Checking consistency of SDL+MSC specifications*

Deepak D'Souza and Madhavan Mukund

Chennai Mathematical Institute
92 G N Chetty Road, Chennai 600 017, India
Email: {deepak,madhavan}@cmi.ac.in

Abstract. We consider the problem of checking whether a distributed system described in SDL is consistent with a set of MSCs that constrain the interaction between the processes. In general, the MSC constraints may be both positive and negative. The system should execute all the positive scenarios "sensibly". On the other hand, the negative MSCs rule out some interactions as illegal. We would then like to verify that all the remaining legal interactions satisfy a desired global property, specified in linear-time temporal logic. We outline an approach to solve this problem using Spin, building in a modular way on existing tools.

1 Introduction

A distributed system involves several modules that interact with each other to produce a global behaviour. The specification of each module in the system describes its local data structures and control flow. At each step, a module either executes an internal action to update its local data or communicates and exchanges data with another module.

The interaction between modules is usually described in terms of scenarios, using mechanisms such as use-cases and message sequence charts. In general, it is difficult to exhaustively analyze all possible interaction scenarios and arrive at a distributed implementation that permits precisely the set of desired scenarios.

A more realistic approach is to iteratively maintain two sets of scenarios, positive and negative. Positive scenarios are those that the system is designed to execute—for instance, these may describe a handshaking protocol to set up a reliable communication channel between two hosts on a network. Negative scenarios indicate undesirable behaviours that the designer is aware of, such as a situation when both hosts simultaneously try to set up the channel, leading to a collision. In general, the set of positive and negative scenarios is not likely to be exhaustive—an interaction may not be ruled out by the set of negative scenarios, even though it is not explicitly one of the positive scenarios.

A reasonable expectation at each stage in the iterative design of the system is the following:

* Partly supported by a grant from Tata Research Development and Design Centre, Pune, India.

- The system should be able to execute every positive scenario in at least one way. Each such execution should leave the system in a specified safe state.
- All legal behaviours of the system—those that do not exhibit any of the negative scenarios—should satisfy a desired global property.

If the system fails the first test, the existing design has a major flaw that must be fixed. On the other hand, failure to pass the second test probably reveals an incomplete understanding on the part of the system designer of which interaction scenarios are undesirable. In either case, the test provides some insight into how the design should be refined in the next iteration.

When the system passes both tests, the designer can concentrate on cutting out the current set of negative scenarios to complete the design. The interactions permitted by the system at this point may exceed the positive scenarios required by the original design, but this relaxation on its own does not violate the global specification. One virtue of this approach is that the designer naturally arrives at a less constrained, simpler implementation of the specification, rather than a precise implementation that may be unnecessarily complex.

To make the problem more concrete, we fix the following context: individual modules are described using the visual specification language SDL [11, 14]. Scenarios involving specific subsets of modules are specified using collections of message sequence charts (MSCs) [10, 16]. The global specification is a formula in linear-time temporal logic (LTL) [15, 12].

We propose a solution to the problem using the model-checking system Spin [9]. The tools described in [4, 3] jointly provide an automated framework for translating a large class of SDL specifications into Promela, the process description language used by Spin. Our approach is to add an extra *monitor* process to the Promela translation of an SDL specification. Each of the Promela processes arising out of the translation is modified so that it synchronizes with the monitor process whenever it sends or receives a message. The monitor process can thus track the communication pattern executed by the original set of processes.

One complication is that we interpret the positive and negative MSC specifications as templates that may embed loosely in the actual communication graph of the processes [13]. The monitor process has to determine how the template is embedded. For positive specifications, it suffices to guess the embedding. To deal with negative specifications, however, we need to check embedability deterministically. We establish a graph-theoretic property of embeddings that permits us to construct a deterministic monitor process. After constructing an appropriate monitor process, depending on whether we are checking the positive or negative scenarios, we enhance the original temporal logic specification for the system with a temporal logic assertion about the monitor to form a more complex formula that can be automatically converted into a Spin *never claim*.

The paper is organized as follows. We begin with a description of how we interpret positive and negative MSC specifications. The next section provides some background on synthesizing finite-state automata from MSCs. Section 4 is the heart of the paper, describing how we tackle the consistency problem in Promela. We conclude with a summary and discussion of future work.

2 Positive and negative MSC specifications

We assume that the processes in the distributed system specified in SDL are connected by one-way, point-to-point, FIFO channels. In general, a pair of processes may be connected by more than one such channel.

A sequence of messages transmitted between the processes can be described graphically using a message sequence chart (MSC). We do not formally define either SDL or MSCs in this paper—both notations are reasonably intuitive and we will explain the notation through some representative examples.

