specialized versions of basic time sequence diagrams have been proposed under many synonyms, e.g. object interaction diagrams, flow diagrams, execution scenarios, use cases, and more recently *message sequence charts*. The visual elements of a message sequence chart are simple and intuitively clear to also first-time users. Few people, for instance, will need an elaborate explanation to understand what is shown[1] in the message sequence chart of Figure 2.

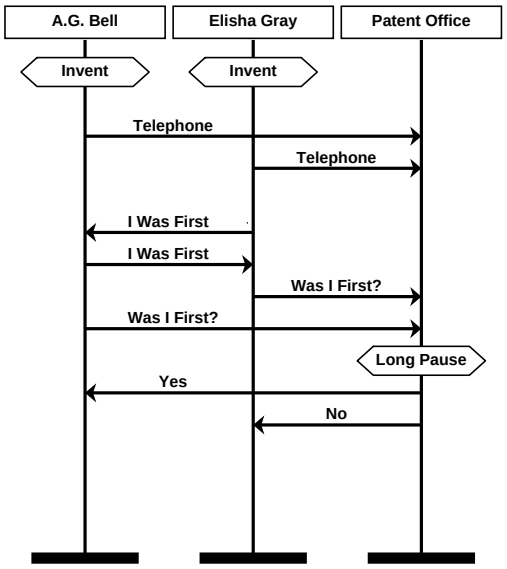


Fig. 2. Message Sequence Chart With At Least One Race Condition.

In the first phases of a design, different visual representations of the structure and behavior of a new system are explored with techniques that are perceived to be informal. It is important to recognize, though, that the sketches that are produced in this way are in reality nothing other than small abstract models of an evolving systems design.

To facilitate the designer's task in the early phases of a design, therefore, we have begun by developing tools that simplify the construction and manipula-

[1] Two people filed very similar patent claims for the telephone on 14 February 1876. One was Elisha Gray, the other Alexander Graham Bell. Since Bell's patent claims turned out to have been filed shortly before Gray's, he won the patent rights.

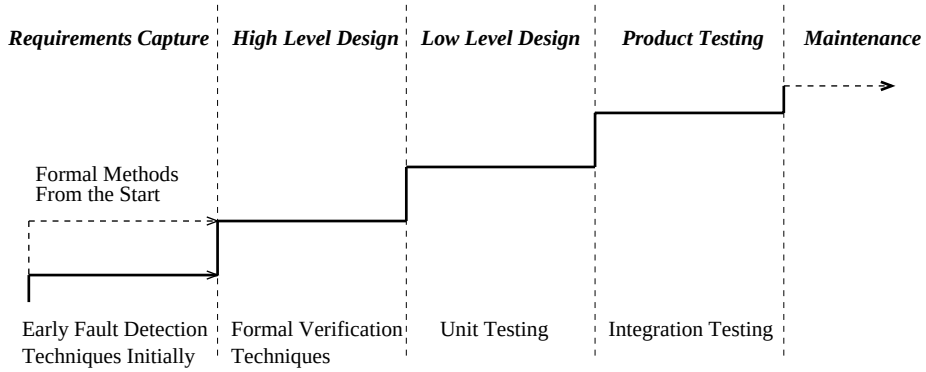


Fig. 1. Design Hurdles. Lowering Barriers To Formal Methods.

designers, because after the experiment is over, they are free to return to their regular mode of working. There is something not quite right about this, but what?

Too Much, Too Soon. The design of a new system usually starts in a rather tentative, exploratory, and iterative way with a *Requirements Capture* phase, as illustrated in Figure 1. The problem domain is surveyed, and fragments of a trial solution are sketched. Most of these sketches lead a short life, and are modified frequently. Some of them will survive and will become a permanent part of design documents, as soon as the understanding of the new system has settled sufficiently that such documents can indeed be written. In the initial phases of a design, all-out formal specification and verification techniques offer little help to the designer. They appear to require a level of formality and precision that is simply not available yet. The demands placed on the designer seem extreme: utter precision and detail is expected at a most inopportune time in the design cycle. In return, only fairly abstract properties may be established. The initial price to be paid is too high, the initial rewards are far too small.

The work on early fault detection methods attempts to bridge the gap between formal methods and design practice, by providing a more targeted set of tools for dealing with the early phases of a design. The full benefit of a formal method will first become evident during the high-level design phase, so all we need is a method that eases the transition to the more formal method, not one that makes absolute demands right from the start.

Instead of inventing a completely new design technique for the initial design phases, and then trying to persuade designers to adopt it, we have decided to study the informal techniques that designers naturally use, and consider if they could be sufficiently formalized to bridge the gap towards more formal methods in a natural way.

Formal Methods for Early Fault Detection

Gerard J. Holzmann
gerard@research.bell-labs.com

Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Abstract. A traditional formal verification method becomes an effective weapon in the arsenal of a designer only *after* sufficient insight into a design problem has been developed for a draft solution to be formalized. In the initial phases of a design the designers can therefore perceive formal methods to be more of a hindrance than an assistance.
Since formal methods are meant to be problem solving tools, we would like to find ways to make them both effective and attractive from the moment that a design process begins.

Invited paper: 4th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems. Sept 1996, Uppsala, Sweden

1 Introduction

Our aim is to develop a suite of tools that can support a design method based on the paradigm of *early fault detection* [4]. Tools of this type should be usable literally from the first day that a new design process starts, being applied to the early sketches of a new system. This means that these tools should be able to tolerate incomplete and sometimes inconsistent designs, and still be able to provide useful feedback to a designer.

Section 2 summarizes the design method based on early fault detection. Section 3 briefly discusses two light-weight design tools that we have built to support this paradigm, and comments on the first experiences with the application of these tools in an industrial design project. Section 4 discusses how the design decisions that are documented and analyzed with early fault detection tools can be used throughout a systems design cycle.

2 Scenario Based Design

The Devil's Advocate. Formal methods have a number of recognizable characteristics. First, each method in this class is reported to be highly successful when it is applied by the developer of the method. Second, it is often used only reluctantly by traditional designers, and often on an experimental basis only. Third, each such experiment is considered to be a success by the initiators, but usually only a mixed blessing by the designers. Admittedly, all those participating in a formal methods experiment have reason to be content: the advocates, because we have one more documented case of a successful application, and the