Automatic Generation of Conformance Tests From Message Sequence Charts

By

Paul Baker, Paul Bristow, Clive Jervis, David King, Bill Mitchell

System and Software Engineering Research Lab (UK), Motorola Labs.

ABSTRACT

Over the past five years our group has developed a tool that automatically generates conformance test scripts from a combination of Message Sequence Chart (MSC)s, specifying dynamic behaviour, and Protocol Data Units (PDUs), specifying data formats. This paper outlines how tests are derived from MSCs and PDU specifications, and summarises different test strategies. It describes the testing required to prove conformance of various MSC language features ranging from time constraints to MSC variables, in-line expressions and high-level MSCs. We cover test generation for both single process test scripting and concurrent test scripting, where a test is distributed across autonomous processes, co-ordinating through additional messaging. All of the above aspects have been implemented in our tool that is in widespread use across Motorola. Tool support has not only resulted in cycle-time benefits for test preparation, but quality improvement in the testing process, improved test coverage, and validation of requirements specification.

KEY WORDS – *Message Sequence Charts (MSC), Protocol Definition Units (PDUs), test generation, SDL, TTCN, co-ordination messages, conformance testing.*

Introduction

Over the past five years the UK System and Software Engineering Research group of Motorola Labs. has been developing *ptk*, a sophisticated tool used to generate conformance tests from Message Sequence Charts (MSCs) and protocol data unit specifications (PDUs). The tool is in use by Motorola engineering organisations around the globe. In this paper we present the theory that underpins the algorithms, some we have patented, that have been implemented in *ptk*. Today, *ptk* generates tests scripts in SDL and TTCN, but it also has a flexible code generator that can, in principle, be customised via C-macro definition libraries to produce scripts in other languages.

A distinguishing aspect of *ptk* compared to other MSC analysis tools is its adherence to MSC semantics, although there are places where we offer a user non-standard interpretations in addition. Correctly supporting MSC's non-compositional semantics results in high computational costs, both in time and space, compared to the approaches taken by other tools. Essentially, all constructs such as MSC references have to be in-lined.

1 Motivation

Our work on *ptk* began because we saw an opportunity to automate a manual process in the development lifecycle. Test teams were being asked to develop functional tests from suites of MSCs that had been used to define the requirements of the system under test. Tests were being constructed by hand from these MSCs, but invariably the tests were not exhaustive, and there was no guarantee that the tests reflected the correct test sequences exactly. Indeed since there were no tools available to the requirements engineers that did any kind of processing of MSCs, they were often used informally, and contained inconsistencies like the naming of messages or instances.

Not only did we see an opportunity to generate behavioural aspects of tests, but in generating the complicated packing routines used to construct outgoing test messages, and the unpacking/checking routines used for incoming messages. We had already developed a domain specific language for specifying these PDUs, from which we had built a tool to generate application code.

In developing *ptk*, the use of MSCs and the associated PDU definitions became more formal, since they could now be checked, and they were providing material benefit (i.e. the generated tests). This payoff exceeded the investment in the extra formality required. In using *ptk*, the engineers were closing the development loop from requirements specification through to system test.

2 MSCs & Testing

The aim of test generation is to extract from requirements MSCs test sequences that can be used to verify that an implementation has the behaviour specified by the MSCs. Typically the object of the tests is an implementation of several of the instances represented in the MSCs. That is, architecturally, the implementation consists of a number of concurrent processes, each represented by one of the instances in the requirements MSCs.

Given an MSC and a declaration of which instances constitute the Implementation Under Test (IUT), test generation is about picking out the MSC events that are responsible for communicating with the IUT instance, and assembling tests from these that respects the partial order defined by the MSC. For example, neither of the two events representing a message that is passed between two IUT instances, or between two non-IUT instances, will appear in a test script. Only messages that cross between an IUT instance and a non-IUT instance or vice-versa will appear in a

test script.

Even though such *internal* events will not appear in the generated tests, they can impose an order on the *test* events that our test generation algorithms have to take into account. Aside from messages, events such as non-ITU action boxes may also be retained in test scripts, for example, if they are used to compute values used to set the parameters of subsequent outgoing test messages.

A test may have to cope with specified non-deterministic behaviour of the IUT, and conversely, where non-deterministic behaviour is admitted by the non-IUT instances, we generate separate tests for each possible behaviour. In this way a single MSC may generate many tests.

3 ptk

Today *ptk* covers most of the MSC language, including co-regions, MSC references, in-line expressions, high-level MSCs, time constraints, message parameters, and MSC variables. One distinguishing feature of *ptk* compared to most other tools that formally process MSCs is that it closely adheres to the correct MSC semantics.

In addition to MSCs and PDU specifications, *ptk* reads a directives file, loosely based upon an MSC document. The directives file contains MSC information, such as variable declarations, the names of files containing the message (PDU) specifications, and test generation information, such as the names of the instances that constitute the Implementation Under Test, or the names of MSCs that are to be used to generate preamble and postamble test sequences.

