Implementing Message Sequence Charts in Promela
– Preliminary Extended Abstract –

Stefan Leue
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
`sleue@swen.uwaterloo.ca`

Peter B. Ladkin
Technische Fakultät
Universität Bielefeld
D-33501 Bielefeld, Germany
`ladkin@techfak.uni-bielefeld.de`

Abstract

We have previously defined a formal semantics for Message Flow Graphs and Message Sequence Charts, capturing most of the syntactic features contained in ITU-T recommendation Z.120. We discuss here a translation of MSCs into the language *Promela*, and report on experiments executing the Promela code using the *SPIN* simulator and validator.

1 Introduction

Message Sequence Charts (MSCs) describe sequences of message exchanges by communicating, concurrent processes. While other specification languages like SDL or Promela describe the process behaviour explicitly, leaving message flows to be inferred, MSCs specify explicit message flows while other details of process behaviour must be inferred from the specification. The syntax of MSCs is described in the ITU-T Z.120 document [CCI92].

MSCs are frequently used both formally and informally for the description of message flow amongst communicating, concurrent processes. MSCs can illustrate partial execution traces, as when simulating SDL specifications or when simulating Promela with the xspin system [Hol91]. They are often found in writing on telecommunications [Hol91, Tan89], and object-oriented specifications [RBP$^+$91, SGW94].

In an MSC, vertical lines represent process control flows, while horizontal or sloping arrows represent messages. See Figure 1. Although MSCs look simple, when certain syntactic features such as *conditions* are used, figuring out what they describe can become complicated and automated help is desirable. We describe a modeling of MSCs in the Promela language so that they can be *executed* using the spin validator. We used a straightforward translation of the finite-state interpretation of [LL95b].

We had three goals in mind:

- to simulate an MSC using `spin`, so one could, for example, obtain samples of the traces generated by an MSC;
- to automate the exploration of the state space of an MSC specification;

- to see what happens in a Promela implementation of some of the semantically tricky examples in [LL95a] and see what light it might shed, if any.

2 A few observations.

Our MSC formal semantics allows us to make the following observations:

- MSCs describe asynchronous communication, as in Z.120. However, our semantics also deals with synchronous communication (see [LL95b]). Thus so will the Promela translation. We will restrict ourselves here to discussing the asynchronous case.

- MSCs are inherently finite state. The state of an MSC specification (which in general consists of a collection of MSCs connected by so-called *conditions* as in Figure 10) is determined by the state of each process (i.e., at which point the process control lies in each process), and by the "state" of each of the messages in the system (whether the message is on the way, or not). As there are only finitely many processes, and each process has finitely many control states, and there are a finite number of message arrows in an MSC specification, we may suspect there are only finitely many states. For additional arguments for the finite-state interpretation, see [LL95a]

- Some people (including the committee responsible for Z.120) claim that the communication between processes in MSCs is buffered. If so, the behavior of the buffers is completely hidden. This could lead to trouble–generally, specifications should be explicit about everything they deal with. We suggest that for a specification style based on graphics, *what you see* should be *what you get*. If not, *what you get* are *problems.*

- Liveness properties in MSCs are underspecified. See [LL95b].

- Even somewhat restricted use of conditions yields specifications with problematic meaning. Sense may be made of them only if substantial assumptions are made about the behavior of the environment. Difficult cases arise from message cross-over, as well as non-local choice points as in Figure 10. See [LL95a].

3 Translating MSCs into Promela

3.1 Basic concepts

We model an MSC in Promela by instantiating a Promela process for each of the MSC processes at system setup time. This is implemented by an `init` clause in Promela. For the concurrent instantiation of two or more processes, we need to employ the `atomic` keyword. For example, to initialize an MSC with two processes `P1` and `P2` we write `init { atomic { run P1(); run P2() } }`. Messages can have types in Promela as well as in MSCs. We choose the `mtype` construct to specify the message types. `mtype = {a, b}` generates two one-byte integer constants with increasing values, here, `a`=1 and `b`=2.

To model the message behaviour of MSCs in Promela we choose channels with a capacity 1, one for each message arrow in the chart. This represents the invariant that any given message is either on the way (in which case there is a message of the expected type in the Promela channel), or not on the way [LL95b]. The channels must have type consistent with the message type. In Promela, channels are implemented as arrays of finite length ≥ 0. The declaration `chan vw = [1] of { byte }` defines a channel with name `vw` and capacity of one element of the message type – in this case, one byte. A Promela implementation of an MSC has the following overall syntactic structure:

- First, necessary data definitions, including the global channel declarations, denoted by the keyword `chan`.
- Next, the definition of the process bodies as indicated by the keyword `proctype`. In our examples the processes do not have parameters – all names used (i.e., the channels) have global scope.
- Finally, the instantiation of the whole system using an `init` statement.

3.2 Basic MSCs

MSC example 1

The left hand side of Figure 1 shows a basic MSC in which process `P1` sends a message of type `a` to `P2` and `P2` then sends a message of type `b` back to `P1`.

To implement an MSC, one must translate the graphic into a textual or mathematical representation. Z.120 proposes a textual syntax. We pointed out ambiguities in the mapping from the graphic to this textual representation in [Leu94] . The Z.120 textual syntax is therefore not suitable for our purposes.

