Implementing and Verifying Scenario-Based Specifications Using Promela/XSpin
– 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

In previous work we defined a finite state semantics for Message Sequence Charts (MSCs) and suggested a translation of MSC specifications into Promela. We call this translation an '*implementation*'. In this paper we reconsider the implementation of MSCs and discuss what information needs to be added when implementing MSC specifications containing so-called *non-local choices*. Next, 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 have found their way into many software engineering methodologies and toolsets. These methods and tools include SDL tools and environments [21, 3, 12, 15], Object-oriented methodologies [22, 14, 23, 5], tools to analyse the design of message exchanges at early stages in the software lifecycle [11, 2], and methods describing design patterns [6].

In previous work we have defined a finite state semantics for Message Sequence Charts [17], and discussed in [16] implications of the MSC notation as defined in ITU-T recommendation Z.120 [13]. Furthermore, in [19] we discussed how to translate MSCs into Promela [9] code in conformance with our semantics. We call this translation an *implementation*.

Uses of MSCs. There are three major ways in which MSCs are used:

1. to visualise actual system execution, during debugging and program understanding (as in [11, 2]),

2. as a language to document early design decisions (as in ObjecTime [23], EFD [11]),

3. to document test cases or functional validation criteria that an implementation must satisfy (ROOM[23], SDT [1]).

Items 1 and 3 concern finite execution scenarios in which one event node corresponds to a single event and a single message. The standard calls such finite MSCs *Basic MSCs* (BMSCs). For examples of BMSCs consider the three MSCs on the left hand side of Figure 1. BMSCs are non-branching and non-iterating. In [17] we described how to represent BMSCs algebraically as so-called basic *Message Flow Graphs* (MFGs). Here we are not primarily interested in BMSCs since their meaning is straightforward and implementation is trivial. We interest ourselves here in 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 designers use MSCs 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 the example in Figure 1 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 establishement 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'll use the following definition of HMSC. An HMSC 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 can thus 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

[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.

[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.

1) 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 note `MSC1`).

HMSC graphs hold out the hope for a clearer notion of MSC composition than possible with *conditions*. The MSC specifications with which we are concerned will 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. We note that many of the problems in interpretation noted in [17, 16] 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. However, they become apparent when pursuing Promela simulations of MSC specification.

Motivation. We interested in implementing (i.e., simulating) MSCs because:

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 is an exercise in translating MSCs into process code which 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 and asynchronous composition of concurrent processes; and communication channels) that were necessary to implement MSC specifications. Furthermore, the XSpin [9, 8] tool allows for randomly simulating Promela specifications, which helps in debugging, and for model-checking of 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 like for example *SMV* [20]. 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 Implementation of MSCs.

We briefly reiterate some implementation decisions discussed in [19]. First, the graphical object MSC specification is translated into a corresponding *Message Flow Graph* [17]. See Figure 2 for

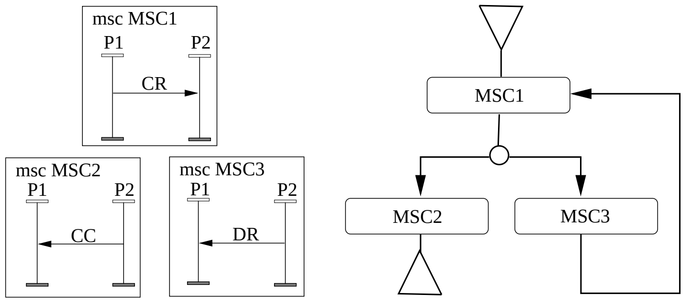

Figure 1: MSC Specification 1.

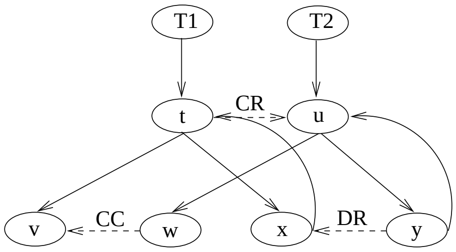

Figure 2: Message Flow Graph 1.

the MFG corresponding to the MSC specification in Figure 1. Figure 9 shows a possible Promela implementation as suggested in [19][3]. Further implementation choices included:

- 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 9.

