Four Issues Concerning the Semantics of Message Flow Graphs

Peter B. Ladkin[a] and Stefan Leue[b*]

[a]INRIA Lorraine, 615 rue du Jardin Botanique, F-54602 Villers-Lès-Nancy, France
Peter.Ladkin@loria.fr

[b]Institute for Informatics and Applied Mathematics,
University of Berne, Länggassstr. 51, CH-3012 Berne, Switzerland
leue@iam.unibe.ch

Abstract

We discuss four issues concerning the semantics of Message Flow Graphs (MFGs). MFGs are extensively used as pictures of message-passing behavior. One type of MFG, Message Sequence Chart (MSC) is ITU Standard Z.120. We require that a system described by an MFG has global states with respect to its message-passing behavior, with transitions between these states effected by atomic message-passing actions. Under this assumption, we argue (a) that the collection of global message states defined by an MFG is finite (whether for synchronous, asynchronous, or partially-asynchronous message-passing); (b) that the unrestricted use of 'conditions' requires processes to keep control history variables of potentially unbounded size; (c) that allowing 'crossing' messages of the same type implies certain properties of the environment that are neither explicit nor desirable, and (d) that liveness properties of MFGs are more easily expressed by temporal logic formulas over the control states than by Büchi acceptance conditions over the same set of states.

Keyword Codes: F.3.2; D.2.1; 2.
Keywords: Semantics of Programming Languages; Requirements/Specifications; Protocol specification, testing and verification.

1. INTRODUCTION

This paper discusses issues arising from giving a mathematical meaning to Message Flow Graphs (MFGs). Seemingly innocuous syntactic choices may have profound semantic consequences. This highlights the danger of introducing syntactic features without thinking through their semantic consequences, and points to the need for resolution. A precise semantics for MFGs is defined in [19], and was demonstrated for Message Sequence Charts (MSCs) in [18]. MSCs are a well-used class of MFGs, standardised in ITU Z.120 ([16]), which includes some of the syntactic features whose consequences we discuss. Our interpretation is based on the set of partial message-passing states that a system described with MFGs must exhibit.

*This author's work was partly supported by the Swiss National Science Foundation and the Swiss Federal Office for Education and Scientific Research.

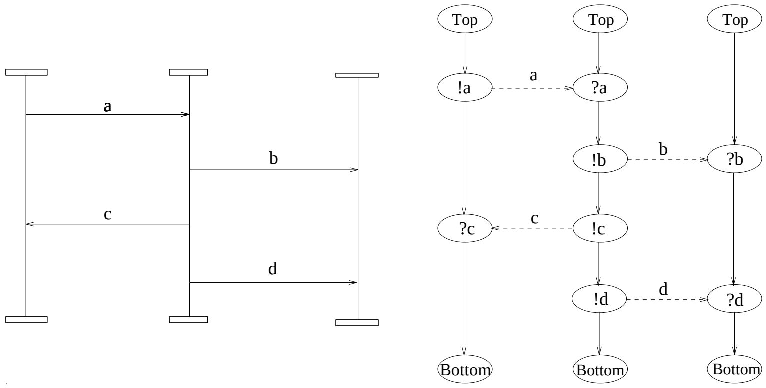

Figure 1. A simple MSC and its corresponding MFG

We present arguments here mainly by means of words rather than in mathematics. It suffices to point to the mathematical formulation of the issues.

What is an MFG? Figure 1 shows an MSC and the same information represented as an MFG. An MFG is a graph with an underlying ontology of message `send` and `receive` events, represented as nodes. Figure 2 presents an MFG which in addition to communication events contains so-called *conditions*, used in Z.120 to allow for branching and iterative behaviour.[2] In a simple MFG (one without conditions, as in Figure 1) the nodes are in one-to-one correspondence with events. In MFGs with conditions, a node may be traversed more than once during a 'run' of an MFG, and therefore does not denote an event *per se*, but rather a 'program statement' indicating that an event of the specified type occurs at this point.

It is generally accepted that the underlying ontology of MFGs (e.g. in their MSC form) is that of events (e.g. [23, 9, 13]). We can summarise features of events as follows: (a) represented events are *message* `sends` and `receives` *only*; (b) the *occurrence* of an event is represented; (c) the *order of occurrence* of events within an individual process is represented; (d) the *types* of the events are represented. These features are represented in the message-passing fragments of other imperative languages such as 'classic'-CSP [14], Esterel [4], Estelle [10] or SDL [6]. The MFG graph is a mathematically more precise formulation of the pictorial information about events and their ordering. Additionally, MFGs (but not CSP, Esterel, Estelle or SDL) connect a `send` to a *unique* `receive` in a different process.[3] A semantics of MFGs must explain how and what this connection is supposed to denote. Here are the issues:

The semantics of MFGs are inherently finite-state. There are only a finite number of control states of all processes described by an MFG, but it might nevertheless be possible, if communication is asynchronous, for there to be an unbounded number of messages 'in the system' (i.e. sent but not received). This is often used as an argument for why MSCs in particular are not finite-state.