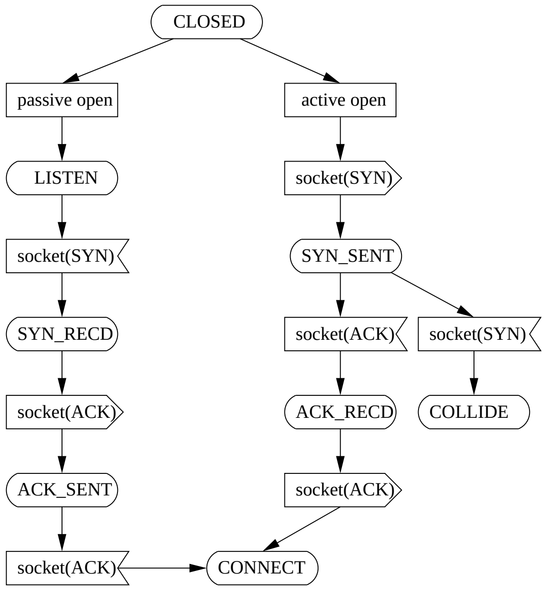

Fig. 1. A simple TCP-like connection establishment protocol

In Figure 1, a simplified version of the connection phase of the TCP protocol is specified in SDL. The system has two identical copies of this process. One of the copies is expected to be passive (the *server*) while the other is active (the *client*). The server performs a passive open and waits in state LISTEN. The client performs an active open and sends a SYN to the server, who responds with an ACK. The client then replies with another ACK and both client and server move into the state CONNECT. If the client receives SYN after an active open, it aborts the connection and goes into the state COLLIDE, indicating that a collision has occurred with an active open of another client.

The desired global property for this run is that whenever both processes have moved off the CLOSED state and at least one of them is the client, a connection is established—that is, both processes reach the state CONNECT. However, it is easy to observe that the property fails if both processes simultaneously perform an active open. This will lead to both processes moving to the state COLLIDE.

We can rule out such a deadlock with the positive and negative MSC specifications in Figure 2. The positive MSC specifications are given by the MSCs Allowed while the negative MSC specification is given by the MSC Disallowed. The positive MSCs describe the two symmetric desirable scenarios where one process acts as a client and the other as a server. The negative MSC describes an undesirable scenario where both processes simultaneously start off as clients. Notice that the positive and negative scenarios in this case are matched *exactly* in the communication pattern of the system being analyzed.

We shall assume, in general, that both the sets Allowed and Disallowed are finite—this is a reasonable assumption because most real world system specifications do, in fact, enumerate only a finite set of scenarios. We suggest how to deal with a relaxation of this finiteness requirement in Section 5.

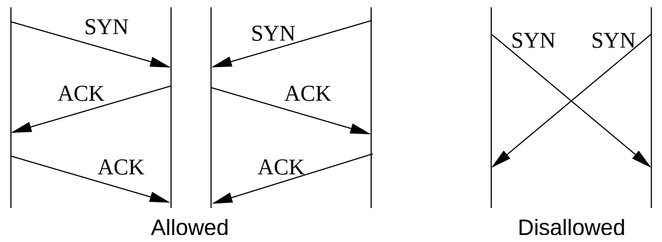

Fig. 2. MSC specifications for the connection establishment protocol

As we have remarked, in the first example, the MSCs in Allowed and Disallowed are matched *exactly* by the communication pattern between the processes. In general, we relax the interpretation of the sets Allowed and Disallowed and regard them as templates that may be *embedded* in the system behaviour.

The notion of one MSC being embedded in another is the usual one—there is an injective function mapping the messages in the first MSC into the messages in the second MSC that preserves the partial order between the events of the first MSC. (The next section describes how to represent MSCs as labelled partial orders. For a more formal definition of MSC embedding, see [13].)

For positive scenarios, embeddings permit the implementation to use auxiliary messages to implement the specification. Suppose we enhance our connection establishment protocol to permit the processes to handle a collision as shown in Figure 3. Nondeterministically, one of the processes decides to remain a client and requests the other process to exchange roles by sending an XCH message. If exactly one of the processes sends the XCH message, a connection is established.

The communication patterns exhibited by the enhanced protocol are shown in Figure 4. Observe that two new patterns lead to connection. Both of these embed the original positive specifications in Figure 2 (the embedded pattern is shown using larger arrowheads). The negative behaviours can be characterized

