Model Checking of Message Sequence Charts

Rajeev Alur* and Mihalis Yannakakis**

Abstract. Scenario-based specifications such as message sequence charts (MSC) offer an intuitive and visual way of describing design requirements. Such specifications focus on message exchanges among communicating entities in distributed software systems. Structured specifications such as MSC-graphs and Hierarchical MSC-graphs (HMSC) allow convenient expression of multiple scenarios, and can be viewed as an early *model* of the system. In this paper, we present a comprehensive study of the problem of verifying whether this model satisfies a temporal requirement given by an automaton, by developing algorithms for the different cases along with matching lower bounds.

When the model is given as an MSC, model checking can be done by constructing a suitable automaton for the linearizations of the partial order specified by the MSC, and the problem is coNP-complete. When the model is given by an MSC-graph, we consider two possible semantics depending on the *synchronous* or *asynchronous* interpretation of concatenating two MSCs. For synchronous model checking of MSC-graphs and HMSCs, we present algorithms whose time complexity is proportional to the product of the size of the description and the cost of processing MSCs at individual vertices. Under the asynchronous interpretation, we prove undecidability of the model checking problem. We, then, identify a natural requirement of *boundedness*, give algorithms to check boundedness, and establish asynchronous model checking to be PSPACE-complete for bounded MSC-graphs and EXPSPACE-complete for bounded HMSCs.

1 Introduction

Message sequence charts (MSCs), and related formalisms such as time sequence diagrams, message flow diagrams, and object interaction diagrams, are a popular visual formalism for documenting design requirements for concurrent systems such as telecommunications software [Z.120,RGG96]. MSCs are often used in the first attempts to formalize design requirements for a new system and its protocols. MSCs represent typical execution scenarios, providing examples of either normal or exceptional executions ('sunny day' or 'rainy day' scenarios) of the proposed system. The clear graphical layout of an MSC immediately gives an intuitive understanding of the intended system behavior.

In the simplest form, an MSC depicts the desired exchange of messages, and corresponds to a single (partial-order) execution of the system. In recent

* Department of Computer and Information Science, University of Pennsylvania, and Bell Laboratories, Lucent Technologies. Email: alur@cis.upenn.edu. Supported in part by NSF CAREER award CCR-9734115 and by the DARPA grant NAG2-1214.

** Bell Laboratories, Lucent Technologies. Email: mihalis@research.bell-labs.com.

years, a variety of features have been introduced so that a designer can specify multiple scenarios conveniently. In particular, *MSC-graphs* allow MSCs to be combined using operations such as choice, concatenation, and repetition. *Hierarchical MSCs* (HMSC), also called *high-level MSCs*, allow improved structuring of such graphs by introducing abstraction and sharing. All these features are incorporated in an international standard, called Z.120, promoted by ITU [Z.120]. MSCs or similar formalisms are increasingly being used by designers for specifying requirements. Such specifications are naturally compatible with object-oriented design methods, and are being supported by almost all the modern software engineering methodologies such as SDL [Z.100], ROOM [SGW94] and UML [BJR97].

We believe that scenario-based requirements will play an increasingly prominent role in design of software systems that require communication among distributed agents. Requirements expressed using MSCs (or HMSCs) have a 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 have a high pay-off. This has already motivated development of algorithms for detecting race conditions and timing conflicts [AHP96], pattern matching [MPS98], and detecting non-local choice [BL97], and tools such as uBET [AHP96,HPR97]. In this paper, inspired by the success of model checking in debugging of high-level hardware and software designs [CE81,CK96,Hol97], we develop a methodology and algorithms for model checking of scenario-based requirements.

It is worth noting that the traditional high-level model for concurrent systems has been communicating state machines. Both communicating state machines and HMSCs can be viewed as specifying sets of behaviors, but the two offer dual views; the former is a parallel composition of sequential machines, while the latter is a sequential composition of concurrent executions. Analyzing communicating state machines is known to be computationally expensive—PSPACE or worse, and in spite of the remarkable progress in developing heuristics, still remains the main bottleneck in application of model checking. Consequently, translating MSC-based specifications to communicating state machines, as suggested in previous approaches [LL94,Hol96], may not lead to the most efficient procedures. Also there is a difference in expressive power between the two formalisms in general. The problem of analyzing HMSCs is interesting and important in its own right, and is investigated in this paper.

We formalize the model checking problem using the automata-theoretic approach to formal verification [VW86,Hol97,Kur94]. The system under design is described by an MSC, or a MSC-graph, or an HMSC, in which the individual events are labeled with symbols from an alphabet Σ. The semantics of the system is a language of strings over Σ. The specification is described by an automaton over Σ whose language consists of the undesirable behaviors, and model checking corresponds to checking if the intersection of the two languages is empty. When the system is described by an MSC-graph, or an HMSC, the choice for the associated language depends on the interpretation of the concatenation of two MSCs. We consider two natural choices: in the *synchronous* concatenation of two MSCs M_1 and M_2, any event in M_2 is assumed to happen after all the events in M_1; while the *asynchronous* interpretation corresponds to concatenating two MSCs process by process.

