Requirements-Based Test Generation: An Industrial Perspective

Paul Baker[2], Paul Bristow[2], Clive Jervis[2], David King[2], Bill Mitchell[1], Robert Thomson[2]

[1] w.mitchell@surrey.ac.uk, Department of Computing, University of Surrey, Guildford, GU2 7XH, UK

[2] {paul.baker, paul.bristow, clive.jervis, David.King, brt007}@motorola.com
Motorola Labs, Jays Close, Basingstoke, RG22 4PD, UK

Abstract. This paper discusses our experience with the deployment of requirements-based test generation within a large industrial setting. In doing so, we present an overview of our technologies and changes to the technical approach needed to aid deployment, specifically the introduction of requirements validation.

1 Introduction

Like many other large industrial organizations Motorola is looking to reduce the cost and time required for the development of systems and software. In doing so, increased emphasis is being placed on reducing costs related to testing and inspection activities, by introducing automation and reuse into the development process. Automation is enabled by the development of abstract, yet rigorous models, throughout the development process, and reuse generally through the use of standards and common frameworks. In this paper, we present our experiences in trying to deploy automation through model-based development technologies that support standard notations, such as Message Sequence Charts (Message Sequence Charts (MSCs) [8], UML 2.0 Sequence Diagrams [16], and TTCN-3 [5]. In particular, technologies that reduce the:

- time to develop conformance test plans and test suites, through the automatic generation of tests from scenario-based requirement and architecture specifications – *automatic test generation.*
- cost of appraising requirement specifications by introducing technologies that automated the discovery of defects – *requirements based validation.*

1.1 Automatic test generation

Experience has shown that during the development of telecommunication software as much as 40-75% of the resources are spent on testing [2,6]. The benefits of automatic test generation can reduce the cost of producing and maintaining test

cases, as well as improving test coverage and test quality. Test generation tools produce comprehensive sets of test scripts from a user-defined high-level representation, or test model, of the system to be tested. There are several choices for the language used for describing the test model, and there are different test script generation tools for each language [7].

All test generation tools perform a comprehensive analysis of the supplied test model in order to produce a set of tests. Typically, they attempt some kind of exhaustive exploration of the test model. If the process is exhaustive, then the set of tests produced is functionally complete relative to the test model. That is, if the system under test passes all of the generated tests, then we can be certain that it has the precise behaviour specified by the test model.

Even simple test models have the capability of generating very high numbers of tests, and are often potentially infinite. Therefore, test generation tools generally employ some method of constraining the exploration of the test model in order to produce a finite but reasonable set of generated tests. Thus, a user generally has to supply a set of constraints along with the test model to the test generation tool. Clearly, the definition, or selection, of constraints is critical to the adequacy of the generated tests.

The key benefits of using automatic test generation are:

- Reduced time and effort needed for producing test suites;
- Greater test coverage;
- Improved test suite quality;
- Easier test maintenance;
- Reduced opportunity for introducing defects.

Generally, the cost of constructing a test model and any constraints is far less than the cost of hand producing an equivalent test suite. Indeed, the number of tests produced by an automatic test generation tool is far higher than that normally produced by hand. This is because cost and time restrictions, and possibly complexity issues, imposed upon hand produced tests usually lead to 'representative' tests being written only. Whereas, the exhaustive nature of auto-testing tools mean that many, possibly all, variations of each basic test are generated. This leads both to greater coverage of the system under test, and greater quality of testing. The improvement in quality is both a subtle and an important point to note.

Since test generation tools tend to blindly explore all possible scenarios described by the test model (except any restricted by supplied constraints) they can, and do, uncover unforeseen combinations of events that are not always considered by the human tester. By their nature, such cases are more likely to have been incorrectly implemented in the system under test; thereby increasing the value of the obscure tests produced by test generation tools, hence our claim of improved quality.

This paper introduces techniques for automating the generation of conformance test suites from requirements and architecture specifications based upon a model-based development process. Where, specification models are typically defined using graphical scenario-based languages, such as Message Sequence Charts

(MSCs) and UML 2.0 Sequence Diagrams. These have been implemented in the *ptk* [4] tool. In the paper we describe some of our experiences with its deployment.

