Inference of Message Sequence Charts

Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis

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 "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.

Index Terms—Message sequence charts, requirements analysis, formal verification, scenarios, concurrent state machines, deadlock freedom, realizability, 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 payoff. 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 nonlocal 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.

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 (co-NP-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 the full paper ([25], Section 8), we point out how our results can be generalized to a variety of communication architectures in a generic manner.

Because of space limitations in this journal, we must exclude some pieces of the full paper. Several proofs and sections have been omitted or shortened. Please see the full paper ([25]).

1.1 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

- R. Alur is with the Department of Computer and Information Science, 609 Levine, University of Pennsylvania, 200 S. 33rd St., Philadelphia, PA 19104. E-mail: alur@cis.upenn.edu.
- K. Etessami is with the School of Informatics, JCMB Room 1509, University of Edinburgh, Edinburgh EH9 3JZ, Scotland UK. E-mail: kousha@inf.ed.ac.uk.
- M. Yannakakis is with the Department of Computer Science, Room 462 Gates Hall, Stanford University, Stanford, CA 94305-9040. E-mail: mihalis@cs.stanford.edu.

Manuscript received 18 Sept. 2000; revised 3 Dec. 2002; accepted 30 Dec. 2002.
Recommended for acceptance by J. Palsberg.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112879.

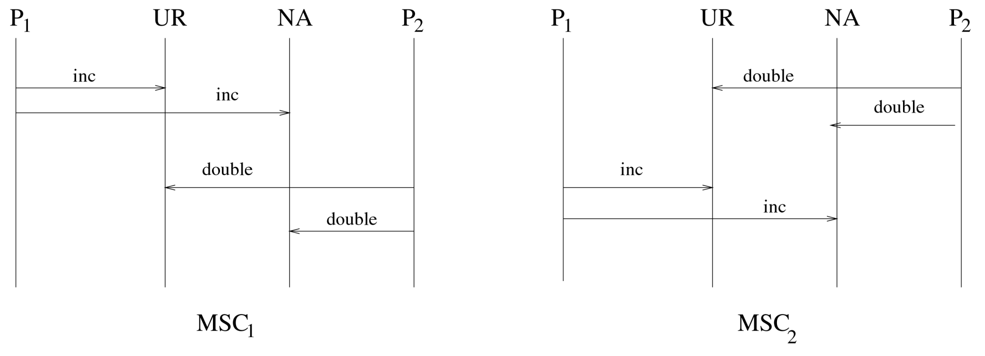

Fig. 1. Two seemingly "correct" scenarios, updating fuel amounts.

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 Fig. 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} in Fig. 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 Fig. 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].

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

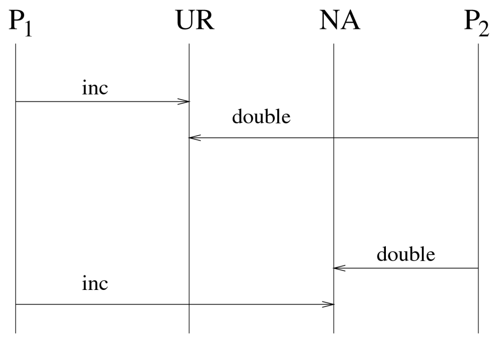

Fig. 2. Implied MSC_{bad}: Incorrect fuel mix.

Let $\mathcal{P} = \{P_1, \dots, P_n\}$ be a set of processes and Σ be a message alphabet. We write $[n]$ for $\{1, \dots, n\}$. 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 $\hat{\Sigma}^S = \{send(i, j, a) \mid i, j \in [n] \;\; a \in \Sigma\}$ 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 $l(f(s)) = receive(i, j, a)$ 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 Fig. 1 is shown in Fig. 3.

Besides the above, we require our MSCs to satisfy an additional *nondegeneracy* 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 Fig. 4. In both MSC_I and MSC_{II}, $P1$ sends two as and $P2$ receives two as. 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, in the rest of this paper, MSCs refer to *nondegenerate* MSCs.

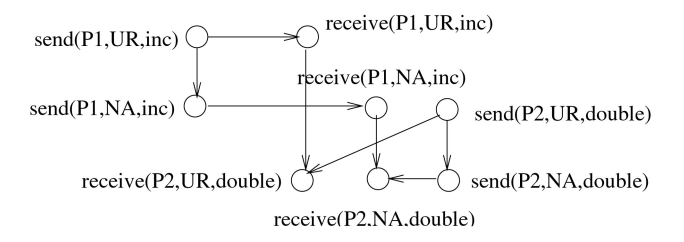

Fig. 3. Partial order representation of MSC_1.

