SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:999–1027 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Validating feature-based specifications

Kenneth J. Turner[†]

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK

SUMMARY

It is argued that specifications should be rigorously validated against requirements. This is useful to build confidence in a specification, and to check a specification after it or the requirements have changed. MUSTARD (Multiple-Use Scenario Test and Refusal Description) is introduced as a means of formulating and formalising validation scenarios. The practical use of MUSTARD on a number of case studies is introduced. The MUSTARD notation is then explained, using examples from Internet Telephony to show how validation scenarios are written. The core MUSTARD constructs are augmented by domain-specific vocabularies that adapt it for different kinds of system. It is seen how MUSTARD can validate specifications written in two standardised formal languages: LOTOS (Language Of Temporal Ordering Specification) and SDL (Specification and Description Language).

KEYWORDS: Feature, LOTOS (Language Of Temporal Ordering Specification), SDL (Specification and Description Language), Service, Validation

1. Introduction

1.1. Scenario-Based Validation

Testing is standard practice in development. In fields such as hardware design, software engineering and protocol development, there are well-defined techniques for testing implementations. In an ideal world, testing would be unnecessary. Specifications would initially be proven correct. An implementation would be rigorously derived from its specification. The implementation would run in a hardware/software environment that had also been proven correct. However, current practice is very far removed from this ideal. Testing is therefore likely to remain an essential activity.

Most research on testing has focused on checking whether an implementation conforms to its specification. However, it is easily forgotten that the specification has to be checked too. Like

[†]E-mail: kjt@cs.stir.ac.uk

implementations, specifications are prone to bugs and evolve over time. It is therefore important to have systematic techniques to evaluate specifications as well. A specification plays an important role in software development as it affects all later stages of the work. Ideally a specification should be precise, i.e. specified in some formal language.

In principle, the accuracy of a specification can be assessed by some form of verification technique such as equivalence checking, model checking or theorem proving. In practice, this is often infeasible due to the complexity of the specification and the size of its state space. The state space can be reduced by imposing limits on data values. However, combinatorial effects mean that a small change in data limits can dramatically affect the size of the state space [38].

A much more practical approach is the one adopted in this paper. Specifications can instead be validated (i.e. tested), checking that they have desirable properties. Since most specifications are behavioural, the properties to be checked can be formulated as scenarios. These view the specified system as a black box, and so might be termed use-case scenarios. The approach is designed for reactive systems that can be characterised by their input-output behaviour. The specification should be executable (or animatable) in the sense that its behaviour can be explored automatically. Ideally the specification should be in some formal language. However the technique described in this paper is also applicable to lower-level, less formal specifications such as might be written in a programming language.

Validating a specification gives no guarantee of correctness. However a well-chosen set of scenarios will exercise the critical functionality of a specification. Scenario-based validation can thus build confidence in a specification. In effect, equivalence classes of behaviour are identified and captured in scenarios. A limited number of scenarios can therefore check a significant portion of a specification.

A system design method (e.g. UML) may well require use-case scenarios to be defined. Scenarios for validating the system specification are then immediately available. Although the design method may not require them, it is good practice to define scenarios even if they are only for validating the specification. Such scenarios are high-level and abstract, so they present a reasonably independent perspective on what the specification should do. Ideally, scenarios should be written by someone other than the specifier.

Most obviously, scenarios can be used to validate the specification once it is written. However requirements are likely to change, resulting in new versions of the specification. Scenarios can then be used for the equivalent of regression testing in system development: checking that the new specification includes its original behaviour.

1.2. Feature-Based Validation

Many classes of system are feature-based. Such systems have a core behaviour that is extended by additional features. This situation is well known in telephony, where the basic telephone call is enhanced by features such as call diversion, credit card calling or automatic call-back. Typically, the core behaviour and the features are deployed by the service provider. End users then individually select the features they wish. Some features (e.g. call diversion) may be parameterised by the user (i.e. the forwarding number).

Feature-based systems are not restricted to traditional telephony. For example, Internet Telephony and Interactive Voice Response can also be regarded as feature-based [35]. Features are also common in non-voice systems, for example:

- a word processor may be configured for additional languages, import/export of other formats, and specialised graphics
- a web browser usually accepts plug-ins for new kinds of data streams
- a computer may be extended with additional drives, external ports or devices
- a car may be ordered with extra capabilities such as air conditioning, anti-lock braking or security.

The well-known feature interaction problem arises because it is usually not possible to mix features arbitrarily [7]. In practice, developers spend a lot of time to discover and eliminate interference among features. However this is a tedious and expensive task. A PBX (Private Branch Exchange) may easily have 500 features. Checking pairwise compatibility of these requires more than 100,000 feature combinations to be evaluated.

Scenario-based validation can therefore be usefully extended to evaluating feature-based systems. For modularity, each feature should be associated with its own set of scenarios. The validation technique should also be feature-based, allowing features to be checked in isolation and in combination. A common view of feature interaction is that it occurs when a feature behaves differently in the presence of other features. In a validation setting, this means that a feature fails to satisfy its scenarios when other features are deployed.

1.3. Requirements for A Scenario Notation

The author has developed a scenario-based and feature-based approach for validating specifications. The notation is called MUSTARD (Multiple-Use Scenario Test and Refusal Description). The name was chosen as the culinary counterpart to CRESS (Chisel Representation Employing Systematic Specification [34]). CRESS describes services graphically, translating their pictorial representation into specification languages or implementation languages. Although MUSTARD was designed for validation of CRESS specifications, the approach is generic and can be used independently of CRESS.

MUSTARD checks that the behaviour defined by each scenario is respected by the specification. A scenario may have multiple paths, while the specification almost certainly has many paths; all paths are fully explored during validation. If a scenario path is respected by the specification, the path is said to lead to success. If a scenario path is violated at some point, it is considered to fail; no further behaviour along that path is investigated. If all paths lead to success, a pass verdict is returned by the scenario. If no paths lead to success, a fail verdict is returned. If some paths succeed and some fail, the result is an inconclusive verdict. This interpretation agrees with most work on testing (e.g. [13]). However, some authors (e.g. [19]) use an inconclusive verdict to mean that the observed behaviour was correct but the test purpose could not be achieved.

MUSTARD meets the following general objectives for a scenario notation:

Simplicity The notation should be simple and compact, but expressive enough for realistic scenarios.

Black Box The approach should treat the system abstractly as a black box. Knowledge of the system internals may be used to devise appropriate scenarios, but only the input-output behaviour of the system may be validated.

Validation The notation should describe high-level behavioural scenarios that can be automatically validated against the system specification in whatever language.

Features Where appropriate, the notation should recognise the existence of features in a system. It should be possible to validate features individually and together. Two levels of feature deployment should be recognised: generic provision of a feature within a system, and selection (plus parameterisation) of a feature by an end user.

Application Independence It should be possible to write validation scenarios for a wide variety of application domains. In this paper, voice services are used as examples. However the approach is extensible for other kinds of application.

Language Independence The notation should be independent of the language used to express a specification. In fact, specifications could be written in some low-level language (such as a programming language). The approach can therefore be used to validate implementations as well as high-level specifications.

Tool Independence The notation should be independent of how validation is actually performed. As will be seen, the scenario notation can be automatically translated into a variety of underlying languages for use with various tools.

MUSTARD also meets the following language objectives:

Outcomes Three outcomes should be recognised for scenario behaviour: failure (termination of a scenario path due to unexpected specification behaviour), exit (normal termination enabling further behaviour), and success (completion of a scenario path).

Input-Output The language should recognise system input and output as the fundamental actions. Scenarios should be built from these using a variety of combinators. Input-output may be synchronous or asynchronous, depending on the specification language.

Extensibility The input-output vocabulary used in scenarios should not be part of the core language. Rather, it should be easy to add other kinds of signals and parameters for new application domains.

Sequencing Sequential scenarios should be straightforward to write.

Concurrency It should also be possible to define concurrent scenarios with a number of parallel test behaviours. Parallelism may be through true concurrency or through concurrency by interleaving, depending on the specification language.

Alternatives Scenarios should be allowed to have alternative branches. Choices in a scenario may be resolved by the scenario itself (non-deterministic) or by the system (deterministic). Choices that depend on feature characteristics should also be permitted.

Looping Explicit loops should be discouraged as they may lead to unbounded execution of a scenario. However implicit loops should be allowed in a controlled fashion.

1.4. Related Work

Implementation validation has been thoroughly researched. In the hardware field, design verification is the term used for checking of design flaws as opposed to manufacturing defects (e.g. [39]). Software validation techniques include black/white-box testing, alpha/beta testing, module/system testing and integration/regression testing (e.g. [30]).

