An Automaton Interpretation of Message Sequence Charts

Peter B. Ladkin   Stefan Leue

University of Berne
Institute for Informatics and Applied Mathematics
Länggassstrasse 51
CH-3012 Bern, Switzerland
ladkin@iam.unibe.ch, leue@iam.unibe.ch

IAM 92-012

June 1992

Abstract

We give a precise semantics to Message Sequence Charts (MSCs), by interpreting MSC specifications by Büchi automata. The state transition graph is uniquely determined by the specification, but different automata may be defined to accept different sets of traces allowed by the specification. The precise automaton chosen therefore depends on reliability properties of communications assumed by the specification.

CR Categories and Subject Descriptors: B.4.4 Formal Models [Input/Output and Data Communications]; D.2.1 Requirements/Specifications [Software Engineering]; F.1.1 Automata; F.3.1 Specifying and Verifying and Reasoning about Programs [Logics and Meanings of Programs]; F.3.2 Semantics, Operational
General Terms: Algorithms, Design, Languages
Additional Key Words: Message Sequence Charts

Chapter 1

Introduction

Telecommunications protocol specifications are distinguished amongst general system specifications by an emphasis on communication amongst process instances rather than computation within a process, and by the relatively simple nature of the messages exchanged (often called signals). Message Sequence Charts or MSCs (also called Time Sequence Charts) are a form of system specification often used in specification of telecommunications systems. They are the subject of an international standardisation effort by CCITT [X92]. Although the syntax definition is quite well developed, the semantics is still in its early stages. Our purpose in this paper is to propose a precise mathematical semantics for MSCs.

We proceed via the definition of an abstract syntax in the form of a labelled graph, an ne/sig graph, whose two sorts of edges represent signals sent and process transitions. We show that a single such graph may be used to represent an MSC specification. From this graph, we obtain the transition graph of a Büchi automaton [Tho90] of global system states. Büchi automata may be defined with different acceptance criteria which we show are related to various reliability assumptions for signal transmission that may be made by the specification. Although reliability properties are thus necessary for a precise semantics, they are not currently included (except by default) in MSC specifications. The Büchi automaton defines the semantics of the MSC by specifying a set of traces (the accepted traces) which is taken to be the set of traces denoted by the specification.

MSCs describe the signal-passing behaviour of collections of communicating sequential processes. Computations, if any, which processes perform are not specified with MSCs. Events in an MSC are communication events. A simple example of an MSC is given in Figure 1.1, in which a system with three processes is specified. The first process sends a signal of type a to the second process, which upon reception sends a signal of type b to the third process, a signal of type c to the first process, and finally a signal of type d to the third process. The system terminates when all processes have terminated. MSCs assume that communication is asynchronous. Various reliability assumptions may also be made,

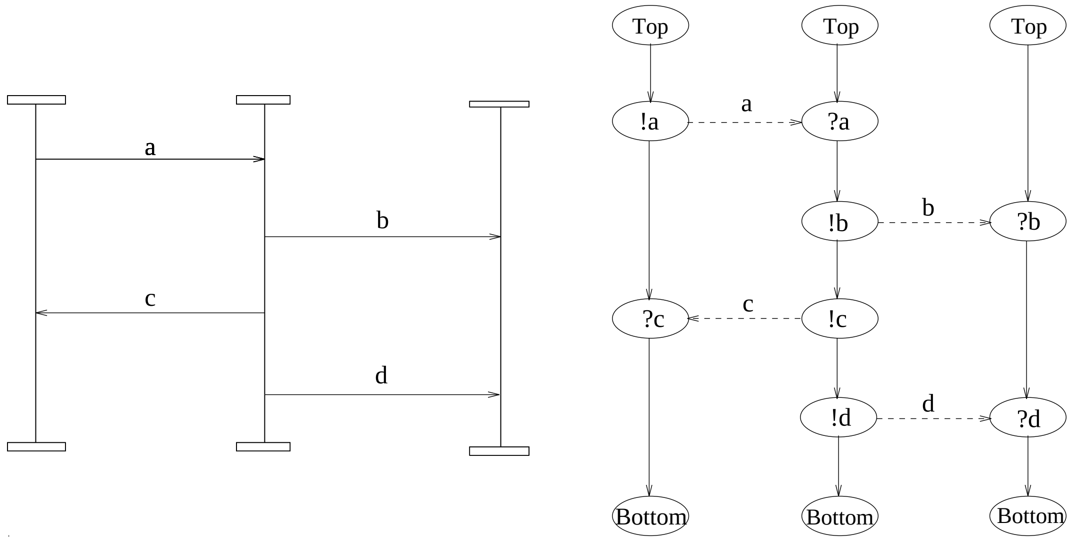

Figure 1.1: A simple MSC and its corresponding ne/sig graph

such as that all signals sent will eventually be received, and that there is no duplication of signals.

Extension of the basic MSC example allows a system to be specified by multiple MSCs, which contain labelled conditions at which MSCs may be 'glued' together. We call these *MSCs with Conditions*, or cMSCs.

The first step translates cMSCs into a single graph structure that provides a mathematical syntax for the specification. This labelled graph with two sorts of edges is called a *next-event/signal graph* or ne/sig graph. The second step obtains from the ne/sig graph the state transition graph of an ω-regular automaton. Finally, the definition of a Büchi automaton follows from this state transition graph by defining acceptance criteria from reliability properties of communication in the MSC specification. Different reliability assumptions yield different automata, all with the same underlying transition graph.

The advantages of this approach are:

- Ne/sig graphs have also been used to give an abstract syntax for other message-passing specification methods such as simple SDL (sSDL) [GHL$^+$92] and loop processes [LS91]. This versatility is evidence that they represent a natural syntactical abstraction useful for many different forms of protocol specification.

- Büchi automata methods have already been shown useful in distributed system specification [VW86b], [VW86a], [AS87], [AS89]. Use may therefore be made of the established connections between reliability properties and Büchi automata [AS89] to make explicit the reliability of communication in MSC specifications.

- By factoring the semantic definition into the transition graph definition and the acceptance criterion definition, we separate the reliability properties of the communication, which are not explicit in the MSC specification, from the set of global states of the system, which are determined from the MSC specification.

Interleaving Semantics and Traces. We consider the semantics of any specification to be the set of *traces* that the specification allows. A trace is an interleaving of all observable events of the system, which is consistent with the linear ordering of events within each process. Thus, we consider events as atomic (i.e. indivisible). It is beyond the scope of this paper to debate interleaving vs. partial-order semantics, so we merely note that interleaving semantics is appropriate for many important specification styles, including TLA [Lam91], and CSP [Hoa85]. We identify the set of system traces specified by the MSC with the set of accepted traces of the Büchi automaton we define.