In [LL95b] we defined the mapping of MSCs to Message Flow Graphs (MFGs), where each node in the MFG corresponds to a communication event in the corresponding MSC. An MFG corresponding to the MSC example 1 is given on the right hand side of Figure 1. The nodes are connected by directed arrows representing two relations on the set of nodes: the *next-event* (*ne*) relation representing the control flow in a process, and the *signal* (*sig*) relation representing message flows. In this paper, we draw *ne*-relation arrows as solid lines and *sig*-relation arrows as dashed lines.

Figure 2 shows the code for the MSC in Figure 1. We have define two processes, as in the MSC. The core of each process implements the communication behaviour, plus instructions to print output to the screen for debugging purposes (the `printf` statements). The statement `vw!a` denotes a *send* of a message of type `a` over the channel named `vw`, and `xy?b` denotes reception of a message of type `b` from channel `xy`.

Promela requires the use of the `atomic` keyword to ensure that operations inside the following curly parentheses are executed as an atomic action, without other interleaved events . We therefore ensure that the execution of the communication statements and the related `printf` statements are atomic events by use of the `atomic` keyword.

Reception of messages in Promela is not blocking. Thus, when executing a `xy?b` statement, a message of type b will be received if there is a message of that type at the head of the channel `xy`. However, if there is no such message at the head, the statement will nevertheless be executed, a

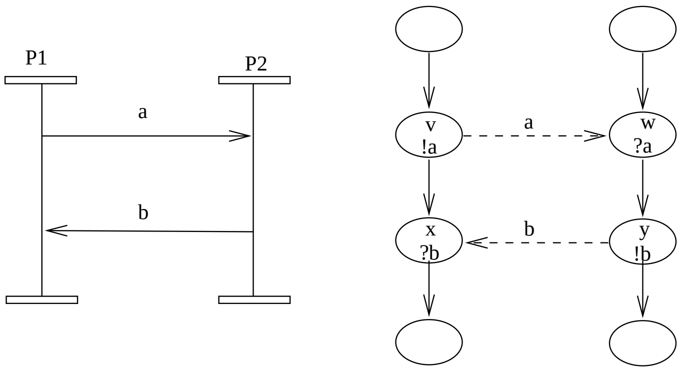

Figure 1: MSC example 1

```
mtype = {a, b};

chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ atomic {
     vw!a;
     printf("!a\n") };
  atomic {
     xy?[b] -> xy?b;
     printf("?b\n")}
}

proctype P2()
{ atomic {
     vw?[a] -> vw?a;
     printf("?a\n")};
 atomic {
     xy!b;
     printf("!b\n")}
}

init { atomic { run P1(); run P2() } }
```

Figure 2: Promela code for example MSC example 1

message of undefined type will be received, and process control will advance beyond the reception statement. This doesn't happen in MSCs, which block on *receive* of a message that isn't there.

In order to implement blocking on reception we use a guard, namely a predicate which checks whether a message of the suitable type is ready to be received. This is the `xy?[b]` statement, which is true if the first element of the channel `xy` is of type `b`, and false otherwise. The `->` operand serves as an enabling operator such that the operation on its right is only enabled if the guard on its left is *true*. In order to protect the compound guarded receive statement from undesired interleaving, it needs to be embraced by an `atomic` statement.

Execution MSC example 1. The execution of this example using spin yields exactly one execution trace:

```
swen12:/swen12/u/sleue/spin/specs/workshop.305 % spin msc1.prm
!a
?a
!b
?b
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.306 %
```

MSC example 2

The MSC example 2 in Figure 3 is similar to the example 1. However, we inverted the direction of the message arrow of type `b`. This MSC specifies a "message overtaking" – message `b` is sent later

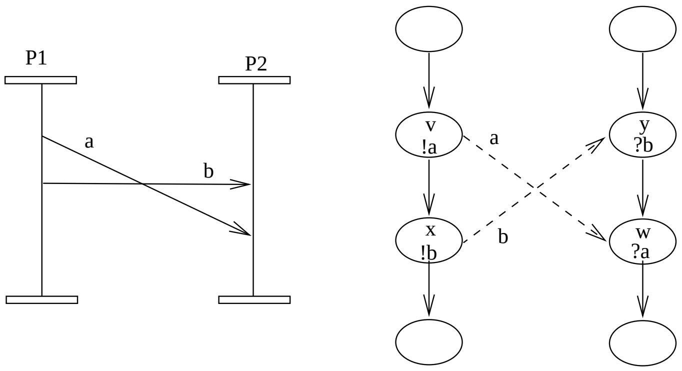

Figure 3: MSC example 2

```
mtype = {a, b};

chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ atomic {
    vw!a;
    printf("!a\n") };
  atomic {
    xy!b;
    printf("!b\n") }
}

proctype P2()
{ atomic {
     xy?[b] -> xy?b;
     printf("?b\n")};
  atomic {
     vw?[a] -> vw?a;
     printf("?a\n")}
}

init { atomic { run P1(); run P2() } }
```

Figure 4: Promela code for example MSC example 2

but received earlier than message a. This gives rise to the use of dedicated channels per message in Promela because message-overtaking within a Promela channel is not possible.

