3

TTCN test case correctness validation

Finn Kristoffersen* Thomas Walter†

Abstract

We define a framework for the validation of TTCN test cases against SDL specifications. An essential component of this framework is a semantical model *Common Semantics Representation* (CSR) for concurrent TTCN and basic SDL, which defines an operational semantics for both languages. We propose an instantiation of the validation framework. Starting from a general notion of test case correctness based on a relation between behaviour of a test case and protocol specification, we focus on one particular correctness criterion. We show how this correctness criterion can be formalised in terms of the CSR. A methodology to prove TTCN test cases correct with respect to this criterion is outlined.

1 Introduction

OSI conformance testing [1] is based on the assumption that protocol implementations are tested against standardized test suites. In the past, test suites were manually defined without tool support [2], and quite often errors were found in these test suites.

Meanwhile this situation has slightly changed. Tools are now available that can check test cases for syntactical and static semantical correctness as defined for the *Tree and Tabular Combined Notation* (TTCN) [3]. However, these tools do not check a test suite for dynamic correctness, i.e. they do not perform a validation of test cases against the behaviour expressed in a protocol specification. Second, tool supported methods for test case generation from formal specifications are still not widely available and used.

Besides these reasons our work on test case validation has been motivated by the following observations: First, a number of test suites were standardised and later a specification of the protocol in SDL were produced. Second, in some cases test suites have been derived from an SDL specification but only using *manual* methods. Thus test case validation is still an issue that can help to improve test suite quality.

Although several aspects are to be covered by test case validation this paper focuses on correctness validation of test case behaviour with respect to system behaviour of an SDL specification. To perform such a validation, a model is needed sufficiently powerful to permit comparison of behaviour of a test case and an SDL specification.

*Tele Danmark Research, Lyngsø Allé 2, 2970 Hørsholm, Denmark, e-mail: finn@tdr.dk

†Computer Engineering and Networks Laboratory, ETH Zürich, 8092 Zürich, Switzerland, e-mail: walter@tik.ethz.ch

T. Mizuno et al. (eds.), *Protocol Test Systems*

We define a validation framework in which the essential components are a semantical model for SDL [4] and concurrent TTCN [3, 5], and an appropriate behaviour relation. The model called *Common Semantics Representation* (CSR) [6, 7, 8] defines an operational semantics for both languages providing an abstract execution model which allows to 'execute' a test case and a specification and to compare their behaviours.

The outline of the paper is as follows. Section 2 introduces the main features of the CSR. In Section 3 the framework for test case validation is defined and a possible instantiation proposed. The conclusions are given in Section 4.

2 CSR - a common semantical model

The *Common Semantics Representation* (CSR) is a semantical model for SDL'92[1] and concurrent TTCN[2]. *Concurrent TTCN* extends TTCN. The concern of TTCN is a single test component executing a test case. Concurrent TTCN allows several test components running in parallel to execute a test case. A system[3] in the CSR is structured into a set of hierarchically ordered entities (Fig. 1). The CSR is *compositional* in the sense that properties of higher level entities are expressed in terms of properties of lower level entities. The compositionality of the model enables analysis of (system) behaviour at different hierarchical levels.

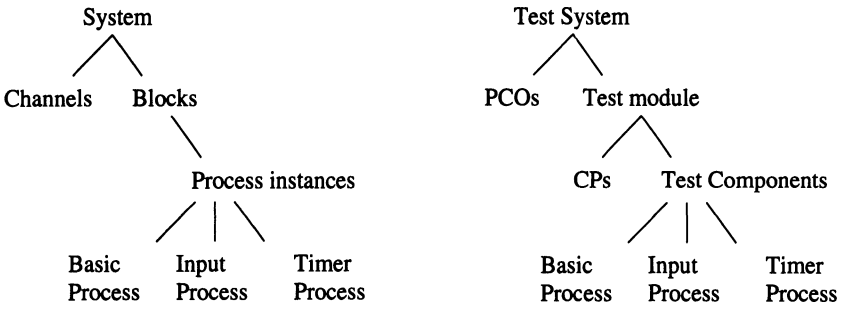

Figure 1: The CSR structure for SDL specifications and TTCN test cases

2.1 The CSR entities

The main CSR entities are *system, modules, links,* and *process instances.* A process instance is further decomposed into a *basic process,* an *input process,* and a *timer process.*

Basic process: A basic process is the control part of the behaviour description of a process instance. The behaviour description is given as a sequence of *atomic events*: events for input and output of messages, control of timers, manipulation of data, etc.

[1]Basic SDL'92 (or SDL for short) is covered except for channels with no delay and service definitions.
[2]If not stated otherwise, *TTCN* always refers to concurrent TTCN.
[3]If no further distinction is made then *system* refers to an SDL system or a TTCN system.

A basic process is defined over a *basic process algebra* (BPA). The BPA defines a set of operators to compose atomic events into behaviour expressions. The operators include *event prefixing*, denoted ';', *choice*, denoted '+', and *priority choice* denoted '⊕'. The priority choice operator is introduced to model the evaluation order of sets of alternatives in TTCN.

Input process: An input process consists of two queue-like data structures: one for the intermediate storage of received messages and one for the storage of *saved* messages. When messages are received they are appended to the first queue. The temporal order of message receptions is preserved in that queue.

In case of SDL, an input process models the *input port* and a list of currently *saved* signals. In case of TTCN, an input process models all input queues of all *points of control and observation* (PCO) and *coordination points*[4] (CP) of a test component.

Timer process: A timer process maintains a list of *active timers* in SDL and *running timers* in TTCN ordered with respect to their time-out values. Expired timers are transferred to the input process. The first timer in the list is the one to expire next.

Process instance: Besides a basic process, an input process, and a timer process, a process instance comprises an environment function that maps variables to memory locations and a storage function that maps memory locations to values. In the CSR model, a process instance is related to a *process instance* in SDL and to a *test component* in TTCN.