An MSC M specifies a partial ordering of the events it contains, and the model checking problem for M can be solved by constructing an automaton that accepts all possible linearizations of the partial order and checking this automaton against the given specification property automaton. We establish the model checking problem for MSCs to be coNP-complete. For model checking of MSC-graphs under the synchronous interpretation of concatenation, we replace each vertex of the MSC-graph by an automaton that accepts all the linearizations of the associated MSC, construct the product with the specification automaton, and check for emptiness. For HMSCs, a similar strategy reduces the model checking problem to a problem for *hierarchical state machines*, and then, we employ the efficient algorithms of [AY98] for searching the hierarchical structure without flattening it. The resulting complexity for both MSC-graphs and HMSCs is proportional to the size of the system description times the complexity of model checking individual MSCs. In both cases, the model checking problem is proved to be coNP-complete.

Under the asynchronous interpretation for concatenation, the model checking problem for MSC-graphs turns out to be undecidable. The problem can be traced to descriptions which allow unbounded drift between the processes and whose correct implementation requires potentially unbounded buffers. We identify a subclass of *bounded* graphs which rule out such problems and entail decidability. We give an algorithm to check if an MSC-graph or an HMSC is bounded. The boundedness requirement is similar (though not identical) to the condition identified in [BL97] to avoid process divergence. The algorithm in [BL97] to detect divergence is exponential in the number of vertices for flat MSC-graphs (and its straightforward extension to HMSCs is doubly exponential). Our algorithm for checking boundedness extends also to process divergence and is exponential in the number of processes, but linear in the size of the MSC-graph or HMSC. We show the problem of checking boundedness (and process divergence) to be coNP-complete. Finally, we establish that the asynchronous model checking problem is PSPACE-complete for bounded MSC-graphs, and EXPSPACE-complete for HMSCs. In particular, for asynchronous model checking of HMSCs, the flattening of the hierarchy is unavoidable in the worst case (unlike the synchronous model checking and the testing of boundedness, where the flattening can be avoided).

2 Message Sequence Charts

A sample message sequence chart is shown in Figure 1. Vertical lines in the chart correspond to asynchronous processes or autonomous agents. Messages exchanged between these processes are represented by arrows. The tail of each arrow corresponds to the event of sending a message, while the head corresponds to its receipt. Arrows can be drawn either horizontally or sloping downwards, but not upwards. Each arrow is labeled with a message identifier. We proceed to

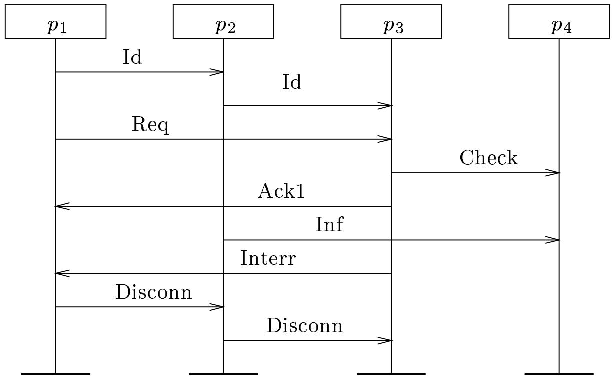

Fig. 1. A sample message sequence chart

define MSCs formally. It is worth noting that our definitions capture the spirit of the standard Z.120, but differ in details and focus only on a subset of the features for the sake of clarity and simplicity.

2.1 Formalization

Formally, a message sequence chart M is a labeled directed acyclic graph with the following components [AHP96]:

- *Processes:* A finite set P of processes.
- *Events:* A finite set E of events that is partitioned into two sets: a set S of send events and a set R of receive events.
- *Process Labels:* A labeling function g that maps each event in E to a process in P. The set of events belonging to a process p is denoted by E_p.
- *Send-receive Edges:* A bijection map $f : S \mapsto R$ that associates each send event s with a unique receive event $f(s)$ and each receive event r with a unique send event $f^{-1}(r)$.
- *Visual Order:* For every process p there is a local total order $<_p$ over the events E_p which corresponds to the order in which the events are displayed.

The local visual orders, together with the send-receive edges, define the relation

$$< = [\cup_p <_p \ \cup \{ \ (s, f(s)) \mid s \in S \ \}]^*.$$

The relation $<$ is a partial order over E since send-receive edges cannot go upwards in the chart. This formalization provides a simple, but precise, way to treat MSCs as mathematical objects. There are many alternative formalizations, for instance, via translation to process algebras [MR94]. Furthermore, the above formalization assumes that the ordering of the receipts of messages at a process coincides with the visual order. Depending on the underlying communication architecture, we may wish to employ alternative orderings, but this choice does not affect the complexity of the problems studied in this paper.

