♦ Design Tools for Requirements Engineering

Gerard J. Holzmann, Doron A. Peled, and Margaret H. Redberg

Industrial software design projects often begin with a requirements capture and analysis phase. During this phase, the main architectural and behavioral requirements for a new system are collected, documented, and validated. To date, however, requirements engineers have had few reliable tools to guide and support this work. We show that a significant portion of the design requirements can be captured in formalized message sequence charts (MSCs) using a set of tools that we built to reliably create, organize, and analyze such charts.

Introduction

In large software projects, the design task is often divided between systems engineers and developers. The first part of the design is in the hands of the systems engineers. They perform feasibility studies, consider how the new design interacts with existing systems, or system components, and decide on the main architectural and structural issues. The required or anticipated behavior of the new software is typically described in a requirements document, in English prose.

The architectural issues that must be decided in this phase include how to apportion new functionality across existing and new components. Design requirements are captured in detailed specifications for each system component. Often, the systems engineers are also asked to create validation tests for new system functionality, and they guide the application of these tests after the new functionality has been implemented.

Traditionally, the medium of the systems engineer has been a written requirements document, illustrated with pictures of sample behaviors. Many of the documented requirements describe the expected behavior of a component in response to a particular sequence of external stimuli. These descriptions are restricted to the externally visible behavior of components, called *gray box* descriptions. The internal realization of each behavior is deliberately left unspecified.

Because of the limits of informal text, these high-level descriptions can be both ambiguous and incomplete. Capturing complicated branching scenarios accurately in a written document is a difficult, detailed task that systems engineers often avoid to keep the requirements documents manageable. The descriptions that are included are often limited to the so-called *sunny day* scenarios, that is, scenarios that exclude all possibilities for failure or error. The exceptions are left as an exercise for the developers who will have to implement the system in conformance with the requirements. These exceptions, however, typically usurp the larger part of the development effort and ultimately determine the quality and reliability of the final design.

Although the exceptions could affect both the overall architecture and the design of individual components, they are not dealt with until development is well under way. None of the requirements documents normally contain this information. Aspects of the final implementation that address the exception cases are often documented only as folklore, or not at all. When the need arises—for instance, when the design needs to be enhanced or modified—this knowledge can only be rediscovered by reading code.

The scant coverage of exceptions in requirements documents returns to haunt the systems engineers at the end of the implementation cycle, when acceptance testing is conducted. Test cases are created by manually translating textual requirements into test scripts. The resulting set of test cases

Panel 1. Abbreviations, Acronyms, and Terms

FIFO—first-in first-out
ITU—International Telecommunications Union
MSC— message sequence chart
POGA—pictures of graph algorithms
Tcl/Tk—Tool Control Language/Toolkit
TEMPLE—a template matcher that allows the user to search a POGA database of hierarchical message sequence charts for matches of a negative test case

is no more complete than the original set of requirements. Tests that deal with critical exception-handling capabilities are absent.

The remainder of this paper describes three tools that address the need for tool support by systems engineers. These tools—MSC (message sequence charts), POGA (pictures of graph algorithms), and TEMPLE (template matcher)—encourage a more nearly complete capture of behavioral requirements, including exception cases, and provide organization and search capabilities for a large, complex body of scenario-based requirements. The tools support design validation techniques through automated consistency checking and allow for a straightforward reuse of design requirements for test case generation.

Designed to operate in a simple, intuitive manner that requires no extensive training, the tools make design requirements accessible to all involved in a design effort, from systems engineers to developers and testers. The design requirements are stored in machine- and human-readable, plain-text format that is amenable to automated design verification techniques and automated indexing and cataloging tools.

A Set of Design Tools

We begin by describing how our tools fit into a typical requirements specification process.

The purpose of the requirements specification phase is to determine the intended behavior of a new system or system component. This work begins with discussions between systems engineers and key developers, who together explore the anticipated message exchanges between components. The goal of these discussions is to ascertain whether the capabilities and the behaviors required of each component are consistent and achievable. During these discussions, the engineers often visualize their work with informal graphical message flow diagrams and informal architectural views drawn with boxes and arrows.

MSC,[1,2] the first requirements engineering tool we built, supports the capture of message exchanges as machine-readable, standardized[3] MSCs and conducts simple, automated consistency checks. The sequence charts can be annotated with textual requirements; they can also be symbolically linked to a document to create a precise association. The link can associate a message exchanged between system components, the states of the sending and receiving components, and the gray box requirement that applies to this exchange. The MSC tool is described in more detail in the next section.

