# SYNTHESIS OF PETRI NETS FROM MSC-BASED SPECIFICATIONS

by

Marco Sgroi, Alex Kondratyev, Yosinori Watanabe and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M02/38

19 December 2002

# SYNTHESIS OF PETRI NETS FROM MSC-BASED SPECIFICATIONS

by

Marco Sgroi, Alex Kondratyev, Yosinori Watanabe and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M02/38

19 December 2002

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# Synthesis of Petri Nets from MSC-based specifications

Marco Sgroi[1], Alex Kondratyev[2], Yosinori Watanabe[2], Alberto Sangiovanni-Vincentelli[1]

[1] University of California at Berkeley, EECS Department Cory Hall, Berkeley, CA 94704

[2] Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94704-1144

**Abstract.** The design of protocols, i.e. the set of rules that define how the communication among the components of a distributed system takes place, requires the use of formal techniques and tools to obtain a correct and efficient implementation. This paper presents a new approach for protocol synthesis from a scenario based specification. The designer first describes sequences of interactions between the components using Message Sequence Charts (MSCs), and then composes the MSCs using the Petri net semantics. To apply analysis and synthesis to this specification, it is necessary to formally represent the traces of the events of the composed MSCs. Hence, we propose an algorithm that generates a Petri net that captures these traces. We demonstrate the application of the proposed approach using an example of a wireless protocol.

## 1 Introduction

The design of the communication protocols that rule the interaction among distributed system components is a very challenging problem that requires the use of formal methodologies and tools to obtain high-performance and low-cost implementations. Most of the formal methodologies for protocol design are based on models, such as networks of finite state machines or dataflow networks, that represent the system as a set of interacting blocks. A limitation of these methodologies is that they impose an early partitioning of the protocol behavior and force the designer to decompose from the beginning the overall protocol functionality into components and later describe the complete behavior of each component for all possible cases and scenarios in which it participates. Many errors occur during this phase especially when the protocol operates under multiple distinct but correlated scenarios.

A more convenient approach consists of specifying protocol behavior in terms of a set of related scenarios, where a scenario corresponds to a sequence of interactions describing how the system reacts to the stimuli from its environment. Several design methodologies based on specification of scenarios have been proposed [5] [11] [1] [10]. They differ mostly for the formal models on which they are based (e.g. UML, MSCs, Statecharts, PNs, SDL...) and for the tools that do the mapping between models of the system at different levels of abstraction.

The most popular language for specification of scenarios is Message Sequence Charts (MSCs) [8]. MSCs is a graphical language that allows to visualize the behavior of the system only for what concerns the interaction among its components. This makes an MSC model easy to understand and therefore an ideal tool for the sharing of documentation among design team members. Several variations of the basic MSC model, such as High-level Message Sequence Charts (HMSCs) [8] and Live Sequence Charts (LSCs) [5] have been introduced to address limitations of the basic model and to capture more conveniently complex specifications.

A MSC visualizes a partial order between send and receive message events; however, the standard [8] leaves undefined both the semantics of the communication among the MSCs processes and the process execution policy. Hence, the same MSC can often be subject to multiple interpretations. Several approaches have been proposed to formalize the semantics of MSCs either at the denotational level using algebra [13] or at the operational level using Petri Nets [6] [4], SDL [1] and Statecharts [12]. DeMan [13] formalizes basic MSCs and a set of operators to compose them, such as alternative, parallel composition and preemption. Kuster et al. [12] define consistency between UML Sequence Diagrams and Statecharts models in terms of the containment of event traces defined in the two models. Graubmann et al. [4] define the MSCs semantics using Petri Nets and propose a procedure to derive a PN from a MSCs composing PNs fragments. However, the approach is limited to a restricted class of basic MSCs where conflicts are not allowed. Heymer [6] uses PNs to define a non-interleaving semantics for MSCs in terms of PNs components. Tools based on MSCs have been developed for an early analysis of the properties of a specification and the detection of errors, such as deadlocks, process divergence [2], and race conditions [7]. Several algorithms and tools [5] [11] [1] [7] have been developed also for the automatic synthesis of state-based models from MSCs.

Our approach to protocol design defines a procedure for synthesis from a scenario-based specification. Scenarios are specified using Message Sequence Nets (MSNs), a model that includes multiple MSCs and explicits their relations of ordering, concurrency or conflict using the PNs semantics. Hence, a MSN is a PN whose transitions are labeled with basic MSCs. Synthesizing a HW or SW implementation from a high-level functional specification in MSNs requires to use an intermediate operational model that formally defines the semantics of the MSCs. We have chosen Petri Nets [14] as an intermediate model to support our automatic synthesis procedure from MSNs because it can be analyzed for an early detection of specification errors and synthesized into a hardware or software implementation.

In this paper first we define the semantics of the MSN model and then focus on the notion of consistency between MSNs and PNs models. We show how to derive a consistent PN model from a given MSN using a covering algorithm that makes use of a library of PNs patterns and covers one by one each element of the MSN specification.

The paper is organized as follows. The main notions for MSC, PNs and MSNs are introduced in Sections 2, 3 and 4. Section 5 discusses the analysis of MSNs. Section 6 first describes the most relevant PN patterns, defines consistency between MSNs and PNs and presents a covering algorithm that derives consistent PNs from given MSNs.

## 2 Message Sequence Charts

**Definition 1.** *A (basic) MSC is a tuple* $M = (E, Pr, g, h, m, <)$, *where* $E$ *is a finite set of events,* $Pr$ *is a set of processes,* $g$ *is a mapping that associates each event with a process,* $h$ *is a mapping that associates each event with a type (send or receive),* $m$ *is a bijective function that matches pairs of send and receive events.* $<$ *is a partial order relation between events, defined by the union of a partial order* $<_c$ *and total orders* $<_{P_i}$ *over all the processes* $P_i \in Pr$. *The partial order* $<_c$ *relates events for the same messages, i.e.* $\{(e_i, e_j) | h(e_i) = send, h(e_j) = receive, m(e_i) = e_j\}$. $<_{P_i}$ *is a total order of events for the process* $P_i$. *The transitive closure* $<^*$ *of* $<$ *is called visual order.*

A MSC models the behavior of a system by visualizing the exchange of messages between its components during their lifetime. It represents each system component as a process, and explicitly visualizes all the send and receive events of the processes and their partial order. [3] The implementation of the behavior specified in a MSC must guarantee that the corresponding events satisfy the visual order. For example, if two messages are consecutively sent by a sender process, the order of the events at the receiver can be guaranteed only choosing an architecture with a FIFO communication channel between the two processes. For this reason, together with the visual order, it is usually given an *enforceable order* determined by a set of semantic rules defined by the communication architecture.

MSCs is not an operational model, since it lacks a description of how and when processes produce and consume events, neither it has a fully defined communication semantics. The only assumptions about communication are that it is asynchronous and that a receive event is preceeded by the send event of the corresponding message. Hence, to derive an implementation that is correct and consistent with a given MSC model several steps, such as mapping the MSC specification onto a communication architecture and defining for each process the execution policy, must be taken. Consider for example the MSC in figure 1a. The specified behavior is that process P2 receives messages $a$ and $b$ in the visualized order and then sends message $c$ to process P1. The communication architecture is not specified in MSCs: one can select an architecture that consists of a FIFO channel between P1 and P2 or another including a non-FIFO buffer and a reordering mechanism at the receiver. Both guarantee that the order of

[3] A labeling function $l$ associates MSC events and messages with labels. The alphabet $\Sigma = \{a, b, c...\}$ is usually used to label messages, $\Sigma_s = \{!a, !b, !c...\}$ to label send events and $\Sigma_r = \{?a, ?b, ?c...\}$ receive events.

reception of $a$ and $b$ is maintained. Also, several execution policies can be chosen for each process: e.g. P2 can read $a$ and $b$ in two separate transitions as soon as each message is received or read both messages in a single transition. A refined specification of P2 may include also its internal computation, e.g. the processing of messages $a$ and $b$.

