Implementing and Verifying MSC Specifications Using PROMELA/XSPIN

Stefan Leue and Peter B. Ladkin

Abstract. We discuss a translation of Message Sequence Charts (MSCs) into the language PROMELA (we call this translation an '*implementation*') that is consistent with the formal semantics we have previously defined for Message Flow Graphs and Message Sequence Charts, which handled the syntactic features with mathematical import from ITU-T recommendation Z.120. We report on experiments executing the PROMELA code using the XSPIN simulator and validator. In previous work we found that potential process *divergence* and *non-local choice* situations impose problems on implementations of MSCs, and we discuss how these impact our PROMELA translation and suggest solutions. Finally, we show how to model-check liveness requirements imposed on MSC specifications. We use the PROMELA models obtained from our implementation, describe how to use control state propositions based on these models, use Linear Time Temporal Logic formulas to specify the liveness properties, and demonstrate the use of XSPIN as a model checker for these properties.

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 Recommendation Z.120 [**IT96**]. MSCs are frequently used both formally and informally for the description of message flow amongst communicating, concurrent processes. They have found their way into many software engineering methodologies and toolsets, such as SDL tools and environments [**OFMP$^+$94, Lab95**], Object-oriented methodologies [**RBP$^+$91, Jea92, SGW94**], tools to analyse the design of message exchanges at early stages in the software lifecycle [**Hol96, AHP96**], and methods describing design patterns [**BMR$^+$96**].

In previous work we defined a finite state semantics for Message Sequence Charts [**LL95b**], and discussed in [**LL95a**] implications of the MSC notation as

1991 *Mathematics Subject Classification.* Primary 68Q60, 68Q55, 68Q10; Secondary 68N99, 03B45.

The first author was supported in part by the National Science and Engineering Research Council of Canada (NSERC).

defined in Z.120. Although Basic MSCs look simple, syntactic features such as *conditions* or branchings which are defined for *High-level MSCs* can make it harder to figure out what these latter describe and automated help is desirable. We describe a translation of MSC specifications into PROMELA so that they can be simulated and validated using the XSPIN tool. We call this translation an *implementation*, and discuss in this paper what design choices need to be made.

There are three major ways in which MSCs are used:

1. to visualise actual system execution, during debugging and program understanding (as in [**Hol96, AHP96**]),
2. as a language to document early design decisions (as in ObjecTime [**SGW94**], EFD [**Hol96**]),
3. to document test cases or functional-validation criteria that an implementation must satisfy (ROOM [**SGW94**], SDT [**AB**]).

Items 1 and 3 concern finite execution scenarios in which one event node corresponds to a single event and a single message. Z.120 calls such finite MSCs *Basic MSCs* (BMSCs). For an example of a BMSC, see the left-hand side of Figure 1. The intuitive meaning of this BMSC is as follows. There are two processes `P1`, `P2` with vertical 'time lines', bounded above and below. `P1` sends a message of type `a` to `P2`, which (asynchronously) receives it, and then sends a message of type `b` to `P1`, which (asynchronously) receives it. This example should suffice to understand what BMSCs are supposed to mean. BMSCs do not have any branching or iteration. In [**LL95b**] we described how to represent BMSCs algebraically as so-called basic *Message Flow Graphs* (MFGs), and then translated MFGs into (global-) state machines. We are not primarily interested in BMSCs here since their meaning is straightforward and implementation is trivial. We concern ourselves with MSCs describing repeating or infinite behavior, in which a given event 'node' typically represents many repeating events in an execution sequence (as, for example, statements in loops in procedural programming languages).

Composition of MSCs. When MSCs are used at early system design stages to represent desired behaviour of the system, many BMSCs will typically be written, and these sets of BMSCs only make sense if some sort of relationship between individual BMSCs is intended. In practice, a single BMSC often corresponds to a particular software feature, described by a finite message exchange *scenario.* In Figure 12, for example, the MSC labelled `MSC1` represents the scenario in which process `P1` is requesting connection establishment from process `P2` by sending a `CR` message (*connect request*), while `MSC2` shows `P2` answering by a `CC` message (*connect confirm*) and `MSC3` shows `P2` answering by a `DR` message (*disconnect request*). Intuitively, these scenarios form the building blocks for a simple connection-establishment protocol, provided their relationship is properly defined.

Explicitly to represent this relationship formally calls for some sort of composition operator. The latest version of the Z.120 standard introduces so-called *High-level MSCs* (HMSCs) to specify the interrelation of MSCs[1]. HMSCs may represent branching and iterating behaviour. Composition is described by a graph that we will call an *HMSC graph.* We use the following definition of HMSC. An HMSC

[1] According to Z.120, both *conditions* in BMSCs and *composition* expressed by HMSCs can be used simultaneously. The expressive capability of HMSCs is greater because it allows expression of the n-fold repetition of a BMSC M for a fixed, finite n.

graph is a graph with *start nodes* (nodes with only out-edges), *end nodes* (nodes with only in-edges) and *interior nodes* (nodes with both out- and in-edges). Each interior node is labelled either with a BMSC or with another HMSC graph. We assume that there is at least one start node and one interior node. The intuitive meaning is that the edges of an HMSC graph indicate control flow between BMSCs (or other HMSCs) that are the node labels. The composition of two BMSCs is thus represented by an edge between the corresponding nodes in the HMSC graph[2].

We define an *MSC specification* to be a collection S of one or more BMSCs, plus an HMSC graph G whose nodes are labelled with members of S. The MSC Specification in Figure 1 specifies a finite scenario in which process `P1` sends an `a` message to `P2` which replies by sending a `b` message. The MSC Specification in Figure 12 specifies branching and iterating behaviour. It can be interpreted as specifying a simple connection establishment protocol in the following way: the request for connection establishment (node `MSC1` in the HMSC on the right hand side of Figure 12) can be followed either by connection confirmation (node `MSC2`) which means that the protocol ends, or by request of disconnection (node `MSC3`) which means that a new connection establishment must be attempted (loop back to node `MSC1`).

