TSF, a Test Specification Formalism

Loe Feijs, Mark Huizer
Eindhoven University of Technology

Abstract

TTCN is the ISO language for defining tests of protocol services and protocol entities. We report from a project studying the semantics of TTCN using process algebra, which is feasible, in principle. TTCN is a complex language with many features, some of which are essential for testing, others being typically in the style of traditional imperative languages. Using the insights gained from this study we synthesized TSF, a new language, which contains two essential ingredients for testing, but resembles PSF in all other aspects. Amongst other things, TSF contains a novel operator \( \rightarrow \). It is argued that a test itself is a formal object, the correctness of which with respect to a given specification is subject to formal analysis. This is illustrated by a simple example.

Key Words: Protocols, Testing, TTCN, formal semantics, PSF.

1 Introduction and motivation

TTCN is the ISO language for defining tests of protocol entities and protocol services [1]. In general, the subject of software testing is highly relevant because in the industrial practice of software development it often takes a significant fraction of the total development effort. For testing protocols, ISO has established a framework containing terminology and concepts. Important issues in protocol testing are the distinction between conformance testing and interoperability testing (see e.g [2]), the fact that telecommunication is a multi-vendor business where protocol entities from distinct suppliers must cooperate. Not only software is to be tested, but combined hardware-software systems. And of course protocol testing demands languages capable of expressing communication behaviour. Typical studies are [3] (relating tests and sequence charts) and [4] (studying test methodologies in a formal setting). Often communication systems are specified and realized using SDL, or in a combination of a high-level language (such as LOTOS, ESTELLE and PSF) and a programming language (C). TTCN is meant as a means of describing tests at an abstract level whereas interworkings and message sequence charts sometimes play a role when deriving tests from SDL. Much
research has already been done on automated test generation (see [5] for a survey). There is still some ongoing debate whether one should use 'normal languages' for describing tests, or special languages.

We undertook to study TTCN and its semantics. For this purpose we translated an essential part of the language to ACP and did various exercises in manual calculations with this semantics. From this study we learned amongst other things that TTCN is a complex language with an un-orthodox syntax, and that TTCN contains certain concepts which are specific for testing and which are interesting for further study and application. We do not express any particular opinion about the design of the language TTCN, but we found that it is quite remote from the kind of language constructs encountered in the ACP and PSF worlds.

This paper contains an attempt to bridge the gap by proposing a language called TSF. It is our goal to keep TSF sufficiently simple to understand its syntax and semantics in terms of initial algebra specifications and process algebra, just like PSF. This enables the direct application of theories and tool technology already developed for ACP and PSF. PSF has already been successfully applied to a large variety of protocols. [7][8]

One could envisage translators from TTCN to TSF as well as a combined PSF/TSF simulation environment. If protocols are specified in PSF, the tests can thus be simulated themselves before putting them into action on the reals systems under test.

It is important to note that a test itself is a formal object, the correctness of which with respect to a given specification is subject to formal analysis. This will be illustrated by a simple example.

In Section 2 a survey of TTCN is given. In Section 3 a simple example is given in TTCN. The sections on TTCN can be skipped by readers wanting to focus on TSF (sections 4 to 6 do not rely on definitions from the sections on TTCN). In Section 4 the new language TSF is introduced. In Section 5 a formal semantics for TSF is proposed, using ACP. In Section 6 the same example is given in TSF and the work of Section 5 is applied to this example. In Section 7 some conclusions are drawn.

2 TTCN

TTCN has a great variety of features, some of which are essential for its main purpose of testing, whereas other features are similar to those encountered in other imperative languages for describing communicating processes.

In our view, the following two language features of TTCN are essential ingredients for testing:

- verdicts: the purpose of a test is to give a kind of 'yes' or 'no' answer with respect to the question whether the implementation under test is in order or not. This is called a 'verdict'. TTCN has three
verdicts: pass, inconclusive and fail. Furthermore there is a notion of 'preliminary verdict'.

- ordering on alternative statements: typical tests check on 'expected' behaviour first and fall back to other options only if the expected behaviour is not offered by the implementation under test.

Many other features in TTCN do not seem essential for testing purposes, but are orthogonal to the test features. These are needed to give the language sufficient expressive power. In the TSF proposal (Section 4) we do not use these but we adopted PSF-like constructs. These TTCN features include:

- sequential composition by means of increasing level of indentation;
- alternative composition (ordered) by means of remaining at the same level of indentation;
- subroutine mechanism (called tree attachment), denoted by +T when calling a subtree T;
- assignments to local and global variables;
- a CSP-like syntax for input and output ('?' and '!');
- constraints on communications, where a constraint on a send is used to give a package a value, and a constraint on a receive is used as a guard;
- guards.