Message specifications take the form of PDU specifications. We support PDU specifications defined in a TTCN static part file (in the case of TTCN test generation), or a domain specific language for the TETRA [4] air interface protocol (used for SDL test generation). *ptk* will cross-check data use in an MSC with the data declarations contained in the message specification files. For example, it will check that an MSC variable’s type is consistent with a message’s declared field type when used as a message parameter. In the case of the domain specific language that *ptk* supports, the tool generates all the bit exact packing routines required to send test messages, and all the bit exact unpacking and checking routines required to validate the contents of an incoming message.

ptk has been integrated with Telelogic’s TAU suite. It can be invoked from pull down menus in either an MSC editor or organiser window. Motorola and Telelogic have for the past year entered a relationship whereby Telelogic have distributed *ptk* back into Motorola for a small licence fee. *ptk* uses the same licensing software as other Telelogic tools.

4 Overview

Section 2 introduces the basic approach to test generation used by *ptk*, and discusses the various strategies, such as one test per trace, and always complete test where possible. The strategies balance number of tests generated versus the guarantee of test coverage. It also introduces the semantic model, in the form of sets of partial graph structures, we have used to perform the formal analyses, and likewise the code tree structure used to represent test scripts. We also cover some of the optimisations we have implemented to reduce the number and size of the suites we are generating. For example, in selecting one linear ordering of interleaved action boxes, or the automatic extraction of test steps.

MSC is a rich language having many features, and so Section 3 gives some details of how *ptk* treats the more interesting of them. In some cases, such as MSC references, we offer both true MSC semantics support, and more pragmatic support requested by users. In the case of MSC references the correct semantics involves in-lining the references where as the pragmatic support treats the reference as ‘atomic’. However users do not often appreciate the consequences of what they think they want – suppose an MSC reference generates more than one test script? We cover high-level MSCs, MSC reference expressions, in-line expressions, plain MSC references, and action boxes.

In Section 4 we show how MSC time constraints are verified dynamically in the generated tests. Given that Telelogic’s MSC editor does not yet support MSC2000 time constraints, we have used comment boxes to mimic them. It is possible that two or more time constraints lead to an inconsistent MSC, i.e. one that has no trace satisfying all constraints. Our test scripts capture and record such defects dynamically.

Section 5 introduces our solution to concurrent test generation. The theoretical challenge here was the development of an algorithm that generates messages used to co-ordinate and synchronise the operation of each script that guarantees the overall message sequences to and from the implementation under test occur in the order prescribed by the MSCs.

Section 6 covers the data aspects of test generation. For example, unspecified message parameters are treated as "don't care" fields in messages received from the implementation under test. *ptk* has an option where all data occurrences can be treated as uninterpreted strings, so that it may be used for test languages that are not directly supported by it. Currently we have only limited support for MSC variables, but work is underway to provide a more comprehensive treatment. We discuss the correct approach, and place emphasis upon the difference between an MSC variable, and a test script variable that is used in verifying the behaviour of the former – even though it is tempting to think of them as the same thing.

We finish the paper with some of the benefits that *ptk* has brought to users, and the directions of future work, where we are moving towards MSC V&V as the test generation work reaches completion.

Test Generation From MSCs

This section describes the technical details of how *ptk* generates conformance tests from an MSC where each test is executed as a single process. In Section 5 we consider the generation of tests for a distributed test environment.

ptk generates conformance tests from an MSC based on a collection of instances that describe the implementation under test (IUT), the instances are defined by the user. Events from the IUT are not included in the test scripts since *ptk* generates black box tests. *ptk* generates one or more test scripts from a single MSC. Depending on which options are invoked ptk will produce a combination of the following:

- Test script analysis file - contains a set of traces that describe the sequences of events that must be verified by the test scripts, and a set of test scripts in an abstract form. This file can be reviewed to determine how many tests are required.
- Test script graphs and trees - files representing various test graphs and trees in a form that can be viewed using the daVinci [5] graph package.
- Macro templates/test script files - these are the actual test scripts which are for execution by a test environment to verify the functionality of an MSC. *ptk* generates SDL directly but for TTCN, and possibly other languages, it generates a macro code template. These macro templates can then be combined with macro libraries to produce executable test scripts for a particular testing environment.
- Message Sequence Charts (MSCs) - a separate MSC is generated for each test trace illustrating the interactions between the IUT and the test script.

Figure **2.1**

1 Semantic Model of MSCs

As well as generating test scripts, *ptk* can also be used to analyse the semantics of an MSC. This can be particularly useful, since it is common that engineers do not appreciate the full interleaving semantics of MSC when they are developing MSCs. *ptk* can be used to generate all the traces of events through an MSC, which are placed into a semantics log file. A tree of all possible traces can also be generated and displayed using visualisation tools such as daVinci [5], and dot [ref], as well as plain text.

