6

Timed systems behaviour and conformance testing – a mathematical framework

B. Baumgarten
GMD
Rheinstr. 75, D-64295 Darmstadt, Germany
Tel +49 6151 869 263, Fax +49 6151 869 224
baumgart@darmstadt.gmd.de

Abstract

A formal framework for conformance testing is a prerequisite for the verification as well as for the correct generation of test cases. In this paper, we develop a formal view of systems, behaviour, and testing, that includes aspects of time. It deals on a semantic level with systems cooperating via timed input/output rendezvous and was developed with a view to the OSI architecture, service and conformance testing concepts. We outline a theoretical framework for testing, in which many important informal notions of conformance testing are reconstructed as formal notions with clear relationships among them. In the process, some of these notions are refined and new ones are added. The verdict concept is clarified by the introduction of the notions of evidence function, verdict strategies, and additional parameters.

Keywords

Conformance testing, system behaviour, time, observation, test verdicts, system parameters

1 INTRODUCTION

While OSI protocols reportedly are falling back behind other protocol families in the number of installations, the architectural and conformance testing (CT) concepts of OSI seem to have a lasting impact on theory and practice of protocols, even outside of OSI. The OSI Conformance Testing Methodology and Framework (CTMF) standard [ISO91, ISO94a] provides one of the most comprehensive sources for practical concepts of CT, of which it gives informal definitions. Terms and notions of CT can be roughly divided into

- administrative (document formats, mandatory texts and the like),
- procedural (rules for human beings and institutions), and
- behavioural, dealing with the validity of implementation and tester behaviour, their specification and assessment.

In this paper, we outline a theory of systems and testing, clarifying many behavioural CT notions, in particular with respect to the operational semantics of TTCN [BG94, ISO91, ISO94a]. Its starting point was the time-free framework described in [BW95].

A. Cavalli et al. (eds.), *Protocol Test Systems VIII*

1.1 Overview

In Section 2, we develop a formal view of cooperating systems taylored to architectural conventions in the protocol world. This is a semantic view, not a formal specification language. Appropriate formal specifications in many languages can be interpreted in it. We discuss why this view is fairly general. In Section 3, we give a precise and practically useful meaning to behavioural terms related to protocol testing, even to some that were originally not very clearly defined in standards. Moreover, we develop some new formal concepts, such as the evidence of a test outcome and the correctness of a test case. Our approach solves some of the well-known problems in CT specification [Bau94].

Due to space limitations, some of our definitions are given merely in the guise of an informal summary and no specification examples in any of the current specification languages are given. Natural language terms being defined explicitly or implicitly are italicized.

1.2 Comparison with other approaches

There exist a number of formal and informal frameworks for black box testing of protocols, such as [Bri89, ISO95, ISO91, Pha94, Pha94a, Tre92, Tre94]. Available space does not suffice to compare extensively those texts with the present one. We confine ourselves to the observation that in each of the following points our framework differs from one or several of the cited approaches:

- It is not assumed that the IUT (implementation under test) behaviour can theoretically be specified in the same formal language as the specification ('test assumption'). Instead we assume that the specification in the chosen formal language can be, and is, interpreted in the semantic framework of Section 2, and that the testing related real systems' behaviour can be modelled in this framework.
- The implementation relation is not considered to be arbitrary or depending on circumstances. Rather, we attempt to formalize a single implementation relation which appears to prevail implicitly in TTCN.
- We explicitly do without the 'PCO queues' of CTMF. In [Bau94] it was shown that they are ambiguously described, and that some obvious ways to define them more clearly result in their being either superfluous, non-implementable, or contradicting other standards. Therefore, we also permit Send events to be unsuccessful.
- We formalize the intuitive notion of the test verdict INCONCLUSIVE to the effect that it applies to observations which, even when fully exploiting all the knowledge the tester has, leave it open whether the IUT behaved externally as specified or not.
- Tests presupposing restrictions of the non-determinism originally granted by the specification are interpreted as presuming an agreement on different, more restrictive, specifications – a procedural question. The applicability of INCONCLUSIVE is considered to be independent from non-determinism.
- Our approach involves a notion of timed rendezvous with explicit enabling and disabling events. It gives a precise meaning to the observation of refusals of actions: refusals are observed by a disabling action after a specified waiting period, corresponding to the use of the TTCN timeout mechanism.

1.3 Mathematical preliminaries

For any set A, the set of all *finite words* over A is $A^* := \{a_1 \ldots a_n \mid n \geq 0 \wedge \forall 1 \leq i \leq n\colon a_i \in A\}$, while the sets of all (countably) *infinite* and of all *countable words* over A are $A^\omega := \{a_1 a_2 \ldots \mid \forall 1 \leq i\colon a_i \in A\}$ and $A^\infty := A^* \cup A^\omega$, respectively. We use 'countable' in the sense of 'finite or countably infinite.'

For every $w \in A^\infty$, *length*(w) is the length of w, a natural number or ω. ε denotes the empty word, i.e. *length*(ε) = 0.