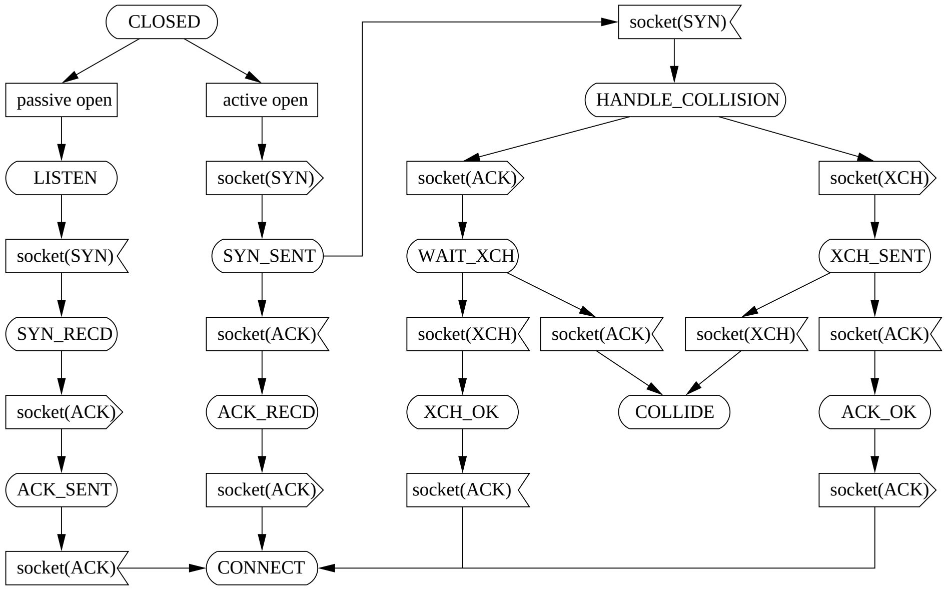

Fig. 3. An enhanced connection protocol

by the occurrence of crossing pairs of ACK messages or XCH messages. This yields a revised set Disallowed—see Figure 5. Observe that both the positive and negative scenarios in this case must be interpreted as templates to be embedded.

The example in Figure 6 illustrates another aspect of treating scenarios modulo embedding. The free behaviour of these two processes permits communication patterns like the k-MSC shown in the centre of Figure 7. The variable n in the first process keeps track of the number of messages in the channel $c2$ that are yet to be read by it.

If we impose the constraint Disallowed shown at the left of Figure 7, we rule out k-MSCs for all $k > 1$. Effectively, the only legal communication pattern is the one shown on the right of Figure 7. In other words, the MSC constraint Disallowed guarantees that the system satisfies the property that in every reachable global state, the value of n is bounded by 1. If we demanded an exact match of the MSCs in Disallowed we would have to generate an infinite family of incomparable MSCs, one for each k, to achieve the same effect.

3 From MSCs to finite-state automata

Each message in an MSC can be broken up into two events, one where the message is sent and the other where it is received. Let $\mathcal{P} = \{p, q, \ldots\}$ denote the set of processes, $Ch = \{c, c', \ldots\}$ the set of channels and $\Delta = \{m, m', \ldots\}$ a finite set of message types. Each channel is a point-to-point FIFO link between a

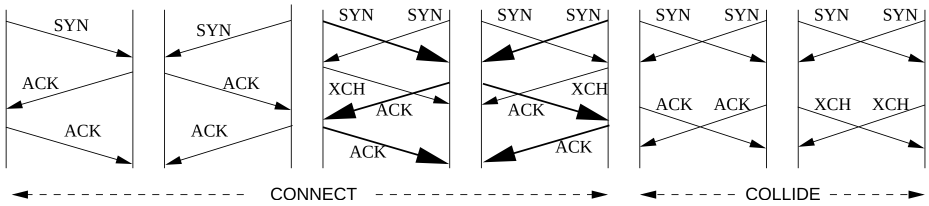

Fig. 4. MSCs exhibited by the enhanced protocol

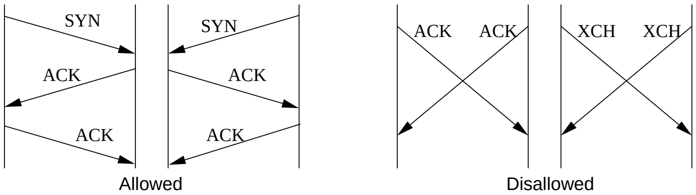

Fig. 5. Positive and negative MSC specifications for the enhanced protocol

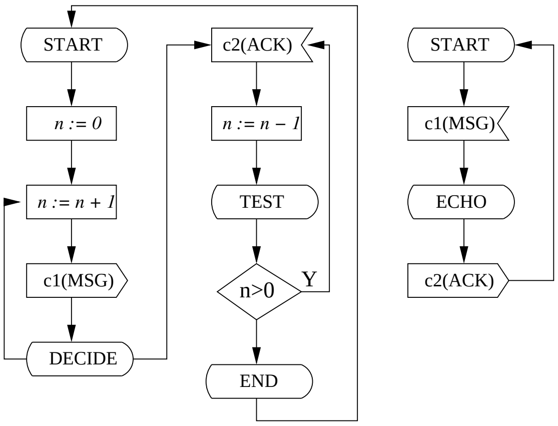

Fig. 6. A counting process

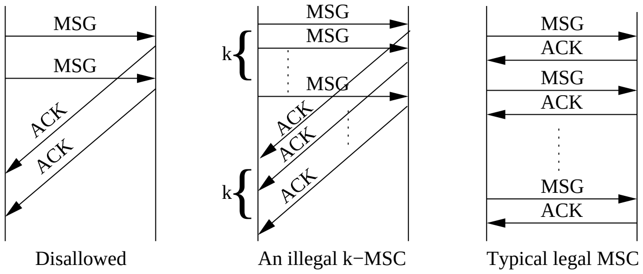

Fig. 7. MSC specifications for the counting process

pair of processes. Thus, we have functions $src : Ch \to \mathcal{P}$ and $tgt : Ch \to \mathcal{P}$ that uniquely identify the source and target process of each channel. The transfer of a message m from p to q on channel c generates a matching pair of events; the send event $c!m$ and the receive event $c?m$. We need not mention the processes p and q because these can be unambiguously recovered as $src(c)$ and $tgt(c)$, respectively.