We gave a formal semantics for a part of TTCN by using ACP. Due to space limitations and the complexity of TTCN, we cannot give the details here; it suffices to say that we worked along the same lines described in Section 5 (but we did some work on TTCN first, turning to TSF later).

During execution, a TTCN 'program' maintains a kind of global variable called R (for result) containing the verdict obtained so far. Verdicts are updated in a particular way, according to the table given below. The following verdict values are considered: N, or None is an initialization value; P, or Pass is a final verdict telling that the implementation under test (IUT) is in order (for some aspect tested); F, or Fail states that the test stops with a negative outcome; I for Inconclusive is an undetermined outcome (e.g. the network failed so that the test could not complete); R or Result means that the test should stop and that the verdict obtained so far becomes the final verdict. E denotes a test error (telling that the test itself is in error, rather than the IUT). Moreover there are preliminary verdicts, also called temporary verdicts, which are written as (P), (I) and (F). These are useful for giving preliminary conclusions, which can be updated later. The following table gives the result of a stated verdict on the variable R:
Some further details of TTCN will be explained using a very small example in the next section.

3 Example in TTCN

3.1 The one-place buffer

Let us assume that we want to test a one-place buffer, called BUF, which will be our model of an IUT. It is supposed to have two points of control and observation (PCO), which is test terminology for 'port'. The PCOs are U (upper) and L (lower). There are two service primitives: DU (data upper) and DL (data lower), which carry data values which are non-negative integers.

The supposed behaviour is as follows: initially the buffer is empty. It accepts one input via U and then it is ready to output its value via L. After that it is empty again and it can accept new input. When non-empty, BUF should not engage in any input communication.

3.2 TTCN test of the buffer

TTCN provides mechanisms for ordering tests into test suites, for declaring variables and constraints, and so on. Here we focus on the dynamic behaviour, which takes the shape of one or more tables in which the essential behaviour is given as a so-called tree. Assume constraints CU0 and CL0 have been declared somehow and that they demand that the data values contained in DU and DL are zero, respectively. Then the following table is a TTCN test case.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Dynamic Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference:</td>
<td>Test1</td>
</tr>
<tr>
<td>Identifier:</td>
<td>Test1</td>
</tr>
<tr>
<td>Comments:</td>
<td>The simplest test possible</td>
</tr>
<tr>
<td>U?DU</td>
<td>CU0</td>
</tr>
<tr>
<td>L?DL</td>
<td>CL0</td>
</tr>
<tr>
<td>L?Otherwise</td>
<td></td>
</tr>
</tbody>
</table>
We ought to explain the tree notation. In TTCN, indentation has meaning: going to a deeper level of indentation denotes sequential composition. Alternatives are expressed by statements at the same level of indentation. Alternatives must be tried in the stated order, so L?DL with constraint CLO must be tried first and only if there is no DL message containing a zero, the next alternative is tried (L?Otherwise), which will result in Fail.

For a correct implementation of a one-place-buffer, the test above leads to the expected answer (a verdict trace with final verdict 'Pass').

4 TSF: a test specification formalism

In order to represent the essentials of TTCN in an alternative way, a new language is constructed, TSF (Test Specification Formalism). This language will contain essential ingredients of TTCN, and will resemble PSF.

Where possible, we refer to PSF (see [7]) for the production rules. This is represented by \texttt{<as-in-psf>} in the syntax-rules. The specification of TSF in Extended Backus Naur Form is given below. The following notation is used: \{ \texttt{item sep} \}+ means one or more occurrences of \texttt{item}, separated by \texttt{sep}.

\[
\begin{align*}
\langle \text{specification} \rangle &::= \langle \text{module} \rangle + \\
\langle \text{module} \rangle &::= \langle \text{data-module} \rangle \mid \langle \text{test-module} \rangle \\
\langle \text{data-module} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{var-ident-list} \rangle &::= \{ \langle \text{var-ident} \rangle \"\\", \}+ \\
\langle \text{test-module} \rangle &::= \"\text{test}\" \"\text{module}\" \langle \text{module-ident} \rangle \\
&\quad \\text{"begin"} \\
&\quad [ \langle \text{t-exports} \rangle ] \\
&\quad [ \langle \text{imports} \rangle ] \\
&\quad [ \langle \text{atoms} \rangle ] \\
&\quad [ \langle \text{tests} \rangle ] \\
&\quad [ \langle \text{sets} \rangle ] \\
&\quad [ \langle \text{communications} \rangle ] \\
&\quad [ \langle \text{tests} \rangle ] \\
&\quad [ \langle \text{sets} \rangle ] \\
&\quad \"\text{end}\" \langle \text{module-ident} \rangle \\
\langle \text{t-exports} \rangle &::= \"\text{exports}\" \\
&\quad \text{"begin"} \\
&\quad [ \langle \text{atoms} \rangle ] \\
&\quad [ \langle \text{tests} \rangle ] \\
&\quad [ \langle \text{sets} \rangle ] \\
&\quad \"\text{end}\" \\
\langle \text{imports} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{atoms} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{tests} \rangle &::= \"\text{tests}\" \langle \text{test-decl-list} \rangle + \\
\langle \text{test-decl-list} \rangle &::= \{ \langle \text{test-ident} \rangle \"\\", \}+ \[ \"\text{::}\" \langle \text{input-type} \rangle \] \\
\langle \text{input-type} \rangle &::= \{ \langle \text{sort-ident} \rangle \"\#\" \}+ \\
\langle \text{sets} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{t-variables} \rangle &::= \"\text{variables}\" \langle \text{t-variable-list} \rangle
\end{align*}
\]
The principles of TSF are as follows. There are two kinds of modules: data modules, as usual, and test modules, similar to process modules. Tests can communicate with processes (supposed to be specified elsewhere, say in PSF), but there is no merge operator. Two constructs need explanation:

- **verdict**(t). The term t can be used for indicating 'pass' or 'fail' values etc. It is assumed that the values themselves are defined in a special data module (e.g. called Verdict) which exports a sort VERDICT, a constant none and a binary function update from VERDICT # VERDICT to VERDICT. There is a built-in assignable variable called result. Its initial value is none. When using **verdict**(t), the type of t must be VERDICT. The meaning of **verdict**(t) is that result is assigned the value update(t, r'), where r' denotes the value of result in the previous state.

- **x** ⊸ **y**. This means that x is tried first, and only if x can make no step (e.g. because no matching communication is offered), then y is done. The operator ⊸ is called preferential alternative composition because it behaves like +, but if possible it chooses the first option amongst its alternatives.

At the syntactic level, there is no + operator. In order to keep things simple, we forbid tests communicating with each other. The top level merge...
is never written down. If one envisages a combined PSF/TSF simulation environment, then the merge takes place.

The prescribed data module could be as follows, where it is understood that the sort \texttt{VERDICT} and the functions \texttt{none} and \texttt{update} are obligatory, whereas \texttt{pass}, ..., \texttt{error} etc. can be added by the user. The rules for processing a given verdict imply that a verdict can be issued only once (later we shall present alternative data modules for \texttt{VERDICT}).

\begin{verbatim}
data module FinalVerdict begin
  exports
  begin
    sorts
      VERDICT
    functions
      none      : -> VERDICT
      update    : VERDICT # VERDICT -> VERDICT
      pass      : -> VERDICT
      fail      : -> VERDICT
      inconclusive : -> VERDICT
      error     : -> VERDICT
    end
  variables
    r : -> VERDICT
  equations
    -- getting started
    [00] update(r,none) = r
    [01] update(none,r) = r
    -- propagating errors
    [02] update(r,error) = error
    [03] update(error,r) = error
    -- processing multiple final verdicts
    [04] update(pass,pass) = error
    [05] update(pass,fail) = error
    [06] update(pass,inconclusive) = error
    [07] update(fail,pass) = error
    [08] update(fail,fail) = error
    [09] update(fail,inconclusive) = error
    [10] update(inconclusive,pass) = error
    [12] update(inconclusive,inconclusive) = error
  end FinalVerdict
\end{verbatim}
5 TSF semantics

5.1 Outline

We claim that ACP offers the features to interpret the new constructs.

- $\Lambda_\rho$ to describe the state-based concept of current ‘result’ value.
- $\theta$ and $+$ for interpreting $x \leftrightarrow y$.

Some care is needed with respect to the use of $\theta$. Consider the test $r(a) \leftrightarrow r(b) \leftrightarrow r(c)$ having three alternatives. Classically one would define $\gamma(s(n), r(n)) = c(n)$ etc. Now we must label the occurrences of receive actions, yielding $r^0(a)$, $r^1(b)$ and $r^2(c)$ according to their syntactic position as an argument of $\leftrightarrow$. Let us call this labeling function $\psi$. We label verdict statements too. Labelling starts at zero. We must adopt a $\gamma$ which preserves the labeling: $\gamma(s(n), r^i(n)) = c^i(n)$. We forbid things like $\gamma(s^i(n), r^j(n))$.

Now we can order the $c^i(n)$ such that $c^0(n) > c^1(n) > c^2(n)$. In this way we find that the ACP interpretation of the test is $r^0(a) + r^1(b) + r^2(c)$.

The translation is non-compositional in the sense that an implementation $I$ and a test $T$, each being an ACP term, must be combined together in the scope of the $\theta$ operator.

$$\theta(\partial_H(I \parallel \Lambda_\rho(\psi(0, T))))$$

Here $\sigma$ is the initial state which maps result to none and $H$ is the set which hides the $r$ and $s$ actions but lets the $c$ actions pass (as usual). So $\theta$ selects amongst the $c$ actions.

