SDL and MSC Based Test Generation for Distributed Test Architectures

Jens Grabowski, Beat Koch, Michael Schmitt, and Dieter Hogrefe

Institute for Telematics, University of Lübeck,
Ratzeburger Allee 160, D-23538 Lübeck, Germany,
eMail: {jens,bkoch,schmitt,hogrefe}@itm.mu-luebeck.de

Abstract

Most of the SDL and MSC based test generation methods and tools produce non-concurrent TTCN test cases only. If the test equipment itself is a distributed system, the implementation of such test cases is a difficult task and requires a substantial amount of additional work. In this paper, we explain how concurrent TTCN test cases can be generated directly from SDL system specifications and MSC test purposes. To do this, explicit synchronization points have to be indicated in the MSC test purposes, and information about the existing test components and their connections has to be provided.

Keywords: SDL, MSC, TTCN, distributed systems, distributed test systems, conformance testing, test generation.

1 Introduction

Supporting simulation, validation, code generation and test generation are the most important reasons for using SDL [4,15] as system specification language and MSC [12,16] as requirement specification language within the software life-cycle for distributed systems. Validation, simulation and code generation tools for SDL and MSC descriptions are commercially available and heavily used in industry [1,14,11].

Contrarily, the use of test generation tools is not very popular yet. There are a lot of reasons for this: Fully automatic test generation methods mainly fail due to complexity. Even for toy examples, they calculate an amount of test cases which cannot be handled in practice. Pragmatic problems are related to the fact that specification based test generation is not applicable equally in all stages of the software life-cycle: In most cases, no detailed SDL specification is developed for small and medium sized software modules and therefore, SDL based test generation methods cannot be used for software module testing and software integration testing.

However, tools which follow a pragmatic approach, such as Autolink, SaMsTaG, TGV, TTCgeN or TVEDA, are used increasingly in industry, research and standardization [3,2,6,7,10,13]. Most of them provide generation of test cases guided by test purposes. SDL is used for system specification; MSC or observer processes are used for test purpose description and the *Tree and Tabular Combined Notation* (TTCN) [9] is used for the representation of the generated test cases.

The current tools compute test cases with one global tester process controlling and observing the entire system under test. Thus, the test cases are described in the non-concurrent form of TTCN. The use of such test cases is problematic if the SUT is a distributed system with components at different locations. In that case, the test equipment itself forms a distributed system, and the test case can be seen as a program running on the distributed test system. It is obvious that the implementation of a non-concurrent TTCN test case on distributed test equipment is a complicated and error-prone task. The use of concurrent TTCN, which directly supports the distribution of test cases among test components, would be more appropriate.

The generation of concurrent TTCN instead of non-concurrent TTCN is not trivial, because additional information is needed which cannot be deduced from an SDL specification. In this paper, we present our ideas of generating concurrent TTCN test cases based on SDL system specifications and MSC test purposes. Throughout the paper, we describe our approach by using the terminology defined in the *Conformance Testing Methodology and Framework* (CTMF) [8].

The paper is structured in the following manner: In Section 2, we give an introduction to the concepts of concurrent TTCN. Section 3 presents methods for the definition of test component configurations and the synchronization of test events. In Section 4, an algorithm for the generation of concurrent test cases is described. Finally, Section 5 contains a summary and an outlook on our future plans.

2 Concurrent TTCN

In 1996, TTCN [9] has been extended with mechanisms for specifying test suites for distributed test systems. In this section, we introduce the most important concepts of concurrent TTCN.

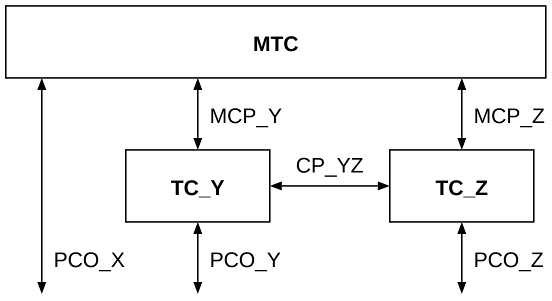

Figure 1. Example test component configuration

2.1 Test component configuration

In TTCN, a test system is structured into a number of *Test Components* (TCs). In the non-concurrent case, only one *Main Test Component* (MTC) exists which is defined implicitly. In the concurrent case, one MTC and one or more *Parallel Test Components* (PTCs) exist. The MTC is responsible for the creation of the PTCs and the computation of the final test verdict.

The TCs may or may not communicate with the *System Under Test* (SUT) via *Points of Control and Observation* (PCOs). A PCO is exclusively assigned to one TC. Communication at PCOs is bidirectional and asynchronous, i.e., the model for a PCO is a combination of two infinite FIFO buffers. The messages exchanged at PCOs are either *Protocol Data Units* (PDUs) or *Abstract Service Primitives* (ASPs). In the following, we will just use the term ASP, but this could be exchanged with PDU at all times.