The MSC defines a labelled partial order on these events. The partial order on events is obtained from the two basic orders implicit in the MSC:

- The send event for a message precedes the corresponding receive event.
- All events executed by a single process are linearly ordered.

An MSC can be uniquely recovered from the set of linearizations of its events. (In fact, a single linearization determines the structure of the MSC.) Stated differently, each MSC has a canonical representation as a finite language over the alphabet $\{c!m, c?m \mid c \in Ch, m \in \Delta\}$. We can thus associate with an MSC M a finite-state automaton $\mathcal{A}_M$ that recognizes the set of its linearizations.

We can extend this framework to define *regular* collections of MSCs. We say that a set $\mathcal{M}$ of MSCs is regular just in case the set of linearizations generated by the MSCs in $\mathcal{M}$ forms a regular language. If we look at the minimum deterministic finite-state automaton (DFA) $\mathcal{A}_\mathcal{M}$ associated with a regular collection of MSCs $\mathcal{M}$, we can uniquely associate with each state s in $\mathcal{A}_\mathcal{M}$ a vector of values indicating the number of messages pending (that is, as yet undelivered) in each channel at that state. This vector is an *invariant* property of the state—no matter which linearization takes us to this state, the channel contents at the state will be according to the given vector [8].

A useful formalism for presenting sets of MSCs is that of a hierarchical message sequence chart (HMSC). The most basic form of an HMSC is a message sequence graph (MSG). An MSG is a finite, directed graph in which each vertex is labelled by a single MSC. A path through the MSG traces out a single MSC obtained by concatenating the MSCs observed at the vertices that lie along the path. The MSG has a start vertex and a set of final vertices. The set of MSCs generated by the MSG is the set traced out by paths that originate at the start vertex and end at one of the final vertices. In general, an HMSC is like an MSG except that the annotation of a vertex can, in turn, be an HMSC, with the restriction that the overall level of nesting be bounded.

It is not difficult to show that HMSCs can define collections of MSCs that are not regular. A sufficient condition is that the HMSC satisfy a structural condition called *boundedness* [2]. Unfortunately, boundedness is not a necessary condition for regularity—in general, it is undecidable whether an HMSC defines a regular language [7]. However bounded HMSCs do satisfy a completeness property. It turns out that bounded HMSCs can describe all finitely-generated regular collections of MSCs—that is, collections that are generated by concatenating MSCs from a finite set of "atomic" MSCs [7].

In this paper, we restrict our attention to finite collections of MSCs, which are always regular. For reasons that we shall make clear in the next section, we will treat each MSC separately. In Section 5, we will discuss the possibility of extending our work to deal with infinite regular collections of MSCs.

4 Monitoring communications in Promela

As we mentioned in the Introduction, the tools described in [4, 3] provide a mechanism for translating SDL processes into Promela, the process description language used by the Spin system. We add a *monitor process* to the Promela translation of the SDL code and modify the code of every other process p to synchronize with the monitor process each time p communicates with another process q in the system. Before describing how to construct such a monitor process, we show how we use it to solve the problem of checking that the given SDL specification is consistent with the positive and negative MSC scenarios.

4.1 Verification using the monitor process

Let M be an MSC scenario. We assume that we can construct a monitor process with a local boolean variable `good` that is initially false and becomes true when the monitor detects that a run of the system embeds the MSC M.

Positive specifications For a positive scenario M, the problem of checking consistency can be broken up into two subgoals:

- *Liveness:* Show that it is possible for the system to exhibit the scenario M (in an embedded form).
- *Safety:* Show that whenever the system exhibits the scenario M, it satisfies a desirable global property. For instance, in Figure 1, the desirable property is that the two processes reach the state CONNECT. In general, we may assume that this desired property is specified by an LTL formula φ_M.

The LTL formula $\Diamond\texttt{good}$ specifies that the monitor process detects an embedding of M. To check the *liveness* condition, we can do conventional LTL model-checking for the formula $\neg\Diamond\texttt{good}$. If the system *does* satisfy this specification, then no execution of the system embeds the MSC M and the system under test fails the liveness condition.

To verify the *safety condition*, we check that the modified system incorporating the monitor process satisfies the LTL formula $\Diamond\texttt{good} \Rightarrow \varphi_M$. This formula asserts that any execution that embeds M must satisfy φ_M.

Thus, both the conditions that we need to verify reduce to model-checking formulas of LTL, which is built in to Spin. An important observation is that we do not need to reimplement the translation from LTL formulas into Spin *never claims*—we can use any standard translation, such as the algorithm from [6] that is built in to Spin or the newer translation described in [5].