MSCs, as standardized by the International Telecommunications Union (ITU), specify only nonbranching message exchanges. MSC fragments can be combined in various ways to obtain hierarchical MSCs that can contain conditional branching and iteration. A single path through such a hierarchical chart defines what is sometimes called a *use case*, or a *test case*. In a use case, each path corresponds to a simple concatenation of MSC fragments. Hierarchical MSCs can be decomposed naturally into finer levels of detail recursively. Our second tool, POGA,[2,4] captures and analyzes hierarchical MSCs. POGA is described in more detail later in this paper.

A better validation of a library of requirements, formalized in MSCs, becomes possible if we allow for the specification of not just positive, but also negative, test cases. These negative test cases are message exchanges that are undesirable and that represent a suspected type of design flaw or system failure. Experienced designers on a project are a good source for such negative test cases, and, over time, a library of negative test cases can be constructed for testing in future design cycles. The negative test case feature can also be used to reliably determine if specific functionality is present in or absent from the documented requirements. Our third tool, called TEMPLE,[5] allows the user to search a POGA database of hierarchical MSCs for matches of a negative test case, called a *template*.

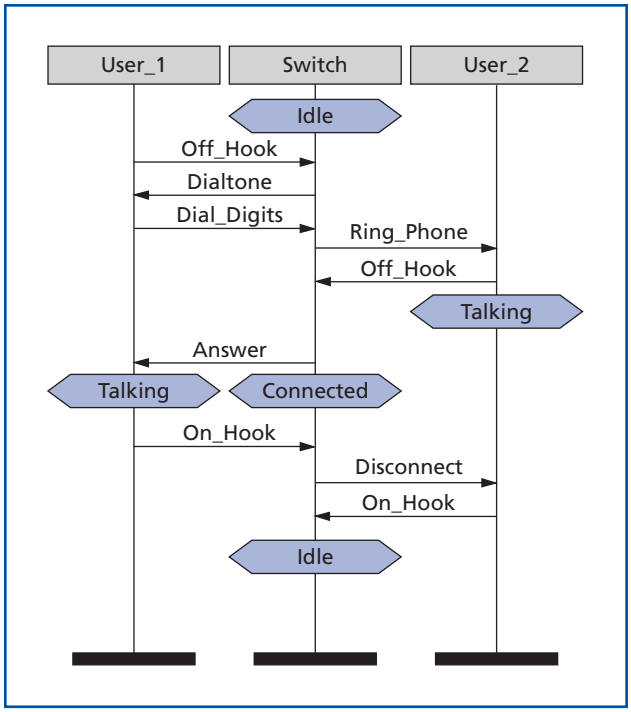

Figure 1.
A message sequence chart.

The tool is described in more detail in the section titled "TEMPLE."

MSC and POGA are relatively lightweight tools that were implemented in about 4,500 lines of Tcl/Tk (Tool Control Language/Toolkit) each,[6] supported by small background programs written in about 600 lines of ANSI standard C. A prototype of the tool TEMPLE was implemented as a translator that converts templates and scenarios into input for an existing state machine-based verifier.[7] The translator comprises about 1,700 lines of C language code.

MSC

Time sequence diagrams appear in almost every textbook on network or protocol design. They are popular as visual records of design decisions, and as such are also frequently included in requirements documents. Often, however, the visual formalism adopted varies from one application to the next. If a commercial word processor is used to construct time sequence diagrams, their semantics are lost altogether, along with their ability to perform automated consistency checks. The ITU has proposed a standardized representation for time sequence diagrams, called MSCs, in its recommendation Z.120.[3] The standard representation applies to both the graphical elements of an MSC and its machine-readable form.

Figure 1 shows an example of an MSC chart, recording the normal flow of events in the setup of a telephone call.

Vertical lines in the chart correspond to asynchronously executing processes in a logically or physically distributed system. The arrows represent messages exchanged between these processes. The tail of each arrow corresponds to the event of sending a message; the head corresponds to its receipt. Time advances from top to bottom; arrows can be drawn either horizontally or sloping downward, but not upward.

Our MSC tool allows the user to construct and edit ITU-compliant MSCs interactively, in graphical form, and to store these charts in Z.120 format in the file system. Requirements text can be inserted directly in comment or text boxes that become an integral part of the charts. Alternatively, a text box can contain symbolic references to the requirements document, for manual lookups, or they may contain hypertext links that connect directly to requirements text in commercial word processors.

The MSC tool also integrates a modest amount of formal verification into the design process in a way that is almost invisible to the users. The tool contains an analyzer that can detect logical inconsistencies in the charts, such as potential race conditions between message arrivals.[1] To enable the user to conduct this type of verification, the tool contains a choice of possible semantic interpretations for each chart, as we will illustrate below. These semantic choices are formally outside the scope of the ITU definitions, but they can directly influence the implementation of the requirements that are expressed.