Chapter 2

The *ne/sig* graph

We define general ne/sig graphs. For this section, we only require an intuitive understanding of MSCs, as obtained for example from Figure 1.1. Simple MSCs will be defined in the next section as particular ne/sig graphs. Nodes of the ne/sig graph correspond to MSC events i.e. heads and tails of signal arrows. There are two kinds of edges: *next-event* edges which represent the succession of events occurring in an individual process; and *signal* edges, which represent the possible communication paths of a signal from a process to others. Signal edges are labelled with the type s of the signal, and the nodes at the tail and head of a signal edge are labelled `send(s)` ($!s$) and `receive(s)` ($?s$) respectively.

In a general ne/sig graph, communication may be synchronous or asynchronous, channelled or broadcast, finitely or infinitely buffered, reliable or unreliable. Processes may be deterministic or non-deterministic, including parallel constructions or not, terminating or non-terminating. All these issues belong to the semantics of the specification style. Because it may be used to describe a mathematical syntax for different specification methods, a general ne/sig graph is neutral with regard to the meaning of signal edges, the types of communication, or the control structure of processes. Thus it is an abstract syntax appropriate for telecommunications specification methods: a syntax because it does not encode semantic assumptions, and abstract because it abstracts from the details of the syntax of any specific method. We choose ne/sig graphs satisfying particular properties to represent simple MSCs and MSCs with conditions (defined later), and in this use they are syntactic sugar. However, we derive from these a single ne/sig graph for an entire MSC specification, which can not in general be represented by a single MSC.

The semantics of MSC specifications is given by defining Büchi automata from the ne/sig graph representing a specification. Although the transition graph is uniquely determined by the ne/sig graph, the set of final states is chosen by considering reliability properties assumed for the communications in a specific MSC specification (so the automaton is under-specified by the raw MSC specification).

The ne/sig graph is mathematically speaking just a graph with labelled nodes and two

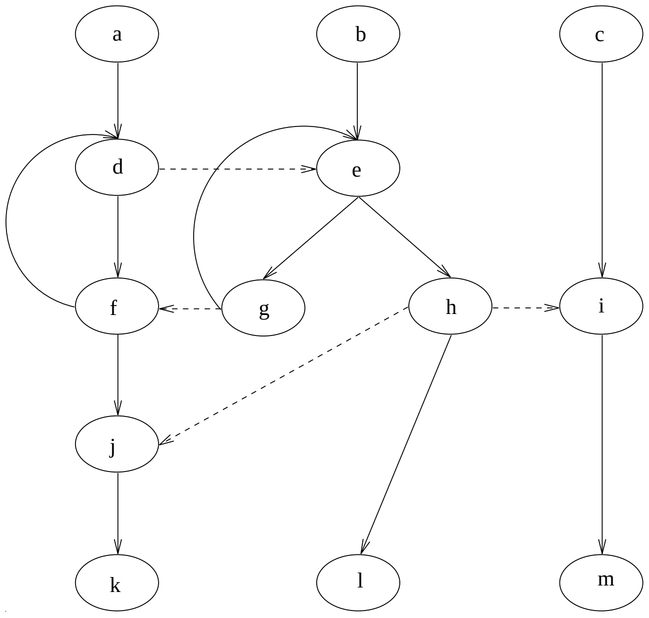

Figure 2.1: Next-event / signal graph (labels not shown)

different kinds of labelled edges. Figure 2.1 is a general example, in which *ne* edges are represented by solid arrows and *sig* edges by dashed arrows, but the labels are omitted. The graph structure may be used to represent behaviour such as *cyclic* process behaviour (the cycles d, f, d and e, g, e), indeterministic control choices (the branching next event relation at node e) or *broadcast communication* (multiple outgoing signal edges at node h). None of these features are allowed in individual MSCs. However, they may be effected by use of a set of MSCs with conditions.

The set of events with the signal relation alone forms a bipartite graph (sources are `send` events, and targets are `receive` events). Also, if we disallow self-sending, the signal relation is disjoint from the transitive closure of the next-event relation. Processes may be identified as components of the next-event relation.

In order to define an automaton corresponding to a particular specification, we need to introduce `start` nodes (in our example the nodes a, b and c), and for finite behaviours also `finish` nodes (nodes k, l and m), neither of which correspond to events, and therefore are not in the field of the signal relation. They have the special labels *Top* and *Bottom*. Other nodes which are not event nodes will be introduced to represent MSC conditions.

Signal and Event Types. If a signal edge is assigned label a, we say that the signal has type a, and it is a restriction on general ne/sig graphs the the label on the source of the edge is $(!, a)$, abbreviated $!a$ (`send a`), and the label on the target is $(?, a)$ ($!a$, `receive a`).

Mathematical definition of ne/sig graphs

Ne/sig graphs contain `send` and `receive` events, and also a set of `extra` 'events', to which start and finish nodes belong, and also nodes marking conditions in MSCs with conditions.

We first introduce some notation. Let $f \subseteq R \times R$ denote a binary relation over a set R, and S a set. We define the following *restrictions* and *operators* on a function f.

$$f \rhd S \triangleq \{(a,b) | (a,b) \in f \ \wedge \ b \in S\}$$

$$S \lhd f \triangleq \{(a,b) | (a,b) \in f \ \wedge \ a \in S\}$$

$$domain(f) \triangleq \{a \mid (\exists b \in R)((a,b) \in f)\}$$

$$range(f) \triangleq \{b \mid (\exists a \in R)((a,b) \in f)\}$$

$$field(f) \triangleq domain(f) \cup range(f)$$

We also extend $\lhd$ and $\rhd$ to n-ary relations by restricting to the first, respectively last, elements in an f-tuple in the obvious way. $(V, E, type, labels)$ is a *digraph with node labels* iff $E \subseteq V \times V$, $type : V \rightarrow labels$, and $labels = range(type)$. $(V, E, type, labels)$ is a *digraph with edge labels* iff $E \subseteq V \times V$, $type : E \rightarrow labels$, and $labels = range(type)$.

Ne/sig graphs. Let S, C and X denote arbritrary pairwise disjoint sets, the elements of which we call *sending* events, *receiving* events and *extra* nodes. Furthermore, let ST and ET denote arbitrary disjoint sets whose elements we call *signal* and *event* types. We define an *ne/sig* graph as a tuple $\mathcal{G}_{ne/sig} = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom)$, where $(S \cup C \cup X, ne, etype, ET)$ is a digraph with node labels and $(S \cup C, sig, stype, ST)$ is a digraph with edge labels satisfying the following conditions:

1. $sig \subseteq S \times C$ is a bipartite relation, where $S = domain(sig)$ and $C = range(sig)$

2. The set $ET = (\{!, ?\} \times ST) \cup \{Top, Bottom\}$ contains the *event types* (we write $!t$ for $(!, t)$ and $?t$ for $(?, t)$).

3. If the type of a signal is t, then the corresponding send and receive events are of type $!t$ and $?t$ respectively: $(a,b) \in sig$ then $(\exists t \in ST)(stype((a,b)) = t \ \wedge \ etype(a) = !t \wedge etype(b) = ?t)$;

4. Start nodes (defined to be nodes in the set $\{e \in X \mid (ne \rhd \{e\}) = \emptyset\}$) are of type *Top* and finish nodes (nodes in the set $\{e \in X \mid (\{e\} \lhd ne) = \emptyset\}$) of type *Bottom*: $e \notin ran(ne) \leftrightarrow etype(e) = Top$ and $e \notin dom(ne) \leftrightarrow etype(e) = Bottom$;

5. Every component of the ne relation graph contains only one start event: $(e, e' \notin ran(ne) \wedge (e, e') \in ne^{\sim}) \rightarrow (e = e')$.

