Inference of Message Sequence Charts*

Rajeev Alur, Kousha Etessami, Mihalis Yannakakis†

January 15, 2003

Abstract

Software designers draw Message Sequence Charts for early modeling of the individual behaviors they expect from the concurrent system under design. Can they be sure that precisely the behaviors they have described are realizable by some implementation of the components of the concurrent system? If so, can we automatically synthesize concurrent state machines realizing the given MSCs? If, on the other hand, other unspecified and possibly unwanted scenarios are

*A preliminary version of this paper appears in *Proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000. A journal version will appear in IEEE Transactions in Software Engineering, but due to space limitations in the journal, this is the fuller version.

†R. Alur is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, USA; email: alur@cis.upenn.edu. K. Etessami is with the School of Informatics, University of Edinburgh, Edinburgh, UK; email: kousha@inf.ed.ac.uk. M. Yannakakis is with the Department of Computer Science, Stanford University, Stanford, USA; email: mihalis@cs.stanford.edu. R. Alur was supported in part by NSF CAREER award CCR97-34115, NSF grant CCR99-70925, and NSF ITR award IR/SY01-21431.

"implied" by their MSCs, can the software designer be automatically warned and provided the implied MSCs? In this paper we provide a framework in which all these questions are answered positively. We first describe the formal framework within which one can derive implied MSCs, and then provide polynomial-time algorithms for implication, realizability, and synthesis.

1 Introduction

Message Sequence Charts (MSCs) are a commonly used visual description of design requirements for concurrent systems such as telecommunications software [1, 2], and have been incorporated into software design notations such as UML [3]. Requirements expressed using MSCs have been given formal semantics, and hence, can be subjected to analysis. Since MSCs are used at a very early stage of design, any errors revealed during their analysis yield a high pay-off. This has already motivated the development of algorithms for a variety of analyses including detecting race conditions and timing conflicts [4], pattern matching [5], detecting non-local choice [6], and model checking [7], and tools such as uBET [8], MESA [9], and SCED [10]. An individual MSC depicts a potential exchange of messages among communicating entities in a distributed software system, and corresponds to a single (partial-order) execution of the system. The requirements specification is given as a set of MSCs depicting different possible executions. We show that such a specification can be subjected to an algorithm for checking completeness and detecting unspecified MSCs that are implied, in that they must exist in every implementation of the input set.

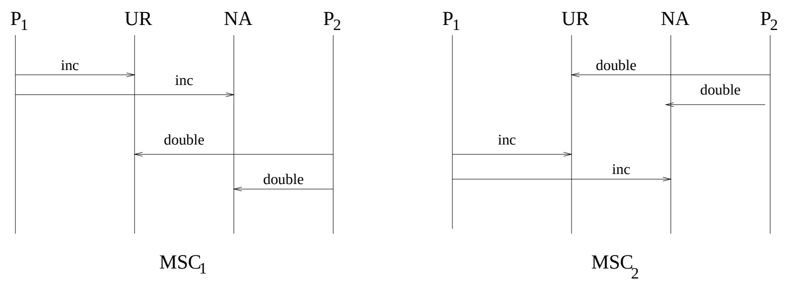

Figure 1: Two seemingly "correct" scenarios, updating fuel amounts

Such implied MSCs arise because the intended behaviors in different specified MSCs can combine in unexpected ways when each process has only its own local view of the scenarios. Our notion of implied MSCs is thus intimately connected with the underlying model of concurrent state machines that produce these behaviors. We define a set of MSCs to be *realizable* if there exist concurrent automata which implement precisely the MSCs it contains.

We study two distinct notions of MSC implication, based on whether the underlying concurrent automata are required to be *deadlock-free* or not. Deadlocks in distributed systems can occur, e.g., when each process is waiting to receive something that has yet to be sent. We give a precise formalization of deadlocks in our concurrency framework.

Using our formalization, we show that MSCs can be studied via their linearizations. We then establish realizability to be related to certain closure conditions on languages. It turns out that, while arbitrary realizability is a global requirement that is computationally expensive to check (coNP-complete), safe (deadlock-free) realizability corresponds to a closure condition that can be formulated locally and admits a polynomial-time solution. We show that with a judicious choice of preprocessing and data structures, safe realizability can be checked in time $O(k^2n + rn)$, where n is the number of processes, k is the number of MSCs, and r is the number of events in the input MSCs. If the given MSCs are not safely realizable, our algorithm produces missing implied (partial) scenarios to help guide the designer in refining and extending the specification.

We first describe our results in the setting of asynchronous communication with non-FIFO message buffers between each pair of processes. In Section 8, we point out how our results can be generalized to a variety of communication architectures in a generic manner.

Related Work

The formalization of MSCs using labeled partially-ordered structures, or as Mazurckiewicz traces, has been advocated by many researchers [4, 6, 17], and we follow the same approach. Many researchers have argued that in order to use MSCs in the automated analysis of software, the information MSCs provide needs to be reconciled with and incorporated into the state-based models of systems used later in the software life-cycle, and consequently, have proposed mechanical translations from MSC specifications to state machines [11, 12, 13, 14, 15, 16, 17]. The question of implication is closely related to this synthesis question. In fact, we give a synthesis algorithm which is in the same spirit as others proposed in the literature: to generate the state-machine corresponding to a process P, consider the projections of the given scenarios onto process P, and introduce a control point after every event of process P. However, our focus differs substantially from the earlier work on translating MSCs to state machines. First, we are interested in detecting implied scenarios, and in avoiding deadlocks in our implementations. Second, we emphasize efficient analysis algorithms, and in particular, present an efficient polynomial-time algorithm to detect safely implied MSCs and solve safe realizability, avoiding the state-explosion which typically arises in such analysis of concurrent system behavior. Finally, we present a clean language-theoretic framework to formalize these problems via closure conditions. A rigorous mathematical treatment of the synthesis problem has been developed independently in [17]. Their formalization differs from ours in two important ways. First, in [17], the MSCs specify only the communication pattern, but not the message content, and the automata implementing the MSCs can choose the message vocabulary. Second, the accepting conditions for the communicating automata are specified globally, while in our framework each automaton has its own local accepting states. The main result of [17] shows how to construct a set of communicating automata that generates the behaviors specified by a *regular* collection of MSCs, and thus, in their formalization, every finite set of MSCs would be realizable. We believe that our definition, particularly, the accepting states being local, is more suitable for distributed systems.

It is worth noting that inferring sequential state machines from example executions is a well-studied topic in automata theory [18, 19]. In our setting, only "positive" examples are given, but the executions are partially ordered and we infer "distributed" implementations.

2 Sample MSC Inference

We motivate inference of missing scenarios using an example related to serializability in database transactions (see, e.g., [20]). Consider the following standard example, described in the setting of a nuclear power plant. Two clients, P_1 and P_2, seek to perform remote updates on data used in the control of a nuclear power plant. In this database the variable UR controls the amount of Uranium fuel in the daily supply at the plant, and the variable NA controls the amount of Nitric Acid. It is necessary that these amounts be equal in order to avoid a nuclear accident. Consider the two MSCs in Figure 1 which describe how distinct transactions may be performed by each of the clients, P_1 and P_2. The "inc" message denotes a request to increment the fuel amount by one unit, while the "double" message denotes a request to double the fuel amount. In the MSCs, we interpret the point where a message arrow leaves the time line of a process to be the instance when the requested operation labeling the transition is issued, and we interpret the point where a message arrives at the time line of its destination process to be the instance when the requested operation is acted on and executed.[1] In the first scenario, P_1 first increments the amounts of both ingredients, and then, P_2 doubles the amounts of both ingredients. In the second scenario, first P_2 doubles the two amounts, and then, P_1 increments both the amounts. In both scenarios, after both transactions have finished, the desired property, equal amounts of uranium and nitric acid, is maintained. However, these MSCs imply the possibility of MSC_{bad}

[1] This interpretation is consistent with our concurrent state machine interpretation of MSCs in the rest of this paper.

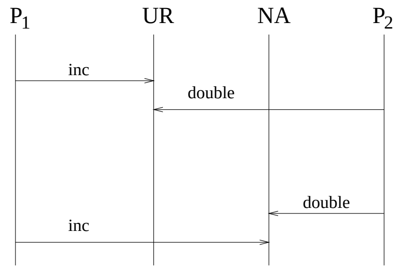

Figure 2: Implied MSC_{bad}: Incorrect fuel mix

in Figure 2. This is because, as far as each process can locally tell, the scenario is proceeding according to one of the two given scenarios. However, the scenario results in different amounts of uranium and nitric acid being mixed into the daily supply, and in the potential for a nuclear accident. Note that either of the MSCs in Figure 1 alone will not necessarily imply MSC_{bad}, because in each case the protocol could specify that client $P1$ updates the fuel levels first, followed by $P2$, or vice versa.

3 Message Sequence Charts

In this section, we define message sequence charts, and study the properties of executions definable using them. Our definition captures the essence of the *basic MSCs* of the ITU standard MSC'96 [1], and is analogous to the definitions of labeled MSCs given in [4, 7].

Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of processes, and Σ be a message alphabet. We write $[n]$ for $\{1, \ldots, n\}$. We use the label *send(i,j,a)* to denote the event "process P_i sends the message a to process P_j." Similarly, *receive(i,j,a)* denotes the event "process P_j receives the message a from process P_i." Define the set $\hat{\Sigma}^S = \{send(i,j,a) \mid i,j \in [n] \;\&\; a \in \Sigma\}$

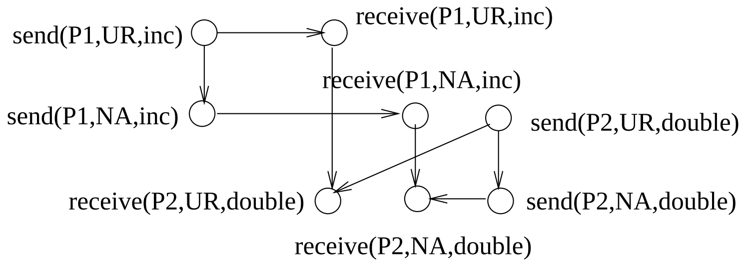

Figure 3: Partial order representation of MSC_1

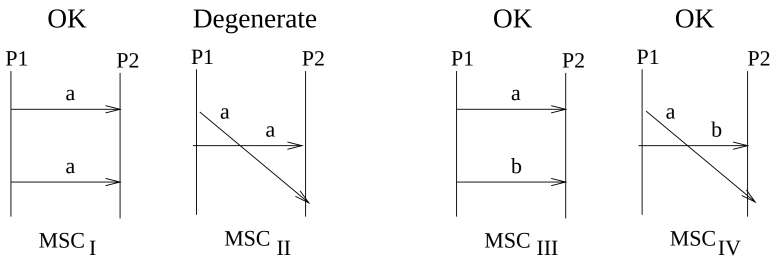

Figure 4: Degeneracy in MSCs

of *send labels*, the set $\hat{\Sigma}^R = \{receive(i,j,a) \mid i,j \in [n] \;\&\; a \in \Sigma\}$ of *receive labels*, and $\hat{\Sigma} = \hat{\Sigma}^S \cup \hat{\Sigma}^R$ as the set of *event labels*. A Σ*-labeled MSC* M over processes $\mathcal{P}$ is given by:

1. a set E of events which is partitioned into a set S of "send" events and a set R of "receive" events;

2. a mapping $p : E \mapsto [n]$ that maps each event to a process on which it occurs;

3. a bijective mapping $f : S \mapsto R$ between send and receive events, matching each send with its corresponding receive;

4. a mapping $l : E \mapsto \hat{\Sigma}$ which labels each event such that $l(S) \subseteq \hat{\Sigma}^S$ and $l(R) \subseteq \hat{\Sigma}^R$, and furthermore for consistency of labels, for all $s \in S$, if $l(s) = send(i,j,a)$ then

$p(s) = i$ and $l(f(s)) = \mathit{receive}(i, j, a)$ and $p(f(s)) = j$;

5. for each $i \in [n]$, a total order $\leq_i$ on the events of process P_i, that is, on the elements of $p^{-1}(i)$, such that the transitive closure of the relation

$$\leq \doteq \cup_{i \in [n]} \leq_i \cup \{(s, f(s)) \mid s \in S\}$$

is a partial order on E.

