CONCUR 2003, 14th International Conference on Concurrency Theory
Proceedings: Roberto Amadio, Denis Lugiez (eds.)
Springer Lecture Notes in Computer Science **2761** (2003), 296–311

Netcharts: Bridging the gap between HMSCs and executable specifications

Madhavan Mukund[1], K. Narayan Kumar[1] and P.S. Thiagarajan[2]

[1] Chennai Mathematical Institute, 92 G N Chetty Road, Chennai 600 017, India
Email: {madhavan,kumar}@cmi.ac.in
[2] School of Computing, National University of Singapore, Singapore
Email: thiagu@comp.nus.edu.sg

Abstract. We define a new notation called *netcharts* for describing sets of message sequence chart scenarios (MSCs). Netcharts correspond to a distributed version of High-level Message Sequence Charts (HMSCs). Netcharts improve on HMSCs in two respects.

(i) Netcharts admit a natural and direct translation into communicating finite-state machines, unlike HMSCs, for which the *realization* problem is nontrivial.

(ii) Netcharts can describe all regular MSC languages (sets of MSCs in which channel capacities have a finite upper bound), unlike HMSCs, which can only describe finitely-generated regular MSC languages.

1 Introduction

Message sequence charts (MSCs) are an appealing visual formalism used to capture system requirements in the early stages of design. They are particularly suited for describing scenarios for distributed telecommunication software [10, 15]. They also appear in the literature as sequence diagrams, message flow diagrams and object interaction diagrams and are used in a number of software engineering methodologies including UML [3, 7, 15]. In its basic form, an MSC depicts the exchange of messages between the processes of a distributed system along a single partially-ordered execution. A collection of MSCs is used to capture the scenarios that a designer might want the system to exhibit (or avoid).

Given the requirements in the form of MSCs, one can hope to discover errors at the early stages of design. One question that naturally arises in this context is: What constitutes a reasonable collection of MSCs on which one can hope to do formal analysis? In [9], the notion of a regular collection of MSCs is introduced and shown to be robust in terms of logical and automata-theoretic characterizations. In particular, regular collections of MSCs can be implemented using a natural model of message-passing automata with bounded queues [12].

A standard way to specify a set of MSCs is to use a High-level Message Sequence Chart (HMSC) [10, 11]. An HMSC is a finite directed graph in which each node is itself labelled by an HMSC, with a finite level of nesting.

HMSCs are an attractive visual formalism for describing collections of MSCs. An HMSC is not, however, an executable model. To obtain one, one must extract from the inter-object specification provided by an HMSC, an *intra-object* specification (say in the form of a finite state automaton) for each process together with a communication mechanism. This is non-trivial because the structure of an HMSC provides implicit global control over the behaviour of the processes in the system. This allows for specifications that are not realizable in practice because of un-implementable global choices. Even for the class of so-called bounded HMSCs [2], the problem of realizing them as, say, a network of finite state automata communicating via bounded queues is non-trivial. Admittedly, the sub-class of bounded HMSCs that are *weakly realizable* can implemented easily, but this is an undecidable property of bounded HMSCs! [1]. A detailed study of the realization problems associated with HMSCs can be found in [4].

HMSCs also have a limitation with respect to expressiveness—they can only describe finitely generated MSC languages. But there are natural regular MSC languages, such as the set of scenarios corresponding to the alternating bit protocol, that are not finitely generated [6].

In this paper, we introduce a new visual formalism for specifying collections of MSCs, called *netcharts.* Netcharts are distributed versions of HMSCs in much the same way that Petri nets are distributed versions of finite-state automata. Netcharts improve on HMSCs in two respects. First, due to their natural — control flow — semantics, netcharts can be directly translated into communicating finite-state machines. In this sense the "realization" problem for netcharts is easy. Second, netcharts can describe all regular MSC languages, including those that are not finitely-generated.

An important aspect of the netchart model is that the compound MSCs that are defined by a netchart are built up from "complete" MSCs. As a result, this formalism can be used to capture system requirements in an intuitive fashion. The distributed structure allows these scenarios to be intertwined together to form complicated new scenarios.

The netchart model is related to Communication Transaction Processes (CTP) [14]. The main difference is that in a CTP, each transition in the high-level control flow model is labelled as a guarded choice of MSCs, where the guards are formed using atomic propositions associated with the processes. The focus in the study of the CTP model is the modelling of complex non-atomic interactions between communicating processes and not on the collections of MSCs that are definable. A second piece of related work is [16] in which it is suggested that the transitions of a 1-safe Petri net be labelled by arbitrary MSCs. The realization problem for this model is also difficult and the authors propose a number of restrictions for their model and study the realization problem in terms of composing a fixed set of basic Petri net templates to obtain a Petri net model.

The paper is organized as follows. After some background on MSCs and HMSCs, we introduce the netchart model in Section 3. In Section 4, we show how to transform netcharts into message-passing automata. Next, we discuss when a netchart definable MSC language is regular. Section 6 compares the expressive power of HMSCs and netcharts. In Section 7, we show that netcharts can describe all regular MSC languages.