In the protocols field, an elaborate methodology has been developed for conformance testing [13]. This is associated with TTCN: originally the Tree and Tabular Combined Notation [14], now termed the Testing and Test Control Notation [10]. TTCN is aimed at conformance testing of an implementation with respect to its specification. TTCN is therefore implementation-oriented whereas MUSTARD is specification-oriented: it assesses a specification with respect to its requirements (formulated as scenarios). TTCN is mainly used to test an implementation externally and remotely, while MUSTARD is used for integrated validation of a specification. Both approaches have similar technical capabilities, though TTCN is a much larger language. TTCN is imperative in style, while MUSTARD is applicative (functional) in style. TTCN has a bias towards networked systems, while MUSTARD is generic. MUSTARD supports non-deterministic tests. However it lacks the timer facilities found in TTCN. This reflects their different focus: real-time behaviour is of much greater importance when checking implementations rather than specifications.

A specific role has been identified for the use of formal methods in conformance testing [12]. Formally-based theories have developed for testing (e.g. [4, 25, 31]). Tools for systematic generation of implementation tests from a specification include TestGen [20] and TGV [19].

This paper reports work based on standardised formal languages that have seen widespread use on industrially relevant applications. LOTOS (Language Of Temporal Ordering Specification [11]) is a process algebra with algebraically defined data types. SDL (Specification and Description Language [18]) is based on communicating extended finite state machines. SDL data types may be defined in various ways, including operationally. An MSC (Message Sequence Chart [16]) is a diagrammatic way of showing signal exchanges among the components of a system. MSCs can be used to specify system behaviour at the level of signal exchange, but are often used to validate SDL specifications. All three languages were conceived for use with communications systems, but have been used more widely as general specification languages. The MUSTARD approach is not limited to these languages; indeed it is not limited to formal languages.

Using MSCs to validate SDL is the prime example of a systematic approach to validating specifications. In principle, MSCs could be used to validate specifications in any language. However, in practice it is normal to find that MSCs and SDL are tightly linked. In addition, MSCs are rather generic in terms of application domain and lack some useful capabilities for writing scenarios. It will be seen that MUSTARD scenarios are more compact than MSCs, but can be translated into MSCs.

Apart from MSCs, there appears to have been little work on systematic validation of specifications. This is partly because most research has been on verification, and partly because most validation appears to be *ad hoc*. A specifier will typically use symbolic execution to validate a specification, or will write a one-off 'test harnesses' to check complex behaviour. Even where a rigorous approach is used, it is usually limited to one particular specification language. The aim of MUSTARD is to make validation independent of the specification language.

A few researchers have identified the need for change control in specifications. [22] investigates how revised requirements can be reflected in the changes needed to a formal specification. [8] discusses how a Z specification can be incrementally evolved. [5] takes the view that an updated specification should be formally compared with the original. For realistic specifications, a formal (equivalence checking) comparison is likely to be impracticable. The philosophy of MUSTARD is that the changed specification should be checked against high-level requirements in the form of scenarios. [23] proposes extensions to SDL that make it more suitable as a testing language. This is contrary to the MUSTARD approach, which insists on validation being language-independent instead of changing the specification language for testing purposes.

Feature-based validation is used to some extent by the community working on telephony services (e.g. [9, 21]). The focus of that work is on detecting feature interactions rather than on validating specifications in general. The author previously developed the ANTEST notation for validating feature specifications generated by ANISE (Architectural Notions in Service Engineering [33]). MUSTARD is a descendant of ANTEST, extended for validating any kind of specification, and generalised for a variety of application domains and specification languages.

1.5. Overview of Paper

Section 2 discusses how MUSTARD has been used in practice, including tool support for the approach. Section 3 gives a technical overview of the MUSTARD scenario notation. This is illustrated with examples chosen from an Internet Telephony case study. Section 4 explains how MUSTARD can be given meaning through translation to LOTOS. Standard tools are able to perform automatic validation of MUSTARD scenarios against a LOTOS specification. Section 5 explains how MUSTARD can be given meaning through translation to MSCs. Standard tools are able to perform automatic validation of MUSTARD scenarios against an SDL specification. Section 6 addresses practical considerations such as how to select and evaluate scenarios, and how to integrate MUSTARD with other testing practices.

2. Practical Application of MUSTARD

This section discusses the applicability of MUSTARD and how it has been used in practice. It will be seen how tool support permits the automated validation of specifications, and aids the diagnosis of faults.

2.1. Applicability of MUSTARD

MUSTARD makes minimal assumptions about the system to be tested and the specification language used. The application should fall into the broad category of reactive system, i.e. one that produces outputs in response to input stimuli. The specification language should be sufficiently expressive to allow MUSTARD scenarios to be translated into the language (or a closely associated one). The language should also be supported by tools that allow the behaviour of a specification to be explored automatically using a MUSTARD scenario as the environment of the system.

MUSTARD has been extensively used to validate specifications generated by CRESS. Descriptions in CRESS are automatically translated into various languages including LOTOS and SDL. Although MUSTARD can exploit certain characteristics of specifications generated by CRESS, it is not dependent on CRESS specifications. MUSTARD can work with virtually any specification in the chosen language.

General information is automatically extracted from the specification, such as the specification name or its input-output ports.

For use with feature-based specifications, MUSTARD requires some additional information to be embedded in the specification. The features incorporated in the specification need to be stated in a comment such as:

features: PROXY PROXY_CFBL PROXY_TCS

The CRESS tools respect this format, but the information can easily be added manually if the specification is created in some other way. The presence of features is made available to scenarios. Each feature is associated with a scenario file, e.g. *proxy.mst* contains the MUSTARD scenarios for the PROXY feature. If a specification does not identify features, MUSTARD expects to find scenarios defined in the file *specification.mst*. A MUSTARD file typically contains a number of scenarios.

Feature selections may be incorporated as user profiles in a specification. The format of this information is language-dependent and application-dependent. For Internet Telephony, suppose that the user at address 3 forwards calls on busy to the user at address 5. This is specified in a format like:

LOTOS:	Status(ForwardBusy,3,5)
SDL:	Status((. ForwardBusy,3 .)) := 5

The CRESS tools respect these formats. Specifications in another language or in another domain would require a small extension to the MUSTARD tools to recognise the different format. Profile information is made available to scenarios in a language-independent manner.

2.2. Application Domains

MUSTARD has so far been used with specifications in five application domains:

Intelligent Networks The IN [15] offers a flexible approach to the provision of telephony services. MUSTARD has validated LOTOS and SDL specifications of around a dozen IN features [34].

Internet Telephony Two popular standards for Internet Telephony are SIP (Session Initiation Protocol [27]) and H.323 (Packet-Based Multimedia Communication Systems [17]). The approach of SIP is discussed in this paper, though H.323 is broadly equivalent. MUSTARD has been used to validate LOTOS and SDL specifications of call features deployed in SIP user agents and proxy servers [35]. Typically these features resemble their IN counterparts.

Interactive Voice Response IVR allows a system to respond flexibly to spoken commands. A widely adopted standard is VoiceXML [40]. MUSTARD has validated LOTOS and SDL specifications of around a dozen IVR features [36].

Web Services Web services provide uniform access to applications running on distributed systems. MUSTARD has been used to validate compositions of web services using BPEL (Business Process Execution Logic [1, 2]) – so-called web service choreography. In this application, MUSTARD scenarios are translated into BPEL. This is more like a conventional implementation language than LOTOS or SDL.

Radiotherapy Accelerators MUSTARD has been used to validate the control system of a radiotherapy accelerator [37]. This is a safety-critical system used in the treatment of cancer. The accelerator specification is written in LOTOS, but was developed manually rather than through use of CRESS.

All these applications are realistic, detailed and industrially relevant. The first four above require specifications of around 2,000 lines, while the fifth is around 1,000 lines. The five case studies were driven by external requirements; for example, the IN specifications were provided by BellCore (now Telcordia). The case studies employed widely-used standard languages: BPEL, LOTOS, SDL, WSDL (Web Services Description Language). MUSTARD has therefore been demonstrated on real-world applications rather than toy problems.

Internet Telephony is used in this paper to illustrate MUSTARD. A brief overview of this area is therefore appropriate. Information about SIP can be found via *www.cs.columbia.edu/sip*. Figure 1 shows the key elements of Internet Telephony with SIP. A user agent is a device or program that provides Internet Telephony services to the end user. It might be a telephone-like device or a softphone. User agents can communicate directly with each other, but more commonly go through a server. A proxy server sets up calls on behalf of users. A redirect server is used to locate users whose specific address is unknown or has changed. SIP services often resemble those in traditional telephony, such as call forwarding and call screening. They may be deployed in user agents or proxy servers.