Links: Links are unlimited first-in-first-out message queues. A link is a means to connect process instances, module processes, or module processes to the environment. Links introduce an arbitrary but finite delay on conveyed messages. Channels in SDL and PCOs and CPs of TTCN are modeled by links. Since SDL signal routes do not introduce a delay on signals conveyed, they need not be modelled explicitly. As CPs are used to connect test components, they are part of (test) modules in the CSR structure for TTCN (Fig. 1).

Modules: A module process is a collection of process instances. Referring to SDL, a module process relates to a *block*. For TTCN, a module process comprises all test components defined in a test configuration. This implies that a single test module is sufficient to model a TTCN test case.

System: A system is an entity which interacts with the environment. Interactions are performed at the interfaces of a system via links.

The system entity is used to model an SDL system or a *TTCN system*, i.e. an entity that sends and receives abstract service primitives (ASPs) and protocol data units (PDUs) to and from the environment via PCOs. The system entity abstracts from the distribution of test components over test modules and the possible distribution over (real) computer systems in a network.

[4] *Coordination points* are first-in-first-out queues connecting test components. They convey coordination messages exchanged between test components to coordinate their joint behaviour.

The CSR defines a discrete time model. Every process instance maintains a local clock which is periodically updated to hold the current global time.

The CSR model assumes an abstract model for data, called *many-sorted algebra* [9]. A *signature* Σ is given for all data types defined in an SDL specification or a TTCN test suite. The interpretation of signature Σ is given by a Σ-algebra. The chosen data model is close to the one used in SDL. A mapping of TTCN data type definitions and value expressions (including ASN.1 data type definitions [10]) is given in [6].

The translation of a TTCN test case or an SDL specification into the CSR result in an instantiation of the low level CSR entities and the data domains. The initialized entities are the links, the basic process instances, the input processes, and the timer processes. The basic processes are derived from the process bodies of SDL processes and the behaviour trees of TTCN test cases, timer processes are initialized with an empty list of active timers, and the queues of the input processes are empty. Links are initialized with an empty message queue.

2.2 An operational semantics for SDL'92 and TTCN

The CSR defines an operational semantics for SDL'92 and concurrent TTCN in terms of *Labelled Transition Systems* (LTS) [11]. Non-determinism is modelled as branching of the LTS and parallelism is modelled as arbitrary interleaving of events. For each entity in the CSR, its operational semantics is defined by an LTS.

Definition 1 A *Labelled Transition System* (LTS) is a tuple

$$(S, \epsilon, \rightarrow, s_0)$$

where S is a set of states, ϵ is a set of atomic events, and $\rightarrow \subseteq S \times \epsilon \times S$ is a transition relation, and s_0 is an initial state. An element in $\rightarrow$ is termed *transition* and is written $s \xrightarrow{e} s'$ for $s, s' \in S$, $e \in \epsilon$, and $(s, e, s') \in \rightarrow$. □

Example 1 As an example consider the LTS of a process instance [6]:

$$PI = (Proc, \epsilon_{\mathsf{PI}}, \rightarrow_{\mathsf{PI}})^5 \tag{1}$$

where $Proc$ is the set of states of a process instance. A state of the process instance is composed from a state of a basic process together with an environment and a storage that represents a set of values of defined variables, and the states of the associated timer process and the input process.

The set of events a process instance may perform, denoted ϵ_{PI}, is defined over the following sets of events:

$$\epsilon_{\mathsf{PI}} = (Receive \cup Output) \cup Create \cup Terminate \cup \{\tau\} \tag{2}$$

where the events belonging to each of these sets are defined by the following properties.

[5] Indices are used to distinguish between LTSs of different entities. If no initial state is given then the semantics of an entity is given by a family of LTSs.

- A process instance performs a *receive event* ($\in$ *Receive*) either if a message is received from another process instance in the same module process or if it is received from a link process. The received message is stored in the input process. A *receive event* models the reception of a signal instance in SDL or the reception of an ASP, PDU or *coordination message* (CM) in TTCN.
- A process instance performs an *output event* ($\in$ *Output*) if a message is sent to another process instance. An *output event* corresponds to the sending of a signal instance in SDL or to the sending of an ASP, PDU or CM in TTCN.
- A process instance performs a *create event* ($\in$ *Create*) if a process instance creates a new process instance in the same module. A *create event* models creation of a process instance or a test component.
- A *terminate event* represents the termination of a test component.
- τ is the *internal event* and models the execution of an event not observable in the environment of the entity, e.g. an assignment statement.

Finally, $\rightarrow_{\mathsf{PI}} \subseteq Proc \times \epsilon_{\mathsf{PI}} \times Proc$ is the transition relation associated with a process instance. Usually, a transition relation like $\rightarrow_{\mathsf{PI}}$ is not defined by enumerating all transitions: instead a set of inference rules for transitions is given [11, 12]. □

The transition relation may be extended to event sequences.

Definition 2 For $(S, \epsilon, \rightarrow, s)$ an LTS.

1. ϵ^* is the set of event sequences over events from ϵ. An element in ϵ^* is called *trace*.
2. Let $\sigma = e_1 \ldots e_n \in \epsilon^*$ and $s, s' \in S$. Then s can evolve into s' by σ, $s \xrightarrow{\sigma} s'$, if $\exists s_0, \ldots, s_n \in S$ such that $s_{i-1} \xrightarrow{e_i} s_i$ for $i = 1, \ldots, n$ and $s = s_0$ and $s' = s_n$.
   $s \xrightarrow{\sigma}$ means that $\exists s' \in S$ such that $s \xrightarrow{\sigma} s'$.

□

Definition 3 For $(S, \epsilon, \rightarrow, s)$ an LTS and $\sigma = e_1, \ldots, e_n$, $e_i \in \epsilon - \{\tau\}$, and $s, s', s_i, s_j, s_m, s_n \in S$

1. $s \stackrel{\sigma}{\Longrightarrow} s'$ if $s \xrightarrow{\tau^*} s_i \xrightarrow{e_1} s_j \xrightarrow{\tau^*} \ldots \xrightarrow{\tau^*} s_m \xrightarrow{e_n} s_n \xrightarrow{\tau^*} s'$
2. $s \stackrel{\sigma}{\Longrightarrow}$ if $\exists s' \in S$ such that $s \stackrel{\sigma}{\Longrightarrow} s'$
3. $Tr(s) = \{\sigma \in (\epsilon - \{\tau\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\}$ is the set of *traces of observable events* from state $s \in S$.