Any infinite sequence $w_1 w_2 \ldots$ of finite words such that each w_i is a prefix of w_{i+1} and $\lim_{i\to\infty} length(w_i) = \infty$ defines a unique infinite limit word $\lim_{i\to\infty} w_i$. To each (finite-word) language over A, $L \subseteq A^*$, we can associate the limit language L_ω and the countable-word closure $L_\infty = L \cup L_\omega$, obtained by constructing, or adding to L, respectively, the limit words of all suitable sequences of words in L. In the same vein, any prefix-order-preserving mapping from $L \subseteq A^*$ to $K \subseteq B^*$ can be canonically extended to a mapping from L_∞ to K_∞.

2. SYSTEMS AND SYSTEM COOPERATION

We develop a behaviour-oriented view of discrete systems performing in Newtonian physical time. We also show how our view can be used to model other views.

2.1 Actions and system signatures

Systems can perform actions, which are either internal, intermediate, or external. An external action is either an input or an output action. Intermediate actions enable or disable inputs and output actions. Each action is associated with a data type. Each performance of an action is associated with a data object of that type and occurs at some moment in time. A system may perform behaviour sequences, i.e. time-ordered countable sequences of action occurrences.

Throughout this paper, the term 'data type' may be interpreted in an intuitive sense. If more formality is desired, all definitions should be considered to be given relatively to a fixed model M of some many-sorted abstract data type [EM85] that is rich enough to encompass all of the finitely many sorts of interest. A *data type* is then a sort domain of M.

A *system signature*, describing types of actions and data objects, is an octuple

$$\Sigma = (\mathit{IntActs}(\Sigma), \mathit{IEnActs}(\Sigma), \mathit{InpActs}(\Sigma), \mathit{IDisActs}(\Sigma), \mathit{OEnActs}(\Sigma), \mathit{OutActs}(\Sigma), \mathit{ODisActs}(\Sigma), \mathit{Type}_\Sigma)$$

such that (cf. Figure 1)

- $\mathit{IntActs}(\Sigma)$, $\mathit{IEnActs}(\Sigma)$, $\mathit{InpActs}(\Sigma)$, $\mathit{IDisActs}(\Sigma)$, $\mathit{OEnActs}(\Sigma)$, $\mathit{OutActs}(\Sigma)$, and $\mathit{ODisActs}(\Sigma)$, are mutually disjoint sets with fixed bijections

  En_Σ: $\mathit{InpActs}(\Sigma) \cup \mathit{OutActs}(\Sigma) \to \mathit{IEnActs}(\Sigma) \cup \mathit{OEnActs}(\Sigma)$

  such that $\mathit{En}_\Sigma[\mathit{InpActs}(\Sigma)] = \mathit{IEnActs}(\Sigma)$ and $\mathit{En}_\Sigma[\mathit{OutActs}(\Sigma)] = \mathit{OEnActs}(\Sigma)$, and

  Dis_Σ: $\mathit{InpActs}(\Sigma) \cup \mathit{OutActs}(\Sigma) \to \mathit{IDisActs}(\Sigma) \cup \mathit{ODisActs}(\Sigma)$

  such that $\mathit{Dis}_\Sigma[\mathit{InpActs}(\Sigma)] = \mathit{IDisActs}(\Sigma)$ and $\mathit{Dis}_\Sigma[\mathit{OutActs}(\Sigma)] = \mathit{ODisActs}(\Sigma)$.
- Type_Σ assigns to each action a data type.

We call the elements of $\mathit{IntActs}(\Sigma)$, $\mathit{IEnActs}(\Sigma)$, $\mathit{InpActs}(\Sigma)$, $\mathit{IDisActs}(\Sigma)$, $\mathit{OEnActs}(\Sigma)$, $\mathit{OutActs}(\Sigma)$, $\mathit{ODisActs}(\Sigma)$, *internal*, *input enable*, *input*, *input disable*, *output enable*, *output*, and *output disable actions*, respectively. The set

$$\mathit{ExtActs}(\Sigma) := \mathit{InpActs}(\Sigma) \cup \mathit{OutActs}(\Sigma)$$

comprises all *external actions*. Other systems may participate in them, as described in 2.4. The set

$$\mathit{ItrmActs}(\Sigma) := \mathit{IEnActs}(\Sigma) \cup \mathit{IDisActs}(\Sigma) \cup \mathit{OEnActs}(\Sigma) \cup \mathit{ODisActs}(\Sigma)$$

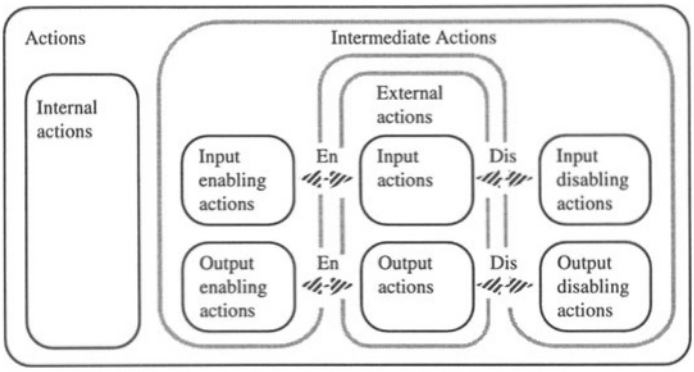

Figure 1 Action sets.

comprises all *intermediate actions*. $VisActs(\Sigma) := ExtActs(\Sigma) \cup ItrmActs(\Sigma)$ is the set of all *visible actions*, and $Actions(\Sigma) := IntActs(\Sigma) \cup VisActs(\Sigma)$ is the set of all *actions*. In natural language terms, we will often omit the attribute 'over Σ', as long as we are not dealing with more than one system signature.