2 Background

2.1 Message sequence charts

Throughout this paper, we let $\mathcal{P}$ denote a finite set of processes. A *message sequence chart (MSC)* over $\mathcal{P}$ is a graphical representation of a pattern of interactions (often called a *scenario*) consisting of an exchange of messages between the processes in $\mathcal{P}$. In an MSC, each process is represented by a vertical line with time flowing from top to bottom. Messages are drawn as directed edges from the source process to the target process, with an annotation denoting the message type, if any. In addition, internal events are marked along the process lines.

Figure 1 is an MSC involving processes $\{p_1, p_2, p_3\}$. In this scenario, clients p_1 and p_3 both request a resource from the server, p_2. The server p_2 first responds to p_3. After p_3 notifies p_2 that it has returned the resource, p_3 hands over the resource to p_1, but this message crosses a "reminder" from p_1 to p_3.

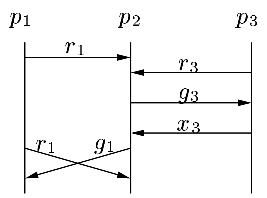

Fig. 1. An MSC

The semantics of an MSC is given in terms of labelled partial orders. Formally, an MSC over $\mathcal{P}$ is a structure $M = (E, \leq, \tau, \varphi, \pi)$, where:

- E is a finite set of *events*.
- $\pi : E \to \mathcal{P}$ associates a unique process with each event.
- $\tau : E \to \{\mathsf{send}, \mathsf{receive}, \mathsf{internal}\}$ specifies the nature of each event.
- φ is a bijection from the set of send events $\{e \mid \tau(e) = \mathsf{send}\}$ to the set of receive events $\{e \mid \tau(e) = \mathsf{receive}\}$. Each pair in the set $\{(e, \varphi(e)) \mid \tau(e) = \mathsf{send}\}$, constitutes a *message*. Messages are often labelled using finite set Δ of message types, with typical elements m, m' etc.
- E is partially ordered by $\leq$, such that for each $p \in \mathcal{P}$, the set $E_p = \{e \mid \pi(e) = p\}$ is linearly ordered by $\leq$. Let us denote this linear order $\leq_p$. Further, $\leq$ is the reflexive, transitive closure of $\bigcup_{p \in \mathcal{P}} \leq_p \cup \{(e, \varphi(e)) \mid \tau(e) = \mathsf{send}\}$.
- Messages between each pair of processes are delivered in a FIFO fashion.[3] More formally, let e, e' be a pair of events such that $\pi(e) = \pi(e')$, $\tau(e) = \tau(e') = \mathsf{send}$ and $\pi(\varphi(e)) = \pi(\varphi(e'))$. Then, $e \leq e'$ iff $\varphi(e) \leq \varphi(e')$.

Let M be a message sequence chart. The *type* of M is the set $\pi(M) = \{p \in \mathcal{P} \mid \exists e \in M.\ \pi(e) = p\}$ of processes that are *active* in M.

[3] We will see how to relax this restriction later.

For $X \subseteq \mathcal{P}$, $\mathcal{M}_X$ denotes the set of MSCs of type X. The set of all possible MSCs over the set of processes $\mathcal{P}$ is given by $\mathcal{M} = \bigcup_{\emptyset \neq X \subseteq \mathcal{P}} \mathcal{M}_X$.

2.2 Regular MSC languages

A communicating system is normally specified using a set of scenarios or, equivalently, a set of MSCs. An *MSC language* is a (finite or infinite) collection of MSCs. We can also represent MSC languages in terms of word languages.

Each send or receive event e in an MSC is characterized by the values $\pi(e)$, $\tau(e)$ and the message type, if any. Let $p!q(m)$ denote the action associated with sending message m from p to q and let $q?p(m)$ denote the corresponding receive action. We then have an alphabet of *communication actions* $\Gamma = \{p!q(m) \mid p, q \in \mathcal{P}, m \in \Delta\} \cup \{p?q(m) \mid p, q \in \mathcal{P}, m \in \Delta\}$. For an MSC M, viewed as Γ- labelled poset, the set $lin(M)$ of valid linearizations of M describes a word language over Γ. An MSC language $L \subseteq \mathcal{M}$ can thus be represented by the word language $\bigcup\{lin(M) \mid M \in L\}$. Notice that internal events do not play any role in defining MSC languages.

We say that an MSC language L is *regular* if the corresponding word language over Γ is regular.

Let $M = (E, \leq, \tau, \varphi, \pi)$ be an MSC. A configuration is a set of events $c \subseteq E$ such that c is closed with respect to $\leq$. The channel capacity of c, $\chi(c)$, specifies the number of unmatched send events in c—that is, $\chi(c) = |\{e \in c \mid \pi(e) = \mathsf{send}, \varphi(e) \notin c\}|$. We define the channel capacity of the MSC M to be maximum value of $\chi(c)$ over all configurations c of M. The following is easy to show.

Proposition 1. *Let L be a regular MSC language. Then, there is a uniform upper bound $B \in \mathbb{N}_0$ such that for every MSC $M \in L$, $\chi(M) \leq B$.*

The converse statement is not true, in general. We shall see a counterexample.