Communication between two TCs can be performed by an asynchronous exchange of *Coordination Messages* (CMs) at *Coordination Points* (CPs). Similar to a PCO, a CP can also be understood as a combination of two infinite FIFO buffers.

Even if there is no CP specified between the MTC and a PTC, two types of implicit communication are defined: (1) PTCs assign their test verdict to the global result variable which is used by the MTC to compute the final test verdict, and (2) the MTC can check if a PTC has terminated.

The connections between the MTC, PTCs and the SUT are described with a *Test Component Configuration* (TCC). A test suite may contain several TCCs. For each concurrent test case, one TCC has to be chosen. Figure 1 shows a TCC: The MTC is connected with two PTCs through coordination points `MCP_Y` and `MCP_Z`. In addition, the MTC controls one PCO (`PCO_X`). The PTC `TC_Y` communicates with the SUT through `PCO_Y`. `TC_Z` controls and observes the SUT through `PCO_Z`. Finally, the two parallel test components may exchange CMs through coordination point `CP_YZ`.

2.2 Behavior description in concurrent TTCN

The behavior description in concurrent and non-concurrent TTCN is basically identical. The behavior of an MTC is described by using a *Test Case Dynamic Behaviour* table. PTCs are specified as test steps. The behavior of these test steps may be defined as *local trees* within the *Test Case Dynamic Behaviour* table or by using separate *Test Step Dynamic Behaviour* tables.

An example of a concurrent TTCN test case is shown in Figure 10. The behavior of the MTC is specified by the main behavior tree. The PTC descriptions are included as local trees. The MTC creates all PTCs by calling the corresponding behavior descriptions with the `CREATE` construct (line 1). The termination of the PTCs is checked by means of `DONE` events (lines 7 and 11).

Communication among TCs is treated the same way as communication with the SUT. In Figure 10, statement `MCP_2!Proceed(Sync2,Y)` on line 6 denotes the sending of CM `Proceed(Sync2,Y)` via CP `MCP_2` to PTC `PTC_2`. The corresponding receive event of the PTC can be found on line 24.

2.3 Synchronization of test components

The synchronization of TCs may be done implicitly or explicitly. Implicit synchronization is performed at the start and the termination of the test case execution: The MTC creates all PTCs and is able to check their termination. Explicit synchronization can be performed by exchanging CMs between TCs.

The exchange of CMs can only be used to coordinate the actions of TCs controlling different PCOs. It cannot be used to ensure the correct order of test events at different PCOs. This is due to the asynchronous communication mechanism via infinite FIFO buffers and can be explained by means of a simple example:

We have two TCs, `TC1` and `TC2`, controlling different PCOs. `TC1` sends an ASP `M1` to the SUT and as a reaction, an ASP `M2` is sent by the SUT to `TC2`. We can try to ensure the correct order of sending of `M1` by `TC1` and the reception of `M2` by `TC2` by using one of the following two strategies:

(1) A coordination message `CM1` is sent by `TC1` to `TC2` as an indication that `M2` is the next message to be received from the SUT.
(2) `TC2` knows that `M2` is the reaction to `M1` sent by `TC1` and therefore sends a coordination message `CM2` to `TC1` as a request to send `M1`.

In the first case, `M2` may overtake `CM1` and `TC2` will interpret this as a failure although the actual order was correct. In the second case, neither `TC1` nor `TC2` will detect if the SUT sends `M2` during the transmission of `CM2` (i.e. before `M1` has been sent), and an incorrect order will pass the test. Thus, neither of the strategies can be used to ensure the correct order of sending `M1` and receiving `M2`. Assuming additional knowledge about the transmission time of messages does not help either: According to CTMF, "the relative speed of systems executing the test case should not have an impact on the test result".

3 Test purpose specification for distributed test architectures

Test purpose description with *system level MSCs* has proven to be an effective and intuitive method for tool-supported TTCN test generation from SDL specifications [5]. In a system level MSC, there is one instance for the SUT[1] and one instance for every PCO, with all channels to the environment of the SDL system being considered to be PCOs. For the generation of test cases in the concurrent TTCN format, additional information has to be provided. This information concerns the TCC and the synchronization of TCs.

3.1 Defining test component configurations

An SDL specification defines the functionality of a system by describing its dynamic behavior. The concrete and finally implemented system architecture, including the distribution of the different components, is not described. The use of structural SDL concepts, e.g., blocks, processes, services, or procedures, may give some hints about the system architecture, but they are only means for structuring the specification. Whether two blocks or processes execute on the same or on several computers at different locations is not described. Therefore, it is not possible to determine an appropriate TCC from an analysis of the SDL specification for which we intend to generate test cases. Additional information has to be provided which identifies

(1) the TCs and their roles (either MTC or PTC),
(2) the assignment of PCOs to TCs (connections between TCs and the SUT),
(3) the CPs, and
(4) the assignment of CPs to TCs (connections among TCs).