1.2 Requirements-based validation

Recent studies indicate that 50 percent of test failures are caused by defects found in the requirements [15], and these defects can cost, sometimes 100 times more to fix in the development phase, than if they were fixed in the requirement or design phase [3]. Therefore, great effort is expended conducting Formal Technical Reviews (FTRs) [14], or appraisals, as a means for discovering such defects earlier within the software lifecycle. To date, however, appraisals generally rely upon manual analysis, with few (if any) tools for enabling the automated detection of defects. As a consequence early defect discovery is becoming a key focal point in reducing development costs.

This paper introduces techniques for automating the discovery of defects found within requirements and architecture specifications, typically defined today. Where, specifications are developed using Message Sequence Charts (MSCs) and UML 2.0 Sequence Diagrams. We describe some typical properties that manifest themselves as defects, discuss some of the issues relating to partial models, and some preliminary results from automating their detection, using a new tool called Mint. By using this tool we can start to consider moving testing earlier within the development lifecycle.

2 Process Overview

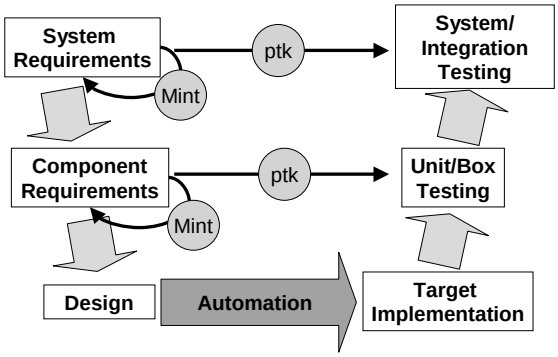

Fig. 1. Process Overview

Figure 1 typifies a development process in which requirements are firstly constructed using a combination of textual and graphical notations, sometimes

including scenario or use-case based models defined using MSCs or UML diagrams. After construction of the requirements appraisals are conducted to check their validity and to identify possible defects in the specification. When completed, requirements are then handed off to different teams who then refine and decompose requirements, pertinent to the component concerned, using a similar methodology to that used for the construction for the initial requirements. After the definition of requirements, design models are developed that are not only subject to appraisal, but are sometimes verified against the requirements using simulation techniques. Code is then automatically generated from design models, which is then executed on the target platform.

During or after this part of the process tests are manually derived from the requirements for unit, integration, and system testing. It is this manual process of generating tests that gave us our initial focus. By using the graphical models developed during these requirements as a basis for test generation we developed the *ptk* tool [4] to perform comprehensive, yet user-friendly, analysis of behaviours thereby performing a range of analyses for test planning and test suite generation. More recently, we have been evaluating the Mint tool as a means for identifying defects during the appraisal of requirements.

3 MSC/UML Sequence Diagram Semantics

It is not within the scope of this paper to give a full description of all the current constructs for MSC/UML sequence diagrams. In this section we will give an informal description for the constructs found in the examples of the paper.

Within a UML 2.0 Sequence Diagram [16] process life-lines progress non-linearly down the page. So that an event on one life-line that is visually later than an event on another life-line is not necessarily temporally later. In UML 2.0 the visual ordering can be forced by adding the `strict` keyword to the diagram header. In this paper, however, we do not consider strict diagrams.

In this paper we are concerned with technologies that are applicable to communications protocols. We therefore give UML 2.0 Sequence Diagrams the ITU MSC-2000 semantics [8]. Hence, messages are taken to be asynchronous, latency is assumed to be arbitrary and there are no queuing semantics associated with message channels. This means, for example, message overtaking is possible. A message is regarded as a pair of events, a send event and a receive event. In accordance with the ITU TTCN-2 standard [10] we define the send event for message m as $!m$ and the receive event as $?m$.

A parallel construct, denoted by `PAR`, describes a set of concurrent threads that occur in the sequence diagram. Figure 2 gives an example of a diagram containing two parallel constructs. The first parallel construct contains messages a, b and c, which can occur in any order since they are in separate threads. Concurrent threads in a parallel construct are delineated by dotted lines. An inline reference, denoted by `REF`, is a place holder for another sequence diagram. The reference can be replaced by the contents of the other sequence diagram if desired. The reference is weakly composed with the referring diagram when inlined.

Figure 2 contains an inline reference spanning processes A through D. The alternatives construct, denoted by `ALT`, denotes mutually exclusive alternatives, which are delineated by dotted horizontal lines. Figure 3 shows an alternative construct with two mutually exclusive choices. Figure 3 also shows an iterative loop, denoted by `loop`, which continues indefinitely until the events within the break construct occur. The iteration has weak compositionality semantics. This can result in the processes within a loop becoming unsynchronized, and executing different iterations at a given moment unless there is sufficient coordination.