In 2.3, systems will be defined by pairing system signatures with behaviours.

Application to CTMF

In TTCN test cases, after the expansion of constructs, Receive and Send at a PCO are input and output actions, respectively. More precisely, the entering of a list of Receive alternatives is an input enable action for each pair (PCO,service primitive) appearing in this list, while the success of one alternative on this list is the corresponding input action. The immediate success of Send lines claimed by TTCN would mean that the matching input action in the accepting entity must always be enabled before the Send line is enabled. Timeout is internal or disabling, depending on the context, and pseudo-events are internal actions. Send constraints are the data objects of send occurrences, while Receive constraints are subtypes of the input action type. They are used to determine subsequent behaviour, represented by the subtree below the successful alternative, cf. 2.2.2.

2.2 Timed behaviour

In this subsection, we consider a fixed system signature Σ.

Time is counted in seconds having passed after 1.1.1900, 0h00, GMT, for example. The actual choices of the zero point and the time unit do not really matter, of course, but it is necessary to choose in order to be unambiguous. Purely relative times can be expressed easily by using sets of absolutely timed traces, as we will see in 2.3.1.

Occurrences and occurrence sequences

An action may be performed repeatedly and with various data objects; we speak of various possible occurrences. For example, an action *message_reception* may occur at several points in time and with various, sometimes even identical, messages. In each occurrence *occ* of an action, the unique action $Act(occ)$ is associated with a data object $Obj(occ)$ of type $Type_\Sigma(Act(occ))$ and with a real number $Time(occ)$. The data object may be trivial, as in the case of a synchronization event, or it may encompass several parameters, such as message type identifier, sender and receiver addresses, and user data in a message. We assume that

actions are instantaneous and that *Time*(*occ*) denotes the point in time at which *occ* 'happens.' 'Time-consuming activities' can be represented by two instantaneous actions each, representing start and end.

Mathematically, we define the set of all *occurrences* over Σ as

$$Occs(\Sigma) := \{(act,obj,t) \mid act \in Actions(\Sigma), obj \in Type_\Sigma(act), t \in \mathbb{R}\},$$

entailing $\forall\ occ \in Occs(\Sigma)$: $occ = (Act(occ), Obj(occ), Time(occ))$.
For a finite word over the occurrences, $occseq \in Occs(\Sigma)^*$, and an occurrence *occ*, we define the logical expression

$$After(occseq,occ) :\Leftrightarrow occseq = \varepsilon \vee (occseq \neq \varepsilon \wedge Time(last(occseq)) \leq Time(occ)).$$

We partition $Occs(\Sigma)$ according to their *Act*-values, thus defining $IntOccs(\Sigma)$, $IEnOccs(\Sigma)$, $InpOccs(\Sigma)$, $IDisOccs(\Sigma)$, $OEnOccs(\Sigma)$, $OutOccs(\Sigma)$, $ODisOccs(\Sigma)$, and $ExtOccs(\Sigma)$ in the obvious way.
The set of all *finite occurrence sequences over* Σ, $FOccSeqs(\Sigma)$, and the *sets of enabled actions* after these sequences are simultaneously inductively defined by the following rules:

$$\varepsilon \in FOccSeqs(\Sigma) \wedge Enabled(\varepsilon)=\emptyset$$

$$\begin{aligned} &occseq \in FOccSeqs(\Sigma) \wedge occ \in IntOccs(\Sigma) \wedge After(occseq,occ) \\ \Rightarrow\ &occseq \circ occ \in FOccSeqs(\Sigma) \\ &\quad \wedge\ Enabled(occseq \circ occ) = Enabled(occseq) \end{aligned}$$

$$\begin{aligned} &occseq \in FOccSeqs(\Sigma) \\ &\wedge\ occ=(En_\Sigma(act),obj,t) \\ &\wedge\ act \in ExtActs(\Sigma) \setminus Enabled(occseq) \\ &\wedge\ After(occseq,occ) \\ \Rightarrow\ &occseq \circ occ \in FOccSeqs(\Sigma) \\ &\quad \wedge\ Enabled(occseq \circ occ) = Enabled(occseq) \cup Act(occ) \end{aligned}$$

$$\begin{aligned} &occseq \in FOccSeqs(\Sigma) \\ &\wedge\ occ \in ExtOccs(\Sigma) \\ &\wedge\ Act(occ) \in Enabled(occseq) \\ &\wedge\ After(occseq,occ) \\ \Rightarrow\ &occseq \circ occ \in FOccSeqs(\Sigma) \\ &\quad \wedge\ Enabled(occseq \circ occ) = Enabled(occseq) \setminus Act(occ) \end{aligned}$$

$$\begin{aligned} &occseq \in FOccSeqs(\Sigma) \\ &\wedge\ occ=(Dis_\Sigma(act),obj,t) \\ &\wedge\ act \in Enabled(occseq) \\ &\wedge\ After(occseq,occ) \\ \Rightarrow\ &occseq \circ occ \in FOccSeqs(\Sigma) \\ &\quad \wedge\ Enabled(occseq \circ occ) = Enabled(occseq) \setminus Act(occ). \end{aligned}$$