Execution MSC example 2. As for example 1 the execution of this example using spin yields exactly one execution trace:

```
swen12:/swen12/u/sleue/spin/specs/workshop.306 % spin msc2.prm
!a
!b
?b
?a
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.307 %
```

MSC example 3

Note that Example 3 allows two execution sequences: `<!a, !b, ?a, ?b>` or `<!a, ?a, !b, ?b>`. After the `!a` event has occurred, two independent events are enabled: the `!b` and the `?a` event. The Promela semantics specifies a nondeterministic choice in this situation.

Execution MSC example 3. Spin implements the nondeterministic choice in Promela by using a random (non-repeating) choice. Consecutive invocations of the simulator with the same Promela code can generate different traces.

```
swen12:/swen12/u/sleue/spin/specs/workshop.310 % spin msc3.prm
```

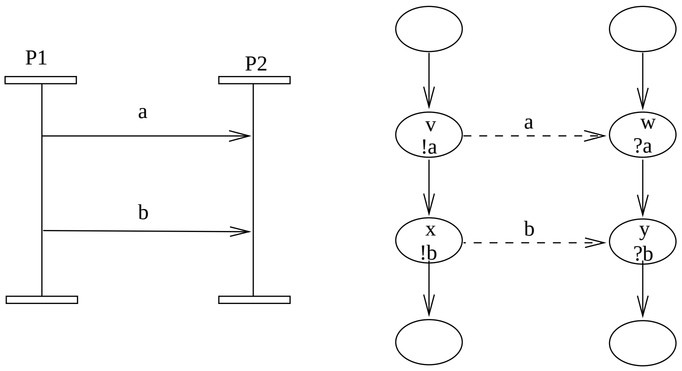

Figure 5: MSC example 3

```
mtype = {a, b};

chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ atomic {
    vw!a;
    printf("!a\n") };
  atomic {
    xy!b;
    printf("!b\n") }
}

proctype P2()
{ atomic {
     vw?[a] -> vw?a;
     printf("?a\n")};
  atomic {
     xy?[b] -> xy?b;
     printf("?b\n")}
}

init { atomic { run P1(); run P2() } }
```

Figure 6: Promela code for MSC example 3

```
!a
?a
!b
?b
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.311 % spin msc3.prm
!a
!b
?a
?b
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.312 %
```

Examples 4 and 5

Examples 4 and 5 in Figure 7 are similar to Examples 2 and 3, except that both message arrow are of the same type (a).

Execution MSC example 4 and 5. The Promela implementation generates similar outputs to those in examples 2 and 3 except for `!b` replaced by `!a` and `?b` replaced by `?a`. Example 4 generates <!a, !a, ?a, ?a> as trace, whereas example 5 generates <!a, !a, ?a, ?a> or <!a, ?a, !a, ?a> as traces, randomly chosing between them.

In [LL95a] we discussed an anomaly arising with these two examples. Both specifications stand for the same "code" with respect to the communication events, namely for two consecutive statements of type `!a` for the left and of type `?a` for the right process. In other words, the left and

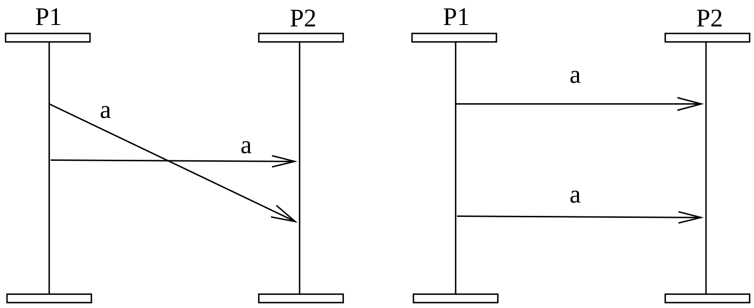

Figure 7: MSC example 4 (left) and 5 (right)

```
mtype = {a};

chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ atomic {
    vw!a;
    printf("!a, 1\n") };
  atomic {
    xy!a;
    printf("!a, 2\n") }
}

proctype P2()
{ atomic {
    vw?[a] -> vw?a;
    printf("?a, 1\n")};
 atomic {
     xy?[a] -> xy?a;
     printf("?a, 2\n")}
}

init { atomic { run P1(); run P2() } }
```

Figure 8: Promela code for example 4

```
mtype = {a};

chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ atomic {
    vw!a;
    printf("!a, 1\n") };
  atomic {
    xy!a;
    printf("!a, 2\n") }
}

proctype P2()
{ atomic {
     xy?[a] -> xy?a;
     printf("?a, 2\n")};
  atomic {
     vw?[a] -> vw?a;
     printf("?a, 1\n")}
}

init { atomic { run P1(); run P2() } }
```

Figure 9: Promela code for example 5

the right processes in both examples are code-identical. However, they do not allow the same set of traces. We conclude that there must be an implicit assumption about the environment which distinguishes the specifications. What would this be?

The implementations given here support this conclusion. In order to properly implement the desired behaviour we needed to define that both messages are implemented by different channels. The channels belong to the environment. The processes thus receive the messages from distinct environment entities, which allows for modeling the faster delivery of one message than of the other. This indicates that in an implementation we indeed need to exploit the environment to get the "right" behavior.