The traces of a sequence diagram are given by constructing all possible interleavings of the events from the processes in the diagram (after inlining referenced diagrams) that are consistent with the implied temporal ordering defined by the diagram. For a precise definition see the ITU MSC-2000 standard [8].

A basic sequence diagram is a diagram whose trace semantics can be defined solely in terms of a single partial order on the events in the diagram. This partial order is known as the *causal order*. The traces of a basic diagram are precisely the set of total orders on the events in the diagram that are an extension of the causal order. Hence, for any trace T of sequence diagram S, an event e_1 can occur earlier in the trace than another event e_2 if and only if $e_1 \not> e2$, where $<$ is the causal order of S. Diagrams containing the alternative construct, for example, are not basic diagrams since each alternative requires a separate partial order to define its trace semantics. Similarly diagrams containing unbounded iteration are not basic. Examples of diagrams that are basic are any that only contain messages, internal actions, states, continuation symbols, process creation and destruction and the parallel construct. Note this categorization is not complete. For this paper we allow sequence diagrams to contain any basic diagram construct, together with loops and alternatives.

When a sequence diagram is not iterative it contains only finitely many alternative branches. Replacing each alternative construct with one of its mutually exclusive choices defines a basic sequence diagram representing one particular combination of all the alternatives. The union of the traces from these basic sequence diagrams is exactly the set of traces from the original diagram. Therefore a general non-iterative sequence diagram can be identified with a finite set of basic sequence diagrams, up to trace equivalence. In the case of unbounded iterative diagrams we can replace the diagram by an infinite sequence of non-iterative diagrams, where each finite diagram is given by unfolding each iteration a finite number of times. Therefore we can regard any sequence diagram as a (possibly infinite) set of basic sequence diagrams, up to trace equivalence. See the ITU MSC-2000 standard [8] for further details.

A consequence of alternative and loop constructs in UML 2.0 Sequence Diagrams is that the general property checking problem is undecidable for arbitrary sequence diagrams [1].

4 Related Work

Here we summarise some software tools related to Mint and *ptk*.

– MESA [11] - was originally developed by Stefan Leue et al, at the University of Waterloo, and is now maintained at the University of Freidburg. MESA provides an MSC editor, and can detect non-local choice, timing consistency, and can also generate Promela for Spin [5] process modeling. MESA is mainly a research vehicle, and is currently only available for non-commercial use.
– UBET [12] - was developed in Bell Research Labs, and Lucent Technologies. UBET provides an MSC editor, and can graphically highlight race conditions and timing violations in an MSC. The user is able to select different queuing semantics for these checks. UBET also can be used to generate test scripts in MSC and process models in Promela, the language of the Spin verification tool. UBET can also generate test cases in MSC, but *ptk* is more advanced in supporting several test generation algorithms, and several test script languages including TTCN-3.
– AutoLink [17] - is a test generation tool that must be used with Telelogic Tau [18]. 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 MSC constructs that it can handle than *ptk*, and doesn't support a variety of test strategies.

5 Requirements

Automated test generation relies on the initial requirements sequence diagrams being semantically consistent. This is equally important when requirements scenarios are used for developing architecture models. For these reasons Motorola has developed a tool, Mint, to automatically detecting pathologies in MSC and UML 2.0 Sequence Diagrams.

The current tool detects a variety of pathologies that make a sequence diagram semantically inconsistent. In a distributed environment each time-line in a sequence diagram should completely describe the expected behaviour for the associated process. It is possible within a sequence diagram to specify global behaviour that may not be a result of the concurrent local behaviour from each process. Certain kinds of global behaviour may only be achieved through implicit access to some global state that might not exist in a distributed environment. Below we list a number of ways this can occur that are detected by the Mint tool. The pathologies detected by the current Mint tool are:

– *Blocking Conditions.* These are a form of race condition. They describe a discrepancy between the message ordering specified in the requirements scenario and the order that events can occur in practice.
– *Non-local Choice.* These pathologies occur where independent processes must take non-deterministic mutually exclusive actions without sufficient coordination to guarantee exclusivity.
– *False-underspecification.* These occur where the local order for a process is weaker than the implied global order for the whole scenario. Therefore the

behaviour for an individual process can not be inferred from the specification of the process alone.
- *Non-local Ordering.* These occur when events on separate lifelines are ordered with constructs that can not force the ordering to occur in practice. For example, if a general-ordering arrow is used between events on separate lifelines.