5.2 Details

In this section we shall formalize the essential steps outlined in Section 5.1. The ACP terms are as usual, but some of the atomic actions can be labeled with numbers. Furthermore we allow for guarded action of the form $[e] \rightarrow x$, where the guard $e$ is an equation between data terms. These guarded actions are precisely as in PSF (version 1.1 and up). For finite data domains, the following two axioms allow for elimination of guards:

$$[t_1 = t_2] \rightarrow x = x \quad \text{if } t_1 = t_2 \text{ holds}$$

$$[t_1 = t_2] \rightarrow x = \delta \quad \text{if } t_1 \neq t_2 \text{ holds}$$

When dealing with infinite data domains, the terms $t_1$ and $t_2$ may contain bound variables, as for example $d$ in $\sum_{d \in D}[d = 0] \rightarrow x(d)$. This means that the applicability of the above two axioms is restricted to terms in which all such variables have become bound.

First we must describe the function $\psi$ which maps instances of the non-terminal <test> to ACP terms. In this definition, the set of atomic actions of TSF, with typical element $a$, is the set containing $\text{skip}$, $\text{verdict}(v)$ for...
all $v$ of type VERDICT and all instances of the non-terminal <atom>. Furthermore, $e$ is an arbitrary equation, that is, an instance of the non-terminal <equation>.

$$ψ(n, a) = a^n \quad \text{(for atomic action $a$)}$$

$$ψ(n, [e]→ x) = [e] → ψ(n, x)$$

$$ψ(n, \sum (d \in D, x(d))) = \sum_{d \in D} ψ(n, x(d))$$

$$ψ(n, x . y) = ψ(n, x) · ψ(0, y)$$

$$ψ(n, x +> y) = ψ(n, x) + ψ(n + ϕ(x), y)$$

where we use an auxiliary function $ϕ$ which returns as $ϕ(x)$ the number of fresh labels issued when labeling $x$, that is, when mapping $x$ to $ψ(n, x)$.

$$ϕ(a) = 1 \quad \text{(for atomic action $a$)}$$

$$ϕ([e]→ x) = ϕ(x)$$

$$ϕ(\sum (d \in D, x(d))) = 0 \quad \text{(if $D = \emptyset$)}$$

$$ϕ(x . y) = ϕ(x)$$

$$ϕ(x +> y) = ϕ(x) + ϕ(y)$$

The <definitions> clause of a TSF specifications contains a number of defining equations for named tests. It is understood that the function $ψ(0, ...)$ is applied to the right-hand side of each instance of <definition>. We must restrict ourselves to defining equations which are guarded in the sense that each recursive call is preceded by at least one atomic action. Therefore $ψ$ need not be applied to recursive calls, that is, $ψ(n, T) = T$ if $T$ is an instance of <simple-test>.

Next we must give the definition of the communication function $γ$, which will be derived from the <communications> clause of the TSF text under consideration. Syntactically, the <communications> clause is the same as in PSF, but whereas in PSF it can be directly interpreted as a definition of $γ$, for TSF we should now explain precisely the way in which the labeling passes through $γ$. The <communications> clause consists of a number of rules and we consider the following typical form for such rule:

$$r_0(n) \mid r_1(n) = c(n) \text{ for } n \in D$$

For example, $r_0(n)$ could be a sending atom and $r_1(n)$ a matching receive action. We must adopt a restriction for the arguments of $\mid$ in each rule: one argument must be an atom declared in the TSF text, whereas the other one must be declared in an imported PSF process module. Each rule satisfying this restriction gives rise to the following equations for $γ$:

$$γ(r_0(n)^m, r_1(n)) = c(n)^m \quad \text{(all } m \in \mathbb{N}, n \in D)$$

$$γ(r_0(n), r_1(n)^m) = c(n)^m \quad \text{(all } m \in \mathbb{N}, n \in D)$$

Then it is understood that $γ$ is defined by the conjunction of these equations, combining the results of all rules in the <communications> clause, together with the following two closure clauses:
\[ \gamma(a,b) = \gamma(b,a) \]
\[ \gamma(a,b) = \text{undefined} \quad \text{if } \gamma(b,a) \text{ defined} \]
\[ \gamma(a,b) = \text{undefined} \quad \text{otherwise} \]

We work in ACP, and then the process algebra axioms describe how \( \gamma \) is extended to the usual \( \mid \) between arbitrary terms.