2.3 High-level message sequence charts

The standard mechanism to generate an MSC language is a High-level Message Sequence Chart (HMSC). A simple type of HMSC is a Message Sequence Graph (MSG). An MSG is a finite, directed graph with designated initial and terminal vertices. Each vertex in an MSG is labelled by an MSC. The collection of MSCs represented by an MSG consists of all those MSCs obtained by tracing a path in the MSG from an initial vertex to a terminal vertex and concatenating the MSCs that are encountered along the path.

An HMSC is just an MSG in which a vertex can, in turn, be labelled by another HMSC, with the restriction that the overall nesting depth be finite. Thus, an HMSC can always be "flattened out" into an MSG. Henceforth, when we use the term HMSC we always assume the flat structure of an MSG.

The edges in an HMSC represent the natural operation of MSC concatenation which can be defined as follows. Let $M_i = (E_i, \leq_i, \tau_i, \varphi_i, \pi_i)$, $i \in \{1, 2\}$, be a pair for MSCs such that $E_1 \cap E_2 = \emptyset$. The *(asynchronous) concatenation* of M_1 and M_2 is the MSC $M_1 \circ M_2 = (E, \leq, \tau, \varphi, \pi)$ where $E = E_1 \cup E_2$, $\pi(e) = \pi_i(e)$ and $\tau(e) = \tau_i(e)$ for $e \in E_i$, $i \in \{1, 2\}$, $\varphi = \varphi_1 \cup \varphi_2$ and $\leq$ is the reflexive, transitive closure of $\leq_1 \cup \leq_2 \cup \{(e, e') \mid e \in E_1, e' \in E_2, \pi_1(e) = \pi_2(e')\}$.

Formally, an HMSC is a structure $\mathcal{H} = (S, \longrightarrow, S_{in}, F, \Phi)$, where:

- S is a finite and nonempty set of states.
- $\longrightarrow \subseteq S \times S$.
- $S_{in} \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of final states.
- $\Phi : S \to \mathcal{M}$ is a (state-)labelling function.

A *path* $\sigma = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n$ through an HMSC $\mathcal{H}$ generates the MSC $M(\sigma) = \Phi(s_0) \circ \Phi(s_1) \circ \cdots \circ \Phi(s_n)$. A path $\sigma = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n$ is a *run* if $s_0 \in S_{in}$ and $s_n \in F$. The language of MSCs accepted by $\mathcal{H}$ is $L(\mathcal{H}) = \{M(\sigma) \in \mathcal{M} \mid \sigma \text{ is a run through } \mathcal{H}\}$.

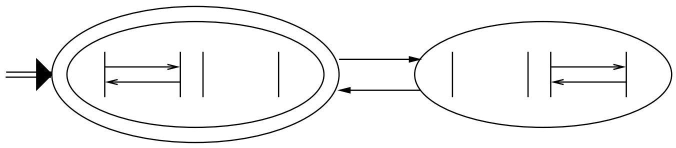

Fig. 2. An HMSC

Figure 2 is an example of an HMSC. The initial state is marked with an incoming arrow and the final state is marked with a double outline. It is easy to see that the language $\mathcal{L}$ defined by this HMSC is *not* regular. Though all the channels are uniformly bounded, there is a "context-free" correlation between the number of iterations of two disjoint sets of communications.

A sufficient criterion for an HMSC to generate a regular MSC language is for it to be *locally synchronized* [13].[4]

For an MSC $M = (E, \leq, \tau, \varphi, \pi)$, *the communication graph of* M is the directed graph $CG_M = (\mathcal{P}, \mapsto)$ where $\mathcal{P}$ is the set of processes of the system and $(p, q) \in \mapsto$ if p sends a message to q in M. More formally, $(p, q) \in \mapsto$ if there exists an $e \in E$ with $\pi(e) = p$, $\tau(e) = \mathsf{send}$ and $\pi(\varphi(e)) = q$.

The MSC M is said to be *connected* if CG_M consists of one nontrivial strongly connected component and isolated vertices. An MSC language $L \subseteq \mathcal{M}$ is connected in case each MSC $M \in L$ is connected.

The HMSC $\mathcal{H}$ is *locally synchronized* if for every loop $\sigma = q \to q_1 \to \cdots \to q_n \to q$, the MSC $M(\sigma)$ is connected. Every locally synchronized HMSC defines a regular MSC language [2] (though the converse does not hold). The HMSC in Figure 2 is *not* locally synchronized.

[4] This notion is called "bounded" in [2].

2.4 Finitely generated MSC languages

Every MSC generated by an HMSC is, by definition, a concatenation of the MSCs that label the vertices of the HMSC. We say an MSC M is *atomic* if M cannot be written as the concatenation $M_1 \circ M_2$ of two nonempty MSCs. An MSC language L is said to be *finitely generated* if there is a finite set of atomic MSCs $Atoms = \{M_1, M_2, \ldots, M_k\}$ such that every MSC $M \in L$ can be decomposed as $M_{i_1} \circ M_{i_2} \circ \cdots \circ M_{i_\ell}$, where each $M_{i_j} \in Atoms$.