- 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 was 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 was modeled using Promela labels and `goto` statements.

- It was 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 used `full(..) -> skip` statements to model the non-blocking send of an MSC.

- Receive statements, however, are blocking. This was implemented using a Promela guard-statement pair, see for an example the `vw?[CC] -> vw?CC` statement pair in Figure 9. It was particularly important to guarantee atomicity of these guard-statement pairs.

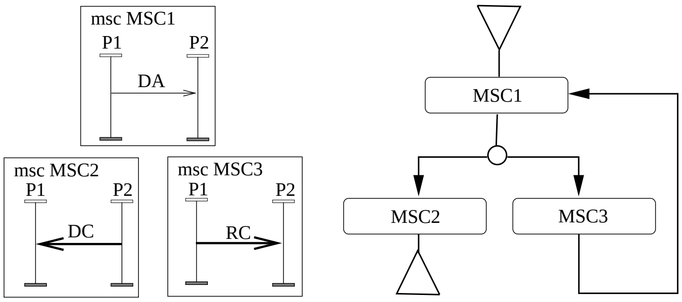

Figure 3: MSC Specification 2, with non-local choice

3 Implementing Non-Local Choice

The MSC in Figure 3 is similar to the example in Figures 1 and 2. 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 (Figures 6 and 7) 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 conditions `C1` and `C2`, 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 `C2` 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 in `C2` was called a 'non-local choice' in [17, 16] 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 1. [19] discussed two somewhat unsatisfactory variants of the implementation of the example in Figure 3.

Executing Non-local Choice with history variables. As described in [16], 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

[3]Note that we suggested two alternative implementation strategies of branching MSC specifications: the *wait-and-see* strategy and the *history variable* based coordination of both processes. Here we have chosen the wait-and-see strategy, for more discussion see Section 3.

described in [16].

We use $N+1$ global history variables: variables `i1` ... `i`N and variable `hist`. In our example, $N=2$. Process `P`k keeps track of the iteration it is on by setting history variable `i`k. However, the choice history only records M previous choices (M is thus the bound on the size of the choice-history variable), so `i`k is approximated by `n`k = `i`k mod 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, `P`j checks whether `hist[`k`]` has been previously set by checking whether some other `n`k is greater than `n`j, which must be done in any case, as we see next.) `P`k is the only process setting `i`k and `n`k and may only set `hist[`m`]` if `n_`$m \geq$ `n_`j for $k \neq j$ when `i`k= k, otherwise it must follow the decision indicated by the value of `hist[`k`]`.

Figures 6 and 7 show 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 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 3 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

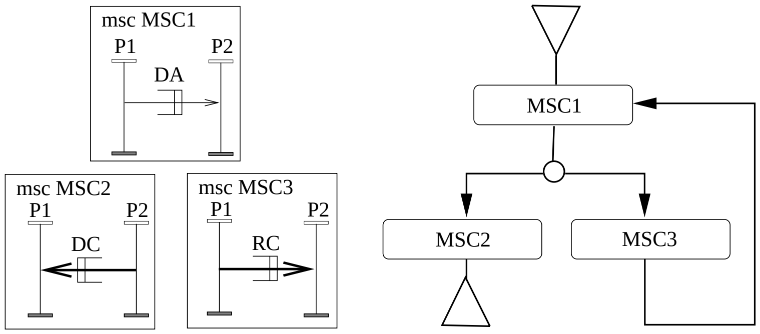

Figure 4: MSC example with non-local choice and explicitly represented channels

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.)

This scenario leading to deadlock may be avoided. We will describe how in the next section. However, any such suggestion must necessarily involve an extension to the expressive capabilities of the MSC language as it now is.