The first two pathologies are the most serious and will be considered in more detail in the next sections. The last two are more minor and were not detected during the case study discussed in section 5.3. We will therefore not discuss these further here.

5.1 Blocking Conditions

Blocking conditions are a form of race condition. We prefer the term blocking condition because this is closer to the process behaviour when such pathological MSC specifications are implemented.

Definition 1. *Let S be a basic MSC/UML sequence diagram with causal ordering $<$, as defined by ITU MSC semantics. We define S to be block free when for every event x in S and every message n where $x \neq !n$, if $x <?n$ then $x <!n$.*

Figure 2, which is anonymized from an example in a Motorola case study, gives an example of a blocked sequence diagram. The original diagram describes traffic channel allocation and activation between various processes. Process A has delegated the task of determining what resource to allocate to process B. This example contains multiple blocking conditions.

Notice that event $?a1$ is blocked by event $!b$, for example. This is a blocking condition since $!b <?a1$, but it is not true that $!b <!a1$. We can see that in practice there can be no guarantee that the specified behaviour will occur. Without additional messages to coordinate their actions processes B and C have no mechanism to force $!b$ to occur before $?a1$. Without knowledge of B's behaviour, process C is effectively blocked from proceeding, hence our choice of name for the pathology.

In total we have the following blocking conditions. Event $?a1$ is blocked by $!b$ and by $!c$. Event $?c2$ is blocked by $!a$ and $!b$. Also event $?b2$ is blocked by $!a$ and $!c$. In total there are six blocking conditions in this diagram. Note one simple way to remove these blocks would be to regroup the messages within a single parallel construct. Messages a and $a1$ could presumably be grouped within the same thread of a parallel construct. Similarly b, $b1$, $b2$ and the inline reference could be grouped in a second thread. Finally c, $c1$ and $c2$ could be grouped in the third thread.

Definition 2. *Let S be a basic sequence diagram with causal order $<$ containing a blocked event. That is there is an event x and message n where $x <?n$ but $x \not<!n$. If there is some event y such that $y < x$ and either $y <!n$, or y lies on the same instance as $!n$ then we say the block is resolvable.*

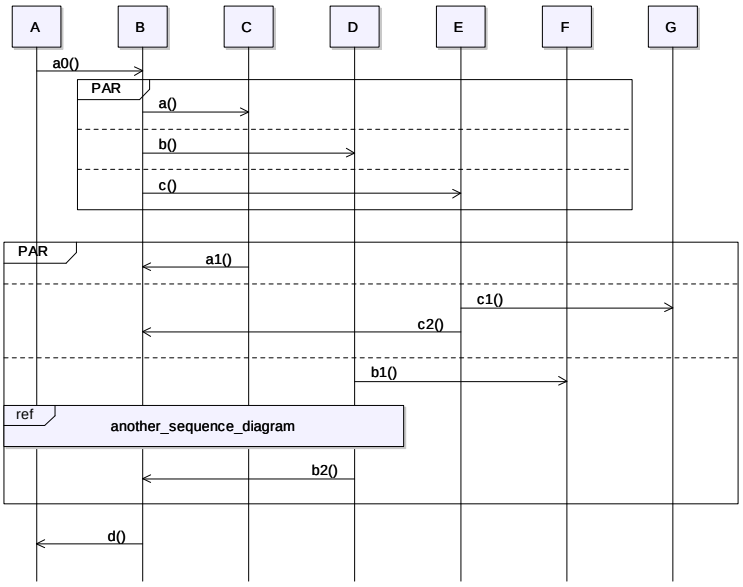

Fig. 2. Case study example with multiple resolvable blocking conditions.

The motivation for this definition is that often message latency is known at a given point in a protocol. If there is a chain of messages from a common source leading to the processes involved in a block, then this latency information could in principle be encoded in the message parameters. In which case it is now possible for the processes to coordinate their messages. In figure 2 all the blocking conditions are resolvable since $!a0$ is a common ancestor to all the other events. Given that this situation often arises in the communications domain, the Mint tool distinguishes between resolvable and irresolvable blocking conditions.