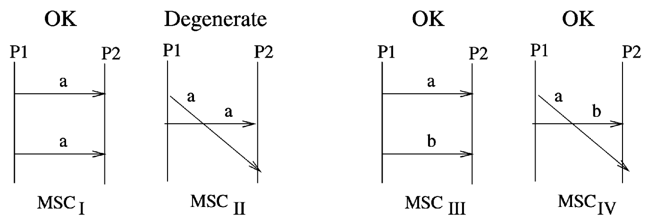

Fig. 4. Degeneracy in MSCs.

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 conditions 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 $x = receive(i, j, a)$ with $\#(v, send(i, j, a)) - \#(v, receive(i, j, a)) > 0$. A word w is *well-formed* if, for every prefix vx of w, x is possible after v.

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 $a \in \Sigma$,

2. When dealing specifically with FIFO architectures, (via the general framework in Section 8 of the full version [25]), we will explicitly forbid crossing of the kind in MSC_{IV} as well.

$$\#(w, send(i,j,a)) - \#(w, receive(i,j,a)) = 0.$$

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.*

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. Similarly, define 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 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 1. *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.*

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). We note that CC1 can alternatively be formalized using semitraces over an appropriately defined independence relation over alphabet $\hat{\Sigma}$ (see, e.g., [21]).

3. Note that the MSC M is nondegenerate by assumption and this assumption is required.

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 show in Section 8 of the full version ([25]) that our results apply in a general framework which captures a variety of alternative architectures. For clarity of presentation, in this 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, 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 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 $B_{i,j}^s$ and $B_{i,j}^r$. The first buffer, $B_{i,j}^s$, 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 $B_{i,j}^r$ 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 multisets 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 $B_{i,j}^s, B_{i,j}^r \in Q_\Sigma$. 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 alphabet $\hat{\Sigma}$, 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 $B_{i,j}^s$ and $B_{i,j}^r$. 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 $x = receive(j, i, a)$, 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 $x = send(i, j, a)$, 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 3. *For a 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 a 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 and 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. (For proof, see [25].)

Theorem 2. *A language L over the alphabet $\hat{\Sigma}$ is weakly realizable iff L contains only well-formed and complete words and satisfies CC2.*

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

Corollary 1. *Given MSC set $\mathcal{M}$, and MSC M': $\mathcal{M} \vdash^W M'$ if and only if for each process $i \in [n]$, there is an MSC $M^i \in \mathcal{M}$ such*

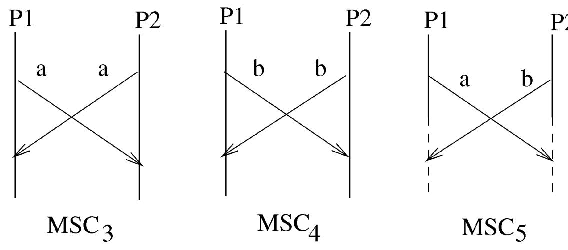

Fig. 5. Weakness of weak realizability.

that $M'|_i = M^i|_i$. An MSC family $\mathcal{M}$ is weakly realizable iff $L(\mathcal{M})$ satisfies CC2.

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 nonaccepting 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 Fig. 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 $L(\{\mathrm{MSC}_3, \mathrm{MSC}_4\})$ 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 Fig. 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,

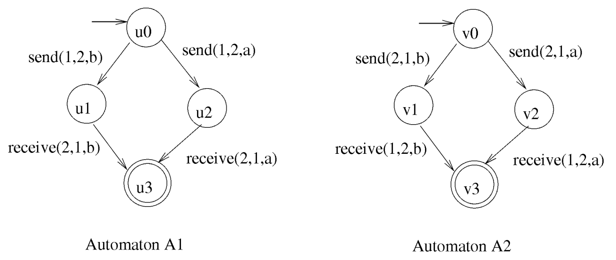

Fig. 6. Concurrent automata corresponding to MSC_3 and MSC_4.

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 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 $pref(L)$ 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 $v^i \in pref(L)$ such that $w|_i = v^i|_i$, then w is in $pref(L)$.[4]

An equivalent definition, which is easier to check algorithmically, is:

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

Proposition 4. *L satisfies CC3 iff it satisfies CC3′.*

Proof. See [25] □.