These rules ensure the 'local rendezvous order' of enable, external, and disable action occurrences, cf. either side of Fig. 2. This order concerns the (rendezvous-) *related action set* of an external action $act \in ExtActs(\Sigma)$, defined by

$$Rend_\Sigma(act) := \{En_\Sigma(act), act, Dis_\Sigma act)\}.$$

Now, the set of possible *occurrence sequences* is defined by

$$OccSeqs(\Sigma) := FOccSeqs(\Sigma)_\infty.$$

The *action restriction* operation on occurrence sequences is defined inductively, and via subsequent canonical extension to infinite words, by :

$\forall w \in Occs(\Sigma)^*, a \in Occs(\Sigma), A \subseteq Actions(\Sigma)$:

$$\varepsilon|_A := \varepsilon \quad \text{and} \quad (wa)|_A := \text{IF } Act(a) \in A \text{ THEN } (w|_A)a \text{ ELSE } w|_A.$$

Behaviour

A *behaviour* over Σ is defined as a subset *beh* of all occurrence sequences over Σ, i.e. $beh \subseteq OccSeqs(\Sigma)$, fulfilling the following requirements, which are presented in an informal manner, due to limited space:

- While an external action is enabled, it can happen at any moment, but only once and unless it is disabled (*rendezvous interval* requirement). It is possible that disabling is not intended, in which case the external action will remain enabled forever if it does not occur. Occurrences of other actions can happen in the interval between the enabling and actual performance or disabling of the action.

  For example, if some occurrence sequence in *beh* consists of an enabling of external action a at time 0 and a disabling of a at time 1, then for all $0 \le t \le 1$, *beh* contains a sequence consisting of $En_\Sigma(a)$ at time 0 and of a itself at time t.
- Whenever the behaviour permits an input action *get*, it is prepared to receive any data object of $Type_\Sigma(get)$ (*free-input* requirement).

The elements of a behaviour are called *behaviour sequences*. Intuitively spoken, *beh* represents the set of 'behaviourally maximal occurrence sequences' and thus gives information on termination: the system may stop working after such a behaviour sequence, even if *beh* contains continuations of it. This corresponds to the distinction of "potentially last actions" in a tree-representation, or the distinction of "potentially maximal words" in a prefix-closed language representation of the behaviour.

2.3 Systems and Conformance

A *system* (over *Signature*(*S*)) is a pair $S := (Signature(S), Behaviour(S))$ such that *Signature*(*S*) is a system signature and *Behaviour*(*S*) is a behaviour over *Signature*(*S*). $Systems(\Sigma)$ is defined as the class of all systems over the signature Σ.

System descriptions usually define behaviour sequences only indirectly, e.g. by logical conditions on sequences or by operational models that generate or accept the desired sequences. Examples of operational system description languages that permit to express systems in our sense are time(d) Petri nets [Mer74, Ram74], timer nets [Bau90], timed automata [AD94], and TTCN. These languages have, for example, much more compact ways of specifying that an action may ocur in a time interval, than our semantic model.

Concrete examples

Let us model simple timers that can be used at most once (to keep things simple). Ideally, a timer can be set and started (say, together, in one atomic action) at any time $t\_set \ge 0$ and will then ring after the chosen duration. Apart from noting that *Start* is input and *Ring* is output, signature definitions are omitted. We use trees, written by means of indentation in the style of TTCN, to denote several occurrence sequences with common prefixes.

- *Behaviour*(*ExactTimer*) consists of the (uncountably many!) sequences

(*EnableStart*, *duration*, 0)
(*Start*, *duration*, *t_set*)
(*EnableRing*, *"Ring"*, *t_set+duration*)
(*Ring*, *"Ring"*, *t_set+duration*)
(*DisableRing*, *dummy*, *t_set+duration*),

where *duration* > 0, and *t_set* ≥ 0.

- *Behaviour*(*InexactTimer*) comprises the sequences

(*EnableStart*, *duration*, 0)
(*Start*, *duration*, *t_set*)
(*EnableRing*, *"Ring"*, *t_set+duration+allowance*)
(*Ring*, *"Ring"*, *t_set+duration+allowance*)
(*DisableRing*, *"Ring"*, *t_set+duration+allowance*),

where *duration*>0, *t_set* ≥ 0, and -1 ≤ *allowance* ≤ 1.

- *Behaviour*(*UnreliableTimer*) comprises the sequences

(*EnableStart*, *duration*, 0)
(*Start*, *duration*, *t_set*)
(*InternalOK*, *dummy*, *t_set+duration*)
(*EnableRing*, *"Ring"*, *t_set+duration*)
(*Ring*, *"Ring"*, *t_set+duration*)
(*InternalBreakdown*, *dummy*, *t_set+duration*),

where *duration*>0 and *t_set* ≥ 0.