This information may be provided in a graphical form, e.g., an SDL system or block diagram, in form of the corresponding TTCN tables, or in form of a tool specific command language. A more sophisticated tool may also be able to calculate a default configuration automatically, e.g., a TCC where each PTC handles one PCO only and the MTC is responsible for PTC creation, synchronization, and calculation of the final test verdict.

In order to implement the automatic generation of concurrent TTCN test cases from SDL system specifications and MSC test purposes, we have to restrict ourselves to certain types of TCCs. Therefore, we require that each PTC handles at least one PCO and that it is connected to the MTC with a

[1] The restriction that the entire SUT is represented only by one instance can be weakened without problems. Due to our experience [13], we keep it here.

CP. The MTC primarily controls and synchronizes the PTCs. It does not have to handle PCOs on its own, but we require that it is able to exchange CMs with all PTCs.

3.2 Synchronization of test components and test purpose descriptions

For test specification, we distinguish two types of synchronization: Implicit and explicit synchronization.

3.2.1 Implicit synchronization

Implicit synchronization is done at the start and may be done at the end of a test case by the MTC. The corresponding `CREATE` constructs and `DONE` events can be generated automatically by a tool.

Further synchronization is needed if it has to be guaranteed that the first send event happens after the creation of all PTCs or if the PTCs should indicate their termination to the MTC. For these cases, one of the explicit synchronization mechanisms has to be used.

3.2.2 Explicit synchronization

During the execution of a concurrent TTCN test case, the TCs are not aware of the state of the other TCs. If the execution of a test event `PCO_X!DataReq` by a TC `TC_A` should happen after the execution of test event `PCO_Y!ReleaseReq` by TC `TC_B`, then `TC_A` and `TC_B` have to exchange CMs to accomplish the execution of `PCO_X!DataReq` and `PCO_Y!ReleaseReq` in the correct order.

The points in the control flow of the TCs at which such a synchronization should take place can not be calculated automatically, because they may depend on the intention of the test designer. For example, for the situation described above, the test designer intends to test the appropriate treatment of a `DataReq` during the release of a connection. Without additional information, a test generation tool will not care whether the `ReleaseReq` is sent before `DataReq` or not. As a consequence, for such cases the synchronization of TCs has to be specified explicitly by the test designer within the MSC test purpose.

Depending on the number of TCs involved and test events to be coordinated, the CM exchange which is necessary for an explicit synchronization may become very complex. To cope with simple as well as complex situations, we provide two means for the specification of explicit synchronization. First, it can be done by describing the CM exchange directly and second, by using MSC conditions for the definition of synchronization points. For the latter case, the exchange of CMs among the TCs is generated automatically.

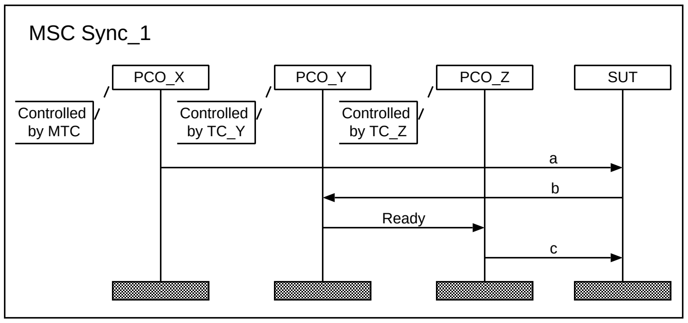

Figure 2. Explicit synchronization by means of a coordination message

Explicit synchronization with coordination messages. Defining explicit synchronization by describing the exchange of CMs within the MSC test purpose is not as easy as it seems to be at first glance. The reason is that in the MSCs we use for test purpose description, the instances represent PCOs and not the TCs controlling the PCOs.[2] However, CMs are exchanged between TCs and therefore the actual sender and receiver processes of the CMs are not represented in the MSC test purposes.

Nevertheless, we allow to specify CMs between PCO instances and interpret them as follows: A CM `CM1` with an origin at PCO instance `PCO_A` and a target at PCO instance `PCO_B` coordinates test events at `PCO_A` and `PCO_B`. The origin refers to the send event of `CM1` by the TC controlling `PCO_A`. The target refers to the corresponding receive event by the TC controlling `PCO_B`. The order of events (including send and receive events of CMs) along a PCO instance has to be preserved by the TC controlling the PCO. It should be noted that no graphical distinction between CMs and ASPs has to be made in the MSC test purposes. Origin and target of a CM arrow have to be PCO instances. For ASPs, either the origin or target has to be the SUT instance.

Figure 2 shows an MSC test purpose description. The corresponding TCC is the one presented in Figure 1. The MSC includes the CM `Ready` drawn by the test designer. It has to be interpreted as follows: `TC_Y` should send the CM `Ready` after the reception of ASP `b` at `PCO_Y`, and `TC_Z` should send ASP `c` after the reception of CM `Ready`. In this case, the CM `Ready` forces the TCs to perform exactly the order `PCO_X!a` $\rightarrow$ `PCO_Y?b` $\rightarrow$ `PCO_Z!c` of test events during their communication with the SUT.