As suggested by figure 1, the Internet Telephony user does not see the underlying protocol (and in fact it could be H.323). The terminology used for the user interface intentionally resembles that for conventional telephony, even though a standard telephone is not used. In place of signals like busy tone, Internet Telephony carries announcement messages such as 'busy here' (the called user is busy) and 'trying' (the system is trying to connect the call). Instead of answering, the callee may reject the incoming call for a reason such as 'decline' (the call cannot be accepted).

2.3. Automated Validation

The MUSTARD tool is written in Perl for portability; MUSTARD has been used without changes on three operating systems. The top-level tool coordinates the translation, execution and diagnosis of validation scenarios. Translation is performed using the m4 macro processor [29]: one collection of m4 macros exists for each target specification language. MUSTARD is invoked by a simple command line call such as 'mustard SIPSystem'. Various tool options allow manual rather than automatic validation of scenarios, and define optional parameters such as the search characteristics and the application vocabulary. The result of calling MUSTARD should be a series of pass verdicts for the validation scenarios. For example, validating a SIP user agent yields the following. This extract shows tests of features AGENT CFBL (Call Forward Busy Line) and AGENT TCS (Terminating Call Screening).

```
Test AGENT CFBL Busy Forward ...    Pass     1 success   0 fail   0.5 secs
Test AGENT CFBL No Forward ...      Pass     1 success   0 fail   0.4 secs
Test AGENT CFBL Free ...            Pass     1 success   0 fail   0.5 secs
Test AGENT TCS Reject ...           Pass     1 success   0 fail   0.4 secs
Test AGENT TCS No Reject ...        Pass     1 success   0 fail   0.4 secs
Test AGENT TCS No Screen ...        Pass     1 success   0 fail   0.4 secs
```

It is possible for a scenario to succeed or fail in a number of ways if the specification has alternative branches, non-determinism or concurrency. MUSTARD computes the number of successes and failures

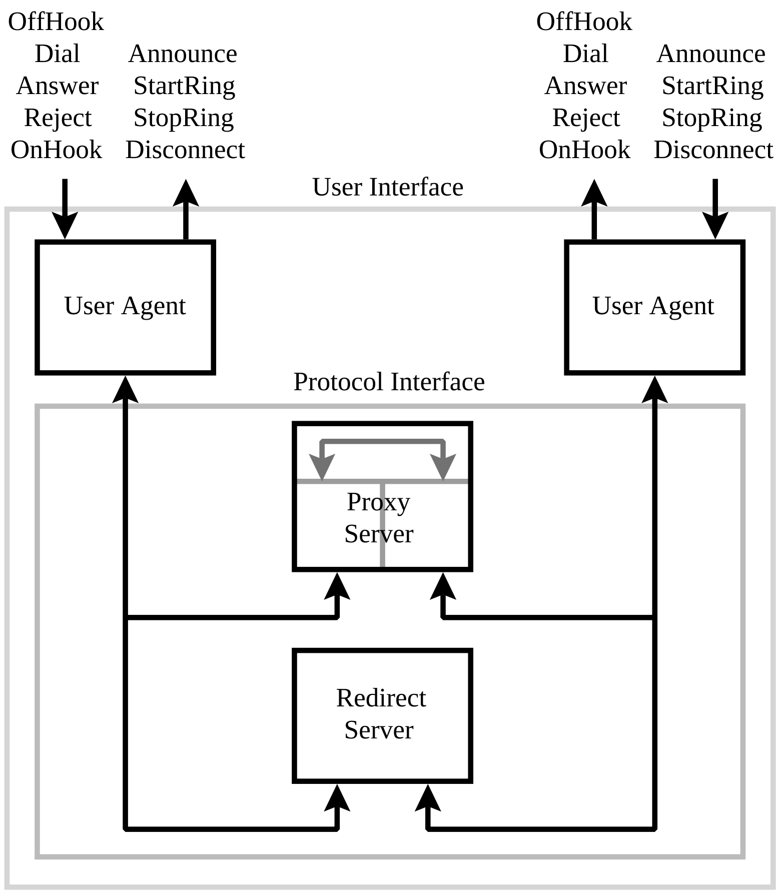

Figure 1. Elements of Internet Telephony with SIP

for each test. A verdict is then reported: pass (only successes), inconclusive (some successes and some failures) or fail (only failures).

For each failure, MUSTARD displays traces that lead up to the failure point. The aim of MUSTARD is to be language-independent. Traces are therefore shown in MUSTARD form rather than in specification language form. Suppose that the specification incorrectly responds with ‘unobtainable’ instead of ‘decline’ when a call is blocked by the callee. The scenario that validates the SDL specification of a user agent will report the following:

```
Test AGENT TCS Reject ...          Fail        0 success    1 fail   0.4 secs

send(OffHook,1)
read(Announce,1,DialTone)
```

```
send(Dial,1,2)
read(Announce,1,Unobtainable)
<unexpected signal>
```

This means that the 'unobtainable' signal was unexpected. In general, a number of traces leading to failure may be given. For validation of a LOTOS specification with the same fault, the report is similar:

```
Test AGENT TCS Reject ...          Fail          0 success     8 fail   0.7 secs

send(OffHook,1)
<internal event>
send(Announce,1,DialTone)
send(Dial,1,2)
    <internal event>
    <failure point>
```

Due to internal concurrency and non-determinism, more paths lead to failure in the LOTOS case. In general, failure is reported as a tree of behaviour (abbreviated above to its first branch). The LOTOS validator reports failure when an incorrect signal is offered by the system, not failure after issuing this as with the SDL validator. Internal events are also shown in the LOTOS trace. The differences in these validation reports are, however, small – both pinpoint where the problem occurs. This can be investigated through, for example, manual simulation using the failure traces. MUSTARD logs all test results to a file. This can be used as proof of successful validation, or to study problems further.

2.4. Feature Validation

Apart from validating features in isolation, MUSTARD can be used to validate features in the presence of other features. This is useful to determine if the features interfere with other other. For example, feature interactions in the IN [33] and IVR [36] have been investigated in this way. Interactions are usually undesirable, but can sometimes be beneficial. A feature interaction may therefore not indicate a problem: this can be determined only by a domain specialist.

There is a significant difference between this approach and other techniques to detect feature interactions. Many researchers assume that it is sufficient to compare features pairwise for compatibility. In fact, there is debate in the feature interaction community about whether to check for interference among more than two features [6]. In the MUSTARD approach, features can be evaluated in the presence of as many other features as desired. The time to validate a feature's scenarios is similar, irrespective of whether one or many other features are present in the specification [33]. The reason for this is that features are usually designed to be independent; the extra specification due to additional features has only limited effect on the cost of validating a given feature.

The main factor affecting validation time is the degree of concurrency in the specification. When dealing with voice services, this largely depends on the number of concurrent users of the system. Validation time with MUSTARD varies roughly as the square of the number of users. The approach thus scales satisfactorily, both in terms of the number of features and in terms of the number of users. Since validation scenarios are associated with each feature individually, scenarios for a large number of features can be readily managed.

MUSTARD	Meaning
% text	explanatory comment
call(*feature*,*scenario*)	invokes behaviour of another scenario; the feature name is optional, defaulting to the current feature
decide(*behaviour*)	non-deterministic (scenario-decided) choice of alternative behaviours
depend(*condition*,*behaviour*,...)	behaviour depends on whether the condition holds; an optional final behaviour acts as an 'otherwise' case
exit(*behaviour*)	sequential behaviour with normal termination
interleave(*behaviour*)	concurrent execution of behaviours
offer(*behaviour*)	deterministic (system-decided) choice of alternative behaviours
present(*feature*)	holds if the feature is present in the system
read(*signal*,*parameters*)	inputs a signal from the system; the variant **Read** absorbs other signals before the desired one is input
refuse(*behaviour*)	sequential behaviour with abrupt termination if the final behaviour occurs, or successful termination if not
send(*signal*,*parameters*)	outputs a signal to the system; the variant **Send** absorbs other signals before the desired one is output
sequence(*behaviour*)	sequential behaviour with abrupt termination
succeed(*behaviour*)	sequential behaviour with successful termination
test(*name*,*behaviour*)	defines a test for the given name and behaviour

Figure 2. Core MUSTARD Notation

3. The MUSTARD Scenario Notation

This section provides an overview of the MUSTARD notation for validation scenarios. Internet Telephony is used to illustrate how scenarios can be formulated.

3.1. Core Notation

The core MUSTARD notation is summarised in figure 2. MUSTARD supports standard types for parameters such as currency, date, network address, number, telephone number and text string.

