Formal Aspects of Computing (1994) 3: 1–000

Interpreting Message Flow Graphs

Peter B. Ladkin[1] and Stefan Leue[2]

[1]Dept. of Computing Science, University of Stirling, United Kingdom
[2]Institut für Informatik, Universität Bern, Switzerland

Keywords: Message Flow Graph, Semantics, Büchi Automaton, Temporal Logic, Liveness

Abstract

We give a semantics for Message Flow Graphs (MFGs), which play the role for interprocess communication that Program Dependence Graphs play for control flow in parallel processes. MFGs have been used to analyse parallel code, and are closely related to Message Sequence Charts and Time Sequence Diagrams in telecommunications systems. Our requirements are firstly, to determine unambiguously exactly what execution traces are specified by an MFG, and secondly, to use a finite-state interpretation. Our methods function for both asynchronous and synchronous communications. From a set of MFGs, we define a transition system of global states, and from that a Büchi automaton by considering safety and liveness properties of the system. In order easily to describe liveness properties, we interpret the traces of the transition system as a model of Manna-Pnueli temporal logic. Finally, we describe the expressive power of MFGs by mimicking an arbitrary Büchi automaton by means of a set of MFGs.

1. Introduction

The purpose of this paper is to present a precise mathematical semantics for Message Flow Graphs (MFGs). MFGs were introduced in [LS91] as a means of analysing parallel code, in order to perform deadlock analysis and other optimisation at compile time. Analysis based on MFGs may be found in [LS92a],

Correspondence and offprint requests to: Peter Ladkin, Technische Fakultät, Universität Bielefeld, Postfach 10 01 31, 33501 Bielefeld, Germany.

[LS94b], and [LS94a]. MFGs also occur in telecommunications system description in the form of Message Sequence Charts [CCI92][1] and Time Sequence Diagrams [ISO91b], both International Standards. The semantics presented in this paper translates directly into a semantics for Message Sequence Charts, shown in [LL94], and similarly for Time Sequence Diagrams.

Structures which may be interpreted by MFGs are found in the following references. In formal or semi-formal use in telecommunications systems: [CC91] ('temporal message flow diagrams'), [CCHK90], [Sie88] ('message sequence charts'), [Tan89] (many examples), [VFV92] ('primitive sequences'); in object-oriented analysis: [RBP^{+}91] ('event traces'); in analysis of parallel code: [LS92a] [LS94b] [LS94a]. The object-oriented development methods of [SGW94] explicitly use Message Sequence Charts.

What is an MFG? Intuitively, an MFG is a graph representing concurrent processes exchanging messages. The nodes represent `send` and `receive` actions. There are two kinds of edges: *next-event* edges connect nodes to their successors within a process, and *signal* edges connect nodes to nodes in other processes with which they communicate. The essential property of an MFG is that (*) *each node is connected by precisely one signal edge to a unique node in another process.* The nodes may represent *atomic events*, as in Message Sequence Charts, or *communication statements* of a concurrent program as in [LS92a]. Pure computation statements inside processes are ignored in the MFG, which focuses merely on the communication behavior of the processes, e.g. in order that deadlock and reachability analyses may be performed, or that the progress of the system may be checked for conformance to its specification.

Figure 1 gives a simple intuitive picture of message passing between processes, in the style of a Message Sequence Chart or Time Sequence Diagram. Processes are represented by vertical lines with time progressing downwards, and messages exchanged between processes are represented by horizontal or sloping directed lines. The MFG corresponding to this picture is on the right in the figure.

The essential property (*) is guaranteed for such representations of message-passing as Message Sequence Charts and Time Sequence Diagrams. However, for other structures such as processes with a single non-terminating loop, proving the existence of an MFG describing the progress of the system can be non-trivial. An algorithm for constructing a minimal MFG from a collection of concurrent so-called loop processes was given in [LS92a]. Proof of correctness of this algorithm rests on some non-obvious mathematical combinatorial conditions.

Motivation for this Work. Our motivation for this work came from two different directions. We believe that it is a touchstone of a worthwhile abstraction that it applies in different contexts. Firstly, it was demonstrated in [LS91] and [LS92b] (summaries in [LS94b], [LS92a], with the complete material in [LS94a]) that MFGs are very useful in deadlock and reachability analyses of parallel code. The MFGs were rather simple, involving loops but no branching. To extend the analysis, it became clear that some mechanism to keep track of branching was required. Secondly, in apparently unrelated work, we wanted to provide a rigorous semantics for Message Sequence Charts and Time Sequence Diagrams, which are

[1] We note that the ITU-TS has taken over the responsibilities of the former CCITT organisation.

widely-used informal industrial description methods for some aspects of telecommunications. See [LL94] for a summary, and [LL93] for details of this semantics, which uses 'ne/sig graphs', a form of MFG. We shall not repeat here the well-known arguments for rigor and unambiguity in system description, since this is preaching to the converted. Methods such as Z, VDM and LOTOS followed a path from academia to industry. In contrast, we chose to precisify methods already in industrial use. MFGs were used as an abstract syntax, and this work necessitated that branching and control-flow conditions were handled. Given that MFGs have proved useful in different contexts, a natural next step is to define an unambiguous formal interpretation of each MFG, hence the present work.

1.1. Requirements for the Semantics

Our requirements for the MFG semantics are that traces are interleavings, and that the semantics is finite-state. We discuss both requirements here.

Traces are Interleavings. We consider a semantics to be a precise determination of which execution *traces* a description allows. Since the MFG focuses on communication events, this is all we represent in a trace. Internal process computation is ignored in the MFG, thus in the semantics, although it can easily be added if desired. We use an interleaving model, in which a trace is an interleaving of all observable atomic events of the system which is consistent with the linear ordering of events within each process, from the point of view of a global observer. Interleaving models are used for many important specification styles, including TLA [Lam94], CSP [Hoa85], and LOTOS [ISO88]. Partial order models are often seen as an alternative to interleaving models. We believe much of this contrast is superficial, but it is beyond the scope of this paper to discuss this issue, and we refer the interested reader to [Lad93].

Finite-State Semantics. Finite-state interpretations are often used for verification of telecommunications system software. Such interpretations first define the set of global system states, alternatively an automaton, whose accepted language is identical with the set of system traces allowed by the specification. This collection of global system states must be finite, since various variations on exhaustive search are used to verify safety properties, and sometimes also liveness properties. Such methods can exhibit a high degree of automation, and have been used successfully to validate systems with up to 10^{14} states [CD93]. Justification for this approach may be found in [Hol91]. See also [CK91] [LS92c] for other practical finite-state techniques. In contrast, non-finite-state methods usually employ theorem-proving techniques which are comparatively human-labor-intensive, are often research vehicles, and currently find favor in verification of small, complex, critical parts of safety-critical systems [RvH93]. It may be preferable to mix the two sorts of techniques to obtain the advantages of both [Lam92].

There is another reason of principle to require a finite-state semantics. This principle is based on inquiring what system information is explicit in concurrently-running processes, and which information is hidden. We argue that *the explicit information available in an MFG allows only finitely many global system control states to be identified.*

The argument proceeds as follows. Each individual process control is a finite-state device with respect to `send`s and `receive`s. Irrespective of its size, a finite

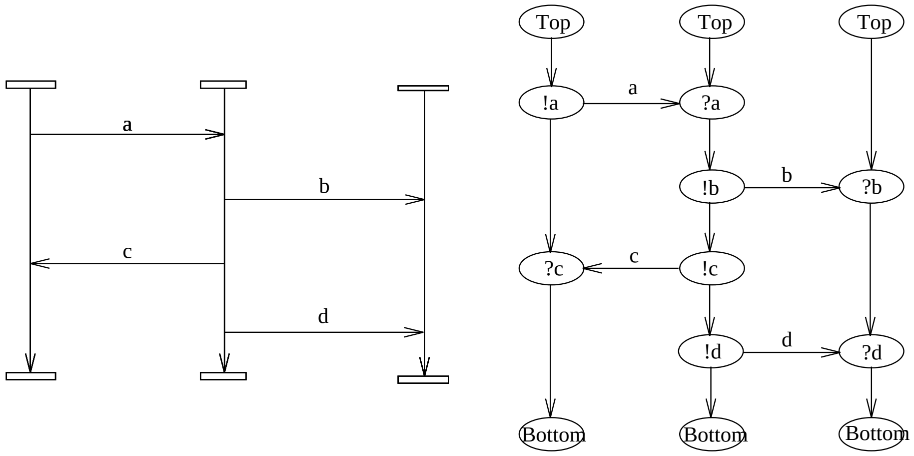

Fig. 1. A simple flow diagram and its corresponding formal MFG

state device can only remember a bounded computation history. Suppose, as the system runs, communication channels are compromised (someone cuts a cable). Also assume the processes themselves are not compromised. A consistent state must be reconstructed. Each process must be asked its state, namely where it thinks its control is, and what it remembers from what it has done. No other information may be assumed to be available. The system itself can have been operating for a very long time, much longer than the bounded memory of any single process. What can be reconstructed from the memories of the processes is bounded, no matter how long the system has been running previous to the fault. Hence, two such failures which result in the same local states to the processes are equivalent from the point of view of the potentially knowable state of the system. So each such equivalence class can be identified with a *global state*. Since there are finitely many finite-state processes, the global states are some equivalence (probably the identity) relation on a subset of the cartesian product of the state spaces of the individual processes, and thus there are only finitely many global states. A conservative upper bound to the number of these states is the size of this cartesian product.

Büchi Automata. Since traces may be infinite, we need a finite-state automaton which accepts infinite strings. The Büchi automaton has been used in the determination of safety and liveness properties of distributed systems [AS87], [AS89b]. Büchi automata are similar to ordinary finite automata, except for the acceptance condition. A (possibly infinite) string is accepted by a Büchi automaton just in case the automaton passes through a final state unboundedly often on the string (this definition also works for finite strings, normally turned into infinite strings for this purpose by simply repeating the last item for ever). Given an MFG, we show how to define a Büchi automaton and identify the set of system traces denoted by the MFG with the set of accepted traces of the automaton. We also reverse this interpretation by showing how, given an arbitrary Büchi automaton, it may be simulated by some MFG. In this sense, MFGs are equally expressive with Büchi automata, providing additional evidence that a Büchi automaton semantics is natural. Büchi automata are introduced and defined in Section 7.

However, we do note that although Büchi automata are the most well-known of ω-automata, they do not suffice to define all the desired liveness properties of MFGs, according to our construction, as shown by an example in [LL95]. It may be preferable to use other definitions of *acceptance* than the Büchi definition, but it is beyond the scope of this paper to discuss this issue in depth. The interested reader may see [Kur92] for an extended discussion of *L-automata*.

Complexity. In order to represent communicating systems by a single MFG where possible, it's necessary to define a *composition* operation on MFGs, whereby one obtains a single MFG from a collection of simple MFGs representing fragments of behavior. The single MFG obtained from a composition may be exponential in the size of the arguments [LS92b], and it should be obvious that the global system state graph may be exponential in the size of this MFG. But this complexity should be expected. There would be little point in using MFGs for description if it were 'just as easy' to write down a global system automaton directly. The MFG description can be regarded as shorthand for the larger object, and used in this way, as shown in [LS91]. One should expect that a formal semantics can refer to very large structures, which are still mathematically accessible without explicit representation. The single MFG is a structure of comparable size to the reduced state-graph structures used in [GW92], based on trace theory [Maz87]. However, it is a more picturesque, and therefore we would argue more intuitively appealing, structure.

Synchronous and Asynchronous Communication: Definitions. MFGs are most useful in an environment in which the sender and receiver of a process are statically determined, as in CSP or OCCAM, but unlike ADA or Remote Procedure Call. In the communication we consider, there is a single sender and a single receiver. Thus, all communication is *transparent two-process* in the terminology of [LS92b]. Synchronous communication in this paper is an atomic action participated in by two processes; sender and receiver block if one of them is not ready. Sender action and receiver action in asynchronous communication are separate atomic actions. Sender never blocks, while receiver blocks if all sent communications have already been received.