2.4 Interactions, system composition, and observations

Interactions and compatibility

Interactions are common external actions of two system signatures Σ_1 and Σ_2,

$$Interacts(\Sigma_1,\Sigma_2) := ExtActs(\Sigma_1) \cap ExtActs(\Sigma_2).$$

Σ_1 and Σ_2 are called *interaction compatible* if they have only interactions, and no other actions, in common, if these interactions have the same type in both signatures, and if these interactions consist of input-output pairs, i.e. if

$$Actions(\Sigma_1) \cap Actions(\Sigma_2) = Interacts(\Sigma_1,\Sigma_2)$$
$$\wedge\ \forall\ int \in Interacts(\Sigma_1,\Sigma_2):\ Type_{\Sigma 1}(int) = Type_{\Sigma 2}(int)$$
$$\wedge\ int \in InpActs(\Sigma_1) \Leftrightarrow int \in OutActs(\Sigma_2),$$

We call a finite set of system signatures $\{\Sigma_1,\ldots,\Sigma_n\}$ *compatible* if its members are pairwise interaction compatible and if, for any three different Σ_i, Σ_j, Σ_k,

$$ExtActs(\Sigma_i,) \cap ExtActs(\Sigma_j) \cap ExtActs(\Sigma_k) = \varnothing.$$

Thus, within a compatible set of signatures, each single interaction is bilateral.

Cooperations and Composite systems

Compatible systems can be composed to form cooperations and larger systems. First we look at the cooperation of systems, for which we define a behaviour, but not a signature; then we define composed systems and a default signature for them.

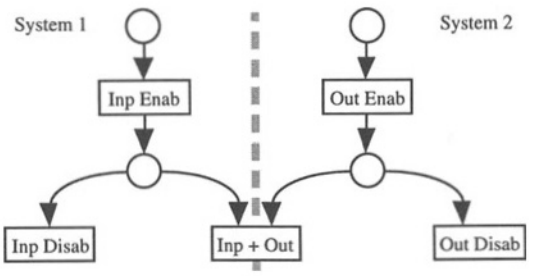

Figure 2 Rendezvous structure.

A *cooperation* is a finite set $\{S_1,\ldots,S_n\}$ of systems with compatible signatures. We prepare the definition of its cooperation behaviour:

$$Occs(S_1,\ldots,S_n) := \cdot\, Occs(S_1) \cup \ldots \cup Occs(S_n)$$

$$PreBehaviour(S_1,\ldots,S_n) := \{\, w \in Occs(S_1,\ldots,S_n)^{\infty} \mid \forall\, S \in \{S_1,\ldots,S_n\}: w|_{Actions(S)} \in Behaviour(S) \wedge \forall\, 1 \le i < j \Rightarrow Time(w_i) \le Time(w_j) \,\}.$$

Again omitting the full formalism, $Behaviour(S_1,\ldots,S_n)$, the *cooperation behaviour* of $\{S_1,\ldots,S_n\}$, is the subset of those *PreBehaviour* sequences in which interactions must occur at the earliest moment, i.e. at the same time as, and in the order behind, the matching enabling by the second partner. This way, interactions shall always occur in both systems in a common, atomic and synchronous step, with the same data object, and such that original local effects (i.e. how behaviour can continue) are achieved in both systems, cf. Figure 2. Internal, intermediate and unmatched external actions happen without any partner, as if the system were running alone.

A *composite system* is a system

$$S_1 \oplus_{\Sigma} \ldots \oplus_{\Sigma} Sn := (\Sigma, Behaviour(S_1,\ldots,S_n)),$$ where Σ is a system signature such that

$$Actions(\Sigma) = Actions(\Sigma_1) \cup \ldots \cup Actions(\Sigma_n)$$ and

$$Type_{\Sigma} = Type_{\Sigma 1} \cup \ldots \cup Type_{\Sigma n}.$$

In principle, input, output, and internal actions can be chosen freely. For our purposes, we choose in the remainder of this paper the *default composite signature* $\Sigma_1 \oplus \ldots \oplus \Sigma_n$ as follows: internal actions and interactions between component systems, as well as their related intermediate actions, are internal. Unmatched external actions form the external actions of the composite signature, and their related intermediate actions play the same role in the composite signature.

Remark:

$Behaviour(S_1,\ldots,S_n)$ is a behaviour over $\Sigma_1 \oplus \ldots \oplus \Sigma_n$.

Application to CTMF

Examples of systems and external actions are the protocol entities and service primitives in OSI terminology [ISO92, ISO94]. For user and provider entities cooperating at the same

interface (called service access point) service primitives are executed in common atomic actions. Service primitives usually have an initiator system, which is also the source of an information exchange encompassing the data object associated with the interaction (output action). Input service primitives may occur with any content of the type specified for the service primitive. Intermediate actions are hidden in the protocol mechanisms: both protocol specifications and TTCN test cases require that e.g. an 'incoming primitive' will only be performed if the entity or test case is ready for it (enabled). TTCN semantics take care that enabled primitives are performed as soon as possible, by continuously repeated attempts. Zero delay is our approximation to a very small delay. Disabling is not uncommon: protocol entities and test cases avoid getting stuck (especially in receive primitives) by taking other action after a timer-controlled period of unsuccessful attempts or waiting.

2.5 Discussion

Our restriction of interactions to identical actions could easily be relaxed by the introduction of name associations or a renaming operator; our conventions merely lead to simpler definitions.

Our restriction of synchronous interactions to be two-sided covers many practical applications, such as OSI modelling, and excludes various implementation problems. Principally, many-sided interactions can be simulated by properly coordinated two-sided interactions. Non-default signatures permit many-sided interactions by repeated system composition.

Some system modelling techniques associate 'partial behaviour descriptions' to interfaces, maybe even different ones to 'each side of an interface.' A priori, systems connected by this interface can only execute a particular subset of the possible sequences of interaction occurrences. Examples are service definitions in OSI. Without loss of generality, we will consider such occurrence sequence restrictions as integrated into the behaviour descriptions of the systems.

