[3] E.R. Gansner, E. Koutsofios, S.C. North, K-P. Vo. A Technique for Drawing Directed Graphs, *IEEE Trans. on Software Engineering*, Vol. 19, No. 3, May 1993, pp. 214–230.

[4] G.J. Holzmann, Early Fault Detection Tools, *Software Concepts and Tools*, Vol. 17, No. 2, pp. 63–69, 1996.

[5] G.J. Holzmann, Formal Methods for Early Fault Detection, *Proc. Formal Techniques for Real-Time and Fault Tolerant Systems*, Uppsala, Sweden, Sept. 1996. *Lecture Notes in Computer Science.* Vol. 1135, pp. 40–54.

[6] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), March 1993. (Includes [10] as Annex B.)

[7] R.P. Kurshan, *Computer-Aided Verification*, Princeton University Press, 1994.

[8] D. Lee, M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey, *Proc. of The IEEE*, Vol. 84, No. 8, August 1996, pp. 1090–1123.

[9] V. Levin, D. Peled. Verification of Message Sequence Charts via Template Matching, *TAPSOFT (FASE)'97, Theory and Practice of Software Development*, Lille, France. *Lecture Notes in Computer Science.* Springer, 1997.

[10] S. Mauw, M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 1994, Vol. 37, No. 4.

[11] J. Ousterhout. *Tcl and the Tk toolkit*, Addison-Wesley, 1994.

discussed here) on the consistency of timing assumptions that are made in message sequence charts [1]. Libraries of message sequence charts can be organized in complex scenarios with the POGA tool. An extension of POGA is planned, to support also test sequence generation, and the evaluation of test sequence coverage for given test suites, cf. [8]. With this extension, experiments with test-case derivation may 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 request has been understood appropriately by the systems engineers, well before the system reaches the implementation phase.

The first trials of these tools, tentatively named *Early Fault Detection* tools [4, 5], have been very positive. The close connection that MSC and POGA provide between formalized and written (textual) requirements are seen to add considerable value, filling a gap not properly addressed by other tools. The tools appear to offer a simple and effective way to create and maintain living descriptions of generic system requirements. It is also seen as an advantage that the descriptions provided by these tools comply with international standards that are supported also by complementary commercial development tools.

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

Acknowledgements

The authors would like to thank Rajeev Alur, Brian Kernighan and Vladimir Levin, who participated in the development of the tools presented in this paper. Discussions with Bob Kurshan and Mihalis Yannakakis have influenced the development of this toolset and the earlier papers on this subject. The authors are grateful to Margaret Eng for providing support and valuable input in the first industrial trials of the toolset at AT&T.

4 References

[1] R. Alur, G.J. Holzmann, D. Peled. An Analyzer for Message Sequence Charts. *Software Concepts and Tools*, Vol. 17, No. 2, pp. 70–77, 1996.

[2] T.H. Cormen, C.E. Leierson, R.L. Rivers. *Introduction to Algorithms*,

That is, any event that is ordered *before* some event f within slice S, must also be part of S, for S to qualify as a proper *slice* of partial order $\prec$. Next, we define a *border* element e of slice S to be any event for which:

> there exists an f in $E \setminus S$ (i.e., 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 which is the transitive closure of $\prec_M$. It also defines a slice S_N of an MSC N, with respect to a partial order which is the transitive closure of $\prec_M$. (Note that $\prec_M$ 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 less 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 thus allows us to translate the problem to match two causal structures, into a set of transition rules between global states. A 'global state', corresponds here 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 for manipulating transition systems can be used. The template matching algorithm is implemented by a translation of the template and MSC to be matched into COSPAN [7] finite state machines.

3 Conclusions

MSC, POGA, and TEMPLE are tools that can are meant to support and encourage a more rigorous capture, analysis, and documentation of the temporal requirements that define a new design.