3.3 MSCs with Conditions - Iterations

Z.120 introduces so-called *global conditions* - labels covering all process axes of an MSC. Labels can be thought of as connectors, to continue an MSC with a label named `C`, another MSC with an initial condition with the same label may be attached. A message arrow is not allowed to 'cut through' a condition symbol. *Initial* conditions occur at the beginning of a particular MSC, and *final* conditions at the end. We only consider initial and final conditions. MSCs with conditions are not Basic MSCs.

The left hand side of Figure 10 shows an MSC with conditions. Informally, the condition `C` means that when any process has reached a final condition label `C`, then it may continue at any other initial condition label of the same name, provided that all the other processes follow this MSC-continuation when they reach the same point. In [LL95b] we suggested translating such an MSC into an iterating MFG as shown on the right hand side of Figure 10. In Promela, we model the iteration in the process code by a `goto`–label construct. As a convenience we choose the label names in the Promela code identical to the condition name in the MSC specification.

We must consider one aspect of the semantics of Promela more thoroughly. We see that a process may reach a *send* statement repeatedly. The sending primitive in Promela is blocking, i.e. when the channel `vw` is full, the statement `vw!a` blocks. MSCs, on the other hand, don't know the notions of channels and capacities. It would therefore be counterintuitive if an MSC could block on sending. This means that we need to add a void operation which is carried out if the receive operation blocks. Promela is a guarded command language. We use the Promela predicate `full(vw)` as a guard which is enabled if and only if the actual send operation `vw!a` blocks. We use the dummy `vw?[a]; printf("full, ")` statement to indicate that a write operation to a full channel was attempted.

Execution of MSC example 6. Below we give the prefix of a (supposedly) infinite execution trace of this example. Note that in conformance with our semantics the execution of one receive statement `?a` may disable $n \geq 1$ send operations `!a`.

```
swen12:/swen12/u/sleue/spin/specs/workshop.366 % spin msc6a.prm
!a, ?a, !a, full, !a, ?a, !a, full, !a, ?a, !a, full, !a, full, !a,
?a, !a, ?a, !a, ?a, !a, ?a, !a, ?a, !a, full, !a, full, !a, ?a, !a,
?a, !a, full, !a, ?a, !a, full, !a, full, !a, ?a, !a, full, !a, ?a,
!a, full, !a, ?a, !a, full, !a, full, !a, ?a, !a, ?a, !a, full, !a,
?a, !a, full, !a, full, !a, full, !a, full, !a, full, !a, full, !a,
full, !a, full, !a, ?a, !a, ?a, !a, full, !a, full, !a, full, !a,
```

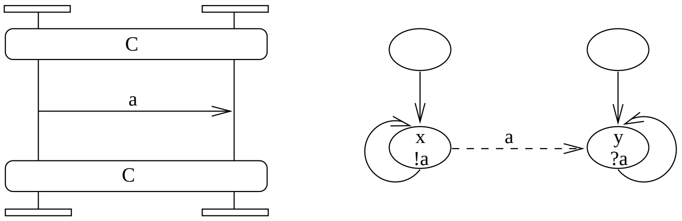

Figure 10: MSC example 6

```
mtype = {a};

chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ C:
  atomic {
    if
    :: vw!a
    :: full(vw) ->  printf("full,  ")
    fi;
    printf("!a, ");
    goto C}
}

proctype P2()
{ C:
  atomic {
    vw?[a] -> vw?a;
    printf("?a, ");
    goto C}
}

init { atomic { run P1(); run P2() } }
```

Figure 11: Promela code for MSC example 6

It is worth noting that there is nothing in the MSC specification which would make an infinite sequence of `!a` events an illegal trace. In other words, there is nothing in the MSC specification which would ever require a `?a` event to occur [LL95b]. However, the algorithm which resolves nondeterminism in Promela (based on a random number generator, we believe) appears to ensure some fairness condition on the nondeterministic choice alternatives.

3.4 MSCs with Conditions - Branching

The MSC example in Figure 12 uses conditions to specify *branching* behaviour. It may be interpreted to specify a very lightweight connection establishment protocol: process `P1` requests establishment of a connections by means of a `CR` protocol data unit (PDU), depending on a nondeterministic decision `P2` either acknowledges the establishment by a `CC` PDU, or refuses connection establishment by a `DR` PDU. In the first case the system execution is assumed to go into a data transfer phase (here indicated by a terminating condition called `END`), in the latter case the system goes back to a state from which connection establishment can be re-initiated.

In [LL95b] we suggested an operation called *unfolding* to translate an MSC with conditions into a branching an iterating MFG. We'll adopt that construction here. This means that when `P1` has sent the `CR` message, it will have to decide whether to move left (i.e. to continue with the MSC labeled "left"), or to move right. The right process is expected to make the same decision after receiving `CR`. Intuitively, we expect both processes move in the same direction each time they reach the point of decision.

Our semantics specifies a non-deterministic decision for both processes, but it does not specify how to implement the decision making. Let's assume the following strategy: `P2` makes a random decision whether to send `CC` or `DR`. We call this the *random-choice* strategy. `P1`, we assume, sits on condition `C2` monitoring the incoming messages. Depending on whether it sees a `CC` or a `DR` it will react accordingly and continue with either moving to condition `END` or to `C1`. Let's call this the *wait-and-see* strategy.