Process type. A process is defined as a *connected component* of the *ne* relation. Since every component contains only one start node, we could define the set PT of all process types to consist of all start nodes, i.e. $ptype(a) = e$ *iff* $a \in range(\{e\} \lhd ne^{\sim})$. However, we shall later wish to identify processes across different cMSCs when we define cMSC composition, so we specify only that $ptype \subseteq S \cup C \cup X \times PT$ is a functional relation relating every node of the ne/sig graph to its process type, and the set of process types PT is disjoint from every other set in sight.

Chapter 3

Simple Message Sequence Charts

We first define an abstract syntax for *simple Message Sequence Charts* (sMSCs), which are simple in the sense that they do not contain conditions. Conditions allow the expression of infinite or non-deterministic behaviour alternatives. In contrast, sMSCs allow the description only of finite system behaviour.

Graphical representation. An MSC describes the behaviour of two or more communicating processes. We call vertical lines in the graphical representation *instance axes*. Each instance axis yields a total ordering of some communication events and represents the control flow of exactly one process[1]. Arrows between instance axes denote the sending and consumption of messages. According to [X92] the tail of an arrow is a `message output` symbol and the head is an `message input` symbol. We identify `message output` symbols with *sending* and `message input` symbols with *consuming* events.

sMSC instance axes do not *branch*, which means that there is no indeterminism in the process control flows. Furthermore, the instance axes contain *no cycles*, consequently process behaviours are acyclic and finite. There is only *1:1* communication in sMSCs, so at any `message output` symbol there is only *one* arrow leaving. *Self-sending* is disallowed[2], so an arrow always starts at one process and ends at another process.

Simple Message Sequence Charts. Given an sMSC in graphical form we define a set of *sending* events S of which each element corresponds to a `message output` symbol and a set of *consuming* events C of which every element corresponds to a `message input` symbol. We call the arrow connecting a `message input` and a `message output` symbol a `message` symbol. For simple MSCs, the ne/sig graph is so close to the MSC that it

[1]We only consider static systems without process generation and termination, therefore we do not distinguish between process *type* and process *instance*.

[2]The proposed standard does not include self-sending, except for the `timer` symbol[X92]. Though the `timer` symbol is an arrow with tail and head at the same instance axis, we suppose that its currently undefined semantics will not be equivalent to a self-sending.

may be regarded as just syntactic sugar. So we shall identify an sMSC with its ne/sig graph by identifying elements of S and C with their graphical MSC representation if they correspond in the above sense. Let $ne' \subseteq (S \cup C) \times (S \cup C)$ denote a *core next-event* relation and let $sig \subseteq S \times C$ denote a *signal* relation such that $(x, y) \in ne'$ iff y is a direct successor of x on some instance axis, and $(v, w) \in sig$ iff v and y are connected by a `message` symbol.

We define an sMSC as an *ne/sig* graph

$$\mathcal{G}_{ne/sig} = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom)$$

where

- $X = T \cup B$, $T \cap B = \emptyset$, $t \in T \Rightarrow etype(t) = Top$, and $b \in B \Rightarrow etype(b) = Bottom$: elements of T are called *top nodes* and elements of B *bottom nodes*;
- there is an injective functional relation $init \subseteq T \times (S \cup C)$ with $(t, a) \in init$ iff $(ne' \rhd \{a\}) = \emptyset$: we call $init$ an *initial completion* of ne';
- there is an injective functional relation $final \subseteq (S \cup C) \times B$ with $(a, b) \in final$ iff $(\{a\} \lhd ne') = \emptyset$: *final* is a *final completion* of ne';
- $ne = ne' \cup init \cup final$ (the *next-event* relation ne is obtained from the core next-event relation and its inital and final completions);
- $(\forall a \in E)(| dom(\{a\} \lhd ne) | = 1$ (there is no branching in the ne relation, we use E as previously defined to denote the set of *all* events),
- $ne^+ \cap id_E = \emptyset$ (there are no cycles in the ne relation),
- $(\forall a \in S)(| range(\{a\} \lhd sig) | = 1)$ (for every sending there is exactly one receiving event in the signal relation),
- $(\forall (a, b) \in sig)(ptype(a) \neq ptype(b))$ (there is no self-sending),
- $(\forall e \in T)(ran(\{e\} \lhd ne^{\sim}) = ran(\{e\} \lhd ne^*))$ (all elements in some component are *reachable* from the start node), and
- $(\forall x \in ST)(| range(field(dom(stype \rhd \{x\}))) \lhd ptype | \leq 2)$ (for any signal type, there is a *unique sender* and a *unique receiver* process).

Figure 1.1 shows a simple Message Sequence Chart using the usual notation, and its corresponding *ne/sig* graph according to our definition.