The tool all work with standardized message sequence charts, and can be annotated with textual requirements or symbolically linked to requirements in a document. The analysis part for MSC also includes checks (not

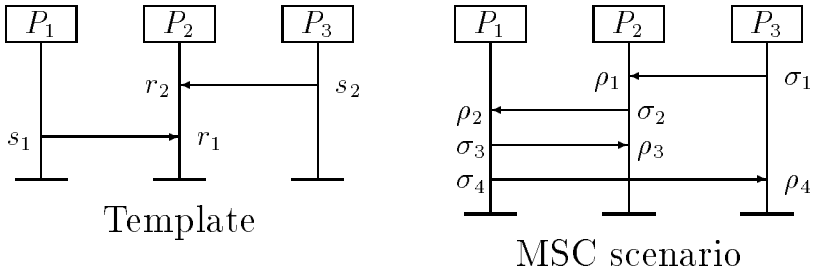


FIGURE 6. A template and a matching scenario

where $\prec_N$ is the enforceable order from the message sequence chart. Thus, the template is interpreted as reflecting only a subset of the events, and a subset of the order between them; additional inferred causal order can be imposed by causal chains that include additional events not appearing in the template.

We will formalize the template matching and briefly explain the matching algorithm. For a more detailed description, the reader is referred to [9].

A template and an MSC have the same representation and can both 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 $e \in E_M$, $L_N(\mu(e)) = L_M(e)$, i.e., matching events belong to the same process.

2. For each $e \in E_M$, $T_N(\mu(e)) = T_M(e)$, i.e., matching events agree on the type of event.

3. If $e_1 \prec_M e_2$, then $\mu(e_1) \prec_N \mu(e_2)$, i.e., 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 $\prec$ (i.e., 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 $e \in S$.

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 via an algorithm, search a set of MSCs for a match with a template specification. At the conclusion of the search, TEMPLE reports the results to the user.

Template matching can be used for various purposes. Specifications based on MSCs may include thousands of scenario fragments organized in POGA graphs. It can be difficult to search these scenario libraries for the occurrence of specific features. The use of template matching allows one to mechanize such searches. The match can be used for:

- 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 a trial and error process.

- Determining Use and Context of Use: finding charts that contain a specific exchange of messages in order to determine the frequency of and the context (i.e. in which scenarios) in which the exchange occurs.

- Existence Checking: determining whether a required feature is already included or remains to be added.

An example of a template and a matching MSC scenario appears in Figure 6. 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 type (*send* or *receive*) and process; moreover, the order between them must be preserved. In Figure 6, for example, event s_1 agrees with σ_3 and σ_4 on both 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.

It is important to notice that 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 $\sigma_1 \prec_N \rho_1 \prec_N \sigma_2 \prec_N \rho_2 \prec_N \sigma_3 \prec_N \rho_3$,

serve as a test-case suite or as an input specification for generation of a test harness in commercial tools that import standardized MSCs.

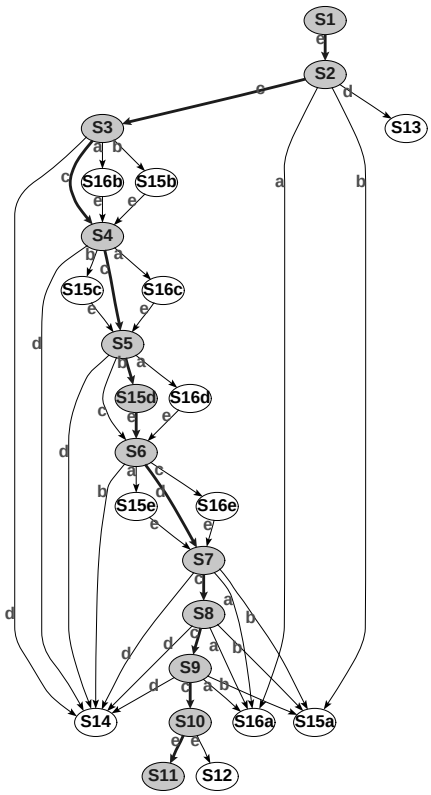


FIGURE 5. Path through a POGA dependency graph.

2.3 TEMPLE

The tools MSC and POGA allow for the creation, debugging, organization, and maintenance of message sequence charts and related text requirements. The tool TEMPLE [9] 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 skeleton MSC, which

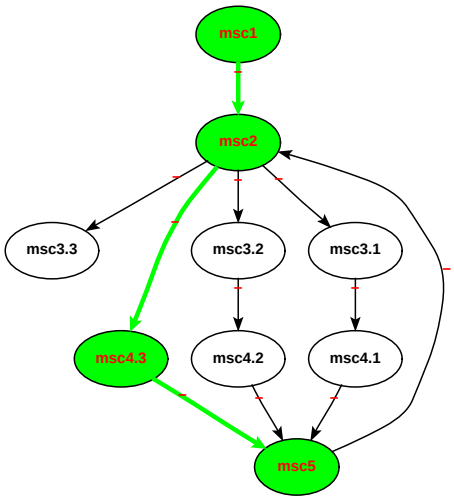


FIGURE 4. Interdependencies of Scenario Fragments

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. By selecting a node in an MSC dependency graph, the user of the POGA tool can request the corresponding scenario fragment to be displayed with MSC, where it can be edited and analyzed.