Note that the total order $\leq_i$ denotes the (visual) temporal order of execution of the events of process P_i. The requirement that $\leq$ is a partial order enforces the notion that "messages cannot travel back in time". Thus, an MSC can be viewed as a set E of $\hat{\Sigma}$-labeled events partially ordered by $\leq$. The partial order corresponding to the first MSC of Figure 1 is shown in Figure 3.

Besides the above, we require our MSCs to satisfy an additional *non-degeneracy* condition. We will say an MSC is degenerate if it reverses the order in which two *identical* messages sent by some process P_i are received by another process P_j. More formally, an MSC M is *degenerate* if there exist two send-events e_1 and e_2 such that $l(e_1) = l(e_2)$ and $e_1 < e_2$ and $f(e_2) < f(e_1)$. To understand this notion, consider the four MSCs in Figure 4. In both MSC_I and MSC_{II}, $P1$ sends two a's and $P2$ receives two a's. The receiving process has no way to tell which of the messages is which, since the messages themselves are indistinguishable. If one wants to distinguish the two MSCs, then one needs to associate, e.g., time-stamps to the two messages. But then we are really dealing with distinct messages, as in MSC_{III} and MSC_{IV}. In these scenarios, process $P2$ can clearly tell the distinct messages apart, and we in general accept such reorderings.[2] Note that, the partial order on the events induced by MSC_I is more general than that induced by MSC_{II}, in that it allows strictly more possible interleaved executions. Henceforth, throughout the rest of this paper, MSCs refer to *non-degenerate* MSCs.

Given an MSC M, a *linearization* of M is a string over $\hat{\Sigma}$ obtained by considering a total ordering of the events E that is consistent with the partial order $\leq$, and then replacing each event by its label. More precisely, a word $w = w_1 \cdots w_{|E|}$ over the alphabet $\hat{\Sigma}$ is a linearization of an MSC M iff there exists a total order $e_1 \cdots e_{|E|}$ of the events in E such that (1) whenever $e_i \leq e_j$, we have $i \leq j$, and (2) for $1 \leq i \leq |E|$, $w_i = l(e_i)$.

Not all sequences of *send*'s and *receive*'s can arise as legitimate linearizations of MSCs. For example, a message received must already have been sent. What characterizes the words that can arise as linearizations of MSCs? Let $\#(w, x)$ denote the number of times the symbol x occurs in w. Let $w|_i$ denote the projection of the word w that retains only those events that occur on process P_i (that is, events of type *send(i,j,a)* or *receive(j,i,a)*). The two conditions necessary for a word to be in an MSC language are the following:

Well-formedness. A word w over $\hat{\Sigma}$ is well-formed if all receive events have matching sends. Formally, a symbol $x \in \hat{\Sigma}$ is *possible* after a word v over $\hat{\Sigma}$, if, either $x \in \hat{\Sigma}^S$ or $x = \mathit{receive}(i,j,a)$ with $\#(v, \mathit{send}(i,j,a)) - \#(v, \mathit{receive}(i,j,a)) > 0$. A word w is *well-formed* if for every prefix vx of w, x is possible after v.

[2] When dealing specifically with FIFO architectures, via the general framework in section 8, we will explicitly forbid crossing of the kind in MSC_{IV} as well.

Completeness. A word w over $\hat{\Sigma}$ is complete if all send events have matching receives.

More precisely, a well-formed word w over $\hat{\Sigma}$ is called *complete* iff for all processes $i, j \in [n]$ and messages $a \in \Sigma$, $\#(w, send(i,j,a)) - \#(w, receive(i,j,a)) = 0$.

It is easy to check that every linearization of an MSC is well-formed and complete. The converse also holds:

Proposition 1 *A word w over the alphabet $\hat{\Sigma}$ is a linearization of an MSC iff it is well-formed and complete.*

Proof. It is easy to verify that any linearization of an MSC is well-formed and complete. For the other direction, given a well-formed and complete word w, we build from it a *canonical* MSC, $msc(w)$. $msc(w)$ is build progressively from prefixes of w, starting with the empty prefix. For a prefix $w'receive(i,j,a)$, we *match* the last receive in the prefix with the first occurrence of $send(i,j,a)$ in w' which is yet to be matched. By the fact that w is well-formed, we know this can always be done. By the fact that w is complete, we know that all sends will be matched with corresponding receives. $msc(w)$ will automatically be non-degenerate, because we always match receive messages with the first possible send message, so crossings on the same message cannot occur. In fact, this is necessary because of non-degeneracy, hence $msc(w)$ is the unique nondegenerate MSC with linearization w.

Given an MSC M, define the *projection* of M on the ith process, denoted $M|_i$, to be the ordered sequence of labels of events occurring at process i in the MSC M. Define similarly the projection $w|_i$ of a word w on the ith process to be the subsequence of w that involves the send and receive events of process P_i. Note that in the proof of Proposition 1, the canonical MSC $msc(w)$ only depends on the sequences $w|_i$, and not on the their actual interleaving in the linearization w. Since $msc(w)$ is the only nondegenerate MSC with linearization w, it follows that:

Proposition 2 *An MSC M over $\{P_1, \ldots, P_n\}$ is uniquely determined by the sequences $M|_i$, $i \in [n]$. Thus, we may equate $M \cong \langle M|_i \mid i \in [n] \rangle$.*[3] *Likewise, a well-formed and complete word w over $\hat{\Sigma}$ uniquely characterizes an MSC M_w given by $\langle w|_i \mid i \in [n] \rangle$.*

For an MSC M, define $L(M)$ to be the set of all linearizations of M. Note that, by the proposition, any two different MSCs have disjoint linearization sets. For a set $\mathcal{M}$ of MSCs, the language $L(\mathcal{M})$ is the union of languages of all MSCs in M. We say that a language L over the alphabet $\hat{\Sigma}$ is an *MSC-language* if there is a set $\mathcal{M}$ of MSCs such that L equals $L(\mathcal{M})$. What are the necessary and sufficient conditions for a language to be an MSC-language? First, all the words must be well-formed and complete. Second, in the MSC corresponding to a word, the events are only partially ordered, so once we include a word, we must include all *equivalent* words that correspond to other linearizations of the same MSC. This notion of equivalence corresponds to permuting the symbols in the word while respecting the ordering of the events on individual processes and the matching of send-receive events. This notion is formalized below.

Closure Condition CC1. Given a well-formed word w over the alphabet $\hat{\Sigma}$, its *interleaving closure*, denoted $\langle w \rangle$, contains all *well-formed* words v over $\hat{\Sigma}$ such that

[3] Note that the MSC M is non-degenerate by assumption, and this assumption is required.

for all i in $[n]$, $w|_i = v|_i$. A language L over $\hat{\Sigma}$ satisfies closure condition CC1 if for every $w \in L$, $\langle w \rangle \subseteq L$.

Note that CC1 considers only well-formed words, so matching of receive events is implicitly ensured. Also, if a word is complete, then so are all the equivalent ones. Now, the following theorem characterizes the calss of MSC-languages:

Theorem 3 *A language L over the alphabet $\hat{\Sigma}$ is an MSC-language iff L contains only well-formed and complete words and satisfies closure condition CC1.*

Proof. The proof follows immediately from the fact that one can recover uniquely an MSC M from its projections $M|_i$, as well as from $w|_i$'s, where w is a linearization of M (Proposition 2).

It is worth noting that CC1 can alternatively be formalized using semi-traces over an appropriately defined independence relation over the alphabet $\hat{\Sigma}$ (see, for instance, [21]).

We will find useful the notion of a partial MSC. A *partial MSC* is given by a well-formed, not necessarily complete, word v, or, equivalently, by the projections $v|_i$ of such a sequence. We call an MSC M a *completion of* a partial MSC $v \cong \langle v|_i \mid i \in [n] \rangle$, if $v|_i$ is a prefix of $M|_i$ for all i.

4 Concurrent Automata

Our concurrency model is based on the standard buffered message-passing model of communication. There are several choices to be made with regard to the particular communication architecture of concurrent processes, such as synchrony/asynchrony and the queuing disciplines on the buffers. We will show in section 8 that our results apply in a general framework which captures a variety of alternative architectures. However, for clarity of presentation in the main body of the paper, we fix our architecture to a standard asynchronous setting, with arbitrary (i.e., unbounded and not necessarily FIFO) message buffers between all pairs of processes. We now formally define our automata A_i, and their (asynchronous) product $\Pi_{i=1}^n A_i$, which captures their joint behavior.

As in the previous section, let Σ be the message alphabet. Let $\hat{\Sigma}_i$ be the set of labels of events belonging to process P_i, namely, the messages of the form $send(i, j, a)$ and $receive(j, i, a)$. The behavior of process P_i is specified by an automaton A_i over the alphabet $\hat{\Sigma}_i$ with the following components: (1) a set Q_i of states, (2) a transition relation $\delta_i \subseteq Q_i \times \hat{\Sigma}_i \times Q_i$, (3) an initial state $q_i^0 \in Q_i$, and (4) a set $F_i \subseteq Q_i$ of accepting states.

To define the joint behavior of the set of automata A_i, we need to describe the message buffers. For each ordered pair (i, j) of process indices, we have two message buffers $B^s_{i,j}$ and $B^r_{i,j}$. The first buffer, $B^s_{i,j}$, is a "pending" buffer which stores the messages that have been sent by P_i but are still "in transit" and not yet accessible by P_j. The second buffer $B^r_{i,j}$ contains those messages that have already reached P_j, but are not yet accessed and removed from the buffer by P_j. Define Q_Σ to be the set of multi-sets over the message alphabet Σ. We define the buffers as elements of Q_Σ (FIFO queues, on the other hand, can be viewed as sequences over Σ). Thus, for $i, j \in [n]$, we have $B^s_{i,j}, B^r_{i,j} \in Q_\Sigma$. The operations on buffers are defined in the natural way: e.g., adding a message a to a buffer B corresponds to incrementing the count of a-messages by 1.

We define the asynchronous product automaton $A = \Pi_{i=1}^n A_i$ over the alphabet $\hat{\Sigma}$, given by:

States. A state q of A consists of the (local) states q_i of component processes A_i, along with the contents of the buffers $B^s_{i,j}$ and $B^r_{i,j}$. More formally, the state set Q is $\times_{i=1}^n Q_i \times Q_\Sigma^{n^2} \times Q_\Sigma^{n^2}$.