Clearly, every HMSC language is finitely generated. However, there exist regular MSC languages that are *not* finitely generated [6, 9]. Every finitely generated regular MSC language can be represented as a locally synchronized HMSC [8].

3 Netcharts

Netcharts are distributed versions of HMSCs in much the same way that Petri nets are distributed versions of finite-state automata. (We refer the reader to [5] for basic concepts and results on Petri nets.)

3.1 Nets

- A net is a triple (S, T, F) where S is a set of *places* (or local states), T is a set of *transitions* and $F \subseteq (S \times T) \cup (T \times S)$ is the *flow relation*. For an element $x \in S \cup T$, we use ${}^\bullet x$ and $x^\bullet$, as usual, to denote the immediate predecessors and successors, respectively, of x with respect to the flow relation F.
- An S-net is a net (S, T, F) in which $|{}^\bullet t| = 1 = |t^\bullet|$ for every $t \in T$.

At the top level, a netchart has the structure of a safe Petri net that can be decomposed in a natural way into sequential components. Transitions synchronize distinct components and are annotated by MSCs involving those components.

3.2 The Netchart Model

A *netchart* is a structure $((S, T, F), M_{in}, loc, \lambda)$ where:

(i) (S, T, F) is a net.
(ii) $M_{in} \subseteq S$ is the *initial marking*.
(iii) The function loc maps each state $s \in S$ to a process in $\mathcal{P}$. For $s \in S$, we refer to $loc(s)$ as the *location* of s. For $p \in \mathcal{P}$, let $S_p = \{s \mid loc(s) = p\}$
(iv) For $t \in T$, let $loc(t) = \{loc(s) \mid s \in {}^\bullet t \cup t^\bullet\}$. The labelling function λ associates an MSC M with each transition t such that $loc(t) = \pi(M)$.
(v) We impose the following restriction on the structure of a netchart:
  - For each $t \in T$, for each $p \in loc(t)$, $|{}^\bullet t \cap S_p| = |t^\bullet \cap S_p| = 1$. In other words, each transition t has exactly one input place and one output place for each process that participates in t.
  - For each $p \in \mathcal{P}$, $|M_{in} \cap S_p| = 1$. In other words, M_{in} places exactly one token in each component S_p.

– For $p \in \mathcal{P}$, let $T_p = \{t \mid p \in loc(t)\}$ denote the set of transitions that p participates in. For each $p \in \mathcal{P}$, the net (S_p, T_p, F_p), where $F_p = F \cap ((S_p \times T_p) \cup (T_p \times S_p))$ is an S-net.

Figure 3 shows an example of a netchart. A more elaborate example modelling the alternating bit protocol is shown in Figure 6.

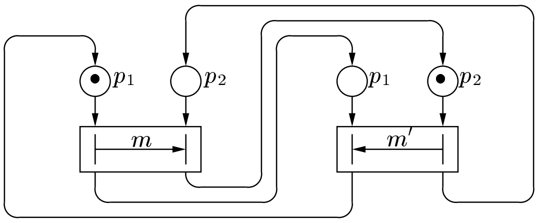

Fig. 3. A netchart

3.3 Semantics

The operational semantics of a netchart is obtained by converting each MSC that labels a transition into a Petri net and "plugging in" these nets into the top-level safe net of the netchart.

The crucial feature of our semantics is that each MSC that is used to label a high-level transition has a private set of channels. Thus, if a message m is sent from p to q in the MSC labelling transition t, it can only be read when q participates in the same high level transition t.

We convert an MSC M into a net (S_M, T_M, F_M) in an obvious way. The set of transitions T_M corresponds to the events of the MSC. Since each process is sequential, we insert a place between every adjacent pair of events along each process line. In addition, for each pair of processes (p, q), we introduce a buffer place $b_{(p,q)}$. For each send event of a message from p to q, the corresponding transition in T_M feeds into the place $b_{(p,q)}$. For each receive event of a message from p to q, the corresponding transition in T_M has $b_{(p,q)}$ as an input place.

In the netchart, for a high-level transition t labelled by the MSC M , for each process $p \in loc(t)$, we connect the transition corresponding to the $\leq_p$-minimum p-event in the MSC M to the (unique) place in S_p that feeds into t. Similarly, we connect the transition corresponding to the $\leq_p$-maximum p-event in the MSC M to the (unique) output place in S_p of t. Observe that we do not need to model the channels between processes as *queues*. It suffices to maintain a count of the messages in transit between each pair of processes. The structure of the MSC and the labelling of the events ensures that messages are consumed in the order in which they are generated.

The behaviour of the netchart is now given by the normal token game on the "low level" net that we have generated by plugging in a net for each MSC in the netchart. In the low level net, the control places that are inherited from the original high level net are safe. However, the buffer places can have an arbitrary number of tokens. Thus, the low level net is not, in general, safe.

All processes need not traverse the high level transitions in the same order. Each process proceeds asynchronously and only gets blocked if, within an MSC, it requires a token from a buffer place and the buffer place is empty.