Next we must give the definition of the encapsulation operator \( \partial_H \), which requires the derivation of a set \( H \) from the <communications> clause of the TSF text under consideration. We define \( H \) as the union of sets

\[ \{r(n) | n \in D\} \cup \{r(n)^m | n \in D, m \in \mathbb{N}\} \]

where \( r \) ranges over all atom identifiers, assuming that \( D \) is the type of \( r \), which occur in the left hand side of one or more of the rules of the <communications> clause.

Next we must give the definition of the priority ordering \( < \) on the set of atomic actions, which is required for the proper application of the priority operator \( \theta \).

\[ a^m < b^n \text{ iff } m > n \quad \text{(all } m, n \in \mathbb{N}\)\]

for all atomic actions \( a, b \in \{c(n) | n \in D\} \cup \{\text{skip}\} \cup \{\text{verdict}(v) | v \in \text{VERDICT}\} \). In this way an action which appears first in a list of alternatives with respect to \( +> \), and which thus has the lowest label, gets the highest priority.

Finally we give the definition of the functions 'action' and 'effect', which are required for the proper application of the generalized state operator \( \Lambda \). We describe them for all communications actions \( r \) occurring in the left hand side of one or more of the rules of the <communications> clause and also for skip and verdict\((v)\). The set of states, with typical element \( s \), is defined as the set of functions which maps our only variable, 'result', to elements of the set \( \text{VERDICT} \).

\[
\begin{align*}
\text{action}(r(n)^m, \sigma) & = r(n)^m \\
\text{effect}(r(n)^m, \sigma, r(n)^m) & = \sigma \\
\text{action}(& \text{skip}^m, \sigma) = \text{skip}^m \\
\text{effect}(\text{skip}^m, \sigma, \text{skip}^m) & = \sigma \\
\text{action}(& \text{verdict}(v)^m, \sigma) = \text{verdict}(v, \text{update}(v, \sigma(\text{result})))^m \\
\text{effect}(\text{verdict}(v)^m, \sigma, \text{verdict}(v, v')^m) & = \sigma[v'/\text{result}]
\end{align*}
\]

In this way the \( \Lambda \) operator lets most actions pass unchanged, except for the verdicts, which are processed in two ways: first, whenever a verdict is executed, it leaves a term \( \text{verdict}(v, v') \) in the resulting trace, where \( v \) is the stated verdict, and \( v' \) is the accumulated effect of this and all previous verdicts, as accumulated in \( \sigma \). Secondly, the state \( \sigma \) is updated. Please note that we use some overloading: the binary verdict occurs in the resulting traces, whereas the single-valued verdicts, roughly speaking, are the verdict statements written in the PSF text.

If the test developer or executer is only interested in the final outcome of the test, he can take the second argument \((v')\) of the last verdict term in the execution trace.
5.3 Algebraic properties

The following laws should hold on the basis of the intended application and the given intuition for the preferential alternative composition.

\[
\begin{align*}
  x \rightarrow \delta & = x \\
  \delta \rightarrow x & = x \\
  x \rightarrow (y \rightarrow z) & = (x \rightarrow y) \rightarrow z \\
  (x \rightarrow y) \cdot z & = x \cdot z \rightarrow y \cdot z
\end{align*}
\]

For the model given in Section 5.2 where TSF tests are mapped by \( \psi \) to ACP terms, these laws hold (see the theorem below). But it would be interesting to have other models for \( \rightarrow \) which abstract away from the particularities of our labeling function \( \psi \).

**Theorem.** For \( x, y \) and \( z \) instances of the nonterminal \(<\text{test}>\) and identifying \( \sum u \) with \( \delta \) for all \( u \), the following hold:

1. \( \text{ACP}, \vdash \psi(n, x \rightarrow \delta) = \psi(n, x) \)
2. \( \text{ACP}, \vdash \psi(n, \delta \rightarrow x) = \psi(n, x) \)
3. \( \text{ACP}, \vdash \psi(n, x \rightarrow (y \rightarrow z)) = \psi(n, (x \rightarrow y) \rightarrow z) \)
4. \( \text{ACP}, \vdash \psi(n, (x \rightarrow y) \cdot z) = \psi(n, x \cdot z \rightarrow y \cdot z) \)

**Proof** We use the definitions of \( \psi \) and \( \varphi \) together with some of the process algebra laws \((A1,A2,A4,A6)\).