In the corresponding Promela example, the *random-choice* strategy is implemented by a `do ... od` construct embracing two vacuously enabled guarded commands, namely `vw!CC` and `xy!DR`. We rely on the randomness of the choice between both to be implemented by spin. The *wait-and-see* is implemented by a `do ... od` construct which embraces two complementarily enabled statements, namely `vw?CC` and `xy?DR`. In other words, `P1` follows faithfully the decision made by `P2`.

Executing the example with branching control. The execution traces show that the system runs in iterations until `P2` decides to send a `CC` which leads the system into a terminating state. Again, there is nothing in the MSC specification which would keep it from repeating a `CC` – `DR` loop forever.

```
swen12:/swen12/u/sleue/spin/specs/workshop.406 % spin msc9.prm
!CR, ?CR, !DR, ?DR, !CR, ?CR, !CC, ?CC, 3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.407 % spin msc9.prm
!CR, ?CR, !DR, ?DR, !CR, ?CR, !CC, ?CC, 3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.408 % spin msc9.prm
!CR, ?CR, !DR, ?DR, !CR, ?CR, !DR, ?DR, !CR, ?CR, !CC, ?CC, 3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.409 % spin msc9.prm
!CR, ?CR, !CC, ?CC, 3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.410 %
```

4 A Formalisation of the Translation for Basic MSCs

A precise formal definition of the translation from MSCs to Promela is indispensable when using both in an automated tool environment. In this section we present a formalisation for basic MSCs.

Preliminaries

Let M be an MSC.

Let	NE_M	$\triangleq$	the set of ne-edges of M
	SIG_M	$\triangleq$	the set of sig-edges of M
	N_M	$\triangleq$	$\{\Pi_1(n) : n \in NE_M\} \cup \{\Pi_2(n) : n \in NE_M\} / \{Top, Bottom\}$
		$\triangleq$	$\{\Pi_1(n) : n \in SIG_M\} \cup \{\Pi_2(n) : n \in SIG_M\}$
		$\triangleq$	the set of internal nodes
	L_M	$\triangleq$	$[n \in N_M \mapsto L(n)] \cup [m \in NE_M \mapsto L(m)] \cup [s \in SIG_M \mapsto L(n)]$ is the labelling function
	$L_M(S)$	$\triangleq$	$\{L(s) : s \in S\}$, for $S \subseteq N$
	CC_M	$\triangleq$	the set of NE_M-components
	$Name(P)$	$\triangleq$	the label of P, $P \in CC_M$

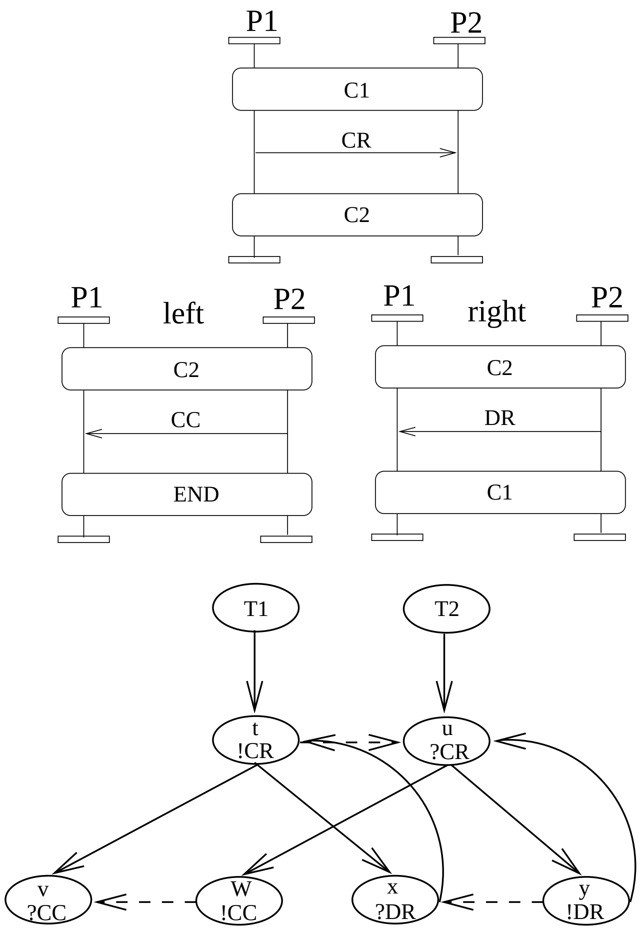

Figure 12: MSC example with branching control

```
mtype = {CR, CC, DR};

chan tu = [1] of { byte };
chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ C1:
  atomic {
    if
    :: tu!CR
    :: full(tu) -> printf("full, ")
    fi;
    printf("!CR, ");
    goto C2};

  C2:
  do
  :: atomic {
       vw?[CC] -> vw?CC;
       printf("?CC, ");
       goto END}

  :: atomic {
       xy?[DR] -> xy?DR;
       printf("?DR, ");
       goto C1}
  od;

  END: skip
}

proctype P2()
{ C1:
  atomic {
     tu?[CR] -> tu?CR;
     printf("?CR, ");
     goto C2};

  C2:
  do
  :: atomic {
       if
       :: vw!CC
       :: full(vw) -> printf("full, ")
       fi;
       printf("!CC, ");
       goto END }
  :: atomic {
       if
       :: xy!DR
       :: full(vw) -> printf("full, ")
       fi;
       printf("!DR, ");
       goto C1 }
  od;

 END: skip
}

init { atomic { run P1(); run P2() } }
```

Figure 13: Promela code for MSC example with branching control

Note

- $Top, Bottom \notin L_M$,
- Note $CC_M \in \wp(M)$ and $A \neq B \in CC_M \Rightarrow A \cap B = \emptyset$.