As we fire the "low level" transitions corresponding to the events of the MSCs labelling the high level transitions, we build up in the obvious way a partial order that can be regarded as an MSC. A complete MSC is generated when all send events have been matched up with corresponding receive events. This is captured by declaring a marking to be an accepting one if all the buffer places are empty at the marking. Thus, the language of the netchart is defined to be the set of MSCs generated by all firing sequences leading to accepting markings. (Note that there are only a finite number of such markings.) We can further control the language of a netchart by defining a set $\mathcal{F}$ of final control markings and insist that a marking is accepting only if all buffer places are empty *and* the control marking belongs to $\mathcal{F}$. Figure 4 describes the low-level net associated with the netchart in Figure 3 and also exhibits a typical MSC in the language defined by this netchart.

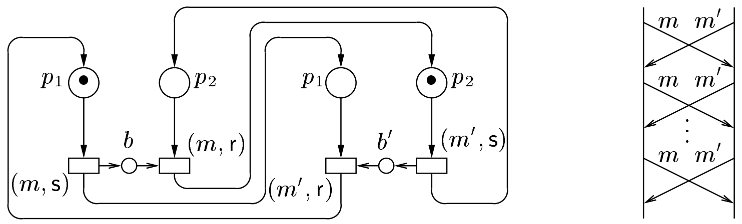

Fig. 4. The low level net for Figure 3 and a typical MSC that it generates

4 From netcharts to message-passing automata

The low level net associated with a netchart makes it an executable specification. We now show that a netchart can also easily be transformed into the executable model often used in connection with HMSCs [1, 12].

4.1 Message passing automata

A natural implementation model for MSC languages is a message-passing automaton. Recall that the set of processes $\mathcal{P}$ determines the communication alphabet Γ. For $p \in \mathcal{P}$, let $\Gamma_p = \{p!q(m), p?q'(m') \mid q, q' \in \mathcal{P}, m, m' \in \Delta\}$ denote the actions that process p participates in.

A *message-passing automaton* over Γ is a structure $\mathcal{A} = (\{\mathcal{A}_p\}_{p\in\mathcal{P}}, Aux, s_{in}, \mathcal{F})$ where:

- Aux is a finite alphabet of (auxiliary) messages.
- Each component $\mathcal{A}_p$ is of the form $(S_p, \longrightarrow_p)$ where
  - S_p is a finite set of p-local states.
  - $\longrightarrow_p \subseteq S_p \times \Gamma_p \times Aux \times S_p$ is the p-local transition relation.
- $s_{in} \in \prod_{p\in\mathcal{P}} S_p$ is the global initial state.
- $\mathcal{F} \subseteq \prod_{p\in\mathcal{P}} S_p$ is the set of global final states.

The local transition relation $\longrightarrow_p$ specifies how the process p sends and receives messages. The transition $(s, p!q(m), \mu, s')$ specifies that in state s, p can send the message m augmented with auxiliary information μ to q by executing the communication action $p!q(m)$ and go to the state s'. Similarly, the transition $(s, p?q(m), \mu, s')$ signifies that at the state s, p can receive the message (m, μ) from q by executing the action $p?q(m)$ and go to the state s'.

The behaviour of a message-passing automaton is described in terms of *configurations*. Channels are modelled as queues. A configuration specifies the local state of each process together with the contents of each queue. A transition of the form $(s, p!q(m), \mu, s')$ appends the message (m, μ) to the queue (p, q). Similarly, a transition of the form $(s, p?q(m), \mu, s')$ removes the message (m, μ) from the head of the queue (q, p)—this transition is blocked if such a message is not available. A final configuration is one in which the local states of the processes constitute a final state of the message-passing automaton and all queues are empty. We can define a run of such an automaton, as usual. Runs that lead to final configurations are called *accepting*. The language accepted by the automaton is the set of words that admit accepting runs. The structure of the automaton ensures that this word language corresponds to the set of linearizations of an MSC language, so we can associate an MSC language in a natural way with a message-passing automaton. For more details, the reader is referred to [9, 12].

4.2 From netcharts to automata

Each non-buffer place in the low-level net corresponding to a netchart can be uniquely assigned a location in $\mathcal{P}$. The places that belong to process p define, in a natural way, a local sequential component of a message-passing automaton.

The resulting message-passing automaton differs in one important aspect from the original netchart—each MSC that labels a transition in a netchart writes into a private set of channels, while there is only one common set of channels in a message-passing automaton. So long as the MSCs generated by the netchart obey the FIFO restriction on each channel, this collapsing of channels does not affect the MSC language and the language accepted by the message-passing automaton is the same as the one defined by the netchart.

However, netcharts can define MSCs that violate the FIFO restriction on channels, as seen in Figure 5. We cannot directly translate such a netchart into a message-passing automaton. The solution is to weaken the definition of an MSC to permit multiple channels between processes. Different channels may carry messages of the same type. Each channel is FIFO, but messages on different channels can overtake each other (even between the same pair of processes).