1.2. Handling Synchronous Communication

The constructions in this paper are given for asynchronous communication, and we show that synchronous communication is a special case. Minor modifications only are required to incorporate synchronous communication, and mixtures of synchronous and asynchronous message passing are handled easily, as shown in Section 11. We provide here some reasons for wanting to include synchronous communication signals.

Firstly, arguments given in [Hoa85], [Mil89] suggest that effective formal methods are much easier to devise for communication relying on synchronous primitives. Synchronous primitives are useful as an abstraction. The technique for deriving MFGs in [LS92b] relies on synchronous communication.

Secondly, many specification and description languages rely on synchronous communication primitives, for example the process-algebra-based specification languages *CSP* [Hoa85], *CCS* [Mil89] and *LOTOS* [ISO88], the CSP implementation language *OCCAM* [Inm84], and the family of so-called *synchronous languages* such as *ESTEREL* [BG88], [BG92], *SIGNAL* [BLJ91, BL90] and *LUSTRE* [BB91]. For such formal methods to avail themselves of MFG analysis techniques, it is necessary that synchronous primitives be handled in MFGs.

Thirdly, within the context of telecommunications systems conforming to ISO OSI [ISO84], synchronous events naturally occur at the interface between service layers, because the same event is looked at in two different ways by the two different layers. This point may be expressed another way, that the relation between different levels of abstraction in the design of a communications system can elegantly be expressed by means of synchronous communication primitives. It is closely related to the first point above.

Fourthly, there are examples where synchronicity and asynchronicity co-exist in one specification style, so for example in the dialect *ESTELLE** ([Cou88, Cou89]) of the specification language *ESTELLE* ([ISO87]). In a suggested extension of SDL with synchronous communication primitives ([Hog89]), it is argued that synchronous communication mechanisms are needed in open systems specifications.

1.3. Overview of the Paper

We introduce MFGs in Section 2. We describe the composition or unfolding of MFGs, with conditions to indicate non-local choice of control branching in processes, in Section 3. Section 4 formalises the previous informal discussions. In Section 5 we obtain a *global state transition graph* (GSTG) from an MFG. A GSTG is like a finite-state automaton but lacks definition of end-states. Section 6 formalises the GSTG notions. We consider end-state definitions in Section 7. Each possible end-state definition gives a Büchi automaton. The MFGs under-define the resulting automaton, in that end-state definitions are related to different liveness properties not explicit in the MFGs. We show in Section 8 how these may be made explicit via a connection with temporal logic, in which these properties may be formulated. In Section 9 we describe this connection formally. We discuss some properties expressed in temporal logic which all MFGs satisfy, and some potentially desirable ones which some MFGs might be required to satisfy in some uses, in Section 10. We then show in Section 11 how synchronous communication can be accommodated along with asynchronous communication in MFGs. We also note that the occurrence of synchronous communication in an MFG can simplify the liveness analysis. Finally, in Section 12 we show how to simulate an arbitrary Büchi automaton with MFGs.

We abuse notation mildly by using the phrase 'message `A`' when we really mean 'instance of a message of type `A`', which is an awkward, although more accurate, phrase. Simple 'type-checking' can resolve any infelicity engendered by our use of the shorter phrase.

2. Some Features of MFGs

Figure 1 shows a *simple* MFG (without conditions) describing a system with three processes. In the intuitive picture on the left, similar to a Message Sequence Chart or a Time Sequence Diagram, processes are represented by vertical lines, and the signals sent between processes are represented by horizontal arrows. Communication is asynchronous. The junction between a vertical process line and a horizontal signal line represents an event at which a signal of the type specified is sent or received by the process. In each process axis, the events are temporally ordered from top to bottom. 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. The basic idea of the MFG on the right is that the events are made explicit as nodes, and the process control-flow edges and signal edges are explicit relations on the nodes.

MFGs have two kinds of edges, *next event* (ne) and *signal* (sig) edges, representing the signals and the progression of processes between events. The nodes represent events, and are labeled with the event type. We use a variant of a common notation. The event node at the tail of a sig edge must be labeled with $!a$ (`send` a message of type a), for some symbol 'a' denoting the message type, and the event node at the head with $?a$ (`receive` a message of type a), for the same 'a'. (In some uses, it might be preferred to label the sig edge with a and omit the node labels.) An MFG has `start` nodes (in the domain but not the range of the ne relation) labeled *Top*, and maybe `end` nodes (in the range but not the domain of ne) labeled *Bottom*. (In the MFG examples in this paper, we also write a lower-case letter within a node to allow us to refer to that node in the text. These additional identifying letters do not occur in the MFG itself.)

Different types of communication can have contrasting features, for example synchronous or asynchronous, channelled or broadcast, finitely or infinitely buffered, reliable or unreliable. Similarly, processes may be deterministic or non-deterministic, including parallel constructions or not, terminating or non-terminating. Because an MFG may be used to describe a mathematical syntax for the message-passing features of different specification methods, the graph is neutral with regard to the meaning of signal edges, the types of communication, or the control structure of processes. However, it does determine statically the sender and recipients of each communication. Thus it is an abstract syntax: a *syntax* because it encodes only minimal semantic assumptions, and *abstract* because it abstracts from the details of the syntax of any specific application. The semantics arises from how the MFG is interpreted, as a particular kind of global state transition system. One major purpose of the MFG is to be able to derive a single MFG, where possible, from a collection of simpler descriptions of communicating processes. To this end, we define conditions in MFGs, and a composition operation which allows MFGs to be 'joined' at these conditions. This allows us to define the (partial) use of conditions as conceived in the Message Sequence Chart standard [CCI92], but as we note in [LL95], unrestricted use of conditions seems to entail that the environment has powerful implicit properties, such as the ability to retain an unbounded control history of each process, accessible by other processes. We don't pursue these questions further in this paper.

Conditions may be used in MFGs to specify non-determined behavior, such as conditional branching ('`if .. then .. else`') or loops ('`while .. loop ..... endloop`'). Figure 3 shows MFGs with conditions, represented by the diamond-shaped nodes. The idea is that control that arrives at a condition node may continue in another MFG from a condition node with an identical label, as if the MFGs were 'joined' at these condition nodes. Thus different MFGs starting with identical condition nodes provide different ways to continue control. (We are not

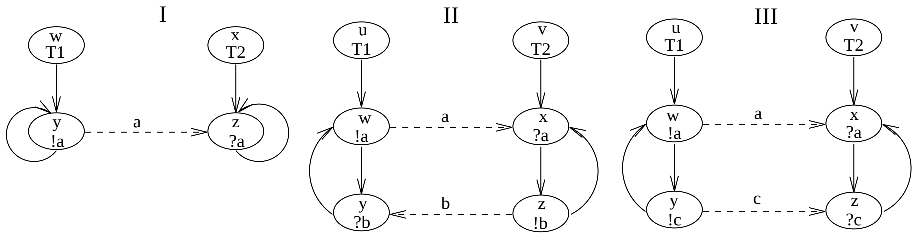

Fig. 2. MFGs I, II and III

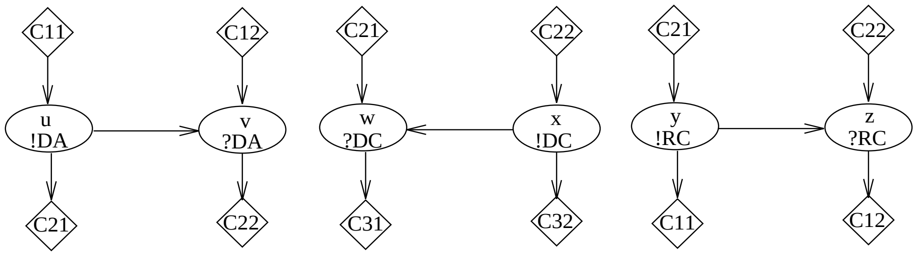

Fig. 3. MFGs with conditions

totally convinced that this can be done without difficulty, as noted in [LL95]. Nevertheless, we provide in Section 6 the apparatus to effect it.)

3. Non-Local Choice

MFGs such as that in Figure 3 induce multiple possible traces. From conditions `C21` and `C22` in the leftmost MFG, two different possible execution paths are possible, represented by the middle and the rightmost MFGs. This collection of MFGs thus signifies branching control, represented by the two *ne* out-edges from each of nodes `u` and `v` in the composition in Figure 4. These control branches must be coordinated according to the intuitive meaning of the three MFGs with conditions, in that if one process takes one branch, the other process is constrained to take its corresponding branch. This non-local coordination of branching is made explicit in the syntax of MFGs by including choice diamonds on the out-edges from choice nodes, with the constraint that all processes may take only out-edges that bear the same symbol (processes without the appropriate symbol on an out-edge are not so constrained). In the real world, this synchronisation must be accomplished either locally by each process, or explicitly by message exchange, or through some sort of coordination with the environment. In the example in Figures 3 and 4, synchronisation cannot be achieved by methods purely local to each individual process. Each process must somehow decide whether it is to `send` a signal, or to await one. If one process decides to `send`, it must ensure that the other is waiting to `receive`, and vice versa. Thus coordination is required. Below and in Section 5.1 we note that history variables or their equivalent are needed to handle control branching that cannot be achieved by local means.

In this section, we illustrate MFGs with predicates which provide, in a natural way, the non-local synchronisation required for choosing branch of control,

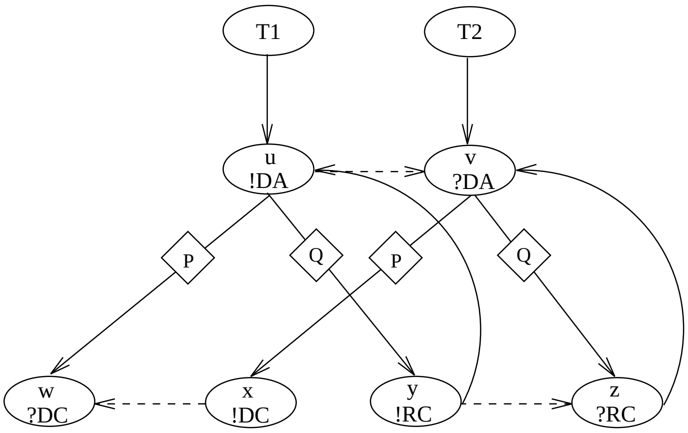

Fig. 4. MFG with non-local-choice nodes

without using explicit messages. In Section 5.1, we define the transition relation for these 'MFGs with non-local choice'.

Our example of non-local control branch synchronisation given in this section is very simple, using just three different global conditions at the beginning and at the end of three sequence charts.

An Example

The example in Figure 3 could arise from part of a confirmed data exchange between two communication partners. Both partners start in a global *connected* state. Data may then be transferred by a *data request PDU* (`DA`). Upon receipt of the data the second process may acknowledge the receipt by a *data confirm PDU* (`DC`). Alternatively, if no `DC` is received, the data sending process may request acknowledgement from the receiving process by sending a *request for confirmation* PDU (`RC`) and then returning to the *connected* state, where for example the previous `DA` PDU may be retransmitted.

The first and second processes must coordinate to decide whether they are going along the `DC` route or along the `RC` route. According to the obvious intuitive meaning of the MFGs with conditions in Figure 3, the resultant MFG must specifically disallow that one process could follow its left branch and the other its right branch.

We denote this required control synchronisation by labeling *ne* edges at branches with predicates. We assume that there is an unbounded collection of predicate symbols $P_1, P_2, \ldots$, different from all other symbols used. Given a condition symbol for which there exist at least two MFGs starting with that symbol (e.g. the symbol `C2x`, where `x = 1,2` in Figure 3), we label corresponding *ne* edges with predicate symbols as we construct the unfolding. For example, in Figure 4 we label edges corresponding to a transition through `C2x` as we unfold. The labels are shown in the figure using diamonds on the edges (these diamonds here represent labels, not conditions). The labels used on the *ne* edges are predicates from the list $P_1, P_2, \ldots$ that so far have been unused in the unfolding construction, say the first such ones. Labels are the same if each branch of each process with that label arises from the same MFG argument. In our example, labels `P` and `Q` used in Figure 4 would be respectively P_1 and P_2 according to this scheme, `P` being used to label the branches from the second MFG and `Q` those arising from the third MFG.