2 Test Generation

The input to *ptk* necessary to generate test scripts is an MSC together with a directives file. The directives file has a similar purpose to MSC documents, see [1], and contains information such as which part of the MSC represents the Implementation Under Test (IUT), sometimes called the System Under Test (SUT). The directives can also be placed on the MSC, as with the IUT definition in Figure 2.2 which illustrates an MSC, and directive, where the IUT is given as instance Q.

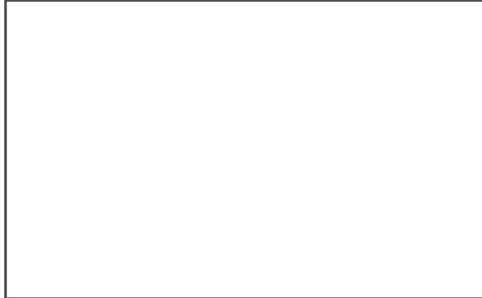


Figure **2.2, MSC with directive**

When generating test scripts from an MSC the test script only contains events that interact with the IUT. This is black box testing, so events on the IUT are not included in a test, and events that do not send to or receive from the IUT are not included in the test. For example, the test scripts for the MSC in Figure 2.2 will include the sending of message *x* and the receipt of messages *y* and *z*, but other events such as receiving message *x* or the events for message *w* are not included.

3 Test Strategies

ptk supports three non-concurrent test strategies, each potentially generating different test cases. The different tests are due to the fact that a test can express non-determinism with receive events, so it is possible to do more testing per test case. For example, Figure 2.3 and Figure 2.4 demonstrate two possible sets of test scripts that could arise from the same MSC (Figure 2.2).

Figure 2.3, one test script

Figure **2.4, two test scripts**

Note that these figures are abstract forms of the test scripts, and don't include details such as timeouts or recording of log messages that are typical in a test script. These abstract test scripts may be generated by *ptk*, and viewed with tools such as dot [cite], or daVinci [5], or just in plain text. The visualisation of the abstract form can be extremely useful in understanding what the test case is actually testing.

The three test strategies employed by *ptk* are as follows:

1. **One test per trace**: here each trace through an MSC is placed into a separate test script. This is the simplest test strategy used by *ptk* and is the same as the strategy used by Telelogic's AutoLink tool [autolink]. To avoid false fails (i.e. tests which fail when the implementation is behaving to specification), we use INCONCLUSIVE verdicts. An INCONCLUSIVE verdict means that we don't know if the test has passed or failed. With the above example, Figure 2.4, we would have test scripts as depicted in Figure 2.5 and Figure 2.6. The advantage of using one test per trace is that we can ensure that all possible traces have been tested by checking the results of the tests.

Figure **2.5, 1st script with INCONCLUSIVE**

Figure **2.6, 2nd script with INCONCLUSIVE**

2. **Optimised tests**: here non-determinism of receives is exploited so, for example, the test script for the MSC in Figure 2.2 will be as shown in Figure 2.3. This strategy generally generates fewer tests than the one-test-per-trace strategy and is the default used by *ptk*. One property of tests generated with this strategy is that each test has a fixed order of non-receive events. For example, a test such as that shown in Figure 2.7 is not allowed, but the test shown in Figure 2.8 is allowed. The rationale for maintaining this non-mixed order of non-receives property is that the test system has control over these events, and we would like to test all possible orders of these events. There is also the benefit of knowing which orders fail, by looking at the failed tests.

Figure **2.7, incorrect send order**

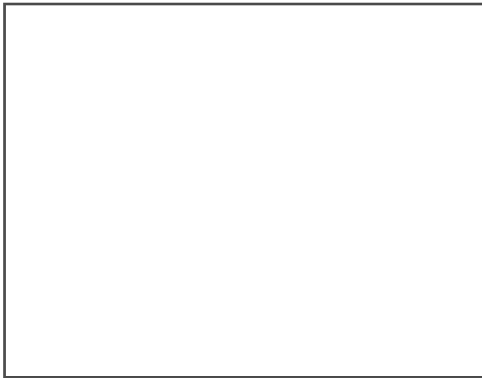


Figure **2.8, correct send order**

3. Optimised tests - merging alternatives: this test strategy is a variation on the second strategy that changes the test cases when alternatives are used. With the strategy above, (2), separate operands of an alternative are placed into separate test scripts. This holds with the intuition that separate operands are representing different behaviours, and should be tested separately. If this is not an issue, and it is possible to merge separate operands into the same script, then a merging strategy may be used. With the merging alternatives strategy operands of an alternative that can be placed into the same script are done so.

Figure 2.9, MSC with alternatives

For example, with the MSC in Figure 2.9 we may have scripts as shown in Figure 2.10. In general this test generation strategy generates the fewest number of tests, although there are cases where this strategy generates more test cases than in (2), such as with the example in Figure 2.11.