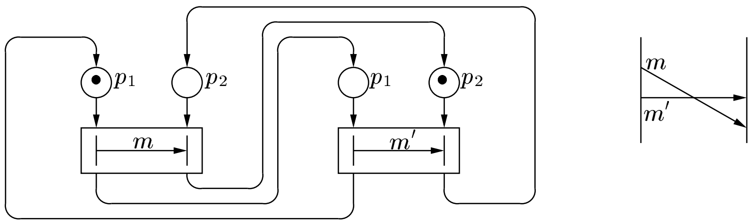

Fig. 5. A netchart that generates a non FIFO MSC

We can then augment message-passing automata so that each send and receive specifies the channel name in addition to the message type. With this extension, netcharts can always be converted into message-passing automata. However, for simplicity, we shall assume that we only deal with netcharts that generate MSCs that respect the FIFO restriction and work with the simpler definition of message-passing automata that we presented initially.

5 Regular netchart languages

We begin with a simple observation about Petri nets. Let $N = ((S,T,F), M_{in})$ be a Petri net and $L(N)$ denote the set of firing sequences of N. Recall that a Petri net is bounded if there exists a uniform upper bound B such that at any reachable marking, every place contains at most B tokens.

Proposition 2. *If N is a bounded Petri net then, $L(N)$ is regular.*

This easily yields a sufficient criterion for a netchart language to be regular.

Lemma 3. *If the low level net associated with a netchart is bounded, the MSC language of the netchart is regular.*

Since boundedness is a decidable property for nets, this condition can be effectively checked. Since the converse of the preceding lemma does not hold, deciding the regularity of a netchart language is not straightforward. For the netchart language to be non-regular, we have to find an unbounded family of markings, each of which can be reduced to a legal final marking where all buffer places are empty.

However, for the special case where each component of a netchart is a cyclic process, regularity is decidable.

Theorem 4. *Consider a netchart in which each component is a cyclic process. It is decidable if the language of such a netchart is regular.*

Proof. We construct the *communication graph* for the entire netchart. This graph has the set of processes as vertices and an edge from p to q if there is a transition in the netchart labelled by an MSC with a message from p to q.

Let C, C' be maximal strongly connected components (scc's) of this communication graph. We draw an edge from C to C' if there exists $p \in C$ and $q \in C'$ such that there is an edge in the communication graph from p to q. This induced graph of scc's is a dag. This dag may itself break up into a number of connected components. We can analyze each of these components separately, so, without loss of generality, assume that the entire dag is connected.

The buffer places that connect processes within each scc are automatically bounded (for the same reason that a locally synchronized HMSC generates a regular MSC language). Thus, the only buffer places that can be unbounded are those that connect processes in different scc's.

Suppose $p \in C$, $q \in C'$ and there is an edge from p to q in the communication graph. Then, by the structure of a netchart, each time p enters a high-level transition where it sends a message to q, there must be a matching instance where q enters so that the buffer places are cleared out. This has two implications.

(i) If p is not live—that is, it can execute only a finite number of actions—then q is not live. Indeed, all of C and C' are not live.
(ii) If q is not live, then any messages generated by p after q quits will never be consumed, so the resulting behaviour will not lead to an MSC in the language. Thus, effectively the buffer place between p and q is bounded.

Thus, if a process p is not live, all processes in the scc C containing p as well as all scc's that are descendants of C in the dag are not live. Moreover, after a bounded initial period, any message produced by an ancestor scc will never be consumed. So, as far as the MSC language is concerned, the ancestors of C also have only a bounded life. In other words, if even a single process is not live, the entire dag has a bounded behaviour and the resulting MSC language is bounded. Conversely, the language is unbounded iff all the scc's in the dag are live.

Thus the problem reduces to checking whether all the scc's in the dag are live. Observe that the minimal scc's in the dag have no input buffer places, so their liveness can be analyzed in isolation. This is not difficult to check given the simple cyclic structure of the components. If all the immediate predecessors of an scc are live, then the liveness of the scc again depends only on its internal structure, so we perform the same check as for the minimal scc's. Thus, we can systematically check the liveness of all the scc's in the dag by sorting them topologically and analyzing them in this sorted sequence. $\square$

6 HMSCs vs netcharts

Every HMSC language is finitely generated (and all finitely generated regular MSC languages are HMSC representable). It turns out that netcharts can also generate *regular* MSC languages that are not finitely generated. An example of this is the alternating bit protocol in which two processes communicate over a lossy channel. The sender alternately tags the data it sends with bits 0 and 1. The receiver acknowledges the bit corresponding to each data item it receives. The sender flips its bit each time it gets the acknowledgement it is waiting for (and ignores any incorrect acknowledgement that it may receive).

Figure 6 shows a netchart corresponding to the alternating bit protocol. In the figures, messages d_0 and d_1 represent data sent with bit 0 and 1, respectively, and messages a_0 and a_1 represent the corresponding acknowledgements.