The modifications to the definition of MFG to allow *ne* edge-labels, and the formal definition to accomplish this method of handling control branching in MFGs formed by composition are straightforward. We omit these technical details from Section 4 for reasons of brevity.

4. Formal Definition of MFGs

This section provides the technical definitions of the concepts introduced in the previous section, and may be skipped on a first reading.

Preliminary Definitions

Most of our notation is fairly standard, and is somewhat Z-like [Spi89]. Let $f \subseteq R \times R$ denote a binary relation over a set R, let $x, y \in R$ and S a set. We define the following *restrictions* and *operators* on a relation f.

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

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

$$(*, y)_f \triangleq f \rhd \{y\}$$

$$(x, *)_f \triangleq \{x\} \lhd f$$

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

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

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

In $(*, y)_f$ and $(x, *)_f$, we sometimes omit the reference to the relation f if this is clear from the context. We also extend $\lhd$ and $\rhd$ to n-ary relations by restricting to the first, respectively last, elements in an n-tuple in f 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 $labels = range(type)$.

A relation f is *functional* if and only if each element in its domain is related to a unique element in its range. A relation f is *injective* if and only if it's functional, and furthermore an element in its range is related to at most one element in its domain. A relation f is *bijective* if and only if it's functional, and furthermore every element in its range is related to exactly one element in its domain. We use f^+ to denote the transitive closure of a relation f, and f^* to denote the transitive reflexive closure of f.

Message Flow Graphs

We have remarked previously that we may define a Message Flow Graph with either sig labels or node labels, and that either may be useful. For convenience, we define an MFG formally with both, along with a coherence condition saying that the *sig* label had better say what the corresponding node labels say, and *vice versa.*

Let S, C and X denote arbitrary 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 (also disjoint from S, C and X), whose elements we call *signal* and *event* types. We define a *Message Flow Graph* as a tuple

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

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 (necessarily bipartite) bijective relation, where $S = domain(sig)$ and $C = range(sig)$ ($\mathcal{G}$ satisfies the condition (*));
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. Every component of the ne relation graph contains at most one start event: $(e, e' \notin range(ne) \wedge (e, e') \in ne^*) \rightarrow (e = e')$.

A *Basic MFG* (bMFG) is an MFG which satisfies the following additional condition:

- 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*: $e \notin range(ne) \leftrightarrow etype(e) = Top$ and $e \notin domain(ne) \leftrightarrow etype(e) = Bottom$;

A Basic MFG *with predicate labels* (pbMFG) has in addition a functional relation $predlab \subset ne \times PS$, where PS is a set disjoint from all the others, interpreted as a set of predicate symbols. We noted in Section 3 how pbMFGs are used to interpret communicating systems with control branching.

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* $a \in range(\{e\} \lhd ne^+) \ \wedge \ e \notin range(ne)$. However, we shall later wish to identify processes across different cMFGs when we define cMFG composition, so we specify only that $ptype \subseteq S \cup C \cup X \times PT$ is a functional relation relating every node of the MFG to its process type, and the set of process types PT is disjoint from every other set in sight.

Simple MFGs. An MFG is *simple* (an sMFG) if the following conditions are satisfied:

- $(\forall a \in E)(\mid domain(\{a\} \lhd ne) \mid = 1)$ (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),
- $(\forall (a,b) \in sig)(ptype(a) \neq ptype(b))$ (there is no self-sending),
- $(\forall e \in T)(range(\{e\} \lhd ne^+) = range(\{e\} \lhd ne^*))$ (all elements in some component are *reachable* from the start node), and
- $(\forall x \in ST)(\mid range(field(domain(stype \rhd \{x\})) \lhd ptype) \mid \leq 2)$ (for any signal type, there is a *unique sender* and a *unique receiver* process).

Note that sMFGs may not be basic, since they may include nodes, such as condition nodes, that start or finish the MFG, but are not Top or $Bottom$ nodes. The three MFGs with conditions in Figure 3 are not bMFGs in that they do not start with start nodes or finish with finish nodes, however they are simple, and may easily be obtained from Message Sequence Charts describing the scenario which motivated the example.

Basic MFGs with or without predicates are for us the major descriptive objects. However, we have also noted the need, when interpreting Message Sequence Charts and also when analysing parallel code, for MFGs with condition nodes, which may be composed to form pbMFGs. Figure 3 showed three MFGs with conditions, in which the control branching was shown as two separate MFGs, which may be composed to form the pbMFG in Figure 4. We define MFGs with conditions below. Simplicity arises from purely practical considerations. In most examples we have seen in which it was necessary to compose MFGs, the MFGs are simple. Certainly, Message Sequence Charts and Time Sequence Diagrams yield simple MFGs. So we have defined simplicity here, and the reader may like to consider our further constructions under the assumption of simple MFG arguments, although the constructions also take non-simple arguments.

MFGs with Conditions

We define MFGs *with conditions* (cMFGs). Generally, it is only necessary to consider cMFGs that are also simple, but we do not assume simplicity in the definition. We introduce a set I of *condition nodes*, which are elements of X.

Definitions. An *MFG with conditions* (cMFG) is a labeled digraph

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

where

- $M' = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom)$ is an MFG;
- $X = T \cup B \cup I$ with T, B and I pairwise disjoint (we call elements of T *top nodes*, elements of B *bottom nodes*, and elements of I are *condition nodes*);
- $(\forall x \in T)(etype(x) = Top)$, $(\forall x \in B)(etype(x) = Bottom)$, and $(\forall x \in I)(etype(x) = \emptyset)$; (the event type of top nodes is Top and that of bottom nodes is $Bottom$; condition nodes have no event type);
- $ne \subseteq ((S \cup C \cup I \cup T) \times (S \cup C)) \cup ((S \cup C) \times (S \cup C \cup I \cup B))$ (start nodes may only be condition nodes or Top nodes, finish nodes may only be condition nodes or $Bottom$ nodes, and condition nodes may only be start or finish nodes);

- 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 labeling*;
- $(\forall l \in CL)(\mid domain(cond \rhd \{l\}) \mid = \mid range(domain(cond \rhd \{l\})) \lhd ptype \mid)$ (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 cMFG M is *conditions(M)*.

Types of Conditions.

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

A cMFG may have only initial and final conditions, by definition, but conditions may or may not be global.

Continuations. Given an MFG M, define $init_M \triangleq \{(a,b) \mid (a,b) \in ne_M \wedge ne_M \rhd \{a\} = \emptyset\}$, and $final_M \triangleq \{(a,b) \mid (a,b) \in ne_M \wedge \{b\} \lhd ne_M = \emptyset\}$. Let $\mathcal{M}$ be a set of cMFGs, $M_1, M_2 \in \mathcal{M}$, 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);
- $(\forall x \in C_1)(x \in range(final_{M_1})) \;\wedge\; (\forall x \in C_2)(x \in range(init_{M_2}))$ (C_1 is a final condition and C_2 is an initial condition).

We shall restrict ourselves to composition of cMFGs via global conditions.

Composition. The *composition* of cMFGs is the 'gluing together' of cMFGs at common conditions, i.e. where one is a continuation of the other. During this process, some condition nodes are removed. We also define the *composition graph* of a set of cMFGs.

Let $\mathcal{M}$ be a set of cMFGs, $M_1, M_2 \in \mathcal{M}$, and suppose the event sets of both cMFGs 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 cMFG $M' = (S', C', X', ne', sig', ST', stype', ET', etype', Top, Bottom, CL', cond')$, $M' \triangleq M_1 \oplus M_2$, 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 unioned),
- $X' = I_1 \cup T_1 \cup B_2 \cup I_2 - C - D$ (start nodes of M_2, which form condition D, and finish nodes of M_1, which form condition C, are eliminated);
- $ne' =$

$$(ne_1 - (ne_1 \rhd domain(C \lhd cond_1) - (ne_1 \rhd B_1))$$
$$\cup\, (ne_2 - ((domain(D \lhd cond_2)) \lhd ne_2) - (T_2 \lhd ne_2)))$$
$$\cup\, \{(a,b) \mid ptype(a) = ptype(b) \wedge a \in domain(ne_1 \rhd (domain(C \lhd cond_1)))$$
$$\wedge\, b \in range((domain(D \lhd cond_2)) \rhd ne_2)\},$$

  (the new ne relation is obtained as the union 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 with top and bottom nodes; we then add new ne edges to connect M_1 and M_2)
- $sig' = sig_1 \cup sig_2$, $ST' = ST_1 \cup ST_2$, $stype' = stype_1 \cup stype_2$,
- $ET' = (\{!, ?\} \times (ST_1 \cup ST_2)) \cup \{Top, Bottom\} - ((B_1 \lhd etype_1) \cup (T_2 \lhd etype_2))$, $etype' = etype_1 \cup etype_2$,
- $CL' = (CL_1 - range(C \lhd cond_1)) \cup (CL_2 - range(D \lhd cond_2))$, and $cond' = (cond_1 - (C \lhd cond_1)) \cup (cond_2 - (D \lhd cond_2))$.

Let $\mathcal{M}$ be a set of cMFGs. We define the *composition* relation $comp \subseteq \mathcal{M} \times \mathcal{M}$ such that $comp \triangleq \{(M_i, M_j) \mid M_i, M_j \in \mathcal{M} \;\wedge\; M_i \oplus M_j \mathit{is\ defined}\}$. From this we derive the *composition graph* $\mathcal{C} = (\mathcal{M}, comp)$ ($\mathcal{C}$ is a digraph whose nodes are individual cMFGs, and whose edges lead from a cMFG to its continuations).

Unfolding of MFG Specifications. Composition is defined between two cMFGs only. We obtain a pbMFG such as in Figure 4 from the cMFGs in Figure 3 by making all compositions possible from the cMFGs. The composition of cMFGs according to a composition graph yields a single graph, paths through which correspond to system traces. However, infinite traces could only be obtained in this manner from cMFGs (which specify a finite number of signals each) by infinite composition. We need a *finite* representation which contains the same information about traces as the 'infinite composition'. 'Infinite composition' may only happen from a set of MFGs for which there is a loop in the composition graph. If we 'fill in' the composition graph by 'plugging in' the cMFGs in the appropriate places, we obtain the desired finite structure. So, we define the *unfolding* operation on a set of MFGs which composes a cMFG with all possible successors, intuitively by taking the composition graph and 'plugging in' each actual cMFG (without its initial and terminal condition nodes) in the appropriate place. The result of this operation is a pbMFG with branching and cycles, and provides us with a single finite structure, a pbMFG, corresponding to the original set of cMFGs.

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

- $S = \bigcup_{i=1,\ldots,n} S_i$, $C = \bigcup_{i=1,\ldots,n} C_i$,
  $X = \bigcup_{i=1,\ldots,n} X_i - \{C \in Cond(\mathcal{M}) \mid (\exists M_i, M_j \in \mathcal{M})\ (Cont(M_i, M_j) \;\wedge\; C \in (final_{M_i} \cap init_{M_j}))\}$,

- $ne =$

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

  (The ne relation is obtained by a union 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 cMFGs.)
- $sig = \bigcup_{i=1,\ldots,n} sig_i$, $ST = \bigcup_{i=1,\ldots,n} ST_i$, $stype = \bigcup_{i=1,\ldots,n} stype_i$.