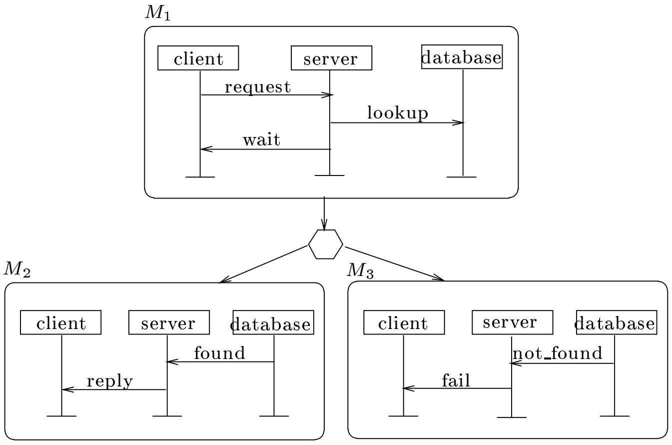

Fig. 2. A sample MSC graph G

2.2 MSC-graphs

A natural way to structure multiple scenarios is to employ graphs whose nodes are MSCs. An *MSC-graph* is a graph whose nodes are labeled with MSCs, and whose edges correspond to concatenation of MSCs. A sample MSC graph is depicted in Figure 2. The first node corresponds to a scenario M_1 in which the server initiates a database lookup to process a client request. The scenario M_1 is followed by either scenario M_2 or by the scenario M_3. The scenario M_2 corresponds to a positive response from the database, while the scenario M_3 models a negative response from the database. The hexagonal box is called a *condition* in the MSC standard, and is used to indicate a *choice* or *branching* in MSC graphs. For the purpose of this paper, conditions will be uninterpreted, and hence, can be ignored in the formalization. Formally, an MSC-graph G consists of a set V of vertices, a binary relation $\rightarrow$ over V, an initial vertex v^I, a terminal vertex v^T, and a labeling function μ that maps each vertex v to an MSC. The paths that start at the initial vertex and end at the terminal vertex represent the finite executions of the system modeled by the MSC-graph, while the infinite executions are represented by all the infinite paths starting at the initial vertex. Note that the definition can be modified to allow multiple terminal vertices without affecting any of the complexity bounds in this paper.

2.3 Hierarchical MSCs

Hierarchical MSCs (HMSC) (also called *high-level MSCs*) offer an improved structuring mechanism. Consider a sample HMSC shown in Figure 3. It is like an MSC-graph, and has three nodes M_i, M_b, and M_f. The nodes M_i and M_f are MSCs as in an MSC-graph, but the node M_b is labeled by another MSC-graph

G. Thus, the node M_b is like a *superstate* in hierarchical state-machines such as Statecharts [Har87]. The MSC M_i depicts the sequence of messages for initialization. As seen earlier, the MSC-graph G depicts the sequence of messages for processing individual requests. After completing one request, either G gets repeated, or the system terminates after executing the termination sequence of messages depicted in M_f. Note that the structure of G is not visible at the top level. In a typical graphical interface, the graph G itself can be viewed by clicking onto the node M_b.

More generally, a hierarchical MSC consists of a graph whose nodes are either MSCs or are labeled with another hierarchical MSC. Thus, the definition allows nesting of graphs, provided the nesting is finite. In other words, the definition of HMSCs cannot be mutually recursive: if a node of an HMSC M is labeled with another HMSC M', then a node of M' cannot be labeled with M (or any other HMSC that refers to M). Another important aspect of the definition is that different nodes can be labeled with the same HMSC. For instance, once we have defined the request-processing scenario G (Figure 2) it can be used multiple times, possibly in different contexts, just like a function in a traditional programming language. This allows reuse and sharing, and leads to succinct representation of complex scenarios.

Formally, a *Hierarchical MSC* is a tuple $H = (N, B, v^I, v^T, \mu, E)$, where

- N is a finite set of nodes.
- B is a finite set of boxes (or supernodes).
- $v^I \in N \cup B$ is the initial node or box.
- $v^T \in N \cup B$ is the terminal node or box.
- μ is a labeling function that maps each node in N to an MSC, and each box in B to another (already defined) HMSC.
- $E \subseteq (N \cup B) \times (N \cup B)$ is the set of edges that connect nodes and boxes to each other.

The meaning of an HMSC H is defined by recursively substituting each box by the corresponding HMSC to obtain an MSC-graph. For an HMSC $H = (N, B, v^I, v^T, \mu, E)$, the flattened MSC-graph H^F is defined as follows. For each box b, let b^F be the MSC-graph $(\mu(b))^F$ obtained by flattening $\mu(b)$. The MSC-graph H^F has following components:

Vertices. Every node of H is a vertex of H^F. For a box b of H, for every vertex v of b^F, the pair (b, v) is a vertex of H^F.

Initial vertex. If $v^I \in N$ then the initial vertex of H^F is v^I. If $v^I \in B$, then if the initial vertex of $(v^I)^F$ is v then the pair (v^I, v) is the initial vertex of H^F.

Terminal vertex. If $v^T \in N$ then the terminal vertex of H^F is v^T. If $v^T \in B$, then if the terminal vertex of $(v^T)^F$ is v then the pair (v^T, v) is the terminal vertex of H^F.

Labeling with MSCs. For a node u of H, the label of u in H^F is same as the label $\mu(u)$. For a box b of H, for every vertex v of b^F, the label of (b, v) in H^F is same as the label of v in b^F.