Also indicated in Figure 4, with bold lines, is a possible *traversal* of the dependency graph, from the root to one of the two possible termination points. The graph traversal identifies one possible composite message sequence chart that can be constructed from the fragments $msc1 \cdots msc5$. 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 instruct POGA to concatenate the scenario fragments along this path and pass it to the MSC tool, for a more detailed analysis.

The scenarios that are handled by MSC and POGA are deliberately defined at a high-level of abstraction, i.e., defining only *gray box* type components. The allowable, or required, sequences of events at the external interface to the system under design is described, 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, that 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

2.2 POGA

POGA (Pictures Of Graph Algorithms) is a generic graphical tool for constructing and analyzing directed labeled graphs [5]. The tool includes known algorithms for finding, for instance, shortest paths, strongly connected components, roots and leaves, etc. In its prototype version, POGA relies on the program **dot**, a standard graph layout tool, for handling the visual display of graphs [3]. A small example of a POGA display is given in Figure 3.

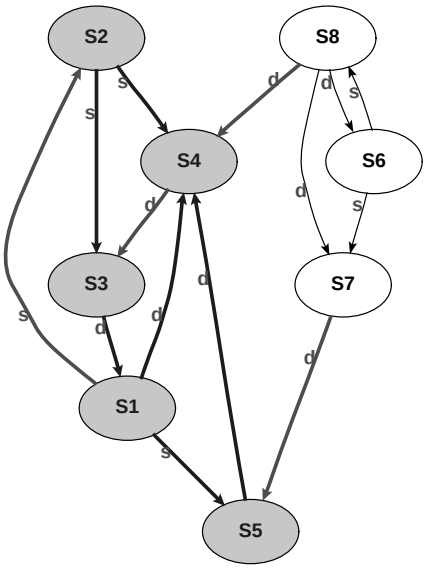


FIGURE 3. POGA View Of A Labeled Graph. One of Three Strongly Connected Components is Shaded.

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

The nodes in the graph from 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

four cases:

1. A matching pair of send and receive events are 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 that are ordered by the event relation $<_c$ defined before.

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 $\prec$. It can occur that a pair of events appears in the visual order, but not in the enforceable order. This means that 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 [6]. 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, e.g. [2]. 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 message sequence charts is rarely over 10^3, 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.

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 $\langle E, \prec, \mathcal{P}, \mathcal{L}, \mathcal{T} \rangle$, 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 event e_1 has terminated. The transitive closure $\prec^*$ of $\prec$ is a partial order on events, that may be called the implied *causal order* of the chart. Note that two events that are unordered by $\prec^*$ cannot be prevented from occurring independently or concurrently by any implementation.

In the message sequence chart of Figure 2 the *enforceable* order appears on the lower right of the figure. The distinction between the visual order and the enforceable order can reveal logical inconsistencies in an MSC. For example, in Figure 2 the receive event r_1 may not always be guaranteed to precede r_2, as the sending processes of the two corresponding messages execute asynchronously. The Z.120 standard has a mechanism for identifying cases where event orders are undefined, using so-called *co-region* definitions. Such definitions, however, appear to be rarely used in practice, and are often a cause of confusion to requirements engineers.