Some approaches favour asynchronous interactions between systems S_1 and S_2 via some intermediate machinery such as queues etc. Usually, this machinery can be modelled as a separate system C performing synchronous interactions with S_1 (e.g. put into queue) and with S_2 (e.g. take out of queue). Alternatively the queue in question can be treated as an integral part of S_1 and S_2. Hence, synchronicity does not impose essential restrictions.

Typing restrictions could be relaxed without changing system behaviour. The type associated to an interaction on the 'sending' side could be permitted to be a true subset of the type associated at the 'receiving' side.

If 'action refusal' is to be decided by timeouts, this presupposes exact timing specifications, because otherwise there will always be uncertainty whether specifications are met by an implementation or not. If, on the other hand, refusal is communicated explicitly, this will happen in an interaction occurrence.

Rendezvous, with its earliest timing policy, is a natural cooperation mechanism, e.g. for operating on queues: pushing succeeds as soon as both the queue user pushes an object and the queue has the available space, and pulling succeeds as soon as both the user pulls and the queue offers an object.

3. CONFORMANCE TESTING

3.1 Observations

We consider 'direct' observations first. In 3.3 we will introduce indirect observations by means of direct observations in three-sided cooperations. An *observation* of a system S by a system T is a 'finite common trace' of S-T-interaction occurrences, enriched by the observing system's intermediate action occurrences associated to the interactions, allowing to observe

that an enabled interaction does not succeed within an appropriate time. Letting $\tau := Signature(T)$, the set of all observations is defined as

$$Obs(S,T) := Behaviour(S,T)|_{\cup Rend_\tau[Interacts(S,T)]} \cap Occs(S,T)^* .$$

Note that $Obs(S,T)$ is a subset of $Behaviour(S,T)|_{VisActs(T)}$, the 'visible behaviour' of the observer.

We say that a system S' *cannot be distinguished from* system S, or S' *nodif* S, if, for any system T:

$Obs(S',T) \subseteq Obs(S,T)$.

S' *nodif* S means that no finite observation of S' can ever reveal that it is not S that is being observed.

Even though we do not pursue the topic any further, we could call S and S' *observation-equivalent* if neither of them can be distinguished from the other:

$$S' \text{ obseq } S :\Leftrightarrow S' \text{ nodif } S \wedge S \text{ nodif } S'.$$

For non-deterministic systems, a possible distinction generally cannot be guaranteed (i.e. enforced) by any number of arbitrarily long tests, even if both their number and their length were infinite. Here, we are using the term non-determinism/tic in an informal sense, but leaving aside time, it amounts to the testing non-determinism defined in [Pha94, Pha94a].

An observation example

Let us consider observations of all three example systems in 2.3.1. The system *A_TimerTester* with the behaviour (*sometime* being an arbitrary fixed real number)

(*EnableStart*, 10, *sometime*)
  (*Start*, 10, *sometime*)
    (*EnableRing*, *"Ring"*, *sometime*+9)
      (*Ring*, *"Ring"*, *sometime*+*x*)
      (*DisableRing*, *dummy*, *sometime*+12)
  (*DisableStart*, 10, *sometime*),

where $9 \le x \le 12$. Here, in contrast with the timers, *Start* is output and *Ring* is input. *A_TimerTester* may distinguish *UnreliableTimer* from *InexactTimer*, because *Obs*(*UnreliableTimer*, *A_TimerTester*) \ *Obs*(*InexactTimer*, *A_TimerTester*) contains the observation

(*EnableStart*, 10, *sometime*)
  (*Start*, 10, *sometime*)
    (*EnableRing*, *"Ring"*, *sometime*+9)
      (*DisableRing*, *dummy*, 12).

It can be shown that both *ExactTimer* and, of course, *InexactTimer* cannot be distinguished from *InexactTimer*, to name but two more relationships.

3.2 Tester systems

Obviously, no real test is ever performed through an infinite time period or with infinitely many observed events. Therefore we define a *tester system* T as a system whose occurrence sequences are all finite and take place within uniformly bounded time (see also 2.2), the latter meaning that either *occ* is empty, or

$$\sup\{\ Time(occ_n) - Time(occ_1) \mid (occ_1, \ldots, occ_n) \in Behaviour(T) \wedge n \in \mathbb{N}\ \} \in \mathbb{R}.$$

3.3 Test configuration

Figure 3(a) shows the common situation in protocol testing [ISO91]. Three systems interact in a compound:

- an examinee system *Ex*, in CTMF the 'implementation under test' (IUT),
- a tester system *T*, and
- a testing context *Co*, in CTMF the 'underlying service provider.'

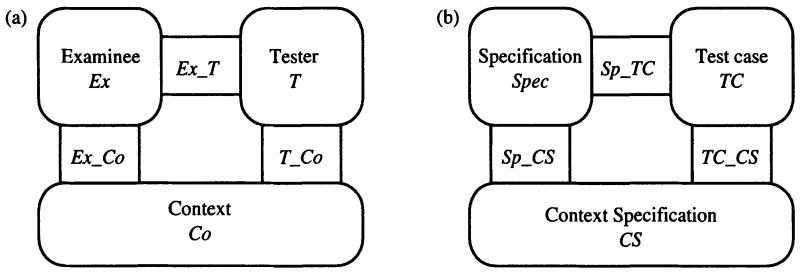

Figure 3 The test configuration (a) and the reference configuration (b).