$\Longrightarrow$ abstracts from all internal events in sequences of events. □

2.2.1 The CSR - A compositional model

The compositionality of the CSR model is reflected in the definition of inference rules for transitions.

Definition 4 The generic form of an inference rule is $\frac{t_1 \ldots t_n}{t}\ C$ where $t_1, \ldots, t_n$, and t are transitions and C is an optional side-condition. Transitions $t_1, \ldots, t_n$ are termed premises and transition t is termed conclusion. □

The premises and conclusion may be transitions of different entities; therefore we term the model *compositional.*

Example 2 A process instance can perform a *receive event* provided that the input process can perform a *receive event.* This is defined by the following inference rule:

$$\frac{(\sigma, \sigma') \xrightarrow{\text{receive}(sig)}_{\mathsf{I}} (\sigma'', \sigma''')}{\langle P_{\varepsilon,\rho}, (\sigma, \sigma'), T\rangle \xrightarrow{\text{receive}(sig)}_{\mathsf{PI}} \langle P_{\varepsilon,\rho}, (\sigma'', \sigma'''), T\rangle} \quad \begin{array}{l} \| \, now \, \|_{\varepsilon,\rho} = time \\ To(sig) = \rho(\varepsilon(self)) \end{array} \tag{3}$$

where (σ, σ') and (σ'', σ''') denote states of the input process and *sig* is the message received by the input process. The states of process instance, $\langle P_{\epsilon,\rho}, (\sigma, \sigma'), T\rangle$ and $\langle P_{\epsilon,\rho}, (\sigma'', \sigma'''), T\rangle$, consist of a basic process instance P with associated environment ε and storage ρ, and T is the state of the timer process.

The rule is to be interpreted as follows: If it can be inferred that an input process can perform a *receive event* (premise of the above rule) by applying the inference rules defined for an input process, then the process instance can also perform a *receive event* provided the side-condition holds. That is, the value of the local clock of the process instance $\| \, now \, \|_{\varepsilon,\rho}$ must equal the current global time *time* and the message to be received should be addressed for the receiving process instance. The received message is stored in the input process. □

2.2.2 Observability in the CSR

The advantage of the CSR is that we do not have to limit observability to events exchanged at the system boundary but can also analyse internal events. Referring to Fig. 1, a system consists of link processes and module processes. A system state s is denoted by the states of all links and the states of all modules:

$$s = l_{lid_1} \times \ldots \times l_{lid_k} \times m_1 \times \ldots \times m_n \tag{4}$$

where a state of a link is a sequence of messages currently held by the link, and a state of a module is given by the states of all active process instances (SDL) or by the states of all coordination points and test components (TTCN).

If a system performs an event, e.g. an input message is received from the environment, then a state transition at the system level is performed. The LTS of the CSR system $(S_{\mathsf{S}}, \epsilon_{\mathsf{S}}, \rightarrow_{\mathsf{S}}, s)$ performs the transition

$$s' \xrightarrow{\text{input}(sig)}_{\mathsf{S}} s'' \tag{5}$$

where $s' = (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_k, m_1, \ldots, m_n)$ and $s'' = (\sigma_1, \ldots, sig \cdot \sigma_i, \ldots, \sigma_k, m_1, \ldots, m_n)$, i.e. signal *sig* is received by link i. Similarly, when an entity inside a system performs an event, this also results in a state transition at the system level

$$s' \xrightarrow{\tau}_{\mathsf{S}} s'' \tag{6}$$

E.g. $s' = (\sigma_1, \ldots, \sigma_k, m_1, \ldots, m_i, \ldots, m_n)$ and $s'' = (\sigma_1, \ldots, \sigma_k, m_1, \ldots, m_i', \ldots, m_n)$. This transition would be possible if the system level transition was due to module i performing

an internal event. As we have also access to lower level components of a system we can determine the process instance performing the internal event:

$$m_i \xrightarrow{\tau}_{\mathsf{M}} m_i' \tag{7}$$

If, in a TTCN test system, the state m_i of module i is $m_i = (\sigma_1, \ldots, \sigma_l, p_1, \ldots, p_i, \ldots p_q)$ then m_i' may be $(\sigma_1, \ldots, \sigma_l, p_1, \ldots, p_i', \ldots p_q)$, i.e. the transition in (7) is caused by a test component performing transition $p_i \xrightarrow{\tau}_{\mathsf{PI}} p_i'$. In a state description of a module σ_i, $i = 1, \ldots, l$, are states of links and p_j, $j = 1, \ldots, q$, are process instance states.

Again, in the CSR model we may decompose the behaviour further and determine the event that caused the transition of the test component. Assume that the initial state of test component p_i is $\langle P_{\epsilon,\rho}, (\sigma, \sigma'), T\rangle$ and the resulting state p_i' is $\langle P'_{\epsilon,\rho'}, (\sigma, \sigma'), T\rangle$ then the event performed might be an assignment. In this case, the basic process state P could be a behaviour expression like $P ::= R := \mathsf{Pass}; P'$ for some behaviour expression P'. Then the basic process may perform the assignment event $R := \mathsf{Pass}$, i.e. performs transition $P \xrightarrow{\mathsf{R} := \mathsf{Pass}}_{\mathsf{B}} P'$. Although at the process instance level an assignment is modelled as an internal event, we may determine the entity that performs an event which is modelled as an internal event at the process instance level.

3 Test case validation

Test case validation covers several aspects. A basic aspect is to check a TTCN test case for syntactic and static semantics correctness. If additional information is available then the validation process may check more properties. In [13], validation of the use of TTCN is considered based on heuristic rules. If a formal specification of the system is available the test case validation process may also cover e.g. analysing test suite coverage [14, 15]. The validation framework defined here is concerned with the correspondence between the dynamic behaviours of a test case and an SDL specification.