However, the specification of CMs by the test designer becomes difficult if more than two TCs are involved, and if receive events of CMs are alternatives to the reception of ASPs.

Figure 3 provides a test purpose example for the same TCC as in the previous example (Figure 1). The test run should execute as follows: First, `TC_X` sends ASP `a` via `PCO_X` to the SUT, which in turn answers with ASPs `b`, `c` and `d` to be observed at `PCO_Y` and `PCO_Z`, respectively. ASP `e` should be sent via `PCO_Z`

[2] Note that a TC may control and observe more than one PCO.

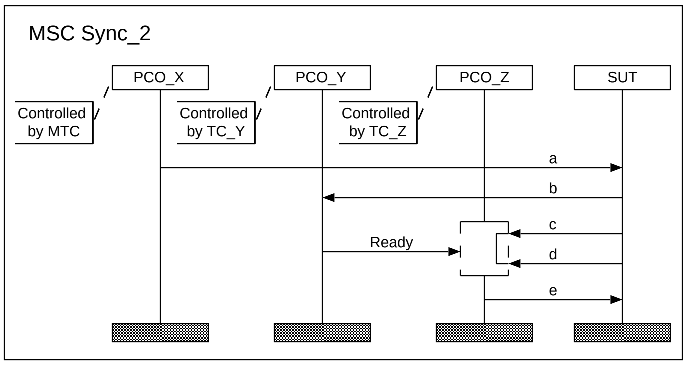

Figure 3. Complex explicit synchronization by means of coordination messages

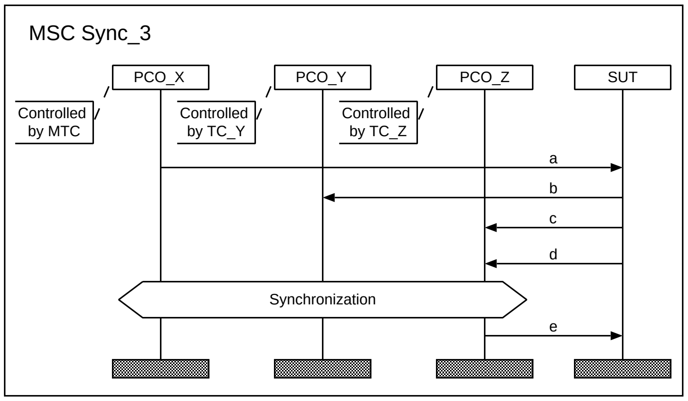

Figure 4. Explicit synchronization by means of a condition

after the reception of `b`, `c` and `d`. To ensure this, the reception of `b` has to be confirmed by means of CM `Ready` sent by `TC_Y`.

ASPs `c` and `d` are received via the same FIFO queue `PCO_Z`, and their order is given by the test purpose. The CM `Ready` is received by `TC_Z` via a second FIFO queue, i.e., the CP between `TC_Y` and `TC_Z`. It cannot be predicted if the CM `Ready` is received before, after or between the reception of `c` and `d`. Thus, a coregion with generalized ordering has to be used to specify all possible orders of reception.

It is obvious that the manual drawing of CMs becomes increasingly complicated if even more TCs, CMs and CPs are involved. In order to ease the test specification, we present another possibility to describe explicit synchronization in the next paragraph.

Explicit synchronization with conditions. In order to get a simple but robust and consistent mechanism to specify synchronization, we use MSC conditions. The conditions used for test synchronization purposes should only cover PCO instances. We call them *synchronization conditions*. They define common *Synchronization Points* (SPs) within the message flow at the different PCOs.

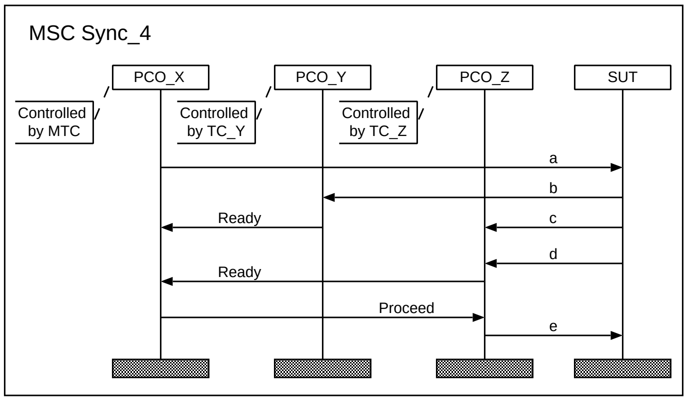

Figure 5. Automatically generated CM exchange for the condition in Figure 4

Figure 4 shows an MSC test purpose description with a synchronization condition. The desired effect of the synchronization is the same as in Figure 3: `TC_Z` should send ASP `e` not before the ASPs `b`, `c` and `d` have been received.