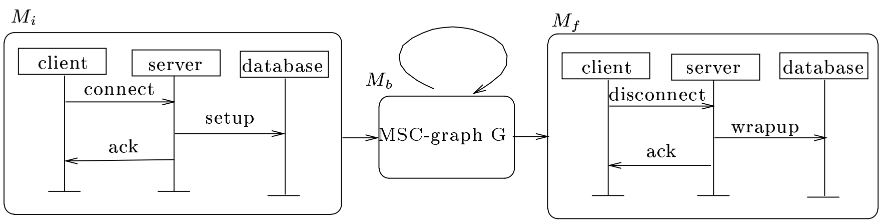

Fig. 3. A sample hierarchical MSC graph

Edges. For an edge (u, v) of H, (u', v') is an edge of H^F, where if $u \in N$ then $u' = u$ else $u' = (u, u'')$ for the terminal vertex u'' of u^F, and if $v \in N$ then $v' = v$ else $v' = (v, v'')$ for the initial vertex v'' of v^F.

Thus, the vertices in the flattened graph are tuples whose last component is a node, and remaining components are boxes that specify the context. The MSC labeling the last component determines the label of a vertex. Note that the number of components in a vertex is bounded by the nesting depth of the description, and the number of vertices can be exponential in the nesting depth.

In our definition, a box can be entered only at its entry vertex, and can be exited only at its terminal vertex. This choice is only for the sake of simplicity of presentation, and we can allow edges connecting to and from specific vertices inside a box without a significant penalty on the complexity of algorithms.

3 Model Checking of MSCs

To formalize the model checking problem, given an MSC M with event-set E, we introduce another component in the MSC-specification, namely, labeling of events in E with symbols in a given alphabet. For an alphabet Σ, a Σ-labeled MSC is a pair (M, ℓ), where M is an MSC and ℓ is a function from E to Σ. A Σ-labeled MSC can be viewed as a *partially-ordered multiset* (POMSET) [Pra86].

Consider a Σ-labeled MSC M with events E and labeling ℓ. Recall that the MSC specifies a partial ordering of the events in E. If we consider all possible linearizations of this partial order, and map each ordering to a string over Σ by replacing each event e by its associated symbol $\ell(e)$, the resulting set of strings is called the *language of* M, and is denoted $L(M)$. Alternatively, we can label the messages in M, or we can label both the events and the messages. Such a choice would not affect the complexity of the model checking algorithms.

For a Σ-labeled MSC M, the language $L(M)$ represents the possible executions of the system. The requirement can be specified by an automaton A over Σ which accepts all the undesirable executions: the system M satisfies the specification A iff the intersection $L(M) \cap L(A)$ is empty. The *model checking problem* for MSCs is, then, given a Σ-labeled MSC M, and an automaton A over Σ, determine whether or not $L(M) \cap L(A)$ is empty.

Let M be an MSC with event set E and partial order $<$. To solve the model checking problem, we can construct an automaton A_M that accepts $L(M)$ using the standard technique of extracting global states from a partial order as follows. A *cut* c is a subset of E that is closed with respect to $<$: if $e \in c$ and $e' < e$ then $e' \in c$. Since all the events of a single process are linearly ordered, a cut can be specified by a tuple that gives the maximal event of each process. The states of the automaton A_M correspond to the cuts. The empty cut is the initial state, and the cut with all the events is the final state. If the cut d equals the cut c plus a single event e, then there is an edge from c to d on the symbol $\ell(e)$. It is easy to verify that the automaton A_M accepts the language $L(M)$. The size of A_M corresponds to the number of cuts, and is bounded by n^k, if M has n events and k processes. The model checking problem with respect to a specification automaton A can now be reduced to a reachability problem over the product of A_M and A.

Theorem 1. *Given a Σ-labeled MSC M with n events and k processes, and an automaton A of size m, the model checking problem (M, A) can be solved in time $O(m \cdot n^k)$, and is coNP-complete* [1].

4 Model Checking of MSC-graphs

For an alphabet Σ, a Σ-labeled MSC-graph G is a graph $(V, \rightarrow, v^I, v^T, \mu)$, where μ maps each vertex to a Σ-labeled MSC. To define the model checking problem for such graphs, we must associate a language with each graph. First, let us note that there is no unique interpretation of the concatenation. As an example, consider the concatenation of two MSCs M_1 and M_2 depicted in Figure 4. Under the *synchronous* interpretation, all the events in the MSC M_1 finish before any event in the MSC M_2 occurs. Thus, the event r_2 is guaranteed to occur before the event s_3. The *asynchronous* interpretation corresponds to concatenating the two MSCs process by process. Thus, the event s_3 will happen after the event s_2, but has no causal relationship to the event r_2. The partial orders of events resulting from these two interpretations are shown in Figure 4.

The synchronous interpretation is closer to the visual structure of the MSC-graph and may be closer to the behavior of the system that the designer of the MSC-graph has in mind. However it has a high implementation cost to enforce (some additional messages must be introduced to ensure that processes do not commence to execute M_2 unless all the events in M_1 have occurred). The asynchronous interpretation is advocated by the standard Z.120. It has no implementation overhead, but it introduces potentially unbounded configurations. We will study both possibilities.

[1] The proofs are omitted due to lack of space. To obtain the full version that includes the proofs, please contact the authors.