[2]We doubt it's a good idea to include conditions without significant restrictions.

[3]MSCs used informally sometimes have `send`s going nowhere, to indicate a lost message. But this isn't in the MSC standard, and it's not significant for the argument made here.

We shall argue in Section 2 that provided that the global state assumption is satisfied (Section 2.4), and traces are interleavings, there are only finitely many global states with respect to the message-passing behavior of the system described by an MFG. Thus we may define a global finite-state automaton, whose accepted language is identical with the set of system traces described by the MFG. Such a semantics was defined in [18, 19].

Besides this argument which is particular to MFGs, justification for a finite-state requirement in general for telecommunications system specifications may be found in [15]. The primary advantage for finite-state interpretations is that reasonable verification and validation techniques may be applied. As of writing, the state-of-the-art in telecommunications system verification and validation is finite-statecraft (see for example [7, 21, 8] and [15]).

A problem with non-local choice. The MSC standard requires 'conditions'. Conditions are global labels such that two MSCs may be 'joined' at this label. We defined conditions, and composition, for MSCs and MFGs in [18, 19]. By having more than one 'joining' possibility, one obtains the effect of non-deterministic choice (conditionals defined on the values of general state predicates are not defined in the standard). Non-terminating-loop-like behavior may be obtained by writing the same condition at the beginning as well as the end of an MFG.

However, unimpeded use of conditions requires unbounded history variables containing the choices in order to resolve non-local choices (Section 3). Such a history of control branching must either be available from the environment or held locally to a process in a history variable of potentially unbounded size. Neither of these options is appealing. We believe that a good specification style should only make explicit assumptions on environment behaviour. Further, they are another potential source of non-finite-stateness.

Crossing message arrows create anomalies. The Z.120 standard allows crossing message arrows as in Figure 7 (a sort of 'message overtaking'). This also leads to implicit requirements on environment behaviour (Section 4).

Büchi acceptance conditions are insufficient to express general liveness properties. A Global State Transition Graph (GSTG) was defined from a given MFG in [18, 19]. However, the GSTG alone does not satisfactorily specify liveness properties for the MFG. We suggested using either Büchi acceptance conditions [26] or Temporal Logic [22] formulae for this purpose. However, in Section 5 we show that one cannot specify every useful liveness property by using Büchi acceptance conditions over the unique GSTG defined from an MFG. We therefore suggest the use of Temporal Logic to specify the desired liveness properties.

Some Other Proposals for MSC Semantics such as [23, 9, 13] do not define the semantics of composed MSCs, therefore cannot observe the semantic problems raised by conditions. These methods provide much less coverage than [19].

2. WHY A FINITE-STATE SEMANTICS?

Finite-state property-checking methods are largely automatic (although controlling state-explosion is a significant problem). This alone is an argument for desiring a finite-state semantics for MFGs. However, there is also an argument from the intuitive meaning of MFGs that they are inherently finite-state. We argue that the *explicit* (as contrasted with hidden) information in an MFG allows only finitely many global system control states.

2.1. What is the Event 'Connection'?

Message-passing connections are shown in the MFG in Figure 1 by means of dotted arrows, in the MSC by means of horizontal (or inclined) arrows, between different processes. What does this symbolic connection correspond to in reality?

`Send` and `receive` events of the same type are connected. Maybe it is intended that *the identical message instance that is sent by the event statement at the arrowtail is received by the event statement at the arrowhead*? However, this is too strong an identity to be useful. Channels, even Ethernet channels, may be lossy. Protocols can try to ensure that if a message of a particular type is sent but not acknowledged, then the contents of the message, along with any message-ID, is regenerated and resent, until successful reception is acknowledged. If message-identity was taken to be message-instance identity, (the actual voltage values raised on the cable), MFGs would be unable to describe higher-level services based on a reliable underlying protocol. So, rather than this strong identity condition, the connection could represent a successful reception of some uncorrupted message instance with a particular message-ID. This is a reasonable interpretation for a higher-layer interaction, such as in a service description (e.g. INRES in [3], JVTOS in [11]), or in Object Models [24, 17]. In some sense, the message-arrow represents the 'same' message sent and received, where 'same message' is a potentially finer individuation than 'message of the same type', and potentially coarser than 'identical message-instance' (while allowing either in the appropriate circumstances). Call such a creature a *message occurrence*. Thus, MSCs represent individual message occurrences, whose properties are assured if necessary by underlying protocols. Unless message occurrences are defined to be message instances in the particular application, one thus cannot discriminate message-instance behavior at the level of description of the MFG.