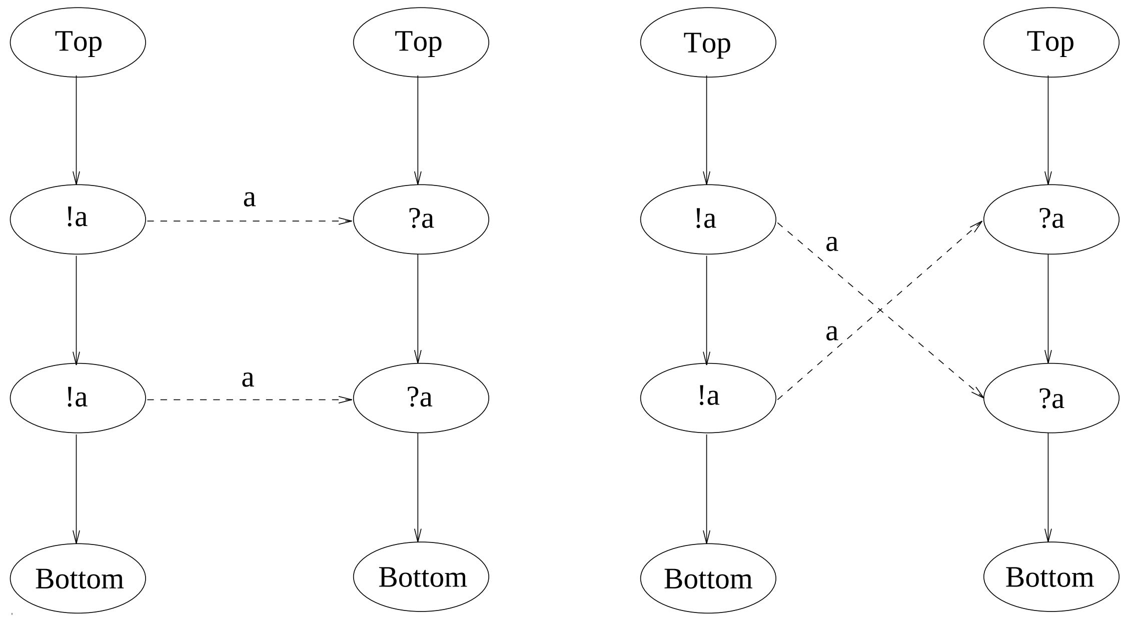

Figure 3.1: A cross-over MSC example, expressed as ne/sig graphs

A Cross-Over Example. The proposed MSC standard allows crossing of signals to occur[X92]. The two ne/sig graphs of Figure 3.1 derived from two simple MSCs represent different system behaviours. In both cases an identical type of signal is transmitted twice. The second case differs from the first in that a 'cross-over' of the messages is specified. The observable behaviour of each individual process is identical (one sends two a signals, the other receives two a signals), hence code for implementing either process will be the same in both cases. However, the set of traces (interleaved observable events) of the first MSC is $\{<!a, !a, ?a, ?a>, <!a, ?a, !a, ?a>\}$, and that of the second is is $\{<!a, !a, ?a, ?a>\}$, hence their meanings differ. A system exhibiting behaviour $<!a, ?a, !a, ?a>$ satisfies the first specification but not the second. However, a system exhibiting behaviour $\{<!a, !a, ?a, ?a>$ $\}$ and no other may satisfy either specification. Since process code is the same, the different meaning must be accounted for by differing properties of the environment. Thus properties of the environment may be influenced by MSCs, even though the environment is not explicitly represented. (Even if the environment is represented as a process axis, the example can be modified to produce a similar result.) Since this is allowed in the MSC specification style, we don't judge whether this is a bug or a feature. However, it raises for us also the question of whether MSCs specify merely observable behaviours.

Chapter 4

Message Sequence Charts with conditions

We now extend sMSCs to *MSCs with conditions* (cMSCs). We shall also consider *MSC specifications* for which we define an operation, *unfolding*, which yields the (single) ne/sig graph for the MSC specification. Conditions cover one or more MSC instance axes in between two `message` symbols. They allow the definition of *iterations, cycles* and *non-determinism* in control flow, as well as the specification of systems by sets of MSCs. The ne/sig graph with conditions may be regarded as syntactic sugar for cMSCs, but composition and unfolding lead to ne/sig graphs that do not correspond to individual cMSCs. In fact, we obtain one ne/sig graph per specification, and obtain the transition graph for an automaton from this single ne/sig graph.

A MSC with conditions (cMSC) is similar to an sMSC, but now various subsets of process instances may be interrupted, started or terminated by *conditions*. Identically labelled conditions in different MSCs may be 'glued together' (composed) to describe the behaviour of a system.

As before, we identify a cMSC with an ne/sig graph of a certain form, and then complete the ne/sig graph. This *ne/sig* graph has *condition nodes* (in addition to event nodes and start/finish nodes) which are introduced wherever a condition covers an instance axis in between two subsequent communication events. The *ne* relation is now defined not only between event nodes but also on condition nodes. The *sig* relation may not 'cut through' pairs of condition nodes. To group multiple condition nodes together to form a single condition we employ a *labelling* relation assigning a label to some set of condition nodes. We will identify conditions with these condition labels. See Figure 4.1.

S and C are as for sMSCs. We introduce the set I of *condition nodes*, such that S, C and I are pairwise disjoint. Condition nodes intuitively correspond to particular segments of instance axes in between directly connected `message` symbols. Let $ne' \subseteq ((S \cup C \cup I) \times (S \cup C)) \cup ((S \cup C) \times (S \cup C \cup I))$ denote a *core next-event* relation such

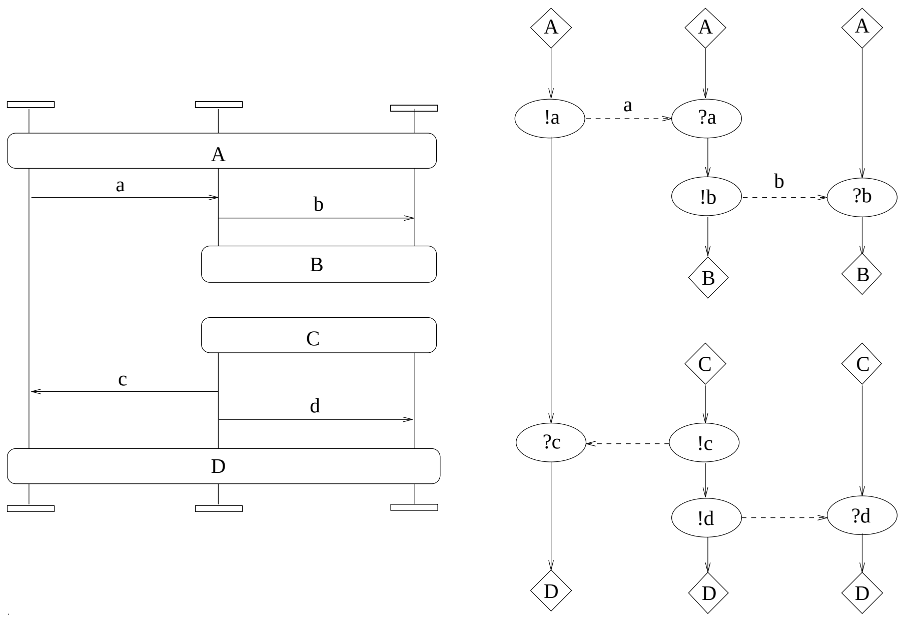

Figure 4.1: MSC with conditions and its corresponding ne/sig graph

that $(x, y) \in ne'$ iff y is a direct successor of x on some instance axis, and *sig* is defined as for sMSCs.

Definitions. Let S, C, ne' and *sig* be as for sMSCs. A cMSC is a labelled digraph

$$M = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom, CL, cond)$$

where

- (S,C,X,ne,sig,ST,stype,ET,etype,Top,Bottom) is an *ne/sig* graph,
- $X = T \cup B \cup I$ with T, B and I pairwise disjoint: we call elements of T *top nodes*, elements of B *bottom nodes*, satisfying the condition as for sMSCs, and elements of I are *condition nodes*;
- there is an injective functional relation $init \subseteq T \times (S \cup C \cup I)$, and an injective functional relation $final \subseteq (S \cup C \cup I) \times B$ as for sMSCs, and all constraints stated for sMSCs are satisfied;
- CL is pairwise disjoint from any other set defined, and $cond \subseteq I \times CL$ is a functional relation: elements of CL are called *condition labels* and *cond* the *condition labelling*;
- $(\forall l \in CL)(\mid domain(cond \rhd l) \mid = \mid range(domain(cond \rhd l)) \lhd ptype \mid)$ (every condition node belonging to a given condition belongs to a different process).

We define a *condition* to be a set C such that for some $q \in CL$, $C = \{c \in I \mid cond(c) = q\})$. The *set of all conditions* of a cMSC M is *conditions(M)*.