To extend the modeling capability of the basic MSC model several constructs, such as conditions and branching to express data-dependent control decisions, and co-regions [8] to identify events in process timelines that are not totally ordered have been introduced. However, for complex specifications basic MSCs become soon unreadable as the number of interactions captured in the model grows. To explicitly visualize sets of MSCs and their relations models that allow to compose multiple MSCs have been proposed. [8] defines High-level MSCs (HMSCs) as graphs whose nodes can be labeled with basic MSCs and arcs represent relations among MSCs. An HMSC is a transition system whose execution begins at the initial node and proceeds through a sequence of nodes. At each node one of the outgoing transitions is selected and the system moves to the next node. The paths from the initial to the final nodes represent the possible executions of the system.

## 3 Petri Nets

A Petri Net [14] is a triple $(P, T, F)$, where $P$ is a non-empty finite set of places, $T$ a non-empty finite set of transitions and $F : (T \times P) \cup (P \times T) \rightarrow N$ the weighted flow relation between transitions and places. If $F(x, y) > 0$, there is an arc with the weight $F(x, y)$ from node $x$ to node $y$. If $\forall x, y \; F(x, y) = 1$ the PN is called *ordinary*. Given a node $x$, either a place or a transition, we define its preset as ${}^\bullet x = \{y | (y, x) \in F\}$ and its postset as $x^\bullet = \{y | (x, y) \in F\}$. For a node $y$, $Pre[X, y]$ is a vector whose i-th component is equal to $F(x_i, y)$. A transition (place) whose preset is empty is called source transition (place), a transition (place) whose postset is empty is called sink transition (place). A place $p$ such that $|p^\bullet| > 1$ is called choice. A choice is called free (equal) if all its successor transitions have at most one predecesor (the same preset, i.e. $F(p, t_i) = F(p, t_j) \; \forall p \in P$). A place $p$ such that $|{}^\bullet p| > 1$ is called merge. A transition $t$ such that $|t^\bullet| > 1$ ($|{}^\bullet t| > 1$) is called fork (join). A *marking* $\mu$ is an n-vector $\mu = (\mu_1, \mu_2, ..., \mu_n)$ where $n = |P|$ and $\mu_i$ is the non-negative number of tokens in place $p_i$. The vector $\mu_0$ is used to indicate the initial marking. During execution a PN moves from one marking to another by firing a transition. A transition $t$ such that each input place $p_i$ is marked with at least $F(p_i, t)$ tokens is enabled and may fire. When transition $t$ fires, $F(p_i, t)$ tokens are removed from each input place $p_i$ and $F(t, p_j)$ tokens are produced in each output place $p_j$.

An arbitrary PN $N = (P, T, F)$ could be unfolded into an equivalent acyclic PN $N' = (P'T'F')$ (possibly infinite) called *occurrence net*. This net is defined by a homomorhism $h$ between $P$ and $P'$ and $T$ and $T'$ in such a way that for any $t = h(t')$, $h(\bullet t') = \bullet t$ and $h(t' \bullet) = t \bullet$ and none of the places in $N'$ has more than one predecessor. $t = h(t')$ and $p = h(p')$ are called origins of $t'$ and $p'$

respectively, while $t'$ and $p'$ are called instances of $t$ and $p$. The formal defintions of occurrence net could be found in [3].

## 4 Message Sequence Nets

A Message Sequence Net (MSN) is a model that represents the interactions of a distributed system as a set of MSCs and expresses the relationships among MSCs using PNs semantics. A MSN is a PN whose transitions are labeled with basic MSCs and whose places represent the preconditions for the firing of the MSN transitions, i.e. for the occurrence of the events in the corresponding MSCs. Places hold tokens whose count defines the marking of a MSN.

**Definition 2.** *A MSN is a pair $\tilde{M} = (\tilde{N}, bmsc)$, where $\tilde{N}$ is an underliying PN $\tilde{N} = (\tilde{P}, \tilde{T}, \tilde{F}, \mu_0)$, and bmsc is a map that associates each transition with a basic MSC.*

**Definition 3.** *An MSN occurrence net $\tilde{M}'$ is an infinite MSN $\tilde{M}' = (\tilde{N}', bmsc')$, where $N'$ is an occurrence net of $\tilde{N}$ and bmsc' associates the label bmsc(t) of transition t with all its instances in $N'$.*

The PN graph $N$ defines the relations among MSCs and their events as follows.

- *Sequential composition* (Figure 1b): $M_i$ is a predecessor of $M_j$ if there exists a path in $\tilde{M}$ from a transition $t_i = bmsc^{-1}(M_i)$ to a transition $t_j = bmsc^{-1}(M_j)$. Sequential composition can be weak ($M_i \rightarrow M_j$) or strong ($M_i \Rightarrow M_j$). If it is weak for each process any event in $M_j$ follows any event in $M_i$ (i.e. there is no synchronization among processes from $M_i$ to $M_j$). If it is strong any event in $M_j$ follows any event in $M_i$, i.e. processes synchronize before execution of $M_j$.
- *Alternative composition* (Figure 2b): a non-deterministic choice models conflict between a pair of MSCs and between their successors. $M_i$ is in direct conflict with $M_j$ if $\exists t_i = bmsc^{-1}(M_i)$ and $t_j = bmsc^{-1}(M_j)$ s.t. ${}^\bullet t_i \cap {}^\bullet t_j \neq \emptyset$. $M_i$ is in conflict with $M_j$ ($M_i \# M_j$) if either it is in direct conflict or $\exists M_h \rightarrow M_i, M_k \rightarrow M_j$ s.t. $M_h$ is in direct conflict with $M_k$.
- *Parallel composition* (Figure 1c): $M_i$ is concurrent with $M_j$ ($M_i || M_j$) if $t_i = bmsc^{-1}(M_i)$ and $t_j = bmsc^{-1}(M_j)$ are concurrent, i.e. there exists a marking in $N$ in which both transitions are enabled and ${}^\bullet t_i \cap {}^\bullet t_j = \emptyset$.
- *Synchronization*: modeled as a MSC $M_i$ with mulitiple predecessors (Figure 1d).
- *Merge*: modeled as a place with multiple predecessors (Figure 2a). Merge is safe if all its predecessors are in conflict relation, unsafe otherwise.
- *Iteration*: modeled as a cycle including the MSCs to be iterated (Figure 2c).