Figure **2.10, Test for MSC in Figure 2.9**

Figure **2.11, An example MSC where test strategy 3 is worse than strategy 2.**

1 Implementing the Test Generation Algorithms

ptk uses several different datatypes before generating target test code. The MSC input to *ptk* must be in the textual MSC format that is generated by MSC editors such as Telelogic Tau [tau]. Once this is parsed into a syntax tree, internally a graph is used to represent the MSC where events are graph vertices, and an order between two events is a graph edge.

Figure **2.12, Graph for Figure 2.2**

Figure 2.12 shows a graph representing the MSC in Figure 2.2. When we generate test scripts the graph representing an MSC is reduced to a smaller graph that just contains the vertices that interact with the IUT. Each test script algorithm takes this graph and generates an abstract test tree similar to those in Figure 2.3 and Figure 2.4, where events are tree nodes. The abstract trees are then converted to another more detailed tree format that we call a code tree. Code trees contain events like abstract test trees, but also contain more detailed information such as timer starts, timeouts, cancel timers, receive anything nodes, besides others. These code trees are the basis for each of the code generators. They contain enough information to generate code, but not too much so that each code generator can use

the same datatype as input.

4 Optimising the Test Scripts

Test scripts may be thought of as just trees of events. These trees can become extremely large even from modestly sized MSCs, because all the possible interleavings of events need to be expressed. This was in fact an issue for a group of *ptk* user's working on Dimetra [ref], where the SDL test scripts that *ptk* generated where so huge that SDL compiler was unable to compile the scripts.

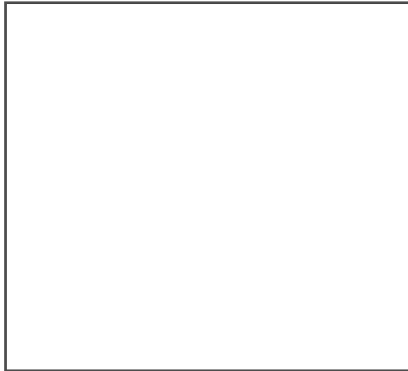


Figure **2.13, MSC with several interleavings**

We were able to reduce the size of test scripts quite dramatically by factoring out common subtrees from the generated tests. The common subtree is replaced by a procedure call, the body of which contains the factored tree. In some cases this optimisation reduced the size of the tests by tenfold. The factoring was done, both in each script itself, and in the scripts as a whole.

Figure **2.14, Test script with common subtrees**

Advanced MSC features

ptk supports many advanced features of the MSC language, here we describe our support for HMSCs, inline and reference expressions, and data guards.

1 HMSCs, inline, and reference expressions

A consequence of using MSCs in industrial-size applications is the tendency to use them in a modular and hierarchical fashion, just like other specification languages.

Figure 3.1, An example of a high-level MSC

Hence, in 1996, the standard Z.120 [1] evolved to allow the description of larger systems by introducing the notion of in-line expressions, MSC reference expressions and High-Level MSCs (HMSCs); each of these constructs enabling the parallel, sequential, looping, and alternative composition of MSCs. For example, Figure 3.1 illustrates a simple HMSC in which the behaviour defined by *msc1* is sequentially composed with the alternative composition of *msc2a* and *msc2b*. In other words after executing the actions defined by *msc1* the environment has the choice of executing the actions defined by *msc2a* or *msc2b*, but not both before terminating. Where, *msc1*, *msc2a* and *msc2b* are references to either other HMSCs or basic MSCs.

MSC reference expressions and in-line expressions allow the composition and modularisation of actions defined by an MSC specification. For example, Figure 3.2 illustrates a simple MSC containing two instances P and Q upon which a *loop, in-line expression* has been placed. This expression (as denoted by the rectangular box) means that the passing of message *m* must be executed at least 0 and at most 2 times, before message *n* can be passed. MSC references can contain expressions using *alt, par, seq* and *loop* operators.

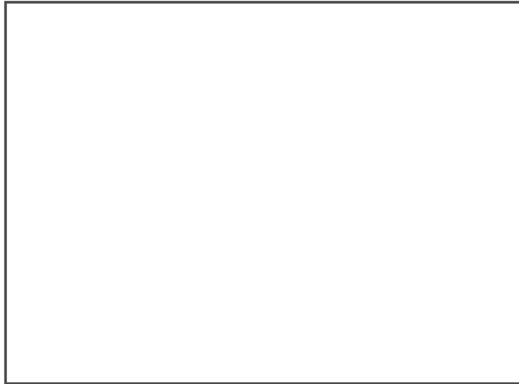


Figure 3.2, An MSC with a loop expression

HMSCs, inline, and reference expressions except atomic references are all supported in *ptk* by combining them to form one larger MSC, or multiple MSCs in the case of alternative expressions. The combined MSCs can be processed by *ptk* in the same manner as if they were a single MSC. HMSCs cause a few problems due to the fact that they do not explicitly contain instances, and hence, this information must be derived from the referenced MSCs. We have imposed the following rules on the use of instances to ensure consistent use of instance names:

1. An instance can be created only once.
2. An instance can be stopped only once.
3. An instance cannot be created once it has been used: it is assumed to already exist.
4. An instance cannot be recreated once it has been stopped.
5. An instance cannot be used after being stopped.
6. MSCs in parallel cannot create or stop instances used in other parts of the parallel expression.