3.2. Domain-Specific Vocabulary

The core notation is specialised by the input-output vocabulary for each application domain. Vocabularies are defined in a modular fashion so that new ones can be added easily. This includes domain-specific types and functions.

MUSTARD	Meaning
read(Announce,*address*,*message*)	the user receives a call progress message
read(Disconnect,*address1*,*address2*)	the user at address 1 is told that the user at address 2 has ended a call
read(StartRing,*address1*,*address2*)	the user at address 1 starts ringing for a call from the user at address 2
read(StopRing,*address1*,*address2*)	the user at address 1 stops ringing for a call from the user at address 2
send(Answer,*address*)	the user answers a call
send(Dial,*address1*,*address2*)	the user at address 1 calls the user at address 2
send(OffHook,*address*)	the user initiates a call
send(OnHook,*address*)	the user terminates a call
send(Reject,*address*, *reason*)	the user rejects a call for the given reason

Figure 3. MUSTARD Vocabulary for Internet Telephony

As an example, this section uses Internet Telephony to illustrate the application of MUSTARD. For Internet Telephony, the signals shown in figure 3 are defined by MUSTARD. This vocabulary also defines types for address, message (to indicate call progress, e.g. 'trying') and reason (cause for rejection, e.g. 'busy here'). In addition, domain-specific functions are defined. For example, the predicate **call_divert**(*address*,*condition*) checks whether the address has call diversion for the given condition (e.g. if the callee is busy or does not answer). The predicate **screen_in**(*address1*,*address2*) checks whether the user at address 1 screens (i.e. blocks) incoming calls from the user at address 2.

3.3. Basic Scenarios

Scenarios contain **read/send** actions as elementary behaviours; these are input/output from the scenario's perspective, i.e. output/input from the system's perspective. Combinators take behaviours as parameters and produce more complex behaviour. Elementary or composite behaviour may be used as a parameter. The style of the scenario language is thus applicative.

A scenario has a name (which is automatically qualified by the current feature). As an example, the following is a test for a SIP proxy. It defines a simple sequence of events: address 1 goes off-hook, receives dial tone, and goes on-hook. If the specification respects this sequence, a pass verdict is recorded for the scenario.

```
test(No_Dial,                                  test of call without dialling
  succeed(                                            successful sequence
    send(OffHook,1),                                      1 goes off-hook
    read(Announce,1,DialTone),                           1 gets dial tone
    send(OnHook,1)))                                       1 goes on-hook
```

The following scenario introduces a choice. After address 1 dials address 4, the outcome depends on the status of 4. If this is a valid number, it will start ringing from 1; but if it is an invalid number, then 1 will receive an ‘unobtainable’ message. The **offer** combinator allows a deterministic (system-decided) choice. This is appropriate here, as only the system knows whether the callee can be rung.

```
test(Ring_Or_Unobtainable,                         test of ring or unobtainable
  succeed(                                         successful sequence
    send(OffHook,1),                               1 goes off-hook
    read(Announce,1,DialTone),                     1 gets dial tone
    send(Dial,1,4),                                1 dials 4
    offer(                                         system choice
      read(StartRing,4,1),                         4 rings from 1
      read(Announce,1,Unobtainable))))             1 gets unobtainable
```

It is sometimes desirable to use the **decide** combinator for a non-deterministic (scenario-decided) choice. This ensures that all alternatives are explored. After address 1 goes off-hook, the following scenario requires both dialling and going on-hook to be tried as alternatives. In a full description of this scenario, the sequences following each of these actions would be defined.

```
test(Dial_Or_Hangup,                               test of dialling or hang-up
  succeed(                                         successful sequence
    send(OffHook,1),                               1 goes off-hook
    read(Announce,1,DialTone),                     1 gets dial tone
    decide(                                        test choice
      send(Dial,1,9),                              1 dials 9
      send(OnHook,1))))                            1 goes on-hook
```

3.4. Conditional Scenarios

The **depend** combinator makes a scenario conditional; the dependency is evaluated when the scenario is defined, not when it is executed. Dependencies are most commonly used when the specification has features. For example, a scenario can be varied according to the features that have been provisioned. The following tests the feature CFBL (Call Forward on Busy Line), deployed in either a user agent or a proxy. Call forwarding is handled differently in each case: a called user agent will announce a temporary change of number to the caller, while a proxy will automatically re-route the call. The following scenario depends on which feature has been deployed.

```
test(Busy_Forward,                                 test of forward on busy
  succeed(                                         successful sequence
    send(OffHook,1),                               1 goes off-hook
    read(Announce,1,DialTone),                     1 gets dial tone
    send(OffHook,3),                               3 goes off-hook
    read(Announce,3,DialTone),                     3 gets dial tone
    send(Dial,1,3),                                1 dials 3
    depend(                                        feature dependency
      present(AGENT_CFBL),                         agent forwarding present?
        ..,                                        behaviour for agent forwarding
      present(PROXY_CFBL),                         proxy forwarding present?
        ...)))                                     behaviour for proxy forwarding
```

Scenarios can also be made finer-grained, depending on whether a feature applies to a particular pair of users. TCS (Terminating Call Screening) is a feature that allows the called user to block calls from certain users. In the following, the dependency is on whether the user at address 2 screens calls from the user at address 1. If so, an attempt by 1 to call 2 should lead to a ‘decline’ announcement. If not, the call attempt should read to ringing. Such a condition depends on MUSTARD being aware of each user’s profile.

```
test(Screen_Caller,                                    test of caller screening
  succeed(                                             successful sequence
    send(OffHook,1),                                   1 goes off-hook
    read(Announce,1,DialTone),                         1 gets dial tone
    send(Dial,1,2),                                    1 dials 2
    depend(                                            feature dependency
      screen_in(2,1),                                  2 screens calls from 1?
        read(Announce,1,Decline),                      1 declined if so
        read(StartRing,2,1))))                         2 rings from 1 if not
```

MUSTARD currently does not have conditions that are evaluated when a scenario is executed. This is merely because a strong need for them has not been found. Because MUSTARD defines black-box tests, the conditions are strictly external to the specification. They therefore depend on values that appear in the scenarios and not in the specification. Decisions can therefore be taken when scenarios are defined, rather than during test execution. However it would be straightforward to add a conventional **if** combinator to the language.

3.5. Partial Scenarios

Scenarios often begin in similar ways. It is undesirable if the same opening behaviour has to be repeated. Instead, opening sequences can be defined as scenarios that are called by other scenarios. The *Ring_Or_Unobtainable* scenario in section 3.3 begins with one user going off-hook and dialling. This is a common start to many scenarios. The following defines this behaviour as a separate scenario that may be called as a prefix for other scenarios. Because *OffHook_Dial* does not deliver a verdict, **exit** is used to define a sequence that simply terminates normally. This allows a partial scenario to be defined separately, and then invoked as required.

```
test(OffHook_Dial,                                     initial off-hook then dial
  exit(                                                exiting sequence
    send(OffHook,1),                                   1 goes off-hook
    read(Announce,1,DialTone),                         1 gets dial tone
    send(Dial,1,4)))                                   1 dials 4

test(Ring_Or_Unobtainable_Prefix,                      test of ringing or unobtainable
  succeed(                                             successful sequence
    call(OffHook_Dial),                                invoke off-hook/dial
    offer(                                             system choice
      read(StartRing,4,1),                             4 rings from 1
      read(Announce,1,Unobtainable))))                 or 1 gets unobtainable
```

A scenario may fully define the expected input-output traces of the system. However, this is often undesirable since it may overspecify the scenario. Typically a scenario concentrates on just one aspect of behaviour; what precedes the scenario is less important. A plain **read/send** action can be replaced by **Read/Send** to absorb intervening signals (i.e. allow but ignore them).

The *Ring_Or_Unobtainable* scenario defines all actions in full. The following version focuses only on the key actions: address 1 goes off-hook, calls address 4, and then gets ringing or unobtainable.

```
test(Ring_Or_Unobtainable_Key,                         test of ring or unobtainable
  succeed(                                             successful sequence
    Send(OffHook,1),                                   1 eventually goes off-hook
    Send(Dial,1,4),                                    1 eventually dials 4
    offer(                                               system choice
      Read(StartRing,4,1),                  4 eventually rings from 1
      Read(Announce,1,Unobtainable))))  or 1 eventually gets unobtainable
```

This approach can considerably shorten many scenarios, though the difference here is small. Note that it is still necessary for the scenario to provide key inputs such as going off-hook or dialling. Without these, the scenario would not be grounded in definite values for the signal parameters. However, it is also possible to use symbolic names for signal parameters.