HMSC graphs hold out the hope for a clearer notion of MSC composition than possible with *conditions*. We are particularly interested in MSC specifications which involve HMSCs containing cycles: it is intuitively only possible to 'visit' a given MSC node (corresponding to a communication event in the MSC) more than once in an execution sequence if there is some control path leaving that node which returns to it; the control path is some (here unspecified) construct of the BMSC of which that node is part plus edges in the HMSC graph.

Many of the problems in interpreting iteration and branching in MSCs noted in [**LL95b, LL95a**] occur regardless of the syntactic form in which a composition is proposed. The current Z.120 HMSC proposal does not appear to contain syntactic restrictions that would avoid many of these interpretation problems. We will show that they become apparent when pursuing PROMELA simulations of MSC specification.

Motivation. Our reasons for implementing (i.e., simulating) MSCs are:

1. we want to demonstrate the practical use of MSCs in behavior specification; in particular,
2. we want to demonstrate the practical use of our semantics;
3. the synthesis of process code from MSCs requires an understanding of what behavior they express and the assumptions they embody, which can be most easily seen from simulation;
4. this exercise in translating MSCs into process code reveals underspecified assumptions in the intended meaning of the MSCs;
5. we want to generate models that can be used for model checking properties of MSC specifications.

Choice of PROMELA/*XSpin.* We chose PROMELA/XSPIN because PROMELA provides all necessary concepts (sending and receiving primitives; parallel

[2]An HMSC graph that intuitively represents a single BMSC may be constructed as follows: a single start node leads to a single interior node labelled with the BMSC, leading to a single end node. Thus these particular HMSC graphs can be identified intuitively with the BMSCs with which they are labelled.

and asynchronous composition of concurrent processes; and communication channels) that were necessary to implement MSC specifications. Furthermore, the X-SPIN [**Hol91, Hol**] tool allows for randomly simulating PROMELA specifications, which helps in debugging, and for model-checking properties expressed as LTL formulas. The availability of suitable language features and the simulation capability distinguishes PROMELA/XSPIN from other finite-state modelling-language and model-checker packages such as SMV [**McM93**]. The communication primitives and channels that are readily available in PROMELA would need to be hand-coded into SMV specifications in order to obtain models identical to the ones we obtain from our PROMELA implementation.

2. A few observations

Our MSC formal semantics [**LL95b**] 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 does the PROMELA translation. We restrict ourselves to discussing the asynchronous case here.
- MSCs are inherently finite-state. The state of an MSC specification 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). Since 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, and these are all the meaningful 'parts' of an MSC, there is a strong *prima facie* argument that there are only finitely many states. For additional arguments for a finite-state interpretation, see [**LL95a**].
- Some 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 feel 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 basic MSC composition 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 in Figures 3 and 7), as well as non-local choice points as in Figure 14. See [**LL95a**].

3. Implementation of Basic MSC specifications.

3.1. Message Flow Graphs. MSC specifications are graphical objects - ink on paper, lines on a screen. To implement an MSC, one must translate the graphical 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.

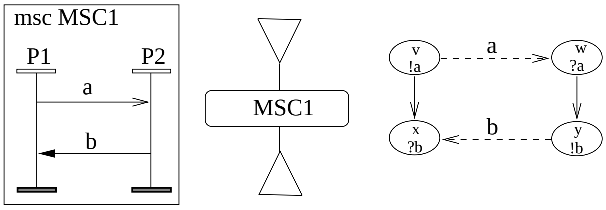

FIGURE 1. MSC example 1

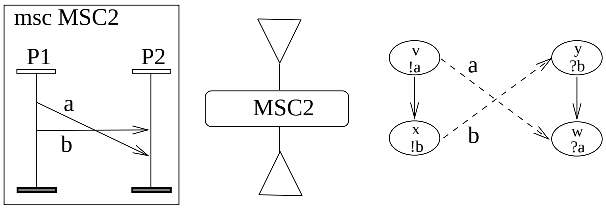

FIGURE 2. MSC example 2

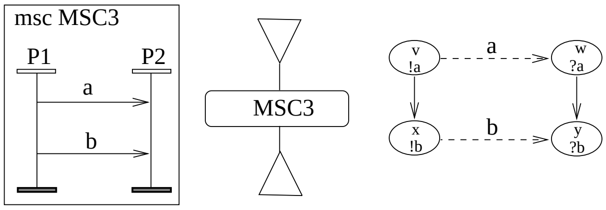

FIGURE 3. MSC example 3

```
mtype = {a, b};

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

proctype P1()
{ atomic {
     vw!a;
     printf("!a\n") };
  atomic {
     xy?[b] -> 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 4. PROMELA code for MSC example 1

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. The translation of an MSC specification into a corresponding MFG is described in [**LL95b**]. While there the concept of *conditions* was used to specify possible continuations of one MSC by another, in this paper we use the HMSC concept, recently introduced into Z.120, to specify composition of basic MSCs. The translation procedure from MSC specifications into MFGs based on HMSCs is a straightforward extension of the *"unfolding"* operation as defined in [**LL95b**], and we will not elaborate on this translation here.

We distinguish two types of MSC specifications:

1. MSC specificatios may describe *finite* system executions. This is depicted in Figure 1. In this case the corresponding MFG is a finite, cycle- and branching-free graph.
2. MSC specifications may represent *branching* and *iterating* behaviour, which is reflected by the branching and cyclic structure of the corresponding MFG (see the MFG in Figure 13 that corresponds to the MSC specification in Figure 12).

Note that neither the HMSC concept nor the condition concept in Z.120 imply any sort of synchronization between processes in an MSC specification when sequential composition or branching occurs[3]. As we shall see later, this has bearing on our implementation choices.

MFGs will be the basic underlying data structure for the implementation although we will in most cases not explicitly refer to them. In the remainder of this section we discuss the implementation of basic MSC specifications. In Section 4 we will discuss the implementation of iterating and branching specifications.

3.2. Basic implementation concepts. We model an MSC specification in PROMELA by instantiating a PROMELA process for each of the MSC processes at system-setup time[4]. 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 names `a` and `b` and increasing values `a`=1 and `b`=2.

To model the message behaviour of MSCs in PROMELA we choose channels with 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

[3] C.f. [**IT96**]: "*A sequential execution of two nodes that are related by an edge is described byt the* seq *operator.*" and "*The* seq *operator denotes the weak sequencing operation where only events on the same instance are ordered*". Concerning the semanitcs of conditions, c.f. [**IT95**]: "*Note that the semantics of a chart containing conditions is simply the semantics of the chart with the conditions deleted from it.*"

[4] Note that a 'process' is a connected component of the *ne* relation of an MFG [**LL95b**]

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 (c.f. Figure 5):

- 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.

3.3. Basic MSC specifications. *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`. Figure 4 shows the code for the MSC in Figure 1. We have defined 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`.

The semantics in [**LL95b**] relies on the interleaving model with communication events as atomic actions[5]. 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` debugging statements are atomic events by use of the `atomic` keyword.

Reception of messages in PROMELA is not blocking. Thus, when executing an `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 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. The execution of this example using SPIN yields exactly one execution trace.

MSC example 2. The MSC example 2, Figure 2, is similar to the MSC 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 but received earlier than

[5] [**Sel96**] chooses a receive event and a subsequent send event to be executed in one atomic transition of a ROOM actor. However, they derive code for one single test actor from an MSC specification. Interleaving semantics, however, is only relevant for systems consisting of more than one concurrent processes.