The set of events of a MSN $\tilde{E} = \bigcup_i E_i$ is the union of all the sets of events of the basic MSCs associated with the MSN transitions. In the occurrence net $M'$ derived by MSN $M$ pairs of events in $\tilde{E}'$ can be in one of the following relations:

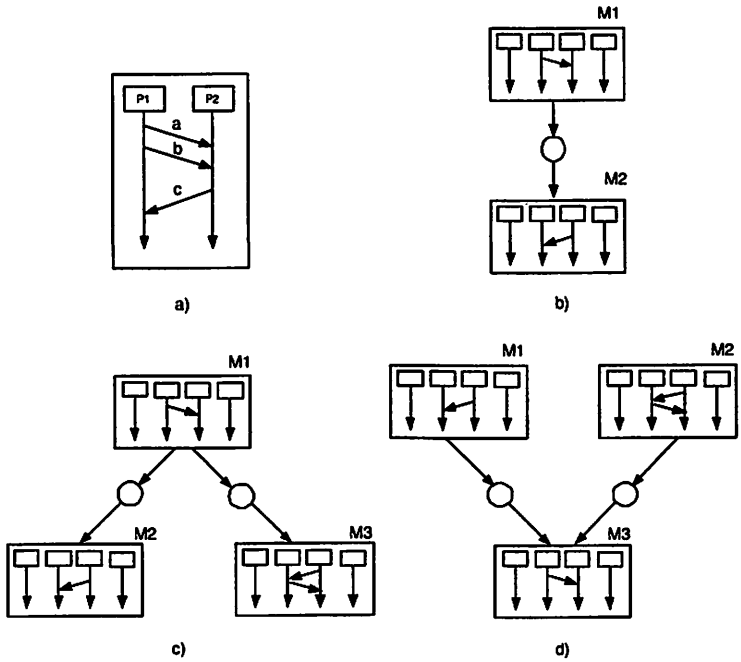


**Fig. 1.** Message Sequence Chart (a), Weak or Strong Sequential Composition (b), Parallel Composition (c), Synchronization (d)

- ordering ($\tilde{<}$): $e'_1 \tilde{<} e'_2$ if one of the following is true
  - $e'_1, e'_2 \in E'_i$ and $e'_1 <_i e'_2$
  - $e'_1 \in E'_i, e'_2 \in E'_j$, $M'_i \Rightarrow M'_j$
  - $e'_1 \in E'_i, e'_2 \in E'_j$, $g(e'_1) = g(e'_2)$, $M'_i \rightarrow M'_j$
  - $e'_1 \in E'_i, e'_2 \in E'_j$, $\exists e'_k \in E'_k$ s.t. $g(e'_k) = g(e'_2)$, $M'_k \rightarrow M'_j$, $e'_1 \tilde{<} e'_k$
- conflict ($\tilde{\#}$): $e'_1 \# e'_2$ if $e'_1 \in E'_i, e'_2 \in E'_j$ and $M'_i \# M'_j$
- concurrency ($\tilde{\|}$): $e'_1 \| e'_2$ if $e'_1 \in E'_i, e'_2 \in E'_j$ and $M'_i \| M'_j$

It is assumed that every process $P_i$ begins execution of an MSC $M_j$ issueing an event with meaning "start" that is not visualized in the MSC and is labeled $P_i M_j$. If we consider the alphabet $\Sigma = \{a, b, c...\}$ of the message labels and the alphabet $\Sigma^* = \{!a, ?a, !b, ?b, ...\} \cup \{P_1 M_1, P_2 M_2, ...\}$ of the corresponding send, receive and start events labels, the language $L(\tilde{M})$ is the set of traces over $\Sigma^*$ that satisfy the $\tilde{<}, \tilde{\#}, \tilde{\|}$ relations of $\tilde{M}$.

In the rest of the paper we consider MSNs for which the following assumptions hold:

1. All MSCs in a MSN include the same set of processes $\{Pr_l\}$. A MSC may include no events associated with a certain process.
2. Conflicts are explicitly expressed as choices external to MSCs, i.e. pairs of events belonging to the same MSC cannot be in conflict

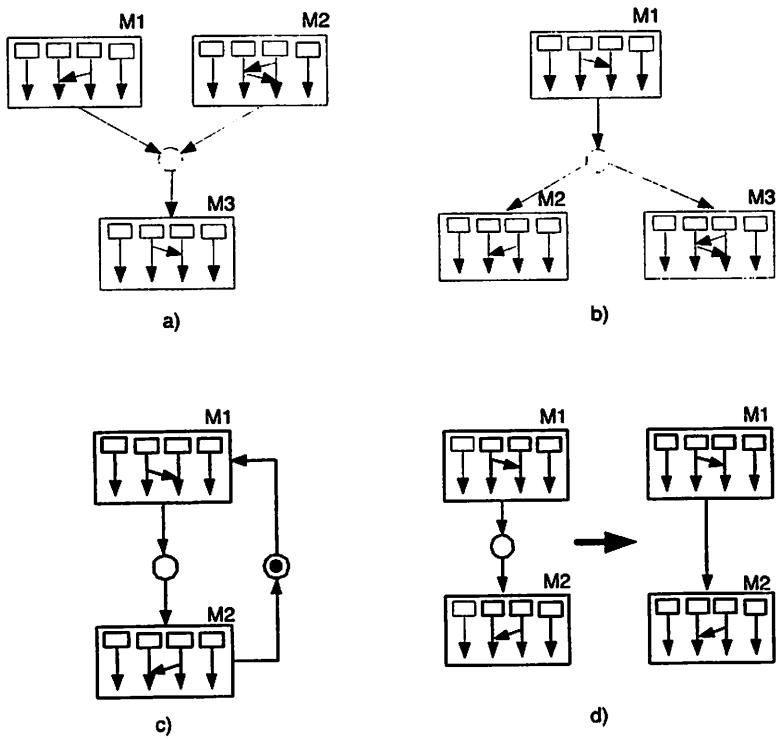


**Fig. 2.** Merge (a), Alternative Composition (b), Iteration (c), Simplified MSCs (d)

3. MSCs do not have co-regions and concurrency is explicitly expressed in terms of MSCs relationships
4. Sequential composition is weak unless specified otherwise
5. To improve readability
   (a) MSN places with a single predecessor and a single successor and not holding tokens in the initial marking are replaced by an arc connecting directly the predecessor MSC with the successor MSC (Figure 2d)
   (b) empty MSCs are represented as PNs transitions

There is one more limitation that needs to be imposed on MSN model. The need for it is less obvious and we first motivate it through an example.

Figure3(a)(b) shows a MSN $M$ and its occurrence net $M'$. The language that is realized by this MSN is $\{M1^n, M2\}$. When scenario $M2$ is implemented all instances $a^i, b^i$ of events $a$ and $b$ issued during preceding firings of $M1$ must be synchronized with the process $P3$ (due to the transitive nature of precedenece relation: $x, y > z$, $x, y > a^i, b^i$ implies $z > a^i, b^i$). This requires from the process $P3$ to keep track of how many times $M1$ was invoked which could be implemented with an unbounded memory only.

To keep a MSN operation bounded we impose a safeness restriction and assume that in MSN $M$ between two consecutive instantiations $Mk^i$ and $Mk^{i+1}$

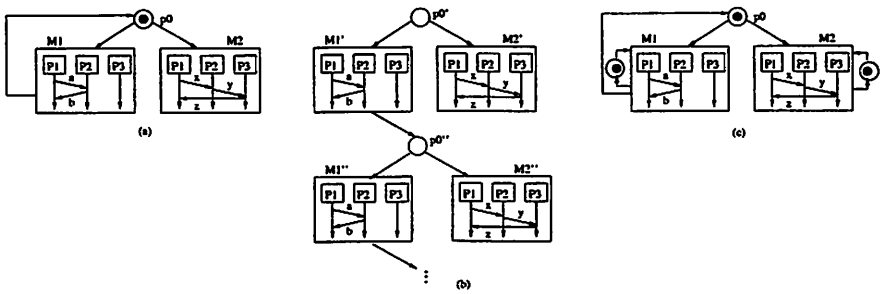

**Fig. 3.** Unbounded MSN (a) its occurrence net (b) and the safe counterpart (c)

of MSC $Mk$ all processes are synchronized. Informally this means that each MSC in MSN has a self-loop with a strong synchronization with itself (see Figure 3(c)). To avoid cluttering we would not include this self-loop in MSN figures but will always assume that it exists for every MSC. Note that due to the presence of "start" event in the beginning of each process the synchronization between processes is easy to define even when processes do not issue observable events. In that case (see e.g. process $P3$ in Figure3(a)) the synchronization occurs between start events.