During test case generation, the synchronization conditions are used to calculate the actual exchange of CMs between TCs. There are several possibilities to perform a synchronization by means of message exchange, but a discussion about the advantages and disadvantages of the different possibilities would go beyond the scope of this paper.

For an implementation, we have decided to support one mechanism only which allows to synchronize the ASP exchange of an arbitrary number of PCOs. The synchronization is managed by the MTC, and the principle of the mechanism is simple: As already stated above, synchronization conditions define common SPs within the ASP exchange at different PCOs. A PCO can be seen as a sequential process, and the TC handling the PCO as the process manager. If a PCO reaches an SP, this is reported to the MTC, and the PCO goes into a waiting state. If the test event following the SP is the receiption of an ASP, the PCO waits for the ASP. If it is a send event, the PCO has to wait for a CM from the MTC to get the permission to send the next ASP to the SUT. This mechanism assures that all PCOs involved in the synchronization reach the SP before the execution of the test case continues.

By applying this mechanism, the CM exchange shown in Figure 5 is generated automatically for the condition in Figure 4. The `Ready` CMs are used to indicate that the SP has been reached, and the CM `Proceed` is used to allow the sending of ASP `e`.

4 Test case generation procedure

In this section, we will show how a test case specified in concurrent TTCN can be derived from an SDL specification and an MSC test purpose. Two algorithms are presented for the computation of the behavior trees of an MTC and

PTCs. Explicit exchange of CMs is not considered; however, we will explain the way synchronization conditions can be treated.

4.1 Interface to an SDL/MSC simulator

The test generation algorithm is based on the basic functionality provided by a general purpose state space exploration tool which allows the combined simulation of an SDL specification and an MSC test purpose [7,11,14]. In particular, we require that the following two functions are available:

next_events($state_{sim}$**)** – Given a state $state_{sim}$, `next_events` returns a set of all events which may occur next. State $state_{sim}$ describes both the current global state of the simulated SDL system (i.e., the state, timers and variable values of each individual process, queue contents etc.) and the progress in the MSC (i.e., the events which are to be simulated next at each instance). If `next_events` returns the empty set, we assume that the MSC has been verified completely, i.e., a path through the reachability graph of the SDL system has been found which satisfies the MSC.[3]

next_state($state_{sim}$**,**e**)** – Given a state $state_{sim}$ and an event e, `next_state` returns the state which is obtained if e is executed in $state_{sim}$.

There are several different types of events that might happen during simulation. Some of them may only refer to the SDL system (e. g., internal events that are not represented in the MSC), while others may only refer to the MSC (e. g., messages related to synchronization conditions). During test generation, we consider four types of events which are listed below. All other events which might be reported by the simulator engine are skipped during test generation. That means, they are executed in order to get to the next system state, but they are not transformed into TTCN events.

Event 'Send from SDL environment' (pco**!**sig**)** – ASPs sent from the environment into the SDL system during simulation become TTCN send events. In order to be able to specify a send event, the simulator is required to return both the complete signal and the channel (PCO) through which the ASP was sent.

Event 'Send to SDL environment' (pco**?**sig**)** – ASPs sent by the SDL system to its environment become receive events in the TTCN test case. In analogy to send events, the simulator has to report both the PCO and ASP.

Event 'Enter synchronization' (`enter_sync(`id`,`pco`,`Cur`,`All`)`**)** – Whenever an instance in the MSC test purpose may reach a synchronization condition, a special event called `enter_sync` has to be returned by function

[3] For simplicity, we neglect the possibility that a deadlock occurs or the simulation is stopped, because the behavior of the SDL system does not comply to the MSC.

```
testgen
{
  for $i$ = 1 ... $n$ do
  {
    $root_{ptc_i}$ := new_start_node();
    testgen_ptc($i$,$root_{sim}$,$root_{ptc_i}$)
  }
  $root_{mtc}$ := new_start_node();
  testgen_mtc($root_{sim}$,add_trans($root_{mtc}$,
              CREATE($PTC_1$:Test$PTC_1$,...,$PTC_n$:Test$PTC_n$)))
}
```

Figure 6. Invocation of the test generation for the PTCs and the MTC

`next_events`. `enter_sync` has four parameters: *id* denotes the unique identifier of the condition; *pco* is the name of the PCO instance which reached the synchronization condition; *Cur* is the set of instances that have reached the condition so far; and *All* denotes the set of all instances which are involved in the synchronization.

Event 'Leave synchronization' (`leave_sync(`*id*`,`*pco*`)`) – When all instances engaged in a synchronization have entered the condition, the simulator skips the condition silently. However, for all instances in the MSC which send a message directly after the condition, an additional notification is required to indicate that the message is allowed to be sent now. To find out whether such an event has to be created by the simulator engine, an initial static analysis of the MSC is sufficient.