3.6. Complex Scenarios

In complex scenarios, it is often necessary to group sequences of actions as parameters to higher-level combinators. On its own, the **sequence** combinator produces a sequence of actions that terminates abruptly. This is always used in the context of a combinator such as **decide**, **offer** or one of those discussed below. The following scenario fragment shows address 1 going off-hook, receiving dialling tone, and dialling address 3:

```
sequence(                                  sequence
  send(1,OffHook),                     1 goes off-hook
  read(Announce,1,DialTone),           1 gets dial tone
  send(Dial,1,3))                           1 dials 3
```

Many specification problems arise due to concurrency, for example race conditions. It is desirable to check if a specification suffers from these kinds of problems. This requires scenarios that independently execute multiple behaviours in parallel (i.e. through interleaving). In the following example, addresses 1 and 2 concurrently go off-hook and dial address 3. The outcome depends on the system, so there is a system-defined choice: 3 starts ringing from 1, and 2 receives busy; or 3 starts ringing from 2, and 1 receives busy. Note how **sequence** is used to group the interleaved and alternative behaviours.

```
test(Raced_Calls,                     test of simultaneous call
  succeed(                                 successful sequence
    interleave(                           concurrent behaviour
      sequence(                                       sequence
        send(OffHook,1),                       1 goes off-hook
        read(Announce,1,DialTone),             1 gets dial tone
        send(Dial,1,3)),                             1 dials 3
      sequence(                                  plus sequence
        send(OffHook,2),                       2 goes off-hook
        read(Announce,2,DialTone),             2 gets dial tone
        send(Dial,2,3))),                            2 dials 3
    offer(                                       system choice
      sequence(                                       sequence
        read(StartRing,3,1),                    3 rings from 1
        read(Announce,2,BusyHere)),                2 gets busy
      sequence(                                    or sequence
        read(StartRing,3,2),                    3 rings from 1
        read(Announce,1,BusyHere)))))              1 gets busy
```

In fact, this scenario unnecessarily sequentialises the behaviour. It should be equally acceptable for the busy announcement to appear before ringing. To reflect this in the scenario, the alternative choices should be interleaved:

```
test(Screen_Caller_Refusal,                                        test of caller refusal
  refuse(                                                          refusal sequence
    Send(OffHook,1),                                               1 eventually goes off-hook
    read(Announce,1,DialTone),                                     1 gets dial tone
    send(Dial,1,2),                                                1 dials 2
    Read(StartRing,2,1)))                                          2 must eventually not ring from 1 (refusal)
```

Figure 4. Test of Refusing Calls from Screened Callers

```
test(Raced_Calls_Fully_Interleaved,                                test of interleaved call
  succeed(                                                         successful sequence
    ...
    offer(                                                         system choice
      interleave(                                                  concurrent behaviour
        read(StartRing,3,1),                                       3 rings from 1
        read(Announce,2,BusyHere)),                                2 gets busy
      interleave(                                                  or concurrent behaviour
        read(StartRing,3,2),                                       3 rings from 2
        read(Announce,1,BusyHere)))))                              1 gets busy
```

The parameters of **interleave** normally need to be wrapped with **sequence**. However, as shown above a single action may be used as a degenerate sequence.

The scenarios presented so far have described what the system must do. However they do not exclude undesirable behaviour, i.e. what the system must *not* do. For this situation, MUSTARD allows refusal scenarios to be defined. The **refuse** combinator defines a sequence of behaviours. If the final behaviour (that should be refused) does not happen, the scenario is considered to pass. However if the final behaviour does happen, the scenario fails.

The next example is a variant on the *Screen_Caller* scenario in section 3.4. It is shown in figure 4 for convenient reference later. Since the purpose of call screening is to block calls, it might be preferable to state this explicitly in a refusal test. If it is known that address 2 screens calls from address 1, then calls from 1 must not ring 2. The behaviour to be refused is the last parameter of **refuse**, here the act of ringing 2 from 1. **Send/Read** are used here purely to illustrate the translation of scenarios later.

In general, the refused behaviour may be composite. When 1 calls 2, suppose that neither ringing nor unobtainable should result. The refused behaviour in this case is a combination made with **offer**:

```
test(Screen_Caller_Composite,                                      test of caller refusal
  refuse(                                                          refusal sequence
    Send(OffHook,1),                                               1 eventually goes off-hook
    read(Announce,1,DialTone),                                     1 gets dial tone
    send(Dial,1,2),                                                1 dials 2
    offer(                                                         system choice
      read(StartRing,2,1),                                         2 must not ring from 1 (refusal)
      read(Announce,1,Unobtainable))))                             or 1 must not get unobtainable (refusal)
```

3.7. Extending MUSTARD

MUSTARD scenarios are handled by the m4 macro processor. This means that the scenario notation can easily be extended by defining additional macros.

The use of a prefix scenario as in section 3.5 might thus be achieved by a macro. The **define** call gives the name of a macro and its expansion (written {...} in MUSTARD). The *OffHook_Dial* macro is called simply by naming it, causing the prefix behaviour to be inserted:

```
define(OffHook_Dial,                                   macro for off-hook/dial
  {send(OffHook,1),                                    1 goes off-hook
   read(Announce,1,DialTone),                          1 gets dial tone
   send(Dial,1,4)})                                    1 dials 4

test(Ring_Or_Unobtainable_Macro,                       test of ring or unobtainable
  succeed(                                             successful sequence
    OffHook_Dial,                                      call off-hook/dial macro
    offer(                                             system choice
      read(StartRing,4,1),                             1 rings from 4
      read(Announce,1,Unobtainable))))                 or 1 gets unobtainable
```

The *Screen_Caller_Refusal* scenario in figure 4 is fixed for caller 1 and callee 2. By defining a macro, this can be made generic for any addresses. The *Screen_Caller_Any* macro is parameterised by a pair of addresses ($1 and $2 are the first and second macro parameters). It is called to define the scenario *Screen_Caller_3_4*:

```
define(Screen_Caller_Any,                              macro for parameterised screening
  {test(Screen_Caller_$1_$2,                           test of caller screening
    refuse(                                            refusal sequence
      send(OffHook,$1),                                first goes off-hook
      read(Announce,$1,DialTone),                      first gets dial tone
      send(Dial,$1,$2),                                first dials second
      read(StartRing,$2,$1)))})                        second must not ring from first (refusal)

Screen_Caller_Any(3,4)                                 call screening macro for 3 and 4
```

Repetition can be defined with a macro. In the following example, macro *thrice* repeats its parameters three times ($* means all macro parameters). This is used to repeat the sequence off-hook, dial tone, on-hook:

```
define(thrice,                                         macro for three-times repeat
  {$*,$*,$*})                                          use parameters three times

test(No_Dial_Thrice,                                   test of three calls without dialling
  succeed(                                             successful sequence
    thrice(                                            call three-times macro
      send(OffHook,1),                                 1 goes off-hook
      read(Announce,1,DialTone),                       1 gets dial tone
      send(OnHook,1))))                                1 goes on-hook
```

Macros can also be used to define special-purpose types and functions. This in fact is how MUSTARD works internally, with domain-specific vocabularies being defined as collections of macros.

4. Using MUSTARD with LOTOS

This section explains how MUSTARD is used with specifications written in LOTOS (Language Of Temporal Ordering Specification [11]). A knowledge of LOTOS would help in understanding the details of this section. However the LOTOS code is commented to explain what it means. Tutorials on LOTOS can be found in [3, 32] and online at *http://www.cs.stir.ac.uk/well/*.

MUSTARD	LOTOS
call(*feature*,*scenario*)	*feature_scenario* [*gates*] ≫
decide(*behaviour1*,...)	**i**; *behaviour1* [] ...
exit(*behaviour1*,...)	*behaviour1*; ... **Exit**
interleave(*behaviour1*,...)	*behaviour1* ||| ...
offer(*behaviour1*,...)	*behaviour1* [] ...
read(*signal*,*parameter1*,...)	*gate* !*signal* !*parameter1* ...;
Read(*signal*,*parameter1*,...)	*Wait_N* [*gates*] ≫
refuse(*behaviour1*,...,*behaviourN*)	*behaviour1*; ... (*behaviourN*; **Stop** [] **i**; OK; **Stop**)
send(*signal*,*parameter1*,...)	*gate* !*signal* !*parameter1* ...;
Send(*signal*,*parameter1*,...)	*Wait_N* [*gates*] ≫
sequence(*behaviour1*,...)	*behaviour1*; ...
succeed(*behaviour1*,...)	*behaviour1*; ... OK; **Stop**
test(*name*,*behaviour*)	**Process** *name* [*gates*,OK] : *functionality* := *behaviour* **EndProc**

Figure 5. LOTOS Interpretation of MUSTARD

4.1. Mapping MUSTARD to LOTOS

The mapping of MUSTARD onto LOTOS is summarised in figure 5; see the comments in figure 6 for an indication of what the various constructs mean. A scenario is translated into a LOTOS process whose functionality (exiting or not) depends on the scenario behaviour. The event gates are those of the specification. The LOTOS behaviour uses the special event *OK* to signal that a scenario has passed.