3.1 Preliminaries and limitations

For the sort of TTCN test case validation we aim at, the following assumptions are made. The CSR is used for the comparison of behaviours. However, we only consider test cases without unbounded recursion to ensure that every sequence of test events is finite. The CSR itself also implies limitations on the framework. Because the CSR models concurrency by interleaving, validation of real concurrent behaviour is not covered. Furthermore, because the CSR is based on a discrete time model validation of real time aspects is out of scope of the validation framework. This also implies that we consider traces equivalent if they differ only with respect to time events. We assume that the interface between an SDL system and a test system is empty which means that the two systems exchange events synchronously. This assumption is similar to the 'ideal test architecture' [16] where no test context, e.g. an underlying service provider, is assumed.

The TTCN test cases and SDL specifications are supposed to be syntactically and static semantically correct. Validation of a TTCN test suite is based on the validation of individual test cases because there is no higher level entity for which a dynamic semantics is defined, neither in [3] nor in the CSR.

The framework covers test case validation of valid behaviour and test cases with inopportune events when these messages are declared. The framework does not cover test cases that test for invalid behaviour using syntactically or semantically invalid test events since these events cannot be related to any SDL event.

3.2 The framework for test case validation

Let S denote the SDL specification against which the test case t is to be validated. The mapping functions from an SDL specification to the CSR and from a TTCN test case to the CSR are denoted $\mathcal{B}_S$ and $\mathcal{B}_T$ respectively. For a test case t and an SDL specification S we denote the CSR representations of mappings $\mathcal{B}_T(t)$ and $\mathcal{B}_S(S)$ by B_t and B_S.

The term *correctness criterion* denotes the requirements to be satisfied by a test case in order to be termed a *valid test case*. We define a correctness criterion as a binary relation R_C over the set of test cases $Spec_{TTCN}$ and SDL specifications $Spec_{SDL}$:

$$R_C \subseteq Spec_{TTCN} \times Spec_{SDL} \tag{8}$$

where index C may range over different correctness criteria, e.g. syntactic and static semantics correctness. However, because our emphasis is on dynamic behaviour validation and because the CSR is an appropriate model to represent the behaviour of systems, we seek for a refinement of correctness criterion R_C in terms of the CSR:

$$\leq_C \subseteq CSR_{TTCN} \times CSR_{SDL} \tag{9}$$

where $\leq_C$ denotes a refined definition of criterion R_C.

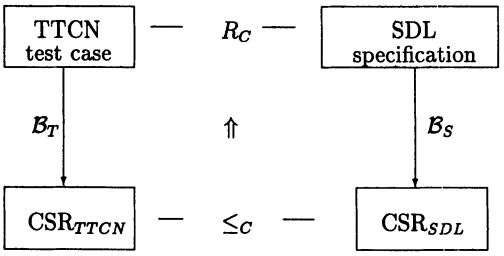

Figure 2: A model of the validation process for dynamic behaviour

This general definition of a correctness criterion is elaborated to provide an instance of the validation framework. Fig. 2 illustrates the validation process for dynamic behaviour. R_C defines a correctness criterion between TTCN test cases and SDL specifications. The correctness of a test case is not determined directly in terms of the TTCN test case and the SDL specification. Instead, the test case and specification are transformed to entities in the CSR. If the corresponding entities are in relation $\leq_C$ we can conclude that the test case complies with the defined correctness criterion R_C.

Although we focus on the validation of test cases we would like to emphasize that our framework can easily be extended to cover a complete test suite:

Definition 5 Let TS denote a TTCN test suite, where $TS = \{t_1, \ldots, t_n\}$ is the set of test cases of the test suite except of those test cases testing for syntactically and semantically invalid behaviour. Let S denote an SDL specification.

A test suite TS is valid with respect to SDL specification S, denoted $Valid(TS, S)$, if correctness criterion R_C is satisfied by every test case of the test suite.

$$Valid(TS, S) \text{ iff } \forall t \in TS : (t, S) \in R_C$$

As $\forall t \in TS : (B_t, B_S) \in \leq_C \Rightarrow (t, S) \in R_C$ then $\forall t \in TS : (B_t, B_S) \in \leq_C \Rightarrow Valid(TS, S)$. □

Validation of the dynamic behaviour of a test case can be mapped to the problem of validating that the assignment of test verdicts, i.e. **Pass**, **Fail**, and **Inconclusive**, is consistent with the behaviour expressed in the specification. To check if the assignment of verdicts is consistent with a specification it suffices to analyse the sequences of test events assigned a final verdict. Let $Tr_{\mathsf{FV}}(t)$ be the set of test event sequences with a final verdict assignment. $Tr_{\mathsf{FV}}(t)$ can be partitioned into disjoint sets dependent on the verdict assigned: $Tr_{\mathsf{Pass}}(t)$, $Tr_{\mathsf{Inconc}}(t)$, and $Tr_{\mathsf{Fail}}(t)$. Intuitively, validation of a test event sequence $\sigma \in Tr_{\mathsf{Pass}}(t)$ has to confirm that the sequence of events is possible according to the specification.

The validation framework also has to define a mapping of test case events to events of the SDL specification and vice versa. To define the mapping, it has to be decided for which events the mapping should be defined.

Example 3 A possible choice of CSR events to include in the validation process is the events which are exchanged at the system level. In terms of the CSR that is: $\epsilon_S = \{\mathsf{input}(sig), \mathsf{output}(sig), \tau\}$.

One reason to select this set of events is that they constitute the events of interest when conformance testing is performed. However, an SDL system may always receive a signal if there is a channel that can convey the signal. Then to perform a more detailed analysis of the consistency of a test case, the validation process must cover also internal behaviour of the SDL system. So, the validation should not be restricted to system level events. □

For the definition of a relation between TTCN test case events and SDL specification events it should be noted that the behaviour of an SDL specification and a test case reflect two different views on the system requirements. An SDL specification expresses the dynamic requirements that an implementation must satisfy, while a test case specifies sequences of test events to be applied to determine the conformance of an implementation.

