

Inference of Message Sequence Charts

Rajeev Alur
Dept. of Computer & Info. Sci.
U. of Pennsylvania, and
Bell Labs
alur@cis.upenn.edu

Kousha Etessami
Bell Labs
600 Mountain Ave.
Murray Hill, NJ 07974
kousha@research.bell-labs.com

Mihalis Yannakakis
Bell Labs
600 Mountain Ave.
Murray Hill, NJ 07974
mihalis@research.bell-labs.edu

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 one 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 we then provide polynomial-time algorithms for implication, realizability, and synthesis. In particular, we describe a novel algorithm for checking deadlock-free (safe) realizability.

Keywords

Message sequence charts, 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 [16, 19], and have been incorporated into software design notations such as UML [20]. 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 yields a high pay-off. This has already motivated the development of algorithms for a variety of analyses including detecting race conditions and timing conflicts [1], pattern matching [17], detecting non-local

choice [4], and model checking [2], and tools such as uBET [11], MESA [5], and SCED [13]. 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 MSCs to be *realizable* if there exist concurrent automata which implement precisely those MSCs.

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 concurrent automaton 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 pre-processing 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.

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 [14, 15, 10, 12, 9, 7]. 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 present a clean language-theoretic framework to formalize these problems via closure conditions. Lastly and importantly, 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.

It is worth noting that inferring sequential state machines from example executions is a well-studied topic in automata theory [6, 3]. 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., [18]).

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} 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 P_1 (P_2) updates the fuel levels first, followed by P_2 (resp., P_1).

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 ITU standard MSC’96 [16], and is analogous to the definitions of labeled MSCs given in [1, 2].

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 $s \in S$,

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

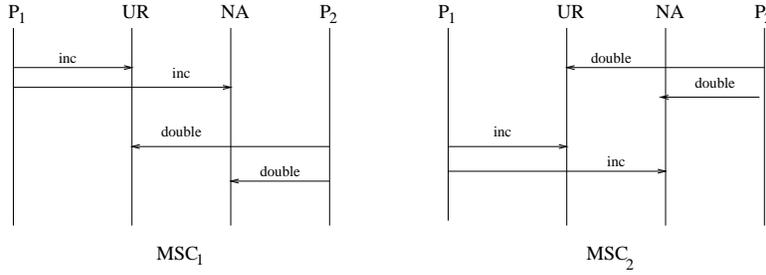


Figure 1: Two seemingly “correct” scenarios, updating fuel amounts

if $l(s) = \text{send}(i, j, a)$ then $p(s) = i$ and $l(f(s)) = \text{receive}(i, j, a)$ and $p(f(s)) = j$;

- 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 can’t 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} , P_1 sends two a ’s and P_2 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 P_2 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.

²When 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.