5. From MFGs to Global State Transition Graphs

Using unfolding, we may represent a set of cMFGs by a single pbMFG. The set of cMFGs we start with will have come from an attempt to describe the message-passing features of some system of communicating processes, and we wish to obtain a pbMFG as a description of this system. Accordingly, we call any set of MFGs whose unfolding yields a pbMFG, an *MFG specification*. Use of the word 'specification' should not be taken to suggest that we are advocating sets of cMFGs as a specification method. Sets of Message Sequence Charts and analyses of parallel code yield sets of cMFGs, so MFG specifications are the starting point for our semantic interpretation. We have shown already how to use unfolding to yield a single pbMFG. In order to obtain a finite-state automaton from such a pbMFG, we have to define the global states, the start state, and the state transition function, which we do in this section. This triple defines the global state transition graph (GSTG), and is uniquely determined by the initial set of cMFGs. To make an automaton from the GSTG, we need further to define the set of final states, which will depend on a later discussion of liveness properties. We require that there must be a finite number of global states.

Obtaining the Global States, the Start State, and the Transition Relation. The *global states* are certain sets of edges of the MFG, and the transition relation between states is obtained by deleting particular edges from the state and adding others. The *start state* q_0 is simply the set of edges leading from Top

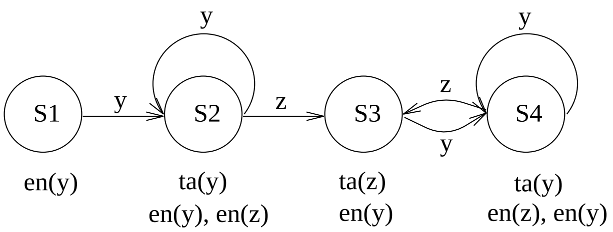

Fig. 5. Global State Transition Graph for MFG I

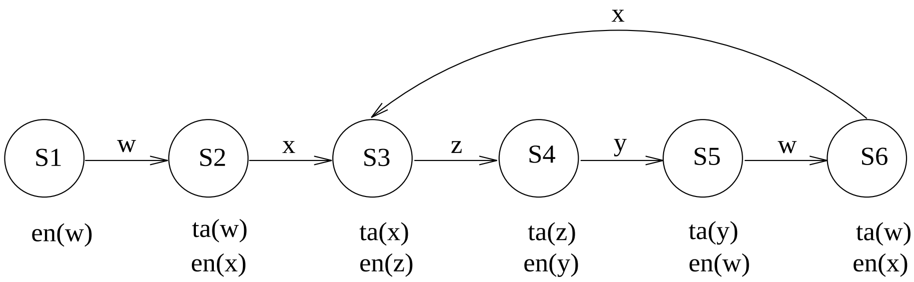

Fig. 6. Global State Transition Graph for MFG II

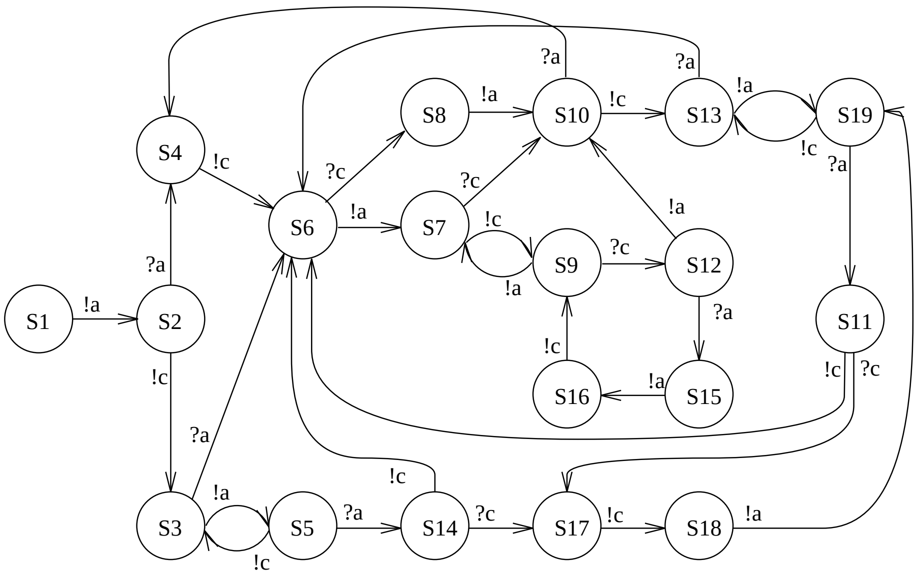

Fig. 7. Global State Transition Graph for MFG III

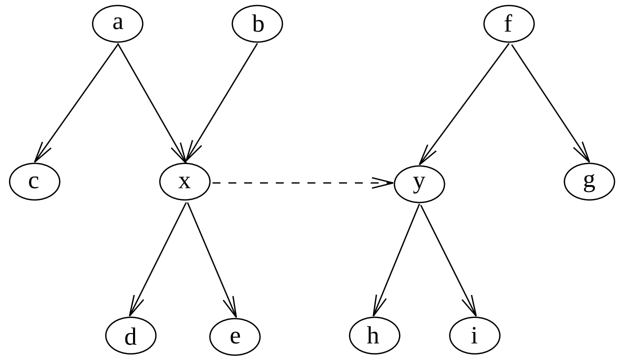

Fig. 8. Part of an MFG with asynchronous communication

nodes in the graph. We shall walk through the derivation of the GSTG for MFG II (Figure 2) given in Figure 6, to illustrate states and the *transition relation* between states. The labels we use in Figure 6 are those given to the nodes in Figure 2, so we can illustrate the `ta` and `en` predicates (one would normally just label transitions with the message types, as in Figure 7). The start state is $q_0 = \{(u,w),(v,x)\}$. The *ne* edges occurring in a state may be thought of as the set of positions where control lies in each process (the 'program counter'), and *sig* edges occurring in the state may be thought of as signals sent but not yet received. In state q_0 (labeled $S1$ in Figure 6) the event of type $!a$ at node w is *enabled*, because node w represents a `send` node (a `send` node p is enabled in a state S if there is an *ne* edge with p as second coordinate in S). Node x is not enabled, because the `send` corresponding to it has not been taken in $S1$. Since w is enabled, the event corresponding to it may be *taken*, i.e. executed, next to give a new state $S2$. The triple $\langle S1, w, S2\rangle$ will thus be a member of the *transition relation*. (Conversely, if $\langle S1, w, S2\rangle$ is in the transition relation, then w is enabled in state $S1$.) The new state $S2$ is obtained by omitting the edge (u,w), and adding the edge (w,y) to the state (to represent the change in location of the 'program counter' of the first process), and adding the *sig* edge $\langle w,x\rangle$ to represent the a signal sent but not received. Thus $S2 = \{(v,x),(w,y),\langle w,x\rangle\}$. In $S2$, node x is enabled, since it is a `receive` node and requires not only that its 'program counter' be at the right position (i.e. an *ne* edge with x as second coordinate is in the state), but that the signal has been sent (i.e. a *sig* edge with x as second coordinate is also in the state). When the action corresponding to node x is *taken*, the edges $\langle w,x\rangle$ and (v,x) are removed from the state $S2$, and (x,z) is added to represent advance of the program counter. The resulting state is $S3 = \{(w,y),(x,z)\}$. $\langle S2, x, S3\rangle$ is in the transition relation. Node z is enabled in $S3$, and so on. The GSTG in Figure 6 is annotated with the list of actions enabled (*en()*) and taken(*ta()*) in each state. Figure 5 shows the GSTG for MFG I.

Enabling and State Transitions for Branching MFGs. In the previous example we showed how the system transits from one global system state to a successor state in case of a non-branching MFG. To illustrate the concept of *enabling* for branching MFGs, we walk through the partial MFG in Figure 8. Assume that the graph is a part of a larger pbMFG, and that c and g are `send` events. All arrows in the chart belong to the *ne* relation except for the pair $< x, y >$ which belongs to the *sig* relation. As before, a global system state (gss) is a set of edges of the MFG. Consider a global system state $G = \{(a,c),(a,x),(f,y),(f,g)\}$. `Send` events c, x and g are enabled as the necessary (and sufficient) condition for enabling of send events, namely that at least one of their incoming ne-edges is in the current state, is satisfied. We will focus on the occurrence of event x. As it executes event x the system will advance the 'program counter' by omitting the edge (a,x) from G and adding all outgoing *ne* edges of node x, in this case (x,d) and (x,e), to the successor state G'. Thereby both potential successor events d and e are potentially enabled, depending only on whether they are `send` or `receive` events. However, one has to do a bit more, removing also the possibility of choosing the enabled actions represented by c and g, which represent choice alternatives to the occurrence of the x event. Hence the edges (a,c) and (f,g) are removed from G as we transit via x to G'. Finally, one has to represent that the sending event x leaves a message in transit, and thus the *sig* edge $< x, y >$ is added to G to form the successor state G'. Hence $G' = \{(x,d),(x,e),< x,y >,(f,y),(f,g)\}$. We define the transition relation formally in Section 6.

5.1. Definition of Transition Relation With Non-Local Conditions.

The definition of transition must be modified in the following way to deal with labelled conditional branches in a pbMFG. Consider a transition from state S to state S', in which the transition occurs *through* node u in process (ne-component) `A`. The intuitive meaning is that the transition happens because an event denoted by node u occurs. We can formally define a transition to occur *through node u* just in case the only out-edges added in the transition from S to S' are out-edges to u. Note that *ne* in-edges to u must also be removed in a transition through u, and maybe also other in-edges from a predecessor of u.

Suppose u has an incoming ne edge labeled `P` that is removed from S. The transition through u indicates that the branch corresponding to predicate `P` has been taken, and represents a potentially non-local control choice. Suppose there are other *ne* edges labeled `P` in S. These edges are ne-edges to other processes. The branch on `P` has been taken on transition through u, and this information must be conveyed to other processes which have a control choice to make which includes `P`.

Consider such an edge $\langle x, y \rangle$ labeled `P`, belonging to state S but in a different ne-component (thus in a different process). One way the information about the transition through u might be conveyed is by retaining edge $\langle x, y \rangle$ in S', but removing other branching possibilities for this process, i.e. all other *ne* edges of the form $\langle x, * \rangle$ with labels different from `P` are removed in the transition from S to S'. This technical device retains the finite-state character, however, it is inappropriate for the following reason. Consider the case in which the node u is part of a loop in the MFG. Transitions in this loop can potentially occur unboundedly often, and although this time the transition through u, and thus on predicate `P`, has occurred, other processes may not be on the same iteration through the loop. In particular the process containing x may be on some earlier iteration, on which a different choice of control branch had been made. Thus it would be inappropriate to remove the non-`P` ne out-edges from x, on a transition through u as suggested above, without knowing which iteration each process was on. Some processes may reach a particular control branch choice later than others, because of the asynchronous communication, thus some history of control branch choices must be retained.

Retaining the history of control branch choices necessitates the use of control history variables or their equivalent. It is beyond the scope of this paper to deal with history variables in detail, however they could be used as follows to maintain the history of control branch choices. A global list of control predicates is created by consecutive choices of branching control that are made. Define the *control vocabulary* of each process to be those control predicates which it uses. Associated with each process `B` is a sublist of the global list, consisting of some tail of the global list restricted to the control vocabulary of `B`, defined below. Call this list the `B`-*list*. The `B`-list is maintained in the following way. As a predicate `P` is added to the global list, it is added to a particular `B`-list if and only if `P` is in the control vocabulary of `B`. When control transitions through a node u of `B` which involves branching control, it must transition along that ne edge whose predicate matches that at the head of the `B`-list, and at the same time the head of the `B`-list is removed from the `B`-list.

Maintenance of the `B`-lists leads directly to a non-finite-state (in fact non-context-free) semantics. Thus such use of control predicates contravenes our desire to maintain a finite-state semantics. The example used in this section shows that the the need for history variables or an equivalent arises directly from unrestricted use of global initial and final conditions. Thus use of conditions should be restricted in order to maintain a finite-state semantics. However, we have as yet no principled general restrictions to offer for all applications. We may need to consider restrictions on an application-by-application basis.