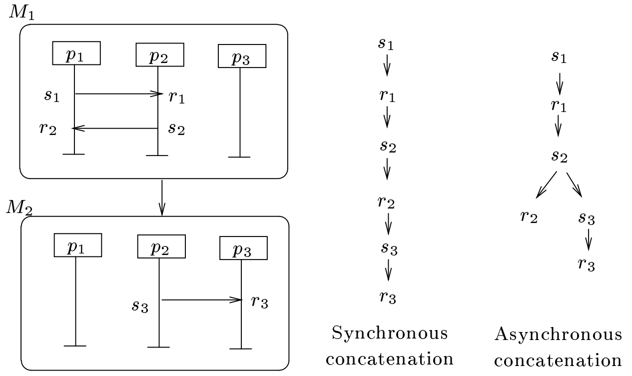

Fig. 4. Two interpretations of concatenation

4.1 Synchronous concatenation

In the synchronous interpretation, the language of concatenation of two MSCs is the concatenation of languages of the component MSCs. For an MSC-graph $G = (V, \rightarrow, v^I, v^T, \mu)$, a *path* is a sequence $\rho = v_0 v_1 \dots v_n$ such that $v_i \rightarrow v_{i+1}$ for $0 \leq i < n$. An *accepting path* is a path $v_0 v_1 \dots v_n$ such that $v_0 = v^I$ and $v_n = v^T$. In a Σ-labeled MSC-graph G, each vertex is mapped to a Σ-labeled MSC, and thus, has a language associated with it. The language of G is obtained by considering concatenation of languages of vertices along accepting paths. Formally, given a Σ-labeled MSC-graph $G = (V, \rightarrow, v^I, v^T, \mu)$, the *synchronous-language* $L^s(G)$ is the set of strings $\sigma_0 \cdot \sigma_1 \cdots \sigma_n$ such that there exists an accepting path $v_0 v_1 \dots v_n$ in G with $\sigma_i \in L(\mu(v_i))$ for $0 \leq i \leq n$.

In the synchronous model checking problem, we are given a Σ-labeled MSC-graph G, and an automaton A over Σ, and we wish to determine whether or not the intersection $L^s(G) \cap L(A)$ is empty. To solve the problem, we construct an automaton A^s_G that accepts the language $L^s(G)$ as follows. Replace each node v of G by the automaton $A_{\mu(v)}$ that accepts the language corresponding to the MSC-label of v. An edge from a vertex u to a vertex v is replaced by edges that ensure concatenation of the languages of $A_{\mu(u)}$ and $A_{\mu(v)}$ (concatenation of automata is a standard operation, and the details are omitted here). If each of the MSCs labeling the vertices of G has at most n events and k processes, then each of the individual automata has at most n^k states. If G has m vertices then A^s_G has at most $m \cdot n^k$ states.

Theorem 2. *Given a Σ-labeled MSC-graph G with m vertices, each of which is labeled with an MSC with at most n events and k processes, and an automaton A of size a, the synchronous model checking problem (G, A) can be solved in time $O(m \cdot a \cdot n^k)$, and is coNP-complete.*

4.2 Asynchronous concatenation

The asynchronous concatenation of two MSCs gives another MSC. Let $M_1 = (P_1, E_1, g_1, f_1, \{<_p^1 | \ p \in P_1\})$ and $M_2 = (P_2, E_2, g_2, f_2, \{<_p^2 | \ p \in P_2\})$ be two MSCs. *The asynchronous concatenation* of M_1 and M_2 is the MSC M defined by

- The set of processes is the union $P_1 \cup P_2$.
- Assuming the two event sets E_1 and E_2 are disjoint, the set of events is the union $E_1 \cup E_2$.
- The process labels stay unchanged: for $e \in E_1$, $g(e) = g_1(e)$ and for $e \in E_2$, $g(e) = g_2(e)$.
- The send-receive edges are unchanged: for $e \in S_1$, $f(e) = f_1(e)$ and for $e \in S_2$, $f(e) = f_2(e)$.
- For $p \in P_1 \setminus P_2$, $<_p$ equals $<_p^1$, and for $p \in P_2 \setminus P_1$, $<_p$ equals $<_p^2$. The ordering of events belonging to a common process $p \in P_1 \cap P_2$ is the concatenation of the component orderings: $<_p$ equals $<_p^1 \cup <_p^2 \cup E_{1_p} \times E_{2_p}$.

The asynchronous concatenation operation extends to Σ-labeled MSCs. To associate a language with a Σ-labeled MSC-graph G under the asynchronous interpretation, we can associate an MSC with each path by asynchronously concatenating MSCs corresponding to individual vertices. The language of the graph is the union of the languages of all such MSCs associated with the accepting paths. Formally, given a Σ-labeled MSC-graph $G = (V, \rightarrow, v^I, v^T, \mu)$, given a path $\rho = v_0 v_1 \ldots v_n$, the Σ-labeled MSC $\mu(v_0) \cdot \mu(v_1) \cdots \mu(v_n)$ is denoted M_ρ. The *asynchronous-language* $L^a(G)$ is the set

$$\{L(M_\rho) \mid \ \rho \text{ is an accepting path in } G\}.$$

Under the asynchronous interpretation, the language of a graph need not be regular. For instance, consider an MSC M containing a single send-receive edge: send-event s by process p_1 followed by a receive-event r by process p_2. In the MSC-graph M^*, under the asynchronous interpretation, process p_1 can send arbitrarily many messages to process p_2 before any message is actually received by process p_2. A key property that contributes to the complexity is the following: the language of the asynchronous concatenation of two MSCs with no processes in common is the shuffle of the languages of the components. This can be exploited to encode computations of Turing machines as shown below. This result strengthens the result in [MPS98], where the intersection of two MSC graphs is shown to be undecidable.