```
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 5. PROMELA code for 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 {
     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

message a. This gives rise to the use of per-message dedicated channels in PROMELA because message-overtaking within a PROMELA channel is not possible. As for MSC example 1, execution using SPIN yields exactly one execution trace.

MSC example 3. 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. Spin implements the nondeterministic choice in PROMELA by using a random (non-repeating) choice. As expected, experimentation with the SPINsimulator shows that two different traces will be generated (see also [**LL96**]).

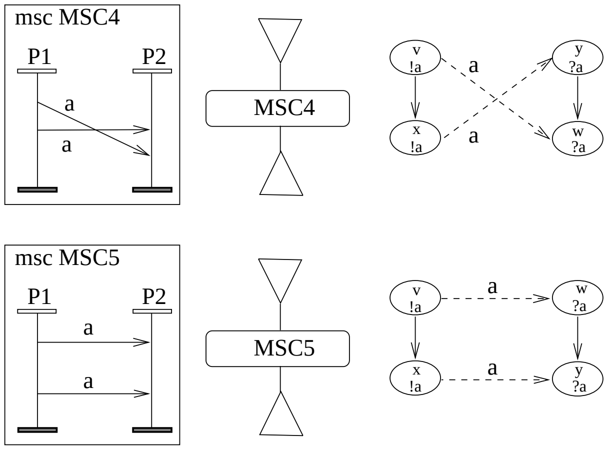

FIGURE 7. MSC example 4 (top) and 5 (bottom)

```
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 ex. 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 ex. 5

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

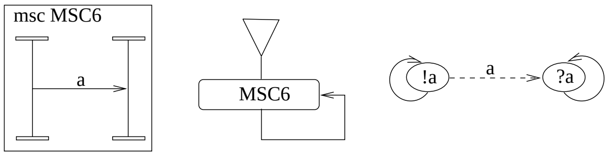

FIGURE 10. MSC example 6

The implementations given here support this conclusion. In order properly to 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 "expected" behavior.

4. Iterating and branching MSC specifications

As discussed above, HMSCs may specify a composition of basic MSCs such that an iterating or branching system is described; iterating and branching MSC specifications translate into iterating and branching MFGs (Figure 10).

4.1. Iteration. Figure 10 shows an HMSC which allows `MSC6` to be followed by `MSC6`. In PROMELA, we model the iteration in the process code by a `goto`–label construct. 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 according to Z.120, on the other hand, do not have 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 send operation blocks. We use the PROMELA predicate `full(vw)` as a guard which is true 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.

In the following 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,
```

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.

4.2. Branching. The MSC example in Figure 12 specifies *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

```
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

data unit (PDU); depending on a non-deterministic 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 triangular symbol indicating an HMSC end-node), in the latter case the system returns to a state from which connection establishment can be re-initiated.

In [**LL95b**] we suggested an operation called *unfolding* to translate an MSC specification into a branching and iterating MFG. We'll adopt that construction here. In order to adapt the unfolding operation for an HMSC, we shall assume that each BMSC in the HMSC has the same processes (i.e., the same number of processes with the same process labels). The unfolding is a single graph that is intended to reflect exactly the behavior intended by the HMSC. For example, when `P1` in Figure 12 has sent the `CR` message, it will have to decide whether to move left (i.e. to continue with the MSC labeled "`MSC2`"), or to move right. The right process is expected to make the same decision after receiving `CR`. Intuitively, the choice of each processes of which 'branch' to follow at the decision point must correspond with the choice (to be) made by the other process at the same point, during the same iteration. These choices can differ from iteration to iteration, as long as the processes make corresponding choices.

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, remains in its post-`CR` state in `MSC1`, before the 'choice point', monitoring the incoming messages. Depending on whether it sees a `CC` or a `DR` it will react accordingly and continue with either moving through `MSC2` to the 'triangle' or through `MSC3` and back to the beginning of `MSC1`. Let's call this the *wait-and-see* strategy. Obviously, for a consistent implementation of the branching there has to be one process that performs a random choice between sending different messages, and all other processes follow suit by implementing a wait-and-see strategy. We will later see MSC specifications for which such a consistent implementation is impossible.

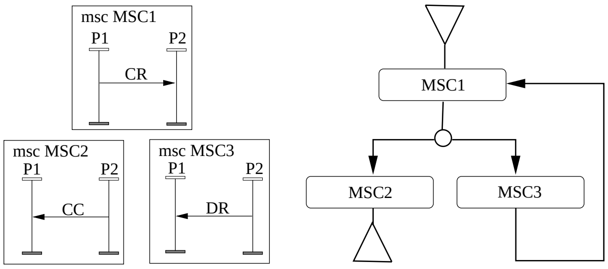

FIGURE 12. MSC Specification example 7.

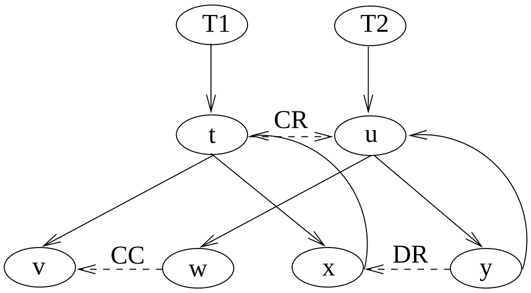

FIGURE 13. Message Flow Graph for example 7.

In the corresponding PROMELA example in Figure 20, 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 following 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. (We used a variant of the code in Figure 20 with debugging `printf` statements to generate the following output).