2.2. In Real Life, There Are a Finite Number of Message Occurrences in a Given MFG Description

The usual means of individuating message occurrences uses an identifier such as a timestamp added at message-generation time. We refer to all such IDs as timestamps. We know of no timestamping mechanism used in real protocols that allows infinite timestamps. Timestamps are often generated by a mechanism formally equivalent to picking numbers in some increasing order from the integers modulo N, for N some large integer. Since there are at most N different timestamps that may be used, only finitely many message occurrences may be individuated.

MSCs are in this regard different from SDL specifications. SDL allows explicit data variables ranging over an unbounded set of values. In principle, one may use a data variable with an infinite range as a timestamp in messages in an SDL specification, allowing distinction of unboundedly many message occurrences. Furthermore, in SDL the communication between processes is by means of explicit unbounded FIFO queues. It follows from these observations that an SDL specification may have unboundedly many states. However, there is nothing in Z.120 that would require MSCs similarly to have unboundedly many states.

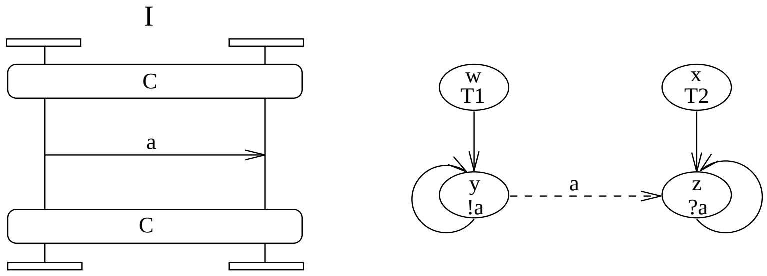

Figure 2. MSC I and corresponding MFG

2.3. Timestamps May Be Eliminated

In Figure 1, the system generates and processes four message occurrences, and each of these message occurrences has a different type. Therefore, the types may be used to individuate messages. However, in Figure 2, an unbounded number of message occurrences is specified, indicated by the condition in the MSC, and the corresponding loop in the MFG obtained by 'joining' the two condition occurrences. Timestamps may be used to individuate the occurrences of messages. Using timestamps modulo N, N iterations through the loop of MSC I in Figure 2 may be individuated. Suppose we duplicate the loop body N times (*loopbody duplication* is defined in [20]). This corresponds to a (maybe much!) larger MFG, in which there are N message arrows, and in which message-occurrence individuation corresponds to different message-occurrence arrows. For MFGs with conditions but without explicit loops, such as MSCs, loopbody duplication corresponds to MFG composition, defined in [19]. We assert without proof that this operation can be carried out for all MFGs with cycles.[4] We call this loopbody operation *timestamp-reduction*. Different message instances generated by the same arrows in the timestamp-reduced graph cannot be individuated. The timestamps individuate precisely the different message arrows in the timestamp-reduced MFG and we may thus remove them. The timestamp-reduced MFG may be much larger than the original MFG, but this increase in size is not semantically relevant. Our concern is only that *in principle* this reduction may be carried out, so that we may identify the arrows in such a reduction with the message occurrences in the MFG. We shall assume from now on that all MFGs have been timestamp-reduced in this way.

2.4. There are Global Control States.

We assume that at any point of time control in each process is located between or at specific message-passing statements or events. Each process $P_i, 0 \leq i \leq n$ contains a finite number of statements defining message-passing events $e_{i_1}, \ldots, e_{i_{n_i}}$ in P. These statements in P correspond with nodes in an MFG. Thus we require that at any point, for any P_i, the Boolean value of the state predicate

$$Last(e)_{P_i} \stackrel{\triangle}{\Leftrightarrow} \textit{the last message event that occurred in } P_i \textit{ was one corresponding to node } e$$

is well-defined, where e is one of the e_{i_k}. This entails there is a well-defined vector $\langle e_{p_1}, \ldots, e_{p_n}\rangle$ of the *next message event* for all processes. To handle the startup case, we

[4]Writing out the details from [19] would be tedious but unilluminating.

also include a predicate

$$Last(\texttt{start})_{P_i} \overset{\triangle}{\Leftrightarrow} \textit{no message event has yet occurred in } P_i$$

This is the only information available in the MFG concerning the control state of each process P_i; namely, precisely which one of the state predicates $Last(e)_{P_i}$ is true.