The intuition behind the above is as follows: 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 Fig. 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. Note 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.

4. Note that this corresponds to the CC2 closure condition on $pref(L)$, without the requirement of completeness on w.

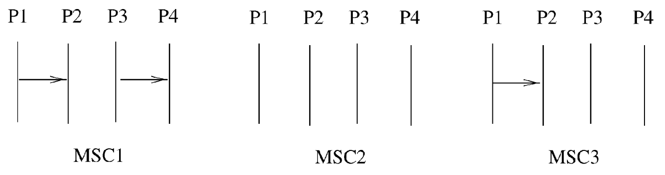

Fig. 7. Safe realizability is not entirely captured by CC3.

Safe realizability is not entirely captured by closure condition CC3 [22]. This is illustrated by the example scenarios in Fig. 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 3. *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. See [25]. □

Corollary 2. *An MSC family $\mathcal{M}$ is safely realizable iff $L(\mathcal{M})$ satisfies CC3 and CC2′.*

7 Algorithms: 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 2, 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 4. *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′. Fig. 8 gives a simple version of our polynomial time algorithm for checking CC3.

7.1.2 Correctness

Correctness of the algorithm is based on Proposition 4. 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 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 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 Fig. 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

```
proc $Condition\_CC3(\mathcal{M})$ $\equiv$
  foreach $(s,t,i) \in [k] \times [k] \times [n]$ do
    $T[s,t,i] := \min\{c \mid (M[s,i,c] \neq M[t,i,c])\}$
  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$[s,t,i] = \bot$ */
  Let $\leq^s$ be the partial order of events in $M_s$.
  foreach $s \in [k]$ and event $x$ in $M_s$ do
    foreach process $j \in [n]$ do
      $U[s,x,j] :=$
      $\begin{cases} \|M[s,j]\| + 1 & \text{if } \forall c\ x \not\leq^s M[s,j,c] \\ \min\{c \mid (x \leq^s M[s,j,c])\} & \text{otherwise} \end{cases}$
    od;
  od;
  /* U$[s,x,*]$ gives the events of $M_s$ dependent on $x$ */
  foreach $(s,t,j) \in [k] \times [k] \times [n]$such that$T[s,t,j] \neq \bot$ 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 $M[s,i][1 \ldots U[s,x,i]-1]$ contains more */
    /* $send(i,j,a)$'s than $M[s,j][1 \ldots U[s,x,j]-1]$ */
    /* contains $receive(i,j,a)$'s. */
    if $x'$ is eligible to replace $x$ then
      /* Find if some $M_p$ realizes this replacement */
      if $\exists\ p \in [k]$ such that
        $M[p,j,c] = x'$ and
        $\forall j' \in [n]\ U[s,x,j'] \leq T[s,p,j']$
        then() /* This eligible replacement exists */
        else
         "$\mathcal{M}$ DOES NOT Satisfy CC3"
         Missing Implied partial MSC given by $\forall j'$
         $M[s,j'][1 \ldots U[s,x,j']-1]$ and $M[s,j][c] := x'$
         return;
      fi;
    fi;
  od;
  "YES. $\mathcal{M}$ DOES Satisfy CC3"
```

Fig. 8. Algorithm for checking condition CC3.

largest prefixes on each process is that the "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.

7.1.3 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 details are involved and are given in [25].

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.

7.1.4 Example

Consider the two MSCs of Fig. 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, it 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

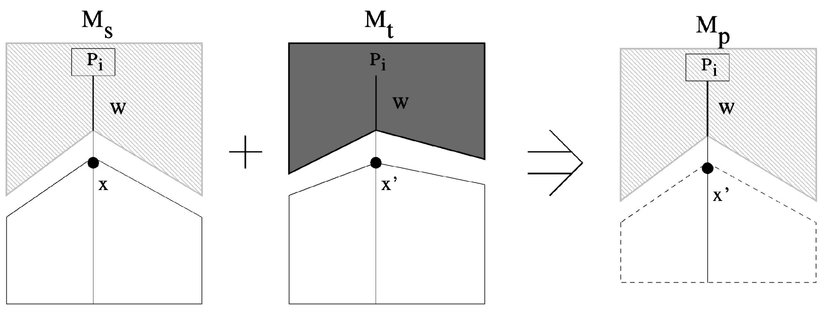

Fig. 9. Inference algorithm illustration.