Analysis of MSCs

An MSC can be formalized as a fivetuple $\langle E, <, \mathcal{P}, L, T \rangle$, where

- E is a set of *events* (sends and receives),
- $< \subseteq E \times E$ is an event ordering (an acyclic relation on events),
- $\mathcal{P}$ is a set of asynchronous processes,
- $L : E \mapsto \mathcal{P}$ maps each event to the process in which it appears, and

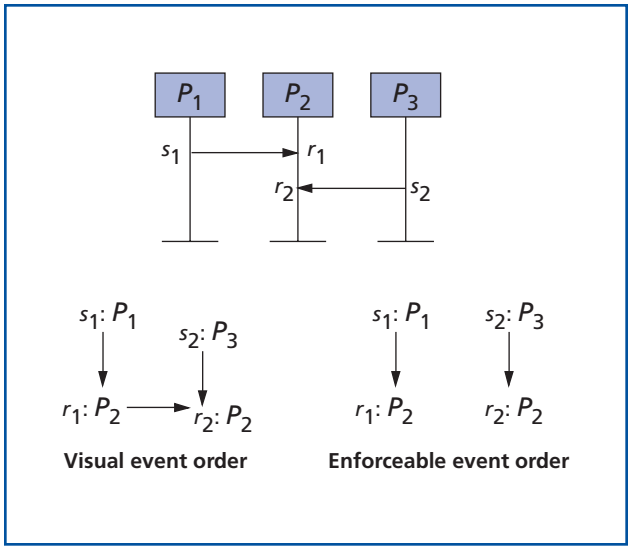

Figure 2.
A message sequence chart and two associated event orders.

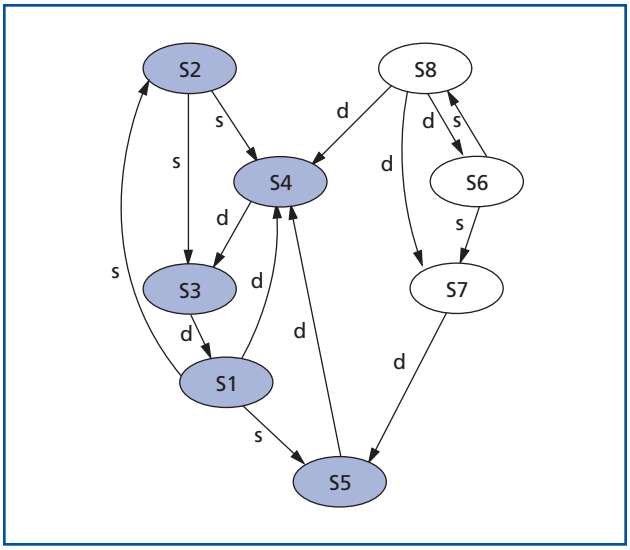

Figure 3.
POGA view of a labeled graph, with one of three strongly connected components shown in blue.

- $T : E \mapsto \{\mathsf{s}, \mathsf{r}\}$ maps each event to its type, that is, send or receive.

The relation < formalizes the visual ordering of events, as it is displayed in an MSC. Thus, we have $e < f$ if e is a send event and f is the corresponding receive, or when e and f appear within the same process, and e appears above f in the process line.

We can distinguish between the two types of ordering that are captured in the relation < as follows. We can define the relation $<_c = \{(e, e') | e < e' \wedge T(e) = \mathsf{s} \wedge T(e') = \mathsf{r} \wedge L(e) \neq L(e')\}$ to record the ordering relations between sends and receives alone. Thus, a message in the chart is formalized as a pair of events, $(e, e') \in <_c$.

Let $E_{P_i} = \{e | e \in E \wedge L(e) = P_i\}$, that is, E_{P_i} is the set of all events that belong to process P_i alone. We define the relation $<_{P_i} \subseteq (E_{P_i} \times E_{P_i})$ to record the ordering relations between events within the process P_i alone.

The relation < is simply the union of all these orders, that is, $<_c \cup \left(\bigcup_{P_i \in \mathcal{P}} <_{P_i}\right)$.

For example, for the MSC in **Figure 2**, we have $E = \{s_1, r_1, s_2, r_2\}$, $\mathcal{P} = \{P_1, P_2, P_3\}$, $<_c = \{(s_1, r_1), (s_2, r_2)\}$, $<_{P_1} = <_{P_3} = \phi$, and $<_{P_2} = \{(r_1, r_2)\}$.