Theorem 3. *The asynchronous model checking problem* (G, A) *for MSC-graphs is undecidable.*

4.3 Bounded MSC-graphs

Given an MSC M with set P of processes, define the *communication graph* H_M of M to be the graph with P as its vertices and with an arc from process p

to process q if p sends a message to q in M. Given an MSC-graph G and a subset S of its vertices, the communication graph H_S of S is the union of the communication graphs of the MSCs corresponding to the vertices in S: the set of vertices of H_S is the set P of all the processes, and there is an arc from process p to process q if p sends a message to q in the MSC $\mu(v)$ for some $v \in S$. For a set S of vertices, we denote by P_S the set of processes that send or receive a message in the MSC of some vertex in S, and call them the active processes of the set S. We call an MSC-graph *bounded* if for every cycle ρ of G, the subgraph of the communication graph H_ρ induced by the set P_ρ of active processes of the cycle is strongly connected. In other words, communication graph H_ρ on all the processes consists of one nontrivial strongly connected component and isolated nodes corresponding to processes that are inactive throughout the cycle.

We proceed to establish that the asynchronous model checking problem for bounded MSC-graphs is decidable. Given a bounded MSC-graph G, we wish to construct an automaton that generates the asynchronous language of G. Basically, the automaton traverses a path in G, and generates a linearization of the MSC obtained by concatenating the MSCs labeling the nodes on the traversed path. Such linearization can be generated by letting, at every step, one of the processes execute its next step. Due to the asynchronous nature of concatenation, the processes can drift, that is, even before all the events in the MSC corresponding to one node are executed, some processes may proceed to the next node. If we could show that processes can drift apart only by a finite distance, say, bounded by the number of nodes in the graph, then it would follow that it suffices for the automaton to remember only a finite suffix of the path. Unfortunately, this does not hold. For example, the processes may be partitioned into two disjoint sets Q and Q' such that all the processes in Q' "overtake" all the processes in Q, and proceed to execute a cycle, possibly multiple times, in which all the processes in Q are inactive. Furthermore, the processes in Q may traverse paths in which all processes in Q' are inactive while processes in Q are active, thus, imposing constraints on what the processes in Q should do in future. In the sequel, we will show that remembering only a finite amount of information suffices even if unbounded intervals of the path are of relevance. To get some intuition for the detailed construction, consider the scenario just described in which processes in Q', after overtaking Q, traverse a path that alternates between intervals in which only processes in Q are active and intervals in which only processes in Q' are active. First, due to the definition of boundedness, the number of such alternations is bounded (otherwise, there would be a cycle with two nontrivial strongly connected components in the communication graph). Second, while the individual intervals can be unbounded, it suffices to remember only the end-points of each interval (in fact, only the end-points of the intervals in which only processes in Q are active). It is worth noting that the construction would be simpler if we had used a weaker definition of boundedness which would require the communication graph of each cycle to be a single strongly connected component. However, allowing inactive processes in cycles seems important to us.

Let G be a bounded MSC-graph. Consider a path $\rho = v_0, v_1, \ldots$ through the graph, and its corresponding MSC M_ρ. Consider some linearization of M_ρ and a prefix σ of the linearization. That is, σ is the set of events executed up to some point in time. We can partition the nodes of ρ into three classes with respect to σ as follows. A node is a *past* node if all the events of the MSC of that node are already executed in the prefix σ, a *present* node if some but not all the events of the MSC of that node are executed, and a *future* node if no events corresponding to that node are executed yet. Since the MSC of each node contains at least one event (this can be assumed without loss of generality), each node of the path gets classified uniquely. Note that a node of G may occur more than once in the path and different occurrences may be classified differently.

Lemma 1. *Consider a subpath of ρ from node v_i to v_j such that the MSC-graph contains a "back" arc from v_j to v_i. Then either (i) all nodes of ρ from v_i to v_j (inclusive) are past, or (ii) all nodes of ρ from v_i to v_j (inclusive) are future, or (iii) there is a process p whose last executed step and next unexecuted step are both from the nodes $v_i, \ldots v_j$.*

We define a *configuration* as a tuple consisting of the following components:

1. A sequence of (not necessarily distinct) nodes $u_1, ..., u_t$ of the MSC-graph G, such that no node occurs more than k times, where k is the number of processes.
2. A mapping from each process p to one of the nodes u_i in the sequence and to a position in the process line of p in the MSC $\mu(u_i)$ (of course if p is not active in u_i, then this last part is vacuous).
3. For every $i = 1, ...t-1$, a bit b_i corresponding to the pair $[u_i, u_{i+1}]$.

Given a path ρ and a prefix σ of a linearization of it, we can define a configuration as follows.