5.2. GSTGs can be Complicated.

It should be no surprise that GSTGs can rapidly become very complicated, for example the GSTG for MFG III in Figure 2 has nineteen states (Figure 7). This is partly due to the asynchronous communication, and partly to interleavings of non-related events. MFG II and MFG III are similar, differing only in that the second message goes in opposite directions. In MFG II this forces a unique execution sequence, and the GSTG is correspondingly simple (Figure 6). However, in MFG III, the two `send`s might occur before either `receive`, or alternatively `send`s and `receive`s might be interleaved. Thus the GSTG is more complex. However, it is not our intention to recommend explicit construction of the GSTG for every MFG, for the usual state-explosion reasons advanced in [Hol91]. We use it later formally to relate liveness and safety properties as expressed in temporal logic or by Büchi automata to MFGs.

6. Formal Definition of GSTGs

This section provides the technical definition of Global State Transition Graphs, and may be skipped on a first reading. 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 ne and sig edges of the pbMFG. 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 a 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.

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

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

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 behavior alternative.

Construction of a Successor State. We now define how a system transits between different global system states in relation to a set of enabled events. 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,
- 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.

Let

$$prune(G, a) \triangleq ((G \ - \ (sig \triangleright \{a\})) \ - domain(ne \triangleright \{a\}) \triangleleft ne) \ \cup \ (\{a\} \triangleleft (ne \cup sig))$$

Formally, we define the transition from G to G' on a, where G, G' are pss's:

$$trans(G, a, G') \triangleq (a \in S \cup C) \ \wedge \ (G' = prune(G, a))$$

We define G' to be a successor of G: $succ(G, G') \triangleq (\exists a)\, trans(G, a, G')$.

The Transition Relation. We define the global 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

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

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 a set of MFGs, $N_{\mathcal{M}}$ the corresponding unfolding. Let q_0 be the set of start states, i.e. $q_0 = \{(a,b) \in ne \mid (\{a\} \lhd ne) = \emptyset\}$. We define G to be a *global system state* (gss) iff $G \in Q$, where $Q = \{q_0\} \lhd \mathcal{T}^{+}_{\mathcal{M}}$ 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 $\mathcal{G}STG_M \triangleq (Q, q_0, T_{\mathcal{M}})$.

7. From GSTGs to Automata via Liveness Properties

The global state transition graph, which we defined in the previous section, is almost an automaton, lacking only a definition of end-states. We now turn to the definition of end-states.

Definition of Global State Automaton. Let $\mathcal{M}$ denote an MFG specification and $\mathcal{G}STG_M$ the corresponding global state transition graph. We can define a Büchi automaton which transits between global system states, by adding to $\mathcal{G}STG_M$ a definition of a set of *final states* F. The definition of a Büchi automaton is very similar to that of the usual finite-state automaton, except for the criterion for *acceptance* of a string. Büchi automata may accept infinite strings. A *global state automaton for* $\mathcal{G}STG_M = (Q, q_0, T_{\mathcal{M}})$ is $A_{\mathcal{M}} \triangleq (Q, q_0, T_{\mathcal{M}}, F)$, where $F \subseteq Q$ is a set of *final states*. Acceptance is Büchi-acceptance [Tho90], namely an infinite word is accepted iff the automaton 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 9 is derived from some MFG 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 $F = \{S1\}$ 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 we wish to assume for a particular MFG specification. Applications of MFGs such as Message Sequence Charts and Time Sequence Diagrams omit explicit discussion of liveness properties. We show in this and succeeding sections how this leads to ambiguity in which set of traces is specified by these methods. The explicit definition of liveness properties is thus required for these applications.

A Discussion of Two Liveness Properties

A Strong Liveness Property for Loop Processes. Consider a system whose pbMFG 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

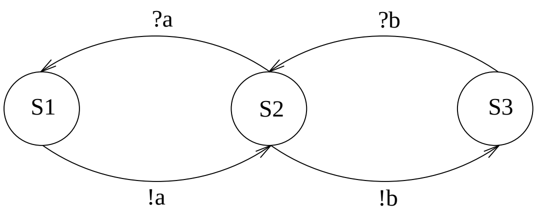

Fig. 9. Global state transition graph

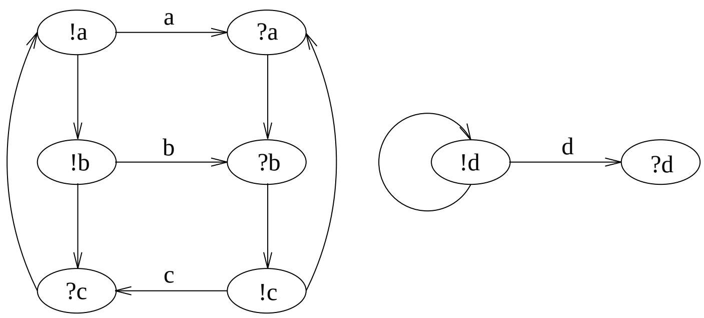

Fig. 10. Strong and weaker liveness examples

[LS91]. Let $P_i, i \leq n$ be the processes. Let $a_i \in P_i$ be some node in P_i's cycle, for $i \leq n$, chosen such that $G = \{ne\triangleright\{a_i\} \mid i \leq n\} \in Q$ (i.e. G is a *global system state* or *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 10, i.e. events preceding the cycle, and all events in the cycle, occur.

The example in Figure 11 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 MFG). 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

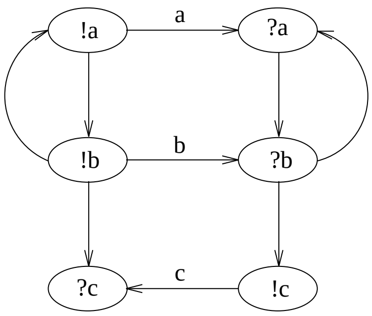

Fig. 11. Strong liveness violated by branching

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 behavior of loop processes. For example, the system of loop processes described by the right hand part of Figure 10 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 $b_i = ne \triangleright \{a_i\}$, 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 $F = \{G\}$.

8. MFGs and their connection to Temporal Logic

In the last section we noted that liveness properties have bearing on the definition of the end-state set of the automaton. A discussion of the use of Büchi automata to specify such properties of distributed systems can be found in [AS87] and [AS89b]. A complementary approach for expressing safety and liveness properties may be found in the use of temporal logic. Temporal Logic has also been advocated in the specification of open systems in [Got92b], [Got92a], and in the specification of communication protocols in [WZ92]. Temporal logic formulae are interpreted over infinite sequences of states, each state being defined by the truth values of state predicates. We relate these formulae to the automata obtained from the semantics definition. We remain informal here, referring the reader to the formal definitions in Section 9 for more precision. We base our temporal logic interpretation on the Manna-Pnueli approach [MP92].

Basic Transition Systems. Following [MP92] we interpret global state transition graphs as so-called *basic transition systems* (BTS). A BTS consists of a *finite set of states* Σ, a *transition function* τ mapping a state to a set of possible successor states, and an *initial condition.* We denote the set of all transitions τ by T. For an MFG M, Σ will be the set of states Q of $\mathcal{G}STG_M$, the transitions τ will be the communication events that lead from one global state to another, and the initial state of the BTS will be the initial state of the GSTG.

Computations and State Predicates. Manna and Pnueli define the following notions [MP92]. An infinite state sequence $\sigma = s_0, s_1, \ldots$ is a *computation* iff s_0 is the initial state of the BTS, and for all consecutive pairs $s_i, s_{i+1} \in \sigma$ there exists $\tau \in T$ such that $s_{i+1} \in \tau(s_i)$. The indices i of σ are *positions.* Transition τ is *enabled at position* i of some computation σ, written as $en(\tau)$, iff $\tau(s_i) \neq \emptyset$. Transition τ is (has been) *taken* at position $i+1$, written as $ta(\tau)$, iff $s_{i+1} \in \tau(s_i)$.

To correlate these definitions with the global state transition graph, we need to define the *enabled* and *taken* predicates. Roughly speaking, a transition is *enabled* if it is enabled in the sense used earlier in Section 5. Similarly, a transition is *taken* in a state if that transition has led to the state from an immediately preceding state (notice the 'past tense' sense of the predicate *taken*).

Temporal Logic. Given these interpretations of a GSTG as a model for temporal logic, we may define a temporal logic in the usual way, e.g. [MP90]. The language has state predicates $en(\tau)$ and $ta(\tau)$ as only basic propositions, includes the Boolean connectives (we use just $\neg$ and $\vee$ for simplicity), and the temporal operators $\Diamond$ (eventually), $\Box$ (henceforth), $\diamondminus$ (sometime in the past), $\ominus$ (previous) and S (since). The semantics are defined as usual. A temporal logic formula p is interpreted over state sequences σ, and we define the usual model-theoretic notion $(\sigma, i) \models p$, that formula p is *satisfied* in position i of sequence σ.

9. Formal Definition of the Connection to Temporal Logic

In this section, we formally define the Manna-Pnueli-style temporal logic interpretation of GSTGs. The section may be skipped on a first reading.

Let $N_{\mathcal{M}} = (S, C, X, ne, sig, ST, stype, Top, Bottom)$ denote the pbMFG unfolded from some MFG specification $\mathcal{M}$ and let $\mathcal{G}STG_M = (Q, q_0, T_{\mathcal{M}})$ denote the corresponding GSTG. First, we relate the GSTG to a *basic transition system* as defined in [MP92]. Then we consider *state predicates*, *computations* and finally the syntax and semantics of *temporal logic*.

Basic Transition System. $\mathrm{BTS}_{\mathcal{M}} \triangleq (\Pi, \Sigma, \mathcal{T}, \Theta)$ is a *basic transition system* corresponding to $\mathcal{G}STG_M$, where

- Π denotes a set of *state variables*, which is empty in the case of a set of cMFGs since they do not contain data[2],
- Σ denotes the finite set of states, so $\Sigma = Q$.
- $\mathcal{T}$ denotes a set of *transitions*, $\tau : \Sigma \to 2^{\Sigma}$, for $\tau \in \mathcal{T}$. For $s, s' \in \Sigma$ let $\tau(s) \triangleq \{s' \mid (s, \tau, s') \in T_{\mathcal{M}}\}$ (the symbol τ now has, harmlessly, both a GSTG syntax and a $\mathcal{B}TS$ syntax). For MFGs, transitions are events of type $!a$ or $?a$, where a is a signal type.
- Θ denotes an *initial* condition, in our case simply that the initial state *is* the initial state q_0 in $\mathcal{G}STG_M$.

The states of $\mathcal{B}TS_{\mathcal{M}}$ correspond to global system states of $\mathcal{G}STG_M$ (they are sets of edges of the *ne* and *sig* relations), and the transitions τ of $\mathcal{B}TS_{\mathcal{M}}$ correspond to communication events of the pbMFG obtained from the MFG specification.

State Predicates. Manna and Pnueli introduce an assertion which they call the *transition relation*[3] of the form $\rho_\tau : C_\tau(\Pi) \wedge (\bar{y}' = \bar{e})$ describing the change of the values of state variables in state s to their values in state s' into which the system transits from state s by taking transition τ. Since $\Pi = \emptyset$ for MFGs, C_τ is a constant, denoting the *enabling condition* which describes the condition under which the state s may have a successor state by taking the τ transition. $(\bar{y}' = \bar{e})$

[2] The only data in MFGs is signal type information, which is encoded in state information.
[3] We will show that this notion is simplified for MFGs, so there will be no confusion with our notion of transition relation for GSTGs.

stands for a conjunct which expresses the values of a sequence of state variables after the transition has been performed. Since there are no state variables in MFGs, this conjunct is vacuous, and so the transition relation ρ_τ is equivalent to the enabling condition C_τ (which is just that there *is* some $\tau' \in \tau(s)$) thus ρ_τ holds in a state s for some transition τ iff there exists $s' \in \Sigma$ such that $s' \in \tau(s)$.

Thus a transition τ is *enabled* in some state s iff $\tau(s) \neq \emptyset$. Conversely, τ is *disabled* in s iff $\tau(s) = \emptyset$. Mapping this to our MFG definitions we have that τ is enabled in state s iff there is some state s' such that $\langle s, \tau, s' \rangle \in T_{\mathcal{M}}$. Consequently, the Manna-Pnueli enabling condition for an action τ is true in precisely those GSTG states in which τ is *enabled* in the sense in which we defined this predicate for GSTGs earlier in the paper.