The event order < is depicted on the lower left side of Figure 2. This order is termed "visual," because it reflects the way in which the chart is drawn. It may not, however, properly reflect the way in which the scenario can be executed, as explained below.

Causal Structures

To facilitate the analysis of an MSC, we can define a *causal structure* for any given set of semantic choices for the underlying system architecture. The semantic choices are defined in rules that will be discussed shortly. The causal structure is defined by a fivetuple $\langle E, \prec, \mathcal{P}, L, T \rangle$, where the only component that differs here from the definition of an MSC is the relation $\prec$. This relation is the enforceable event order of the MSC, or enforceable order for short.

If $e_1 \prec e_2$, we have $e_1 < e_2$, and we know that the occurrence of event e_2 can always be delayed until after event e_1 has terminated. The transitive closure $\prec^*$ of $\prec$ is a partial order on events, which may be called the implied causal order of the chart. Two events that are unordered by $\prec^*$ cannot be prevented from occurring independently or concurrently by any implementation.

In the MSC of Figure 2, the enforceable order appears at the lower right of the figure. The distinction between the visual order and the enforceable order can reveal logical inconsistencies in an MSC. In Figure 2, for example, the receive event r_1 may not always precede r_2, because the sending processes of

the two corresponding messages execute asynchronously. The Z.120 standard has a mechanism that uses co-region definitions to identify cases in which event orders are undefined. Such definitions, however, appear to be rarely used in practice and are often a cause of confusion to requirements engineers.

An enforceable order is derived from a visual order using semantic rules.[1] For example, one such semantic rule asserts that always $<_c \subseteq \prec$, because a send event always precedes the corresponding receive event in any implementation. The arbitrary nature of the order between r_1 and r_2 discussed above is reflected by the absence of a rule that would allow one to derive $e_1 \prec e_2$ from the given conditions

$L(e_1) = L(e_2)$, $T(e_1) = T(e_2) = \mathsf{r}$, and $e_1 < e_2$.

The specific semantic rules, which are beyond the scope of the Z.120 standard for MSCs, depend on the system's architecture. In a system where each process has multiple asynchronous communication queues, one can impose any arbitrary order on independently received messages, reflecting the order in which the messages are processed, rather than the order in which they arrive. In such a system, letting $r_1 \prec r_2$ under the above given conditions can be meaningful.

Semantic Rules for FIFO Queues

A set of semantic rules for an architecture in which each process has access to precisely one first-in first-out (FIFO) message queue sets $e_1 \prec e_2$ in exactly the following four cases:

1. A matching pair of send and receive events is always ordered.
   $T(e_1) = \mathsf{s} \wedge T(e_2) = \mathsf{r} \wedge L(e_1) \neq L(e_2) \wedge e_1 < e_2$
   This refers to all pairs of events ordered by the event relation $<_c$ defined earlier.
2. Messages are ordered by the FIFO queuing discipline.
   $T(e_1) = \mathsf{r} \wedge T(e_2) = \mathsf{r} \wedge e_1 < e_2 \wedge L(e_1) = L(e_2) \wedge$
   Exists $f_1, f_2 : (f_1 <_c e_1 \wedge f_2 <_c e_2 \wedge L(f_1) = L(f_2) \wedge f_1 < f_2)$
   (For a non-FIFO ordered message queue, this rule would be deleted.)
3. Two sends within the same process can always be ordered.
   $T(e_1) = \mathsf{s} \wedge T(e_2) = \mathsf{s} \wedge L(e_1) = L(e_2) \wedge e_1 < e_2$
4. A receive and a later send within the same process can always be ordered.
   $T(e_1) = \mathsf{r} \wedge T(e_2) = \mathsf{s} \wedge L(e_1) = L(e_2) \wedge e_1 < e_2$

Detecting Race Conditions

Consider an MSC with visual order $<$ and enforceable event order $\prec$. A pair of events may sometimes appear in the visual order, but not correspondingly in the enforceable order. Therefore, the chosen semantics cannot guarantee that the events will always occur in the order in which the MSC displays them.

Race Condition: Events e and f from the same process p are said to be in a race if $e < f$ but not $e \prec^* f$.

Race conditions do not apply to events that are enclosed in explicit co-regions, as defined in the Z.120 standard for MSCs.[3] Outside co-regions, however, an event race identifies parts of a system requirement that cannot be implemented.

The transitive closure $\prec^*$ of the enforceable order $\prec$ can be computed with the well-known Floyd-Warshall algorithm.[8] The standard Floyd-Warshall algorithm has computational complexity N^3, where N is the number of events. For the special case considered here, this complexity can be reduced to N^2, using the fact that an initial ordering of all events is known.[1] Because the number of events in even large MSCs is rarely more than one thousand, the time or space requirements for this type of analysis are of no practical significance. The MSC tool can perform the required checks interactively on even small laptop PCs.