Initial state. The initial state q_0 of A is given by having the component for each process i be in the start state q_i^0, and by having every buffer be empty.

Transitions. In the transition relation $\delta \subseteq Q \times (\hat{\Sigma} \cup \{\tau\}) \times Q$, the τ-transitions model the transfer of messages from the sender to the receiver. The transitions are defined as follows:

1. For an event $x \in \hat{\Sigma}_i$, $(q, x, q') \in \delta$ iff (a) the local states of processes $k \neq i$ are identical in q and q', (b) the local state of process i is q_i in q and q'_i in q' such that $(q_i, x, q'_i) \in \delta_i$, (c) if $x = \mathit{receive}(j, i, a)$ then the buffer $B^r_{j,i}$ in state q contains the message a, and the corresponding buffer in state q' is obtained by deleting a, (d) if $x = \mathit{send}(i, j, a)$, the buffer $B^s_{i,j}$ in state q' is obtained by adding the message a to the corresponding buffer in state q, and (e) all other buffers are identical in states q and q'.

2. There is a τ-labeled transition from state q to q', iff states q and q' are identical except that for one pair (i, j), the buffer $B^s_{i,j}$ in state q' is obtained from the

corresponding buffer in state q by deleting one message a, and the buffer $B^r_{i,j}$ in state q' is obtained from that in q by adding that message a.

Accepting states. A state q of A is accepting if for all processes i, the local state q_i of process i in q is accepting, and all the buffers in q are empty.

We associate with $A = \Pi_i A_i$ the language of possible executions of A, denoted $L(A)$, which consists of all those words in $\hat{\Sigma}^*$ leading A from start state q_0 to an accepting state, where τ-transitions are viewed as ϵ-transitions in the usual automata-theoretic sense. The following property of $L(A)$ is easily verified from definitions:

Proposition 4 *Given any sequence of automata $\langle A_i \mid i \in [n] \rangle$, $L(\Pi_i A_i)$ is an MSC-language.*

Note that for any MSC language L and MSC M, either $L(M) \cap L = \emptyset$ or $L(M) \subseteq L$; this follows from the fact that distinct MSCs have disjoint linearization sets. Hence, for any set of concurrent automata A_i, the language $L(\Pi_i A_i)$ of the product of the automata either contains all linearizations of an MSC M or it contains none.

5 Weak Realizability

When can we, given MSCs $\mathcal{M}$, actually realize $L(\mathcal{M})$ as the language of concurrent automata? In other words, when are no other MSCs implied:

Definition 1 *Given a set $\mathcal{M}$ of MSCs, and another MSC M', we say that $\mathcal{M}$* weakly implies *M', and denote this by*

$$\mathcal{M} \overset{W}{\vdash} M'$$

if for any sequence of automata $\langle A_i \mid i \in [n]\rangle$, *if* $L(\mathcal{M}) \subseteq L(\Pi_i A_i)$ *then* $L(M') \subseteq L(\Pi_i A_i)$.

We want to characterize this implication notion, and furthermore detect when a set $\mathcal{M}$ is realizable:

Definition 2 *A language* L *over the alphabet* $\hat{\Sigma}$ *is* weakly realizable *iff* $L = L(\Pi_i A_i)$ *for some* $\langle A_i \mid i \in [n]\rangle$. *A set of MSCs* $\mathcal{M}$ *is said to be weakly realizable if* $L(\mathcal{M})$ *is weakly realizable.*

The reason for the term "weak" is because we have not ruled out the possibility that the product automaton $\Pi_i A_i$ might necessarily contain the potential for *deadlock*. In general we wish to avoid this. We will take up the issue of deadlock in the next section. We now describe a closure condition on languages which captures weak implication and thus weak realizability.

Closure Condition CC2. A language L over the alphabet $\hat{\Sigma}$ satisfies *closure condition CC2* iff for all well-formed and complete words w over $\hat{\Sigma}$: if for every process P_i there exists a word v^i in L such that $w|_i = v^i|_i$, then w is in L.

Condition CC2 says that if, for every process P_i, the events occurring on P_i in word w are consistent with the events occurring on P_i in some word known to be in the language L, and w is well-formed, then w must be in L, i.e., w is *implied*. Intuitively, this notion says that L can be constructed from the projections of the words in L onto individual processes. Note that CC2 immediately implies CC1. The other direction does not hold.

Going back to our example from Section 2, the language $L(\{\mathrm{MSC}_1, \mathrm{MSC}_2\})$ generated by the two given MSCs is not closed under CC2 but is under CC1. In particular, consider the word w, a linearization of MSC_{bad}, given by

$$\begin{array}{l} send(P1, UR, inc) \\ receive(P1, UR, inc) \\ send(P2, UR, double) \\ receive(P2, UR, double) \\ send(P2, NA, double) \\ receive(P2, NA, double) \\ send(P1, NA, inc) \\ receive(P1, NA, inc). \end{array}$$

The word w is not in $L(\{\mathrm{MSC}_1, \mathrm{MSC}_2\})$, but the projections $w|_{P1}$ and $w|_{P2}$ are consistent with both the MSCs, while the projection $w|_{UR}$ is consistent with MSC_1 and $w|_{NA}$ is consistent with MSC_2. Thus, any language satisfying CC2 and containing linearizations of MSC_1 and MSC_2 must also contain w. Thus

$$\{MSC_1, MSC_2\} \overset{W}{\vdash} MSC_{bad}.$$

The next theorem says that condition CC2 captures the essence of weakly realizable languages.

Theorem 5 *A language L over the alphabet $\hat{\Sigma}$ is weakly realizable iff L contains only well-formed and complete words and satisfies CC2.*

Proof. Suppose L is weakly realizable. There exist automata A_i such that $L = L(A)$ for $A = \Pi A_i$. A can accept only well-formed and complete words. We show L satisfies

CC2. Consider a well-formed, complete word w, and for each i, let $v^i \in L$ be a word such that $w|_i = v^i|_i$. Consider the accepting run of A on v^i, retain only the transitions corresponding to events in $\hat{\Sigma}_i$, and retain only the local state of process i. This gives a sequence r_i of states of A_i, which is an accepting run of A_i over $w|_i$. Now, the local runs r_i can be combined to obtain an accepting run of the product A over w. During this construction, we give priority to the τ-transitions: every transition labeled with *send(i,j,a)* is immediately followed by a τ-transition that moves the message a from the buffer $B^s_{i,j}$ to $B^r_{i,j}$.

Conversely, consider a language L with the CC2 property and containing only well-formed and complete words. Define the languages L_i over $\hat{\Sigma}_i$ to contain the projections $w|_i$ of the words $w \in L$. Let A_i be an automaton over $\hat{\Sigma}_i$ that accepts L_i (note that if L is regular, then so is L_i, ensuring that the automata A_i will be finite state). Let A be the product ΠA_i. We must show that $L(A)$ equals L. It follows from definitions that if $w \in L$, then w is accepted by A. On the other hand, if w is accepted by A, then for each process i, $w|_i$ is accepted by A_i, and hence, $w|_i \in L_i$, and by CC2 closure of L, w is in L.

We thus have characterizations of weak implication and realizability of MSCs:

Corollary 6 *Given MSC set $\mathcal{M}$, and MSC M': $\mathcal{M} \overset{W}{\vdash} M'$ if and only if for each process $i \in [n]$, there is an MSC $M^i \in \mathcal{M}$ such that $M'|_i = M^i|_i$. An MSC family $\mathcal{M}$ is weakly realizable iff $L(\mathcal{M})$ satisfies CC2.*

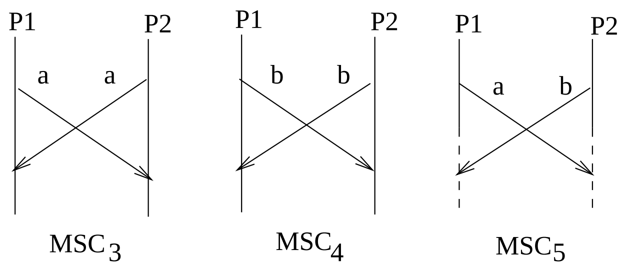

Figure 5: Weakness of weak realizability

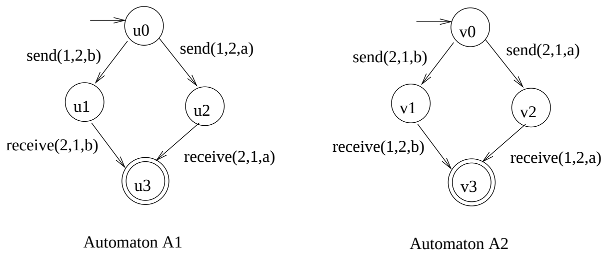

Figure 6: Concurrent automata corresponding to MSC_3 and MSC_4

6 Safe Realizability

The weakness of weak realizability stems from the fact that we are not guaranteed a well behaved product $\Pi_i A_i$. In particular, in order to realize the MSCs, or the language, there may be no way to avoid a deadlock state in the product.

To describe this formally, consider a set A_i of concurrent automata and the product $A = \Pi_i A_i$. A state q of the product A is said to be a *deadlock* state if no accepting state of A is reachable from q. For instance, a non-accepting state in which all processes are waiting to receive messages which do not exist in the buffers will be a deadlock state. The product A is said to be *deadlock-free* if no state reachable from its initial state is a deadlock state.

Definition 3 *A language L over $\hat{\Sigma}$ is said to be* safely realizable *if $L = L(\Pi A_i)$ for some $\langle A_i | i \in [n] \rangle$ such that ΠA_i is deadlock-free. A set of MSCs $\mathcal{M}$ is said to be safely realizable if $L(\mathcal{M})$ is safely realizable.*

Definition 4 *Given an MSC set $\mathcal{M}$, and a partial MSC, M', we say that $\mathcal{M}$* safely implies *M', and denote this by*

$$\mathcal{M} \overset{S}{\vdash} M'$$

if for any deadlock-free product $\Pi_i A_i$ such that $L(\mathcal{M}) \subseteq L(\Pi_i A_i)$ there is some completion M'' of M' such that $L(M'') \subseteq L(\Pi_i A_i)$.

To see that weak realizability does not guarantee safe realizability, consider the MSCs in Figure 5. They depict communication among two processes, P_1 and P_2, who attempt to agree on a value (a or b) by sending each other messages with their preferences. In MSC_3, both processes send each other the value a, while in MSC_4, both processes send each other the value b, and thus, they agree in both cases. From these two, we should be able to infer a partial scenario, depicted in MSC_5, in which the two processes start by sending each other conflicting values, and the scenario is then completed in some way. However, the language $L(\{\text{MSC}_3 , \text{MSC}_4\})$ generated by MSC_3 and MSC_4, contains no such scenarios although it is closed under weak implication, and thus, is weakly realizable. Concurrent automata capturing these two MSCs are shown in Figure 6. Each automaton has a choice to send either a or b. In the product, what happens if the two automata make conflicting choices? Then, the global state would have A_1 in, say, state $u1$, and A_2 in state $v2$, and this global state has no outgoing transitions, resulting in deadlock. We would like to rule out such deadlocks in our

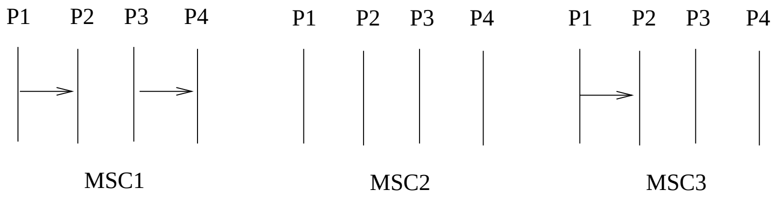

Figure 7: Safe realizability is not entirely captured by CC3

implementations. We need a stronger version of implication closure.