These rules are applied to all referenced MSCs whether they are HMSCs or not. Although this is not strictly necessary for all rules, for example it would not be possible to draw a basic MSC that breaks rule 5, it is simpler to apply the rules to all MSCs than to be selective.

2 MSC inline and atomic references

There are two types of reference allowed within *ptk,* inline and atomic. The semantics for the inline reference are the standard semantics that replaces the reference name by the referenced MSC. An atomic reference acts as a placeholder in test scripts for a call to the test script of the referenced MSC. This will be called with actual parameters replacing the formal parameters. If the referenced MSC does not exist it is assumed that there is a preconstructed test script that can be called when the master test scripts are executed. The semantics for an atomic reference are only valid where the referenced MSC generates a single test script.

Atomic references were introduced because events within inlined references are interleaved with events from the containing MSC, in the final test scripts. This may result in many test scripts being generated where only one was intended. It is often the case that a designer wishes to use a reference in the style of a procedure call in a programming language, which makes the inlined semantics inappropriate.

Figure **3.3**

Figure 3.3 shows an example of the syntax used in *ptk* for defining an atomic reference. Figure 3.4 is the referenced MSC. The ‘+’ symbol denotes that the reference should be atomic rather than inline. The actual parameters to the atomic reference are listed after the MSC name. Within the referenced MSC the formal parameters are listed in a text box using the standard *ptk* comment symbols to delimit a key word.

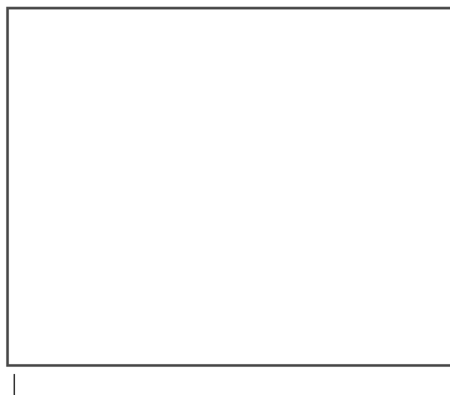


Figure **3.4**

Figure 3.5 and Figure 3.6 give a listing of the ASCII test scripts generated from Figure 3.3. Note that the receive *a* event can interleave with the atomic reference call but not with the events contained within the atomic reference itself. Hence, there are two test scripts generated from this example.

```
|=== MSC Master_MSC_containing_atomic_reference    |
|TEST SCRIPT (1 of 2) ===                          |
|START                                             |
|(                                                 |
|(                                                 |
|RECEIVE a                                         |
```

```
|RECEIVE b                                          |
|(ATOMIC_REF Msc_reference_name(a,3b+w,x))          |
|PASS                                               |
|)                                                  |
|OR                                                 |
|(                                                  |
|RECEIVE b                                          |
|(ATOMIC_REF Msc_reference_name(a,3b+w,x))          |
|RECEIVE a                                          |
|PASS                                               |
|)                                                  |
|)                                                  |
```

Figure **3.5**

```
|=== MSC Master_MSC_containing_atomic_reference     |
|TEST SCRIPT (2 of 2) ===                           |
|START                                              |
|(                                                  |
|(                                                  |
|RECEIVE a                                          |
|RECEIVE b                                          |
|(ATOMIC_REF Msc_reference_name(a,3b+w,x))          |
|PASS                                               |
|)                                                  |
|OR                                                 |
|(                                                  |
|RECEIVE b                                          |
|RECEIVE a                                          |
|(ATOMIC_REF Msc_reference_name(a,3b+w,x))          |
|PASS                                               |
|)                                                  |
|)                                                  |
```

Figure **3.6**

Time Constraints

The MSC standard contains a number of techniques for describing timing constraints. Within *ptk* we have implemented timing constraints based on time intervals as described in the standard. Time intervals use two time points to describe a window of time that can be relative to a given event or based on an absolute time. The implementation only supports timing constraints relative to message events but there is no fundamental reason why the same techniques could not be used for any type of event or even absolute times, as absolute times can be considered to be relative times in relation to some notional start event at the beginning of an MSC.

ptk does not use timers explicitly to constraint events as this is not consistent with the semantics as defined by the ITU MSC2000 standard [1], in which timers are not intended as explicit constraining mechanisms.