2.5. The Different States Engendered by a Message Occurrence

Assuming timestamp-reduction, given the ontology of events, there are three state predicates that a system may satisfy with respect to a given message occurrence m: no `send` or `receive` of a message occurrence m has occurred; a message occurrence m has been sent but not received; a message occurrence m has been sent and received. Since one instance of m may not be discriminated from another instance of m, it follows that there is a finite collection of truth values of these predicates.

2.6. The Total Number of State Predicates is Finite, and There is a Unique Global State Transition Graph

The state predicates of a given state of the MFG are the predicates $Last(e)_{P_i}$ (precisely one of which has the value *true* at any time), and, for each message occurrence m, the truth values of the three state predicates above.[5] The potential global states of the system therefore consist of consistent assignments of truth values to these state predicates. Thus there are only finitely many global states of the MFG.

Since there is a finite collection of global states, it remains to determine the state transition function in order to obtain the *global state transition graph* (GSTG). The nodes of the GSTG are the states. State transitions are represented by edges between pairs of states. State transitions are caused by events (nodes of the MFG), thus an edge of the GSTG may be labelled with the event triggering the transition. Every event causes a change in true value of precisely two predicates of the form $Last(e)_{P_i}$, and corresponding changes in the message-occurrence predicates.

Thus there is a finite number of global states in the timestamp-reduction, and these states and the transitions between them are uniquely determined and may be calculated using the method in [19].[6] We propose this as a *canonical* semantics for MFGs.

2.7. Some General Reasons for Requiring Even Asynchronous Communications to be Finite-State in Telecommunications

There is furthermore a general argument for requiring semantics of message-passing in any real telecommunications protocol or service to be finite-state, even those specified by SDL (which in principle may utilise unboundedly many timestamps).

In a protocol or service definition, each individual process control is usually a finite-state device with respect to `send`s and `receive`s. The unboundedly many states are usually attributed to the unboundedly many states a lossless asynchronous channel may have. But consider now system recovery from faults. Regardless of its size, a finite state device can only retain a bounded computation history. Suppose, as the system runs, communication channels are compromised (someone cuts a cable). Assume also that the

[5] We note that not all combinations of values are possible.

[6] We used sets of edges as a code for the collection of truth values of state predicates in [19].

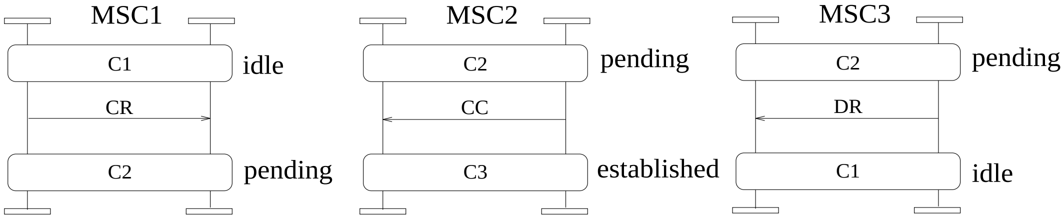

Figure 3. MSC specification with conditions

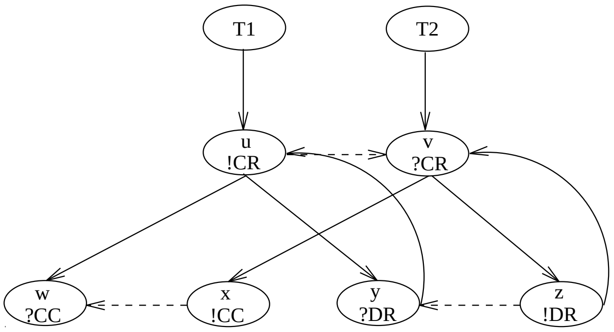

Figure 4. 'Unfolding' an MSC specification into a single MFG

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 is available. The system itself can have been operating for a very long time, much longer than the bounded memory of any single process. However, 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. Two such failures which result in the same local state of each process are therefore equivalent from the point of view of the determinable state of the system. So each such equivalence class can be identified with a *global state* of the system. 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.

It is often suggested that asynchronous communication is equivalent to the presence of lossless queues of unbounded capacity on each channel, e.g. in SDL. It is well known that in theory queues may be configured to contain the entire system history information, which is finite at any one point, but unbounded through the history of the system. By the argument above, since the number of practically distinguishable global states is bounded, the contents of these theoretical queues cannot be part of the system, in general, and properties of 'queue' contents may only be inferred from the processes that generated and received those contents – and that information is bounded.