We will give two closure conditions which, taken together, will characterize safe realizability in the same way that condition CC2 characterized weak realizability.

For a language L, let $\mathit{pref}(L)$ denote the set of all prefixes of the words in L.

Closure Condition CC3. A language L over the alphabet $\hat{\Sigma}$ is said to satisfy *closure condition CC3* iff for all well-formed words w: if for each process i there is a word $v^i \in \mathit{pref}(L)$ such that $w|_i = v^i|_i$, then w is in $\mathit{pref}(L)$.[4]

An equivalent definition, which turns out to be easier to check algorithmically, is the following:

Closure Condition CC3'. A language L over the alphabet $\hat{\Sigma}$ is said to satisfy *closure condition CC3'* iff for all $w, v \in \mathit{pref}(L)$ and all processes i: if $w|_i = v|_i$, and $wx \in \mathit{pref}(L)$ and vx is well-formed for some $x \in \hat{\Sigma}_i$, then vx is also in $\mathit{pref}(L)$.

Proposition 7 *L satisfies CC3 iff it satisfies CC3'.*

[4]Note that this corresponds to the CC2 closure condition on $\mathit{pref}(L)$, without the requirement of completeness on w.

Proof. Suppose L satisfies CC3, and suppose $v, w \in \mathit{pref}(L)$ such that $v|_i = w|_i$, and such that $wx \in \mathit{pref}(L)$. Now, vx has the property that $vx|_j = v|_j$ for $j \neq i$, and $vx|_i = wx|_i$. Thus, if vx is well-formed, then by CC3, $vx \in \mathit{pref}(L)$, establishing this direction of the claim.

Suppose L satisfies CC3$'$. Consider a w which is well-formed and such that for all i, $w|_i = v^i|_i$, where $v^i \in \mathit{pref}(L)$. We will, by induction on the length of w show that $w \in \mathit{pref}(L)$. If $|w| = 1$, then from CC3$'$ we trivially get, using the fact that the empty string is always in $\mathit{pref}(L)$, that $w \in \mathit{pref}(L)$.

Suppose $w = w'x$, where x occurs on some process i. Thus $v^i|_i = (w'|_i)x$. By induction, it is clear that $w' \in \mathit{pref}(L)$. Now, w' and v^i satisfy the condition of CC3$'$, thus $w'x = w$ is also in $\mathit{pref}(L)$.

The basic intuition behind the above is the following. Consider two possible (partial) scenarios w and v such that $w|_i = v|_i$. Then, from the point of view of process i, there is no way to distinguish between the two scenarios. Now, if the next event executed by process i in the continuation of the global scenario w is x, then x must be a possible continuation in the context v also (unless x is a receive event which has no matching send in v).

As our example shows, CC2 does not guarantee CC3. Going back to Figure 5, the event *send(1,2,a)* is a possible partial scenario (according to MSC_3), and the event *send(2,1,b)* is a possible partial scenario (according to MSC_4). Now, CC3 requires that the sequence *send(1,2,a), send(2,1,b)* be a possible partial scenario (since its individual projections are consistent with the input scenarios). However, neither MSC_3 nor MSC_4

corresponds to this case, implying the existence of an additional scenario which completes these two events. Hence, although $\{\mathrm{MSC}_3\ , \mathrm{MSC}_4\}$ has the weak CC2 closure property, it does not have the safe CC3 closure property. Notice that there is no *unique* minimal safe realization which completes MSC_5. The implied partial scenarios can be completed in many incompatible ways, each of which would eliminate the possibility of deadlock.

Safe realizability is not entirely captured by closure condition CC3 [22]. This is illustrated by the example scenarios in Figure 7. If we just consider the two MSCs MSC1 and MSC2, the set satisfies CC3. This is because the prefixes of the two MSCs are all prefixes of MSC1. However, the two MSCs safely imply the scenario MSC3 (and also a symmetric scenario which contains only the message from P3 to P4). The second closure condition we will need to capture safe realizability is in fact a restriction of condition CC2, which is easier to check, and which allows one only to imply new well-formed complete words that are themselves prefixes of words already in L:

Closure Condition CC2'. A language L over the alphabet $\hat{\Sigma}$ satisfies *closure condition CC2'* iff for all well-formed and complete words w over $\hat{\Sigma}$ such that $w \in$ *pref*(L): if for all processes i there exists a word v^i in L such that $w|_i = v^i|_i$, then w is in L.

The correspondence between safe realizability and conditions CC3 and CC2' is established by the next theorem.

Theorem 8 *A language L over the alphabet $\hat{\Sigma}$ is safely realizable iff L contains only well-formed and complete words and satisfies both CC3 and CC2'.*

Proof. Suppose we have a deadlock-free product $\Pi_i A_i$, with $L = L(\Pi_i A_i)$. Using the fact that every partial execution of the automata can be extended to a complete execution, we can show, in a way similar to the proof of Theorem 5, that every well-formed word w which has its projections in $\mathit{pref}(L)$ is itself in $\mathit{pref}(L)$, thus condition CC3 is satisfied. The fact that condition CC2' is also satisfied follows directly from Theorem 5, because safe realizability implies weak realizability, and condition CC2 certainly implies CC2', which is just a restriction of it.

Suppose L satisfies CC3 and CC2', and contains only well-formed and complete words. Consider deterministic automata A_i which accept the sets L_i of projections of L onto process i, and assume all states in A_i reach an accepting state (other states can be removed without changing the language accepted by A_i). We will show that $\Pi_i A_i$ is deadlock-free, and that $L = L(\Pi_i A_i)$. During the execution of $\Pi_i A_i$, at all times the word w seen so far has the property that its projection on each process i belongs to $\mathit{pref}(L_i)$. By CC3, w is in $\mathit{pref}(L)$. Since the A_i's are deterministic, the product automaton must be able to reach an accepting state after processing w, simply by processing the word $w' \in L$ such that w is a prefix of w'. Hence $\Pi_i A_i$ is deadlock free. We need only show that $w \in L$ iff $w \in L(\Pi_i A_i)$. The forward direction follows easily from the construction of the A_i's. Suppose $w \in L(\Pi_i A_i)$, then for each process i, $w|_i \in L_i$. Since we have already established that $w \in \mathit{pref}(L)$, we conclude, by CC2', that $w \in L$.

Corollary 9 *An MSC family $\mathcal{M}$ is safely realizable iff $L(\mathcal{M})$ satisfies CC3 and CC2'.*

7 Algorithms for Inference, Realizability, and Synthesis

Now that we have the necessary and sufficient conditions, we are ready to tackle the algorithmic questions raised in the introduction. Namely, given a finite set $\mathcal{M}$ of MSCs, we want to determine automatically if $\mathcal{M}$ is realizable as the set of possible executions of concurrent state machines, and if so we would like to synthesize such a realization. If not, we want to find counterexamples, namely missing implied (partial) MSCs. Of course, we want any realization to be deadlock-free, and thus we prefer safe realizations.

7.1 An Algorithm for Safe Realizability

Given MSCs $\mathcal{M} = \{M_1 \ldots, M_k\}$, where each MSC is a scenario over n processes $P_1, \ldots, P_n$, we now describe an algorithm which, if $\mathcal{M}$ is safely realizable returns "YES", and if not it returns a counterexample, namely, an implied (possibly partial) MSC, M', which must exist as a (possibly partial) execution of some MSC, but does not in $\mathcal{M}$. By Corollary 9, it suffices to check that $L(\mathcal{M})$ satisfies CC3 and CC2'. We first describe how to check closure condition CC3, followed by an algorithm for checking closure condition CC2'. Combining these two algorithms, we obtain our algorithm for checking safe-realizability, and if not inferring implied but unspecified (partial) MSCs:

1. Check CC3, and if the answer is no, then output an implied MSC and halt.
2. Otherwise, check CC2'. If the answer is no, output an implied but unspecified MSC. If yes, then halt and output "Yes, $\mathcal{M}$ is safely realizable."

These results are summarized by the following theorem:

Theorem 10 *Given a set $\mathcal{M}$ of MSCs, safe realizability of $\mathcal{M}$ can be checked in time $O(k^2n + rn)$, where n is the number of processes, k is the number of MSCs, and r is the number of events in the input MSCs.*

7.1.1 Checking closure condition CC3

By Proposition 2, MSCs are determined by any of their linearizations, and thus by their projections onto individual processes. We can therefore assume that $\mathcal{M}$ is presented to us as a two dimensional table of strings, with $M[l, i]$ giving the projection $M_l|_i$ of the MSC M_l of $\mathcal{M}$ on process i, including an end delimiter. We use $\|M[l, i]\|$ denote the length of the string, and $M[l, i, d]$ to denote the dth letter of the string.

A straightforward algorithm to check CC3 would have exponential complexity. We show how to check CC3 in polynomial time, via its equivalence to CC3'. Figure 8 gives a simple version of our polynomial time algorithm for checking CC3.

Correctness

The correctness of the algorithm is based on Proposition 7. Condition CC3' is violated if and only if the set $\mathcal{M}$ contains two MSCs M_s and M_t, the MSC M_s has a (well-formed) prefix N_s, and for some process P_i the following property holds. The prefix N_s of M_s agrees with M_t on process P_i (i.e. $N_s|_i$ is a prefix of $M_t|_i$), the next event on process P_i of M_s (respectively M_t) after the prefix is x (respectively, x'), the event x' is *eligible* to be appended to N_s (in place of x) in the sense that it would yield a well-formed partial MSC N_s' - that is, x' is either a send event or it is a receive event

```
proc Condition_CC3(M) ≡
  foreach (s, t, i) ∈ [k] × [k] × [n] do
    T[s, t, i] := min {c | (M[s, i, c] ≠ M[t, i, c])}
  od;
  /* T[s, t, i] gives the first position on */
  /* process i where M_s and M_t differ */
  /* If M_s |_i = M_t |_i then T[s, t, i] = ⊥ */
  Let ≤^s be the partial order of events in M_s.
  foreach s ∈ [k] and event x in M_s do
    foreach process j ∈ [n] do
      U[s, x, j] :=
        { ||M[s, j]|| + 1              if ∀c x ≰^s M[s, j, c]
        {
        { min {c | (x ≤^s M[s, j, c])}  otherwise
    od;
  od;
  /* U[s, x, *] gives the events of M_s dependent on x */
  foreach (s, t, j) ∈ [k] × [k] × [n]such thatT[s, t, j] ≠ ⊥ do
    c := T[s, t, j];
    x := M[s, j, c]; x' := M[t, j, c];
    /* Determine if x' is eligible to replace x. */
    /* If x' is a send event, it is always eligible. */
    /* If x' = receive(i, j, a) then x' is eligible */
    /* iff M[s, i][1 . . . U[s, x, i] − 1] contains more */
    /* send(i, j, a)'s than M[s, j][1 . . . U[s, x, j] − 1] */
    /* contains receive(i, j, a)'s. */
    if x' is eligible to replace x then
      /* Find if some M_p realizes this replacement */
      if ∃ p ∈ [k] such that
        M[p, j, c] = x' and
        ∀j' ∈ [n] U[s, x, j'] ≤ T[s, p, j']
        then() /* This eligible replacement exists */
        else
         "M DOES NOT Satisfy CC3"
         Missing Implied partial MSC given by ∀j'
         M[s, j'][1 . . . U[s, x, j'] − 1] and M[s, j][c] := x'
         return;
      fi;
    fi;
  od;
  "YES. M DOES Satisfy CC3"
```