Most of the translation is straightforward except for **Read** and **Send**. These define and call auxiliary processes named *Wait_N* where *N* is a serial number. These processes loop to absorb all signals except the desired one, and exit on the required signal. MUSTARD knows the signals used in each application domain, so it can construct these processes automatically.

As an example of how MUSTARD is translated into LOTOS, the translation of *Screen_Caller_Refusal* from figure 4 is given in figure 6. (Some small simplifications have been made for ease of presentation.) This includes *Wait* processes for the **Send** and **Read** actions. Event gate *User* in figure 6 is the one used by the specification to communicate with the Internet Telephony user.

4.2. Validating MUSTARD Scenarios

When LOTOS is the specification language, MUSTARD is used with the LOLA toolset (LOTOS Laboratory [26]). LOLA is a good choice because it offers specialised support for validating specifications. When MUSTARD processes a LOTOS specification, it goes through the following steps:

```
Process Screen_Caller_Refusal [User,OK] : NoExit :=                          test process
  Wait_1 [User]                                                wait for 1 going off-hook
≫                                                                                enables
  User !Announce !1 !DialTone;                                           1 gets dial tone
  User !Dial !1 !2;                                                              1 dials 2
  (
    Wait_2 [User]                                    wait for 2 ringing from 1 (refusal)
  ≫                                                                              enables
    Stop                                                                      test fails
  []                                                                                  or
    i;                                                                    internal event
    OK;                                                                      test passes
    Stop                                                                      test stops
  )

Where                                                                  local definitions

  Process Wait_1 [User] : Exit :=                              wait for 1 going off-hook
    User !OffHook !1; Exit                           1 going off-hook finishes waiting
  []                                                                                  or
    User !Announce ?par1:Address ?par2:Message;Wait_1 [User]       absorb announcement
  []                                                                                  or
    User !Disconnect ?par1:Address ?par2:Address;Wait_1 [User]     absorb disconnection
  []                                                                                  or
    User !StartRing ?par1:Address ?par2:Address;Wait_1 [User]     absorb start ringing
  []                                                                                  or
    User !StopRing ?par1:Address ?par2:Address;Wait_1 [User]       absorb stop ringing
  EndProc

  Process Wait_2 [User] : Exit :=                              wait for 2 ringing from 1
    User !Announce ?par1:Address ?par2:Message;Wait_2 [User]       absorb announcement
  []                                                                                  or
    User !Disconnect ?par1:Address ?par2:Address;Wait_2 [User]     absorb disconnection
  []                                                                                  or
    User !StartRing ?par1:Address ?par2:Address;                       get start ringing
    (
      [(par1 Eq 2) And (par2 Eq 1)] -> Exit            2 ringing from 1 finishes waiting
    []                                                                                or
      [(par1 Ne 2) Or (par2 Ne 1)] -> Wait_2 [User]    absorb if not 2 ringing from 1
    )
  []                                                                                  or
    User !StopRing ?par1:Address ?par2:Address;Wait_2 [User]       absorb stop ringing
  EndProc

EndProc
```

Figure 6. LOTOS Translation of Sample Scenario

- The specification is processed to extract its name, event gates, user profiles (if any) and features (if any).
- The feature list is used to determine which MUSTARD scenario files to use. These are concatenated and translated into LOTOS test processes that are merged with the original specification.
- Each test is then executed using the *TestExpand* function of LOLA. This makes the scenario acts as the environment of the specification. The state space of the modified specification is fully expanded. Each terminal branch of the state space is checked for the presence (success) or absence (failure) of a special test success event: *OK* in MUSTARD. Any failure tree is converted back into MUSTARD notation.

The following output is produced when MUSTARD validates the LOTOS specification of an Internet Telephony proxy. The PROXY tests are scenarios defined for the proxy. The PROXY CFBL tests are for Call Forward Busy Line in the proxy, while the PROXY TCS tests are for Terminating Call Screening in the proxy. The high number of successes for some tests reflects the high degree of concurrency and non-determinism in the specification.

```
Test PROXY Clear Before Dial ...      Pass     12 success   0 fail  1.5 secs
Test PROXY Clear Before Answer ...    Pass     24 success   0 fail  2.4 secs
Test PROXY Caller Clear ...           Pass     24 success   0 fail  4.2 secs
Test PROXY Callee Clear ...           Pass     23 success   0 fail  3.9 secs
Test PROXY Call Self ...              Pass     57 success   0 fail  1.9 secs
Test PROXY Simultaneous Call ...      Pass    244 success   0 fail  5.1 secs
Test PROXY CFBL Busy Forward ...      Pass     44 success   0 fail 15.5 secs
Test PROXY CFBL No Forward ...        Pass    247 success   0 fail  4.9 secs
Test PROXY CFBL Free ...              Pass     57 success   0 fail  2.4 secs
Test PROXY TCS Reject ...             Pass      4 success   0 fail  0.6 secs
Test PROXY TCS No Reject ...          Pass     24 success   0 fail  2.6 secs
Test PROXY TCS No Screen ...          Pass     75 success   0 fail  2.7 secs
```

5. Using MUSTARD with SDL

This section explains how MUSTARD is used with specifications written in SDL. In fact, validation uses scenarios translated to MSCs (Message Sequence Charts [16]). A knowledge of MSCs would help in understanding the details of this section. However the MSC code is commented to explain what it means. Tutorials on MSC 96 can be found in [28] and online at *www.sintef.no/time/ELB40/ELB/MSC96/MSC96.pdf*.

5.1. Mapping MUSTARD to MSCs

The mapping of MUSTARD onto an MSC is summarised in figure 7; see the comments in figure 8 for an indication of what the various constructs mean. Each scenario is translated into one MSC. *Env0* is the system environment (i.e. the scenario), *Sys1* is the system being validated, and *All* means both of these. Input/output requires pairs of actions: the environment sending and the system receiving (or vice versa).

MUSTARD	MSC
call(*feature*,*scenario*)	**Reference** *feature_scenario*;
decide(*behaviour1*,...)	**Alt Begin**; *behaviour1*; **Alt**; ... **Alt End**;
exit(*behaviour1*,...)	*behaviour1*; ... Env0: **EndInstance**; Sys1: **EndInstance**;
interleave(*behaviour1*,...)	**Par Begin**; *behaviour1*; **Par**; ... **Par End**;
offer(*behaviour1*,...)	**Alt Begin**; *behaviour1*; **Alt**; ... **Alt End**;

read(*signal*,*parameter1*,...)	Sys1: **Out** *signal*(*parameter1*,...) **To** Env0;
	Env0: **In** *signal*(*parameter1*,...) **From** Sys1;

Read(*signal*,*parameter1*,...)	All: **Loop** <0,Inf> **Begin**; All: **Alt Begin**; *system outputs as alternatives*; **Alt End**; **Loop End**; Sys1: **Out** *signal*(*parameter1*,...) **To** Env0; Env0: **In** *signal*(*parameter1*,...) **From** Sys1;
refuse(*behaviour1*,...,*behaviourN*)	*behaviour1*; ... All: **Exc Begin**; *behaviourN*; Env0: **Stop**; Sys1: **Stop**; All: **Exc End**;

send(*signal*,*parameter1*,...)	Env0: **Out** *signal*(*parameter1*,...) **To** Sys1;
	Sys1: **In** *signal*(*parameter1*,...) **From** Env0;

Send(*signal*,*parameter1*,...)	All: **Loop** <0,Inf> **Begin**; All: **Alt Begin**; *system outputs as alternatives*; **Alt End**; **Loop End**; Env0: **Out** *signal*(*parameter1*,...) **To** Sys1; Sys1: **In** *signal*(*parameter1*,...) **From** Env0;
sequence(*behaviour1*,...)	*behaviour1*; ...
succeed(*behaviour1*,...)	*behaviour1*; ... Env0: **EndInstance**; Sys1: **EndInstance**;
test(*name*,*behaviour*)	**MSC** *name* Env0: **Instance**; Sys1: **Instance System** *name*; *behaviour1*; ... **EndMSC**;

Figure 7. MSC Interpretation of MUSTARD

Most of the translation is straightforward except for **Read** and **Send**. These use loops that absorb all signals except the desired one, and continue on the required signal. MUSTARD knows the signals used in each application domain, so it can construct these loops automatically.