We define a *MSC specification* to be a set of cMSCs, specifically $\mathcal{M} = \{M_1, \ldots, M_n\}$ where each M_i is a cMSC. Evidently, MSC specifications are useful where condition labels of the various M_i overlap, and process instances in different MSCs in the set are identified. It is important to restrict the usage of conditions within some specification $\mathcal{M}$ in the following way: a condition label c_1 of some cMSC M_1 may only be identical to some condition label c_2 of some cMSC M_2 if they cover the same set of instance axes. Let C_i be the condition corresponding to label c_i:

$$(C_1 \lhd cond) = (C_2 \lhd cond) \;\Rightarrow\; (C_1 \lhd ptype_1) = (C_2 \lhd ptype_2)$$

The *set of all conditions* of an MSC specification $\mathcal{M}$ is *conditions*$(\mathcal{M})$.

Types of conditions.

- A condition c of some cMSC M_j is *global* with respect to some MSC specification $\mathcal{M}$ iff the set of all process types of $\mathcal{M}$ is equal to the set of process types of the condition nodes of c:
$$\bigcup_{i=1\ldots n} PT_i = (c \lhd ptype)$$
- A condition c of some MSC M is *initial* iff all its predecessor nodes in the ne relation are top nodes:
$$(domain(ne \rhd c)) \lhd etype \;=\; \{Top\}$$
- A condition c of some MSC M is *final* iff all its successor nodes in the ne relation are bottom nodes:
$$(range(c \lhd ne)) \lhd etype \;=\; \{Bottom\}$$

Continuations. Let $\mathcal{M}$ be a MSC specification, $M_1, M_2 \in \mathcal{M}$, C_1 a condition in M_1 and C_2 a condition in M_2, with $cond(x) = c_i$ for every $x \in C_i$. C_2 is a *continuation* of C_1 $(cont(C_1, C_2))$ iff

- $c_1 = c_2$ (the labels are identical)
- $global(C_1) \wedge global(C_2)$ (both conditions are global)
- $(\forall x \in C_1)(x \in range(final_1)) \;\wedge\; (\forall x \in C_2)(x \in range(init_1))$ (C_1 is a final condition and C_2 is an initial condition)

We shall restrict ourselves here to composition of MSCs via *global initial* or *final* conditions, in order to simplify the algebraic treatment of MSC specifications for this paper.

Composition. The *composition* of cMSCs is the 'gluing together' of cMSCs at common conditions (i.e. where one is a continuation of the other). During this process, condition nodes are removed. We also define the *composition graph* of an MSC specification.

Let $\mathcal{M}$ be an MSC specification, $M_1, M_2 \in \mathcal{M}$, and suppose the event sets of both cMSCs are disjoint (i.e. $S_1 \cap S_2 = \emptyset$, $C_1 \cap C_2 = \emptyset$). The *composition* of M_1 and M_2 is the cMSC $M' = (S', C', X', ne', sig', ST', stype', ET', etype', Top, Bottom, CL', cond')$ ($M' \triangleq M_1 \oplus M_2$) iff

- $\exists C \in conditions(M_1)\ \exists D \in conditions(M_2)\ cont(C, D)$ (there is a condition in M_2 continuing a condition in M_1),
- $S' = S_1 \cup S_2$, $C' = C_1 \cup C_2$ (the event sets are unified),
- $X' = I_1 \cup T_1 \cup B_2 \cup I_2$ (top nodes of M_2 and bottom nodes of M_1 are eliminated at the conditions where the cMSCs are composed),
- $ne' =$
$$(ne_1 - (ne_1 \rhd domain(cond \rhd \{C\}) - (ne_1 \rhd B_1))$$
$$\cup\ (ne_2 - ((domain(cond_2 \rhd \{D\})) \lhd ne_2) - (T_2 \lhd ne_2)))$$
$$\cup\ \{(a, b) \mid ptype(a) = ptype(b) \wedge a \in domain(ne_1 \rhd (domain(cond_1 \rhd \{C\})))$$
$$\wedge\ b \in range((domain(cond_2 \rhd \{D\})) \rhd ne_2)\},$$
(the new ne relation is obtained as the unification of the old ne relations minus those pairs which have the connecting condition nodes in their range or domain and minus those pairs which connect these condition nodes the top and bottom nodes; we then add new ne edges to connect M_1 and M_2)
- $sig' = sig_1 \cup sig_2$, $ST' = ST_1 \cup ST_2$, $stype' = stype_1 \cup stype_2$,
- $ET' = (\{!, ?\} \times (ST_1 \cup ST_2)) \cup Top \cup Bottom$, $etype' = etype_1 \cup etype_2$,
- $CL' = (CL_1 - C) \cup (CL_2 - D)$, and $cond' = (cond_1 - (cond_1 \rhd \{C\})) \cup (cond_2 - (cond_2 \rhd \{D\}))$.

Let $\mathcal{M}$ be an MSC specification. We define the *composition* relation $comp \subseteq \mathcal{M} \times \mathcal{M}$ such that $comp \triangleq \{(M_i, M_j) \mid M_i, M_j \in \mathcal{M}\ \wedge\ M_i \oplus M_j is\ defined\}$. From this we derive the *composition graph* $\mathcal{C} = (\mathcal{M}, comp)$ ($\mathcal{C}$ is a digraph whose nodes are individual cMSCs, and whose edges lead from a cMSC to its continuation).

Unfolding of MSC Specifications. We need an *ne/sig* graph obtained by the composition of all composable cMSCs contained in a specification. The composition of cMSCs according to a composition graph yields a single graph, paths through which correspond to system traces. Unfortunately, infinite traces could only be obtained in this manner from cMSCs (which specify a finite number of signals each) by infinite composition. This can only occur from an MSC specification when there is a loop in the composition graph. We therefore need a *finite* representation of the infinite composition. We define the *unfolding* operation on an MSC specification which composes a cMSC with all possible successors, intuitively by taking the composition graph and 'plugging in' the actual cMSC (without its initial and terminal condition nodes) in the appropriate place. The result of this operation is a general *ne/sig* graph with branching and cycles, and provides us with a single, finite, ne/sig graph structure corresponding to the specification.