1 CONSTRAINT REPRESENTATION

The MSC standard includes a graphical notation for time intervals, and when deciding on a representation to use with *ptk*, this is clearly the most desirable. In spite of this, as *ptk* uses the Telelogic Tau MSC editor, we had to take account of the complete lack of support for time intervals within it. As a result we decided on a notation that is attached to events using a comment box. We have tried to keep the semantics close to that in the standard in order to allow a transition to it should support become available in Tau at a later date. However, we have added a little syntactic sugar to try and make the meaning of a constraint obvious.

The representation we have used has four types of constraint, as follows:

- A bounded constraint with an upper and lower bound, e.g. *(lower bound, upper bound) name*;
- A maximal constraint in which two events must be separated by less than a maximal time, e.g. *max < name*;
- A minimal constraint in which two events must be separated by at least a minimal time, e.g. *min > name*;
- A wait constraint that imposes a delay between an event occurring and the sending of a later message, e.g. *wait delay name*.

By combining the constraints with a label, which is called *name* above, an interval is described. The type of the interval is determined by the final event in the interval, as the final event is what is being constrained with respect to the occurrence of the earlier event. The first three constraints are for use when the final event in an interval is a receive; the fourth constraint is for when the final event is a send. In some MSCs, due to only partial ordering, the order of constrained events may be changed resulting in a constraint being applied to the wrong event type. In these situations *ptk* does not apply the constraint to the paths in which its usage is incorrect, however, if there are no valid paths for the usage of a given constraint on an MSC, *ptk* will return an error to the user and terminate.

Figure 4.1, MSC example with two timing constraints

Figure 4.1 shows an MSC with three messages and two timing constraints, a bounded constraint *L1*, and a wait constraint *L2*. Both constraints are relative to the first message *m* but constrain different final messages, *n* and *o* respectively.

2 IMPLEMENTING TIMING CONSTRAINTS WITH TIMERS

Individual timing constraints, although not actually timers, have a straightforward implementation with timers. For example, a bounded constraint can be implemented with two timers, one for the lower and one for the upper bound; both timers are started when the event at the beginning of the constraint occurs and the final message must be received after the lower timer expires and before the upper. The process is considerably complicated when constraints overlap and by differing types of constraints and messages, however, we will not consider these algorithmic issues here.

When using timers to represent timing constraints, we have to test for the expiring of all timers at any point in a test script where we are waiting for an input in the form of a receive or a timeout; we will refer to these as input points. Unfortunately, not only can any timer expire, they all can and in any order. Hence, we must test for all permutations of timers expiring at each input point in a test script, a costly operation with complexity ***O(n!)***. Fortunately, a test script is unlikely to have a large number of timers running concurrently so the cost of this calculation should not be prohibitive. Figure 4.2 shows the TTCN generated by *ptk* for the MSC in Figure 4.1.

Figure 4.2, TTCN for Figure 4.1

3 Alternative expressions

The next version of *ptk*, 3.4, will have limited support for timing constraints in alternative expressions. Alternative expressions complicate the addition of timers for testing timing constraints due to the fact that we do not know which path through an alternative any test run may follow. Additionally, MSC semantics allow the same timing constraint to finish in more than one clause of an alternative expression, further complicating test script generation. Figure 4.3 shows a simple MSC with an alternative expression and two maximal constraints that finish one in each clause of the alternative.

| |

Figure 4.3, Simple MSC with alternative and timing constraints

The addition of timers for alternative expressions is significantly further complicated by the addition of extra successful paths through a test script. These are in addition to those that would exist either without timing constraints or if we retained the timing constraints but had separate MSCs for each clause of the alternative. These extra paths arise because paths with timeouts on that previously led to a fail verdict may no longer do so. This can occur when a timeout means that one path through an alternative cannot be taken but a different path is selected anyway. Figure 4.4 shows the TTCN from Figure 4.3 when *a* is the instance under test (IUT), notice the timeouts of *L1* and *L2* and that it is still possible for the test script to terminate successfully after them occurring, as long as the correct path through the alternative is taken. Note that this is only one of two TTCN scripts generated by *ptk* for this MSC, the other script is the almost the same but with the fail and inconclusive (*inconc*) verdicts reversed, as compared to this script.

| |

Figure 4.4, TTCN for Figure 4.3 when *a* is IUT

Concurrent Test Generation

ptk can automatically generate concurrent conformance test scripts from requirements MSCs. The concurrent conformance test suite defined by a set of MSCs can be regarded as the usual non-concurrent test suit converted into a form that can be executed over a distributed environment. To generate the concurrent conformance test suite *ptk* generates the non-concurrent tests and on the fly converts them to a concurrent form using a patented algorithm.

Concurrent test scripts were incorporated into TTCN in 1996. The TTCN model of concurrency allows the implementation under test (IUT) to be interrogated by a collection of parallel test components (PTCs) to determine if the IUT passes some given concurrent test. The PTCs can be autonomous processes running over a distributed system. In order for the PTCs to synchronise with each other they use coordinating messages (CMs).