Computations and State Predicates *en* and *ta*. An infinite state sequence $\sigma = s_0, s_1, \ldots$ is a *computation* [MP92], iff

- $s_0 \models \Theta$, which means just $s_0 = q_0$, and
- for all consecutive pairs $s_i, s_{i+1} \in \sigma$ there exists $\tau \in \mathcal{T}$ such that $s_{i+1} \in \tau(s_i)$.

The indices i of σ are *positions*. Transition τ is *enabled (disabled) at position i* of some computation σ iff it is enabled (disabled) in s_i. We say that transition τ is *taken at position* $i+1$ iff $s_{i+1} \in \tau(s_i)$. We define the predicate *en*(τ) to hold in state $s_i \in \sigma$ iff τ is enabled at position i and we define the predicate *ta*(τ) to hold in state $s_i \in \sigma$ iff τ is taken at position i[4]. As noted, these definitions cohere with our former GSTG definitions. In Figures 6 and 5, we annotate each state with the instances of *en* and *ta* that are true in that state.

Temporal Logic. We define a temporal logic in the usual way, following [MP90]. The language has state predicates *en*(τ) and *ta*(τ) as basic propositions, includes the Boolean connectives (we use just $\neg$ and $\vee$ for simplicity), and the temporal operators $\Diamond$ (eventually), $\Box$ (henceforth), $\diamondminus$ (sometime in the past), $\ominus$ (previous) and $\mathcal{S}$ (since).

The semantics are defined as usual. A temporal logic formula p is interpreted over state sequences σ, and we define $(\sigma, i) \models p$, i.e. that formula p is *satisfied* in position i of sequence σ.

- If p is a basic assertion, then $(\sigma, i) \models p$ *iff* p is true in s_i as defined above.
- $(\sigma, i) \models \neg p$ *iff not* $(\sigma, i) \models p$
- $(\sigma, i) \models p \vee q$ *iff* $(\sigma, i) \models p$ *or* $(\sigma, i) \models q$
- $(\sigma, i) \models \Diamond p$ *iff* for some $j \geq i$ $(\sigma, j) \models p$
- $(\sigma, i) \models p \mathcal{S} q$ *iff* for some $k, 0 \leq k \leq i, (\sigma, k) \models q$, and for every j such that $k < j \leq i$, $(\sigma, j) \models p$
- $(\sigma, i) \models \ominus p$ *iff* $i > 0$ *and* $(\sigma, i-1) \models p$

As syntactic abbreviations we introduce the following notation.

- $\Box p \triangleq \neg \Diamond \neg p$
- $\diamondminus p \triangleq \mathit{true}\ \mathcal{S}\ p$

[4] In order to avoid notational confusion with our definitions for MFGs we distinguish our notation slightly from the notation used in [MP92].

We say that a formula p *holds* on sequence σ iff $(\sigma, 0) \models p$, that it is *satisfiable* iff it holds for *some* computation, and *valid* iff it holds for *all* computations.

10. Logical Properties of MFGs.

We can now give examples of properties expressed in temporal logic which can characterise pbMFGs obtained by unfolding sets of simple MFGs. In section, the MFG specifications considered all contain simple cMFGs only. The classification of properties as *safety, recurrence*, etc, refers to the classification in [MP90].

Properties Satisfied by all MFG Specifications. The following properties are satisfied by all computations derived from MFG specifications, as may be seen by inspection.

1. **Enabling of a `send` event (a *safety* property):** If a `send` event is taken, it must have been enabled previously. However, the enabling does not have to persist until the event is disabled, because a `send` event may also be disabled by a nondeterministic behavior alternative (some branch is taken rather than another) in the process's control flow.

   $$\Box(ta(x) \supset \diamondminus en(y))$$

   where $\langle x, y\rangle \in sig$.
2. **Persistent enabling of a `receive` event: (a *safety* property)** A `receive` event may only be taken if it has been previously enabled by a `send` event of the same type. Additionally, the enabling of a `receive` event can only be disabled by a `receive` event, therefore an enabling of a `receive` event persists up until the state when it is taken.

   $$\Box(ta(y) \supset \ominus\,(en(y)\ S\ ta(x)))$$

   where $\langle x, y\rangle \in sig$.

10.1. Some Potential Requirements on MFG Specifications.

Some liveness properties are not automatically fulfilled by an MFG specification M. It was noted earlier that some of these properties were definable by making different selections of the set of final states of a Büchi automaton defined on $GSTG_M$. If it is required for an application that these properties should hold, they must explicitly be stated as an annotation to a pbMFG or an MFG specification. Well-known examples of such properties are

1. **Weak fairness (a *recurrence* property):** it is not the case that any transition τ is enabled continuously without ever being taken.

   $$\Box\Diamond(\neg en(\tau) \vee ta(\tau))$$

2. **Strong fairness (a *reactivity* property):** if an arbitrary transition τ is enabled infinitely many times, then it is taken infinitely many times.

   $$\Box\Diamond en(\tau) \supset \Box\Diamond ta(\tau)$$

It is known (and should be clear) that strong fairness implies weak fairness. We note that since `receive` events are persistently enabled, strong fairness and

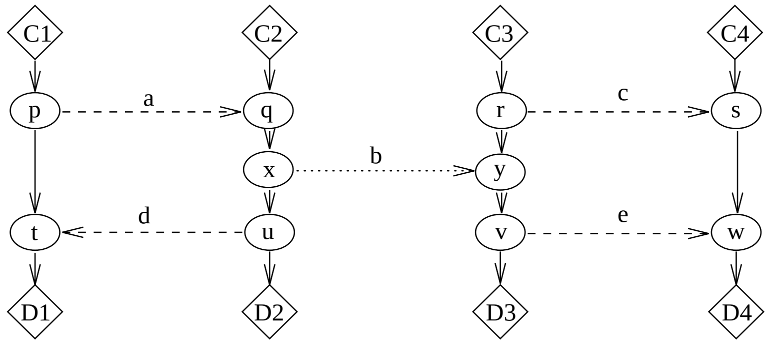

Fig. 12. MFG with synchronous communication

weak fairness just for `receive` events are equivalent statements. However, since a `send` event may be disabled without being taken, strong fairness and weak fairness are not equivalent for `send` events.

11. Representing Synchronous Communication in MFGs

So far, we have considered *sig* edges to represent asynchronous communication between processes. It is relatively straightforward to include synchronous communication sig edges within our definitions, and to provide a definition of state transition within the GSTG for synchronous communication events also. We show the modifications needed to handle synchronous *sig* edges in the following sections.

11.1. Example

The example of Figure 12 shows an MFG specification which includes synchronous communication (the dotted-line arrow), as well as asynchronous communication (dashed-line arrows). Different definitions of the transition relation must be given for synchronous and asynchronous messages, since synchronous symbols represent a single atomic action in which both processes participate, rather than the two separate, maybe temporally distinct, actions of asynchronous message-passing. We introduce two disjoint sig relations: an *asynchronous sig* relation, e.g. the set of pairs $\{< p, q >, < r, s >, < u, t >, < v, w >\}$ in the example above; and a *synchronous sig* relation, in the example just the single pair $\{[x, y]\}$. Members of the asynchronous *sig* relation are *asyncsigs*, and members of the synchronous relation *syncsigs*. We write asyncsigs between angle brackets $< \ldots >$, and syncsigs between square brackets $[\ldots]$.

Sending and reception of a synchronous message are syntactically distinct, even though sending/reception is a single *atomic action* i.e. an action which is indivisible. In particular, this means that if $[x, y]$ is a syncsig with label a, there is no temporal ordering on the $!a$ represented by x and the $?a$ represented by y. Hence, in a trace, these occur as one event. There are two ways this can be technically denoted in traces. One is by writing the $!a, ?a$ adjacent in every trace (in this order, since a syncsig retains a direction from sending to reception). This correctly represents the atomicity of this event, since the sending and reception

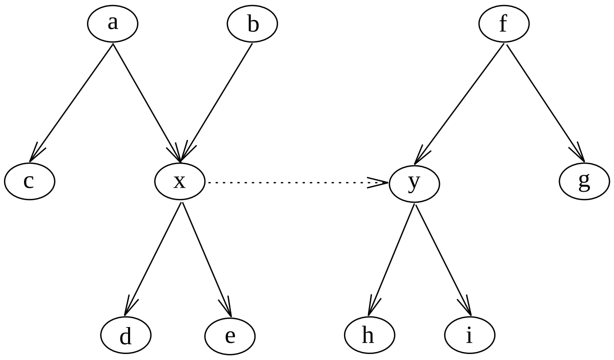

Fig. 13. Part of an MFG with synchronous communication

are not interrupted by any other event in the system in any trace. The second is that, since sending and reception are merely syntactically distinct parts of a single event, the event should be represented in the trace by a single notation, say $[!a, ?a]$. Implementations of the specified processes have distinct code locations corresponding to sending and receiving primitives, which argues for the the first choice. However, it is plausible that a message of type a may be sent over an asyncsig at one point, and another also of type a over a syncsig at another. Under the first choice one would not be able to tell from the trace notation alone whether a particular occurrence of a $!a, ?a$ is a sync send/receive, or an async send followed immediately (for whatever reason) by an async receive. For this reason, we prefer the notation $[!a, ?a]$.

To show how the atomicity of synchronous communication restricts the sets of possible interleavings, consider the example of Figure 12. If message b was asynchronous, a sequence of events $etype(x), s, etype(y)$ could be an admissible part of a system trace, which is not the case if message b is synchronous.

Since synchronous sending and reception are syntactically distinct but represent an atomic action, a synchronous communication event of sending and reception corresponds to exactly one transition in the global state transition graph (rather than the two distinct transitions required for asyncsig edges).

Enabling. To illustrate the concept of *enabling* for syncsig, we walk through the partial MFG in Figure 13. Assume that the graph is a part of a larger MFG, and that c and g are asynchronous `send` events. All arrows in the chart belong to the ne-relation except for the pair $[x, y]$ which belongs to the syncsig relation. As before, a global system state (gss) is a set of edges of the MFG. Consider a global system state $G = \{(a, c), (a, x), (f, y), (f, g)\}$. Events c, x, y and g are enabled. We focus on the synchronous events x and y. As in the asynchronous case, a necessary condition for x to be enabled is that at least one of the in-edges of node x is in G. This condition is satisfied in G, but in contrast to asynchronous sends this is *not* a sufficient condition in the synchronous case for x to be enabled. A synchronous send event may not occur independently of its corresponding receive event, so for either x or its corresponding receive event y to be enabled, we require also that for y there is at least one incoming edge which is in the gss G. Hence for every syncsig $[x, y]$, the send node x (respectively the receive node y) is *enabled* in G if at least one ne edge with x as target (second coordinate) is in G and at least one *ne* edge with y as target is in G.

State Transitions. We now informally explain the state transition associated with a syncsig edge. Occurence of the event associated with $[x,y]$ corresponds to the transition from global system state G to a successor state G'. The transition corresponds to completion of the atomic synchronous communication action with a signal both generated and received. This signal can never be 'in transit' in any distinct system state, so it never appears as part of a system state. Both nodes x and y are enabled in G, and the transition occurs as both processes advance their program counters to the next event, represented by omitting ne edges of the form $(*,x)$ and $(*,y)$ and adding edges of the form $(x,*),(y,*)$ to the ne relation.

As in the asynchronous case, however, one has to do a bit more, removing also the possibility of choosing the enabled actions represented by c and g, which represent choice alternatives to the occurrence of the $[x,y]$ event. Hence the edges (a,c) and (f,g) are removed from G as we transit via $[x,y]$ to G'. Hence $G' = \{(x,d),(x,e),(y,h),(y,i)\}$. We define the transition relation formally below.