Negative specifications Let M be a negative scenario and let ψ denote a desirable global property of the system. The goal is to show that any run of the system that does *not* embed M satisfies ψ. Equivalently, we have to show that every run either embeds M or satisfies ψ. This is captured by the LTL formula $\Diamond\texttt{good} \vee \psi$. Thus, verifying the consistency of negative scenarios also reduces to conventional LTL model-checking.

Nondeterminism in the monitor There is an important distinction between the positive and negative cases. The natural approach to detect whether M can be embedded in the communication pattern of the current run is to use nondeterminism. However, because of the nondeterminism, there will, in general, be runs of the system where the main computation does embed M but the monitor does not reach the state `good`.

This does not matter for positive specifications. For the liveness condition we use the negated formula $\neg\Diamond$`good` which checks that *no* run of the monitor reaches the state `good`. For the safety formula $\Diamond$`good` $\Rightarrow \varphi_M$, for every execution of the system that embeds M, there will be at least one run of the monitor that enters the state `good` and it is sufficient to verify that φ_M holds for such runs.

In the negative case, however, for the LTL formula to correctly capture the property we wish to verify, we must ensure that the monitor process reaches the state `good` *whenever* the current interleaving embeds M. For this, we need a more restrictive monitor process. One way to achieve the stronger requirement is to make the monitor deterministic.

4.2 Constructing the monitor process

Let `mon` be a new Promela process type and let `snoop` be a new channel shared by all the Promela process types, including `mon`, defined as: `snoop = chan[0] of (chan,byte,bit)`. The channel `snoop` is synchronous and each rendezvous exchanges a channel name, a message type and a bit indicating send/receive.

We assume that the SDL specification is written so that whenever p sends a message to q, the first component of the message designates one of the finite message-types used in the scenario specifications. We modify the Promela code of every process p in the SDL translation so that each statement of the form `c!m(a1,...,ak)` is replaced by `atomic{snoop!c,m,0; c!m(a1,...,ak)}`. In a similar fashion, each statement `c?m(x1,...,xk)` is replaced by `atomic{snoop!c,m,1; c?m(x1,...,xk)}`.

The messages on `snoop` inform the monitor process about the messages being exchanged by the main Promela processes. The `atomic` construct ensures that the sequence of communications observed by the monitor process is identical to the actual communication pattern in the current interleaved execution of the Promela processes. In the third parameter sent via `snoop`, 0 indicates a send and 1 a receive. The parameters `(a1,...,ak)` and `(x1,...,xk)` associated with message-type `m` are not relevant and hence ignored by `snoop`.

This transformation of the Promela processes generated automatically from the original SDL specification by the tools described in [4, 3] is completely uniform and can be achieved using a simple edit script.

A nondeterministic monitor Recall that the goal of the monitor is to detect whether the system specification embeds M. As we saw in Section 3, a regular MSC language is one for which we can construct a finite-state automaton over the alphabet $\{c!m, c?m \mid c \in Ch, m \in \Delta\}$ that recognizes the set of linearizations of all the MSCs in the language.

A single MSC M is a trivial example of a regular MSC language for which it is very simple to construct a recognizing automaton. If we project the events of M onto a process p, the semantics of MSCs guarantees that these p-events are linearly ordered. Thus, we can represent the MSC M in a canonical way in terms of the sequences of p-events that it generates, for each process $p \in \mathcal{P}$.

Clearly, for each sequence of p-events, we can construct a DFA that checks that its input matches this sequence—for the sequence $a_1 a_2 \dots a_m$, the automaton has $m+1$ states $s_0, s_1, \dots, s_m$ with initial state s_0, accepting state s_m and transitions $s_{i-1} \xrightarrow{a_i} s_i$, $i \in \{1, 2, \dots, m\}$. We can then run the DFAs for all the p-projections of M in parallel as a (free) product automaton to obtain a DFA $\mathcal{A}_M$ that recognizes all the linearizations of M.

The monitor process simulates $\mathcal{A}_M$ to decide whether the system run exhibits the MSC M. Since we are looking for embeddings, rather than faithful copies, of M, the monitor nondeterministically decides which send and receive events to include in the embedding. For this to work correctly, the monitor must ensure that whenever it includes a send (respectively, receive) event in the embedding, it also includes the matching receive (respectively, send) event.

To make a consistent nondeterministic choice across matching events, the monitor maintains as auxiliary data a list Marked of pairs of type `(chan,int)`. If a pair (c, i) is present in Marked, it means that the message at position i in channel c has been included in the embedding.

The monitor process executes an infinite loop that consists of receiving an event on the channel `snoop` and then dealing with it as follows.

- If the new event is a send event $c!m$, the monitor decides (nondeterministically) whether to include the new message in the embedding.

  If it decides not to include the message, there is no further work to be done and the monitor returns to the head of the loop to await the next event on channel `snoop`.

  If the monitor does include the message in the embedding, it performs the following steps:
  - Add the pair $(c, len(c) + 1)$ to the list Marked, where *len* is the built-in Promela function that returns the length of the queue on channel c.
  - Simulate $\mathcal{A}_M$ for one step on the action $c!m$.
- If the new event is a receive event $c?m$, the monitor deterministically performs the following action:
  - For each pair (c', i') in Marked, if $c' = c$ then decrement i'.
  - After the decrement, if the pair $(c, 0)$ appears in Marked, delete it and simulate $\mathcal{A}_M$ for one step on the action $c?m$.