We shall use the notation $\overline{X}$ to denote corresponding behaviour of X, where X denotes behaviour of either a test case or an SDL specification: If x denotes valid behaviour of a test case t, then the corresponding behaviour of the SDL specification is denoted $\overline{x}$. One way of defining the static relation between specification and test case events is by mapping tables as described in [17]. Note that we shall overload the overline operator $\overline{x}$ to indicate corresponding descriptions for behaviours, traces, events, and data parameters.

3.3 An instance of the validation framework

The previous discussion suggests to define correctness criterion $\leq_C$ in terms of the dynamic behaviour of test cases and SDL systems. The CSR provides an abstract model that allows to reason about the dynamic behaviour of test cases and SDL systems.

3.3.1 Trace validation

As our validation framework aims at checking the correctness of verdict assignments in a test case, only behaviour is considered for which a verdict is assigned. Thus, if the validation process is based on traces of system level events only a subset of the traces is used during the validation process. We denote the set of traces of a test case t by $Tr(B_t)$ and the trace set of the specification S by $Tr(B_S)$. A trace for which a verdict is assigned is termed a *complete trace* and the set of complete traces is $Tr_{cp}(B_t)$, where $Tr_{cp}(B_t) \subseteq Tr(B_t)$. Similar to the partitioning of the behaviour sets described previously, we introduce three sets of traces, one for each type of verdict assigned:

$$Tr_{cp}(B_t) = Tr_{\mathsf{Pass}}(B_t) \cup Tr_{\mathsf{Inconc}}(B_t) \cup Tr_{\mathsf{Fail}}(B_t) \tag{10}$$

A minimal requirement that a test case must satisfy in order to be considered valid with respect to an SDL specification is that all behaviours indicating non-conformance, i.e. result in a Fail verdict assignment, should have no corresponding behaviour in the SDL specification.

Definition 6 The behaviour B_t of a TTCN test case t is valid with respect to the behaviour of an SDL system B_S if

$$B_t \leq_{c1} B_S \text{ iff } \forall \sigma \in Tr_{\mathsf{Fail}}(B_t) : \overline{\sigma} \notin Tr(B_S)$$

where $\leq_{c1}$ denotes the relation expressing the correctness criterion and $\overline{\sigma}$ the trace that corresponds to σ. □

Additionally, for complete traces assigned a Pass or Inconclusive verdict, it may be required to have a corresponding behaviour of the SDL system.

Definition 7 The behaviour of a TTCN test case B_t is valid with respect to the behaviour of an SDL system B_S if

$$B_t \leq_{c2} B_S \quad \text{iff} \quad B_t \leq_{c1} B_S \wedge \forall \sigma \in Tr_{\mathsf{Pass}}(B_t) \cup Tr_{\mathsf{Inconc}}(B_t) : \overline{\sigma} \in Tr(B_S)$$

where $\leq_{c2}$ denotes the relation that expresses the correctness criterion. □

Definitions 6 and 7 are based on comparison of corresponding traces of input and output events of a TTCN test case and an SDL specification. A prerequisite in order to perform a comparison of traces is that a correspondence between inputs and outputs has been defined. Since such a correspondence cannot usually be defined on a pure syntactical basis we assume that a static mapping between TTCN and SDL events is defined, e.g.

CSR event of TTCN test case	*Corresponding event of SDL system*
input(x)	output($\overline{x}$)
output(x)	input($\overline{x}$)

where the overline operator used for parameter x is to be resolved using a mapping between TTCN and SDL parameter names.

Validation of an invalid trace is not as straightforward as indicated by Definition 6. In TTCN, the Otherwise statement defines a controlled way to handle unforeseen events. So traces which include an Otherwise statement lead to a Fail verdict assignment according to [3]. An Otherwise construct will not only match unforeseen events but any event that is not mentioned in the set of alternatives. This means that also input actions that are correct with respect to the specification may be handled by an Otherwise construct. We therefore propose another approach for the validation of such invalid traces.

Intuitively, an Otherwise statement is valid if the traces specified in the SDL specification are all covered by corresponding traces in the TTCN test case such that these traces are all members of the Pass or Inconclusive trace sets.

In order to formalise this definition we introduce the following notation. The set of *otherwise traces*, denoted $Tr_o(B_t)$, is defined as

$$Tr_o(B_t) = \{\sigma \mid \sigma \cdot x \in Tr(B_t) \text{ and input } x \text{ is consumed by an Otherwise}\}$$

This set consists of traces that may be extended by an event such that this event is handled by an Otherwise statement. Similarly, we define the set of traces of a specification S such that these traces are possible extensions of corresponding traces $\rho \in Tr_o(B_t)$:

$$Tr(\rho, B_S) = \{\lambda | \lambda \in Tr(B_S) \wedge x \in \epsilon_S - \{\tau\} \wedge \overline{\rho} \cdot x = \lambda\}$$

Next we define predicate *ValidOTr* that is satisfied if for all valid extensions σ of trace $\rho \in Tr_o(B_t)$ with respect to specification S, there is a $\sigma' \in Tr_{\mathsf{Pass}}(B_t) \cup Tr_{\mathsf{Inconc}}(B_t)$ such that $\overline{\sigma}$ is a prefix of σ', denoted $\overline{\sigma} \preceq \sigma'$. The intended use of this predicate is to check that for all possible traces with respect to the specification these traces are mapped to traces in the test case that result in a Pass or Inconclusive verdict assignment.

Definition 8 Let S be an SDL specification. Let ρ be a trace in the set of otherwise traces $Tr_o(B_t)$. Predicate *ValidOTr* is defined as follows:

$$\mathit{ValidOTr}(\rho) \quad \text{iff} \quad \forall \sigma \in Tr(\rho, B_S) : \exists \sigma' \in Tr_{\mathsf{Pass}}(B_t) \cup Tr_{\mathsf{Inconc}}(B_t) : \overline{\sigma} \preceq \sigma'$$

□

Then we define a correctness criterion that integrates the validation of the Otherwise statement. To support the validation process we define a reduced set of traces associated with a Fail verdict. The trace set $Tr_{\mathsf{eFail}} \subseteq Tr_{\mathsf{Fail}}$ is the set of Fail traces such that no event is handled by an Otherwise statement.