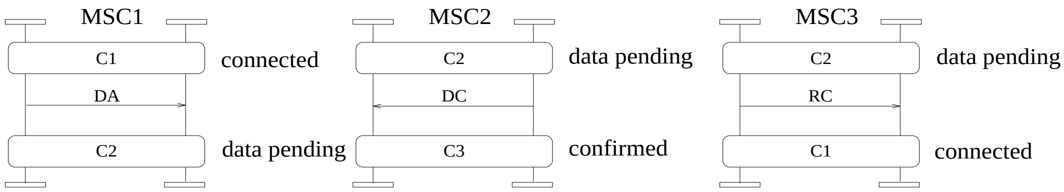

Figure 5. An MSC specification generating non-local control choice

3. CONDITIONS AND NON-LOCAL CHOICE

MSC specifications which include even simple use of conditions, such as in Figure 3, may induce multiple traces. From condition `C2` in `MSC1`, two different possible execution paths are possible, represented by `MSC2` and `MSC3`. This results in branching control, represented by the two next-event out-edges from each of nodes `u` and `v` in the MFG (Figure 4). If one process takes one branch and issues a `send`, the other process is constrained to take its corresponding branch to execute the corresponding `receive`. Thus the control branching must be synchronised. Since an MFG (and the MSCs) represent message exchange, it is consistent with this information that the synchronisation should be accomplished locally by each process, determined from 'observing' what happens during an execution. However, there are cases in which synchronisation cannot be achieved by methods purely local to each individual process (see Figure 6). We show that this non-local decision-making requires that each process have potential access to its complete choice history, a record which is of unbounded size in non-terminating processes.

3.1. Non-Local Choice, and Choice History

We show that history variables or their equivalent are needed to handle control branching that cannot be achieved by local means. The MSC standard Z.120 which considers conditions contains no recognition of the need for history variables in these circumstances. The spirit of MSCs would require that control choice synchronisation between processes which cannot be accomplished by each process acting independently should be accomplished by explicit exchange of messages indicating that a particular control branch is followed. We conclude that either control history variables should be explicitly introduced, or that the use of global initial or final conditions to represent control branching should be limited in some way by the standard. We are content here merely to show the need for a choice, but make no recommendation.

The example is a modification of Figure 3. In the MFG in Figure 4, the control-branch choice may be resolved locally. The first process, equipped with message-type-identification, awaits a signal from the second, and determines whether it is a `CC` or a `DR` message. Indeed, this would be the sensible way to implement it. Thus this example involves no non-local control choice synchronisation. An example in which the control choice must be somehow communicated non-locally is given by the MSC specification in Figure 5, which generates the MFG in Figure 6. In this example, the first and second processes must decide somehow 'together' whether they are going along the `DC` route or along the `RC` route. The MSC specifically disallows that one process could follow its left

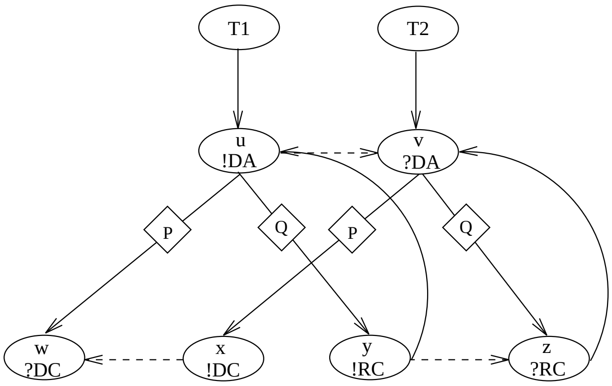

Figure 6. An MFG with non-local-choice nodes

branch and the other its right branch.

We denote this control synchronisation by labeling next-event edges at branches with predicates as in [19]. 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 MSCs starting with that symbol (e.g. the symbol `C2` in both Figures 3 and 5), we label corresponding next-event edges with predicate symbols as we construct the unfolding. For example, in Figure 6 we label edges corresponding to a transition through `C2` 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 next-event 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 MSC. In our example, labels `P` and `Q` used in Figure 6 would be respectively P_1 and P_2 according to this scheme, `P` being used to label the branches from MSC2 and `Q` those arising from MSC3. The modifications to the definition of MSC to allow next-event edge-labels, and formal definition of unfolding are straightforward (see [18, 19]).

3.2. Non-Local Choice May Imply Non-Finite-State Control

We show that if unrestricted conditions are allowed in MFGs, then some systems described by MFGs require non-finite-state control within individual processes. Either unbounded history variables are required to keep track of control choices, or very simple MFGs with conditions that require non-local choice, such as those in Figure 5, must be regarded as ill-formed.