Thus, the monitor decides the fate of each message when it sees the send event. If the message is included in the embedding it is marked and tracked as it progresses through the queue. When it reaches the head of the queue, the corresponding receive event is also included in the embedding.

Note that the state space of the monitor consists of the state space of the DFA $\mathcal{A}_M$ augmented with the list Marked. We can maintain Marked as an array.

A trivial upper bound for the number of entries in Marked is the sum of the capacities of the channels as declared in the Promela specification. A much better upper bound is the maximum of the channel capacities assigned to the states of the DFA $\mathcal{A}_M$ (as described in Section 3). Further, each entry in Marked is bounded by the channel capacities in the Promela specification. Thus, the monitor process always has a bounded state space.

A deterministic monitor As we noted earlier, to check the consistency of a negative scenario M, we need to construct a deterministic monitor for M. An obvious approach is to apply the subset construction to the nondeterministic monitor described above. This will blow up the state space of the monitor by an unacceptable amount since the set of possible states includes all possible configurations of the list Marked.

A more realistic approach is to use a greedy algorithm to discover the shortest embedding of the negative scenario in the system run. To explain this approach we need to establish a result about MSC embeddings.

Recall that an MSC M can be equipped with a partial order $\leq_M$ on the events in M (see Section 3). Let m_1 and m_2 be two messages in M, on channels c_1 and c_2, respectively. We can extend the partial order $\leq_M$ from events to messages, as follows: $m_1 \leq_M m_2$ if $c_1!m_1 \leq_M c_2!m_2$ and $c_1?m_1 \leq_M c_2?m_2$. Notice that if m_1 and m_2 are both messages on the same channel then the FIFO semantics for channels ensures that either $m_1 \leq_M m_2$ or $m_2 \leq_M m_1$.

We can now order embeddings of MSCs. Let $f_1 : M_1 \rightarrow M_2$ and $f_2 : M_1 \rightarrow M_2$ be two embeddings of MSC M_1 into MSC M_2. We say that $f_1 \leq f_2$ if for each message m in M_1, $f_1(m) \leq_{M_2} f_2(m)$.

Theorem 1. *Let M_1, M_2 be MSCs such that M_1 can be embedded into M_2. Then, there is a unique minimum embedding (with respect to $\leq$) of M_1 into M_2.*

Proof Sketch: For any pair of embeddings f_1, f_2, we construct a new embedding f' such that for each message m, $f'(m)$ is the minimum of $f_1(m)$ and $f_2(m)$. Clearly $f' \leq f_1$ and $f' \leq f_2$. To complete the proof, we have to show that f' is indeed an embedding. We omit the details due to lack of space. □

We can now program the monitor process to recognize the minimum embedding of M_1 into M_2 in a greedy manner. The monitor records a finite history of the messages exchanged by the system that it has heard about via the channel `snoop`. This history is recorded as a (possibly partial) MSC in terms of the projections of the MSC onto each process p (see Section 3).

We say that an MSC M is atomic if it cannot be written as a concatenation of smaller MSCs [7]. For instance, in Figure 2, the MSC on the right is atomic, while both MSCs on the left can be decomposed into three atomic MSCs.

Let M be a negative scenario to be matched against the system. Let $M_1 \cdot M_2 \cdots M_k$ be a decomposition of M into atomic MSCs. Then, we can sequentially search for embeddings of the atomic MSCs M_1, M_2, $\ldots$, M_k. Thus, we may assume, without loss of generality, that at each stage we are trying to detect an embedding of an atomic MSC M.

Suppose, then, that we want to check whether the scenario M can be embedded into the (possibly partial) MSC M', where M and M' are both atomic. We may assume that M' consists of only those message types that occur in M—we need not record messages that will never be matched. Our strategy is to check the embedding at the level of sequences, for canonical linearizations of M and M'. To fix a canonical linearization, we specify an arbitrary linear order on the channels. There is then a unique linearization of the events of the MSC such that e precedes e' in the linearization if (i) $e < e'$ in the underlying partial order on events or (ii) e and e' are unordered but the channel on which e occurs is below the channel on which e' occurs or (iii) e is a send event and e' is a (non-matching) receive event along on the same channel.

In [13], a naïve one pass algorithm is used to check the embedding of one MSC in another. Essentially, this algorithm checks that the linearization of the template M is a subsequence of the linearization of the system behaviour M'. The correctness of this algorithm crucially relies on closure with respect to *race conditions* [1]. In this semantics, along a process line, if a send event is immediately followed by a receive event, the two events can also occur transposed. This would imply that a template where two messages cross (for instance, the SYN messages in Figure 2) would be matched by an execution where the first message is received by the second process before it despatches the second message. This does not seem reasonable, so we interpret scenarios literally.