1. The sequence of nodes $u_1, \ldots u_t$ consists of all the present nodes, those past nodes that are adjacent to future nodes (if the first and last node of a contiguous segment of past nodes are occurrences of the same node of the MSC graph, then we only need to keep one copy of the node), and the last node of the path ρ if it is a past node; these past nodes will be needed later to fill in the future nodes consistently. It follows from the lemma that every node v of the MSC-graph occurs at most k times in the recorded sequence u_i: If a node v occurs $k+1$ times in the path ρ, then all k processes must be executing steps in the previous k intervals from v to v, and therefore all nodes of the path ρ up to and including the $(k+1)$th last occurrence of v must be past nodes and are not selected.
2. For the second component of the configuration, map every process p to the node of the path ρ that contains the last step executed by p, and to the corresponding event of the MSC, if the node is one of the selected nodes u_i; if the node is not among the u_i's, then map p to the earliest subsequent u_i. We call this node the current node of p.

3. For the third component, if the subpath of ρ between u_i and u_{i+1} does not contain any future nodes, then set $b_i = 0$; if it contains some future nodes, then set $b_i = 1$. Note in the latter case that the subpath of ρ consists in fact entirely of a sequence of future nodes bordered by u_i and u_{i+1} (which are past or present nodes).

It is clear from the above derivation of a configuration from a partial execution that it satisfies several consistency and nonredundancy conditions. We call a configuration *legal* if it satisfies the following conditions. The mapping of the processes in the second component of a configuration induces a cut of the MSC formed by the concatenation of the MSCs $\mu(u_i)$ of the sequence of nodes u_i in the first component; i.e. if a node u_i contains a message from process p to process q, and process p is mapped before u_i or at u_i before the sending of the message, then process q is mapped before u_i or at u_i before the reception of the message. Based on the mapping of the processes we can classify the selected nodes u_i of the configuration as past, present or future. Then every node u_i of the sequence is past or present; if a node u_i is past then either $b_i = 1$ or $b_{i-1} = 1$; furthermore, if $b_{i-1} = 0$, then u_{i-1} and u_i are not occurrences of the same node of the MSC graph. If (u_i, u_{i+1}) is not an arc of the graph, then either $b_i = 1$ and there is a path in the MSC graph from u_i to u_{i+1} using only future nodes (i.e. nodes of the graphs whose MSCs involve only processes that are mapped at or before u_i), or $b_i = 0$ and there is a path from u_i to u_{i+1} using only past nodes (i.e. nodes of the graph whose MSCs involve only processes that are mapped after u_i). An obvious upper bound on the number of possible configurations is $(km)!2^{km}(mnk)^k$. The following lemma gives a better upper bound on the number of legal configurations.

Lemma 2. *The number of legal configurations is no more than* $2^{(k-1)m} \cdot (mnk)^k$*, where k is the number of processes, m is the number of vertices of G and n is the maximum number of events in a basic MSC of a vertex.*

To solve the asynchronous model checking problem, given a bounded Σ-labeled MSC-graph G, we construct an automaton A_G^a that accepts the language $L^a(G)$. The states of A_G^a are all the legal configurations. The initial state is the configuration with one node u_0, the initial node of the MSC-graph, and all processes are mapped to it, at the beginning of their process lines. The accepting state is the configuration with one node u_T, the terminal node of the MSC-graph, and all processes mapped to it at the end of their process lines. There are transitions representing the update of the configuration by execution of a single event. In addition we have $\epsilon-$transitions that allow the addition of new nodes in the middle or the end of the sequence, the removal of nodes that are not needed any more, the advancement of a process (once it is finished with the steps of a node) and so forth.

In practice of course we will construct the automaton on the fly, generating states as needed. The automaton A_G^a which accepts the linearizations of the MSC-graph G has size at most $O(2^{km} \cdot (mnk)^k)$. This leads to the following bound.

Theorem 4. *Given a bounded Σ-labeled MSC-graph G on k processes with m vertices, each of which is labeled with an MSC with at most n events, and an automaton A of size a, the asynchronous model checking problem (G, A) can be solved in time $O(a \cdot 2^{km} \cdot (mnk)^k)$.*

A precise bound on the complexity is PSPACE:

Theorem 5. *The asynchronous model checking problem (G, A) for bounded MSC-graphs is* PSPACE*-complete. Furthermore, the* PSPACE*-hardness holds even if we bound the number of processes and the number of events in individual MSCs, and even for a fixed property.*

Finally, we address the problem of determining if a given MSC-graph is bounded. For an MSC-graph with process set P, a subset Q of processes is said to be a *witness* for unboundedness if there exists a cycle ρ such that in the MSC M_ρ, no process in Q sends a message to a process in $P \setminus Q$, and there is a process of Q and a process of $P \setminus Q$ that are active (perform some step) in ρ. Verify that if G is not bounded, then some subset must be a witness to the unboundedness.

Whether a given set Q of processes is a witness for unboundedness can be checked in linear time as follows. Remove from G all vertices v such that in the MSC $\mu(v)$ some process in Q sends a message to some process in $P \setminus Q$, and let G' be the resulting graph. Find the strongly connected components of G'. If for some strong component C of G' the corresponding set P_C of active processes intersects both Q and $P \setminus Q$, then Q is a witness for unboundedness, and G is not bounded.