Figure 8: Algorithm for Checking condition CC3

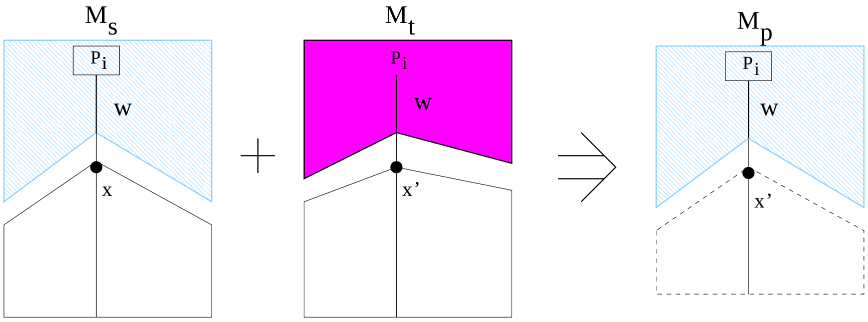

Figure 9: Inference algorithm illustration

that can be matched to an unmatched send event of the prefix N_s - but the resulting partial MSC N'_s is not a prefix of any MSC M_p in the given set $\mathcal{M}$. Stated in the above form, the main source of complexity is that after fixing MSCs M_s and M_t of $\mathcal{M}$, and the process P_i, the number of prefixes of M_s can in general be exponential in the number of processes. The choice of M_s, M_t and process P_i fixes the prefix N_s in process P_i: it must include all events until the first disagreement between M_s and M_t, i.e. the first step in which M_s performs an event x and M_t performs a different event x'. Obviously, in the other processes we can not include in the prefix N_s any events that depend on x (otherwise it will not be well-formed), but this still leaves much freedom. The key observation underlying the algorithm is that we only need to check the condition for the largest possible prefix of M_s, namely, the prefix that includes in the other processes all events that do not depend on the event x of P_i. This situation is depicted in Figure 9, where the shaded regions in M_s and M_t denote those events that do not depend on x and x', respectively. The reason it suffices to check against the largest prefixes on each process is that "possibility" of an event on process i can only become true and cannot become false as the prefix on process $j \neq i$ increases, while the prefix on i stays fixed. Thus, by considering only maximal prefixes, we are considering the maximal set of events x' eligible to take the place of x.

Time Complexity

The algorithm to check condition CC3 can be implemented with suitable data structures to run in time $O(k^2 \cdot n + r \cdot n)$, where k is the number of MSCs, n is the number of processes, and r is the total number of events in all the MSCs. The algorithm computes first the table T. The table has size $O(k^2 \cdot n)$, and can be computed with essentially the same time complexity, as follows. For each process P_i, $i \in [n]$, we can construct a trie S_i for the projections on i of all the MSCs of $\mathcal{M}$. That is, S_i is a rooted tree whose edges are labeled with the event symbols and such that the root-to-leaf paths spell the strings $M[l,i]$. For each node v of the trie we record the depth and we attach a list $list(v)$ of the indexes of all the MSCs M_l which go through that node, i.e., $l \in list(v)$ iff the string that labels the path from the root to the node is a prefix of $M[l,i]$. The trie and the node lists can be constructed in a standard way incrementally, processing the strings one by one, in time proportional to the sum of the lengths of the strings.[5] Thus, the time to construct all the tries S_i is $O(r)$. As usual, we can compress nodes that have only one child (i.e. form compressed tries). The table T can be constructed now easily from the tries and the associated node lists: for each pair of siblings v, w of S_i, for every $s \in list(v)$ and $t \in list(w)$, set $T[s,t,i]$ equal to 1 plus the depth of the (common) parent of v and w. Likewise, for each parent node v of S_i, with children $w_1, \ldots, w_d$, for each $s \in list(v) \setminus \cup_{j=1}^{d} list(w_j)$: for each $t \in \cup_j list(w_j)$, set $T[s,t,i]$ equal to 1 plus the

[5]Provided we can index on the letters of the alphabet Σ, otherwise we have to multiply by a logarithmic factor.

depth of the parent v, and for each $t' \in list(v) \setminus \cup_{j=1}^{d} list(w_j)$, set $T[s, t', i] = \perp$. For a leaf node v, and for all $s, t \in list(v)$ set $T[s, t, i] = \perp$.

The algorithm computes next $U[s, x, j]$ for all MSCs M_s, events x of M_s, and processes $j \in [n]$. There are rn such U entries and they can be computed in time linear in their number: For each MSC M_s, order topologically the nodes (events) and process them bottom up. An event x has at most two immediate successors: the next event (if there is one) on the same process, and if x is a send event, the matching receive event. If the process of event x is i, i.e., $x = M[s, i, c]$ for some c, then $U[s, x, i] = c$. For any other process $j \neq i$, the entry $U[s, x, j]$ is equal to the minimum of the U entries corresponding to the immediate successors of x.

The algorithm then considers every pair of MSCs M_s, M_t and every process j on which the two MSCs differ, and determines if they yield a counterexample to the condition CC3'. Namely, if the first disagreement between the j-projections of the MSCs is in the cth step, where M_s has x whereas M_t has x', the algorithm determines (i) if x' is eligible to be appended to the maximal prefix $N_s(x)$ of M_s that does not contain x, and (ii) if the resulting partial MSC is a prefix of some other MSC M_p of the given set. Note that the maximal prefix $N_s(x)$ is defined by the entries $U[s, x, j]$. We describe now how to implement (i) and (ii) within the stated bounds.

For each MSC M_s and process i, we find the events that are eligible to replace the events of M_s on process i, using a pass over the MSC from the top down as follows. We process the events of M_s on process i one by one. As we process the events x on i, we form the corresponding prefixes $N_s(x)$, by incrementally extending the other processes j adding new steps up to $U[s, x, j]$. Every step of M_s is considered at most once, when it is added for the first time in a prefix. We maintain an array A indexed by the pairs (j, a), $j \in [n], a \in \Sigma$, for which process j sends message a to i. The array is initialized to all 0. After processing event x of process i, the entry $A[j, a]$ gives the number of unmatched $send(j, i, a)$ messages in the maximal prefix $N_s(x)$ corresponding to x; that is, $A[j, a]$ is the difference between the number of $send(j, i, a)$ events in the first $U[s, x, j] - 1$ steps of process j and the number of $receive(j, i, a)$ events in process i before x. Clearly, the array A can be updated from one step of process i to the next step in time proportional to the number of nodes of the other processes that are added to the prefix. Thus, the total time spent in updating A over all events of process i is $O(|M_s|)$. When processing event x on process i, after the update of A, we first determine which other events x' need to be considered as potential replacements of x, and then determine which of them are eligible. Note that x' needs to be considered if there is another MSC M_t which agrees with M_s on process i up to this step, and which has x' at this step. The set of possible x' is given by the trie S_i: if the event x corresponds to the edge (u, v) of S_i, then the set of the x' is precisely the set of labels of the other edges connecting u to its other children. For each such event x', if x' is a send event then it is of course eligible, and if it is a receive event, say $receive(j, i, a)$, then it is eligible if $A[j, a] > 0$. The time spent to find the eligible replacement events x' for all events x of M_s on process i is certainly no more than $O(k + |M_s|)$, and thus the total time over all processes and all MSCs is no more than $O(k^2 n + rn)$. It remains to determine for each eligible x' whether there is an MSC M_p that contains the partial

MSC formed by appending x' to $N_s(x)$. A straightforward test takes too much time. We'll describe below how to perform this check within our time bound.

Let M_s, M_t be two MSCs and consider for each process i the first event of M_s which differs from M_t, i.e., the $T[s,t,i]$-th event. If one of these n events, say the one on process i, precedes all the other events in the partial order of M_s, then we say that M_t *covers* M_s on process i. Note that there can be at most one such process for which one MSC covers another. Note furthermore that M_t covers M_s on process i if the prefix $N_s(x)$ of M_s is also a prefix of M_t, where x is the earliest step of M_s on process i on which it disagrees with M_t. We determine for each pair of MSCs whether one covers the other, and if so, we record the corresponding process. This can be done in time $O(k^2n + r)$ as follows. For each MSC M_s, fix an arbitrary linearization and label each node of M_s with its order in the linearization. This takes linear time. Consider another MSC M_t. The entries $T[s,t,i], i \in [n]$ give the earliest steps on each process in which M_s disagrees with M_t. Look up their labels in M_s and determine the one that has the smallest label, say the step on process i. Note that if one of these disagreement steps precedes all the others in M_s (i.e., if M_t covers M_s), then it must be the step with smallest label, thus the step on process i. Let x be that step. By the definition of U, step x precedes the steps on the other processes iff $U[s,x,j] < T[s,t,j]$ for all processes j. Thus, we can determine if M_t covers M_s and record the corresponding process i in time $O(n)$.

Consider now an MSC M_s, event x of M_s on process i, and an eligible replacement x' for x. Consider the set of MSCs that agree with M_s up to this step at which point they perform x' instead of x; the set is available in the information recorded at the trie S_i: if x corresponds to the edge (u, v) of S_i and x' to the edge (u, w), then the set of MSCs is given by the list attached to the node w. It follows from the definitions that there is an MSC M_p that contains the partial MSC formed by appending x' to $N_s(x)$ if and only if the list of node w contains an MSC M_p that covers M_s on process i. Thus, we can determine if the eligible replacement x' leads to a violation of safe realizability in time proportional to the number of MSCs in the list of w. Summing over all events of M_s on process i, this amounts to time $O(k)$, and hence, summing over all the processes and all the MSCs, the contribution of this part is $O(k^2 n)$.

Therefore, the algorithm to check CC3 can be implemented as explained above to run in time $O(k^2 \cdot n + r \cdot n)$.

As given, the algorithm stops as soon as it finds a single missing partial MSC. One can easily modify the algorithm in several ways to find more missing scenarios if present. One such modification would derive not only one implied partial MSC, but a *complete* set of implied partial MSCs, in that for every MSC M implied by the given set, there would be a partial MSC M' present in the derived set such that M is a completion of M'. This set contains at most $k^2 \cdot n$ partial MSCs. This upper bound holds because, in the main loop, we need only check for each pair of MSCs, and for each process, whether the first event where the two MSCs differ on that process introduces a new implied MSC.

A second way in which the algorithm can be modified is to substitute not just the first eligible event, x' of M_t for x in M_s, but to use the *longest eligible subsequence* w'

beginning at x' on process j in M_t. That is, we can extend the prefix $N_s(x)$ of M_s by appending on process j the event x' and all subsequent events of M_t which are either send events or receive events that can be matched with unmatched send events, thus yielding a well-formed partial MSC that is also safely implied by the given set. This will fill out the partial MSCs, completing them as much as possible.