If we do not implicitly close scenarios with respect to race conditions, we need to use backtracking for template matching. Consider Figure 8. Let the channels corresponding to m and m' be c and c', respectively, with c less than c' in the linear order on channels. Then, the canonical linearizations of the two MSCs are $c!m\ c'!m'\ c?m\ c'?m'$ and $c!m\ c!m\ c?m\ c'!m'\ c?m\ c'?m'$. The naïve greedy subsequence algorithm will incorrectly try to match the event $c!m$ from the template on the left to the first occurrence of $c!m$ on the right.

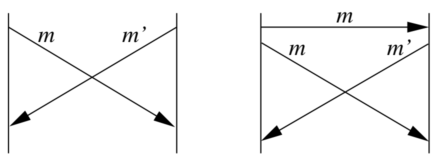

Fig. 8. Template matching requires backtracking

The backtracking algorithm proceeds as follows. Let $\sigma = e_0e_1 \ldots e_n$ be the canonical linearization of the template M and $\sigma' = e'_0e'_1 \ldots e'_\ell$ be the canonical linearization of the system history M' (which may not be a complete MSC). For each index $j \in \{0, \ldots, \ell\}$, we maintain a pointer $\mu(j)$ into the set $\{-1, 0, 1, \ldots, n\}$.

If event e'_j has already been matched to event e_i, then $\mu(j) = i$. Otherwise e'_j is unmatched and $\mu(j) = -1$. Initially, we set $\mu(j) = -1$ for all j.

We now scan σ and σ' from left to right. Assume that we are currently scanning e_i and e'_j and the partial embedding constructed so far is reflected in the values of $\mu(k)$, for $k < j$.

- If $e_i = e'_j = c!m$, set $\mu(j) = i$ and increment both i and j.
- If $e_i = e'_j = c?m$, let $e'_k = c!m$ be the matching send event in σ'. If $\mu(k) \neq -1$, set $\mu(j) = i$ and increment both i and j. Otherwise, set $\mu(j) = -1$ and increment j.
- If $e_i \neq e'_j$ and $e'_j = c'!m'$, set $\mu(j) = -1$ and increment j.
- If $e_i \neq e'_j$ and $e'_j = c'?m'$, let $e'_k = c'!m'$ be the matching send event in σ'. If $\mu(k) = -1$, set $\mu(j) = -1$ and increment j. Otherwise, backtrack by setting i to $\mu(k)$, $\mu(k')$ to -1 for all $k' \in \{k, k+1, \ldots, j\}$ and j to $k+1$.

Thus, backtracking occurs when we try to skip over a receive event whose corresponding send has been matched.

We now describe the deterministic monitor. Each time the monitor receives a new event via `snoop`, it does the following:

- If the message type does not occur in the pattern to be matched, do nothing.
- If the message is a send event, the event is added to the history.
- If the event is a receive event, the event is added to the history and we apply the backtracking algorithm described above to check if the current history embeds M.

  If the algorithm succeeds, we move onto to the next (atomic) MSC to be embedded.

  If the backtracking algorithm fails but the current history is a (complete) atomic MSC, we discard the atomic MSC and start a fresh history.

  If the backtracking algorithm fails but the current history is an incomplete MSC, we can discard any minimal event in the history that was bypassed by the backtracking algorithm *before* reaching the end of the current history. (Since backtracking is deterministic, such an event will *always* be skipped, even after the history is extended.) This prunes the history.

Spin permits `hidden` global variables. The value of such a variable is always undefined when enumerating the state space. Thus, hidden variables do not increase the number of reachable states. All the auxiliary variables used by the monitor for the backtracking algorithm can thus be declared to be `hidden`.

Our tactic for pruning histories does not, per se, guarantee that the history is bounded. It is possible to do a more careful analysis and develop a criterion to discard useless events from the history in such a way that the history size is always bounded. However, it seems more pragmatic to fix a reasonable upper bound on the history size based on the size of the template to be matched and live with the possibility of false negatives rather than add further complexity to the deterministic monitor process.

In the worst-case, our backtracking algorithm takes exponential time. However, in practice we believe that it is relatively efficient because it matches one atomic MSC at a time, and atomic MSCs are generally quite small.

5 Discussion

The monitor processes described here have been constructed by hand for some examples, including the ones described in this paper. However, we still have to automate the process of generating the monitor process directly from the positive and negative scenarios.

It is worth noting that at a theoretical level, the problem we have addressed is relatively straightforward. The system specification S can be modelled as a system of communicating finite-state processes. We can abstract away from internal actions and obtain a corresponding *message-passing automaton* $\mathcal{A}_S$ describing its communication patterns [8]. Similarly, we can represent the single positive and negative MSC scenario specifications by message-passing automata $\mathcal{A}_{pos}$ and $\mathcal{A}_{neg}$, respectively. Checking the positive specification amounts to checking whether the language $L(\mathcal{A}_{pos})$ has a nonempty intersection with $L(\mathcal{A}_S)$, while checking the negative specification amounts to checking whether $L(\mathcal{A}_S)\backslash L(\mathcal{A}_{neg})$ is contained in the set of models $L(\varphi)$ of the property φ.