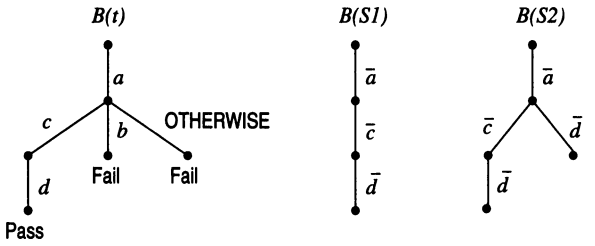

Figure 3: Behaviour models of a test case B_t and two SDL systems $B(S1)$ and $B(S2)$

Definition 9 The behaviour of a TTCN test case B_t is valid with respect to the behaviour of an SDL system B_S if

$$B_t \leq_{c3} B_S \quad \text{iff} \quad \begin{array}{l} \forall \sigma \in Tr_{\mathsf{eFail}}(B_t) : \overline{\sigma} \notin Tr(B_S) \ \wedge \\ \forall \sigma \in Tr_{\mathsf{Pass}}(B_t) \cup Tr_{\mathsf{Inconc}}(B_t) : \overline{\sigma} \in Tr(B_S) \ \wedge \\ \forall \sigma \in Tr_o(B_t) : ValidOTr(\sigma) \end{array}$$

□

Summarizing, $\leq_{c3}$ is a refined definition of $\leq_C$ that serves as our correctness criterion.

Example 4 In Fig. 3 models of behaviour of a TTCN test case t and two SDL systems $S1$ and $S2$ are shown. For the validation process based on Definition 9 we derive from the test case behaviour:

$$Tr_{\mathsf{Pass}}(B_t) = \{\langle acd \rangle\} \quad Tr_{\mathsf{Inconc}}(B_t) = \{\} \quad Tr_o(B_t) = \{\langle a \rangle\} \quad Tr_{\mathsf{eFail}}(B_t) = \{\langle ab \rangle\}$$

To validate the TTCN behaviour against the SDL behaviour B_{S1} we must check that the SDL specification cannot perform the Fail trace $\langle ab \rangle$. This is seen to be the case. Furthermore, we observe that the SDL system can perform a trace $\langle \overline{a}\overline{c}\overline{d} \rangle$ that corresponds to the Pass trace. Finally, as B_{S1} after sequence $\langle \overline{a} \rangle$ cannot perform any other action than $\overline{c}$ that is a prefix of a Pass trace, also the requirement on Otherwise statements is satisfied. Hence test case t is valid with respect to SDL specification $S1$.

For the SDL specification $S2$ it can be shown that the requirements on Fail and Pass traces are satisfied. However, for the Otherwise trace $\langle a \rangle$ it is seen that the SDL specification may perform $\langle \overline{a}\overline{d} \rangle$ and $\langle \overline{a}\overline{c} \rangle$. The former is not a prefix of any Pass trace and so it is handled by the Otherwise statement. From this analysis we conclude that test case t is not valid with respect to SDL specification $S2$. □

3.3.2 Event validation

To perform the trace validation defined we must also define how events of these traces are validated in terms of the CSR model. This is necessary as the validation of an input event of a TTCN test case is done by checking that the SDL system is able to perform the corresponding output event. If a test case sends a message it is checked that the SDL system is able to receive a corresponding signal. Since an SDL system may always receive

a signal if there is a channel that can convey the signal, a stronger requirement should be used for validation of TTCN test case output events. An output event is valid, if it can be proven that the corresponding signal is received and consumed by a process instance within the system.

To perform event validation the set of complete traces needs to be derived first. The *final verdict* predicate FV identifies traces of system level events which are assigned a final verdict.

Definition 10 Let t be a test case and B_t be the corresponding CSR entity. Let the behaviour of test case t be given by $S_{B_t} = (S_t, \varepsilon_t, \rightarrow_t, s_0)$. Let $s', s'' \in S_t$ where $s' = (\sigma_1, \cdots, \sigma_k, (\sigma_{k+1}, \cdots, \sigma_l, p_1, \cdots, p, \cdots, p_n))$, and $s'' = s'[p'/p]$[6]. In the state description σ_i, $1 \leq i \leq k$, denote PCO queues, σ_i, $k+1 \leq i \leq l$, denote CP queues, and p_i denotes process instances of test components. For $\sigma \in \varepsilon_t^*$ the final verdict predicate $FV(\sigma)$ is defined as

$$\begin{aligned} FV(\sigma) \quad \text{iff} \quad & s_0 \stackrel{\sigma}{\Longrightarrow} s' \wedge s' \stackrel{\tau}{\longrightarrow}_t s'' \wedge \\ & p \stackrel{\tau}{\longrightarrow}_{\mathsf{PI}} p' \wedge p = \langle P_{\varepsilon,\rho}, I, T\rangle \wedge p' = \langle P'_{\varepsilon,\rho'}, I, T\rangle \wedge \\ & P ::= \mathsf{R} := \mathsf{v}; P' \ \wedge \ \rho' = \rho[R \mapsto v] \end{aligned}$$

where $\mathsf{v} \in \{\mathsf{Pass}, \mathsf{Fail}, \mathsf{Inconc}\}$, I denotes the state of the input process, and T denotes the state of the timer process. □

Since execution of the test case stops in an assignment of a final verdict (FV), a complete trace is a trace that cannot be extended. $Tr_{cp}(B_t) = \{\sigma \in Tr(B_t) \mid FV(\sigma)\}$ is the set of complete traces of test case t.

For the validation of output events of a test case it must be validated that the SDL system is able to receive and to consume the corresponding input event. We define predicates R and C related to the reception and consumption of a message.