Finally, one can repeatedly apply the algorithm, inferring more and more partial MSCs, until the set of implied partial MSCs closes, i.e., no more partial MSCs can be implied. Of course, doing so could entail an exponentially large set of implied MSCs.

Example

Consider the two MSCs of Figure 1 as input to the algorithm, where we assume the implication algorithm is modified according to the second suggestion above. To see how MSC_{bad} is derived by the algorithm, consider the first events on UR where MSC_1 and MSC_2 differ. In MSC_1 the first event is $x = receive(P_1, UR, inc)$, whereas in MSC_2 the first event is $x' = receive(P_2, UR, double)$. Since in MSC_1 no events, other than those on UR, depend on the first event x, the corresponding prefix $N_1(x)$ consists of all events of MSC_1 on the other processes, and no events on UR. The event $x' = receive(P_2, UR, double)$ of MSC_2 on UR is eligible to replace x (since the corresponding send is included in the prefix $N_1(x)$), and so is the second event $receive(P_1, UR, inc)$ of MSC_2. The result of this replacement is precisely MSC_{bad}, the inferred MSC in Figure 2.

7.1.2 Checking closure condition CC2'

We now outline an efficient algorithm for checking that the set of MSCs $\mathcal{M}$ satisfies CC2'. Each piece of the algorithm makes straightforward use of standard graph algorithms. We will describe these only at a high level and not specify them in detailed pseudo-code.

1. For each MSC M_s in $\mathcal{M}$, and for each process P_i, compute $v_i = M_s|_i$. Compute the set $V_s^i = \mathit{pref}(v_i) \cap \{M|_i \mid M \in \mathcal{M}\}$, i.e., prefixes of v_i that also constitute the entire projection of some other MSC in $\mathcal{M}$ on process P_i. We can totally order the set V_s^i as $\{v_i^1, \ldots, v_i^{k_i}\}$, such that for all j, v_i^j is a prefix of v_i^{j+1}. Define the (ordered) multi-set of strings $W_s^i = \{w_i^1, \ldots, w_i^{k_i}\}$ to be the "segments" of v_i such that $w_i^1 w_i^2 \ldots w_i^j = v_i^j$.

2. Build a directed graph $G_s = (V_s, E_s)$, with nodes $V_s = \cup_{i=1}^n \{t_i^1, \ldots, t_i^{k_i}\}$, such that there is a node t_i^j associated with each segment w_i^j, and the set E_s of edges contains (t_i^j, t_i^{j+1}) for $i \in [n], j \in \{1, \ldots, k_i - 1\}$ and $(t_i^j, t_{i'}^{j'})$ if there exists a message sent from segment w_i^j to $w_{i'}^{j'}$ <u>or</u> from $w_{i'}^{j'}$ to w_i^j.

3. Compute the strongly connected components of G_s and also compute the underlying DAG $G'_s = (V'_s, E'_s)$, whose nodes V'_s are the SCCs, and whose edges $(C, C') \in E'_s$ exist from one SCC to another iff there is an edge in G_s from a node in C to a node in C'.

4. For each sink SCC C of G'_s, remove from MSC M_s all messages in all segments associated with nodes in C. Call this new MSC M_s^C. (Note that by construction

M_s^C is indeed a valid, non-partial, MSC). Check that M_s^C is in $\mathcal{M}$. If not, halt, output "No, does not satisfy condition CC2"', and output M_s^C as an implied but unspecified MSC.

5. If for all M_s in $\mathcal{M}$ all such MSCs M_s^C are found to be in $\mathcal{M}$, output "Yes, Condition CC2' is satisfied."

Correctness

Recall that we say MSC M is a prefix of another MSC M' iff for all processes P_i, the projection $M|_i$ of M onto P_i is a prefix of the projection $M'|_i$ of M' onto P_i.

According to CC2', any MSC, M, which satisfies the following two conditions must be in $\mathcal{M}$:

1. M is a prefix of some MSC $M' \in \mathcal{M}$.

2. For every process P_i, there exists an MSC $M_i \in \mathcal{M}$, such that $M|_i = M_i|_i$.

Let us call an MSC that satisfies conditions (1) and (2) a *candidate* MSC.

Thus, in order to check CC2', we need to check that for each $M_s \in \mathcal{M}$, and for all candidate MSCs M that are prefixes of M_s, M is itself in $\mathcal{M}$.

Note that since M must satisfy condition (2), it must be the case that for each process P_i, $M|_i = v_i^j$, for some $j \in \{1, \ldots, k_i\}$, were v_i^j's were defined in step 1 of our algorithm. Thus, we have no loss of generality by restricting our search for candidates M to those prefixes of M_s where each projection $M_s|_i$ constitutes some sequence of "segments" $w_i^1 \ldots w_i^j = v_i^j$.

On the other hand, if a send or receive event x occurring in segment w_i^d is included in candidate MSC M, and the message being sent/received has a corresponding receive/send event x' occurring in some segment $w_{i'}^{d'}$, then segment $w_{i'}^{d'}$ must also occur in M. Also observe that, obviously, if a segment w_i^d is in some candidate MSC M, then so is its immediate predecessor segment w_i^{d-1}.

Accordingly, in our graph G_s, there are edges (t_i^{d-1}, t_i^d) between nodes associated with successive segments on the same process, as well as two edges in both directions $\{(t_i^d, t_{i'}^{d'}), (t_{i'}^{d'}, t_i^d)\}$ between the nodes associated with w_i^d and $w_{i'}^{d'}$ when there is any communication between those two segments. We can think of these directed edges (u, v) as indicating that the presence of the segment (associated with) v in any candidate MSC which is a prefix of M_s necessitates the presence of the segment (associated with) u.

Thus, all segments associated with any SCC, C, of G_s must either be present or absent from a candidate MSC which is a prefix of M_s.

Let an *ideal*, I, of the DAG, $G = (V, E)$ be a subset of the nodes closed under predecessors, i.e., if $v \in I$ and $(u, v) \in E$, then $u \in I$.

By the arguments just given, there is a one-to-one correspondence between ideals I of the DAG of SCCs, G'_s, and candidate MSCs M_I which are prefixes of M_s. Given I, we construct M_I from all segments associated with all nodes t that are contained in all SCCs $C \in I$.

We need to check, for each MSC M_s, that all such candidates M_I are in our set $\mathcal{M}$. Note, however, that it suffices if we check that $M_{I'} \in \mathcal{M}$ for every *maximal* ideal I' which is a proper subset of the vertices of G'_s. This is so, by induction on the size of I', because since we check for candidate prefixes for every MSC $M_s \in \mathcal{M}$, once we check that $M_{I'} \in \mathcal{M}$, we know that (inductively) we will for every $I'' \subset I'$, also check that $M_{I''} \in \mathcal{M}$.

Note that the maximal ideals I' of G'_s are precisely the sets that eliminate exactly one sink node (SCC) C from the nodes V'_s of G'_s. These are precisely the ideals that, in our correspondence, give rise to the candidate MSCs M_s^C. Thus, since step (4) of the algorithm checks for the existence of all such candidates in $\mathcal{M}$, our algorithm determines precisely whether $\mathcal{M}$ satisfies condition CC2'.

Time Complexity

We now show that the algorithm for checking CC2' can be implemented to run in $O(k^2 \cdot n + r)$ time where, again, k is the number of MSCs in our set, n is the number of processes, and r is the total number of events in all MSCs in $\mathcal{M}$.

Recall the tries S_i, described in our complexity analysis for checking CC3 efficiently. In our computation of the S_i's, we will additionally mark each node of the trie S_i as accepting if the projection of some MSC in $\mathcal{M}$ terminates at that node. Recall, we can compute all S_i's in total time $O(r)$, and this additional marking won't alter that analysis. Using the S_i's, for each MSC $M_s \in \mathcal{M}$, we find all the "segments" w_i^j on each process P_i as follows. Take $M_s|_i$ and walk down S_i, noting the accepting nodes that are traversed along the way. These nodes determine the segments w_i^j in an obvious fashion. The time required for this walk down S_i is linear in the size of $M_s|_i$, thus the total time required to compute all segments for all MSCs in $\mathcal{M}$ is linear: $O(r)$.

Once we have found all segments w_i^j for M_s, we can build the graph G_s in time linear in $|M_s|$, the size of M_s. G_s itself obviously has size $O(|M_s|)$. We can then, using the standard DFS algorithm, compute the SCCs of G_s, and compute the DAG G'_s in time $O(|M_s|)$.

Next, for Steps 4 and 5 of the algorithm, we need to enumerate for each MSC M_s, the candidate MSCs M_s^C, for each sink SCC C of G'_s, and check that M_s^C is in $\mathcal{M}$. Let $D_s = \{C_1, \ldots, C_l\}$ be the sink SCCs of G'_s. Let P^C denote the set of processes that have a segment in SCC C. Note that $P^{C_1}, \ldots, P^{C_l}$ form disjoint sets, and in particular $l \leq n$. We will maintain a boolean array B_s of size l to mark which MSCs $M_s^{C_i}$ have been encountered in the set $\mathcal{M}$. Initially $B_s[c] := \texttt{false}$ for all $c \in [l]$. Assume we have already computed the table $T[s, t, i]$ described in the algorithm for CC3. Our analysis showed that T can be computed in time $O(k^2 n)$. For each $M_t \in \mathcal{M}$, we will check in $O(n)$ if M_t is a prefix of M_s, and if so, if $M_t = M_s^{C_j}$ for some $C_j \in D_s$, as follows. First, to check that M_t is a prefix of M_s, we check that for each process i, either $T[s, t, i] = \perp$ (i.e., the two projections are identical), or that $T[s, t, i] = \|M[t, i]\| + 1$ (i.e., the projection of $M_t|_i$ is a prefix of $M_s|_i$. This can be done in time $O(n)$. Next, if this is so, we check if $M_t = M_s^{C_j}$ for some j as follows. For any process i such that $T[s, t, i] \neq \perp$, i.e., $M_t|_i$ is a proper prefix of $M_s|_i$, let C_j be the unique SCC in D_s that contains a segment from process i. Since we already know M_t is a prefix of M_s, we can check whether $M_t = M_s^{C_j}$ by simply checking that for each process i, the length $\|M_s^{C_j}|_i\|$ is the same as $T[s, t, i] - 1$. This can again be done in $O(n)$ time. If $M_t = M_s^{C_j}$, then we mark $B_s[j] := \texttt{true}$. Once we have checked M_s against all other MSCs M_t, the array elements $B_s[c]$ should be set to $\texttt{true}$ for all c. If not, i.e., $B_s[j] = \texttt{false}$ for some j, then we know that $M_s^{C_j}$ is a candidate MSC that is a prefix of M_s but not in $\mathcal{M}$, so we say *"Not safely realizable"*, and output the implied candidate MSC, $M_s^{C_j}$. Since there are k MSCs in $\mathcal{M}$, and we have to compare each of them to all $k-1$ other MSCs in $\mathcal{M}$, and each comparison and check against D_s takes only $O(n)$ time (using the precomputed array T), the total running time for steps 4 and 5 is $O(k^2n)$. Thus, the total running time of the entire algorithm for checking $CC2'$ is $O(k^2n+r)$. Consequently, the overall time complexity of checking safe realizability is $O(k^2n+rn)$.