4 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. [17] 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, [16] 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 [17, 16] 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 4 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 of infinite) capacity.
4. The semantics of the MSC send primitive should be changed from non-blocking to blocking. MSCs as they now are 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 syntactic definitions in [17] 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* [4]. Furthermore, a formal operational semantics for Promela is currently under development[4], 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[5]. However, we conjecture that blocking due to non-local choices can be avoided by this mechanism. A more-detailed study of how one might introduce channels, and what properties those channels should have, is however not our main topic in this paper. 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.

5 Finite State Implementation

Two of the constructs we have suggested lead to an infinite-state model:

[4]According to G.J. Holzmann, personal communication.

[5]For necessary and sufficient criteria for MSCs to be deadlock-free see [18].

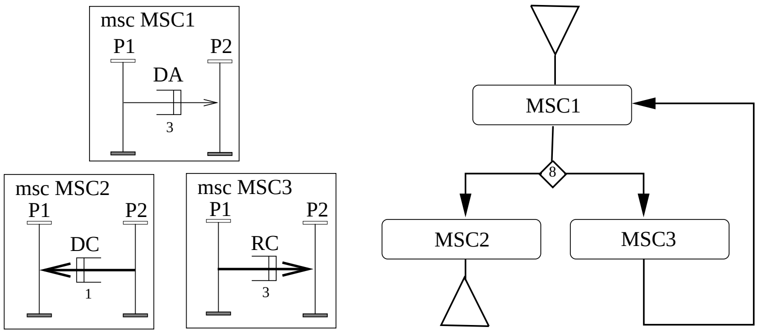

Figure 5: MSC example with non-local choice, explicitly represented channels, and explicit capacities for the non-local choice history variable (=8) and channels (=3, 3 and 1)

- non-local-choice *history* variable(s), and
- *channels* associated with message arrows.

However, finite-state-space validation techniques as well as an implementation using 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 5. Figures 6 and 7 show 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 5 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 8 shows an execution trace generated by Xspin based on the implementation of the MSC specification in Figure 5 as given in Figures 6 and 7. 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 $(\texttt{!DA, !RC})^4$ could be part of an admissible execution sequence, this is not the case for the example in Figure 5.

```
/* Length of history variable */
#define M 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[M];  /* 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) <= M ->   /* 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) < M ->   /* 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 % M;
       vw?[DC] -> vw?DC;
       goto END;
    C21:
       n1 = n1 + 1; i1 = n1 % M;
       xy!RC;
       goto C1}
  END: skip}
```

Figure 6: Implementing non-local choice using a history variable with bounded length and channels with bounded capacity, Part 1

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

init { n1 = 0; n2 = 0; i1 = n1 % M; i2 = n2 % M;
       atomic { run P1(); run P2() } }
```

Figure 7: Implementing non-local choice using a history variable with bounded length and channels with bounded capacity, Part 2

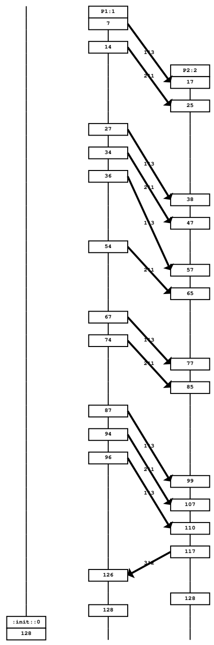

Figure 8: Trace of MSC specification generated by random simulation using XSpin

3. As argued eralier, the size of the history variable limits the divergence of processes `P1` and `P2`. In the example in Figures 6 and 7 `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. Obviously, both channel capacity as well as history variable length have strong bearing on the potential divergence of the processes.

6 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 [16, 17] MSCs underspecify liveness properties, and temporal logic can be used to specify the additional liveness properties. Desired liveness properties specifiable in LTL may be checked in Promela, thus Promela can support this extension of the MSC language.

It is possible using LTL (linear-time temporal logic) to specify properties on the Promela model generated from an MSC specification that this model cannot guarantee. For example, consider MSC 1 in Figure 1. 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 by termination, along which path the system cannot ever again reach a state in which $ta\_\mathtt{DR}$ holds.

The XSpin environment allows model checking of LTL formulas based on Promela models [7, 8]. LTL formulas can be entered in XSpin, and a preprocessor translates them into so-called *never* claims [9, 10]. 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 [19], 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 10 shows the Promela code implementing the MSC in Figure 1. Labels `aftersCR`, `aftersCC` and `aftersDR` denote points in the process control flow as needed to define $ta\_$[6].

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 12). 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.

7 Summary and Outlook

We noted that simulating MSC specifications had advantages for system designers. We have considered the simulation of MSC specifications in Promela, and noted how the questions on the MSC semantics considered in [17, 16] 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

[6] The code in Figure 11 is a 'never'-claim automatically generated by XSpin from 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 9: Promela code for MSC example with branching control