In general MSC/UML sequence diagrams do not have a simple semantics as a single partial order.

Definition 3. *Let S be a general sequence diagram. Let S be represented by the set of basic sequence diagrams* BS*. That is the set of traces for S is the union of the traces of the basic sequence diagrams in* BS*. Then S is block free if and only if every basic diagram in* BS *is block free.*

Definitions 2 and 3 are equivalent when S is a basic sequence diagram characterized by a causal order.

Unlike general property checking problems, checking for blocking conditions is decidable for arbitrary iterative sequence diagrams due to the following theorem. For a sequence diagram S, let $S(2)$ be a sequence diagram that describes the behaviour of S restricted so that each unbounded loop is only permitted to iterate at most twice. It is possible to construct $S(2)$ algorithmically. Although $S(2)$ only contains initial behaviour from S it contains exactly enough behaviour to preserve any blocking conditions that were in S. Note $S(2)$ is finite.

Theorem 1. *Let S be a general sequence diagram that may contain loop constructs. Then S is block free if and only if $S(2)$ is block free.*

We do not have space to include the proof of theorem 1 here, but it follows from results in [1] and observing a certain link between their bounding conditions and the definitions for blocking above.

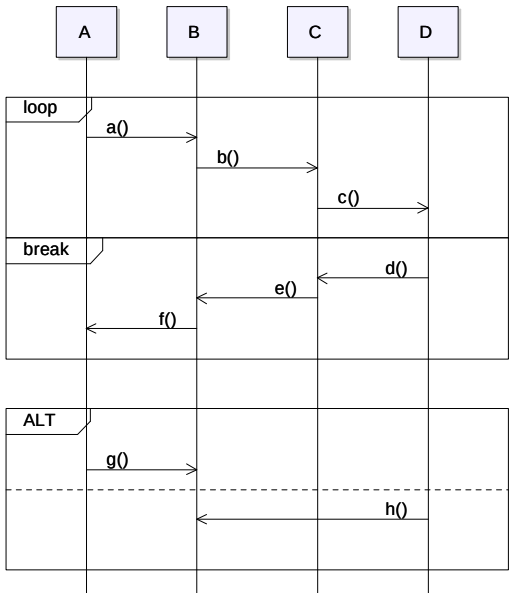

Fig. 3. Case study non-local choice example contained in loop

Hence by theorem 1, for the purposes of detecting blocking conditions, we may assume a sequence diagram does not contain any infinite traces. A finite sequence diagram S can be represented by a finite set of basic sequence diagrams S_i for $1 \leq i \leq n$, as outlined in section 3. Diagram S is block free if and only if each of the S_i are block free. Hence we may apply definition 1 to the causal order for each S_i in turn to decide if S is block free.

Figure 3 contains multiple examples of blocking conditions caused by a loop. As mentioned in section 3, processes in loops are not forced to synchronize at the end of each iteration. For example, in this case the diagram does not impose any synchronization between A and B. The diagram specifies that A must wait to send each a event until B has sent the b message from the previous iteration. However, there is no coordination mechanism given in the diagram to force this to occur, which is the cause of one of the blocking conditions. Similarly there are blocking conditions between B and C, and C and D.

The Mint tool constructs the causal orders for the set of basic sequence diagrams that represent a general diagram. Each of these is analyzed for blocking

conditions and suitable reports are then generated. The Mint tool reports resolvable blocking conditions as a warning, but irresolvable conditions as errors.