Definition 11 Let S denote an SDL specification and B_S the corresponding CSR entity. The CSR system level description of the behaviour is denoted $S_{B_S} = (S_S, \varepsilon_S, \rightarrow_S, s_0)$. Let $s, s's'', s''' \in S_S$ where s denotes the current state

$$s = (\sigma_1, \cdots, \sigma_k, m_1, \cdots, (\cdots, p_{pid_i}, \cdots)_j, \cdots, m_m),$$

$s' = s[\sigma_i'/\sigma_i]$ and $s'' = s'[p'_{pid_i}/p_{pid_i}, \sigma_i''/\sigma_i']$. In the state description σ_i, $1 \leq i \leq k$, denote queues of links (channels) and m_i, $1 \leq i \leq m$, denote modules (blocks). Then the *receive* predicate $R(e, s)$ where $e = \mathsf{input}(msg) \in \varepsilon_S$ and $msg \in Sig$ is satisfied when

$$\begin{aligned} R(e,s) \quad \text{iff} \quad & s \stackrel{e}{\longrightarrow}_S s' \wedge \sigma_i' = msg \cdot \sigma_i \wedge \\ & s' \stackrel{\tau}{\longrightarrow}_S s'' \wedge p_{pid_i} \stackrel{\mathsf{receive}(msg)}{\longrightarrow}_{\mathsf{PI}} p'_{pid_i} \wedge \\ & p_{pid_i} = \langle P_{\varepsilon,\rho}, (\sigma, \sigma'), T\rangle \wedge p'_{pid_i} = \langle P_{\varepsilon,\rho}, (msg \cdot \sigma, \sigma'), T\rangle \wedge \\ & \sigma_i'' = \sigma_i \end{aligned}$$

where (σ, σ') denotes the state of the input queue and T the state of the timer process. Note that σ_i must be empty to have $R(e, s)$ be fulfilled.

[6]The notation $s' = s[t'/t]$ denotes that s' is equal to s except for that t' has been substituted for t.

The *consume* predicate $C(e, s)$ may now be defined based on the receive predicate. Let state $s''' = s''[p''_{pid_i}/p'_{pid_i}]$. Then the consume predicate $C(e, s)$ where $e = \mathsf{input}(msg) \in \varepsilon_S$ and $msg \in Sig$ is satisfied when

$$\begin{aligned} C(e,s) \quad \text{iff} \quad & R(e,s) \wedge \\ & s'' \xrightarrow{\tau}_S s''' \wedge p'_{pid_i} \xrightarrow{\tau}_{PI} p''_{pid_i} \wedge \\ & p'_{pid_i} = \langle P_{\varepsilon,\rho}, (\sigma \cdot msg, \sigma'), T \rangle \wedge p''_{pid_i} = \langle P'_{\varepsilon,\rho'}, (\sigma, \sigma'), T \rangle \wedge \\ & P ::= \mathsf{input}(x); P' \wedge \rho' = \rho[x \mapsto msg] \end{aligned}$$ □

The validation of Pass, Inconclusive, and Otherwise traces can be defined using the requirements on events. The following predicate sketch how these can be validated.

Let B_S denote the CSR entity of an SDL system against which a trace σ of system level events is to be validated. Let the system level of B_S be $S_{B_S} = (S_S, \varepsilon_S, \rightarrow_S, s_0)$ and $s, s' \in S_S$ where s denotes the current state. Finally the CSR representation in state s' is denoted B'_S. The predicate $CheckTrace(\sigma, B_S)$ then defines the requirement.

$$\begin{aligned} CheckTrace(\sigma, B_S) \quad \text{iff} \quad & (\sigma = \langle\rangle) \\ \vee \ & ((\sigma = e \cdot \sigma') \wedge (e = \mathsf{input}(x)) \wedge \\ & (s \xrightarrow{\mathsf{output}(x)}_S s') \wedge (CheckTrace(\sigma', B'_S))) \\ \vee \ & ((\sigma = e \cdot \sigma') \wedge (e = \mathsf{output}(x)) \wedge \\ & (s \xrightarrow{\mathsf{input}(x)}_S s') \wedge C(\bar{e}, s) \wedge (CheckTrace(\sigma', B'_S))) \end{aligned}$$

We can use this predicate also as a basis for the validation of Fail traces, i.e. we must show that the predicate *CheckTrace* is not satisfied.

Example 5 A test case t and a process of an SDL specification S are shown in Fig. 4. The following process may be performed to validate the correctness of test case t according to the proposed instantiation of the validation framework.

Identifier: t	
Behaviour Description	V
PCO!a	
PCO?x	PASS
PCO?OTHERWISE	FAIL

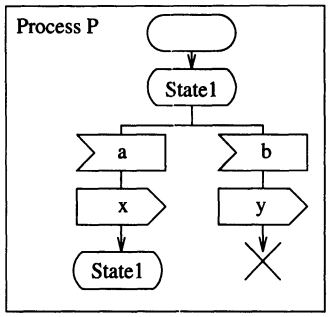

Figure 4: A test case t and process part of a corresponding SDL specification.

First, from the CSR representation of test case t we derive the set of complete traces to be validated. The basic process expression of the test case behaviour in the CSR is

$$\begin{aligned} P_t \ ::= \ & \mathsf{output}(a) \text{ to } PCO\,; \\ & (\mathsf{input}(x) \text{ from } PCO\,;\ R := \mathsf{Pass}\,;\ \mathsf{stop}\,;\ nil) \\ & \oplus\ (\mathsf{otherwise}\ PCO\ [true]\,;\ R := \mathsf{Fail}\,;\ \mathsf{stop}\,;\ nil) \end{aligned}$$

The final verdict predicate FV is used to derive all complete traces. From the CSR representation we get the non-empty sets of system level traces: $Tr_{\mathsf{Pass}}(B_t) = \{\langle \mathsf{output}(a) \cdot \mathsf{input}(x)\rangle\}$ and $Tr_o(B_t) = \{\langle \mathsf{output}(a)\rangle\}$.