POGA

POGA is a generic graphical tool for constructing and analyzing directed labeled graphs.[4] The tool includes known algorithms for finding such things as the shortest paths, strongly connected components, and roots and leaves. In its prototype version, POGA relies on the program dot, a standard graph layout tool, for handling the visual display of graphs.[9] **Figure 3** shows a small example of a POGA display.

POGA Dependency Graphs

POGA allows for a convenient graphical display of the dependencies between MSC fragments as they are formalized and captured with the MSC tool. **Figure 4**

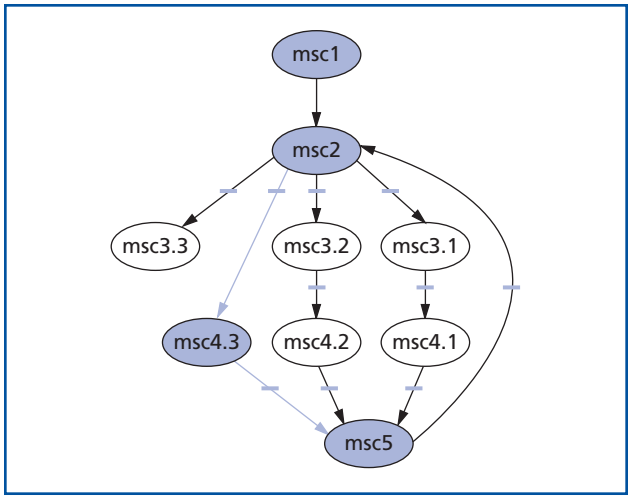

Figure 4.
Interdependencies of scenario fragments.

shows such a dependency graph, derived from an existing graph that was produced in an industrial trial of MSC and POGA.

The nodes in Figure 4 can represent complicated scenarios from the distributed system that is being designed—interaction sequences in which all structural components of the system can in principle participate. The first node in Figure 4, labeled msc1, for instance, represents the first series of steps in a call setup procedure for a telephone call. A subsequent processing step, msc2, representing, for example, call routing, can have four different outcomes. The call might have to be rejected (msc3.3), or it might proceed in various ways, depending on the specific call features that have been invoked. By selecting a node in an MSC dependency graph, the POGA user can request the corresponding scenario fragment to be displayed with MSC, where it can be edited and analyzed.

The bold lines in Figure 4 indicate a possible traversal of the dependency graph, from the root to one of two possible termination points. The graph traversal identifies one possible composite MSC that can be constructed from the fragments msc1…msc5. When the POGA user selects a series of nodes along a path through the dependency graph, POGA will construct the shortest path through those nodes, as illustrated in **Figure 5**, and can concatenate the scenario fragments along this path and pass the concatenation to the MSC tool for a more detailed analysis.