1. \( \psi(n, x \rightarrow \delta) \)
   \[
   = \psi(n, x) + \psi(n + \varphi(x), \delta) \\
   = \psi(n, x) + \delta =_{A6} \psi(n, x).
   \]
2. \( \psi(n, \delta \rightarrow x) \)
   \[
   = \psi(n, \delta) + \psi(n + \varphi(\delta), x) \\
   = \delta + \psi(n + 0, x) \\
   = \delta + \psi(n, x) =_{A1,A6} \psi(n, x).
   \]
3. \( \psi(n, x \rightarrow (y \rightarrow z)) \)
   \[
   = \psi(n, x) + \psi(n + \varphi(x), y \rightarrow z) \\
   = \psi(n, x) + (\psi(n + \varphi(x), y) + \psi(n + \varphi(x) + \varphi(y), z)) \\
   =_{A2} (\psi(n, x) + \psi(n + \varphi(x), y)) + \psi(n + \varphi(x) + \varphi(y), z) \\
   = \psi(n, x \rightarrow y) + \psi(n + \varphi(x \rightarrow y), z) = \psi(n, (x \rightarrow y) \rightarrow z).
   \]
4. \( \psi(n, (x \rightarrow y) \cdot z) \)
   \[
   = \psi(n, x \rightarrow y) \cdot \psi(0, z) \\
   = (\psi(n, x) + \psi(n + \varphi(x), y)) \cdot \psi(0, z) \\
   =_{A4} \psi(n, x) \cdot \psi(0, z) + \psi(n + \varphi(x), y) \cdot \psi(0, z) \\
   = \psi(n, x \cdot z) + \psi(n + \varphi(x \cdot z), y \cdot z) = \psi(n, x \cdot z \rightarrow y \cdot z).\]

\(\square\)
6 Example in TSF

6.1 TSF test of the buffer

The test example already discussed in Section 3 can be cast into the TSF format. We use the data module Naturals from the PSF standard library. The data module Verdict is given in the appendix (FinalVerdict as given before would do well too). The process module OnePlaceBuffer models the IUT. It could be specified in PSF (see Section 6.2), or it could be some real IUT.

```plaintext
test module TestOnePlaceBuffer
begin
exports
begin
atoms
  s-DU : NATURAL
  r-DL : NATURAL
tests
  TST1
end
imports
Naturals, Verdict, OnePlaceBuffer
tests
TST1
communications
  s-DU(n) | r-DU(n) = DU(n) for n in NATURAL
  s-DL(n) | r-DL(n) = DL(n) for n in NATURAL
definitions
TST1 = s-DU(zero)
  . sum( n in NATURAL
      , [n=zero] -> r-DL(n) . verdict(pass)
      +> r-DL(n) . verdict(fail)
    )
end TestOnePlaceBuffer
```

6.2 Formal analysis of the tests

Using a a correct implementation of a One-place-buffer, we show that the test above leads to the expected answer \( \text{pass} \). We start from the following PSF specification of the IUT, which is called \( \text{BUF} \). 
process module OnePlaceBuffer

begin
exports
begin
atoms
r-DU : NATURAL
s-DL : NATURAL
processes
BUF
end

imports
Naturals
processes
BUF
definitions
BUF = sum(n in NATURAL, r-DU(n) . s-DL(n)) . BUF
end OnePlaceBuffer

First of all we present BUF as an equation in process algebra. The data set $D$ equals $\mathbb{N}$, as can be concluded from an analysis of Naturals.

$$BUF = \sum_{n \in D} (r-DU(n) \cdot s-DL(n)) \cdot BUF$$

Next we interpret the test module TestOnePlaceBuffer as an equation in process algebra. We apply the labeling function $\psi$ immediately, finding the following translation for the definition of TST1.

$$\psi(0, TST1) = s-DU(0)^0 \cdot \sum_{n \in D} ([n = 0] \rightarrow r-DL(n)^0 \cdot \text{verdict(pass)}^0 + r-DL(n)^1 \cdot \text{verdict(fail)}^0$$

The initial state $\sigma$ is given by $\sigma = (\text{result} \mapsto \text{none})$. Next we can derive our communication function $\gamma$.

$$\gamma(s-DU(n)^m, r-DU(n)) = DU(n)^m \quad (\text{all } m \in \mathbb{N}, n \in D)$$
$$\gamma(s-DU(n), r-DU(n)^m) = DU(n)^m \quad (\text{all } m \in \mathbb{N}, n \in D)$$
$$\gamma(s-DL(n)^m, r-DL(n)) = DL(n)^m \quad (\text{all } m \in \mathbb{N}, n \in D)$$
$$\gamma(s-DL(n), r-DL(n)^m) = DL(n)^m \quad (\text{all } m \in \mathbb{N}, n \in D)$$

And of course we have the corresponding equations with swapped arguments of $\gamma$. The set $H$ contains all actions of the form $s-DU(n)$, $r-DU(n)$, $s-DL(n)$, $r-DL(n)$ together with all labeled versions of these actions. After these preparations we are ready to perform a calculation and check whether BUF will pass the test TST1. We write simply none for the state $\sigma$ such that
\( \sigma(\text{result}) = \text{none} \) and similarly for \text{pass} and \text{fail}. We have to use some shorthands: \( v \) abbreviates verdict, \( p \) abbreviates \text{pass} and \( f \) abbreviates \text{fail}.