The standard formal semantics for MSCs in [1] define messages to have arbitrary latency. In [2] it is shown that if the coordinating messages also have arbitrary latency then it will not be possible to automatically generate suitable coordinating messages which allow the PTCs to conduct a given concurrent test.

In order for *ptk* to overcome the problems raised in [2] it assumes that all CMs can be implemented with negligible latency compared with ordinary messages which travel through PCOs. This is a reasonable assumption where the CPs are channels implemented by the testing teams separately from the IUT. This assumption ensures that a CM sent prior to an ordinary message is guaranteed to be received before the ordinary message.

The TTCN model of concurrent testing assumes that the IUT is running in parallel with a unique master test controller (MTC) and a number of PTCs. The MTC is in overall control and it is also responsible for storing the test verdict. This verdict can be pass, fail, or inconclusive. The TTCN model supposes that there are dedicated communication channels between the test components. Points of control and observation (PCO) are the channels for the PTCs and MTC to interact with the IUT, the coordination points (CPs) are channels between the PTCs and MTC which are used to coordinate their activity via special messages known as coordinating messages (CMs). Figure 5.1 depicts the TTCN test architecture model.

Figure **5.1**

There are two difficulties with the use of coordinating messages in concurrent conformance test scripts. The first of these is to ensure that a concurrent test is sound, that is, if the PTCs and the IUT are acting correctly with respect to the test, then the IUT will pass the test. The second problem is to ensure that the coordinating messages are faithful, that is, the observed behaviour by the PTCs exactly corresponds to the intended behaviour defined by the test. If this is not the case the IUT may pass a test run because it complies with the behaviour that the PTCs are capable of observing, even though it should have failed because it did not comply with the test itself.

The *ptk* algorithm generates concurrent conformance tests that are sound and faithful provided that the MSC is race free, see [3] for details. The algorithm is close to optimal in that the number of CMs it uses to define a concurrent test are close to the minimum required. The algorithm is only a polynomial time factor more complex than the *ptk* algorithm for generating non-concurrent conformance tests. The algorithm introduces CMs directly between the PTCs, which results in more compressed test scripts compared to message exchanges routed through the MTC.

As an example consider the MSC in Figure 5.2. One non-concurrent test for this example is where *m2* is sent before *m1*, then after *m1* is received *m3* is sent. If each of the PTCs and the IUT are dislocated then a concurrent version of this test will require CMs to ensure that the PTCs can synchronise correctly and establish what the test verdict should be. Recall that *ptk* generates tests for black box testing, so it is not possible to make assumptions about the order in which events occur within the IUT. Hence, with black box testing, it is only possible to test that *m3* is received after *m1* is sent.

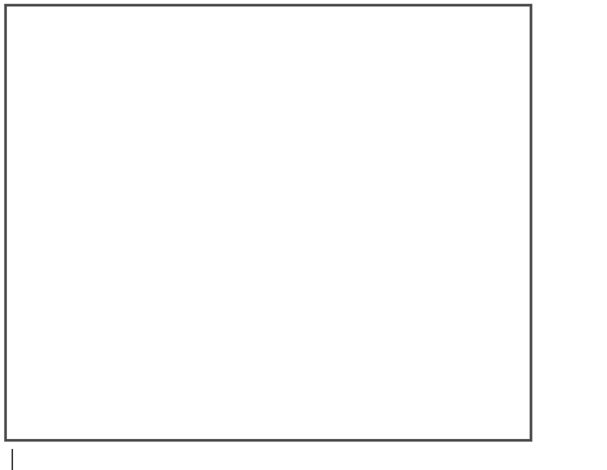
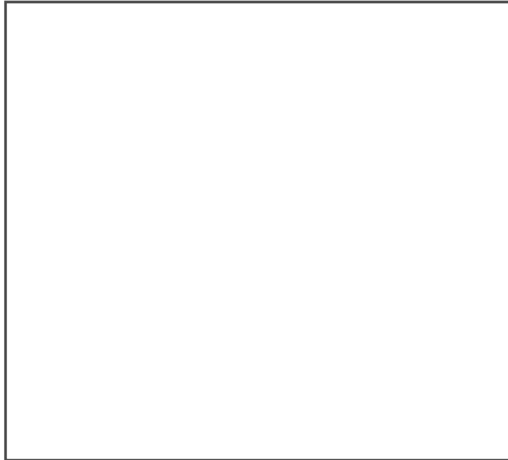


Figure **5.2**

To define this concurrent test requires two coordinating messages. These are shown in Figure 5.3. Message *cm2* is a CM that ensures that PTC A waits for *m2* to be sent before sending *m1*. Hence, *cm2* ensures that the PTCs are correctly synchronised with respect to the test. Message *cm3* is a coordinating message that allows PTC C to decide what the correct verdict should be after the test has run. If *m3* is received before *cm3* then the test will judge that *m3* may have been sent before it was triggered by *m1*. The test will then result in a fail verdict. If *m3* is received after *cm3* then the test will be given a pass verdict.