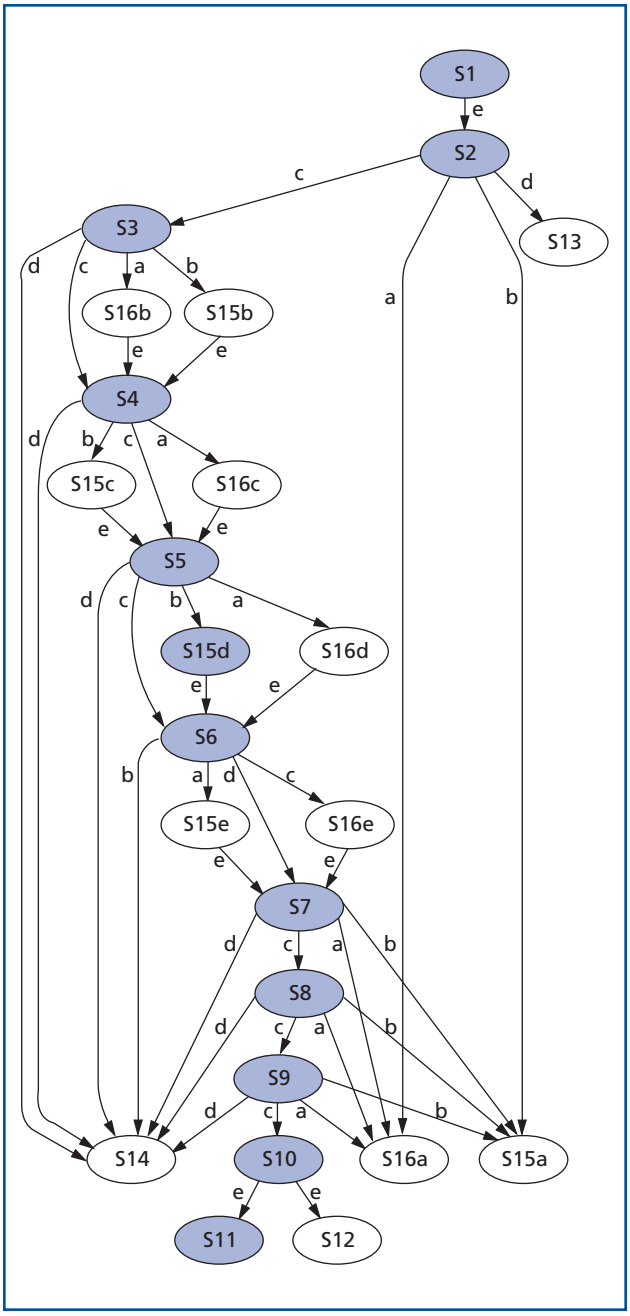

Figure 5.
A path through a POGA dependency graph.

The scenarios that are handled by MSC and POGA are deliberately defined at a high level of abstraction, that is, defining only gray box components. MSC and POGA describe the allowable, or required, sequence of events at the external interface to the system under design, but not the internal details of an implementation. Once more is known about the design, these

high-level scenarios can be reused and linked to the more detailed descriptions provided by developers, which show precisely how the elements of a design implement the required behaviors.

A set of longer MSCs constructed from the many possible paths can also serve as a test case suite or as an input specification to generate a test harness in commercial tools that import standardized MSCs.

TEMPLE

The tools MSC and POGA help create, debug, organize, and maintain MSCs and related text requirements. TEMPLE[5] adds to these tools the ability to search an MSC, or a library of MSCs grouped in POGA graphs, for fragments or paths that match a sample behavior, also specified as an MSC.

The specification MSC is called a *template* and denotes a set of events (sending and receiving of messages) and their relative order. A specification matches any MSC fragment or path that contains at least those events that appear in the template, while preserving the order between them. Thus, a specification can also be regarded as an incomplete or skeletal MSC, which allows additional events besides those specified. The template is constructed using the MSC tool, and the TEMPLE tool allows the user to specify appropriate queue semantics, and with an algorithm, to search a set of MSCs for a match with a template specification. When the search concludes, TEMPLE reports the results to the user.

Template matching can serve various purposes. Specifications based on MSCs may include thousands of scenario fragments organized in POGA graphs. Searching these scenario libraries for the occurrence of specific features can be difficult. Template matching mechanizes such searches using the following techniques:

- *Debugging*—Determining whether a bad MSC (sometimes referred to as a *negative test case*), with an unwanted sequence of message exchange, exists in the system. Libraries of negative test cases can be constructed and accumulated over time to preserve this knowledge, which is acquired through trial and error.
- *Determining use and context of use*—Finding charts that contain a specific exchange of messages to determine the frequency of and the context (that is, the scenarios) in which the exchange occurs.
- *Existence checking*—Determining whether a required feature is already included or remains to be added.

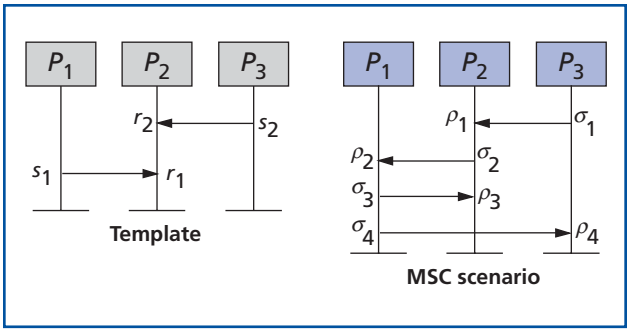

Figure 6.
A template and a matching scenario.

Figure 6 shows an example of a template and a matching MSC scenario. In both charts, there are three processes, P_1, P_2, and P_3 (although the template may also contain fewer processes). We will assume single FIFO-queue semantics. The events in a template and a scenario are said to match if they agree on both the type (send or receive) and process; moreover, the order among them must be preserved. In Figure 6, for example, event s_1 agrees with σ_3 and σ_4 on both the type and process, while r_1 agrees with ρ_1 and ρ_3. However, to agree on the enforceable order, the event that matches s_1 must immediately precede the event that matches r_1. This limits the match to pair s_1 with σ_3, and r_1 with ρ_3. Similarly, s_2 is paired with σ_1, and r_2 with ρ_1. One can also label messages and allow only messages with the same label to be matched.