\[
\begin{align*}
\theta(\partial_H(\text{BUF} \parallel \Lambda_{\text{none}}(\psi(0, \text{TST1})))) &= \\
\theta(\partial_H(\text{BUF} \parallel s - \text{DU}(0)^0 \cdot (\Lambda_{\text{none}}(\sum_n ([n = 0] \to \ldots + \ldots)))))) &= \\
\theta(\text{DU}(0)^0 \cdot \partial_H((s - \text{DL}(0) \cdot \text{BUF}) \parallel \Lambda_{\text{none}}(\sum_n ([n = 0] \to \ldots + \ldots)))) &= \\
\text{DU}(0)^0 \cdot \theta(\partial_H((s - \text{DL}(0) \cdot \text{BUF}) \parallel (\sum_n \Lambda_{\text{none}}([n = 0] \to r - \text{DL}(n)^0 \cdot v(p)^0) \\
+ \sum_n \Lambda_{\text{none}}(r - \text{DL}(n)^1 \cdot v(f)^0))) &= \\
\text{DU}(0)^0 \cdot \theta(\partial_H((s - \text{DL}(0) \cdot \text{BUF}) \parallel (\sum_n \Lambda_{\text{none}}(r - \text{DL}(0)^0 \cdot v(p)^0) \\
+ \sum_n \Lambda_{\text{none}}(r - \text{DL}(0)^1 \cdot v(f)^0))) &= \\
\text{DU}(0)^0 \cdot \theta(\text{DL}(0)^0 \cdot \partial_H(\text{BUF} \parallel \Lambda_{\text{none}}(v(p)^0)) \\
+ \text{DL}(0)^1 \cdot \partial_H(\text{BUF} \parallel \Lambda_{\text{none}}(v(f)^0))) &= \\
\text{DU}(0)^0 \cdot \theta(\text{DL}(0)^0 \cdot \partial_H(\text{BUF} \parallel (v(p, \text{update}(p, \text{none})) \cdot \Lambda_{\text{pass}}(\epsilon)))) &= \\
\text{DU}(0)^0 \cdot \text{DL}(0)^0 \cdot v(p, p) &= \\
\text{DU}(0)^0 \cdot \text{DL}(0)^0 \cdot \text{verdict(pass, pass)}
\end{align*}
\]

To get the final outcome of the test, we take the second argument of the last \text{verdict} term in the execution trace, which is \text{pass}.

When aiming at a thorough analysis of the test, this is only half the job. We should also compose the test with one or more incorrect IUTs and check if the test yields fail for at least one of them.

### 7 Conclusions

It is nice that the entire mechanism of verdict updating can be captured by an algebraic data module, as demonstrated in Section 4 (final verdicts only), and Appendices A (preliminary verdicts only) and B (the full TTCN mechanism).

Please note that tests need not be deterministic, nor should the combinations of a test and an implementation under test (IUT) be deterministic. Yet we expect that typical tests are written in such a style that the TSF text exorcizes much of the non-determinism in the IUT. In many cases (as in Section 6.2), the combination of a test and a correct IUT will lead to a unique verdict (which must be \text{pass}). Even if the outcome is not unique, a
correct test when combined with a correct IUT may lead only to values pass or inconclusive.

In this paper we have focused on semantical issues, interpreting things in process algebra terms. This is because we believe that it is of prime importance to have the semantics of a new language as clear as possible. Of course we considered adding more features to the language or making it more TTCN-like, but we restricted ourselves to two new language constructs only (+> and verdict) in order to avoid additional semantic complications.

In practice, one might consider simulating tests together with specifications of correct and incorrect IUTs rather than performing a manual analysis in process algebra terms. The following tools could be useful: a simulator (in combination with PSF), and translators from and to TTCN. Typically one would develop tests in TTCN, translate them to TSF and check them against PSF specifications of the IUT. Of course one can also work the other way around: start from TSF and derive the TTCN from that, or avoid TTCN altogether. In this way TSF has the same relation to PSF as the relation between TTCN and SDL. For TTCN some tools available, notably the ITEX compiler of Telelogic, who make also an SDL design tool SDT.

Arie van Deursen has already constructed a syntax-directed editor and checker using the ASF+SDF toolset ([10], [11]). This revealed one priority conflict, which was easily resolved. No other difficulties in the concrete syntax have been encountered and the generated tool has already been used to check small examples, including the example of Section 6. The various data modules describing verdicts have also been typechecked using the PSF toolset.

The newly introduced operator +> deserves further study, because we only gave one model, and it is interesting to have other and more abstract models.

Acknowledgements: The authors wish to thank Jos Baeten, Jan Bergstra, Sjouke Mauw, Hans Mulder, Yat Man Lau, Jennifer Witham and Arie van Deursen for the discussions and the help that contributed to the work presented in this paper.