5.2 Non-local Choice

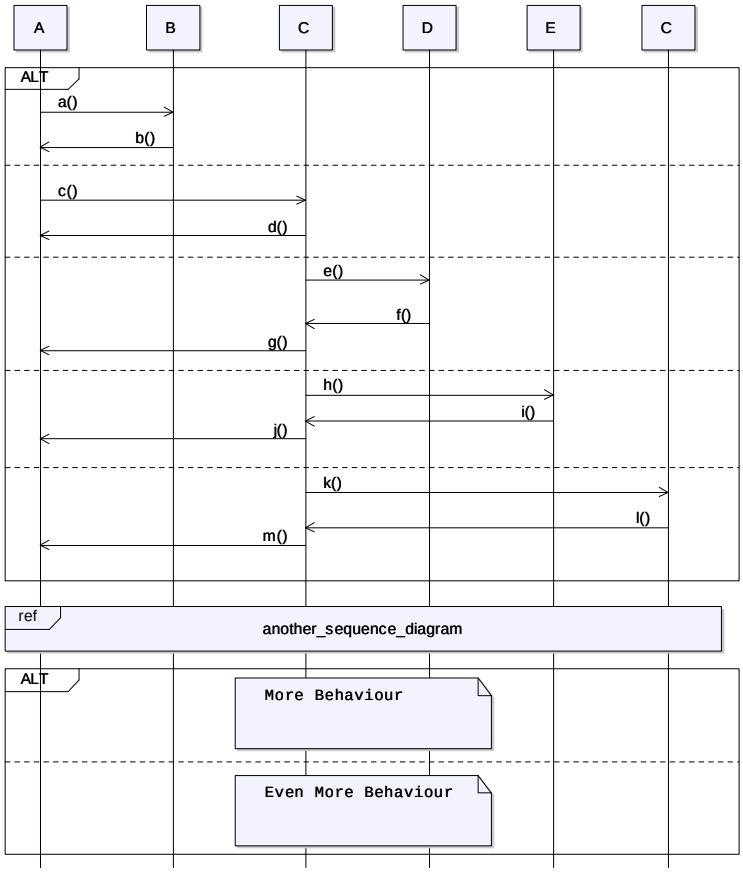

Fig. 4. Case study example with multiple non-local choice

An event is *active* if the process that the event belongs to can choose when that event occurs. A non-local choice occurs in a sequence diagram when there is a non-deterministic choice of mutually exclusive active events that belong to separate processes. This implies there is some global mechanism for ensuring only one of the processes can execute. Figure 3 is an anonymized version of a diagram from an internal case study within Motorola. In figure 3 there is a simple non-local choice between event $!g$ and $!h$. However there is a more subtle non-local choice between $!a$ and $!d$. The $!d$ event should terminate the loop, but there is

nothing to stop A repeatedly sending event a until f is received. Therefore, due to latency, $!a$ may occur multiple times before $?f$ occurs, leading to potential deadlock.

The original diagram defined how process A can buffer excess data with process D when A's own buffer is full. The loop breaks when D's buffer also reaches capacity. At some later stage D again has spare capacity in it's buffer and signals A that this is the case.

Definition 4. *Active events x and y cause a non-local choice if there is a trace prefix t such that:*

1. *x and y lie on distinct instances;*
2. *tx and ty are both valid trace prefixes;*
3. *txy is NOT a valid trace prefix.*

Note according to this definition even with guards placed at the appropriate places in figure 3 non-local choice conditions would still arise. This time however the non-local choices would be between the guard conditions. In practice it is not generally possible to know whether such guards can be guaranteed to be mutually exclusive since the conditions are essentially arbitrary. The current Mint tool reports non-local choice conditions as errors, except when they occur between guards. In that case they are reported as warnings.

Figure 4 is another example showing multiple non-local choice conditions. The original diagram described various cases of exceptional behaviour and how the system should fail gracefully in these cases. In the original diagram A is attempting to initialize various other processes, which may fail. The original diagram described how these failures can occur and how to mitigate them. Which alternative is chosen in practice is determined by global meta-conditions concerning the configuration of the whole system. In figure 4 there are non-local choices between process A and process B. A initiates the first two alternative, whereas in all the others process B initiates the alternatives.

5.3 Mint Evaluation

The Mint tool is new and still undergoing evaluation with a number of Motorola engineering groups. Recently it has been applied in a case study of UML 2.0 sequence diagram specifications for part of a communications protocol stack. Mint was applied to approximately a hundred and fifty sequence diagrams and detected numerous pathologies. After filtering trivial pathologies those remaining fell into the following categories:

- 6 diagrams containing multiple non-local choice conditions
- 2 diagrams containing multiple non-local choice conditions between break and loop constructs
- 5 diagrams containing multiple resolvable blocking conditions caused by loop constructs
- 1 diagram containing an irresolvable blocking condition

– 1 diagram containing multiple resolvable blocking conditions between different parallel constructs

Figures 2, 3 and 4 are anonymized versions of examples taken from those listed above. Other groups currently using Mint on protocol stack design have reported similar findings, and subsequently modified their specifications to remove the defects. In summary Mint appears to be successful at detecting semantic inconsistencies in industrial size case studies, and has been found valuable by groups who have used it.