**Definition 4.** *MSN $M$ is safe if*

*a) its underlying PN is safe and*

*b) for any two instances $Mk^i$ and $Mk^{i+1}$ ($Mk^i \rightarrow Mk^{i+1}$) of MSC $Mk$ in the occurrence net $M'$ and $\forall e_j^i, e_p^{i+1} : e_j^i \in Ek^i$; $e_p^{i+1} \in Ek^{i+1}$ follows $e_j^i < e_p^{i+1}$*

From now on only safe MSNs will be considered.

Figure 8a shows an MSN model of a fragment of the Datalink Layer (DLL) of the Picoradio wireless network [15]. The DLL is responsible for establishing sessions during which pairs of neighbor nodes exchange the packets they receive from the upper layers. A scenario begins with one node, the session initiator (DL1), sending a channel request (!a) to another node (DL2). Then, upon reception of an acknowledgment (?b) from DL2 indicating that the session can start, DL1 transmits a data packet (!c) to DL2. DL2 replies sending to DL1 a data packet (!e) or just an acknowledgment (!d). In the first case DL1 replies sending an acknowledgment (!f) to close the session. Only one session at the time can be active for each node, i.e. a complete scenario is executed only after completion of the previous one. Hence, the MSN modeling the DLL scenarios is cyclic and safe.

## 5 Analysis of MSNs

Formally specifying protocol scenarios using MSNs allows to perform an early analysis and check that the system satisfies correctness properties such as deadlock-freedom, liveness and schedulability.

A MSN is a hierarchical model consisting of two layers:

- an upper layer with PNs semantics (below called PN layer), that is obtained removing the MSC labels from the MSN transitions
- a lower layer defined by the MSCs labeling the PN layer transitions (called MSC layer)

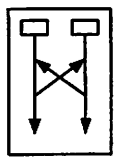

**Fig. 4.** Deadlock

The properties of the PN layer can be analyzed using PN analysis techniques such as unfolding to check safeness and boundedness [9]. However, analyzing only the PN layer is not sufficient to determine the properties of the MSNs model, which depend also on the MSC layer. For example, the PN layer may be deadlock-free, but a MSC associated with one of the transitions may contain a deadlock state as in the case shown in Figure 4 [2].

Our approach to MSNs analysis is divided in two steps: first the PN layer is analyzed to check high-level properties of the specification, then the whole specification is analyzed after a consistent PN model is derived using the procedure described in Section 6.3.

## 6 PNs Synthesis

The procedure to derive a PN implementation from a MSN is based on selecting PN patterns from a library and composing them to form a cover whose consistency with the given MSN is guaranteed by construction.

### 6.1 PN Patterns

**Definition 5.** *A PN pattern $\pi$ is a PN (P,T,F,l), where l associates places and transitions with a label.*

The definition of a PN pattern $\pi$ includes:

1. a set of transitions $T$, each modeling a set of computation or communication actions of a process,
2. a set of places $P$, each modeling either a I/O port buffer (e.g. the place labeled a in Figure 5a) or the state of a process (e.g. the two places without labels in Figure 5a).
3. a flow relation $F$, which defines the connectivity among places and transitions
4. a function $l$ that associates places and transitions with a label.

Using a notation similar to the one used for MSCs, in the rest of the paper $\pi$ transitions are labeled $!a$ if they model a *send message* $a$ event, $?a$ if they model a *receive message* $a$ event. Similarly, places modeling I/O port buffers are labeled with the same label(s) ($!a$ or $?a$) as the types of events they store.

PN patterns may cover single or multiple elements of a MSN model. The PN patterns used by our synthesis algorithm in Section 6.3 are called *Send* and *Receive*: the Receive pattern is used to cover receive events in a MSC and the Send pattern send events. Different types of PN patterns may correspond to different semantics interpretations of send and receive events. For example consider receive events. In MSCs messages are received by a process that any time after the reception of a message can read its content. Read is *blocking* if the process cannot perform any activity until the arrival of the corresponding event, *non-blocking* otherwise (in this case the process often reads the last occurrence of the same type of event). Figure 5a shows a PN model of a receive pattern with blocking read semantics. A place labeled $?a$ represents the input buffer at the port where messages a arrive and the transition labeled $?a$ represents the read action. The input and output places with no label represent respectively the state of the process before and after the transition is taken. Figure 5b shows a PN model of non-blocking read for the receive of message $b$. The two examples in Figure 5c and 5d represent blocking reads where the process is blocked until reception of multiple input events (AND semantics) or at least one of a set of input events (OR). Write is the action taken by a process when it shares a message with another process over a common channel. Write is blocking when the process cannot write until a condition becomes true, non-blocking when there is no condition. Figure 5e shows the corresponding PNs patterns that consist of a transition modeling the write action, a place modeling the buffer of the output port and, in case of blocking write, an input place holding a token modeling the precondition "space is available". Figure 5f shows the model of the transfer of a message over a channel taht consists of a transition that connects the output buffer of the sender to the input buffer of the receiver.

### 6.2 PN covers

The procedure to derive a consistent PN model from a MSN selects patterns from the library described above and composes them to generate a PN model xthat covers all the elements in the MSN model.

A set of patterns can be *composed* by merging a set of interface places $p_i$, one [4] for each pattern $\pi_i$, into a single place $p^*$, called *junction place*, whose preset and postset are respectively $\bigcup^{\bullet} p_i$ and $\bigcup p_i^{\bullet}$. The result of the composition is a PN $(P, T, F, l)$, that includes the transitions and places of all patterns and junction places in replacement of the interface places through which patterns are composed. Figure 6 shows an example of composition of the patterns $\pi_1$ and $\pi_2$ through the places $p_2^1$ and $p_1^3$.

[4] An interface place is either a source or a sink place. A pattern in our library has at least two interface places and no interface transitions.

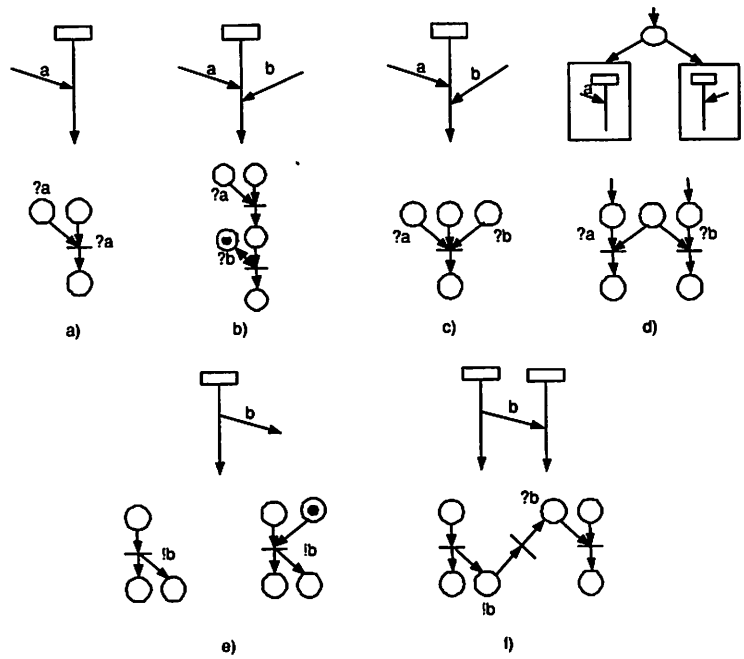


**Fig. 5.** Read: blocking (a), non-blocking (b), AND (c), OR (d), Write (e), Compute (f)

**Definition 6.** *Given a set of patterns* $\Pi = \{\pi_1, \pi_2, ...\pi_n\}$, $\pi_i = (P_i, T_i, F_i, l_i)$, *the merge relation* $\oplus$ *over* $\overline{P} = \bigcup_{i=1}^{n} P_i$ *is a set of tuples of interface places* $(p_1, p_2, ...p_m) \subseteq P_1 \times P_2 \times ... \times P_m$, *with* $j \neq k$.