Theorem 6. *Given an MSC-graph G on k processes with m vertices, each of which is labeled with an MSC with at most n events, checking whether G is bounded can be solved in time $O(m \cdot n \cdot 2^k)$, and is coNP-complete.*

A requirement similar to boundedness was identified in [BL97] in the context of *process divergence*, a situation in which a process sends a message an unbounded number of times ahead of a receiving process (thus requiring unbounded buffers). The condition for absence of divergence is that for every cycle ρ of the MSC-graph G, the transitive closure of the communication graph H_ρ is symmetric. This is equivalent to the requirement that every weakly connected component of H_ρ be strongly connected (thus, every bounded MSC-graph is divergence-free, but not necessarily vice-versa.) The algorithm given in [BL97] for process-divergence requires checking each cycle, and thus, is exponential in the number of vertices in G, and no lower bound was given. We can use the same approach as for boundedness to give an algorithm for process divergence that is exponential only in the number of processes, and we can also show a lower bound along the same lines.

Theorem 7. *Given an MSC-graph G on k processes with m vertices, each of which is labeled with an MSC with at most n events, checking G for process divergence can be solved in time $O(m \cdot n \cdot 2^k)$, and is coNP-complete.*

5 Model Checking of HMSCs

For an alphabet Σ, a Σ-labeled HMSC $H = (N, B, v^I, v^T, \mu, E)$ is like an HMSC, where μ maps nodes to Σ-labeled MSCs. By flattening a Σ-labeled HMSC H, we obtained a Σ-labeled MSC-graph H^F. Depending on whether the interpretation of concatenation is synchronous or asynchronous, we get two languages associated with H: the synchronous language $L^s(H)$ and the asynchronous language $L^a(H)$.

In the synchronous model checking problem for HMSCs, we are given a Σ-labeled HMSC H, and an automaton A over Σ, and we wish to decide if $L(A) \cap L^s(H)$ is empty. For this purpose, we translate H into a *hierarchical Kripke structure* [AY98] by replacing each atomic node v in H, and recursively in every HMSC associated with the boxes of H, by the automaton $A_{\mu(v)}$, and replace edges by the edges that ensure concatenation of the languages. The resulting hierarchical Kripke structure A^s_H captures the language $L^s(H)$, and model checking of H reduces to model checking of A^s_H, which can be solved using the algorithms of [AY98] without flattening the hierarchy.

Theorem 8. *Given a Σ-labeled HMSC H of size m, each of which nodes is labeled with an MSC with at most n events and k processes, and an automaton A of size a, the synchronous model checking problem (H, A) can be solved in time $O(m \cdot a^2 \cdot n^k)$, and is coNP-complete.*

The asynchronous model checking problem for HMSCs is, given a Σ-labeled HMSC H and an automaton A, determine if $L(A) \cap L^a(H)$ is empty. Since the problem is undecidable even for MSC-graphs, we will consider only bounded HMSCs: an HMSC is bounded if the flattened MSC-graph H^F is bounded. The asynchronous model checking problem (H, A) can be solved by first constructing the MSC-graph H^F, and then using the model checking algorithm for the bounded MSC-graphs. If the size of H is m, and its nesting depth is d, the size of H^F is $O(m^d)$. If each of the MSCs has at most n events and k processes, and A has a vertices, the resulting time bound for model checking is $O(a \cdot 2^{m^d k} \cdot (m^d nk)^k)$. A precise bound on the complexity is exponential-space:

Theorem 9. *The asynchronous model checking problem (H, A) for bounded HMSCs is* EXPSPACE*-complete.*

To determine if a given HMSC is bounded or not, for every process-set Q, we need to check if Q is a witness for unboundedness. This reduces to detecting cycles of a specific form in the hierarchical graph, and using the algorithms described in [AY98], can be done in linear time. A similar algorithm can be used for checking process divergence.

Theorem 10. *Given an HMSC H of size m, each of which is labeled with an MSC with at most n events and k processes, checking whether H is bounded can be solved in time $O(m \cdot n \cdot 2^k)$, and is coNP-complete.*

Acknowledgements. We thank Anca Muscholl for helpful comments.

References

[AHP96] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[AY98] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proc. Sixth ACM FSE*, 175–188, 1998.

[BJR97] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.

[BL97] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. of TACAS*. 1997.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71, 1981.

[CK96] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[FKL91] J. Feigenbaum, J. A. Kahn, and C. Lund. Complexity results for pomset languages. In *Proc. CAV*, 1991.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hol96] G.J. Holzmann. Early fault detection tools. *Software Concepts and Tools*, 17(2):63-69, 1996.

[Hol97] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[HPR97] G.J. Holzmann, D.A. Peled, and M.H. Redberg. Design tools for for requirements engineering. *Lucent Bell Labs Technical Journal*, 2(1):86–95, 1997.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.

[LL94] P. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 3, 1994.

[LP97] V. Levin, and D. Peled. Verification of message sequence charts via template matching. In *Proc. TAPSOFT*, 1997.

[MPS98] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Found. of Software Science and Computation Structures*, 1998.

[MR94] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *Computer Journal*, 37, 1994.

[Pra86] V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.

[RGG96] E. Rudolph, P. Graubmann, and J. Gabowski. Tutorial on message sequence charts. In *Computer Networks and ISDN Systems*, volume 28. 1996.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-time object oriented modeling and design*. J. Wiley, 1994.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First LICS*, pages 332–344, 1986.

[Z.100] CCITT Specification and Description Language (SDL). ITU-T, 1994.

[Z.120] Message Sequence Charts (MSC'96). ITU-T, 1996.