The set of MFGs in Figure 5 may represent a higher-level requirement on system behavior. A system implementing this behavior must conform to the requirement. Let us suppose that the implementation may perform some rudimentary amount of error recovery on, say, temporary loss of transmission, such that the system continues to satisfy the MSC requirement.

We refer to the process whose 'line' is on the left [resp. right] in Figures 5 and 6 as the 'left' [resp. 'right'] process. Suppose we label the left branches of both processes in the

MFG in Figure 6 with P, to represent a choice of continuation with MSC2, and the right branches of both processes with Q to represent a continuation with MSC3. Suppose now that the system executes a trace in which the successive choices between continuation with MSC2 and MSC3 are made as follows: 1 P choice, followed by 2 successive Q choices, then 3 successive P, then 4 successive Q, then 5 successive P, ... The semantics of asynchronous communication allows the the left process to fall arbitrarily behind the right process. Somehow, the history of control choices made by the right process must be known by the left process, in order that the left process 'knows' which type of message to receive next. Suppose that the history of the last n control choices in each process is retained, in a 'history variable', which we can assume is an array of length n. After at most $\Sigma_{k \leq n} k = n.(n+1)/2$ branch choices, the history array contains either (a) all P's; (b) all Q's; or (c) at most one change from P's to Q's or *vice versa*. It is easy to see there are $2.(n-1)$ such possibilities for (c), which along with (a) and (b) yields $2.n$ possible configurations of the history variable after $n.(n-1)/2$ branch choices.

Suppose a recoverable fault occurs, and the system is restarted with (i) all message buffers intact as when the fault occurred; (ii) all data (including history variables) intact; (iii) program counters in the same position. Suppose the fault has occurred after somewhat more than $n.(n-1)/2$ branch choices. There are only $2.n$ possibilities for the configuration of the history variables in each process. Hence there are only $(2.n)^2$ total pairs of values of both history variables. Suppose there are k outstanding messages. It is easy to show that there are infinitely many possibilities of the choice history of each process compatible with each given configuration of the two history variables plus the number of outstanding messages. However, because of the construction of the example, unless the two processes start in corresponding places in the choice history, they will not fulfil the requirement expressed by the MSCs in Figure 5.

Not only the control branching history of the current process but also the history of other processes that are 'further ahead' must be known by a process, in order to make the appropriate control branches. To ensure that this condition is fulfilled, under even these mild recovery conditions, either (a) the process must have an implicit means of communicating with the environment in order to access this history, which is held somehow by the environment in a history variable; or if not then (b) the environment must communicate this information implicitly to a process as it happens, and the process must itself retain this entire history in a variable, which as we have seen must be of potentially unbounded size.

These situations are unsatisfactory since they require either greater or lesser roles for the environment as an implicit information-passer, and/or require each process to have potentially unbounded memory to retain control choice history.

A third alternative is to simply consider the requirement as expressed in Figure 5 to be ill-formed. But how to tell which uses of conditions are allowed? Well-formedness should be a matter for syntax, not for semantical analysis. Since non-local choice leads to undesirable requirements, it's necessary to find a syntactic restriction which corresponds to permitting local choice only.

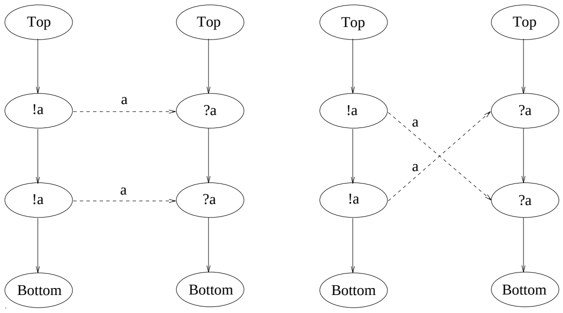

Figure 7. A cross-over MSC example, expressed as MFG

4. A CROSSING ANOMALY

In this section, we point out an anomaly arising from allowing messages to 'cross' in MSCs. We are led to conclude that there are non-trivial but non-obvious properties of the environment implicitly contained in certain types of MSC descriptions. We regard it as infelicitous that such implicit properties should be required.