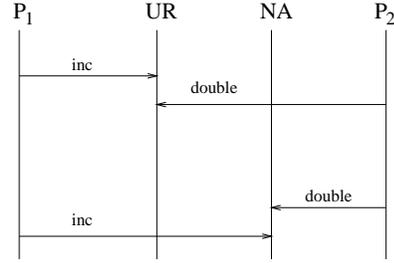


Figure 2: Implied MSC_{bad} : Incorrect fuel mix

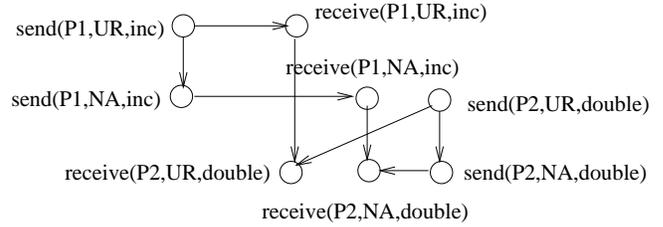


Figure 3: Partial order representation of MSC_I

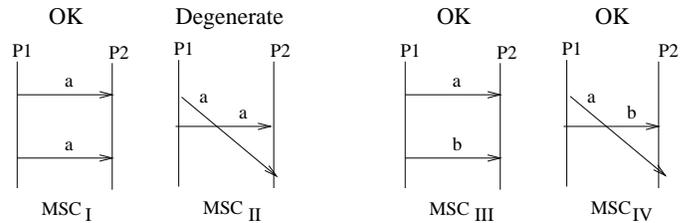


Figure 4: Degeneracy in MSCs

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 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 $x = \text{receive}(i, j, a)$ with $\#(v, \text{send}(i, j, a)) - \#(v, \text{receive}(i, j, a)) > 0$. 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 $a \in \Sigma$, $\#(w, \text{send}(i, j, a)) - \#(w, \text{receive}(i, j, a)) = 0$. 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.*

More is true. Let $M|_i$ denote the ordered sequence of labels of events occurring on process i in the MSC M :

Proposition 2 *An MSC M over $\{P_1, \dots, P_n\}$ 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 $\langle w|_i \mid i \in [n] \rangle$.*

Both propositions follow from the fact that from any well-formed and complete word w one can build a *canonical* MSC, $\text{msc}(w)$, by progressively matching receives in prefixes of the word w with the first corresponding send in w which is yet to be matched. It is easy to verify that if w is some linearization of an MSC M , then M can be reconstructed from w , i.e. $\text{msc}(w) = M$. For an

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

MSC M , define $L(M)$ to be the set of all linearizations of M . For a set \mathcal{M} of MSCs, the language $L(\mathcal{M})$ is the union of languages of all MSCs in \mathcal{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 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. Now, the following theorem characterizes the 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 CC1.*

The proof uses the fact that one can recover uniquely an MSC from any of its linearizations. 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, [8]).

We will find useful the notion of a partial MSC. A *partial MSC* is given by a well-formed, not necessarily complete, sequence v , or, equivalently, by the projections 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 $\prod_{i=1}^n A_i$, 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 $B_{i,j}^s$ 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 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 $B_{i,j}^s, B_{i,j}^r \in Q_\Sigma$. 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 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 $B_{i,j}^s$ 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 $B_{j,i}^r$ 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 $B_{i,j}^s$ 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 $B_{i,j}^s$ in state q' is obtained from

the corresponding buffer in state q by deleting one message a , and the buffer $B_{i,j}^r$ 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:

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

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*

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

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 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*. Note that CC2 immediately implies CC1. The other direction does not hold.

Going back to our example from Section 2, the language $L(\{MSC_1, MSC_2\})$ 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

$send(P1, UR, inc) \ receive(P1, UR, inc)$
 $send(P2, UR, double) \ receive(P2, UR, double)$
 $send(P2, NA, double) \ receive(P2, NA, double)$
 $send(P1, NA, inc) \ receive(P1, NA, inc)$

The word w is not in $L(\{MSC_1, MSC_2\})$, 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

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

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.*

We thus have characterizations of weak implication and realizability of MSCs⁴:

Corollary 6 *Given MSC set \mathcal{M} , and MSC M' :*

$\mathcal{M} \stackrel{W}{\vdash} M'$ *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.*

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 rejecting state in which all processes are waiting to receive messages which do not exist in the buffers will be a deadlock

⁴Due to space limitation, all proofs are omitted. An extended version of the paper can be obtained by contacting the authors.

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*

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

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 $L(\{MSC_3, MSC_4\})$ 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 implementations. We need a stronger version of implication closure. 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 $\hat{\Sigma}$ is said to satisfy closure condition CC3 if: 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 $pref(L)$.⁵*

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

⁵Note that this corresponds to the CC2 closure condition on $pref(L)$, without the requirement of completeness on w .