| |

Figure **5.3**

Data Support

Data may be placed on an MSC in message parameters, data guards, and action boxes. *ptk* supports two data languages which are TL, used with TTCN-2, and DL (Data Language) used with SDL [ref]. Uninterpreted data is also supported, this is data which is not parsed or interpreted, but will be passed to the code generators, and appear in test scripts. A formal data specification file for messages may also be used, and consistency checks are performed between this file and the MSC data. The specification languages currently supported by *ptk* are TTCN-2 and SL (Specification Language) for SDL. We are currently implementing support for TTCN-3 data. *ptk* does not by any means do every possible consistency check on data, since the possibilities are vast. We are gradually adding more checks though, such as checking that variable are used correctly, e.g. that they are in scope.

Related automatic test case generators

Here we briefly summarise related tools that have come to our attention.

- *uBET* [?] – developed in Bell Research Labs, and is currently undergoing productisation by Lucent Technologies. uBET provides an MSC editor, consistency checking of MSC's to detect race conditions, and timing violation, and also generates Promela from MSCs. uBET can also generate test cases in MSC. uBET is more advanced than *ptk* in detecting bad specifications such as those with the inconsistencies mentioned above. *ptk* is more advanced in supporting several test generation algorithms, and several test script languages.
- *TestMaster* – from Teradyne [?] is a testing tool providing automatic test generation and test execution. TestMaster generates black box test cases automatically from a functional model of the system (using Extended Finite State Machines). It includes facilities for developing this model. TestMaster relies upon user-defined code snippets, and can thus generate tests in the user's chosen language. If finite state machines are suitable for modelling you system then TestMaster is an ideal tool. Typically, though telecom systems make heavy use of concurrency, and this can cause an explosion in the complexity of the finite state machine model. We believe that TestMaster is currently not available.
- *AutoLink* – available from Telelogic [?] and must be used with Telelogic Tau. AutoLink generates one test case per trace in TTCN-2 from either MSCs, SDL models, or both. AutoLink is a semi-automatic test generator, and is typically harder to use than *ptk*. It is less sophisticated in the MSCs that it can handle than *ptk*, for example, MSC co-regions are not supported.

Conclusions

In this paper we have described automatic test generation, and in particular described the tool *ptk* which generates conformance test cases from Message Sequence Charts. Recently, *ptk* has been commercialised jointly by Telelogic and Motorola Labs, and is available within Motorola only, under licence. Many of *ptk*'s features were described in

detail such as the different test strategies used, and timing constraints.

Using tools like *ptk* greatly improves productivity, as well as the coverage of tests. It is hard to get reliable metrics, but Motorola Copenhagen have used *ptk* to generate SDL for Dimetra and have estimated a 33% reduction in the time to develop tests. Motorola India have also used *ptk* to generate TTCN-2 for a number of projects, e.g. GPRS, and have estimated a 10X improvement. *ptk* has been successful within Motorola, with approximately a dozen different projects using *ptk*, and each group achieving a software productivity improvement, as well as improving the test coverage.

In the future we hope to get more people in Motorola using automated techniques such as *ptk*, and believe that software productivity will be greatly improved by doing so. We are currently actively supporting the TTCN-3 [ref] test script language standardisation, as well as implemementing support in *ptk* for TTCN-3. We are also moving towards doing more validation and verification of MSCs, for example, detecting specifications that are impossible to implement. Again, we hope that this work will improve the producitivity and quality of the software developed and used in Motorola's products.

References

1] ITU-T Recommendation Z.120, "*Message Sequence Chart*", Geneva 2001. http://www.itu.int/itudoc/itu-t/approved/z/index.html

[2] Jens Grabowski, Beat Koch, Michael Schmitt, Dieter Hogrefe, SDL and MSC Based Test Generation for Distributed Test Architectures. In: 'SDL'99 - The next Millenium' (Editors: R. Dssouli, G. v. Bochmann, Y. Lahav), Elsevier, June 1999. http://www.itm.mu-luebeck.de/english/publications/grabowski.html

[3] W. P. R. Mitchell, Characterising Concurrent Tests Based on Message Sequence Chart Requirements, Proceedings of Applied Telecommunication Symposium (ATS 2001), Ed Bodan Bondar.

[4] TETRA, http://webapp.etsi.org/workprogram/Report_WorkItem.asp?WKI_ID=6661

[5] daVinci, http://www.informatik.uni-bremen.de/daVinci/

SDL
TTCN-3
uBET
TestMaster
AutoLink

----

n

o

P

Q

/*<L1, L2>*/

/*<(5s, 10s) L1>*/

/*<wait 12s L2>*/

m