```
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.408 % spin msc9.prm
!CR, ?CR, !DR, ?DR, !CR, ?CR, !DR, ?DR, !CR, ?CR, !CC, ?CC, 3 processes created
```

4.3. Summary of implementation choices. We summarise the implementation decisions discussed so far. First, the graphical-object MSC specification is translated into a corresponding *Message Flow Graph*, using unfolding. Then:

- Every process (Z.120 terminology: ‘instance’) in an MSC specification is mapped to exactly one PROMELA process. The PROMELA processes are instantiated concurrently when the whole PROMELA specification becomes incarnated, see the `{ atomic { run P1(); run P2() } }` statement in Figure 20.

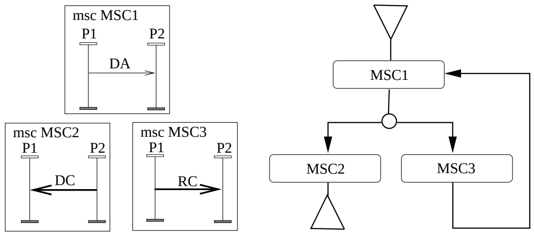

FIGURE 14. MSC specification with non-local choice

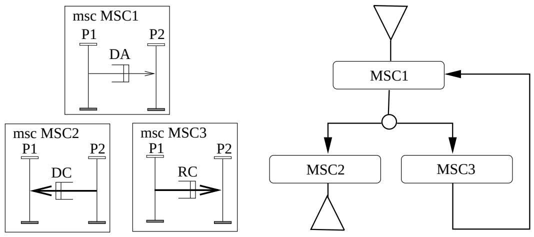

FIGURE 15. MSC example with non-local choice and explicitly represented channels

- Message arrows are represented by PROMELA channels. This allowed for modeling so-called "message crossing". Also, as messages in MSCs can be exclusively either on-the-way or not, the capacity of these channels is defined to be 1.
- Message *send* and *receive* events are modeled by the corresponding PROMELA statements (e.g., `tu!CR` and `tu?CR`, respectively).
- Branching in MSCs is modeled using PROMELA labels and `goto` statements.
- It is necessary to ensure that certain sequences of PROMELA statements were executed atomically using the `atomic` clause.
- There is no notion of 'channel' explicit in MSCs, therefore there can be no blocking-send statement in the corresponding PROMELA code. We use `full(..) -> skip` statements to model the non-blocking send of an MSC.
- Receive statements, however, are blocking. This is implemented using a PROMELA guard-statement pair, see for an example the `vw?[CC] -> vw?CC` statement pair in Figure 20. It is particularly important to guarantee atomicity of these guard-statement pairs.

5. Implementing Non-Local Choice

The MSC in Figure 14 is similar to the example in Figures 12 and 13. It describes a simple data exchange protocol. `P1` transmits data by a `DA` PDU. Then, two scenarios are possible: either `P2` confirms receipt with a `DC` PDU or, due to some unspecified internal decision, `P1` requests explicit acknowledgement from `P2` through an `RC` PDU.

The implementation of this MSC in PROMELA (Figure 16) illustrates some of the intricacies of using HMSCs (or 'conditions') to compose BMSCs to form MSC specifications: the "*n-th-cycle-same-choice*" condition seems to be what users intuitively understand this MSC to express. This condition says that when P1 has gone through n iterations of the cycle described by the MSCs MSC1 and MSC2, and if P1 is ahead of P2, then later when P2 reaches the n-th cycle it will make the same left-right decision in a post-DA state that P1 made in its n-th cycle. If in an implementation P1 were to decide to "go left" (continue with MSC MSC2), and P2 were to decide to go right (MSC3) at the same choice point on the same iteration, then the system would block (it would either dead- or livelock depending on the implementation) because both processes would be waiting for a signal to be sent by the other process. This is not what users understand this MSC to express. Such an "*n-th-cycle-same-choice*" branching was called a 'non-local choice' in [**LL95b, LL95a**] because these choices somehow have to be synchronised by both processes despite the fact they occur at different points in the execution sequence.

It turns out that the synchronisation required by this kind of MSC cannot be implemented in a local, non-coordinated, fashion, in contrast to the situation for MSCs in which a process may branch control without synchronising with other choices, as for example in Figure 12. [**LL95c**] discussed two somewhat unsatisfactory variants of the implementation of the example in Figure 14.

Executing Non-local Choice with history variables. As described in [**LL95a**], existence of a history variable that records the left-right choices is implied by the intuitive meaning of the choice-synchronizing processes P1 and P2. We argued that the length of this variable is finite but potentially unbounded. As PROMELA only allows for the description of finite-state systems, we must bound the size of the history variable, thereby only approximating the history variable algorithm informally described in [**LL95a**].

We use $N+1$ global history variables: variables i1 ... iN and variable hist. In our example, $N = 2$. Process Pk keeps track of the iteration it is on by setting history variable ik. However, the choice history only records M previous choices (M is thus the bound on the size of the choice-history variable), so ik is approximated by nk = i$k \bmod M$. Let $0..N = \{x \mid x \in \mathit{Naturals} \;\&\; 0 \leq x \leq N\}$. The choice-history variable is hist: $0..(M-1) \rightarrow \{0,1\}$, initialised to 0. (There is only one choice-point per iteration in this example and only a two-way choice – hist would generally be a doubly-indexed array hist: $0..(M-1) \times$ *choice-point-labels* $\rightarrow$ *branch-labels*.)

A value hist[k] $= 0$ indicates that the process first reaching this branch point in the k'th cycle went 'left'; a value of 1 indicates that it went 'right'. (Since the initial value of *hist* also has a meaning as a branch choice, Pj checks whether hist[k] has been previously set by checking whether some other nk is greater than nj, which must be done in any case, as we see next.) Pk is the only process setting ik and nk and may only set hist[m] if n_m $\geq$ n_j for $k \neq j$ when ik= k, otherwise it must follow the decision indicated by the value of hist[k].

Figure 16 shows the suggested implementation of the non-local choice example. P1 sends DA as an atomic event. In the next atomic step, P1 checks whether it is allowed to set the history variable, or whether it has to follow the path determined by P2 as recorded in the history variable. If P1 may determine which branch to take in the n-th cycle, it will make a nondeterministic decision.

Implications of the History Variable length. We saw that the capacity of the history variable determines the amount by which processes P1 and P2 can 'diverge'. The PROMELA code will only correctly simulate the MSC specification if at all times the difference between the number of the cycle that P1 is on and the number of