The scenario may contain additional order, which is not reflected in the template. Selecting the single FIFO-queue semantics, the events r_1 and r_2 of the template are unordered (in fact, we could depict the template in such a way that r_1 appears above r_2 in the process line of P_2 and obtain the same matching results). In the MSC scenario of Figure 6, ρ_1 and ρ_3, which match the template events r_2 and r_1, respectively, are not directly ordered. However, they are causally ordered, as inferred from the chain

$\sigma_1 \prec_N \rho_1 \prec_N \sigma_2 \prec_N \rho_2 \prec_N \sigma_3 \prec_N \rho_3$, where $\prec_N$ is the enforceable order from the MSC. Thus, the template is interpreted as reflecting only a subset of the events, and a subset of the order among them; additional inferred causal order can be imposed by causal chains that include events that do not appear in the template.

We will formalize the template matching and briefly explain the matching algorithm. A more detailed description can be found in Levin and Peled.[5]

A template and an MSC have the same representation, and both can be translated into causal structures. Consider a causal structure $M = \langle E_M, \prec_M, \mathcal{P}_M, L_M, T_M \rangle$ for a template and a causal structure $N = \langle E_N, \prec_N, \mathcal{P}_N, L_N, T_N \rangle$ for an MSC, where the template processes $\mathcal{P}_M$ are included in the set of MSC processes $\mathcal{P}_N$. Then a match between M and N exists when there is a mapping $\mu : E_M \mapsto E_N$ of the template events to the MSC events, called the *matching function*, such that:

1. For each $e \in E_M$, $L_N(\mu(e)) = L_M(e)$, that is, matching events belong to the same process.
2. For each $e \in E_M$, $T_N(\mu(e)) = T_M(e)$, that is, matching events agree on the type of event.
3. If $e_1 \prec_M e_2$, then $\mu(e_1) \prec_N \mu(e_2)$, that is, matching pairs of events preserve the enforceable order.

The Matching Algorithm

To describe the matching algorithm, we must define some notions from partial-order theory. Given a partial order relation $\prec$ (that is, a transitive, reflexive, and asymmetric relation), with $\prec \subseteq E \times E$, a slice S is a subset of E such that if $e \prec f$ and $f \in S$, then $e \in S$.

For S to qualify as a proper slice of partial order $\prec$, any event ordered before some event f within slice S must also be part of S. Next, we define a *border* element e of slice S to be any event for which there exists an f in $E \setminus S$ (that is, in E and not in S) such that $e \prec f$, and for no $g \in E$, $e \prec g \prec f$. In other words, every border element of S must have an immediate successor in partial order $\prec$ that is outside S. Slices of E are ordered by inclusion, namely, S_2 is a successor of S_1, denoted $S_1 \rhd S_2$, exactly when S_2 contains one element more than S_1.

Our algorithm defines a slice S_M of a template M, with respect to a partial order that is the transitive closure of $\prec_M$. It also defines a slice S_N of an MSC N, with respect to a partial order that is the transitive closure of $\prec_M$ ($\prec_M$ and $\prec_N$ are not necessarily partial orders themselves). The algorithm matches each border element of S_M with a border element of S_N. Denote this by match (S_M, S_N). The matching algorithm checks when successors of matched slices also match; namely, when $match(S_M, S_N)$, $S_M \rhd S_M'$ (or $S_M' = S_M$, as the template may contain fewer events, hence there may be fewer template slices), and $S_N \rhd S_N'$, check whether $match(S_M', S_N')$.

Such a check is easily obtained from the description of the match and the notion of slices; we only need to guarantee that a newly matched pair of events from E_M and E_N agree on the order relations $\prec_M$ and $\prec_N$ with already matched pairs of border events.

The notion of slices and the matching between border events allows us to translate the problem to match two causal structures into a set of transition rules between global states. In this case, a "global state" corresponds to the border events of a pair of template and MSC slices and the matching between them. The main advantage of this translation is that efficient existing tools can be used for manipulating transition systems. The template matching algorithm is implemented by a translation of the template and MSC to be matched into COSPAN[7] finite state machines.

Conclusions

MSC, POGA, and TEMPLE are tools that can support and encourage a more rigorous capture, analysis, and documentation of the temporal requirements that define a new design. All these tools work with standardized MSCs and can be annotated with textual requirements or symbolically linked to requirements in a document. The MSC analysis also includes checks (not discussed here) on the consistency of timing assumptions that are made in MSCs.[1] Libraries of MSCs can be organized in complex scenarios with the POGA tool. An extension of POGA is planned, to support test sequence generation and to evaluate the test sequence coverage for given test suites.[10] This extension will enable experiments with test case derivation

to begin well before the final implementation of a design has been completed. The test cases can then serve not only to verify that a systems implementation conforms to the design requirements, as usual, but also to confirm with the target customers of a new system that the functionality they have requested has been understood by the systems engineers well before the system reaches the implementation phase.