Let $\mathcal{M}$ be a specification and let $\mathcal{C}$ denote the corresponding composition graph. We define the *ne/sig* graph $N_{\mathcal{M}} = (S, C, X, ne, sig, ST, stype, Top, Bottom)$ as the *unfolding* of $\mathcal{M}$ iff

- $S = \bigcup_{i=1,\ldots,n} S_i$, $C = \bigcup_{i=1,\ldots,n} C_i$, $X = \bigcup_{i=1,\ldots,n} X_i$,
- $ne =$

$$\begin{gathered}(\bigcup \{ne_i \cup ne_j \mid (M_i, M_j) \in comp\}) \\ -(\bigcup_{i=1,\ldots,n} ne_i \rhd (domain(cond_i \rhd CL_i))) \\ -(\bigcup_{i=1,\ldots,n} (domain(cond_i \rhd CL_i)) \lhd ne_i) \\ -(\bigcup_{i=1,\ldots,n} \{T_i \mid M_i \in range(comp)\} \lhd ne_i) \\ -(\bigcup_{i=1,\ldots,n} ne_i \rhd \{B_i \mid M_i \in domain(comp)\}) \\ \cup \{(a,b) \mid ptype(a) = ptype(b) \\ \wedge (\exists C \in conditions(M_i), D \in conditions(M_j))\, cont(C, D) \\ \wedge (\exists c, d)(c \in (domain(cond_i \rhd \{C\})) \wedge (a, c) \in ne_i \\ \wedge\, d \in (domain(cond_j \rhd \{D\})) \wedge (d, b) \in ne_j)\},\end{gathered}$$

  (The *ne* relation is obtained by a unification of all the component *ne* relations, minus all condition nodes, minus all *ne* pairs which contain top and bottom nodes over which a composition is performed, plus all those event pairs which need to be connected as a result of the composition of two cMSCs.)

- $sig = \bigcup_{i=1,\ldots,n} sig_i$, $ST = \bigcup_{i=1,\ldots,n} ST_i$, $stype = \bigcup_{i=1,\ldots,n} stype_i$.

Chapter 5

Global State Transition Graph

We define the notions of a *global system state*, of *enabling* a set of events in a global system state, and finally the *global state transition graph.*

Enabling. A *potential system state* (pss) $G \subseteq ne \cup sig$ is any subset of the union of the system's *ne* and *sig* edges. It is useful to define state transitions for pss's. An (actual) global system state (gss) will later be defined as a pss reached by taking the transitive closure of the transition relation from the *start* state (the set of all start nodes of the processes). Definition of a gss therefore waits upon definition of the transition relation.

Let $V \subseteq S \cup C$ denote an set of events and let G denote a potential system state. We call V *enabled in* G iff for every event in V one incoming *ne* edge is in G, and for every receive event in V the corresponding *sig* edge is in G.

$$enabled(V, G) \triangleq range((ne \rhd V) \cap G) = V \wedge (sig \rhd V) \subseteq G$$

$$enableset(G) \triangleq \{V \mid enabled(V, G)\}$$

Figure 5.1 shows an *ne*/*sig* graph with labels not shown. Let $G_1 = \{(c,e),(c,f)\}$ and $G_2 = \{(a,e),(c,e),(c,f)\}$ denote potential system states. Then $enabled(\{f\}, G_1)$ and $enabled(\{e,f\}, G_2)$. Note that in state G_2 two events are enabled simultaneously, which indicates an indeterministic behaviour alternative.

We now define how a system transits between different global system states in relation to a set of enabled events.

Construction of a successor state. Assume that a system is in an actual state G. The following operations need to be performed in order to obtain the successor state G'.

- Select the event a which is to be executed next from $enableset(G)$ (i.e. $\{a\} \in enableset(G)$),
- remove all *sig* edges pointing to a from G,

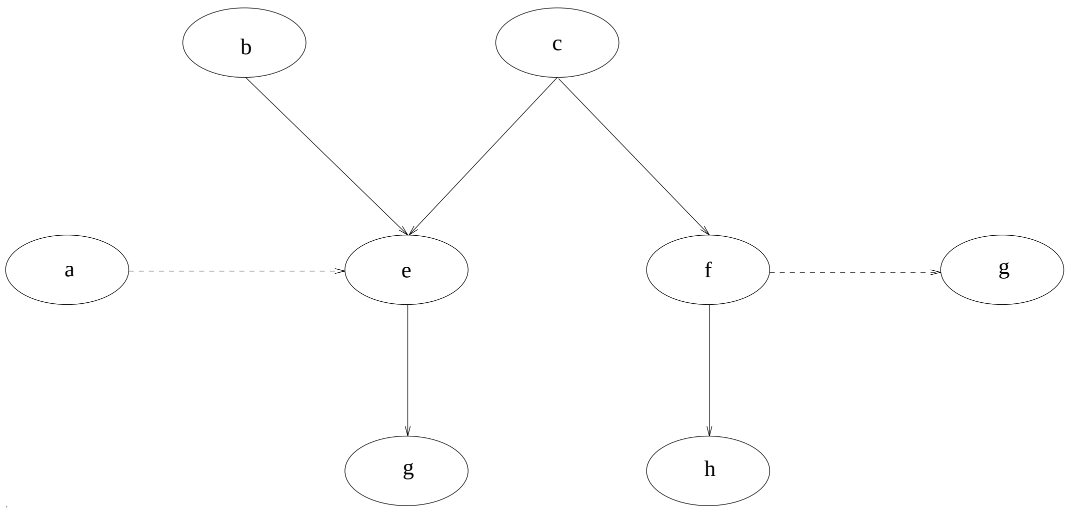

Figure 5.1: An *ne/sig* graph (labels not shown)

- remove all *ne* edges pointing to a from G,
- if a has a directly preceeding event b which has multiple outgoing *ne* edges, remove all edges from G which have source b,
- add all *ne* and *sig* edges which have source a to G.

Formally, we define $succ(G, G')$ where G, G' are pss's iff

$$G' = ((G \; - \; (sig \rhd \{a\})) - domain(ne \rhd \{a\}) \rhd ne) \; \cup \; (\{a\} \lhd (ne \cup sig))$$

The Transition relation. We define a *transition relation* on a pss G, an event a such that $\{a\} \in enableset(G)$ (a is enabled in G), and a successor state G' such that $succ(G, G')$. Let $N_{\mathcal{M}}$ denote an unfolding. The *global state transition relation* is $\mathcal{T}_{\mathcal{M}} \subseteq (ne \cup sig) \times (S \cup C \cup X) \times (ne \cup sig)$ such that

$$\mathcal{T}_{\mathcal{M}} \triangleq \{(G, a, G') \mid enabled(G, \{a\}, G') \; \wedge \; succ(G, G')\}$$

Global States and the Transition Graph. We now distinguish system states that actually can occur in a run of the system. A system starts in its start state, and transits according to the transition relation, so every actual system state (called *global system state* below) is in the transitive closure of the transition relation starting from the start state. Finally, we restrict the transition relation to global system states to obtain the transition graph of the system.

Formally, let $\mathcal{M}$ be an MSC specification, $N_{\mathcal{M}}$ the corresponding unfolding. Let $q_0 = \{(a, b) \in ne \mid (\{a\} \lhd ne) = \emptyset\}$ (the set of start states). We define G to be a *global system state* (gss) iff $G \in Q$, where $Q = \{q_0\} \lhd \mathcal{T}^*_{\mathcal{M}}$, where * is the transitive closure operator Q

is the set of all gss's). Let $T_{\mathcal{M}} = Q \lhd \mathcal{T}_{\mathcal{M}}$ (the transition relation restricted to gss's). The *global state transition graph* corresponding to $N_{\mathcal{M}}$ is $\mathcal{GSTG}_{\mathcal{M}} \triangleq (Q, q_0, T_{\mathcal{M}})$. The global state transition graph is almost an automaton, lacking only a definition of end states, to which we now turn.