Support also exists to translate MUSTARD into SDL, allowing scenarios to be expressed in the same language as the specification. This has some advantages such as sharing the same data types. However it is more awkward to use the generated SDL for validation since the specification must be automatically adjusted to use the scenario as its environment. It is also messy to translate **interleave** into SDL.

A translation of MUSTARD into MSCs is therefore preferred. Validating SDL is one of the prime uses of MSCs. Interleaving is supported directly by MSCs. However as figure 7 shows, there is currently no support for non-determinism: **decide** is translated the same way as **offer**. Non-deterministic extensions have been defined for MSCs (e.g. [24]). When tool support for these extensions is readily available, **decide** and **offer** will be distinguished properly.

The MSC version used is an issue. Currently MUSTARD translates into MSC 96 because tool support for MSC 2000 is still not widespread. Using MSC 96 has some disadvantages, such as lack of proper data handling. When MSC 2000 tools are more widely available, MUSTARD will be modified for this.

As an example of how MUSTARD is translated into MSC, the translation of *Screen_Caller_Refusal* from figure 4 is given in figure 8. (Some small simplifications have been made for ease of presentation.) This example shows loops for the **Send** and **Read** actions.

5.2. Validating MUSTARD Scenarios

When SDL is the specification language, MUSTARD is used with the Telelogic TAU SDL suite (*www.telelogic.com/products/tau/sdl/index.cfm*). TAU is a good choice because it offers specialised support for validating specifications. When MUSTARD processes an SDL specification, it goes through the following steps:

- The specification is processed to extract its name, user profiles (if any) and features (if any).
- The feature list is used to determine which MUSTARD scenario files to use. Scenarios are translated into individual MSC files.
- Each MSC is then executed using the *Verify-MSC* function of TAU. This treats a scenario as the environment of the specification. If the MSC is not respected, failure reports are generated as MSCs. These are then converted back into MUSTARD notation.

The following output is produced when MUSTARD validates the SDL specification of an Internet Telephony proxy. The scenarios are identical to those for LOTOS in section 4.2. A comparison with the LOTOS results in section 4.2 will show that only one success path is given for each scenario. This reflects the different validation strategy used by TAU. Multiple successes or failures are possible, e.g. if there are alternative or concurrent paths in the MSC. Validation times are shorter than for LOTOS, partly because there is less concurrency and non-determinism, and partly because the TAU validator first compiles the scenarios and the system to C (unlike LOLA that interprets a LOTOS specification).

```
Test PROXY Clear Before Dial ...      Pass      1 success   0 fail  0.4 secs
Test PROXY Clear Before Answer ...    Pass      1 success   0 fail  0.5 secs
Test PROXY Caller Clear ...           Pass      1 success   0 fail  0.5 secs
Test PROXY Callee Clear ...           Pass      1 success   0 fail  0.5 secs
Test PROXY Call Self ...              Pass      1 success   0 fail  0.4 secs
```

```
MSC Screen_Caller;                                        test MSC
Env0: Instance;                                           environment instance
Sys1: Instance System SIPSystem;                          system instance
  All: Loop <0,Inf> Begin;                                begin indefinite loop
    All: Alt Begin;                                       begin alternatives
      Sys1: Out Announce To Env0;                         absorb announcement
      Env0: In Announce From Sys1;
    Alt;                                                  or
      Sys1: Out Disconnect To Env0;                       absorb disconnection
      Env0: In Disconnect From Sys1;
    Alt;                                                  or
      Sys1: Out StartRing To Env0;                        absorb start ringing
      Env0: In StartRing From Sys1;
    Alt;                                                  or
      Sys1: Out StopRing To Env0;                         absorb stop ringing
      Env0: In StopRing From Sys1;
    Alt End;                                              end alternatives
  Loop End;                                               end loop
  Env0: Out OffHook(1) To Sys1;                           1 goes off-hook
  Sys1: In OffHook(1) From Env0;
  Sys1: Out Announce(1,DialTone) To Env0;                 1 gets dial tone
  Env0: In Announce(1,DialTone) From Sys1;
  Env0: Out Dial(1,2) To Sys1;                            1 dials 2
  Sys1: In Dial(1,2) From Env0;
  All: Exc Begin;                                         begin exceptional behaviour
    All: Loop <0,Inf> Begin;                              begin indefinite loop
      All: Alt Begin;                                     begin alternatives
        Sys1: Out Announce To Env0;                       absorb announcement
        Env0: In Announce From Sys1;
      Alt;                                                or
        Sys1: Out Disconnect To Env0;                     absorb disconnection
        Env0: In Disconnect From Sys1;
      Alt;                                                or
        Sys1: Out StartRing To Env0;                      absorb start ringing
        Env0: In StartRing From Sys1;
      Alt;                                                or
        Sys1: Out StopRing To Env0;                       absorb stop ringing
        Env0: In StopRing From Sys1;
      Alt End;                                            end alternatives
    Loop End;                                             end loop
    Sys1: Out StartRing(2,1) To Env0;                     2 rings from 1 (refusal)
    Env0: In StartRing(2,1) From Sys1;
    Env0: Stop;                                           stop environment
    Sys1: Stop;                                           stop system
  All: Exc End;                                           end exceptional behaviour
Env0: EndInstance;                                        end environment instance
Sys1: EndInstance;                                        end system instance
EndMSC;
```

Figure 8. MSC Translation of Sample Scenario

```
Test PROXY Simultaneous Call ...      Pass      1 success    0 fail   0.5 secs
Test PROXY CFBL Busy Forward ...      Pass      1 success    0 fail   0.6 secs
Test PROXY CFBL No Forward ...        Pass      1 success    0 fail   0.4 secs
Test PROXY CFBL Free ...              Pass      1 success    0 fail   0.4 secs
Test PROXY TCS Reject ...             Pass      1 success    0 fail   0.4 secs
Test PROXY TCS No Reject ...          Pass      1 success    0 fail   0.5 secs
Test PROXY TCS No Screen ...          Pass      1 success    0 fail   0.4 secs
```

6. Using MUSTARD In Practice

6.1. Scenario Selection

The approach depends on how effectively the scenarios characterise the system (and its features if defined). It is common to define use-case scenarios during requirements specification; this provides direct input into MUSTARD. However, MUSTARD scenarios can be defined manually and separately. In the five case studies mentioned in section 2.2, the scenarios were obtained by following the major specification paths. As an example, consider CFBL (Call Forward Busy Line). There are three cases to consider:

- The callee is not busy, so call forwarding does not apply.
- The callee is busy but has not subscribed to CFBL, so call forwarding does not apply.
- The callee is busy and has subscribed to CFBL, so call forwarding takes place.

Each of these cases becomes a MUSTARD scenario. Similar principles were followed for other features. TWC (Three-Way Calling) is one of the most complex features; it requires 23 scenarios to deal with all its distinct behaviours. On average, around five scenarios are defined for each typical voice feature.

The derivation of scenarios can also be automated. The major problem is when the specification is heavily influenced by data. Without some guidance from a domain specialist, automated test generation is impracticable. As reported in [37], the author has developed a language for annotating specifications with constraints on input values and combinations of inputs. This restricts the specification sufficiently for test scenarios to be generated automatically. Scenarios for radiotherapy accelerators were obtained using this technique.

There is an obvious question of how effectively scenarios cover a specification. Of course, this is an issue for any testing approach. When scenarios are generated automatically, a definite answer can be given. For example, the radiotherapy accelerator case study uses a Chinese Postman tour that guarantees each transition is followed at least once. When scenarios are defined manually, some other measure of coverage must be used. State space exploration tools (e.g. TAU) report coverage during scenario execution. This gives an objective assessment of how complete the scenarios are.

The translation of scenarios into MUSTARD is currently not automated, but could be. The automatically generated tests of [37] can be mechanically rendered in MUSTARD. Scenarios defined by other means, e.g. UML, could also be automatically translated.