6 Test Generation

In this Section we give a brief overview, and some experiences of test generation in Motorola, with the *ptk* tool. More details on *ptk* are available from [4].

Over the last seven years, since circa 1997, our group has developed, *ptk*, which derives conformance test suites from MSC specifications. This has proved to be a popular tool, used by several groups within Motorola for functional testing parts of standard telecom protocols such as TETRA, GSM, and CDMA. On average, *ptk* user's have reported a 33 percent reduction in effort compared to writing tests manually, and they also report a greater quality of tests. These figures account for *ptk*'s popularity with an increased emphasis on reducing the costs and improving quality.

ptk takes as input a collection of MSC specifications along with data specifications called Protocol Data Units (PDUs), and information specific to test generation such as what part of the MSC represents the Implementation Under Test (IUT). There is support for almost all of the MSC-2000 language, as well as support for several data languages. Originally, *ptk* generated test suites in SDL [9] for TETRA. Since then, *ptk* has been enhanced to generate test suites for some Motorola proprietary languages, and also standard test languages such as TTCN-2 [10], and now TTCN-3 [5]. In addition to generating test scripts in these languages, there is also support for analysing the MSC and test semantics, and static checks on the data. Data may be placed on the MSC in message parameters, etc, and in separate PDU files. Data languages supported include TTCN-2, TTCN-3, and some proprietary languages. Syntax checking and consistency checks, such as checking if declarations are given is performed on the data.

6.1 Black box testing

ptk derives black-box tests, where some MSC instances are labelled as the black-box or Implementation Under Test (IUT), and the remaining instances are considered as the test system. This is illustrated in Figure 5 where the IUT is given as instance B. Information such as the IUT and other information relevant to test generation may be placed inside a textbox as in Figure 5, or in a separate file. The test or tests that are derived will contain all message events that interact with the IUT (such as receive x and y, but not events on the IUT (such as

send x and y), or events that don't interact at all with the IUT (such as message z).

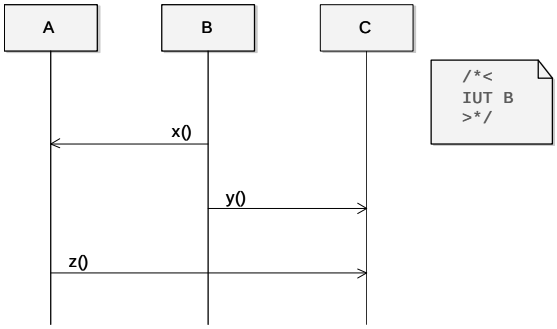

Fig. 5. Test generation example.

6.2 Local vs. Remote testing

There is a choice in black-box testing between generating tests from the perspective of the IUT (called *local testing*), and from the perspective of the test system (called *remote testing*). With remote testing the order of events is consistent with that of the MSC semantics for the whole MSC. The *local* testing approach is to consider the order of events inside the IUT itself, and derive tests based on this order. Local testing can generate the same tests, or fewer, or more tests depending upon the example. With local testing it is possible to generate an order of events that is not consistent with that in the MSC semantics, it is as if the test system is representing the possible message latencies. See [4], for more details about local and remote testing.

6.3 Test strategies

Three test strategies are possible with *ptk*, as well as generating tests for a single processor, or for parallel architectures. The three strategies are:

- *Trace testing* — where each single trace though the MSC is placed in a separate test. This is the simplest strategy and typically generates the most test scripts. One advantage of this strategy is that we can enforce full trace coverage, by repeated running each test until they pass. Valid behaviour that isn't represented in a test is given an inconclusive verdict with this test strategy.
- *Branch testing* — where choices between receive events are placed in a single test script. This typically produces fewer tests than trace testing, since multiple traces can be contained in one test script.

– *Completion testing* — similar to branch testing, but in addition choices from alternatives are also placed into a single test where possible. This can again produce fewer tests than branch testing.

In addition to the test strategies various optimisations are done on the test scripts, where common parts are factored out, reducing the size of tests considerably.

6.4 User experience

ptk has evolved over several year, based mainly on user's requirements. At any one time there are perhaps five test teams using *ptk*. The number of MSCs processed by each team is in the thousands, with hundreds of generated tests. As stated before, the average reduction in effort is 33 percent compared to the manual approach.