Specific symbol strings are written in `this font` and "quoted".
Let P be a sequence, $P = \langle a_1, \ldots, a_m \rangle$.
Let $\Pi_i(P) \triangleq a_i$ for $1 \leq i \leq m$.
We are constructing a string (a well-formed Promela program) from an MSC. Thus, we assume there is a Promela word corresponding to each label $L(h)$, where $h \in N_M \cup NE_M \cup SIG_M$. We use the notation $\$L(h)$ to represent this word. Similarly, we assume there is a Promela word for the name of each *sig*-edge sig, and we denote this word also by $\$sig$; likewise for process names $Name(P)$ – $\$Name(P)$.

Some more definitions:

$1..k$	$\triangleq$	$\{j : 1 \leq j \leq k\}$ for k an integer
cat(P)	$\triangleq$	"$\$L(a_1)$; ... ; $\$L(a_m)$"
lastitem(P)	$\triangleq$	$\Pi_2(P \lhd \mathit{length}(P))$
tailoff(P)	$\triangleq$	$P \lhd 1..(\mathit{length}(P) - 1)$
head(P)	$\triangleq$	$\Pi_2(P \lhd 1)$
rest(P)	$\triangleq$	$[n \mapsto (n+1) : n \in 1..(\mathit{length}(P) - 1)] \circ (P \lhd 2..\mathit{length}(P))$

Notice that *cat* concatenates strings with " ; " as a separator. The usual concatenation operator on strings is denoted by $\frown$.

The *size* of a set is the cardinal number of its elements. For $P \in CC_M$ and $n \in 0..\mathit{size}(P) - 1$, define by induction:

$$\begin{aligned}
\mathit{path}(P)(0) &\triangleq \langle Top, Bottom \rangle, \\
\mathit{path}(P)(n+1) &\triangleq \mathit{tailoff}(\mathit{path}(P)(n))^\frown \\
&\quad ^\frown\langle \iota x : x = \Pi_2(\iota y : \Pi_1(y) = \mathit{lastitem}(\mathit{tailoff}(\mathit{path}(P)(n)), Bottom \rangle \\
\mathit{path}(P) &\triangleq \mathit{path}(P)(\mathit{length}(P))
\end{aligned}$$

Finally, for S a set,

$$\textit{make-seq}(S) \triangleq \mathit{cat}(\langle p : p \in \mathit{choose}\, q : \mathit{sequence}(q)\ \&\ \mathit{range}(q) = S \rangle)$$

make-seq takes a set of things and turns them into some sequence. (The *choose* operator picks something that satisfies the following predicate if such a thing exists, and a random object in the universe if not.)

The Program

cat($\langle$
    "`mtype={`", *make-seq*($\{\$L(sig) : sig \in SIG_M\}$), "`}`",

$$
\begin{array}{l}
\textit{make-seq}\ (\{\textit{cat}(\langle \text{"}\mathtt{chan}\text{"}, \$L(\textit{sig}), \text{"}\mathtt{=\ [1]\ of\ \{\ byte\ \}\ }\text{"}\rangle) : \textit{sig} \in \textit{SIG}_M\}),\\
\textit{make-seq}\ (\{\\
\qquad\qquad \textit{cat}\ (\langle\ \text{"}\mathtt{proctype}\text{"}, \$\textit{Name}(P), \text{"}\mathtt{()\{}\text{"},\\
\qquad\qquad\qquad \textit{cat}(\ \langle \textit{cat}(\langle \text{"}\mathtt{atomic\{}\text{"}, \textit{send-or-receive}(p), \text{"}\mathtt{\}}\text{"}\rangle) : p \in \textit{path}(P)\rangle),\\
\qquad\qquad\qquad \text{"}\mathtt{\}}\text{"}\\
\qquad\qquad\quad \rangle)\\
\qquad\quad : P \in CC_M\}),\\
\textit{cat}(\langle \text{"}\mathtt{init\ \{\ atomic\ \{\ }\text{"}, \textit{cat}(\langle \text{"}\mathtt{run}\text{"}, \$L(P), \text{"}\mathtt{()}\text{"} : P \in CC_M\rangle), \text{"}\mathtt{\}\}}\text{"}\rangle)\\
\rangle)
\end{array}
$$

where

$$
\begin{array}{ll}
\textit{send-or-receive}(p) \triangleq & \\
& \textbf{if}\ \exists\ \textit{sig} \in \textit{SIG}_M : p = \Pi_1(\textit{sig})\ \textbf{then}\\
& \quad \textit{cat}(\langle \$\textit{sig}\frown\text{"}\mathtt{!}\text{"}\frown\$L(\textit{sig}), \text{"}\mathtt{printf\ (\ ``\ sent}\text{"}, \$L(\textit{sig}), \text{"}\mathtt{\backslash n\ ''\ )\ }\text{"}\rangle)\\
& \textbf{elseif}\ \exists \textit{sig} \in \textit{SIG}_M : p = \Pi_2(\textit{sig})\ \textbf{then}\\
& \quad \textit{cat}(\langle \$\textit{sig}\frown\text{"}\mathtt{?}\text{"}\frown\$L(\textit{sig}), \text{"}\mathtt{printf\ (\ ``\ received}\text{"}, \$L(\textit{sig}), \text{"}\mathtt{\backslash n\ ''\ )\ }\text{"}\rangle)\\
& \textbf{endif}
\end{array}
$$