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 Fig. 2.

7.1.5 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 pseudocode.

1. For each MSC M_s in $\mathcal{M}$ and, for each process P_i, compute $v_i = M_s|_i$. Compute the set $V_s^i = pref(v_i) \cap \{M|_i \mid M \in \mathcal{M}\}$, 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) multiset 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 $w_{i'}^{j'}$ **or** 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, nonpartial, 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."

7.1.6 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 $j \in \{1, \ldots, k_i\}$, where v_i^js 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′.

7.1.7 Time Complexity

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 $\mathcal{M}$. See [25].

7.2 Co-NP-Completeness of Weak Realizability

The less desirable realizability notion was weak realizability. There, deadlocks may occur. It turns out this weaker notion is 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 shows we cannot expect a polynomial time solution:

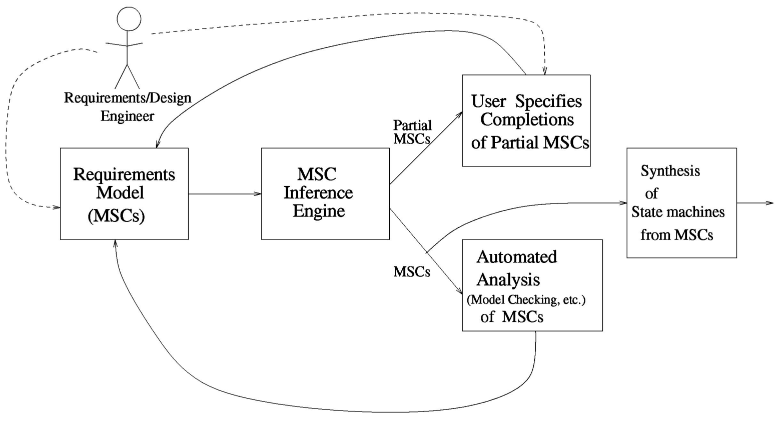
Requirements/Design Engineer
Requirements Model (MSCs)
MSC Inference Engine
Partial MSCs
User Specifies Completions of Partial MSCs
MSCs
Automated Analysis (Model Checking, etc.) of MSCs
Synthesis of State machines from MSCs

Fig. 10. Using the MSC inference framework.

Theorem 5. *Given MSC set $\mathcal{M}$, determining weak realizability of $\mathcal{M}$ is co-NP-complete.*

Proof. See [25]. □

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 2, it is straightforward to synthesize the A_is. 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, $pref(L_i)$, in L_i, and whose transitions are $\delta(q_w, x, q_{wx})$, where $x \in \hat{\Sigma}$ and $w, wx \in pref(L_i)$. 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_is, which collapses leaves and possibly other states, to obtain smaller automata. Note that the A_is can be constructed in time linear in $\mathcal{M}$. Letting $A_{\mathcal{M}} = \Pi A_i$, we claim:

Theorem 6. *$L(A_{\mathcal{M}})$ 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, moreover, if $L(\mathcal{M})$ is safely realizable, then $A_{\mathcal{M}} = \Pi_i A_i$ is deadlock-free.*

8 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 Fig. 10. 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]).

Acknowledgments

A preliminary version of this paper appeared in the *Proceedings of the 22nd International Conference on Software Engineering* [24]. The authors would like to thank

Markus Lohrey for pointing out a bug in the characterization of safe realizability in that preliminary version ([22]). Due to space limitations in this journal, substantial pieces of the full paper have been excluded. For the full paper, please see [25]. R. Alur was supported in part by US National Science Foundation CAREER award CCR97-34115, grant CCR99-70925, and ITR award IR/SY01-21431.

REFERENCES

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

[2] E. Rudolph, P. Graubmann, and J. Gabowski, "Tutorial on Message Sequence Charts," *Computer Networks and ISDN Systems—SDL and MSC,* vol. 28, 1996.

[3] J. Rumbaugh, I. Jacobson, and G. 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," *Foundations of Software Science and Computer Structures,* 1998.

[6] H. Ben-Abdallah and S. Leue, "Syntactic Detection of Process Divergence and Nonlocal Choice in Message Sequence Charts," *Proc. Second Int'l Workshop Tools and Algorithms for the Construction and Analysis of Systems,* 1997.