The MSC standard [16] allows crossing of signals to occur. The two MFGs of Figure 7 representing two simple MSCs describe different system behaviors. In both cases an identical type of signal is transmitted twice. The second case differs from the first in that a 'cross-over' of the messages is specified. The observable behavior of each individual process is identical in the two examples (one sends two 'a' signals, the other receives two 'a' signals), hence code implementing each process will be identical in both examples. However, the two examples have different sets of valid traces. The set of traces (interleaved observable events) of the first MSC is $\{<!a, !a, ?a, ?a>, <!a, ?a, !a, ?a>\}$, and that of the second is is $\{<!a, !a, ?a, ?a>\}$. A system exhibiting behavior $<!a, ?a, !a, ?a>$ satisfies the first specification but not the second. However, a system exhibiting behavior $\{<!a, !a, ?a, ?a>\}$ and no other may satisfy either specification.

Since there is no difference in process code for the two examples, the different trace sets must be accounted for by a difference in the behavior of the environment. Thus, even though the environment is not explicitly represented in this specification, its properties must be invoked implicitly. One may try to resolve this problem by representing the environment explicitly, as a single vertical line like a process axis, provided it engages in message interaction with the processes (cf. [5]). However, this is no solution. Even if the environment is explicitly represented as such a third axis, firstly one can still obtain analogous process behavior by using cross-overs, and secondly this behavior may only be obtained by using a crossover (we leave this as an easy exercise).

One criterion for a good specification method (to distinguish it, say, from the average programming language) is that all asserted properties be represented explicitly, including constraints from the environment. Our example shows that MSCs with cross-over do not pass this test. Further, even if the environment is explicit, cross-over is at best an unintuitive method of enforcing certain orderings on behavior, for which there is no other representation mechanism. Such 'programming tricks' have no place in a good description method.

5. LIVENESS PROPERTIES AND ACCEPTANCE CRITERIA

Liveness Conditions. Given that a system follows a finite state-transition graph, there is nevertheless a question as to whether all traces through this graph are acceptable traces of the system, or whether only a subset of them are. We showed in [19] that general MFGs define a very limited set of liveness properties. In order to facilitate the expression of a wider array of liveness properties, it is necessary to go beyond the Global State Transition Graph (GSTG [19]) to consider which traces through the graph are allowed by the description (along with the liveness properties) and which aren't. A standard way to express these conditions is to consider the GSTG as providing most of the definition of an ω-automaton, lacking only an end-state definition, and to provide that end-state definition.

Büchi- and Other ω-Automata. Since traces may be infinite, a finite-state semantics requires use of a finite-state automaton which accepts infinite strings. The Büchi automaton is probably the most well-known of these, and has been used in the determination of safety and liveness properties of distributed systems [1], [2]. These automata are similar to ordinary finite automata, except for the acceptance condition. Büchi automata include in their definition a set of states called the *end-state set*. A (possibly infinite) string is accepted by a Büchi automaton just in case the automaton passes through an end state unboundedly often on the string (for finite strings, the final state must be an end state).

Given a general MFG specification, involving a family of MFGs with conditions, the GSTG is uniquely determined [19]. From this graph, various different end-state definitions will define various different ω-automata, each of which identifies the set of system traces specified by the MFG with the set of accepted traces of the automaton. The Global-State Transition Graph itself defines a Büchi automaton, namely the one in which the end-states are the set of all states. Even though Büchi automata define a very rich class of trace-sets in order to use them flexibly one must be at liberty freely to design the state set. We are constrained by having to use the global states defined in the GSTG, and we show now that the Büchi acceptance condition does not suffice to define certain natural liveness conditions, given the GSTG states and transitions. Therefore other acceptance conditions may be preferable.[7]

Büchi automata suffice to describe liveness properties of systems, provided that the specifier is free to choose the states of the automaton during the course of designing an automaton to accept precisely the desired traces. However, the global states of an MFG are specified, uniquely, by the MFG. We are not free to choose an alternative state set[8]. If one needs to specify liveness properties of an MFG description, the definition of Büchi acceptance may not suffice. Consider the GSTG on the right hand side of in Figure 8,

[7] We are grateful to Bob Kurshan for this observation.

[8] It may be possible to devise a general transformation of a GSTG into another GSTG G' satisfying identical safety properties such that a different set of of liveness properties may be defined for G' based on Büchi acceptance. However, we don't know of one, or whether such a transformation might have other disadvantages.