```
/* Length of history variable */
#define N 8
/* Channel capacities */
#define Ctu 3
#define Cvw 1
#define Cxy 3

mtype = {DA, DC, RC};
chan tu = [Ctu] of { byte };
chan vw = [Cvw] of { byte };
chan xy = [Cxy] of { byte };

int hist[N];  /* history variable */
int n1;       /* # of iterations through non-local choice by P1 */
int n2;       /* # of iterations through non-local choice by P1 */
int i1;       /* act index to hist for P1 */
int i2;       /* act index to hist for P2 */

proctype P1()
{ C1:
    tu!DA;
  atomic{
    if
    :: (n1 < n2) && (n2 - n1) <= N ->   /* P1 lacks behind */
       if                               /* Decide according to hist */
       :: hist[i1] == 0 -> goto C20     /* go `left' */
       :: hist[i1] == 1 -> goto C21     /* go `right' */
       fi
    :: (n1 >= n2) && (n1 - n2) < N ->   /* P1 is ahead */
       if                               /* Random choice betw 0 and 1 */
       :: hist[i1] = 0; goto C20
       :: hist[i1] = 1; goto C21
       fi
    fi;
    C20:
       n1 = n1 + 1; i1 = n1 % N;
       vw?[DC] -> vw?DC;
       goto END;
    C21:
       n1 = n1 + 1; i1 = n1 % N;
       xy!RC; goto C1}
  END: skip}