One of the reasons for *ptk*'s popularity stems from groups liking to use graphical notations for developing tests, rather than notations like TTCN-2 which can be tedious to write by hand. Typically MSC and *ptk* is used as a graphical test scripting language, rather than as a requirement specification language, which was our original intention. Many of the enhancements have been requested due to its use for test scripting, some of which have extending the MSC language. For example, atomic references which are MSC references, without any interleaving of events inside the reference with events outside the reference. Atomic references translate to procedure calls in test languages. Other extensions have included atomic loops, and special primitives for time constraints.

7 Conclusion

During deployment of requirements-based test generation technology we found that requirement and architecture teams were reluctant to take on board technology that would not directly benefit them in constructing requirements. Hence, ptk was deployed within testing teams who would work the requirements into a more rigorous form so they could be used for test generation, with good results. However, we found in some cases that ptk was used as a graphical scripting tool, meaning that the full potential of this technology was not being realised from the perspective of the overall process. So, we altered our strategy to provide technologies, such as requirements validation and the detection of feature interactions [13], which would give requirement and architecture team's incentives for constructing more rigorous models, through reduce appraisal costs, which would then enable test generation. Pursuing this approach we found that requirements are often incomplete, or *partial*, meaning that only a subset of all possible required system behaviours is specified. This partiality has implications on the type of analysis that can be conducted on requirements - see section 5. Hence, we developed Mint as a tool for automating discovering pathologies, or potential defects, during appraisals. We are now evaluating Mint and looking to expand this capability.

References

1. R. Alur and M. Yannakakis, *Model checking of Message Sequence Charts*, Proceedings of the Tenth International Conference on Concurrency Theory, Springer Verlag, 1999
2. Beizer, B., *Software Testing Techniques*, New York, New York: Van Nostrand Reinhold, 1983.
3. Boehm, B., Basili, V.R., *Software Defect Reduction Top 10 List*, IEEE Computer, Vol. 34, No. 1, 2001.
4. P. Baker, P. Bristow, C. Jervis, D. King, B. Mitchell, *Automatic Generation of Conformance Tests From Message Sequence Charts*, Telecommunications and Beyond: The Broader Applicability of MSC and SDL, pp 170-198, LNCS 2599.
5. ETSI ES 201 873-1, *Methods for Testing and Specification; The Testing and Control Notation version 3 (TTCN-3); Part 1: TTCN-3 Core Language*, European Telecommunications Standards Institute (ETSI), 2001.
6. Ghiassi, M., K.I.S. Woldman, *Dual Programming Approach to Software Testing*, Software Quality Journal, 3:45-58, 1994.
7. Hartman, Alan, *AGEDIS Model-Based Test Generation Tool*, `http://www.agedis.de`.
8. International Telecommunications Union: ITU-T Recommendation Z.120, *Message Sequence Chart (MSC)*, 2000. Available from `http://www.itu.int`.
9. International Telecommunications Union: ITU-T Recommendation Z.100, *Specifications and Description Language (SDL)*, 2000. Available from `http://www.itu.int`.
10. International Telecommunications Union: ITU-T Recommendation Z.100 X.292, *TTCN-2 standard, Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation (TTCN)*, 1997.
11. Leue, Stefan, *MESA: MSC Editor Simulator Analyzer*. Available from: `http://tele.informatik.uni-freiburg.de/Mesa/Mesa_doc/index.html`.
12. Lucent Technologies. *UBET documentation*. Information is available from `http://cm.bell-labs.com/cm/cs/what/ubet/index`.
13. Bill Mitchell, Robert Thomson, Clive Jervis, *Phase Automaton for Requirements Scenarios*, Feature Interactions in Telecommunications and Software Systems VII, 77-84, 2003, IOS Press.
14. NASA, *Formal inspection standard - NASA-STD-2202-93*, `http://satc.gsfc.nasa.gov/fi/std/fistdtxt.txt`.
15. Nelson, Clark, and Spurlock. *Curing the Software Requirements And Cost Estimating Blues*, PM: Nov-Dec, 1999.
16. Object Management Group (OMG), *Unified Modeling Language (UML): Superstructure, Version 2.0*, 2003. Available from `http://www.omg.org`.
17. Telelogic, *AutoLink documentation*, Telelogic Web Site: `http://www.telelogic.com`.
18. Telelogic, *Tau documentation*, Telelogic Web Site: `http://www.telelogic.com`.