7.2 coNP-completeness of Weak Realizability

The less desirable realizability notion was weak realizability. There deadlocks may occur. It turns out that this weaker notion is in fact more difficult to check. CC2 gives a straightforward exponential time algorithm (in fact, a violation can be detected in NP) for checking weak realizability. The following theorem shows that we cannot expect a polynomial time solution:

Theorem 11 *Given a set of MSCs $\mathcal{M}$, determining whether $\mathcal{M}$ is weakly realizable is coNP-complete.*

Proof. To check that a set $\mathcal{M}$ is not weakly realizable in NP is easy using CC2. We guess for each process $i \in [n]$ an MSC $M^i \in \mathcal{M}$ and project it on process i, we check that the projections $M^i|_i$ are consistent, i.e., their combination is a well-formed, complete MSC, and check that this MSC is not in the given set of MSCs $\mathcal{M}$.

The proof of coNP-hardness is established by a reduction from the problem of checking whether a relational database satisfies a given join dependency condition.

For a natural number r, let $[r] = \{1, \ldots, r\}$. For a k-tuple $\mathbf{a} = (a_1, \ldots, a_k)$, and a set $S = \{s_1, \ldots, s_l\} \subseteq [k]$, where we have ordered s_i's so that $s_i < s_{i+1}$, the projection of $\mathbf{a}$ on S, denoted $\mathbf{a}|_S$ is the tuple $(a_{s_1}, \ldots, a_{s_l})$. For a relation $R \subseteq U^k$, and a subset $S \subseteq [k]$, the projections of R on S denoted $R|_S$ is the set of projections of the tuples of R, i.e.,

$$R|_S = \{\mathbf{b} \in U^{|S|} \mid \exists\, \mathbf{a} \in R \text{ such that } \mathbf{a}|_S = \mathbf{b}\}$$

The *join dependency problem* (**JDP**) is the following:

> Given a structure of the form $\langle U, R, \mathcal{S} \rangle$ where U is a finite universe, $R \subseteq U^k$ is a k-ary relation over U, and $\mathcal{S} = \{S_1, \ldots, S_k\}$, a multi-set of subsets of $[k]$ where for each $i \in [k]$ there is some j such that $i \in S_j$ (here k is not fixed, but varies with the input), determine whether it holds that, for all $\mathbf{a} \in U^k$, if for all $S \in \mathcal{S}$, $\mathbf{a}|_S \in R|_S$, then $\mathbf{a} \in R$.

We use the following fact:

Theorem 12 *[23] JDP is coNP-complete.*

Let WRP be the weak realizability problem for a set $\mathcal{M}$ of MSCs. The proof is by a P-time reduction from JDP to WRP.

Assume we are given $\Gamma = \langle U, R, \mathcal{S} \rangle$, an instance of the JDP, with $R \in U^k$. Also assume that for each $i \in [k]$, i belongs to at least two sets $S_i \in \mathcal{S}$. This can easily be assured by repeating the sets in $\mathcal{S}$ (remember that $\mathcal{S}$ is a multiset). Clearly, such repetition does not affect whether $\Gamma \in JDP$. Order the sets $\mathcal{S} = \{S_1, \ldots, S_m\}$ in some fixed total order. Likewise, order the elements in each set $S_i = \{s^i_1, \ldots, s^i_{l_i}\}$ according to the ordinary ordering of natural numbers; thus $s^i_j < s^i_{j+1}$.

We will build a set of MSCs, $\mathcal{M}_\Gamma$, over m processes, $P_1, \ldots, P_m$, one for each set S_i. $\mathcal{M}_\Gamma$ will consist of one MSC for every tuple in the relation R. All MSCs will contain

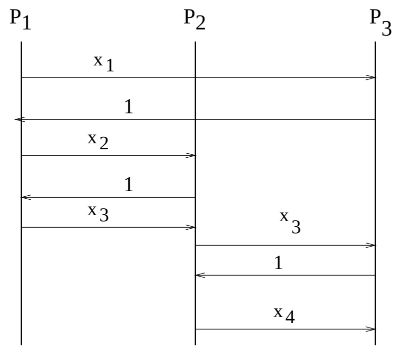

Figure 10: The example MSC template M'

exactly the same pattern of communication. The only thing that will differentiate the MSCs will be the actual messages exchanged; these will correspond to the entries of the tuples. Moreover, there will only be one linearization for each given MSCs.

All MSCs are defined based on one template $M(x_1, \ldots, x_k)$, where the x_i's correspond to the entries of a tuple. In Figure 10, we give a small example M' of the template $M(x_1, \ldots, x_k)$ in the case when $\mathcal{S} = \{S_1, S_2, S_3\}$, with $S_1 = \{1, 2, 3\}$, $S_2 = \{2, 3, 4\}$, and $S_3 = \{1, 3, 4\}$. Note that the projection of M' onto process i contains a message x_j, either sent or received, if and only if $j \in S_i$, and that the messages x_j that each process sees are totally ordered based on the natural number ordering on the indices j.

We now describe the general template $M(x_1, \ldots, x_k)$. For each $i \in [k]$, the total ordering $S_1, \ldots, S_m$ of the sets in $\mathcal{S}$ determines a total order on the sets that contain the index i. Let this subsequence be $S_{i_1}, \ldots, S_{i_{l_i}}$. Then the sequence of message exchanges in the template $M(x_1, \ldots, x_k)$ is as follows: P_{1_1} begins the MSC, sending x_1 to P_{1_2}, which after receiving x_1 sends x_1 to P_{1_3} (if it exists), and so on until $P_{1_{l_i}}$, which after receiving x_1, sends a special symbol "1" that does not occur in relation R to process P_{2_1}. Then P_{2_1} sends x_2 to P_{2_2}, and the entire process gets repeated, for each index i.

It is easy to see that in $M(x_1, \ldots, x_k)$ process P_i sends or receives (or both) x_j precisely when $j \in S_i$, and that P_i sees the x_j's in an order consistent with their total order $x_1, \ldots, x_k$.

Now we are ready to define $\mathcal{M}_\Gamma$. For a k-tuple $(a_1, \ldots, a_k) \in U^k$, let $M(a_1, \ldots, a_k)$ be the MSC obtained by substituting x_i's by a_i's in $M(x_1, \ldots, x_k)$. Define

$$\mathcal{M}_\Gamma = \{M(a_1, \ldots, a_k) \mid (a_1, \ldots, a_k) \in R\}$$

The following claim states the correctness of the reduction:
Claim 1 $\Gamma \in JDP$ *if and only if* $\mathcal{M}_\Gamma \in WRP$.

First the "if" direction. If $\Gamma \notin JDP$, then there is some tuple $\mathbf{a} = (a_1, \ldots, a_k) \in U^k$ such that $\mathbf{a}|_{S_i} \in R|_{S_i}$ for all $i \in [m]$, but $\mathbf{a}$ itself is not in R. Since $\mathbf{a}|_{S_i} \in R|_{S_i}$, there is a tuple $\mathbf{b^i}$ of R that has the same projection on S_i, i.e., such that $\mathbf{a}|_{S_i} = \mathbf{b^i}|_{S_i}$. Consider the MSC $M(\mathbf{a})$. By the construction, the projection of this MSC on process i depends only on $\mathbf{a}|_{S_i}$, hence it is equal to the projection of the MSC $M(\mathbf{b^i})$ on process i. The MSCs $M(\mathbf{b^i})$, $i \in [m]$, are all in $\mathcal{M}_\Gamma$, however the MSC $M(\mathbf{a})$ itself is not in $\mathcal{M}_\Gamma$ because $\mathbf{a}$ is not in R. Thus $\mathcal{M}_\Gamma$ is not weakly realizable.

The "only if" direction follows similarly, because for any $M \notin \mathcal{M}_\Gamma$, but such that $\mathcal{M}_\Gamma \vdash M$, we can "read off" from M tuples $\mathbf{b^i} \in R|_{S_i}$, for each i, such that there is a k-tuple $\mathbf{a} \in U^k$ such that $\mathbf{a}|_{S_i} = \mathbf{b^i}$ for each i, but $\mathbf{a} \notin R$.
That completes the proof.

7.3 Synthesis of State Machines

Given a set $\mathcal{M}$ of MSCs we would like to synthesize automata A_i, such that $L(\Pi A_i)$ contains $L(\mathcal{M})$, and as little else as possible. In particular, if $\mathcal{M}$ is weakly realizable we would like to synthesize automata such that $L(\Pi_i A_i) = L(\mathcal{M})$ (and, when safely realizable, such that $\Pi_i A_i$ is deadlock-free).

Given the proof of Theorem 5, it is straightforward to synthesize the A_i's. The algorithm we provide is not new, and follows an approach similar to other synthesis algorithms in the literature. What is new are the properties these synthesized automata have in our concurrent context. Let the string language of $\mathcal{M}$ corresponding to process i be given by $L_i = \{M|_i \mid M \in \mathcal{M}\}$. We let A_i denote an automaton whose states Q_i are given by the set of prefixes, $\mathit{pref}(L_i)$, in L_i, and whose transitions are $\delta(q_w, x, q_{wx})$, where $x \in \hat{\Sigma}$, and $w, wx \in \mathit{pref}(L_i)$. Letting the accepting states be q_w for $w \in L_i$, A_i describes a tree whose accepting paths give precisely L_i. We can minimize the A_i's, which collapses leaves and possibly other states, to obtain smaller automata. Note that the A_i's can be constructed in time linear in $\mathcal{M}$. Letting $A_{\mathcal{M}} = \Pi A_i$, we claim the following:

Theorem 13 *$L(A_{\mathcal{M}})$ is the smallest product language containing $L(\mathcal{M})$. If $L(\mathcal{M})$ is weakly realizable, then $L(\mathcal{M}) = L(A_{\mathcal{M}})$, and, if moreover $L(\mathcal{M})$ is safely realizable, then $A_{\mathcal{M}} = \Pi_i A_i$ is deadlock-free.*

8 Alternative Architectures

Much of what we have discussed can be rephrased based on different concurrent architectures, but rather than delve into the peculiarities of each architecture, we can abstract away from these considerations and assume we are given a very general "enabled" relation

$$\mathit{enabled} : (\hat{\Sigma}^* \times \hat{\Sigma}) \mapsto \{true, false\}$$

which tells us, for a given prefix of an execution, what the possible next events in the alphabet are. Architectural considerations like the queuing discipline and the synchrony of the processes clearly influence the *enabled* function. Besides architectural considerations, there are other constraints on $enabled(w, x)$. For example, for $enabled(w, receive(i, j, a))$ to hold, it must be that there are more $send(i, j, a)$'s in w than $receive(i, j, a)$'s. We state the following axioms which are assumed to hold for $enabled(w, x)$, regardless of the architecture.

1. If $enabled(w, receive(i, j, a))$ then
$$\#(w, send(i, j, a)) - \#(w, receive(i, j, a)) > 0$$
2. If $enabled(w, x)$ and $enabled(w, y)$ and x and y occur on different processes, then $enabled(wx, y)$.
3. If $enabled(w, x)$ and $w'|_i = w|_i$ for all i, then $enabled(w', x)$.

Justification for these axioms is as follows: the first axiom is obvious. The second axiom says that an event occurring on one process cannot disable an event from occurring on another process, intuitively because unless the two processes communicate they cannot effect each others behavior. The third axiom is another version of the second. It says that the ability of an event to occur on a given process depends only on the sequence of events that have occurred on each process so far, and not their particular interleaving with events of other processes.