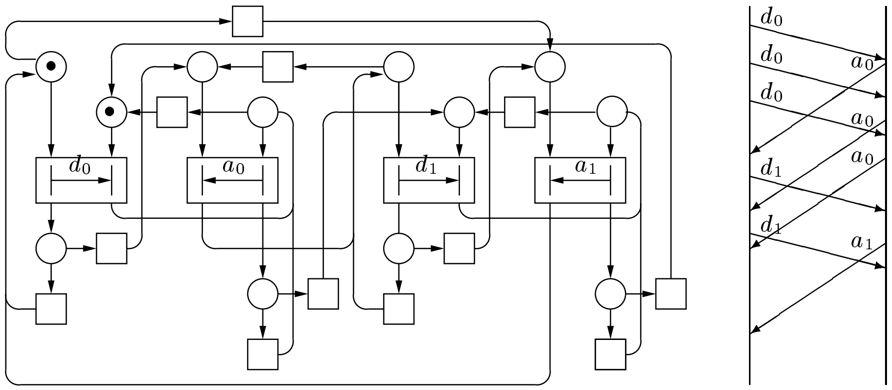

Fig. 6. A netchart for the alternating bit protocol, with a typical MSC

The first HMSC we encountered, Figure 2, generates a non-regular language in which all channels are bounded. It is not difficult to show that any netchart implementation of this HMSC (i.e., its associated low level Petri net) would have bounded buffers. For netcharts, bounded buffers imply regularity. Hence, this HMSC is not representable via netcharts. However, from the result we will prove in the next section it follows that every HMSC that defines a regular MSC language has an equivalent netchart.

7 From regular MSC languages to netcharts

We now show that every regular MSC language can be represented as a netchart. We begin by recalling the characterization of regular MSC languages in terms of message-passing automata. A B-bounded message-passing automaton is one in which in every reachable configuration, there are at most B messages in transit in any queue. We then have the following result from [12].

Theorem 5. *An MSC language L is regular iff there is a B-bounded message-passing automaton that accepts L, for some bound $B \in \mathbb{N}_0$.*

We now show that netcharts with final control markings can generate all regular MSC languages.

Theorem 6. *Every regular MSC language can be represented as a netchart.*

Proof Sketch: We can simulate a B-bounded message-passing automaton using a netchart.

In a B-bounded message-passing automaton, each queue is uniformly bounded by B. We can naïvely regard each channel as a cyclic buffer of size B with slots labelled 0 to $B-1$. Each process begins by reading from or writing into slot 0 along each channel that it is connected to. After a process reads from (or writes to) slot i of a channel c, when it next access the same channel it will read from (or write to) slot $i+1 \bmod B$, which we denote $i \oplus 1$.

In this framework, the complete configuration of a process is given by its local state s and the slot k_i that it will next read from or write to for each channel c_i that it is connected to. Initially, a process is in the configuration $(s^p_{in}, 0, \ldots, 0)$.

Each local transition $s \stackrel{c_i!(m,\mu)}{\longrightarrow} s'$ (or $s \stackrel{c_i?(m,\mu)}{\longrightarrow} s'$) generates a family of moves of the form $(s, k_1, k_2, \ldots, k_i, \ldots, k_n) \stackrel{c_i!(m,\mu)}{\longrightarrow} (s, k_1, k_2, \ldots, k_i \oplus 1, \ldots, k_n)$ (or $(s, k_1, k_2, \ldots, k_i, \ldots, k_n) \stackrel{c_i?(m,\mu)}{\longrightarrow} (s, k_1, k_2, \ldots, k_i \oplus 1, \ldots, k_n)$, for each choice of $k_1, k_2, \ldots, k_n$, corresponding to the n channels that the process reads or writes.

We construct a netchart in which, for each process p, there is separate place corresponding to each configuration $(s, k_1, k_2, \ldots, k_i, \ldots, k_n)$ of p in the B-bounded message-passing automaton. Consider a message (m, μ) that is transmitted on channel c_i during a run of the message-passing automaton. When this message is sent, the sending process is some configuration of the form $(s, k_1, k_2, \ldots, k_i, \ldots, k_n)$ and when this message is received, the receiving process is in some configuration of the form $(t, \ell_1, \ell_2, \ldots, k_i, \ldots, \ell_n)$ (notice that the value of k_i in the two configurations is necessarily the same). For each such pair of sending and receiving configurations, we create a separate transition in the netchart representing c_i, labelled with an MSC consisting of a single message (m, μ) (see Figure 7). We use internal transitions to let the sending process guess the context in which the message will be received and fire the corresponding transition in the netchart. Symmetrically, the receiving process guesses the context in which the transition was sent and enters the corresponding transition in the netchart.

We can then associate a set of global final control states with this netchart corresponding to the accepting states of the original message-passing automaton. It is not difficult to see that this netchart accepts the same MSC language as the original message-passing automaton.