4.2 Test generation algorithms for MTC and PTCs

There are two approaches to generate a distributed test case based on a given SDL specification and an MSC test purpose: On the one hand, a complete behavior tree may be generated first which covers all signals exchanged between the tester and the SUT, plus additional information about synchronization. Based on this sequential test description, single behavior trees for all PTCs and the MTC can be derived. Alternatively, separate behavior trees for the PTCs and the MTC may be created immediately at the time of the simulation of the SDL specification. The test generation algorithms for both approaches are basically the same. For the sake of simplicity, we present a solution for the latter approach.

In Figures 6, 7 and 8, the construction of the behavior trees for the PTCs and the MTC is explained. Figure 6 describes the invocation of the test generation functions. In Figures 7 and 8, the core algorithms are presented.

The algorithms for the MTC and the PTCs are structurally similar. To a certain extend, the algorithm for the MTC is the inverse of the algorithm for

```
 (1)  testgen_ptc($i$,$state_{sim}$,$state_{test}$)
 (2)  {
 (3)    $E$ := next_events($state_{sim}$);
 (4)    if ($\exists\, e \in E:\ e$ = $pco$!$sig \ \wedge\ pco \in PCO_i$)
 (5)    {
 (6)      $nextstate_{test}$ := add_trans($state_{test}$,$e$);
 (7)      testgen_ptc($i$,next_state($state_{sim}$,$e$),$nextstate_{test}$);
 (8)    }
 (9)    else if ($\exists\, e \in E:\ e$ = enter_sync($id$,$pco$,$Cur$,$All$) $\wedge$
(10)                $pco \in PCO_i \ \wedge\ (PCO_i \cap All) \subseteq Cur \ \wedge\ All \not\subseteq PCO_i$)
(11)    {
(12)      $nextstate_{test}$ := add_trans($state_{test}$,$MCP_i$!Ready($id$));
(13)      testgen_ptc($i$,next_state($state_{sim}$,$e$),$nextstate_{test}$);
(14)    }
(15)    else for all $e \in E$ do
(16)    {
(17)      if ($e$ = $pco$?$sig \ \wedge\ pco \in PCO_i$)
(18)        $nextstate_{test}$ := add_trans($state_{test}$,$e$);
(19)      else if ($e$ = leave_sync($id$,$pco$) $\wedge\ pco \in PCO_i$)
(20)        $nextstate_{test}$ := add_trans($state_{test}$,$MCP_i$?Proceed($id$,$pco$));
(21)      else
(22)        $nextstate_{test}$ := $state_{test}$;
(23)      testgen_ptc($i$,next_state($state_{sim}$,$e$),$nextstate_{test}$);
(24)    }
(25)  }
```

Figure 7. Test generation algorithm for PTC$_i$

the PTCs. For example, if a PTC sends a message to the MTC, a corresponding receive event has to be added to the MTC behavior description. In addition, `CREATE` constructs and `DONE` events have to be added to the root and the leaves of the MTC behavior tree (line 10 in Figure 6 and line 5 in Figure 8). Due to the similarity of both algorithms, we will concentrate on the description of the PTC generation in the following.

We assume that the PTCs are numbered ($i \in \{1..n\}$). The set of all PCOs which belong to a parallel test component PTC_i is defined as PCO_i.

The PTC algorithm is invoked with three parameters: i denotes the number of the PTC; $state_{sim}$ is the current node in the reachability graph of the SDL system (initially the root node; see line 6 in Figure 6) and $state_{test}$ is the current node in the behavior tree to be constructed.

At first, all possible next events are requested from the simulator engine by function `next_state` (line 3 in Figure 7). Then it is checked whether the PTC can send either an ASP to the SUT (line 4) or a coordination message to the MTC (lines 9 and 10). Whenever a test component can send a signal, we assume that it should do so immediately. In this case, no other alternatives are taken into account. Instead, the send event is added to the behavior tree

```
 (1)  testgen_mtc($state_{sim}$,$state_{test}$)
 (2)  {
 (3)    $E$ := next_events($state_{sim}$);
 (4)    if ($E$ = $\varnothing$)
 (5)      $nextstate_{test}$ := add_trans($state_{test}$,?DONE($PTC_1$,...,$PTC_n$))
 (6)    else if ($\exists\, e \in E : e$ = $pco$!$sig \;\wedge\; pco \in PCO_{MTC}$)
 (7)    {
 (8)      $nextstate_{test}$ := add_trans($state_{test}$,$e$);
 (9)      testgen_mtc(next_state($state_{sim}$,$e$),$nextstate_{test}$);
(10)    }
(11)    else if ($\exists\, e \in E : e$ = leave_sync($id$,$pco$) $\wedge$
(12)              $\exists\, i \in \{1..n\} : pco \in PCO_i$)
(13)    {
(14)      $nextstate_{test}$ := add_trans($state_{test}$,$MCP_i$!Proceed($id$,$pco$));
(15)      testgen_mtc(next_state($state_{sim}$,$e$),$nextstate_{test}$);
(16)    }
(17)    else for all $e \in E$ do
(18)    {
(19)      if ($e$ = $pco$?$sig \;\wedge\; pco \in PCO_{MTC}$)
(20)        $nextstate_{test}$ := add_trans($state_{test}$,$e$);
(21)      else if ($e$ = enter_sync($id$,$pco$,$Cur$,$All$) $\wedge$
(22)                $\exists\, i \in \{1..n\} : pco \in PCO_i \;\wedge\; (PCO_i \cap All) \subseteq Cur \;\wedge$
(23)                $All \not\subseteq PCO_i$)
(24)        $nextstate_{test}$ := add_trans($state_{test}$,$MCP_i$?Ready($id$));
(25)      else
(26)        $nextstate_{test}$ := $state_{test}$;
(27)      testgen_mtc(next_state($state_{sim}$,$e$),$nextstate_{test}$);
(28)    }
(29)  }
```