An important aspect of our work is that our approach to solve the problem uses an existing verification system. The solution builds on existing work in a modular way. At one end, we use the SDL to Promela translation from [4, 3]. At the other end, we use the standard translation from LTL to *never claim* processes in Spin [5, 6]. Our contribution is to augment the Promela specification with a monitor process that synchronizes with every other process in the system. This requires us to modify the Promela code produced by the translation from SDL, but the modification is uniform and and hence not difficult to implement.

Another important contribution is the way we combine branching-time and linear-time specifications, using MSC constraints in conjunction with LTL formulas. In our setup, the MSC constraints describe those runs of the system that are "interesting", which is a branching-time assertion. The LTL formula is then treated as a conventional linear-time specification that has to hold universally along all the selected runs. This method of combining of branching-time and linear-time specifications does not appear to have been studied and seems to be of independent interest.

An interesting question is how to generalize the analysis to the case where the set of scenarios is infinite, but regular (in the sense of Section 3). For positive scenarios, we can still construct a nondeterministic monitor, so checking safety is a simple extension of what is done for finite sets of MSCs. However, the strategy for establishing liveness of positive scenarios no longer works. If we check for the satisfiability of the formula $\neg\Diamond\mathtt{good}$, where the boolean condition $\mathtt{good}$ denotes that one of a set $\mathcal{M}$ of MSCs has been observed, what we capture is a situation where the system cannot execute *any* of the scenarios in $\mathcal{M}$. Thus, the situation where the system can execute some, but not all, of the scenarios in $\mathcal{M}$, would not be caught by this approach. This problem does not appear to admit an obvious solution even at a theoretical level, using automata.

The analysis for negative scenarios is also complicated when we have an infinite set of scenarios. There does not appear to be an obvious way to construct an unambiguous monitor in this case. Without this, as we indicated earlier, the formula $\mathtt{good} \vee \psi$ no longer captures the property that we are trying to check.

Yet another theoretical issue that remains to be resolved is the exact complexity of the problem of detecting when one MSC embeds into another. As we mentioned earlier, a naïve linear-time greedy algorithm is presented in [13], but with respect to a semantics where MSC events may be reordered in the presence of race conditions. Without this relaxation on the order of events, it is not clear that a deterministic polynomial-time algorithm exists.

References

1. R. Alur, G. Holzmann and D. Peled: An analyzer for message sequence charts. *Software Concepts and Tools*, **17(2)** (1996) 70–77.
2. R. Alur and M. Yannakakis: Model checking of message sequence charts. *Proc. CONCUR'99*, LNCS **1664**, Springer-Verlag (1999) 114–129.
3. D. Bosnacki, D. Dams, L. Holenderski and N. Sidorova: Model checking SDL with Spin, *Proc TACAS 2000*, LNCS **1785**, Springer-Verlag (2002) 363–377.
4. M. Bozga, J-C. Fernandez, L. Ghirvu, S. Graf, J.P. Karimm, L. Mounier and J. Sifakis: If: An intermediate representation for SDL and its applications, *Proc. SDL-FORUM '99*, Montreal, Canada, 1999.
5. P. Gastin and D. Oddoux: Fast LTL to Büchi automata translation, *Proc. CAV 2001*, LNCS **2102**, Springer-Verlag (2001) 53–65.
6. R. Gerth, D. Peled, M.Y. Vardi and P. Wolper: Simple on-the-fly automatic verification of linear temporal logic, *Proc PSTV 95*, Warsaw, Poland, Chapman & Hall (1995) 3–18.
7. J.G. Henriksen, M. Mukund, K. Narayan Kumar and P.S. Thiagarajan: On Message Sequence Graphs and Finitely Generated Regular MSC Languages, *Proc. ICALP 2000*, LNCS **1853**, Springer-Verlag (2000) 675–686.
8. J.G. Henriksen, M. Mukund, K. Narayan Kumar and P.S. Thiagarajan: Regular Collections of Message Sequence Charts', *Proc. MFCS 2000*, LNCS **1893**, Springer-Verlag (2000) 405–414.
9. G.J. Holzmann: The model checker SPIN, *IEEE Trans. on Software Engineering*, **23**, 5 (1997) 279–295.
10. ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU, Geneva (1999).
11. ITU-T Recommendation Z.100: *Specification and Description Language (SDL)*. ITU, Geneva (1999).
12. Z. Manna and A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, Berlin (1991).
13. A. Muscholl, D. Peled, and Z. Su: Deciding properties for message sequence charts. *Proc. FOSSACS'98*, LNCS **1378**, Springer-Verlag (1998) 226–242.
14. A. Olson *et al*: *System Engineering using SDL-92*, Elsevier, North-Holland (1997).
15. A. Pnueli: The Temporal Logic of Programs, *Proc. 18th IEEE FOCS* (1977) 46–57.
16. E. Rudolph, P. Graubmann and J. Grabowski: Tutorial on message sequence charts, *Computer Networks and ISDN Systems—SDL and MSC*, Volume **28** (1996).