proctype P2()
{ C1:
  atomic{
    tu?[DA] -> tu?DA};
  atomic{
    if
    :: (n2 < n1) && (n1 - n2) <= N ->
       if
       :: hist[i2] == 0 -> goto C20
       :: hist[i2] == 1 -> goto C21
       fi
    :: (n2 >= n1) && (n2 - n1) < N ->
       if
       :: hist[i2] = 0; goto C20
       :: hist[i2] = 1; goto C21
       fi
    fi;
    C20:
       n2 = n2 + 1; i2 = n2 % N;
       vw!DC; goto END;
    C21:
       n2 = n2 + 1; i2 = n2 % N;
       xy?[RC] -> xy?RC; goto C1}
  END: skip}
init { n1 = 0; n2 = 0; i1 = n1 % N; i2 = n2 % N;
       atomic { run P1(); run P2() } }
```

FIGURE 16. Implementing non-local choice using a history variable with bounded length and channels with bounded capacity

the cycle that P2 is on is $\leq M$. If one process runs ahead of the other by more than M cycles, the PROMELA code will not correctly simulate the MSC specification. A bound on the 'cycle difference' may currently not be specified in MSCs. In fact, as we have noted in previous work, liveness properties such as requiring that a sent message is eventually received are underdetermined by the current standard. It is easy to see that a 'cycle-difference' bound ensures progress of both processes and therefore that this requirement entails a liveness property.

Experimental results. The experimental simulation with XSPIN shows that this implementation of the history variable algorithm satisfies the *n-th-cycle- same-choice* condition but does not prevent the system from blocking. This is in accordance with our semantics and has the following explanation. Consider the following scenario: the system in Figure 14 starts executing, P1 makes n consecutive `right` decisions, and P2 is in its m-th cycle ($n > m+1$). Now, P2 queries the global history variable and follows the `right` decision that P1 made in the m'th cycle and receives RC. In the $m+1$'st cycle, P2 will again perform a `right` decision, as determined by the global history variable, but find no message RC to be received. This is because, as we argued in *op. cit.*, communication in MSCs in non-buffered and therefore $n > 1$ repeated sendings of a message by a given 'MSC arrow' (= message instance) can be received by one receive event. (We retain in the system state a single copy of a message instance that has been sent but not received, and remove this copy when the message is received. A second message-send of an unreceived message instance does not change the system state because the instance is already recorded in the state. One must also be careful to distinguish the *contents* of a message, which may be identified with message type in MSCs, from a message instance. An MSC may contain multiple arrows representing the sending of a single contents. These multiple arrows are different instances of the contents and the MSC state retains a copy of the instance in our semantics.)

6. Introduction of Channels with Capacities

The example in the previous section shows that recording the history of choices that the system makes with respect to non-local choice situations does not suffice to provide a non-blocking interpretation (i.e., a set of execution sequences, none of whom block) of the specification. To provide a non-blocking interpretation, we may add another history variable, a counter variable recording the number of *sendings* and *receivings* of message instances. [**LL95b**] contains simple examples of potential MSC execution sequences during which this variable would be unbounded; this happens when repeated message instances are continually sent faster than they are received, throughout a non-terminating execution. Employing such counting variables while continuing to admit such executions yields an infinite-state system. However, [**LL95a**] argued for the inherent finite-stateness of MSC specifications. So adding such counter variables is not obviously consistent with other requirements on the interpretation of MSCs.

Making Channels explicit. A chief argument against the use of message queues for the interpretation of the communication mechanism in MSCs follows from the *what-you-see-is-what-you-get* (WYSIWYG) requirement on specifications and specification languages. For a visual graphical specification style like MSCs, this means either not adding information that is not explicit in the diagram, or extending the language to capture such information. (We argued in [**LL95b, LL95a**] for an extension to capture underdetermined liveness properties, but for omitting histories since they weren't already explicit. We also noted, and again above, that histories were required in any case reasonably to interpret some already-standardised MSC syntactic constructs.)

This WYSIWYG requirement on specification languages entails that when introducing history variables for messages in MSCs their existence should be made explicit in the graphical representation of MSCs, which means an extension to the syntax defined in the Z.120 standard. Figure 15 shows a possible syntactic representation of channels. We map each message arrow onto one channel. In some situations it may be useful to require different messages to be sent across one channel. This may syntactically be done by attributing channel symbols with name labels, and to understand messages crossing channel symbols with the same name label to be passed along the same channel.

Some Suggested Criteria for Introducing Channels.

1. Channels serve messages following a first-in-first-out strategy.
2. Channels should be free of loss: this requirement may be weakened later.
3. Channels have a specific (finite or infinite) capacity.
4. The semantics of the MSC send primitive should be changed from non-blocking to blocking. Z.120 MSCs have no notion of channels and capacities, therefore there was no point to defining the send primitive as blocking. However, with the introduction of channels and capacities this now makes more sense. Channels with infinite capacity never block on a send, but the blocking becomes important as we introduce channels with finite capacity.
5. The definitions in [**LL95b**] would no longer be appropriate for MSCs-with-channels. However, a semantics can easily be given, in particular by translating an MSC specification into a collection of *Communicating Finite State Machines* [**BZ83**]. Furthermore, a formal operational semantics for PROMELA is currently under development [**NH96**], hence MSCs-with-channels could inherit a formal semantics from the PROMELA translation once this PROMELA semantics is given.
6. The MSCs obtained by introducing unbounded channels cannot be implemented using PROMELA, whose expressive capabilities are limited to finite-state systems.

Benefits. The suggested introduction of channels with finite capacities does not guarantee deadlock-freedom for MSCs[6]. However, we conjecture that blocking due to non-local choices can be avoided by this mechanism. We do not have space here for a more-detailed study of how one might introduce channels, and what properties those channels should have. We wished mainly to note how introducing them would solve a problem with 'anomalous' blocking execution sequences. The problem could also be solved by simply accepting the blocking execution sequences as valid executions of the MSC specification.

7. Finite State Implementation