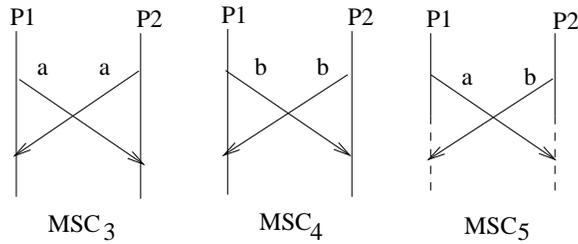


Figure 5: Weakness of weak realizability

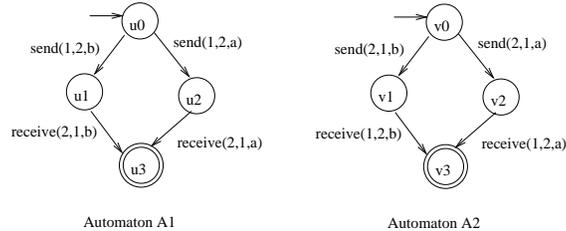


Figure 6: Concurrent automata corresponding to MSC_3 and MSC_4

Proposition 7 L satisfies CC3 iff for all $w, v \in \text{pref}(L)$ and all processes i , if $w|_i = v|_i$ and $wx \in \text{pref}(L)$ for some $x \in \hat{\Sigma}_i$, then if vx is well-formed vx is also in $\text{pref}(L)$.

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 $\text{send}(1,2,a)$ is a possible partial scenario (according to MSC_3), and the event $\text{send}(2,1,b)$ is a possible partial scenario (according to MSC_4). Now, CC3 requires that the sequence $\text{send}(1,2,a), \text{send}(2,1,b)$ 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 $\{MSC_3, MSC_4\}$ 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. The correspondence between safe realizability and condition CC3 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 CC3.

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

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.

An Algorithm for Safe Realizability

Given MSCs $\mathcal{M} = \{M_1, \dots, M_k\}$, where each MSC is a scenario over n processes P_1, \dots, 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 partial MSC, M' , which must exist as a partial execution of some MSC, but does not in \mathcal{M} .

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 three dimensional table, with $M[l][i][d]$ giving the label of the d 'th event in the sequence of events on process P_i in MSC M_l . Thus $M[l][i] = M_l|_i$. Let $\|M[l][i]\|$ denote the length of the sequence $M_l|_i$.

By Corollary 9, it suffices to check that $L(\mathcal{M})$ satisfies

CC3. A straightforward algorithm to check CC3 would have exponential complexity. We show how to check CC3 in polynomial time. Figure 7 gives a simple version of our polynomial time algorithm for CC3.

Correctness: The correctness of the algorithm is based on Corollary 9 and Proposition 7. CC3 is violated if and only if whenever there are x and x' , in M_s and M_t , as in the algorithm, such that x' is eligible to replace x in the *largest prefix on all processes of M_s in which no event depends on x* , there is an M_p which realizes this replacement. The reason it suffices to check 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 .

Complexity: The stated algorithm is somewhat wasteful. With more careful data structures, the running time can be improved to: $O(k^2 \cdot n + r \cdot n)$.

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 $(k^2 \cdot n)$ 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' . The reason $(k^2 \cdot n)$ such partial MSCs suffice is that, 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 , and substitute x by w' . 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 $receive(P_1, UR, inc)$, whereas in MSC_2 the first two events on UR are $receive(P_2, UR, double)$ and

```

proc SafeRealizability( $\mathcal{M}$ )  $\equiv$ 
  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 */
  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  $(s, t, j) \in [k] \times [k] \times [n]$  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  $M[s][i][1 \dots U[s, x, i] - 1]$  contains more */
    /*  $send(i, j, a)$ 's than  $M[s][j][1 \dots U[s, x, j] - 1]$  */
    /* 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
        “ $\mathcal{M}$  NOT SAFELY REALIZABLE”
        Missing Implied partial MSC given by  $\forall j'$ 
         $M[s][j'][1 \dots U[s, x, j'] - 1]$  and  $M[s][j][c] := x'$ 
        return;
      fi;
    fi;
  od;
  “YES.  $\mathcal{M}$  IS SAFELY REALIZABLE”