Given a set of patterns $\Pi$, a merge relation $\oplus$ defines how they are composed; hence, a pair $(\Pi, \oplus)$ uniquely determines a PN. Such PN $(\Pi, \oplus)$ is a valid cover for a MSN if the patterns in $\Pi$ cover all the live events and messages of the MSN. A live event is an event that is included in at least one of the traces in $L(M)$. A live message is a message whose corresponding events are live. A MSN event ($!a$ or $?a$) is covered by a pattern $\pi$, if $\pi$ includes a PN transition with the same label. A MSN message is covered by a channel pattern that covers the I/O port buffer places and connects them with a transition.

**Definition 7.** *Given a MSN* $\tilde{M}$, *a pattern* $\pi_i = (P_i, T_i, F_i, L_i)$ *covers* ($\vdash^e$) *an event* $e \in \tilde{E}$, *i.e.* $\pi \vdash^e e$, *if* $\exists t_i \in T_i$ *s.t.* $l(t_i) = l(e)$.

**Definition 8.** *Given a MSN* $\tilde{M}$, *a pattern* $\pi_i = (P_i, T_i, F_i, L_i)$ *covers* ($\vdash^m$) *a message* $m \in \tilde{M}$, *i.e.* $\pi \vdash^m m$, *if* $\exists p_i, p_j \in P_i$ *s.t.* $l(p_i) = l(p_j) = l(m)$ *and* $\exists t \in T_i$ *s.t.* $p_i \in^\bullet t$ *and* $p_j \in t^\bullet$.

**Definition 9.** $(\Pi, \oplus)$ *is a cover of a MSN* $\tilde{M}$ *if*

- $\forall$ *live events* $e \in \tilde{E}, \exists \pi_i \in \Pi$ *s.t.* $\pi_i \vdash^e e$,

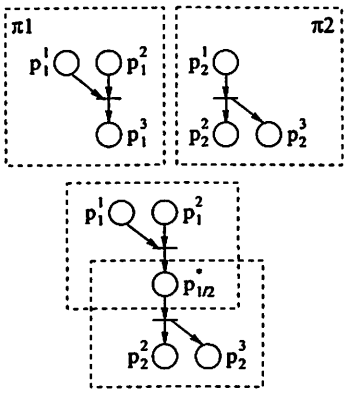


**Fig. 6.** Composing Patterns

– $\forall$ *live messages* $m \in \tilde{M}, \exists \pi_i \in \Pi$ *s.t.* $\pi_i \vdash^m m$,

Given a PN cover N, let $\Lambda(N)$ be the set of execution traces defined over the set of PN transition labels. N is consistent with $\tilde{M}$ if every firing sequence projected over $\Sigma^*$ is contained in $L(\tilde{M})$.

**Definition 10.** *Given a MSN* $\tilde{M}$ *and a PN* $N$, $N$ *is weakly consistent with* $\tilde{M}$ *if* $\Lambda(N) \subseteq L(\tilde{M})$, *strongly consistent if* $\Lambda(N) = L(\tilde{M})$.

### 6.3 Covering Algorithm

The procedure to derive from a MSN $\tilde{M}$ a consistent PN $N$ is based on a covering algorithm that covers the elements of $\tilde{M}$ with PN patterns. The algorithm derives a PN that maintains the same structure of $\tilde{M}$, i.e. includes as many PN fragments or subnets $Q_k$ as the number of process $Pr_k$ in $\tilde{M}$. These subnets are concurrent and interact only in correspondence with the exchange of messages in $\tilde{M}$. Each subnet $Q_k \subseteq N$ is defined by:

- a set of places $P_{Qk} = P_{QkS} \cup P_{QkP}$, modeling state ($P_{QkS}$) and I/O port buffers ($P_{QkP}$)
- a set of transitions $T_{Qk}$ modeling the actions of the process
- a flow relation $F_{Qk}$ that defines the connection among places and transitions
- control tokens whose position determines the value of the *program counter* during execution

Several procedures can be used to derive a PN cover depending on which library of patterns is used and how patterns are selected. The Covering Algorithm presented below uses only the Send and Receive patterns with blocking read and non-blocking write semantics defined in Section 6.1 to cover respectively send and receive events of an MSN, and covers one element (event or message) of the MSN model at the time. It generates a PN that corresponds to a unique

interpretation of the MSNs semantics where processes are blocked to await the incoming messages specified in the MSN one at the time.

Let

- $\hat{E}^i$ be the set of events covered up to the i-th iteration
- $(\Pi_i, \oplus_i)$ the PN cover generated up to the i-th iteration
- $\hat{T}^j \subseteq \tilde{T}$ the set including the first j MSN transitions covered
- $t_j \in \tilde{T}$ the j-th MSN transition being covered
- $t^0 \in \tilde{T}$ the MSN transition enabled in the initial marking and $p^0 \in \tilde{P}$ a place s.t. $\tilde{F}(p^0, t^0) = 1$ ($p^0$ predecessor place of $t^0$)
- $l(p)$ indicates the label of place $p$
- *token* a map that associates each MSN place with the number of tokens it holds

The Covering Algorithm selects one by one the transitions of the MSN and covers all the events of the corresponding MSC labels. It selects a transition if all its predecessor transitions have already been covered or if the predecessor place contains an initial token (transitions are selected in a "fireable" order). The procedures BeginCoverM (37-45) and EndCoverM (47-54) cover the relationships of a MSC with other MSCs, while the events within the MSC are covered one by one using Send and Receive patterns. First, the algorithm selects the next event ($e_i$) to be covered so that the set of covered events $E^i$ is a cut, i.e. there is no event $e' \notin E^i$ such that $e' < e_i$ (5). This ensures that the transitions of the PN cover are ordered as the corresponding events in $M_j$. Then, a pattern ($\pi_i$) is selected to cover $e_i$. If among the eligible events that form a cut a receive event is selected, a pattern representing the channel carrying the message is added to the cover (6-8). Individual send and receive events are communication primitives and therefore are coverable: the SelectPattern procedure selects a Transmit or a Receive pattern from the library to cover send and receive events respectively (10). The procedure *LabelPattern* (11) labels the selected pattern $\pi_i$. Transitions that cover send or receive events and their I/O port places are labeled like the corresponding events in $\tilde{M}$ (i.e. $!x$ or $?x$). *ComposePattern* (32-35) merges the selected pattern $\pi_i$ to the PN ($\Pi_{i-1}$, $\oplus_{i-1}$) according to the following rules:

1. $\oplus$ includes only interface places
2. $\oplus$ includes only pairs of I/O port buffer places or state places (i.e. it is not permitted to merge a state place with a port place)
3. $\oplus$ includes pairs of I/O buffer places with the same label

A new pattern is composed with the cover generated up to the previous iteration by merging its interface places with places in the cover that are marked with the identifier of the current MSC. Upon merge the identifier is used to mark the output state place of the pattern. Input port places are merged only to places in the cover having the same label, output places are merged only if they correspond to a merge. When all the events of a MSC $M_j$ have been covered the corresponding transition $t_j$ is marked as covered and another MSC among those with all the predecessor transitions already covered or with initially marked