5 Implementing Non-Local Choice

The MSC in Figure 14 is very similar to the example in Figure 12. It describes a simple data exchange protocol. `P1` transmits data by a `DA` PDU. Then, two scenarios are possible: either `P2` confirms receipt by a `DC` PDU or, due to some unspecified internal decision, `P1` will decide to request explicit acknowledgement from `P2` through an `RC` PDU. Intuitively we expect both processes to make the same left/right decision in every cycle through the system execution. A cycle is begun for one process by reaching condition `C1` and terminated by returning to `C1` or by reaching `END`. Now, if `P1` is in the n-th cycle, we expect `P2` to carry out the same left/right decision like `P1`, and vice versa.

This example reveals a non-local choice situation [LL95b]. We cannot rely on the wait-and-see strategy according to which one process waits for an incoming signal to move left or right. It may happen that process `P1` decides to go left and await a message, while `P2` decides to go right in order to await a message himself as well. This runs the system into a state of starvation.

In order to illustrate that neither the *random-choice* nor the *wait-and-see* implementation strategy yield the desired behaviour we included Figures 15 and 16. They describe two alternative implementations of the non-local choice. The first alternative (15) follows a wait-and-see strategy. The `do ... od` loop waits for either a send or an receive event to be executable – send events are always enabled, so at least a send will always be executed. The second alternative implements a random choice between going left or right immediately after executing `!DA` or `?DA`, respectively.

Executing non-local choice, alternative 1. As expected, the way to implement alternative 1 never leads into a starvation situation, due to the *wait-and-see* strategy employed. However, if the first of the invocations of the example shows that the system terminates at a point where a signal

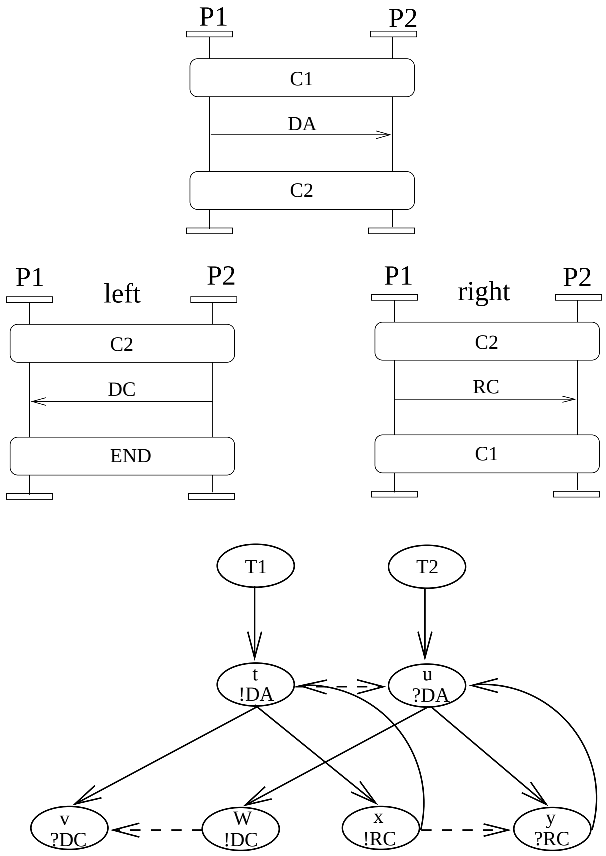

Figure 14: MSC example with non-local choice

```
mtype = {DA, DC, RC};

chan tu = [1] of { byte };
chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ C1:
  atomic {
    if
    :: tu!DA
    :: tu?[DA]; printf("full: ")
    fi;
    printf("!DA, ");
    goto C2};
  C2:
  do
  :: atomic {
       vw?[DC] -> vw?DC;
       printf("?DC - P1 left, \n");
       goto END}

  :: atomic {
       if
       :: xy!RC
       :: xy?[RC]; printf("full: ")
       fi;
       printf("!RC - P1 right, \n");
       goto C1 }
  od;
  END: printf("P1 at END\n");
       skip
}

proctype P2()
{ C1:
  atomic {
     tu?[DA] -> tu?DA;
     printf("?DA, ");
     goto C2};
  C2:
  do
  :: atomic {
       if
       :: vw!DC
       :: vw?[DC]; printf("full: ")
       fi;
       printf("!DC - P2 left, \n");
       goto END }
  :: atomic {
       xy?[RC] -> xy?RC;
       printf("?RC - P2 right, \n");
       goto C1}
  od;

 END: printf("P2 at END\n");
      skip
}

init { atomic { run P1(); run P2() } }
```