The derivation of an enforceable order from a visual order is done via *semantic rules* [1]. For example, one such semantic rule asserts that always $<_c \subseteq \prec$, since it is unavoidable that a send event precedes the corresponding receive event in any implementation. The arbitrariness of the choice of 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 depend on the system's architecture that are beyond the scope of the Z.120 standard for message sequence charts. In a system where each process has multiple asynchronous communication queues, one can impose any arbitrary order on independently received messages, reflecting the order of *processing* the messages rather than their order of *arrival*. 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 *fifo* message queue, sets $e_1 \prec e_2$ in exactly the following

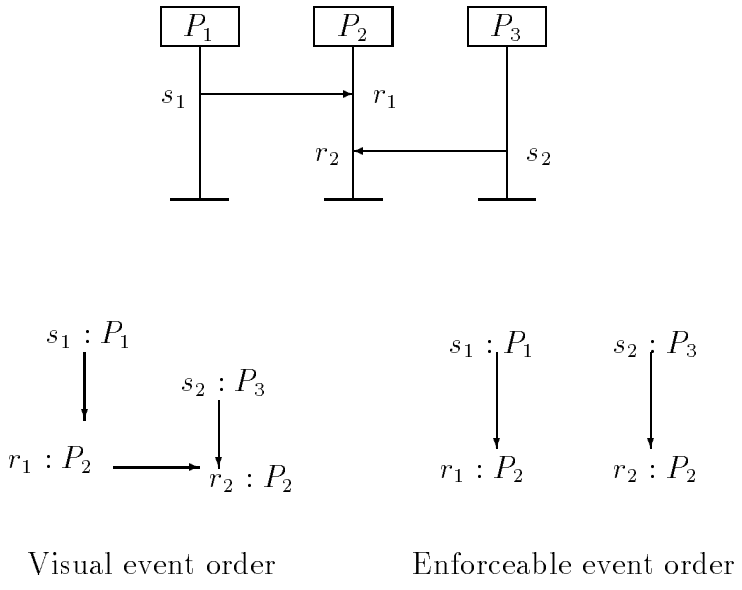


FIGURE 2. A message sequence chart, and two associated event orders

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, i.e., $<_c \cup \bigcup_{P_i \in \mathcal{P}} <_{P_i}$.

For example, for the message sequence chart 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 the figure. This order is termed 'visual', since it reflects the way the chart is drawn, but may not properly reflect the way in which the scenario shown could actually be executed, as explained below.

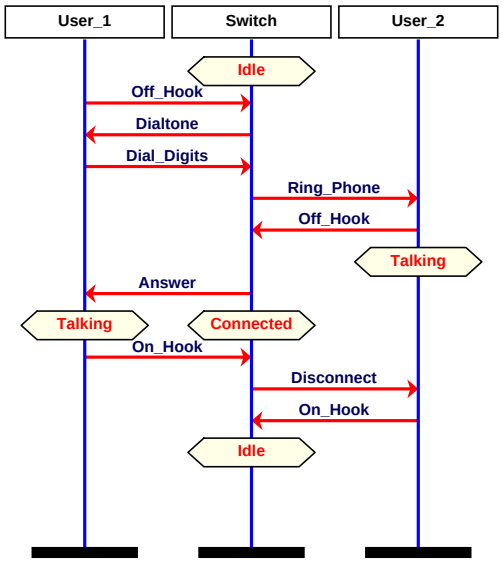


FIGURE 1. Example of a Message Sequence Chart

To allow for this type of verification, the tool provides the user with a choice of possible semantic interpretations of each chart, as will be illustrated below. These semantic choices are formally outside the scope of the ITU definitions, but they can directly influence the implementability of the requirements that are expressed.

The Analysis of MSCs

A message sequence chart can be formalized as a fivetuple $\langle E, <, \mathcal{P}, \mathcal{L}, \mathcal{T} \rangle$, 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
- $T : E \mapsto \{\mathsf{s}, \mathsf{r}\}$ maps each event to its type, i.e., *send* or *receive*.

The relation $<$ formalizes the *visual ordering* of events, as it is displayed in a message sequence chart. Thus, we have $e < f$ if e is a send event and f

a negative test-case, called a *template*. The tool is described in more detail Section 2.3.

The tools MSC and POGA are relatively light-weight tools that were implemented in about 4,500 lines of Tcl/Tk each [11], 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 the input for an existing state machine based verifier [7]. The translator is written in about 1,700 lines of C.

2.1 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 that is adopted varies from one application to the next. If a commercial word processor is used to construct time sequence diagrams, their semantics are alltogether lost, along with any capability to perform automated consistency checks. The International Telecommunications Union (ITU) has proposed a standardized representation for time sequence diagrams, called *Message Sequence Charts* or MSCs, in their recommendation Z.120 [6]. The standard representation applies to both the graphical elements of a message sequence chart and its machine readable form.