Figure 8. Test generation algorithm for the MTC

(lines 6 and 12) and the algorithm is invoked recursively with the successor node of $state_{sim}$ and $nextstate_{test}$ (lines 7 and 13). Only if the PTC cannot send a signal, all possible events have to be considered, as indicated by the loop in lines 15–24.

If an event has to be appended to the behavior tree, a transition to a new node ($nextstate_{test}$) is inserted into the tree (lines 6, 12, 18 and 20). Otherwise, $nextstate_{test}$ is set to $state_{test}$ (line 22). By invoking itself recursively, `testgen_ptc` explores the whole state space of the SDL specification. Due to the interleaving semantics of SDL, the algorithm might reach a state where it wants to add an already existing vertex to the behavior tree. In order to handle this, function `add_trans(`$state$`,`e`)` has been introduced. If a vertex from *state* labelled with e exists, it simply returns the successor node; otherwise a new successor node is added.

There are two CMs used for the communication between a PTC and the MTC: CM `Ready(`id`)` is sent from a PTC to the MTC in order to indicate that

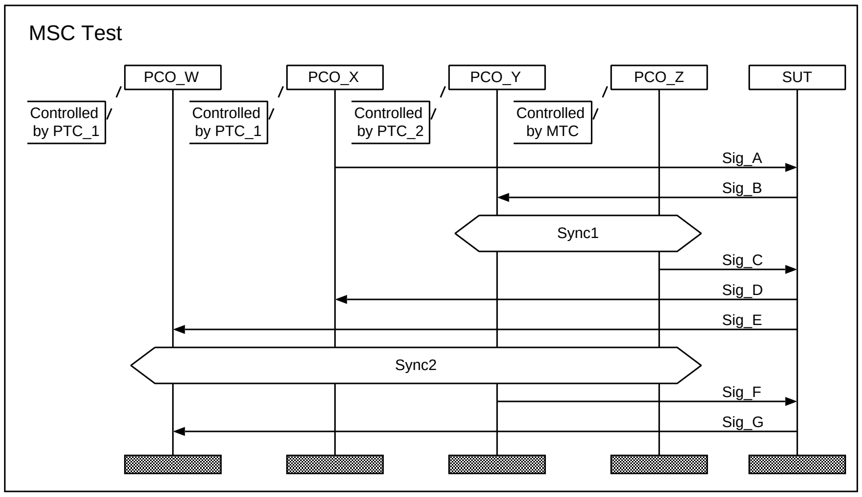

Figure 9. MSC *Test* used for test generation

all relevant instances of the PTC have reached synchronization condition *id* (line 12). It is only sent, if (1) an `enter_sync` event is reported by the simulator engine, (2) all instances of the PTC which are involved in the synchronization have already reached the condition ($(PCO_i \cap All) \subseteq Cur$) and (3) there are other TCs which are also involved in the synchronization ($All \not\subseteq PCO_i$) (lines 9 and 10). In reverse, CM `Proceed(`*id*`,`*pco*`)` is received from the MTC and indicates that the PTC is allowed to send further ASPs via *pco* (line 20).

4.3 Example

The test generation is illustrated by the following example. In Figure 9, an MSC test purpose is shown for which a test case is to be generated. For this MSC *Test*, a test configuration with two PTCs and an MTC is defined. `PTC_1` controls two different PCOs (`PCO_W` and `PCO_X`), whereas `PTC_2` communicates with the SUT only via `PCO_Y`. The MTC exchanges signals both with the PTCs via `MCP_1` and `MCP_2` and with the SUT via `PCO_Z`.

Figure 10 shows the behavior descriptions for both the MTC and the two PTCs (named `Test_PTC_1` and `Test_PTC_2`). As can be seen, only one synchronization message is sent to the MTC for each PTC. The size of an MTC mainly depends on the number of PTCs which are involved in a synchronization, since the MTC must be able to receive `Ready` CMs in every possible order.