Ex_T, *Ex_Co*, and *T_Co* represent *Interacts* (*Ex*,*T*), *Interacts*(*Ex*,*Co*), and *Interacts*(*T*, *Co*). By the definition of composite systems, the three are pairwise disjoint. *Ex* may have more external actions than those contained in $Ex\_T \cup Ex\_Co$ – in the cases of 'hidden buttons' or a multi-purpose implementation: a clock radio can pass for a clock (testers will look at the time display, manipulate the clock buttons, listen to the alarm sounding) and for a radio (testers will look at the frequency display, manipulate the radio buttons, listen to the programs). A multi-purpose protocol implementation may have an operator command interface – different from a service access point – to select a protocol or profile or parameter set.

Application to CTMF and non-standardized testing

Two special cases of Figure 3(a) are single-party OSI testing and direct testing.
In single-party testing [BG94, ISO91, ISO94a, Kni93], the tester is considered as a composite system consisting of Upper Tester, Lower Tester, and Test Coordination Procedures. 1992 TTCN, however, treats all three together as a single system. *Ex_Co*, the lower IUT boundary is considered as inaccessible.

In particular, in the Distributed Test Method, *Ex_T* is the Upper Tester PCO, and *T_Co* is the Lower Tester PCO. The standard postulates two opposing queues at each PCO. For the reasons explained in [Bau94], we omitted these queues.

In non-standardized *direct testing* (e.g. first party, development testing), *Co*, *Ex_Co*, and *T_Co* are empty, i.e. *Signature*(*Co*)=(Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø), etc.

3.4. Test cases

OSI conformance testing is black box testing comparing *Ex* in the test configuration with its specification *Spec* in the reference configuration shown in Figure 3(b). Specifications are assumed to be semantically interpreted as systems in our sense.

The tester would like to collect information as to whether *Ex* 'looks to *T* and *Co* as *Spec* would.' Unfortunately, it can usually only observe the composite subsystem $Ex \oplus Co$ at *Ex_T* and *T_Co*, and cannot observe what is happening at *Ex_Co*.

The tester specification is called the *test case TC*. A set of test cases is called a *test suite*. In real testing it is usually assumed that only the specified service primitives are used and that the context and the tester operate as specified. In our model this amounts to $Ex\_T \subseteq Sp\_TC$, $Ex\_Co \subseteq Sp\_CS$, $T\_Co \subseteq TC\_CS$, Co *nodif* CS, and T *nodif* TC.

For the sake of simplicity, we assume that every test case is run with a new copy of Ex, such that we do not have to deal with restoring Ex 'back to its original state.'

3.5 Test outcomes, evidence and verdicts

Test outcomes and conducts

In the following, let us consider the three specifications *Spec*,*TC*, and *CS* as fixed. A *test outcome* of a tester T performed on Ex is an observation of $Ex \oplus Co$ by T:

$Outcomes(Ex,Co,T) := Obs(Ex \oplus Co,T)$.

A test outcome on Ex is an indirect observation, recorded at the surface of the tester, providing generally less information about Ex than a direct observation of the visible surface of the examinee, a *conduct*, as we call it here. A *valid test outcome* is an observation of $Spec \oplus CS$ by TC:

$ValidOut := Obs(Spec \oplus CS,TC)$.

A *valid conduct* is 'what should happen at the surface of Ex' while it cooperates with Co and T, i.e. what could happen at the surface of *Spec*:

$ValCond := Obs(Spec,CS \oplus TC)$.

Due to possible non-determinism in Ex, valid test outcomes may be produced (fortuitously) by a non-conforming examinee. Even worse, due to the possible masking of errors in Co, valid test outcomes may even be produced by a (non-conforming) examinee performing invalid conduct.

If the examinee is always assumed to operate in conjunction with Co, as shown in Figure 3, then we can speak of *relative conformance* w.r.t. Co, which amounts to direct conformance of $Ex \oplus Co$ to $Spec \oplus CS$.

Assessing test outcomes: the evidence function

From test outcomes the tester may try to draw conclusions about the validity of 'what happened at the surface of Ex.' Of course, from an invalid test outcome it can be concluded that *Spec* was not observed, i.e. that Ex does not conform. We say, that the *evidence* of this test outcome is FAIL. If a particular valid test outcome implies that the conduct of Ex was valid, then we assign the evidence PASS. In all other cases, we assign the evidence INCONCLUSIVE. Let for example Co be a channel that delivers all messages (interactions at Sp_CS) truly (in order, in time, without duplication etc.), until it possibly loses one, in which case it immediately apologizes to the tester. As long as the tester obtains a valid sequence of messages it can infer that the examinee's actual conduct in the run of the composite system was valid: the evidence is PASS. If the tester receives an apology after a hitherto valid sequence, it knows that a message was lost, and that this message may have been valid or invalid. In this a case, the evidence is INCONCLUSIVE. Formally,

$Results := \{PASS, FAIL, INCONCLUSIVE\}$

$Evidence: Outcomes(Ex,Co,T) \rightarrow Results$

$Evidence(out) :=$ IF $out \notin ValidOut$ THEN FAIL
ELSE IF $Orig(out) \subseteq ValCond$ THEN PASS
ELSE INCONCLUSIVE,

where *Orig* maps, informally spoken, each test outcome to the set of all observations that can be made by *CS⊕TC* and may lead to this outcome. Therefore, we complete the definition of *Evidence* as follows:

$$Orig(out) := \{w|_{VisActs(Signature(CS \oplus TC))} \mid w \in Behaviour(CS \oplus TC), w|_{VisActs(Signature(TC))} = out\}.$$

Of course, one can replace the unknown function domain *Outcomes*(*Ex*,*Co*,*T*) by any known superset, such as $Behaviour(TC)|_{VisActs(Signature(TC))}$, cf. 2.3.