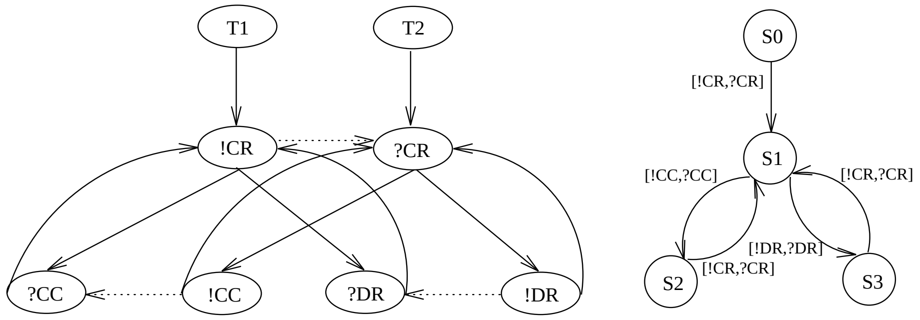

Figure 8. A MFG and the corresponding MFG whose liveness may not be specified by Büchi acceptance

derived from the MFG on the left hand side in the same Figure[9]. It represents a system which when in state $S1$ makes a non-deterministic choice between transiting into two states $S2$ and $S3$, and then returns to state $S1$. A important liveness property may be to require that the system performs a *fair* choice between the subbehaviours $S2$ and $S3$. This is expressed in temporal logic as $(\Box\Diamond at\_S2 \wedge \Box\Diamond at\_S3)$[10]. It is easy to see that there is no set of end-states under which this liveness condition is expressed by Büchi acceptance (just look at the 15 possible non-trivial end-state sets).

The proper definition of liveness properties for the GSTG and therefore for the system described by the MFGs may therefore be accomplished better by temporal logic formulae than by Büchi acceptance. This is because the design of the Büchi automaton is constrained by the necessary selection of a particular transition graph, the GSTG. It remains to be seen whether other acceptance criteria suffice to define automata suitable for all potential liveness criteria. In the meantime, temporal logic appears to be able to define the liveness criteria which users of MFGs may want. We have defined the precise relationship between temporal logic assertions and MFG 'executions' in [19].

6. CONCLUSIONS

We discussed four issues concerning the semantics of Message Flow Graphs (MFGs). We required that a system described by an MFG has global states with respect to its message-passing behavior, with instantaneous transitions between these states effected by atomic message-passing actions. Under this assumption, we argued (a) that the collection of global message-passing states defined by an MFG is finite (whether for synchronous, asynchronous, or partially-asynchronous message-passing); (b) that the unrestricted use of 'conditions' requires processes to have access to control history variables of unbounded size; (c) that allowing 'crossing' messages of the same type implies certain properties of

[9]Note that the signal arrows in this MFG indicate *synchronous* communication, since we can make our point with this GSTG, which is much simpler than that derived from the same MFG with asynchronous communication. MFGs handle synchronous communication without modification. as shown in [19]. This may be another indication that life with synchronous communication is easier.

[10]As described in [22]. at_S2 and at_S3 are state predicates asserting that the system is in state $S2$ or $S3$, respectively.

the environment that are neither explicit nor desirable, and (d) that liveness properties of MFGs are more easily expressed by temporal logic formulas over the control states than by Büchi acceptance conditions over the same set of states.

Acknowledgements
We thank Professor Ken Turner for some helpful discussions on MSCs.

REFERENCES

1. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
2. B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages*, 11(1):147–167, 1989.
3. F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall International, 1991.
4. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
5. M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.
6. CCITT. Recommendation Z.100: CCITT Specification and Description Language (SDL). CCITT, Geneva, 1992.
7. 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.
8. C. Courcoubetis, editor. *Computer Aided Verification: Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.
9. J. De Man. Towards a formal semantics of message sequence charts. In *[12]*, pages 157–166. 1993.
10. M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azéma, and V. Chari, editors. *The Formal Description Technique Estelle*. North-Holland, 1989.
11. A.J.M. Donaldson. Specification of quality of service measurement points in JVTOS. Master's thesis, University of Stirling, Scotland, U.K., September 1993.
12. O. Færgemand and A. Sarma, editors. *SDL '93: Using Objects*. North-Holland, 1993.
13. P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definition for Message Sequence Charts. In *[12]*, pages 179–190. 1993.
14. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
15. G. J. Holzman. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.
16. ITU. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1993. Revised Version.
17. I. Jacobson. *Object Oriented Software Engineering*. 1992.
18. P. B. Ladkin and S. Leue. What do Message Sequence Charts mean? In *[25]*. 1994. To appear.
19. P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 1995. To appear.

20. P.B. Ladkin and B.B. Simons. *Static Analysis of Interprocess Communication*. Lecture Notes in Computer Science. Springer-Verlag, 1994. To appear.
21. 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.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
23. S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous interworkings. In *[12]*, pages 167–178. 1993.
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.
25. R. L. Tenney, P. D. Amer, and M. Ü. Uyar, editors. *Formal Description Techniques, VI*. IFIP Transactions C, Proceedings of the Sixth International Conference on Formal Description Techniques. North-Holland, 1994.
26. 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.