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

Vertical lines in the chart correspond to asynchronously executing processes in a logically or physically distributed system. Messages exchanged between these processes are represented by arrows. The tail of each arrow corresponds to the event of sending a message; the head corresponds to its receipt. Arrows can be drawn either horizontally or sloping downwards, but not upwards: time advances down the page.

Our MSC tool allows the user to construct and edit ITU compliant message sequence charts 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 be used to detect logical inconsistencies in the charts, such as potential race conditions between message arrivals [1].

2 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 in the form of discussions between systems engineers and key developers, together exploring the anticipated message exchanges between components. The goal of such discussions is to ascertain that the capabilities and the behaviors required of each component are consistent and achievable. During these discussions, the engineers often visualize their work with informat graphical message flow diagrams, and informal architectural views drawn with boxes and arrows.

The first tool we have built is called MSC [1, 4]. MSC, an acronym for Message Sequence Chart, supports the capture of message exchanges as machine readable, standardized [6] message sequence charts and provides for simple, automated consistency checks. The sequenc charts can be annotated with textual requirements or can be symbolically linked to a document to create a precise association between 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 Section 2.1.

MSCs, as standardized by the ITU, specify only non-branching message exchanges. MSC fragments can be combined in various ways in so-called hierarchical message sequence charts, that can contain conditional branching and iteration. A single path through such a hierarchical charts defines, what is sometimes called, a *use-case*, or also a *test-case*. As a use-case, each path corresponds to a simple concatenation of MSC fragments. Hierarchical message sequence charts can be decomposed naturally into finer levels of detail recursively. Our second tool, called POGA [4], is used to capture and analyze hierarchical message sequence charts. POGA is described in more detail in Section 2.2.

A more thorough validation of a library of requirements, formalized in message sequence charts, 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, that can be reused in future design cycles. The negative test-case feature can also be used to determine reliably if specific functionality is present in or absent from the documented requirements. Our third tool, called TEMPLE [9], allows the user to search a POGA database of hierarchical message sequence charts for matches of

of the 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. Due to the difficulty of capturing complicated branching scenarios accurately in a written document, systems engineers often avoid detailed descriptions to keep the requirements documents manageable. The descriptions that are included are often limited to the so-called *sunny day* scenarios, that is those scenarios that exclude all possibilities for failure or error. The exception cases are left as an exercise for the developers who will have to implement the system in conformance with the requirements. These exception cases, however, typically take the larger part of the development effort, and ultimately determine the quality and reliability of the final design.

Although the exception cases could impact both the overall architecture and the design of individual components, they are not dealt with until development is well underway. Aspects of the final implementation that address 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. None of the requirements documents normally contains this information.

The scant coverage of exception cases in requirements documents returns as a problem at the end of the implementation cycle, when acceptance testing is done. Test cases are created by manually translating textual requirements into test scripts. The resulting set of test-cases 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: MSC, POGA, and TEMPLE, that address the need for tool support by requirements engineers. These tools encourage a more complete capture of behavioral requirements, including exception cases, and provide organization and search capabilities for a large and complex body of scenario-based requirements. The tools support design validation techniques through automated consistency checking. The tools also allow for a straightforward reuse of design requirements for test-case generation.

The tools are designed to operate in a simple, intuitive manner that requires no extensive training. They aim to make design requirements accessible to all those 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 to automated indexing and catalogueing tools.

Design Tools for Requirements Engineering

Gerard J. Holzmann*
Doron A. Peled*
Margaret H. Redberg†

ABSTRACT Industrial software design projects often begin with a requirements capture and analysis phase. In this phase, the main architectural and behavioral requirements for a new system are collected, documented, and validated. So far, however, there are few tools that the requirements engineer can rely on to guide and support this work. We show that a significant portion of the design requirements can be captured in formalized message sequence charts, and we describe a tool set that can be used to create, organize and analyze such charts.

1 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 the 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 decisions on how new functionality is apportioned 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.

The medium of the systems engineer has traditionally been a written requirements document, illustrated with pictures of sample behaviors. Many

*Bell Labs, Lucent Technologies, 700 Mountain Avenue, Murray Hill, NJ 07974.
Email: {gerard,doron}@research.bell-labs.com
†Lucent Technologies, 101 Crawfords Corner Rd., P.O. Box 3030, Holmdel, NJ 07733
Email: mhr@lucent.com