```
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 10: Promela Code including state labels.

```
/*
 * Formula As Typed: [] (aftsCC -> ! <> aftsDR)
 * The Never Claim Below Corresponds
 * To The Negated Formula !([] (aftsCC -> ! <> aftsDR))
 */
/* Normlzd: (true U ((true U (aftsDR)) && (aftsCC))) */
never {
T0_init:
if
:: (1) -> goto T0_init
:: ((aftsCC)) -> goto T0_S4
:: ((aftsCC) && (aftsDR)) -> goto accept_all
fi;
T0_S4:
if
:: (1) -> goto T0_S4
:: ((aftsDR)) -> goto accept_all
fi;
accept_all:
skip}
```

Figure 11: Promela temporal claim generated automatically from LTL formula.

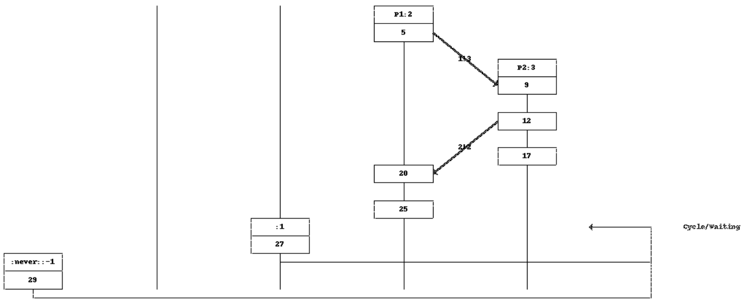

Figure 12: MSC showing trace leading to a state violating `[] <> aftsDR`.

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

branching, which originally arose with conditions but is also present for HMSCs were also considered, and implemented following the development in [16]. To avoid imposing over-stringent liveness conditions (that the executing processes may only lag each other by a bounded 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 and the development of a tool supporting this translation. Furthermore, we investigate the syntactic analysis of MSC specifications. We are interested in specifying syntactic conditions for the occurrence of non-local choices in MSC specifications, as well as conditions for certain liveness problems (not) to arise. 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.

Acknowledgements

The first author wishes to acknowledge the support of the Information Technology Research Centre (ITRC), the National Science and Engineering Research Council of Canada (NSERC), and ObjecTime Limited.

References

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

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

[3] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.

[4] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, Apr 1983.

[5] R.J.A. Buhr and C.S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons Ltd., 1996.

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

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

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

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

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

[12] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki. SDE: Incremental specification and development of communications software. *IEEE Transactions on Computers*, 40(4):553–561, Apr. 1991.

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

[14] I. Jacobson and et al. *Object-Oriented Software Engineering - A Use-case Driven Approach.* Addison-Wesley, 1992.

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

[16] 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.

[17] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

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

[19] S. Leue and P. B. Ladkin. Implementing message sequence charts in PROMELA. In Jean-Charles Grégoire, editor, *Proceedings of the First SPIN Workshop.* INRS Télécommunications, Montréal, Canada, 1995.

[20] K. L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem.* Kluwer Academic Publishers, 1993.

[21] 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.

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

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