5 Summary and outlook

In this paper, we have presented our approach to automatic generation of test cases in concurrent TTCN format. To do this, a test component configuration

Test Case Dynamic Behaviour

Test Case Name	**:**	Test_Concurrent
Group	**:**	
Purpose	**:**	
Configuration	**:**	Conf
Default	**:**	
Comments	**:**	The tester consists of a Main Test Component and two Parallel Test Components

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CREATE(PTC_1:Test_PTC_1, PTC_2:Test_PTC_2)			
2		MCP_2 ? Ready(Sync1)			
3		PCO_Z ! Sig_C	C_Sig_C		
4		MCP_1 ? Ready(Sync2)			
5		MCP_2 ? Ready(Sync2)			
6		MCP_2 ! Proceed(Sync2,Y)			
7		?DONE(PTC_1,PTC_2)		R	
8		MCP_2 ? Ready(Sync2)			
9		MCP_1 ? Ready(Sync2)			
10		MCP_2 ! Proceed(Sync2,Y)			
11		?DONE(PTC_1,PTC_2)		R	
		Test_PTC_1			
12		PCO_X ! Sig_A	C_Sig_A		
13		PCO_X ? Sig_D	C_Sig_D		
14		PCO_W ? Sig_E	C_Sig_E		
15		MCP_1 ! Ready(Sync2)			
16		PCO_W ? Sig_G	C_Sig_G	(P)	
17		PCO_W ? Sig_E	C_Sig_E		
18		PCO_X ? Sig_D	C_Sig_D		
19		MCP_1 ! Ready(Sync2)			
20		PCO_W ? Sig_G	C_Sig_G	(P)	
		Test_PTC_2			
21		PCO_Y ? Sig_B	C_Sig_B		
22		MCP_2 ! Ready(Sync1)			
23		MCP_2 ! Ready(Sync2)			
24		MCP_2 ? Proceed(Sync2,Y)			
25		PCO_Y ! Sig_F	C_Sig_F	(P)	

Detailed Comments :

Figure 10. A concurrent TTCN test case for MSC *Test*

and information concerning the synchronization of TCs has to be provided. If the synchronization is specified by means of conditions, a corresponding exchange of CMs between TCs can be generated automatically. We have provided the corresponding algorithms and presented a small example explaining the output of the algorithms.

The implementation of our approach has already started and the results of the first experiments have been promising. If our approach proves its applicability to real world examples, we also intend to implement it in the Autolink tool.

Acknowledgements
We would like to thank Stefan Heymer for valuable comments on earlier drafts of this paper and for careful proofreading.

References

[1] Cinderella SDL product description. `http://www.cinderella.dk`

[2] M. Clatin, R. Groz, M. Phalippou, R. Thummel. *Two Approaches Linking a Test Generation Tool with Verification Techniques.* Proceedings of *IWPTS'95*, Evry, 1995.

[3] L. Doldi, V. Encontre, J.-C. Fernandez, T. Jéron, S. Le Bricquir, N. Texier, M. Phalippou. *Assessment of Automatic Generation Methods of Conformance Test Suites in an Industrial Context.* Testing of Communicating Systems, vol. 9, Chapman & Hall, 1996.

[4] J. Ellsberger, D. Hogrefe, A. Sarma. *SDL – Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1997.

[5] ETSI TC MTS. *CATG Handbook.* European Guide, ETSI, Sophia-Antipolis, 1998.

[6] J. Grabowski, D. Hogrefe, R. Scheurer, Z.R. Dai. *Applying SAMSTAG to the B-ISDN Protocol SSCOP.* Testing of Communicating Systems, vol. 10, Chapman & Hall, 1997.

[7] J. Grabowski, D. Hogrefe, R. Nahm. *Test Case Generation with Test Purpose Specification by MSCs.* In "SDL'93 – Using Objects" (O. Færgemand, A. Sarma, editors). North-Holland, October 1993.

[8] ISO. *Information Technology – OSI-Conformance Testing Methodology and Framework – Part 1: General Concepts.* ISO IS 9646-1, 1994.

[9] ISO. *Information Technology – OSI-Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN).* ISO IS 9646-3, 1996.

[10] H. Kahlouche, C. Viho, M. Zendri. *An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol.* Testing of Communicating Systems, vol. 11, Kluwer Academic Press, 1998.

[11] ObjectGEODE product description. `http://www.verilog.fr`

[12] E. Rudolph, P. Graubmann, J. Grabowski. *Tutorial on Message Sequence Charts (MSC-96).* Forte/PSTV'96, Kaiserslautern, October 1996.

[13] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, B. Koch. *Autolink – Putting SDL-based Test Generation into Practice.* Testing of Communicating Systems, vol. 11, Kluwer Academic Press, 1998.

[14] Telelogic TAU product description. `http://www.telelogic.se`

[15] ITU-T Rec. Z.100 (1996). *Specification and Description Language* (*SDL*). Geneva, 1996.

[16] ITU-T Rec. Z.120 (1996). *Message Sequence Chart* (*MSC*). Geneva, 1996.