The first trials of these tools, tentatively named *early fault detection* tools,[2,4] have been very positive. The close connection that MSC and POGA provide between formalized and written (textual) requirements adds considerable value, filling a gap not properly addressed by other tools. Our tools offer a simple, effective way to create and maintain living descriptions of generic system requirements. The descriptions provided by these tools supply another advantage; they comply with international standards that are also supported by complementary commercial development tools.

Although it is too early to measure the effect tools such as these can have on the design cycle, we expect they can improve product quality, simplify requirements maintenance, and, ultimately, reduce the time to market and development cost of new products.

Acknowledgments

We thank our colleagues at Bell Labs, including Rajeev Alur of the Computing Principles Research Department, Brian Kernighan of the Computing Structures Research Department, and Vladimir Levin of the Design Automation Center—CAD Products Department, all of whom helped develop the tools presented in this paper. Discussions with Bob Kurshan and Mihalis Yannakakis, members of the Computing Principles Research Department, also influenced the development of this toolset and the earlier papers on this subject. We are grateful to Margaret Eng for her support and valuable input in the first industrial trials of the toolset at AT&T.

References

1. R. Alur, G. J. Holzmann, and D. Peled, "An Analyzer for Message Sequence Charts," *Software Concepts and Tools*, Vol. 17, No. 2, Feb. 1996, pp. 70-77.
2. G. J. Holzmann, "Early Fault Detection Tools," *Software Concepts and Tools*, Vol. 17, No. 2, Feb. 1996, pp. 63-69.
3. ITU-T Recommendation Z.120, *Message Sequence Chart (MSC)*, March 1993. (This recommendation includes S. Mauw and M. A. Reniers, "An algebraic semantics of basic MSCs," Annex B, *The Computer Journal*, Vol. 37, No. 4, Apr. 1994, pp. 269-277.)
4. G. J. Holzmann, "Formal Methods for Early Fault Detection," *Proceedings of Formal Techniques for Real-Time and Fault Tolerant Systems,* Uppsala, Sweden, Sept. 1996, *Lecture Notes in Computer Science*, Vol. 1135, pp. 40-54.
5. V. Levin and D. Peled, "Verification of Message Sequence Charts via Template Matching," *Theory and Practice of Software Development, Lecture Notes in Computer Science, TAPSOFT (FASE) '97,* Springer-Verlag, Lille, France, 1997, to be published.
6. J. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, Reading, Massachusetts, 1994.
7. R. P. Kurshan, *Computer-Aided Verification*, Princeton Univ. Press, Princeton, New Jersey, 1994.
8. T. H. Cormen, C. E. Leierson, and R. L. Rivers, *Introduction to Algorithms,* MIT Press, Cambridge, Massachusetts, 1990.
9. E. R. Gansner, E. Koutsofios, S. C. North, and K-P. Vo, "A Technique for Drawing Directed Graphs," *IEEE Transactions on Software Engineering,*Vol. 19, No. 3, May 1993, pp. 214-230.
10. D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines—A Survey," *Proceedings of the IEEE*, Vol. 84, No. 8, Aug. 1996, pp. 1090-1123.

(Manuscript approved April 1997)

GERARD J. HOLZMANN is a distinguished member of technical staff in the Computing Principles Research Department at Bell Labs in Murray Hill, New Jersey, where he performs research on computer-aided verification tools. He received B.S. and M.S. degrees in electrical engineering and a Ph.D. in technical sciences, all from Delft University in The Netherlands.

DORON A. PELED received B.S., M.S., and D.S. degrees in computer science, all from the Technion, Israel Institute of Technology in Haifa. He performed postdoctural work at the University of Warwick in the U.K. Mr. Peled is a member of technical staff in the Computing Principles Research Department at Bell Labs

in Murray Hill, New Jersey, where he is conducting research in specification and verification of concurrent programs, formal semantics, automata theory, and mathematical logic. He is also a member of the editorial board of the journal Formal Methods in System Design.

MARGARET H. REDBERG is a member of technical staff in the ACTIVIEW®: ASOS & Assets Department at Network Systems, located in Holmdel, New Jersey. She is interested in requirements engineering and methods for large software projects. Ms. Redberg earned B.S. and M.S. degrees in industrial and operations engineering from the University of Michigan in Ann Arbor. ◆