11.2. Formalisation of Extended Message Flow Graphs

Our formal definitions are based on previous definitions All parts of the definitions which are not explicitly redefined here remain valid. We refer to MFGs with both synchronous and asynchronous communication as *extended MFGs* or *XMFGs*.

Extended MFGs. To handle both asyncsigs and syncsigs, we simply split each set of send nodes (receive nodes) into two disjoint sets of syncsend (syncreceive) and asyncsend (asyncreceive) nodes, and the *sig* relation into two disjoint syncsig and asynsig relations.

Let S_a, S_s, C_a, C_s and X denote arbitrary pairwise disjoint sets, the elements of which we call *asyncsend* nodes, *syncsend* nodes, *asyncrec* nodes, *syncrec* nodes, and *extra* nodes. Let ST and ET denote arbitrary disjoint sets whose elements we call *signal* and *event* types. For compatibility with the MFG definition we define $S \triangleq S_a \cup S_s$, and $C \triangleq C_a \cup C_s$. An *extended* MFG is a tuple

$$\mathcal{G}^X = (S_a, S_s, C_a, C_s, X, ne, sig_a, sig_s, ST, stype, ET, etype, Top, Bottom)$$

where $(S \cup C \cup X, ne, etype, ET)$ is a digraph with node labels, and $(S_a \cup C_a, sig_a, stype, ST)$ and $(S_s \cup C_s, sig_s, stype, ST)$ are digraphs with edge labels satisfying the following conditions:

1. $sig_a \subseteq S_a \times C_a$
2. $sig_s \subseteq S_s \times C_s$
3. $sig \triangleq sig_a \cup sig_s$,
4. all other conditions as for MFGs are satisfied.

This splitting of S, C and sig are the only changes required in the MFG (the 'abstract syntax'). Note that sig_a and sig_c are both bipartite relations.

11.3. Semantics of Extended MFGs

We now make the required additions to the formal definitions of *potential system state*, of *enabling* a set of events in a global system state, and finally of the *global state transition graph*, to accommodate XMFGs.

Enabling. Given an extended MFG

$$\mathcal{G}^X = (S_a, S_s, C_a, C_s, X, ne, sig_a, sig_s, ST, stype, ET, etype, Top, Bottom)$$

we define a *potential system state* (pss) $G \subseteq ne \cup sig$ to be any subset of $ne \cup sig_a$ (synchronous communication never leaves a message in transit, therefore there will never be a system state which includes an edge from the sig_s relation).

Let $V \subseteq S \cup C$, and let G denote a potential system state. V is *enabled in* G iff

- if $x \in V, [x, y] \in sig_s$ then $y \in V$, and if $y \in V, [x, y] \in sig_s$ then $x \in V$,[5]
- for every event in V one incoming ne edge is in G,
- for every receive event in V the corresponding sig_a or sig_s edge is in G.

The condition that, for every synchronous send event in V there is at least one incoming ne edge in G of the corresponding synchronous receive event and *vice versa*, is automatically ensured by the conjunction of the first two conditions above. Let $SS_V = domain(V \lhd sig_s)$ be the set of synchronous send events in V; similarly $SR_V = range(sig_s \rhd V)$ the set of synchronous receives in V; $AS_V = domain(V \lhd sig_s)$ the set of asynchronous sends in V; and $AR_V = range(sig_s \rhd V)$ the set of asynchronous receives in V. We formally define the extended *enabling* predicate as

$$\begin{aligned} &enabled^X(V, G) \triangleq \\ &\quad SR_V = range(SS_V \lhd sig_s) \ \wedge \ SS_V = domain(sig_s \rhd SR_V) \\ &\quad \wedge \ range((ne \rhd V) \cap G) = V \\ &\quad \wedge \ ((sig_s \cup sig_a) \rhd V) \subseteq G \end{aligned}$$

and the extended set of enabled events as

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

Construction of a Successor State. Assume that a system is in an actual state G. To obtain a successor state G', all conditions as before in the definition of *succ* in Section 6 must hold; furthermore, if the triggered event is synchronous, its partner event must also go through the transition process. We recall the definition of $prune(G, a)$ from Section 6. Recall also that we can think of a transition on a synchronous event formally as two adjacent transitions consisting of the send followed immediately by the receive. We may use *prune* to generate the two adjacent formal transitions on a synchronous event and its partner. The transition on asynchronous events is as before. Thus, the following operations need to be performed:

[5] As we explained informally above, a synchronous node can only be enabled if both elements of the syncsig are enabled, hence the condition on V.

- Select the event $a \in enableset^X(G)$ which is to be executed next;
- prune G by a, i.e. perform $prune(G, a)$;
- if a is a synchronous send event and b is the corresponding synchronous receive event, $prune(G, a)$ and if G_1 is the result, follow with $prune(G_1, b)$;
- if a is a synchronous receive event and b is the corresponding synchronous send event, $prune(G, b)$ and if G_2 is the result, follow with $prune(G_2, a)$.

Formally, we define the transition relation from G to G' via a:

$$
\begin{aligned}
& trans^X(G, a, G') \triangleq \\
& \quad (a \in (AS_V \cup AR_V) \wedge trans(G, a, G')) \\
& \quad \vee \ (a \in SS_V \ \wedge \ G' = prune(prune(G, a), range(\{a\} \lhd sig_s))) \\
& \quad \vee \ (a \in SR_V \ \wedge \ G' = prune(prune(G, domain(sig_s \rhd \{a\})), a)).
\end{aligned}
$$

We may define the extended successor relation $succ^X(G, G') \triangleq (\exists a \in S \cup C)\, trans^X(G, a, G')$.

The Transition Relation. We define the global transition relation on a pss G, an event a such that $\{a\} \in enableset^X(G)$ (a is enabled in G), and a successor state G' such that $succ^X(G, G')$. Let $N^X_{\mathcal{M}}$ denote an extended unfolding. The *global state transition relation* is $\mathcal{T}^X_{\mathcal{M}} \subseteq (ne \cup sig_a) \times (S \cup C_a \cup X) \times (ne \cup sig_a)$ such that

$$\mathcal{T}^X_{\mathcal{M}} \triangleq \{(G, a, G') \mid enabled^X(G, \{a\}, G') \ \wedge \ trans^X(G, a, G')\}.$$

The notions of global system state and transition graph are as defined for MFGs.

11.4. Postscript

We have treated a synchronous communication event in an MFG by formalising the transition as a pair of adjacent send-receive events, without an intervening state. Since synchronous communication is an atomic action represented by two distinct nodes in an ne-sig graph, why did we not represent such an action by a single node in the MFG? The answer is that pursuing this alternative would have destroyed some nice properties of the MFG definition, namely (a) processes can be identified as connected components of the *ne* relation; and (b) the information about the direction of the communication (from which process to which other) is lost. In particular, the direction of a synchronous communication represents the fact that sending and receiving of messages refer to distinct code-locations in the processes involved, information which may be essential in conformance testing [ISO91a] and debugging.

11.5. Liveness Properties

We discussed earlier some liveness properties that MFGs with asynchronous communication might satisfy. Guaranteeing liveness properties is considerably simplified with the introduction of synchronous communication edges. A syncsig edge enforces synchronisation at its head and tail. Thus, all events preceding any event on a syncsig edge must already have occurred before the synchronous event. For example, suppose the *sig* edge in MFG I (Figure 2) is a syncsig. The enforced synchronisation of the two processes avoids the phenomenon of repeated sending without reception, noted before with respect to asynchronous communication. The unique trace of this MFG is simply $[!a, ?a], [!a, ?a], [!a, ?a], [!a, ?a], \ldots$. Similarly, supposing that the c communication in MFG III (Figure 2) is synchronous, the unique trace is $!a, ?a, [!c, ?c], !a, ?a, [!c, ?c], \ldots$, and the corresponding automaton has five states (cf. the nineteen states of Figure 7).

It is easy to see that if all processes in an ne-sig graph with loops must pass through at least one synchronous communication, that there is only one possible definition of the Büchi automaton corresponding to the GSTG, i.e. the GSTG determines the automaton uniquely. Other similar properties may be formulated by the reader.

12. Abstraction of Automata

In this section, we show how to simulate an arbitrary Büchi automaton by an MFG specification. The simulation relies on the notion of *abstraction* of a Büchi automaton.

Informally, an *abstraction* of a Büchi automaton $\mathcal{A}$ is an automaton $\mathcal{A}'$ whose states are a subset of the states of $\mathcal{A}$, and whose transitions occur not on letters from the alphabet of $\mathcal{A}$, but on sequences of such letters (i.e. words). In other words, the abstraction retains information only on some states, and how one gets from these particular states to others by treating a sequence of transitions of $\mathcal{A}$ as a single transition.

An Example. The GSTG for the MFG specification in Figure 14 is shown in Figure 15. We can form an abstraction of the GSTG, by retaining information only about the global states denoted by conditions C1, C2, and C3, which correspond to the state sets $\{S0, S6\}$, $\{S2\}$, and $\{S4\}$. The abstraction is shown in Figure 16 (in this case, it is in fact the composition graph of the MFGs).

The point of abstractions is that they are in an intuitive sense a summary, a less complex version, of the automaton that they abstract. We show below that an arbitrary Büchi automaton is an abstraction of the global-state automaton derived from some MFG specification, and therefore that MFG specifications are in this sense equally as complex as Büchi automata (and therefore much more expressive than temporal logic [Wol83]) under our semantics.

Abstractions Formally. $\langle \mathcal{A}', I, h \rangle$ is an *abstraction* of $\mathcal{A}$ iff

- I is a mapping of the state set of $\mathcal{A}'$ into the set of state sets of $\mathcal{A}$, i.e. I picks out a subset of the states of $\mathcal{A}$ which correspond with a particular state of $\mathcal{A}'$;
- h maps the alphabet of $\mathcal{A}'$ into words in the alphabet of $\mathcal{A}$, i.e. transitions among states in $\mathcal{A}'$ are translated into sequences of transitions of $\mathcal{A}$;
- there is a transition from s to s' on p in $\mathcal{A}'$ iff $h(p)$ is a path from some state in $I(s)$ to some state in $I(s')$ in $\mathcal{A}$,

It is well-known that such an h can be extended into a homomorphism of the words of $\mathcal{A}'$ into the words of $\mathcal{A}$. We shall abuse terminology and refer to $\mathcal{A}'$

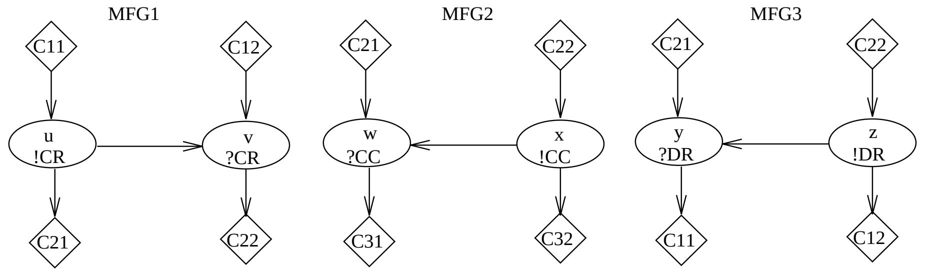

Fig. 14. An MFG Specification

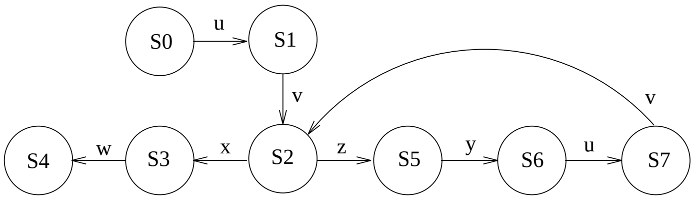

Fig. 15. Global State Transition Graph

alone as an abstraction of $\mathcal{A}$. A particularly important kind of abstraction is the *one-to-one abstraction*, in which every $I(s)$ is required to be a singleton, i.e. there is only one state of $\mathcal{A}$ corresponding to each state in $\mathcal{A}'$. The abstraction in Figure 16 is not a one-to-one abstraction. We categorise the expressiveness of MFGs by the following theorem.