The crucial observation is that any inconsistent choice among nondeterministic transitions in the netchart must lead to deadlock. Suppose that for channel c_i, the sending and receiving processes make incorrect guesses about each other's contexts for message (m, μ) sent in slot k_i along c_i. This would result in the receiving process getting stuck in the wrong netchart transition. The only way for it to make progress is for the sending process to eventually cycle through all the other slots along channel c_i, return to slot k_i and send a fresh copy of the message (m, μ) within the netchart transition where the receiving process is stuck. This can be mapped back into a run of the message-passing automaton in which the channel c_i is not B-bounded, which is a contradiction. □

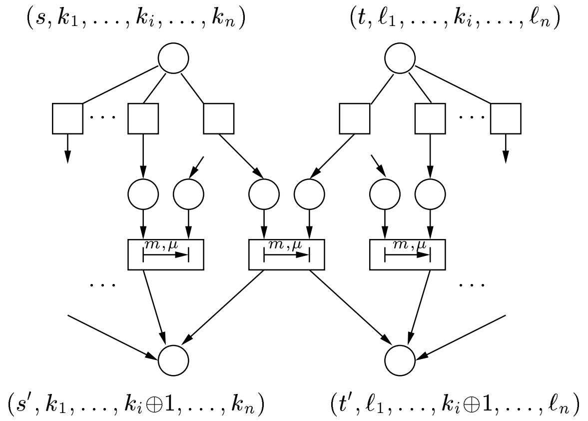

Fig. 7. The translation from B-bounded message-passing automata to netcharts

8 Discussion

We have shown that by distributing the control in an HMSC in appropriate way, we can derive a model that is much more easily implementable than HMSCs while retaining the appeal of a visual formalism. In terms of the regular MSC languages they can represent, netcharts are more expressive than HMSCs. In general, the expressive power of the two formalisms is incomparable.

As mentioned in the introduction, an important aspect of the netchart model is that the compound MSCs that are defined by a netchart are built up from "complete" MSCs. This is important from the point of view of using such a formalism to capture system requirements at a reasonably intuitive level. An alternative approach, suggested in [6], is to label HMSC nodes with *compositional* MSCs that may have unmatched sends and receives. A single send or receive action is a trivial example of a compositional MSC. Thus, a specification in terms of compositional MSCs very quickly becomes indistinguishable from a concrete implementation in terms of distributed automata.

One natural direction in which to extend this work is to use netcharts for capturing requirements in realistic settings.

At a technical level, the following interesting questions remain open:

– What is the exact relationship between the class of Netchart languages and the class of HMSC representable languages?
– Is the problem of checking whether the language of a netchart is regular decidable?

References

1. Alur, R., Etessami, K. and Yannakakis, M.: Realizability and verification of MSC graphs. *Proc. ICALP'01*, LNCS **2076**, Springer-Verlag (2001) 797–808.
2. Alur, R. and Yannakakis, M.: Model checking of message sequence charts. *Proc. CONCUR'99*, LNCS **1664**, Springer-Verlag (1999) 114–129.
3. Booch, G., Jacobson, I. and Rumbaugh, J.: *Unified Modeling Language User Guide.* Addison-Wesley (1997).
4. Caillaud, B., Darondeau, P., Helouet, L., and Lesventes, G.: HMSCs as partial specifications ... with PNs as completions. *Modeling and Verification of Parallel Processes 4th Summer School, MOVEP 2000*, LNCS **2067**, Springer-Verlag (2001) 125–152.
5. J. Desel and J. Esparza: Free Choice Petri Nets. *Cambridge Tracts in Theoretical Computer Science 40*, Cambridge University Press (1995).
6. Gunter, E., Muscholl, A. and Peled, D.: Compositional message sequence charts. *Proc. TACAS'01*, LNCS **2031**, Springer-Verlag (2001) 496–511.
7. Harel, D. and Gery, E.: Executable object modelling with statecharts. *IEEE Computer*, July 1997 (1997) 31–42.
8. Henriksen, J.G., Mukund, M., Narayan Kumar K., and Thiagarajan, P.S.: On Message Sequence Graphs and Finitely Generated Regular MSC Languages. *Proc. ICALP'00*, LNCS **1854**, Springer-Verlag (2000) 675–686.
9. Henriksen, J.G., Mukund, M., Narayan Kumar K., and Thiagarajan, P.S.: Regular Collections of Message Sequence Charts. *Proc. MFCS'00*, LNCS **1893**, Springer-Verlag (2000) 405–414.
10. ITU-TS Recommendation Z.120: *Message Sequence Chart (MSC).* ITU-TS, Geneva (1997).
11. Mauw, S. and Reniers, M. A.: High-level message sequence charts, *Proc. 8th SDL Forum, SDL'97: Time for Testing — SDL, MSC and Trends*, Elsevier (1997) 291–306.
12. Mukund, M., Narayan Kumar, K. and Sohoni, M.: Synthesizing distributed finite-state systems from MSCs. *Proc CONCUR 2000*, LNCS **1877**, Springer-Verlag (2000) 521–535.
13. Muscholl, A. and Peled, D.: Message sequence graphs and decision problems on Mazurkiewicz traces. *Proc. MFCS'99*, LNCS **1672**, Springer-Verlag (1999) 81–91.
14. Roychoudhury, A. and Thiagarajan, P. S.: Communicating transaction processes. *Proc. ACSD'03*, IEEE Press, 2003 (to appear).
15. Rudolph, E., Graubmann, P. and Grabowski, J.: Tutorial on message sequence charts. *Computer Networks and ISDN Systems — SDL and MSC* **28** (1996).
16. Sgroi, M., Kondratyev, A., Watanabe, Y. and Sangiovanni-Vincentelli, A. : Synthesis of Petri nets from MSC-based specifications. Unpublished manuscript.