**Algorithm 1** Derive a strongly consistent PN cover from a MSN

1: $i = j = 1$, $\hat{E}^i = \emptyset$, $\hat{T}^j = \emptyset$, $t_j = t^0$
2: **repeat**
3: BeginCoverM $(M_j)$
4: **repeat**
5: Select $e_i \in E(M_j) \cap \overline{\hat{E}^{i-1}}$ s.t. $\forall e' \in E(M_j) \mid e' < e_i$, $e' \in \hat{E}^{i-1}$ {$\hat{E}^i$ is a cut}
6: **if** $h(e_i) =$" *receive*" **then**
7: AddChannel
8: **end if**
9: $\hat{E}^i = \hat{E}^{i-1} \cup \{e_i\}$
10: SelectPattern$(e_i, Lib) \rightarrow \pi_i$
11: LabelPattern$(\pi_i) \rightarrow \pi_i'$
12: ComposePattern$(\pi_i', M_j, \Pi_{i-1}, \oplus_{i-1})$
13: i++
14: **until** there is no event $e_i \in E(M_j) \cap \overline{\hat{E}^{i-1}}$ that defines a cut
15: **if** $\forall e \in E(M_j), e \in \hat{E}^i$ **then**
16: Mark $t_j$ as covered
17: **end if**
18: EndCoverM $(M_j)$
19: $\hat{T}^j = \hat{T}^{j-1} \cup \{t_j\}$ {$t_j$ covered}
20: Select $t_j \in \overline{\hat{T}^j}$ if $\forall p^* \in^\bullet t_j$ either $p^*$ has an initial token or all $t \in^\bullet p^*$ are covered
21: j++
22: **until** $\forall t \in \tilde{T}$, $t \in \hat{T}^j$ {all $t$ in $\tilde{T}$ are covered}
23: For every choice add choice synchronizer shown in Figure 7d
24: Add a token in all places labeled with a place holding a token in $\tilde{M}$
25: **for all** strong sequential compositions $M_a \Rightarrow M_b$ (or $M_a \Rightarrow M_a$) **do**
26: Add one output place $p_{end}^{k,a}$ to each $t_{end}^{k,a}$
27: Add one input place $p_{start}^{k,b}$ to each $t_{start}^{k,b}$
28: Add one transition $t_{sync}^{a,b}$
29: Add a connection from each $p_{end}^{k,a}$ to $t_{sync}^{a,b}$ and from $t_{sync}^{a,b}$ to each $p_{start}^{k,b}$
30: **end for**
31:
32: **ComposePattern** $(\pi_i', t_j, \Pi_{i-1}, \oplus_{i-1})$
33: Merge state place of $Q_k$ labeled $t_j$ with input state place of $\pi_i$ labeled $t_j$
34: Remove label $t_j$ from junction place and label $t_j$ the output state place of $\pi_i$
35: Merge input port place (if any) $p'$ of $\pi_i$ with place in $Q_k$ having the same label
36:
37: **BeginCoverM** $(t_j)$
38: **for all** $Pr_k \in \tilde{M}$ **do**
39: Add to $T_{Qk}$ a transition labeled $t_{start}^{k,j}$ (corresponding to start event of process $Pr_k$) with one output place and as many input places as the number of predecessor places of $t_j$
40: Label $t_j$ the output place
41: Label each new input place with id of the corresponding places in $\tilde{P}$
42: **for all** pairs $p_1, p_2 \in P_{Qk}$ s.t. $l(p_1) = l(p_2)$ **do**
43: merge $p_1$ and $p_2$
44: **end for**
45: **end for**
46:
47: **EndCoverM** $(t_j)$
48: **for all** $Pr_k \in \tilde{M}$ **do**
49: Add to $T_{Qk}$ a transition labeled $t_{end}^{k,j}$ (corresponding to end event of process $Pr_k$) with one input place and as many output places as the successor places of $t_j$
50: Label each new output place with id of the corresponding place in $\tilde{P}$
51: **for all** pairs $p_1, p_2 \in P_{Qk}$ s.t. $l(p_1) = l(p_2)$ **do**
52: merge $p_1$ and $p_2$
53: **end for**
54: **end for**

predecessor places is selected (20). Note that if no uncovered events of a MSC can be selected because they do not define a cut, as in the case of presence of deadlocks (Figure 4) the procedure exits the MSC covering loop without marking the MSC as covered. In case of strong sequential composition between two MSCs, a transition $t_{sync}$ is introduced between the end transitions of the predecessor MSC to the start transition of the successor MSC (25-30).

In case the MSN includes MSCs in alternative composition, it is necessary to ensure that all the processes in the PN cover are allowed to make transitions corresponding to the same MSC selection. Therefore, the Covering Algorithm introduces a synchronization mechanism between processes in correspondence of each choice to avoid the generation of PN covers with false paths in case of non-local choices. A false path is a firing sequence that includes transitions that correspond to actions in conflicting MSCs. This situation occurs when different processes make conflicting choices. Consider the PN in Figure 7b. It is a cover of the MSN in Figure 7a, but it is not consistent because there is a firing sequence including !b and !e events (and one including !c and !d) that is not an MSN trace (events are in conflict). To avoid generating inconsistent PNs the algorithm imposes constraints on the processes to force them to choose the same MSC. These constraints can be implemented using the following mechanisms:

- the designer specifies one process (master) that makes the choice and communicates it to the other processes that are allowed to take the actions in the selected MSC (Figure 7c).
- the master is selected dynamically. Any process can choose the MSC and then communicate it to the other processes (Figure 7d). The Covering Algorithm described above uses this mechanism to synchronize non-local choices.

As an example of the application of the Covering Algorithm (CA) Figure 8b shows the PN cover derived from the MSN shown in Figure 8a. Note that when the safeness of MSN is ensured by a transitive closure of ordering relations between MSN events the unnecessary synchronizers (transitions $t_{sync}$) are omitted.

Below we demonstrate that the Covering Algorithm generates a PN cover $N$ that is strongly consistent with the original MSN $\tilde{M}$. $\tilde{M}'$ is the occurrence net derived from $\tilde{M}$, $N'$ is the PN cover obtained applying the Covering Algorithm to $\tilde{M}'$, $N''$ is the occurrence net derived from $N$. Theorem 1 proves that $N'$ and $N''$ are strongly consistent. Theorem 2 first proves that $\tilde{M}'$ and $N'$ are strongly consistent and then concludes, using the result of Theorem 1, that $\tilde{M}$ and $N$ are strongly consistent. Let us call $CA : \tilde{M} \rightarrow N$ a function that associates a MSN with the PN cover derived applying the Covering algorithm and $occ : N \rightarrow N'$ a function that associates a MSN or a PN with its corresponding occurrence net.

**Theorem 1.** *The occurrence net $N'$ generated applying the Covering Algorithm to the occurrence net $\tilde{M}'$ derived from a safe $\tilde{M}$ is strongly consistent with the occurrence net $N''$ derived from the PN $N$ generated applying the Covering Algorithm to $\tilde{M}$, i.e $N' = CA(occ(\tilde{M}))$ is strongly consistent with $N'' = occ(CA(\tilde{M}))$.*