The internal behaviour of the SDL specification is also needed to validate the test case traces. The CSR representation of process P is

$$P ::= (\mathsf{input}(a)\,;\, \mathsf{output}(x)\,;\, P) + (\mathsf{input}(b)\,;\, \mathsf{output}(y)\,;\, \mathsf{stop}\,;\, nil)$$

where the occurrence of P in the expression represents the infinite behaviour of process P. Validation of the Pass trace is performed according to the *CheckTrace* predicate. The initial event of the trace is $\mathsf{output}(a)$, then it must be derived from the CSR representation of the SDL specification that a corresponding input action is possible and that this signal is received and consumed. If s_i denote system states of the CSR representation B_S with s_0 the initial state, then to validate the output event we must derive:

$$s_0 \xrightarrow{\mathsf{input(a)}}_{\mathsf{S}} s_1 \;\wedge\; C(\mathsf{input}(a), s_0)$$

i.e. the SDL system can perform an input event and this input is received and consumed by a process instance (predicate C). As the links may always receive a signal that can be conveyed on the channel, the transition from s_0 to s_1 is always possible. From state s_1, a process instance in the system must be able to receive signal a, that is

$$p_{pid} \xrightarrow{\mathsf{receive(a)}}_{\mathsf{PI}} p'_{pid} \;\wedge\; s_1 \xrightarrow{\tau}_{\mathsf{S}} s_2$$

where p_{pid} and p'_{pid} denote states of a process instance. As signal a is in the input set of SDL process P, the process instance that represents the behaviour of the process always can receive the signal in its input process. Then from the derived system state s_2 it must be shown that signal a can be consumed. In the initial system state s_0 the state of the basic process is equal to the basic process expression P. As the previously described transitions has not changed the state of the basic process we can derive that from system state s_2 the process instance can perform an internal event, consuming signal a from the input port.

$$p'_{pid} \xrightarrow{\tau}_{\mathsf{PI}} p''_{pid} \;\wedge\; s_2 \xrightarrow{\tau}_{\mathsf{S}} s_3 \;\wedge$$
$$p'_{pid} = \langle P, (\langle a\rangle, \langle\rangle), T\rangle \;\wedge\; p''_{pid} = \langle P', (\langle\rangle, \langle\rangle), T\rangle \;\wedge P' ::= \mathsf{output}(x)\,;\, P$$

Hence we can conclude that the test case output event is valid. Then from system state s_3 we must show that the system can perform output event x as the next external system level event. This corresponds to the next action of the Pass trace, i.e. input(x). When this event is derived, the trace is empty and predicate *CheckTrace* satisfied. As there is no other Pass trace to be checked, what needs to be done is to show that the requirement on the Otherwise statement is satisfied.

The only system level event that the SDL system can perform after input event a is output event x, so the requirement is satisfied. Finally, we can conclude that test case t is valid with respect to SDL specification S for correctness criterion $\leq_{c3}$. □

From the example it is seen how correctness of a test case can be determined. But from the example it should be clear also that for real test cases the validation process must be supported by tools.

4 Conclusions

This paper has described an approach to validate concurrent TTCN test cases against SDL specifications. The main ideas of the approach have been discussed: a semantical model for TTCN and SDL'92, a notion of test case correctness, and a general validation procedure. All these ideas have been put into a validation framework.

In an introduction to the semantical model CSR, we have outlined its main properties. Properties like compositionality and abstraction of the CSR have been identified as important for the definition of various correctness criteria. As a solution to handle the inherent complexity of test case validation, decomposition of correctness criteria has been suggested. We have demonstrated how the validation framework is applied.

The validation method presented needs tool support to be of practical value. Even if this requirement is fulfilled there may be a need for additional metrics to handle real size specifications.

Acknowledgement The authors would like to gratefully acknowledge the anonymous referees for fruitful comments to the previous version of this paper.

References

[1] ISO/IEC. *OSI – Conformance testing methodology and framework - Part 1: General concepts.* IS 9646, ISO/IEC, 1991.

[2] H. Rudin. *Protocol development success stories: Part I.* In R. J. Linn and M. Uyar, editors, *PSTV XI,* pages 149–160. IFIP, North-Holland, 1992.

[3] ISO/IEC. *OSI – Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation.* IS 9646, ISO/IEC, 1992.

[4] CCITT. *Specification and Description Language (SDL).* CCITT Recommendation Z.100, CCITT/ITU, 1992.

[5] ISO/IEC. *OSI – Conformance testing methodology and framework - Proposed Draft Amendment 1 to ISO/IEC 9646 Part 3: TTCN Extensions.* Draft Amendment 9646, ISO/IEC, 1991.

[6] ETSI/MTS. *Semantical relationship between SDL and TTCN – A common semantics representation.* ETR 071, ETSI, 1993.

[7] T. Walter and B. Plattner. *An operational semantics for concurrent TTCN.* In G. v. Bochmann, R. Dssouli, and A. Das, editors, *IWPTS V,* pages 131–143. IFIP, North-Holland, 1992.

[8] F. Kristoffersen and T. Walter. *Test Case Validation – TTCN test case validation against SDL specifications.* TIK-Report 8 (May 1994) ETH Zürich or TDR RR 1994-1 Tele Danmark Research, 1994.

[9] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification 1. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[10] ISO/IEC. *OSI - Specification of Abstract Syntax Notation 1 (ASN.1).* IS 8824, ISO/IEC, 1987.

[11] G. Plotkin. *A structural approach to operational semantics.* DAIMI FN-19, Aarhus University, September 1981.

[12] J. F. Groote. Process algebra and structured operational semantics. PhD thesis, University of Amsterdam, 1991.

[13] U. Bär and J. M. Schneider. *A common semantics representation for SDL and TTCN.* In R. J. Linn and M. Uyar, editors, *PSTV XI*, pages 279–295. IFIP, North-Holland, 1992.

[14] S. Vuong and J. Alilovic-Curgus. *On Test Coverage Metrics for Communication Protocols.* In J. Kroon, R. Heijink, and E. Brinksma, editors, *IWPTS IV*, pages 31–45. IFIP, North-Holland, 1991.

[15] M. McAllister, S. T. Vuong, and J. Alilovic-Curgus. *Automated Test Case Selection Based on Test Coverage Metrics.* In G. v. Bochmann, R. Dssouli, and A. Das, editors, *IWPTS V*, pages 93–104. IFIP, North-Holland, 1992.

[16] ISO/ITU. *Formal Methods in Conformance Testing.* Wd, ITU TS SG10 Q8 and ISO SC21 WG1 P54, June 1993.

[17] R. J. Velthuys. Conformance testing based on formal system models. PhD thesis, University of Berne, 1992.