```

Figure 7: Algorithm for Safe Realizability

$receive(P_1, UR, inc)$. Since in MSC_1 no events, other than those on UR, depend on the first event on UR, we see that the sequence of two events on MSC_2 are indeed eligible to replace the first event on UR in MSC_1 . The result of this replacement is precisely MSC_{bad} , the inferred MSC in Figure 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, NP) for checking weak realizability, and we can't expect a polynomial time solution:

Theorem 10 *Given a set of MSCs \mathcal{M} , determining whether \mathcal{M} is weakly realizable is coNP-complete.*

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 $L(\Pi A_i) = L(\mathcal{M})$ (and, when safely realizable, such that ΠA_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, $pref(L_i)$, 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_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 11 *$L(A_{\mathcal{M}})$ 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 $A_{\mathcal{M}} = \Pi A_i$ 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

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

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.

As two examples, consider *enabled* when (1) queues are required to be FIFO, and (2) when the message exchanges are synchronous, i.e., when a sending process cannot continue until the message is received (and implicitly acknowledged). In case (1), in addition to the axioms, we require that *enabled*($w, receive(i, j, a)$) can only hold if the first *send*(i, j, \star) in w for which there is no matching *receive*(i, j, \star) is indeed *send*(i, j, a). In case (2), for any event x on process i , we require that *enabled*(w, x) holds only when all sends on process i have a matching receive in w .

We reformulate well-formedness, completeness, and the different closures conditions, in this more general setting: 1. **Well-Formedness**: for every prefix $w'x$ of $w \in L$, *enabled*(w', x), 2. **Completeness**: The definition of completeness remains exactly the same. 3. **CC2** and **CC3**, also remain the same.

We show that, for each architecture, Theorems 5 and 8 remain true under these modified conditions.

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.

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., [14]), and high-level MSCs (as in, e.g., uBET [11]).

Acknowledgements

This research was supported in part by NSF CAREER award CCR97-34115, NSF grant CCR99-70925, DARPA/NASA grant NAG2-1214, and Sloan Faculty Fellowship.

REFERENCES

- [1] R. Alur, G. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
- [2] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pages 114–129, 1999.
- [3] D. Angluin and C. H. Smith. Inductive inference: theory and methods. *ACM Computing Surveys*, 15:237–269, 1983.
- [4] 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.
- [5] 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*, LNCS 1384, pages 118–135, 1998.
- [6] A. W. Biermann and J. A. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Trans. Computers*, pages 592–597, 1972.
- [7] 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)*, pages 293–312, 1999.
- [8] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific Publishing, 1995.
- [9] D. Harel and H. Kugler. Synthesizing object systems from LSC specifications. Unpublished draft, 1999.
- [10] G. Holzmann. Early fault detection tools. *LNCS*, 1055:1–13, 1996.
- [11] G. Holzmann, D. Peled, and M. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, 2(1):86–95, 1997.
- [12] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software-Practice and Experience*, 24(7):643–658, 1994.
- [13] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. Automated support for OO software. *IEEE Software*, 15(1):87–94, Jan./Feb. 1998.
- [14] I. Kruger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. *Distributed and Parallel Embedded Systems*, 1999.
- [15] S. Leue, L. Mehrmann, and M. Rezaei. Synthesizing ROOM models from message sequence chart specifications. In *Proc. 13th IEEE Conf. on Automated Software Engineering*, 1998.
- [16] ITU-T recommendation Z.120. Message Sequence Charts (MSC'96), May 1996. ITU Telecommunication Standardization Sector.
- [17] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Foundations of Software Sci. and Comp. Structures*, 1998.
- [18] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.
- [19] E. Rudolph, P. Graubmann, and J. Gabowski. Tutorial on message sequence charts. In *Computer Networks and ISDN Systems – SDL and MSC*, volume 28. 1996.
- [20] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.