Chapter 6

Global State Automata, Safety and Liveness

Definition of global state automaton. Let $\mathcal{M}$ denote a MSC specification and $\mathcal{GSTG}_{\mathcal{M}}$ the corresponding global state transition graph. We can define Büchi automata which transit between global system states, by adding to $\mathcal{GSTG}_{\mathcal{M}}$ a definition of a set of *final states F*. A *global state automaton for* $\mathcal{GSTG}_{\mathcal{M}} = (Q, q_0, \mathcal{T}_{\mathcal{M}})$ is $\mathcal{A}_{\mathcal{M}} \triangleq (Q, q_0, \mathcal{T}_{\mathcal{M}}, F)$, where $F \subseteq Q$ is a set of *final states*. Acceptance is Büchi acceptance [Tho90], namely an infinite word is accepted iff the automata cycles through some state in F infinitely often on the word (the alphabet is the set of events, e.g. $?a, !b$, and a word is thus a possibly infinite sequence of events, i.e. a possible trace).

Assume that the global state transition graph with 3 global states in Figure 6.1 is derived from some MSC specification, and $q_0 = S1$. The set of infinite paths through the graph is represented by the ω-regular expression

$$(!a(!b?b)^{\omega}) + (!a(!b?b)^{*}?a)^{\omega} + (!a(!b?b)^{*}?a)^{*}.(!a(!b?b)^{\omega}).$$

Selecting $F = \{S2, S3\}$ as end-states means that traces of the form $!a(!b?b)^{\omega}$ would be accepted. Traces in this class do not satisfy the liveness requirement that a sent message will eventually be received (the counter example here is $!a$ in the first and third terms in the sum). However, selecting $F = \{S1, S2\}$ ensures that only the *fair* traces of the form $(!a(!b?b)^{*}?a)^{\omega}$ are accepted. *Thus selection of a set of end states depends fundamentally on the liveness and safety characteristics assumed for a particular MSC specification.*

In the following, let F be a set of *end-states*, P_i denote the process with process type i. We briefly discuss relations between some safety and liveness properties and the definition of the end state set F, without formalism. We wish here only to establish the point that explicit reliability properties are crucial to defining the semantics of an MSC specification completely.