6.2. Integration with Other Techniques

MUSTARD has an applicative syntax. This is convenient for use with the supporting macro processor, but perhaps not so convenient for use with other tools. MUSTARD therefore also has an XML syntax defined by an XML schema. This makes syntax-checking, parsing and processing by other tools relatively straightforward. As an example, the scenario in figure 4 has the following XML representation:

```
<test name="Screen_Caller_Refusal">                              test of caller refusal
  <refuse>                                                              refusal sequence
    <Send name="OffHook" par1="1"/>                                      1 goes off-hook
    <read name="Announce" par1="1" par2="DialTone"/>                     1 gets dial tone
    <send name="Dial" par1="1" par2="2"/>                                      1 dials 2
    <Read name="StartRing" par1="2" par2="1"/>              2 must not ring from 1 (refusal)
  </refuse>
</test>
```

As explained in section 2.2, MUSTARD has been demonstrated on five significant case studies over a period of six years. It is therefore believed to be mature and widely applicable. Industrial adoption would be relatively straightforward. The tool requirements are modest: Perl and m4 are readily and freely available, and run on many platforms. Validation requires acceptable memory (under 10Mb) and execution time (under a few minutes). Transferring the knowledge to use MUSTARD would not require significant effort. In fact, this paper has introduced all the core MUSTARD notation, and has illustrated its extension for one application domain.

As noted earlier, MUSTARD applies to reactive systems that can be validated as black boxes. This encompasses a broad range of systems. MUSTARD requires specifications to be in some executable language. Although LOTOS and SDL have been illustrated in this paper, the use of BPEL has been mentioned for web services. A translation strategy has also been devised for Java. It is therefore believed that MUSTARD can be effectively used to validate implementations as well.

MUSTARD can be used in parallel with other testing techniques. It can therefore supplement existing test practices. Since MUSTARD is automated, the manual effort is confined to developing appropriate scenarios. This is a one-off exercise. As noted in section 6.1, scenarios may well be defined anyway as part of the requirements; a degree of automation is also possible. For certain categories of system, e.g. safety-critical or quality-critical ones, validation is paramount. The additional rigour of MUSTARD can therefore be justified. For less critical systems, a trade-off can be made between manual testing practices (which are very costly) and MUSTARD-based validation (which is automated, but needs more investment in time initially).

MUSTARD is distributed along with the CRESS toolset (*www.cs.stir.ac.uk/~kjt/research/mustard.html*). This may be freely used by researchers for non-commercial activities.

7. Conclusion

Specifications need to be as thoroughly checked as implementations. Unfortunately formal verification of specifications is still impracticable for realistic systems. Scenario-based validation has therefore been proposed as a pragmatic solution for gaining confidence in a specification. This is valuable when a specification is first written, and after changes to the specification itself or to the requirements.

Five case studies have been briefly mentioned. Three of these deal with various forms of voice service: Intelligent Networks, Internet Telephony and Interactive Voice Response. Further case studies have dealt with web services, and with the control system of a radiotherapy accelerator. Internet Telephony has been used throughout the paper to illustrate the formulation and validation of scenarios with MUSTARD. This experience gives confidence that the approach can be used on a large variety of realistic reactive systems.

It has been seen how specifications in LOTOS or SDL are validated. The web services case study uses implementation languages (BPEL, WSDL) for validation. For the voice and web services case studies, the specifications were generated automatically by CRESS. However there is very little dependence on CRESS; for example, it was not used for the radiotherapy accelerator case study. MUSTARD can therefore be used with almost any specification in these languages. It is also possible to use the approach with other languages, whether specification or implementation languages. The main requirements are that specifications be behavioural in nature, and that they can be automatically explored.

Future work will mainly concern the adaptation of MUSTARD for new application domains and new languages. This is fairly straightforward as the design of the tools is modular. For example, a new application just requires extra macros for the domain-specific vocabulary. A new target language would require macro support similar to the existing LOTOS and SDL modules.

Acknowledgements

Telelogic kindly provided an academic licence for the TAU toolset used in this work. The author thanks Rick Reed (TSE Ltd.) for advice on using MSCs. The author is also grateful to the referees for their insightful comments on a draft of the article.

REFERENCES

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, editors. *Business Process Execution Language for Web Services*. Version 1.1. BEA, IBM, Microsoft, SAP, Siebel, May 2003.
2. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Lie, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language*. Version 2.0. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Feb. 2005.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1):25–59, Jan. 1988.
4. E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. K. Sabnani, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1988.
5. D. W. Bustard and A. C. Winstanley. Making changes to formal specifications: Requirements and an example. *IEEE Transactions on Software Engineering*, 20(8):562–568, Aug. 1994.
6. M. H. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41:115–141, Jan. 2003.
7. E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, pages 64–69, Mar. 1993.
8. A. M. L. de Vasconcelos and J. A. McDermid. Incremental processing of Z specifications. In M. Diaz and R. Groz, editors, *Proc. Formal Description Techniques V*, pages 53–70. North-Holland, Amsterdam, Netherlands, Oct. 1992.
9. L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation: A synchronous point of view. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 262–275. IOS Press, Amsterdam, Netherlands, Sept. 1998.
10. ETSI. *Testing and Test Control Notation Version 3*. ETSI-201-873. European Telecommunications Standards Institute, Sofia Antipolis, France, 2003.

11. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.
12. ISO/IEC. *Information Technology – Framework: Formal Methods in Conformance Testing*. ISO/IEC 13245-1. International Organization for Standardization, Geneva, Switzerland, 1997.
13. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework*. ISO/IEC 9646. International Organization for Standardization, Geneva, Switzerland, 1998.
14. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC 9646-3. International Organization for Standardization, Geneva, Switzerland, 1998.
15. ITU. *Intelligent Network – Q.120x Series Intelligent Network Recommendation Structure*. ITU-T Q.1200 Series. International Telecommunications Union, Geneva, Switzerland, 2000.
16. ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 2000.
17. ITU. *Packet-Based Multimedia Communication Systems*. ITU-T H.323. International Telecommunications Union, Geneva, Switzerland, Nov. 2000.
18. ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.
19. C. Jard and T. Jéron. TGV: Theory, principles and algorithms. *Software Tools for Technology Transfer*, 2004.
20. Ji He and K. J. Turner. Verifying and testing asynchronous circuits using LOTOS. In T. Bolognesi and D. Latella, editors, *Proc. Formal Methods for Distributed System Development (FORTE XIII/PSTV XX)*, pages 267–283, London, UK, Oct. 2000. Kluwer Academic Publishers.
21. B. Kelly, M. Crowther, J. King, R. Masson, and J. DeLapeyre. Service validation and testing. In K. E. Cheng and T. Ohta, editors, *Proc. 3rd. International Workshop on Feature Interactions in Telecommunications*, pages 173–184. IOS Press, Amsterdam, Netherlands, 1995.
22. D. R. Kuhn. A technique for analyzing the effects of changes in formal specification. *The Computing Journal*, 35:574–578, Dec. 1992.
23. Q. Li, R. L. Probert, W. Skelton, and Y. Xu. SIMPL-T (SDL intended for management and planning of tests) – A simple test language for SDL. In D. Amyot and A. Williams, editors, *Proc. SAM'04*, Ottawa, Canada, June 2004.
24. N. Mansurov and D. Zhukov. Automatic synthesis of SDL models in use case methodology. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *Proc. SDL'99*. Elsevier Science Publishers, Amsterdam, Netherlands, June 1999.
25. R. D. Nicola. External equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.
26. S. Pavón Gomez, D. Larrabeiti, and G. Rabay Filho. LOLA user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, Feb. 1995.
27. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnson, J. Peterson, R. Sparks, M. Handley, and E. Schooler, editors. *SIP: Session Initiation Protocol*. RFC 3261. The Internet Society, New York, USA, June 2002.
28. E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
29. R. Seindal. GNU *m4* (version 1.4). Technical report, Free Software Foundation, 1997.
30. I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, Massachusetts, USA, 1996.
31. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.
32. K. J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York, Jan. 1993.
33. K. J. Turner. Validating architectural feature descriptions using LOTOS. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 247–261. IOS Press, Amsterdam, Netherlands, Sept. 1998.
34. K. J. Turner. Formalising the CHISEL feature notation. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 241–256. IOS Press, Amsterdam, Netherlands, May 2000.
35. K. J. Turner. Representing new voice services and their features. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 123–140. IOS Press, Amsterdam, Netherlands, June 2003.
36. K. J. Turner. Analysing interactive voice services. *Computer Networks*, 45(5):665–685, Aug. 2004.
37. K. J. Turner. Test generation for radiotherapy accelerators. *Software Tools for Technology Transfer*, Oct. 2004.
38. K. J. Turner and M. Sighireanu. (E-)LOTOS: (Enhanced) language of temporal ordering specification. In M. Frappier and H. Habrias, editors, *Software Specification Methods*, chapter 10, pages 165–190. Springer-Verlag, Godalming, UK, Jan. 2001.

39. F. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. *IEEE Transactions on Very Large Scale Integration Systems*, 3:201–214, 1995.
40. VoiceXML Forum. *Voice eXtensible Markup Language*. VoiceXML Version 2.0. VoiceXML Forum, Jan. 2003.