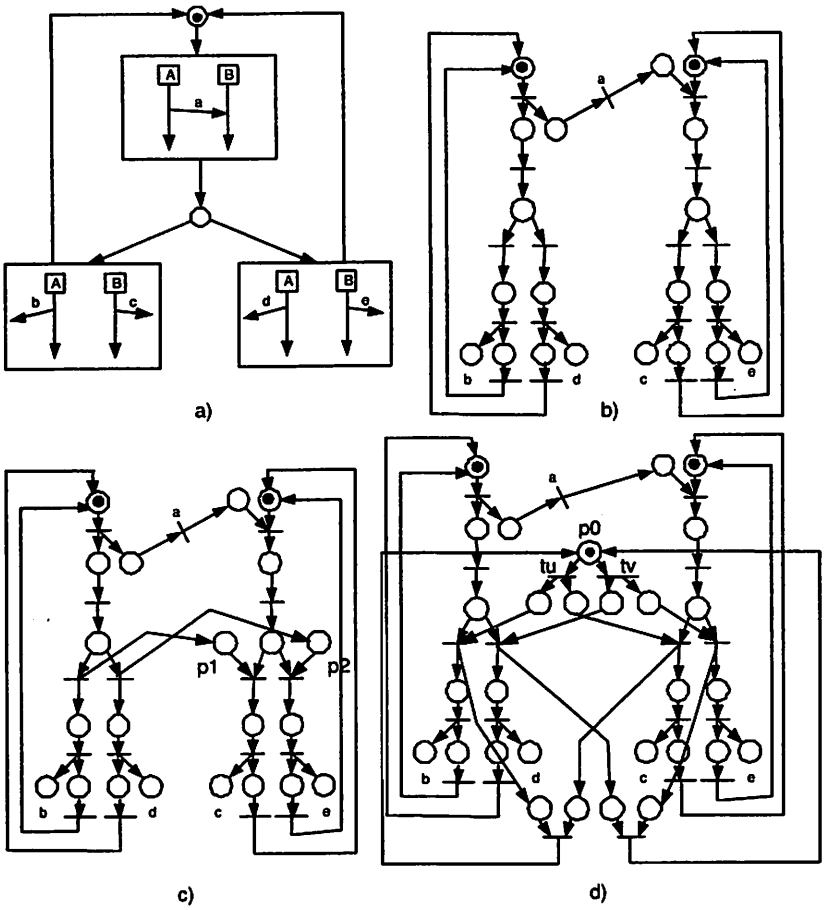


**Fig. 7.** Non-local Choices

The proof directly follows from two properties of the Covering Algorithm:

1) It introduces a one-to-one mapping between events in MSCs and corresponding labeled transitions of PN (this stems from a one-to-one correspondence between events and covering patterns)

2) Merging through places occurs in PN only due to merging through places in the PN underlying MSN. Indeed the only point of CA where merging by output places happens is EndCover routine and in this case merge occurs only if corresponding MSCs share the same output place in MSN. Therefore forward split in unfolding $\tilde{M}'$ happens consistently with a forward split in $N''$ and the one-to-one correspondence between events is preserved in corresponding occurrence nets.

Note that from the one-to-one correspondence between events in MSN and corresponding labeled transitions of PN immediately follows that PN $N$ covers MSN $M$.

**Theorem 2.** *The PN N derived applying the Covering Algorithm to a safe $\tilde{M}$ is a cover and is strongly consistent with $\tilde{M}$.*

*Proof.* By Theorem 1 it is sufficient to prove that the occurrence net $N'$ derived applying the CA to the occurrence net $\tilde{M}'$ derived from $\tilde{M}$ is strongly consistent with $\tilde{M}'$. The proof proceeds in 2 stages: 1) establishing the one-to-one correspondence between ordering and conflict relations between events of $\tilde{M}'$ and $N'$ and 2) proving the language containment between $\tilde{M}'$ and $N'$ and vice versa.

*Stage 1.* The following implications needs to be proven:

1. $e_0 < e_{i+1} \Leftrightarrow t_0 < t_{i+1}$
2. $e_0 \# e_{i+1} \Leftrightarrow t_0 \# t_{i+1}$

As in the occurrence net precedence, conflict and concurrency relations are disjoint from the above two one could conclude that $e_0 || e_{i+1} \Leftrightarrow t_0 || t_{i+1}$.

The proof for stage 1 is done by induction in which every induction step covers a single event in $L(\tilde{M}')$.

*Step 1).* Trivially true for traces of length one.

*Step i+1.)*

*Case 1.* $e_0 < e_{i+1} \Rightarrow t_0 < t_{i+1}$

Consider two events $e_0$ and $e_{i+1}$ in the cut defined up to the (i+1)-th step s.t. $e_0 < e_{i+1}$. Hence, either $e_0$ is a direct predecessor of $e_{i+1}$ or $\exists e_k$ covered during the k-th iteration ($k \leq i$) and s.t. $e_0 < ... < e_k < e_{i+1}$ and there exists no $e'$ s.t. $e_k < e' < e_{i+1}$. From $k \leq i$ it follows that $\exists$ a path from $t_0$ to $t_k$, so we can consider the case that $e_0$ is a direct predecessor of $e_{i+1}$.

1) $e_{i+1}$ is the start event of MSC $M_j$ for process $P_i$.

Then $e_0$ belongs to the MSC $M_i$ that is a predecessor of $M_j$ ($e_0$ and $e_{i+1}$ are from the same proces if $M_i \rightarrow M_j$ or might belong to different processes if $M_i \Rightarrow M_j$). If $M_i \rightarrow M_j$ then BeginCoverM and EndCoverM for each process connect the transition covering the last event in $M_i$ to the start transition in $M_j$. If $M_i \Rightarrow M_j$ BeginCoverM and EndCoverM connect the last transitions of all the processes in $M_i$ to the start transitions of all the processes in $M_j$. In both cases it follows that there is a path between $t_0$ and $t_{i+1}$.

2) $e_{i+1}$ is not the start event of a process. If $e_0$ and $e_{i+1}$ belong to the same process, then $e_{i+1}$ is transitively connected to $e_0$ by the ComposePattern function of CA that maintains the order of covering ($e_{i+1}$ is covered after $e_0$). If $e_0$ and $e_{i+1}$ belong to different processes, then they must be respectively the send and the receive events of the same message. There exist a path between corresponding $t_0$ and $t_{i+1}$ because the CA always introduces a transition connecting the transition for recieve event with the transition covering the send event.

*Case 2.* $t_0 < t_{i+1} \Rightarrow e_0 < e_{i+1}$

By induction, if there is a path from $t_0$ to $t_i$, then $e_0 < e_i$. The (i+1)-th step introduces a transition $t_{i+1}$ that covers $e_{i+1}$. Assuming there is a path from $t_i$ to $t_{i+1}$ we need to prove that $e_i < e_{i+1}$.

If $e_i$ and $e_{i+1}$ belong to the same process, then $e_i < e_{i+1}$ because at each step the event to be covered is selected to define a cut and a new pattern is composed merging an input place with the output place of the cover defined up to the previous step. If $e_i$ and $e_{i+1}$ do not belong to the same process, but are in the same MSC, the CA connects them only if they are send and receive event of the same message and therefore $e_i < e_{i+1}$. Now let us consider the case where $t_i$ and $t_{i+1}$ are connected by a path and $e_i$ and $e_{i+1}$ belong to different processes and different MSCs. This is possible only if there is strong synchronization between the MSCs, that implies also the ordering of the corresponding events.

*Case 3.* $e_0 \# e_{i+1} \Rightarrow t_0 \# t_{i+1}$.

$t_0 \# t_{i+1}$ either if $t_0$ and $t_{i+1}$ are in direct conflict or if there exist two transitions $t_u$ and $t_v$ that are in direct conflict and there is a path from $t_u$ to $t_0$ and a path from $t_v$ to $t_{i+1}$. Consider the MSC $Mp$ and $Mq$ s.t. $e_0 \in Mp$ and $e_{i+1} \in Mq$. Let us consider the general case where $Mp$ and $Mq$ are in conflict, but not in direct conflict, while $Mr$ ($Mr \rightarrow Mp$) and $Ms$ ($Ms \rightarrow Mq$) are in direct conflict. $Mr$ and $Ms$ are successors of the same choice $p$. Choices are covered by the CA introducing the structure shown in Figure 7d, which includes an additional choice place with two successor transitions $t_u$ and $t_v$ that are in direct conflict (assuming without loss of generality that each choice has at most two successor MSCs). By construction there is a path from $t_u$ ($t_v$) and every transition introduced by BeginCoverM to cover the start event of every process in $Mr$ ($Ms$). $e_0$ belongs to $Mp$ which is successor of $Mr$, therefore, regardless of which process $e_0$ belongs to, there is a path from $t_u$ to $t_0$. Similarly there is a path from $t_v$ to $t_{i+1}$. Hence, $t_0$ and $t_{i+1}$ are in conflict.