Theorem 12.1. Every Büchi automaton is a one-to-one abstraction of an automaton derived from an MFG specification involving just two processes.

Proof. [Sketch]: Consider the MFG expressed in MFG IV (Figure 17). It is easy to see that there is only one possible sequence of events, as shown in the GSTG. The following lemma may easily be proved by recursion, and by inspection of the transition between states S1 and S5 in the GSTG.

Lemma 12.2. Let $\mathcal{B}$ be a Büchi automaton based on the GSTG of MFG IV, which additionally satisfies weak liveness. If $\mathcal{B}$ attains the state represented by the beginning condition $C.s$, it will later attain the state represented by the terminal condition $C.s'$.

To each pair of states s and s' of an arbitrary Büchi automaton $\mathcal{A}'$, and transition T between them, we associate a copy of MFG V, with conditions $C.s$

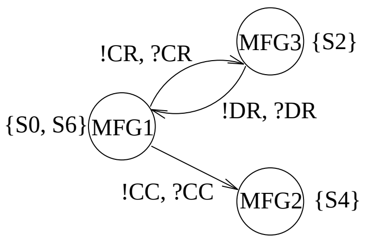

Fig. 16. An Abstraction Graph

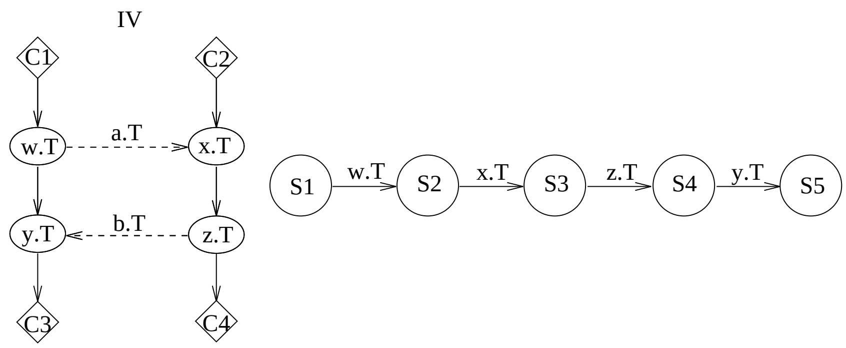

Fig. 17. MFG IV and its GSTG

and $C.s'$, and signals $a.T$ and $b.T$. This defines an MFG specification $\mathcal{M}$, with GSTG $\mathcal{G}STG_M$. We define $I(s) = C.s$ and $h(T) = !a.T?a.T!b.T?b.T$, and finally the global states of the automaton $\mathcal{A}$ derived from $\mathcal{G}STG_M$ are $\{C.s \mid s \textit{ is a final state of } \mathcal{A}'\}$. Using the lemma, it follows that $\mathcal{A}'$ is an abstraction of $\mathcal{A}$. $\square$

A crucial step in this proof is the definition of the end-states of the automaton $\mathcal{A}$ derived from the MFG specification. We are able to do this as we please, because the end-state set is not determined by a pure MFG specification. The reader should note that some additional liveness requirement for MFG specifications could restrict the choice of end-state set for the automaton $\mathcal{A}$, and preclude us from carrying out the simulation of an arbitrary Büchi automaton.

13. Concluding Remarks

We defined Message Flow Graphs, as a construct occurring in the analysis of parallel code, Message Sequence Charts and Time Sequence Diagrams. We argued for, and provided, a finite-state semantics for MFGs. We defined cMFGs, and showed how to obtain pbMFGs from sets of cMFGs by unfolding. We defined a collection of global system states, and transitions between them, from the pbMFG. We discussed the completion of this GSTG to a Büchi automaton, noting the reliance on liveness properties not explicit in the MFG, leading to a connection to temporal logic via the standard semantics. We also showed how to incorporate synchronous communication with asynchronous communication in MFGs, and noted that it greatly simplifies the liveness analysis of MFGs to include synchronous communication signals. We showed how to simulate an arbitrary Büchi automaton by an MFG. Since we have also shown that an arbitrary MFG may be interpreted by a Büchi automaton, we conclude that in this sense MFGs and Büchi automata have equivalent expressive power.

Acknowledgements

We thank Ekkart Rudolph, Ken Turner, Philippe Oechslin and the two referees for their helpful commentary on previous versions of this paper. We are very much obliged to the second referee in particular, who read the paper most thoroughly, for his detailed and insightful commentary. The first author thanks Barbara Simons for joint work which led to the formulation of the concept of Message Flow Graph, and its use in analysing parallel code. The first author was partially supported by IBM Almaden Research Center, and the CEC RACE II R2088 project TOPIC at the University of Stirling. The second author was partly supported by the Swiss National Science Foundation. Work on Message Sequence Charts during which some of the semantic notions were developed was performed by both authors under contract 233 of the Swiss PTT to the University of Berne.

References

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

[AS89a] S. Aggrawal and K. Sabnani, editors. *Protocol Specification, Testing and Verification, VIII*. Proceedings of the IFIP WG 6.1 Eighth International Symposium on Protocol Specification, Testing and Verification. North Holland, 1989.

[AS89b] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages*, 11(1):147–167, 1989.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. Research Report 581, IRISA, Rennes, France, 1991.

[BG88] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. Rapport de Recherche 842, Institut National de Recherche en Informatique et en Automatique (INRIA), 1988.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[BL90] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[BLJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, Sep 1991.

[BM95] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining message sequence charts. In *[HL95]*. 1995. To appear.

[CC91] A. A. R. Cockburn and W. Citrin. An executable specification language for history-sensitive systems. Technical Report IBM RZ 2162, IBM Rüschlikon Research Laboratory, Zürich, 1991.

[CCHK90] A. A. R. Cockburn, W. Citrin, R. F. Hauser, and J. Känel. An environment for interactive design of communication architectures. In *[LPU91]*, 1990.

[CCI92] CCITT. Recommendation Z.120: Message Sequence Chart (MSC). CCITT, Geneva, 1992.

[CD93] D. Cohen and N. Dorn. An experiment in analysing switch recovery procedures. In *[DG93]*, pages 23–34, 1993.

[CK91] E. M. Clarke and R. P. Kurshan, editors. *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[Cou88] J.-P. Courtiat. ESTELLE*: a powerful dialect of ESTELLE for OSI protocol description. In *[AS89a]*, 1988.

[Cou89] J.-P. Courtiat. Estelle and Petri nets: introducing a rendezvous mechnism in Estelle: Estelle*. In M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azéma, and V. Chari, editors, *The Formal Description Technique Estelle*, pages 175–203. North-Holland, 1989.

[DG93] M. Diaz and R. Groz, editors. *Formal Description Techniques, V*. IFIP Transactions C-10, Proceedings of the Fifth International Conference on Formal Description Techniques. North-Holland, 1993.

[Got92a] R. Gotzhein. Formal definition and representation of interaction points. *Computer Networks and ISDN Systems*, 25(1):3–22, Aug 1992.

[Got92b] R. Gotzhein. Temporal logic and applications - a tutorial. *Computer Networks and ISDN Systems*, 24(3):203–218, May 1992.

[GW92] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *[CK91]*, pages 332–341, 1992.

[HL95] D. Hogrefe and S. Leue, editors. *Formal Description Techniques, VII.* Proceedings of the Seventh International Conference on Formal Description Techniques. Chapman & Hall, 1995. To appear.

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

[Hog89] D. Hogrefe. SDL and OSI: On the use of CCITT-SDL in the context of OSI. Habilitation Thesis, University of Hamburg, 1989.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[Inm84] Inmos Ltd. *The Occam Programming Manual.* Prentice-Hall International, 1984.

[ISO84] ISO. Information Processing Systems - Open Systems Interconnection - Basic Reference Model. International Standard 7498, International Standards Organisation, 1984.

[ISO87] ISO. Estelle: A formal description technique based on an extended state transition model. Draft International Standard 9074, International Standards Organisation/IFIP, 1987.

[ISO88] ISO. Information Processing Systems - Open Systems Interconnection - LOTOS : A formal description technique based on the temporal ordering of observational behavior. International Standard 8807, International Standards Organisation/IEC, 1988.

[ISO91a] ISO. Information Processing Systems - Open Systems Interconnection - Conformance Testing Methodology and Framework, Part 1: General Concepts. International Standard 9464, International Standards Organisation, ISO/TC97/SC21, 1991.

[ISO91b] ISO. Revised Text of CD 10731, Information Processing Systems - Open Systems Interconnection - - Service Conventions. ISO/IEC JTC 1/SC21 N 6341, International Standards Organisation/IEC, Jan 1991.

[Kur92] R.P. Kurshan. Automata-theoretic verification of communicating processes. Unpublished lecture notes. AT&T Bell Laboratories, Aug 1992.

[Lad93] P. B. Ladkin. Using tense logic to describe digital computing systems. Technical Report TR 110, Department of Computing Science, University of Stirling, 1993.

[Lam92] L. Lamport. Applying finite-state methods to infinite-state systems. *TLA Mailing List Note*, 92-08-28, 1992.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LL93] P. B. Ladkin and S. Leue. Interpreting Message Sequence Charts. Technical Report TR 101, Department of Computing Science, University of Stirling, 1993.

[LL94] P. B. Ladkin and S. Leue. What do Message Sequence Charts mean? In *[TAU94]*, pages 301–316. 1994.

[LL95] P. B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In *[HL95]*. 1995. To appear.

[LPU91] L. Logrippo, R. L. Probert, and H. Ural, editors. *Protocol Specification, Testing and Verification, X.* Proceedings of the IFIP WG 6.1 Tenth International Symposium on Protocol Specification, Testing and Verification. North Holland, 1991.

[LS91] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. Technical Report RJ 8488, IBM Almaden Research Center, Nov 1991.

[LS92a] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 248–259. ACM Press, 1992.

[LS92b] P.B. Ladkin and B.B. Simons. Static analysis of concurrent communicating loops. Technical Report RJ 8625, IBM Almaden Research Center, Feb 1992.

[LS92c] K. G. Larsen and A. Skou, editors. *Computer Aided Verification: Proceedings of CAV'91*, volume 575 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[LS94a] P.B. Ladkin and B.B. Simons. *Static Analysis of Interprocess Communication.* Lecture Notes in Computer Science. Springer-Verlag, 1994. To appear.

[LS94b] P.B. Ladkin and B.B. Simons. Static deadlock analysis for CSP-type communications. In D. Fussell, editor, *Responsive Computer Systems: Toward Integration of Fault-Tolerance and Real Time.* Kluwer, 1994. To appear.

[Maz87] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri-Nets, Applications and Relationship to other Models of Concurrency, LNCS Vol. 255*, volume LNCS 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer Verlag, 1987.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[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.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[MR94] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[RvH93] J. M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, Jan 1993.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley & Sons, Inc., 1994.

[Sie88] Siemens AG. *EWSD Softwareentwicklungshandbuch (Software Development Handbook), Kapitel B, Register 6, SDL Diagramme.* Siemens AG, München (Munich), 1988.

[Spi89] J. M. Spivey. *The Z Notation.* Prentice-Hall International, 1989.

[Tan89] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall International, 2nd edition, 1989.

[TAU94] R. L. Tenney, P. D. Amer, and M. Ü. Uyar, editors. *Formal Description Techniques, VI.* IFIP Transactions C, Proceedings of FORTE '93, the Sixth International Conference on Formal Description Techniques. North-Holland, 1994.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, chapter 4, pages 132–191. Elsevier Science Publishers B. V. (North-Holland), 1990.

[VFV92] M. Van Sinderen, L. Ferreira Pires, and C. A. Vissers. Protocol design and implementation using formal methods. *The Computer Journal*, 35(5):478–491, 1992.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[WZ92] H. B. Weinberg and L. D. Zuck. Timed Ethernet: Real-time formal specification of Ethernet. In W. R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 370 – 385. Springer Verlag, 1992.