These three basic axioms can be augmented with other axioms to reflect the properties of specific architectures. Consider the following two specific instances:

FIFO queues. When queues are required to be FIFO, for every pair i, j of processes, the sequence in which process i sends messages to process j must coincide with the sequence in which j receives messages from process i. Then, in addition to the basic axioms, we require that $\mathit{enabled}(w, \mathit{receive}(i, j, a))$ only holds if the first $\mathit{send}(i, j, \star)$ in w for which there is no matching $\mathit{receive}(i, j, \star)$ is indeed $\mathit{send}(i, j, a)$.

Synchronous communication. When the message exchanges are synchronous, a sending process cannot continue until the message is received (and implicitly acknowledged). To model this, for any event x on process i, we require that $\mathit{enabled}(w, x)$ holds only when all sends from process i have a matching receive in w.

We reformulate well-formedness, completeness, and the different closures conditions, in this more general setting:

- **Well-Formedness**: for every prefix $w'x$ of $w \in L$, $\mathit{enabled}(w', x)$ holds.
- **Completeness:** The definition of completeness remains exactly the same.
- Conditions CC2, CC2', and CC3, also remain the same.

For each architecture, Theorems 5 and 8 remain true under these modified conditions. Rather than prove the general theorems, we examine the specific architecture with FIFO queues, to hopefully provide better intuition.

Suppose we are given a set of *FIFO MSCs* $\mathcal{M}$ which are weakly realizable in the modified sense above. By FIFO MSCs we mean MSCs where the message arrows between a pair of processes do not cross: there does not exist two send events e_1 and e_2

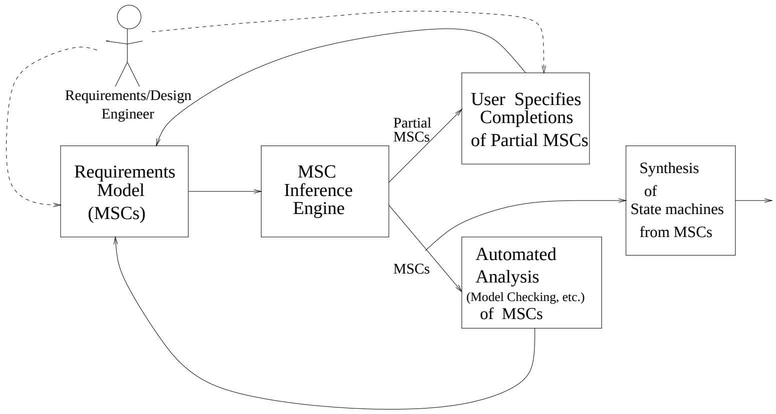

Figure 11: Using the MSC inference framework

between identical processes (i.e., with $l(e_1) = send(i, j, x)$ and $l(e_2) = send(i, j, y)$) such that $e_1 < e_2$ and $f(e_2) < f(e_1)$. All linearizations of such MSCs will be well-formed and complete in the above sense. We wish to show that $\mathcal{M}$ is weakly realizable (via a FIFO architecture) iff $L(\mathcal{M})$ satisfies CC2. Suppose there are $\langle A_i \mid i \in [n]\rangle$ which give a realization A of $\mathcal{M}$, i.e., $L(A) = L(\mathcal{M})$. Let w be a well-formed and complete word such that $w|_i = v^i|_i$, with $v^i \in L(\mathcal{M}) = L(A)$. Then, as in the proof of Theorem 5, we can extract from an accepting run of A on v^i, a run r_i of A_i on $v^i|_i = w|_i$. Thus, since w fulfills the FIFO requirement (which is entailed in the requirement that w be well-formed), we can combine the runs r_i to a run of A on w. In the same way, one can modify the argument for Theorem 5 to prove the other direction of the claim. Theorem 8 can be modified similarly to this setting.

To modify the inference and safe realizability algorithm of Figure 8 for the setting with a FIFO architecture, we need simply to revise the interpretation of when an event x' is *eligible* to replace an event x to appropriately capture the FIFO settings. Recall that the key property that allows us to test safe realizability in polynomial time, is that for every choice of MSCs M_s, M_t and process i, where x, x' are the earliest events of process i in which M_s and M_t disagree, we need to consider only one prefix $N_s(x)$ (the maximal one) in examining whether replacing x by x' yield a violation of the closure condition CC3. The fundamental reason is that the following monotonicity properties holds: if x' can be 'legally' appended to a smaller prefix of M_s (that of course does not include x and its descendants) in the sense that it yields a well-formed partial MSC, then x' can be also legally appended to the maximal prefix $N_s(x)$. This property combined with the obvious fact that if there is an MSC M_p of the given set that contains the larger prefix then there is clearly one that contain also the smaller prefix, permits us to consider only the maximal prefix. The three basic axioms of the predicate *enabled* imply that the relevant monotonicity property holds in general: If a string wx' is well-formed, where x is an event on process i, and if the string ww' is also well-formed where w' consists of steps on other processes $j \neq i$, then the string $ww'x'$ is also well-formed. The reason is that, by the axioms, execution of an event on one process cannot disable events on another process. Therefore, the basic structure of the safe realizability algorithm is sound in general. The only difference is that the specific enabled predicate in each case determines whether event x' is eligible to replace x. The efficiency of the algorithm will depend on how efficiently eligibility can be determined. For FIFO architectures or for synchronous communication, eligibility can be detected easily, but for arbitrary architectures it depends on the complexity of the enabled predicate.

9 Conclusions

We have presented schemes for detecting scenarios that are implied but unspecified. The scenarios inferred by our algorithms can provide potentially useful feedback to the designer, as unexpected interactions may be discovered.

We have given a precise formulation of the notion of deadlock-free implementation and have provided an algorithm to detect safe realizability or else infer missing scenarios. We have shown that our state machines synthesized from MSCs are deadlock-free if the MSCs are safely realizable. Our algorithm for safe realizability is efficient, and thus, the conventional "state-space explosion" bottleneck for the algorithmic analysis of communicating state machines is avoided. Since scenario-based specifications are typically meant to be only a partial description of the system, the inferred MSCs may or may not be indicative of a bug, but the implied partial scenarios need to be resolved by the designer one way or the other, and they serve to provide more information to the engineer about their design.

A way in which we envision our framework can be used is depicted in Figure 11. A user specifying MSCs in a requirements model can feed the MSCs to the inference algorithm. If implied partial MSCs are discovered, the user will be prompted to complete the MSCs. New complete MSCs are added to the requirements model. Meanwhile, complete MSCs in the requirements model can be fed to to other analysis algorithms, such as a model checker ([7]), and finally, once the requirements model is in satisfactory shape, the state machine models for the communicating processes can be synthesized from the MSCs.

The algorithms of this paper can also be used for abstraction and/or verification of programs. A single execution of a distributed program can be viewed as an MSC. Thus, instead of obtaining the input set of MSCs as requirements from the designer, it can be derived by executing the implemented program a certain number of times. The automata synthesized by our algorithms can be considered as an (under-)approximation of the source program, and can be subjected to analyses such as model checking. This approach can be useful when the source program is too complex to be analyzed, or is available only as a black-box (e.g. as an executable).

We have introduced a framework for addressing implication and realizability questions for the most basic form of MSCs. It would be desirable to build on this work, extending it to address these questions for more expressive MSC notations, such as MSCs annotated with state information (e.g., [11]), and high-level MSCs (as in, e.g., uBET [8]).

Acknowledgements.

Markus Lohrey, in [22], pointed out a bug in the characterization of safe realizability given in our conference version of this paper ([24]). Our modified characterization here, and our associated modification of the algorithm, correct this bug. Thanks to Markus for sending us a copy of his manuscript.

References

[1] "ITU-T recommendation Z.120. Message Sequence Charts (MSC'96)," May 1996, ITU Telecommunication Standardization Sector.

[2] E. Rudolph, P. Graubmann, and J. Gabowski, "Tutorial on message sequence charts," in *Computer Networks and ISDN Systems – SDL and MSC*, vol. 28. 1996.

[3] J. Rumbaugh, I. Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[4] R. Alur, G.J. Holzmann, and D. Peled, "An analyzer for message sequence charts," *Software Concepts and Tools*, vol. 17, no. 2, pp. 70–77, 1996.

[5] A. Muscholl, D. Peled, and Z. Su, "Deciding properties of message sequence charts," in *Foundations of Software Sci. and Comp. Structures*, 1998.

[6] H. Ben-Abdallah and S. Leue, "Syntactic detection of process divergence and non-local choice in message sequence charts," in *Proc. 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1997.

[7] R. Alur and M. Yannakakis, "Model checking of message sequence charts," in *CONCUR'99: Concurrency Theory, Tenth International Conference*, 1999, LNCS 1664, pp. 114–129.

[8] G.J. Holzmann, D.A. Peled, and M.H. Redberg, "Design tools for requirements engineering," *Bell Labs Technical Journal*, vol. 2, no. 1, pp. 86–95, 1997.

[9] H. Ben-Abdallah and S. Leue, "MESA: Support for scenario-based design of concurrent systems," in *Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1998, LNCS 1384, pp. 118–135.

[10] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi, "Automated support for OO software," *IEEE Software*, vol. 15, no. 1, pp. 87–94, Jan./Feb. 1998.

[11] I. Kruger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," *Distributed and Parallel Embedded Systems*, 1999.

[12] S. Leue, L. Mehrmann, and M. Rezai, "Synthesizing ROOM models from message sequence chart specifications," in *Proc. 13th IEEE Conf. on Automated Software Engineering*, 1998.

[13] G.J. Holzmann, "Early fault detection tools," *LNCS*, vol. 1055, pp. 1–13, 1996.

[14] K. Koskimies and E. Makinen, "Automatic synthesis of state machines from trace diagrams," *Software-Practice and Experience*, vol. 24, no. 7, pp. 643–658, 1994.

[15] D. Harel and H. Kugler, "Synthesizing object systems from LSC specifications," Unpublished draft, 1999.

[16] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," in *Proc. 3rd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, 1999, pp. 293–312.

[17] M. Mukund, K.N. Kumar, and M. Sohoni, "Synthesizing distributed finite-state systems from MSCs," in *CONCUR 2000: Concurrency Theory, 11th International Conference*, LNCS 1877, pp. 521–535. Springer, 2000.

[18] A. W. Biermann and J. A. Feldman, "On the synthesis of finite state machines from samples of their behavior," *IEEE Trans. Computers*, pp. 592–597, 1972.

[19] D. Angluin and C. H. Smith, "Inductive inference: theory and methods," *ACM Computing Surveys*, vol. 15, pp. 237–269, 1983.

[20] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[21] V. Diekert and G. Rozenberg, Eds., *The Book of Traces*, World Scientific Publishing, 1995.

[22] M. Lohrey, "Safe realizability of high-level message sequence charts," in *CONCUR 2002: Concurrency Theory, 13th International Conference*, LNCS 2421, pp. 177–192, 2002.

[23] D. Maier, Y. Sagiv, and M. Yannakakis, "On the complexity of testing implications of functional and join dependencies," *Journal of the ACM*, vol. 28, no. 4, pp. 680–695, 1981.

[24] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," in *Proceedings of 22nd International Conference on Software Engineering*, pp. 304–313. 2000.