[7] R. Alur and M. Yannakakis, "Model Checking of Message Sequence Charts," *Proc. CONCUR'99: Concurrency Theory, 10th Int'l Conf.,* pp. 114-129, 1999.

[8] G.J. Holzmann, D.A. Peled, and M.H. Redberg, "Design Tools for Requirements Engineering," *Bell Labs Technical J.,* vol. 2, no. 1, pp. 86-95, 1997.

[9] H. Ben-Abdallah and S. Leue, "MESA: Support for Scenario-Based Design of Concurrent Systems," *Proc. Fourth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* pp. 118-135, 1998.

[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," *Proc. 13th IEEE Conf. Automated Software Eng.,* 1998.

[13] G.J. Holzmann, "Early Fault Detection Tools," *Proc. Sixth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* 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," *Proc. Third IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS '99),* pp. 293-312, 1999.

[17] M. Mukund, K.N. Kumar, and M. Sohoni, "Synthesizing Distributed Finite-State Systems from MSCs," *Proc. CONCUR 2000: Concurrency Theory, 11th Int'l Conf.,* 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] *The Book of Traces.* V. Diekert and G. Rozenberg, eds., World Scientific Publishing, 1995.

[22] M. Lohrey, "Safe Realizability of High-Level Message Sequence Charts," *Proc. CONCUR 2002: Concurrency Theory, 13th Int'l Conf.,* 2002.

[23] D. Maier, Y. Sagiv, and M. Yannakakis, "On the Complexity of Testing Implications of Functional and Join Dependencies," *J. ACM,* vol. 28, no. 4, pp. 680-695, 1981.

[24] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," *Proc. 22nd Int'l Conf. Software Eng.,* pp. 304-313, 2000.

[25] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," technical report, Laboratory for Foundations of Computer Science, Univ. of Edinburgh, 2003, http://homepages.inf.ed.ac.uk/kousha/msc_inference_j_v.ps.

Rajeev Alur received the BTech degree in computer science from the Indian Institute of Technology at Kanpur, and the PhD degree in computer science from Stanford University, California. He is a professor of computer and information science at the University of Pennsylvania. Before that, Dr. Alur spent six years at Bell Laboratories, Lucent Technologies, as a member of the technical staff. Professor Alur's research interests include software engineering, design automation for embedded systems, and applied formal methods. He has published more than 100 articles in refereed journals and conference proceedings, and served on numerous scientific committees. He is well-known for his research on timed and hybrid automata, a framework for specification and analysis of embedded real-time systems. He coorganized and cochaired the 1995 Workshop on Hybrid Systems, the 1996 Conference on Computer-Aided Verification, and the 2003 Conference on Embedded Software. His awards include the Sloan Faculty Fellowship and US National Science Foundation CAREER and ITR awards.

Kousha Etessami received the PhD degree in computer science from the University of Massachusetts at Amherst in 1995, and the BS degree in computer science from the State University of New York at Albany in 1990. Dr. Etessami joined the University of Edinburgh as a lecturer in the Laboratory for Foundations of Computer Science in 2002. Prior to that, he was a member of the technical staff in the Computing Principles Research Department of Bell Laboratories (1997-2002). He held postdoctoral positions at the DIMACS center for Discrete Mathematics and Theoretical Computer Science in New Jersey (1995-96), and at the BRICS center for Basic Research in Computer Science in Aarhus, Denmark (1996-97). Dr. Etessami's research interests include automated verification, software analysis and testing, automata and temporal logic, algorithms and complexity, databases, and applications of logic.

Mihalis Yannakakis received the diploma in electrical engineering from the National Technical University of Athens, Greece, in 1975, and the PhD degree in computer science from Princeton University, in 1979. Dr. Yannakakis joined Stanford University in 2002 as a professor of computer science. Prior to that, he was affiliated with Avaya Laboratories (2001-02) and Bell Laboratories (1978-2001), serving as the director of the Computing Principles Research Department since 1991. His research interests include algorithms and complexity, combinatorial optimization, databases, testing, and verification. Dr. Yannakakis is the editor-in-chief of the *SIAM Journal on Computing* and serves on the editorial boards of several other journals. He has served on the program committees and chaired various conferences, including the IEEE Symposium on Foundations of Computer Science, the ACM Symposium on Theory of Computing, and the ACM Symposium on Principles of Database Systems. Dr. Yannakakis is a fellow of the ACM and a fellow of Bell Labs.