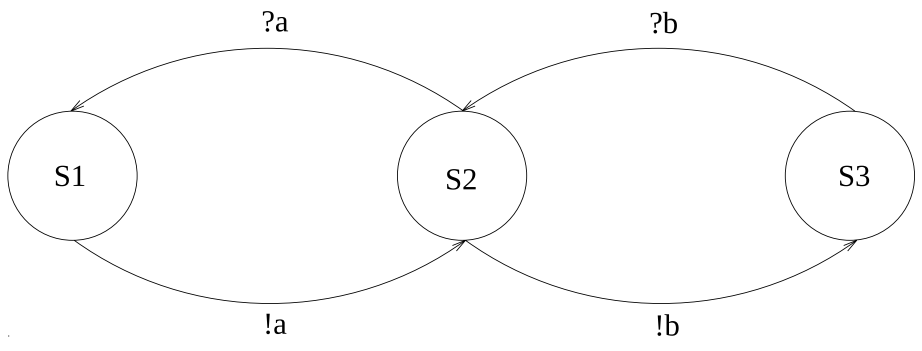

Figure 6.1: Global state transition graph

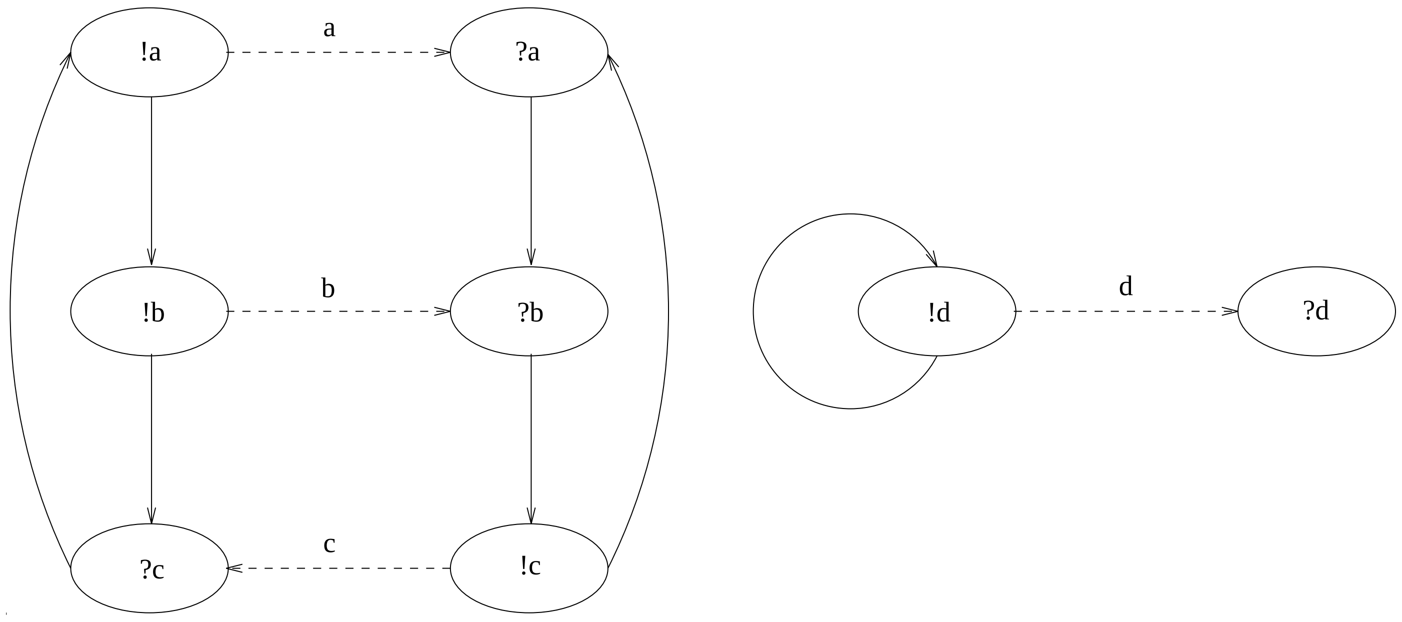

Figure 6.2: Strong and weaker liveness examples

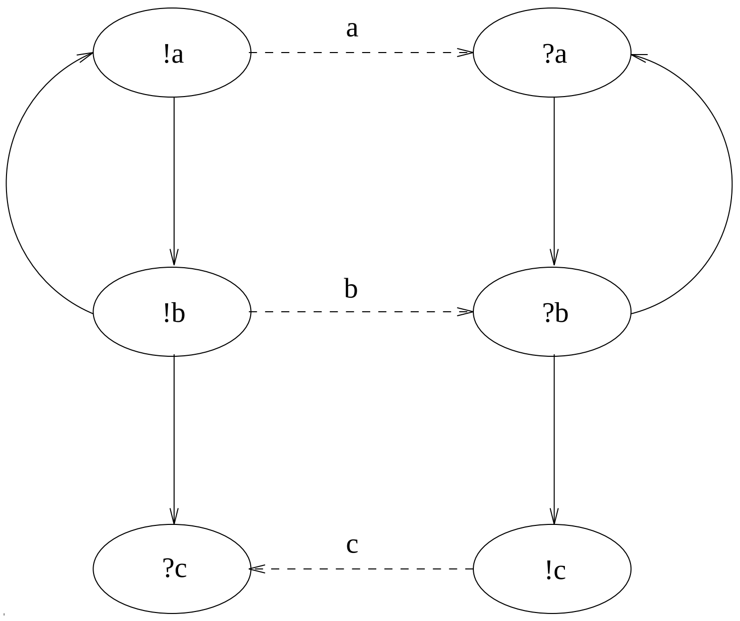

Figure 6.3: Strong liveness violated by branching

A strong liveness property for loop processes. Consider a system whose unfolded ne/sig graph contains precisely one cycle per process, and assume no branching. Then the cycles are terminal, i.e. there are no outgoing edges from the cycles (which would violate branching). Then the processes are *loop processes* as defined in [LS91]. Let $a_i \in P_i$ be some node in P_i's cycle, for $i \leq n$, chosen such that $G = \{a_i \mid i \leq n\} \in Q$ is a gss (we omit the easy proof that there is some such G). Let $F = \{G\}$. A trace is accepted by the automaton with final-state set F if and only if all processes iterate through their cycles infinitely often. This ensures strong liveness for the processes, as in the left-hand example in Figure 6.2, i.e. events preceding the cycle, and all events in the cycle, occur.

The example in Figure 6.3 shows that this condition does not ensure the strong liveness condition that all events eventually happen for examples in which the cycle is non-terminal (this can happen only if there is branching in the ne/sig graph). In this example, the language of receive-events can be described by the expression $(?a?b)^{\omega} \cup (?a?b)^{*}?a?b?c$ which denotes a set of finite and infinite regular sequences. In the infinite trace, the eventual reception of b is actually ensured by the strong liveness requirement that a is received infinitely often. But message c will then never be either sent or received. However, this example does satisfy a weaker 'strong' liveness property that all signals sent will eventually be received.

A weaker liveness condition. A weaker liveness property is to require: (*weak liveness*) for all processes which ever send there is a state in the set of send-states which also is an end-state. Whereas the previous 'strong' liveness property expresses a general claim about the transmission medium, equivalent to requiring for loop processes that infinite sending leads to infinite reception, the 'weak' liveness property only addresses the local behaviour of loop processes. For example, the system of loop processes described by the right hand part of Figure 6.2 satisfies the weak liveness condition, but not the strong liveness condition that all signals sent are received. Infinitely many signals are sent, but only one of the signals is ever received.

A final-states definition for loop processes which encodes this weaker liveness property is: If P_i has a cycle,let $b_i = \{a_i\}$, where $a_i \in P_i$ is any node in P_i's cycle, for $i \leq n$. If P_i has no cycle, then $b_i = \emptyset$. $G = \bigcup\{b_i \mid i \leq n\} \in Q$ (we again omit the easy proof that G is a gss), and let $F = \{G\}$.

Chapter 7

Topics for Further Research

Characterisation of finite bounded buffers Consider a finite bounded buffer capacity of the communication channel between two processes, i.e the channel is restricted by allowing at maximum n messages to be sent and not yet received, where $n \in N$. This situation requires the following *n-safety* property S_n: reception of the k-th message occurs before the $k+n$-th message has been sent. Let i range over the indices of system states, let a range over messages of arbitrary type and let $\#_i(m)$ denote the number of messages of type m received or sent in state i. S_n can then be formulated as follows:

$$S_n : (\forall a, i)(\#_i(!a) \leq \#_i(?a) + n)$$

A global state automaton reflecting this condition needs the ability to count sending and reception of events modulo a fixed integer n. This can be implemented by a finite-state-automaton, but whose states are not necessarily precisely the set of gss's, since some gss's need to be duplicated n times to encode the state of the buffer. It is a topic of ongoing research to supplement our automaton derivation in this way.

Proper delivery The *proper delivery* of sent messages along an infinite computation is described by the following property: at any point of a computation the number of messages sent of some type is greater or equal to the number of messages of that type received. Additionally we might require that for any point in the execution sequence there is a later point at which all previously sent messages are consumed. Formally,

$$L_{pd} : (\forall a, i \exists j \geq i)((\#_i(!a) \leq \#_i(?a)) \wedge (\#_j(?a) = \#_i(!a))).$$

It is easy to show that such a condition cannot be checked by a finite state ω- automaton, because it requires counting up to arbitrary large numbers. We need at least a counting automaton which has one counter for sendings and receptions of messages of each type. It is a topic of further research whether our method may be extended to accomodate counting ω-regular automata.

Chapter 8

Concluding remarks

We have provided a semantics of MSCs which proceeds by defining a single ne/sig graph for an MSC specification, and defining the transition graph of an automaton from this ne/sig graph. An automaton is defined by providing in addition a set of final states, which correlates the set of accepted traces of the automaton with reliability properties of the communication acts in the MSC specification. Such properties are not normally explicit in MSC specifications, and we conclude it may be wise to include them explicitly, to enable a precise interpretation to be assigned to each MSC specification.

We have discussed the relation between properties of the communication in an MSC specification, the underlying automaton model, selection of end-states, assumptions about the communication mechanism (e.g. finite or infinite buffers) and other formal description techniques (regular expressions, and there is a well-known connection between temporal logic formulas and Büchi automata). Clarifying different assumptions about the inherent properties of MSCs may therefore be approached by formal description techniques such as temporal logic. The connections between reliability properties of systems and temporal logic is shown in [MP90].

The connections between various end-state definitions and reliability properties of the communications in an MSC specification are the subject of continuing research.

Acknowledgements

Development of the ne/sig graph concept benefitted from discussions with Robert Nahm. The work was funded by the Swiss PTT under contract 233 with the University of Berne, project leader Prof. Dr. D. Hogrefe.

Bibliography

[AS87] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[AS89] B. Alpern and F.B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages*, 11(1):147–167, jan 1989.

[GHL+92] J. Grabowski, D. Hogrefe, P. Ladkin, S. Leue, and R. Nahm. Conformance testing - a tool for the generation of test cases. Project Report, project contract no. 233, funded by Swiss PTT, University of Berne, may 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, dec 1991.

[LS91] P.B. Ladkin and B.B. Simons. Compile time analysis of communicating loop processes. Technical report, IBM Almaden RJ 8488, nov 1991.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM Press, aug 1990.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, chapter 4, pages 132–191. Elsevier Science Publisher, 1990.

[VW86a] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symp. on Logic in Computer Science*, pages 332–344, 1986.

[VW86b] M. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs. *Journal of Comput. Systems. Sci.*, 32:183–221, 1986.

[X92] CCITT SG X. Draft recommendation Z.120: Message sequence chart. Submitted to CCITT, mar 1992.