Figure 15: Implementing non-local choice, alternative 1

```
mtype = {DA, DC, RC};

chan tu = [1] of { byte };
chan vw = [1] of { byte };
chan xy = [1] of { byte };

proctype P1()
{ C1:
  atomic {
    if
    :: tu!DA
    :: full(tu) -> printf("full: ")
    fi;
    printf("!DA - ");
    if
    :: printf("P1 left, \n"); goto C2left
    :: printf("P1 right, \n"); goto C2right
    fi
    };
    C2left: atomic {
       vw?[DC] -> vw?DC;
       printf("?DC, ");
       goto END};
    C2right: atomic {
       if
       :: xy!RC
       :: full(xy) -> printf("full: ")
       fi;
       printf("!RC\n");
       goto C1 }
  END: printf("P1 at END\n");
       skip
}

proctype P2()
{ C1:
  atomic {
     tu?[DA] -> tu?DA;
     printf("?DA, ");
    if
    :: printf("P2 left, \n"); goto C2left
    :: printf("P2 right, \n"); goto C2right
    fi};
    C2left: atomic {
       if
       :: vw!DC
       :: full(vw) -> printf("full: ")
       fi;
       printf("!DC, ");
       goto END };
    C2right: atomic {
       xy?[RC] -> xy?RC;
       printf("?RC\n");
       goto C1};
  END: printf("P2 at END\n");
       skip
}

init { atomic { run P1(); run P2() } }
```

Figure 16: Implementing non-local choice, alternative 2

RC has been sent, but not yet received. This is due to both processes being in "different cycles": P1 has done <right, left>, whereas P2 has only done <left> when the system terminates. We question that is behaviour intended by the specification.

```
swen12:/swen12/u/sleue/spin/specs/workshop.416 % spin msc10.prm
!DA, ?DA, !DC - P2 left,
!RC - P1 right,
!DA, P2 at END
?DC - P1 left,
P1 at END
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.417 % spin msc10.prm
!DA, !RC - P1 right,
full: !DA, full: !RC - P1 right,
full: !DA, ?DA, full: !RC - P1 right,
?RC - P2 right,
!DA, !RC - P1 right,
?DA, !DC - P2 left,
!DA, full: !RC - P1 right,
full: !DA, full: !RC - P1 right,
P2 at END
full: !DA, ?DC - P1 left,
P1 at END
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.418 %
```

Executing non-local choice, alternative 2. A similar criticism applies to the second alternative. Here, too, we would expect that the three consecutive P1 - rights would be followed by an equal number of P2 -lefts before the system terminates. However, this is not the case as the first of the below given execution traces shows. Secondly, as the choice of whether to go left or right is done as a pure random choice the two processes may run into starvation. Starvation leads to abortion of a spin execution as seen in the second invocation of the example. This is indicated by spin issuing a timeout message.

```
swen12:/swen12/u/sleue/spin/specs/workshop.434 % spin msc10c.prm
!DA - P1 right,
?DA, P2 right,
!RC
!DA - P1 right,
full: !RC
full: !DA - P1 right,
full: !RC
full: !DA - P1 right,
full: !RC
full: !DA - P1 left,
?RC
?DA, P2 left,
!DC, ?DC, P2 at END
P1 at END
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.435 % spin msc10c.prm
!DA - P1 left,
?DA, P2 right,
timeout
#processes: 3
               DA = 1
               DC = 2
```

```
                RC = 3
                queue 1 (tu):
                queue 2 (vw):
                queue 3 (xy):
 12:    proc  2 (P2) line  51 "msc10c.prm" (state 23)
 12:    proc  1 (P1) line  20 "msc10c.prm" (state 18)
 12:    proc  0 (:init:) line  59 "msc10c.prm" (state 4)
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.436 % spin msc10c.prm
!DA - P1 left,
?DA, P2 left,
!DC, ?DC, P2 at END
P1 at END
3 processes created
swen12:/swen12/u/sleue/spin/specs/workshop.437 %
```

6 Conclusion

We have discussed the translation of Message Sequence Charts into Promela and illustrated their execution using Spin. While the translation of condition-free, i.e. non-iterating and non-branching MSCs is relatively straightforward, we discovered some intricate problems related to liveness and non-local choice issues.

Liveness. One may use temporal claims or LTL formulas in Promela to define certain liveness properties. We do not yet know the comparative expressiveness of the various alternatives for MSCs when implemented in Promela. This must, however, play a significant role in choosing a translation. It's the area on which MSCs are most weak.

Non-local choice. We've presented some unsatisfactory codings of the non-local choice example. In [LL95b] we discuss the possibility to use global history variables in order to implement non-local choice properly. We will expand out translation to incorporate this but as we've shown in [LL95b], the incorporation of global history variables may lead to an infinite state space which contradicts our previous observation that MSCs are inherently finite state.

References

[CCI92] CCITT. Recommendation Z.120: Message Sequence Chart (MSC). CCITT, Geneva, 1992. (CCITT has been renamed ITU-T).

[Hol91] G. J. Holzman. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[Leu94] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering.* Doctoral dissertation, University of Berne, Switzerland, December 1994.

[LL95a] P. B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques, VII*, Proceedings of the Seventh International Conference on Formal Description Techniques. Chapman & Hall, 1995.

[LL95b] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7, 1995. To appear.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley & Sons, Inc., 1994.

[Tan89] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International, 2nd edition, 1989.