References


A Preliminary verdicts

These are verdicts like TTCN's preliminary verdicts. They are useful when employing an incremental approach to establishing the test result. No final verdicts or errors verdicts are needed. There is an ordering principle: a verdict can never get better than it already was, e.g. a preliminary fail verdict can never be changed in a preliminary pass verdict.

```plaintext
data module PreVerdict
begin
exports
begin
sorts
  VERDICT
functions
  none : , -> VERDICT
  update : VERDICT # VERDICT -> VERDICT
  pass : -> VERDICT
  fail : -> VERDICT
  inconclusive : -> VERDICT
end
variables
```
\( r : \rightarrow \text{VERDICT} \)

equations

-- getting started

\[00\] update\( (r, \text{none}) = r \)
\[01\] update\( (\text{none}, r) = r \)

-- processing (preliminary) verdicts

\[02\] update\( (\text{pass}, \text{pass}) = \text{pass} \)
\[03\] update\( (\text{pass}, \text{fail}) = \text{fail} \)
\[04\] update\( (\text{pass}, \text{inconclusive}) = \text{inconclusive} \)

\[05\] update\( (\text{fail}, \text{pass}) = \text{fail} \)
\[06\] update\( (\text{fail}, \text{fail}) = \text{fail} \)
\[07\] update\( (\text{fail}, \text{inconclusive}) = \text{fail} \)

\[08\] update\( (\text{inconclusive}, \text{pass}) = \text{inconclusive} \)
\[09\] update\( (\text{inconclusive}, \text{fail}) = \text{fail} \)
\[10\] update\( (\text{inconclusive}, \text{inconclusive}) = \text{inconclusive} \)

end PreVerdict

B  The full TTCN set of verdicts

These are verdicts like TTCN's final and preliminary verdicts. The idea is that a final verdict (pass, fail, inconclusive) can only be given once. Therefore updating them leads to an error. The preliminary verdicts (pre-pass, pre-fail, pre-inconclusive) can be updated however, provided this updating is done in a consistent way.

data module Verdict
begin
exports
begin
sorts
  VERDICT
functions
  none : \rightarrow \text{VERDICT}
  update : \text{VERDICT} \# \text{VERDICT} \rightarrow \text{VERDICT}
  pre-pass : \rightarrow \text{VERDICT}
  pre-fail : \rightarrow \text{VERDICT}
  pre-inconclusive : \rightarrow \text{VERDICT}
  pass : \rightarrow \text{VERDICT}
  fail : \rightarrow \text{VERDICT}
  inconclusive : \rightarrow \text{VERDICT}
  error : \rightarrow \text{VERDICT}
end
variables
\[ r : \rightarrow \text{VERDICT} \]

\text{equations}

\text{-- getting started}

[00] update(r,none) = r
[01] update(none,r) = error

\text{-- propagating errors}

[02] update(r,error) = error
[03] update(error,r) = error

\text{-- processing preliminary verdicts}

[04] update(pre-pass,pre-pass) = pre-pass
[05] update(pre-pass,pre-fail) = pre-fail
[06] update(pre-pass,pre-inconclusive) = pre-inconclusive
[07] update(pre-pass,pass) = error
[08] update(pre-pass,fail) = error
[09] update(pre-pass,inconclusive) = error

[10] update(pre-fail,pre-pass) = pre-fail
[12] update(pre-fail,pre-inconclusive) = pre-fail
[13] update(pre-fail,pass) = error
[14] update(pre-fail,fail) = error
[15] update(pre-fail,inconclusive) = error

[16] update(pre-inconclusive,pre-pass) = pre-inconclusive
[17] update(pre-inconclusive,pre-fail) = pre-fail
[18] update(pre-inconclusive,pre-inconclusive) = pre-inconclusive
[19] update(pre-inconclusive,pass) = error
[20] update(pre-inconclusive,fail) = error
[21] update(pre-inconclusive,inconclusive) = error

\text{-- processing final verdicts}

[22] update(pass,pre-pass) = pass
[23] update(pass,pre-fail) = error
[24] update(pass,pre-inconclusive) = error
[25] update(pass,pass) = error
[26] update(pass,fail) = error
[27] update(pass,inconclusive) = error

[28] update(fail,pre-pass) = fail
[29] update(fail,pre-fail) = fail
[30] update(fail,pre-inconclusive) = fail
[31] update(fail,pass) = error
update(fail,fail) = error
update(fail,inconclusive) = error

update(inconclusive,pre-pass) = inconclusive
update(inconclusive,pre-fail) = error
update(inconclusive,pre-inconclusive) = inconclusive
update(inconclusive,pass) = error
update(inconclusive,fail) = error
update(inconclusive,inconclusive) = error

end Verdict