Two of the constructs we have suggested lead to an infinite-state model: non-local-choice *history* variable(s), and *channels* associated with message arrows. However, finite-state-space validation techniques as well as an implementation using

[6]For necessary and sufficient criteria for MSCs to be deadlock-free see [**LS**].

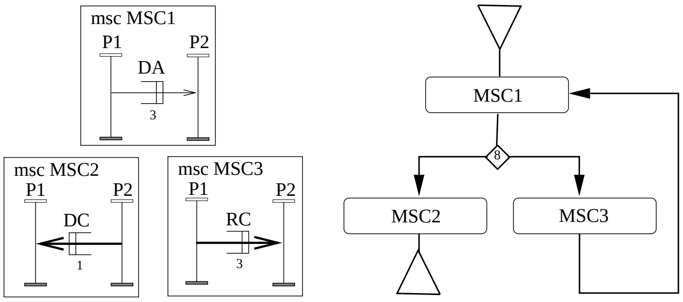

FIGURE 17. MSC example with non-local choice, explicitly represented channels, and explicit capacities for the non-local choice history variable (=8) and channels (=3, 3 and 1)

PROMELA require a finite-state-space model. Any PROMELA implementation must therefore limit both the capacities of the channels and the length of the history variables for all non-local choice situtations to finite values. We represent these restrictions using appropriate labels on channel symbols and the final-condition symbol leading into a non-local choice situation, as seen in Figure 17. Figure 16 shows the corresponding PROMELA source code.

Making the History Variable Length explicit. Following the WYSIWYG requirement discussed earlier, not only the existence of channels and their capacities should be made explicit in MSC specifications, but also the existence of a history variable and its capacity. Figure 17 shows a possible representation for the presence of a history variable (the diamond shaped connector) and the limitation of its length to 8 (the number in the diamond's interior).

Experiment and limitations. Limiting the capacity of history variables and channel capacities leads to a number of limitations when executing the PROMELA implementation.

1. Figure 18 shows an execution trace generated by XSPIN based on the implementation of the MSC specification in Figure 17 as given in Figure 16. Events in a trace generated by XSPIN are totally ordered (by virtue of their absolute distance to the top of the beginning of each process axis). The left axis corresponds to the PROMELA process generated as a father to processes `P1` (middle axis) and `P2` (right axis). Note that the maximum divergence between the `P1` and `P2` is 3 as a consequence of the capacity of the now-blocking communication channel `xy`, which is 3.
2. As the send primitive is now blocking, processes will not send to a full channel. This excludes a number of interleavings as possible traces of the system. Note that in the implementation without bounded channel capacities a subtrace (`!DA, !RC`)^4 could be part of an admissible execution sequence, this is not the case for the example in Figure 17.
3. As argued eralier, the size of the history variable limits the divergence of processes `P1` and `P2`. In the example in Figures 16 `P1` could be at most 8 cycles ahead of `P2`. However, the limitation of the capacity of channel `xy` is more constraining, limiting the maximal divergence to 3. Both channel capacity as well as history variable length determine the potential divergence of the processes.

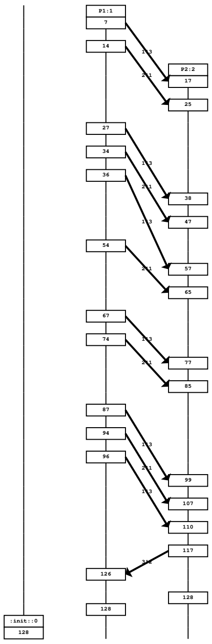

FIGURE 18. Trace of MSC specification generated by random simulation using XSPIN

8. Verifying Temporal Properties.

The translation of MSCs into a language like PROMELA can prove helpful in two main ways. First, as mentioned above, the translated PROMELA model allows simulation and animation of an MSC specification, which may help in debugging early system designs. Second, as we argued in [**LL95a, LL95b**] MSCs underspecify liveness properties, and temporal logic can be used to specify the additional liveness properties. Desired liveness properties specifiable in linear-time temporal logic (LTL) may be checked in PROMELA, thus PROMELA can support this extension of the MSC language.

It is possible using LTL to specify properties on the PROMELA model generated from an MSC specification that this model cannot guarantee. For example, consider MSC 1 in Figure 12. Suppose we wish to assert that process `P2` will always eventually send a message of type `DR`. XSPIN is a state-based, and not an event-based verifier, so we need to define state predicates specifying the control-state of the process with respect to the events defined in the MSC. Define a state predicate $ta\_$ (for '*taken*') such that ta_x holds iff the last state transition was a sending of a message of type x. The desired assertion is expressed by the LTL formula $\Box\Diamond ta\_\mathtt{DR}$. (We also define and use references to the control state of individual processes in the PROMELA code as XSPIN-LTL propositions.) The property that a message of type `DR` will always eventually be sent cannot hold, because `P2` may eventually decide to execute the left path that describes transmission of a `CC` message followed

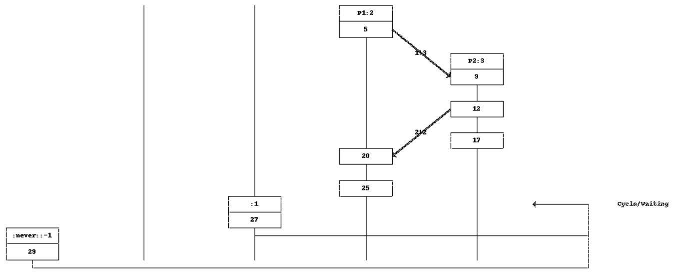

FIGURE 19. MSC showing trace leading to a state violating `[] <> aftsDR`.

by termination, along which path the system cannot ever again reach a state in which *ta*_`DR` holds.

The XSPIN environment allows model checking of LTL formulas based on PROMELA models [**GPVW95, Hol**]. LTL formulas can be entered in XSPIN, and a preprocessor translates them into so-called *never* claims [**Hol91, Hol95**]. How can a basic proposition like *ta_x* be defined as a basic proposition in PROMELA? We make use of a predefined PROMELA predicate of the format `process_name[pid]@label_name`. `process_name` is the name of a process as defined in the `proctype` clause; `pid` is a process identification number, generated by incrementing a counter starting at 1 each time a new process of any type is incarnated; `label_name` is a statement label in the PROMELA code. In order to refer to such a predicate, it is assigned a name `pred_name` by a `#define` clause.

In [**LL95c**], the communication event as well as the control-flow branching was included inside a PROMELA `atomic` clause. In order to implement the *ta_* predicate properly using PROMELA labels, we removed the the control-flow branch statement from within the `atomic` clause and labeled it. Figure 21 shows the PROMELA code implementing the MSC in Figure 12. Labels `aftersCR`, `aftersCC` and `aftersDR` denote points in the process control flow as needed to define *ta_*[7].

As expected, the XSPIN verifier detects a violation of the LTL claim `[] <> aftsDR` which is the translation of the LTL formula $\Box\Diamond ta\_\mathtt{DR}$. For debugging, XSPIN can run a guided simulation into the state that violated the claim, and the violating trace is illustrated using an MSC (see Figure 19). The online use of this MSC enhances debugging because placing the mouse on individual MSC events highlights the PROMELA code (in another window) corresponding to those events; and it attempts to indicate how the temporal property is violated.

We ran a few more temporal properties through the XSPIN verifier to experiment. Table 1 lists the results. In particular, Property 3 expresses an important consistency condition for the protocol represented by the original MSC. This property basically states that once a `CC` (connect confirmation) has been sent it is not possible to send a `DR` (disconnect request) afterwards.

9. Summary and Outlook

We noted that simulating MSC specifications had advantages for system designers. We have considered the simulation of MSC specifications in PROMELA,

[7] XSPIN automatically generates a 'never'-claim from an LTL and allows for adding the claim to the specification (see [**LL96**] for the code of the 'never'-claim of the LTL formula number **3** in Table 1)