*Case 4.* $t_0 \# t_{i+1} \Rightarrow e_0 \# e_{i+1}$.

$t_0 \# t_{i+1}$ implies that there are two transitions $t_u$ and $t_v$ s.t. they are in direct conflict and have a common predecessor choice $p^*$ that corresponds to a non-local choice construct shown in Figure 7d. One could see that there is a one-to-one correspondence between choice place $p^*$ in PN $N$ and choice place in MSN $\tilde{M}$ because $p^*$ is introduced in correspondence with a choice that in the MSN models alternative composition between two MSCs, say $Mr$ and $Ms$. $t_u$ ($t_v$) has by construction a path to each transition introduced by BeginCoverM to start execution of a process in $Mr$ ($Ms$). The successors of $t_u$ ($t_v$) are only transitions covering events of $Mr$ ($Ms$) or events of their successor MSCs in $\tilde{M}'$. Since in the occurrence net all the successors of $Mr$ and $Ms$ are in conflict (due to the forward split on merge) corresponding $e_0$ and $e_{i+1}$ are also in conflict.

*Stage 2.* $L(\tilde{M}') = \Lambda(N')$.

*Case 1.* $L(\tilde{M}') \subseteq \Lambda(N')$.

By contradiction assume that there is a trace $l \in L(\tilde{M}') \cap \overline{\Lambda(N')}$. Since strong consistency is guaranteed by hypothesis for the traces including events covered up to step i, the trace violation occurs at step i+1, i.e. there is a trace $l$ including the event $e_{i+1}$ covered at step i+1 and there is no PN trace $\lambda$ having the corresponding transition $t_{i+1}$ in the (i+1)-th position. This could happen in two cases: a) if there is a pair of ordered events $e_0 < e_{i+1}$ in $l$ for which there is no path from $t_0$ to $t_{i+1}$, b) if there is a pair of concurrent events $e_0, e_{i+1}$

and the corresponding transitions $t_0, t_{i+1}$ are not concurrent. Both assumptions contradicts the one-to-one correspondence between ordering relation of $\tilde{M}'$ and $N'$ established in the previeous item of the proof.

*Case 2.* $\Lambda(N') \subseteq L(\tilde{M}')$.

By contradiction assume there is a trace $\lambda \in \Lambda(N') \cap \overline{L(M')}$. The violation occurs at step i+1, because there is no trace $l$ including event $e_{i+1}$ corresponding to transition $t_{i+1}$ in $\lambda$. This is again impossible because corresponding events and transitions are in the same ordering relations.

This concludes the proof of consistency.

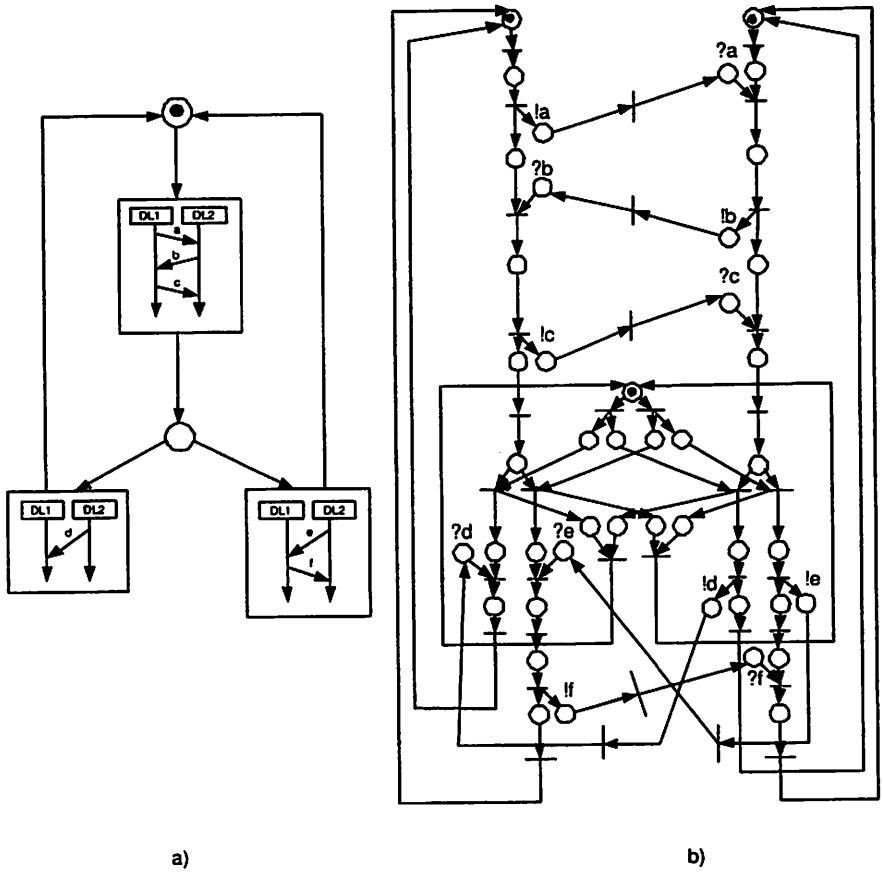


**Fig. 8.** Picoradio Protocol Example

## 7 Conclusions

In this paper we have presented an approach for PNs synthesis from MSCs models. First, we have defined consistency between an MSN and a PN and then proposed a pattern-based covering algorithm that derive a consistent PN from a given MSN. As future work we want to extend the approach to synthesize a consistent PN cover from bounded MSNs and use a broader library of PNs patterns to allow other interpretations of the MSCs.

## References


1. M. Abdalla, F. Khendek, and G. Butler. New results on deriving SDL specifications from MSCs. In *Proceedings of SDL Forum'99, Montreal, Canada*, June 1999.
2. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Third International Workshop, TACAS '97*, Apr. 1997.
3. J. Engelfreit. Branching processes of Petri Nets. *Acta Informatica 28*, pages 575–591, 1991.
4. P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definition for Message Sequence Charts. In *SDL 1993: Using Objects. Proceedings of the Sixth SDL Forum*, June 1993.
5. D. Harel and H. Kugler. Synthesizing state-based object systems from lsc specifications.. *International Journal of Foundations of Computer Science*, 13 (no.1):5–51, Feb. 2002.
6. S. Heymer. A semantics for MSCs based on Petri Net components. In *SAM 2000: 2nd Workshop on SDL and MSC*, June 2000.
7. G. Holzmann, D. Peled, and M. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, 1997.
8. ITU-T. Recommendation z.120. 1994.
9. A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. A structural approach for the analysis of Petri Nets by reduced unfoldings. In *Proceedings of 17th International Conference on Application and Theory of Petri Nets, Osaka, Japan*, June 1996.
10. K. Koskimies, T. Systa, J. Tuomi, and T. Mannisto. Automated support for modeling oo software. *IEEE Software*, pages 87–94, Jan. 1998.
11. I. Kruger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *International Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*, Oct. 1998.
12. J. Kuster and J. Stroop. Consistent design of embedded real-time systems with UML-RT. In *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*, May 2001.
13. J. D. Man. Towards a formal semantics of message sequence charts. In *Proceedings of the Sixth SDL Forum. SDL '93: Using Objects*, May 1993.
14. T.Murata. Petri nets: properties, analysis and applications. In *Proceedings of the IEEE*, Apr. 1989.
15. L. Zhong, J. Rabaey, C. Guo, and R. Shah. Data link layer design for wireless sensor networks. In *Proceedings of IEEE MILCOM 2001, Washington D.C. USA.*, Oct. 2001.