```
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) -> skip
    fi; goto C2};
  C2:
  do
  :: atomic {
       vw?[CC] -> vw?CC;
       goto END}
  :: atomic {
       xy?[DR] -> xy?DR; goto C1}
  od;
  END: skip
}
proctype P2()
{ C1:
  atomic {
     tu?[CR] -> tu?CR; goto C2};
  C2:
  do
  :: atomic {
       if
       :: vw!CC
       :: full(vw) -> skip;
       fi; goto END }
  :: atomic {
       if
       :: xy!DR
       :: full(vw) -> skip
       fi; goto C1 }
  od;
 END: skip
}

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

FIGURE 20. PROMELA code for example 7

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

#define aftsCR P1[1]@aftersCR
#define aftsCC P2[2]@aftersCC
#define aftsDR P2[2]@aftersDR

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

proctype P1()
{ C1:
  atomic {
    if
    :: tu!CR
    :: full(tu) -> skip
    fi; }
    aftersCR: goto C2;
  C2:
  do
  :: atomic {
       vw?[CC] -> vw?CC;
       goto END}
  :: atomic {
       xy?[DR] -> xy?DR;
       goto C1}
  od;
  END: skip}
proctype P2()
{ C1:
  atomic {
     tu?[CR] -> tu?CR;
     goto C2};
  C2:
  do
  :: atomic {
       if
       :: vw!CC
       :: full(vw) -> skip
       fi; }
       aftersCC: goto END
  :: atomic {
       if
       :: xy!DR
       :: full(vw) -> skip
       fi; }
       aftersDR: goto C1
  od;
 END: skip}
init { atomic { run P1(); run P2() } }
```

FIGURE 21. PROMELA code including state labels.

and noted how the questions on the MSC semantics considered in [**LL95b, LL95a**] are reflected directly in the PROMELA executions. We considered in particular verifying properties expressed in LTL. Some liveness properties of MSCs are underdetermined by the standard but any simulation must either allow or avoid each questionable execution sequence, therefore these decisions had to be made. Questions about non-local choice in MSC branching, which originally arose with conditions but is also present for HMSCs were also considered, and implemented following the development in [**LL95a**]. To avoid imposing over-stringent liveness conditions (that the executing processes may only lag each other by a bounded

Property	Outcome
1. `[] <> aftsCC`	not satisfied
2. `<> aftsCC`	not satisfied
3. `[] (aftsCC -> ! <> aftsDR)`	satisfied
4. `[] (aftsCR -> <> aftsCC)`	not satisfied
5. `aftsCR -> <> (aftsCC \/ aftsDR)`	satisfied
6. `([]<>aftsDR) -> !<> aftsCC`	satisfied

TABLE 1. Temporal Properties verified using XSPIN

amount), we considered introducing message-instance-channels and discussed their implementation in PROMELA and some consequences.

Current work includes a formalisation of the MSC-to-PROMELA translation (see [**LL95c**] for a preliminary version) and the development of a tool supporting this translation. Furthermore, we investigate the syntactic analysis of MSC specifications. We have defined syntactic conditions for the occurrence of non-local choices, process divergence and deadlocks in MSC specifications [**BAL96b**]. Combined with the analysis of MSC specifications as discussed in this paper the resulting tool can provide software engineers with substantial support in providing unambiguous first design specifications [**BAL96a**].

References

[AB] Telelogic AB, *SDT*, Participant's Proceedings of the 8th International Conference on Formal Description Techniques FORTE'95, List of tools for demonstrations (G. von Bochmann, R. Dssouli, and O. Rafiq, eds.), p. 455.

[AHP96] R. Alur, G. J. Holzmann, and D. Peled, *An analyzer for message sequence charts*, Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055 (T. Margaria and B. Steffen, eds.), Springer Verlag, 1996, pp. 35–48.

[BAL96a] H. Ben-Abdallah and S. Leue, *Architecture of a requirements and design tool based on message sequence charts*, Tech. Report 96-13, Department of Electrical & Computer Engineering, University of Waterloo, October 1996, 18 p., submitted for publication.

[BAL96b] H. Ben-Abdallah and S. Leue, *Syntactic analysis of message sequence chart specifications*, Tech. Report 96-12, Department of Electrical & Computer Engineering, University of Waterloo, October 1996, 32 p., submitted for publication.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture - a system of patterns*, Wiley and Sons Ltd., 1996.

[BZ83] D. Brand and P. Zafiropulo, *On communicating finite-state machines*, Journal of the ACM **30** (1983), no. 2, 323–342.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, Protocol Specification, Testing and Verification XV (P. Dembiński and M Ŝredniawa, eds.), Chapmann & Hall, 1995, pp. 1–18.

[Hol] G. J. Holzmann, *What's new in SPIN version 2.0*, http://netlib.att.com/netlib/spin/index.html, Version April 17, 1996.

[Hol91] G. J. Holzmann, *Design and validation of computer protocols*, Prentice-Hall International, 1991.

[Hol95] G. J. Holzmann, *The verification of concurrent systems*, AT&T Bell Laboratories, to be published by Prentice-Hall, 1995.

[Hol96] G. J. Holzmann, *Early fault detection tools*, Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055 (T. Margaria and B. Steffen, eds.), Springer Verlag, 1996, pp. 1–13.

[IT95] ITU-T, *Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts*, ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1995.

[IT96] ITU-T, *Recommendation Z.120*, ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996, Review Draft Version.

[Jea92] I. Jacobson and et al., *Object-oriented software engineering - a use-case driven approach*, Addison-Wesley, 1992.

[Lab95] NTT Software Laboratories, *SDE technical tour*, Dec. 1995, Presentation slides.

[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*, , 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), no. 5, 473–509.

[LL95c] S. Leue and P. B. Ladkin, *Implementing message sequence charts in PROMELA*, Proceedings of the First SPIN Workshop (J.-Ch. Grégoire, ed.), INRS Télécommunications, Montréal, Canada, 1995.

[LL96] S. Leue and P. B. Ladkin, *Implementing and verifying scenario-based specifications using Promela/XSpin*, Participants Proceedings of the 2nd International Workshop on the SPIN Verification System, DIMACS/Bell Labs/INRS-Télécommunications, 1996, pp. 129–146.

[LS] P. B. Ladkin and B. B. Simons, *Static analysis of communicating processes*, To appear, Springer Lecture Notes in Computer Science.

[McM93] K. L. McMillan, *Symbolic model checking: An approach to the state-explosion problem*, Kluwer Academic Publishers, 1993.

[NH96] V. Natarajan and G. J. Holzmann, *Outline for an operational-semantics definition of Promela*, Participants Proceedings of the 2nd International Workshop on the SPIN Verification System, DIMACS/Bell Labs/INRS-Télécommunications, 1996, pp. 175–191.

[OFMP+94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith, *Systems engineering using SDL-92*, Elsevier Science B.V. (North-Holland), 1994.

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

[Sel96] B. Selic, *Automatic generation of test drivers from MSC specs*, Tech. Report TR 960514 - Rev. 01, ObjecTime Limited, Kanata, Ontario, Canada, 1996.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward, *Real-time object-oriented modelling*, John Wiley & Sons, Inc., 1994.

Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada
E-mail address: sleue@swen.uwaterloo.ca

Technische Fakultät, Universität Bielefeld, D-33501 Bielefeld, Germany
E-mail address: ladkin@techfak.uni-bielefeld.de