Verdicts and verdict strategies

One part of the story that we have not told yet is that a test case has one additional external action *Verdict*, which it performs exactly at the end of each behaviour sequence. *Verdict* is typically an interaction with a human test operator or a test case driver and logging process. Occurrence events of *Verdict* have a data object of type *Results* which is determined by the outcome up to this moment. We continue, however, to ignore verdict actions and represent verdict occurrences instead by a *verdict function*

$$Verdi(T): Outcomes(Ex,T) \rightarrow Results,$$

viewing a *tester* as 'a system plus a verdict function.'

Ideally, a test case would use the evidence of the test outcome as the verdict, but often it is difficult or impossible to calculate the evidence. A *verdict strategy* is a mapping *Strat* from *Results* to *Powerset*(*Results*). *Verdi*(*T*) *complies* with *Strat* if

$$\forall\ outc \in Outcomes(Ex,T): Verdi(T)(outc) \in Strat(Evidence(outc)).$$

Of course, the *perfect* strategy is $PerfStrat(x)=\{x\}$. Considering *Results* as ordered by FAIL < INCONCLUSIVE < PASS, *Strat* is

- *acceptable* if $\forall x \in Results, y \in Strat(x): \ y \geq x,$
- *wary* if $\forall x \in Results, y \in Strat(x): \ y \leq x,$ and
- *strict* if $Strat(\text{FAIL}) = \{\text{FAIL}\}$

Non-acceptable verdicts amount to false accusations. Strict verdicts discover all 'errors,' i.e. invalid outcomes. Non-strict test cases are often much easier to write: at the extreme, the tester could immediately terminate with PASS. We assume that all test cases of a test suite follow the same verdict philosophy, because otherwise test results are very hard to interpret for anyone without detailed knowledge of the test suite.

As the present paper is apparently among the first to fully formalize the evidence and verdict strategy concepts, research on how to approach perfect verdicts is only beginning. In [ISO91], the notion of verdict was based on a vaguely described concept of test purpose [Bau94, CL93].

3.6 System parameters and non-determinism

Specification parameters

Protocol specifications often contain 'static nondeterminism' in the form of protocol *parameters* and options, where the latter can be considered a special cases of the former. Parameters are replaced (i.e. values assigned to them) either rigidly by the implementor or controllably by the operator of the implementation. In CTMF, parameter values are assigned in ICS and IXIT documents. Usually a test case has a subset of the specification parameters as its own parameters.

A *parameter set* (over M, cf. 2.1) is a triple

$Params = (ParNames, ParTypes, ParDecl)$,

where *ParNames* is a finite set, *ParTypes* a set of types, and *ParDecl* is a mapping from *ParNames* to *ParTypes*. A *parameter assignment* for *Params* is a mapping

$Assig: ParNames \rightarrow \cup\, ParTypes$ with $\forall parm \in ParNames: Assig(parm) \in ParDecl(parm)$.

Let *Assigs*(*Params*) be the set of all parameter assignments for *Params*. A *parameterized* (by *Params*) *system* over Σ is a mapping

$ParSys: Assigs(Params) \rightarrow Systems(\Sigma)$.

Parameterized systems are usually defined inductively as terms with variable names in *ParNames*, i.e. syntactically. Then *ParSys* is also inductively defined by the evaluation function on terms canonically associated with the parameter assignments. Once the parameters have been replaced by values, we are back to the ordinary systems. Still, the generation of parameterized test cases is a challenge.

Additional parameters

Dynamic non-determinism (DND) refers to internal behaviour alternatives remaining in *Spec*, after all parameters have been assigned. As shown in 3.1, DND decreases the diagnostic powers of testing. An incorrectly implemented alternative may just 'happen not to occur' during the performance of a test suite, and the error will remain undetected.

A way out of this dilemma is the introduction of additional parameters, which allow to specify on a voluntary basis, but with a binding effect, in which way the examinee has less DND than *Spec*. This additional information increases testability and the value of positive test results, but also the chance of failing test cases.

In CTMF, IXITs are the documents in which additional parameters are assigned values. A typical case is assigning a time limit to reaction times of the examinee, if such limits are unreasonably high for practical testing, or were plainly forgotten in *Spec*.

An *additional parameterization* of *Spec* with *AddParms* is a parameterized system

$SpecPlus: Assigs(AddParms) \rightarrow \{S \in Systems(\Sigma) \mid S\ nodif\ Spec\}$ such that (1)

$Spec \in SpecPlus(Assigs(AddParms))$. (2)

Practically expressed, assignment shall not violate *Spec* (1), and the test client may always choose not to restrict DND at all (2).

Test case behaviour may depend on the additional parameters, and test verdicts must respect the additional parameter assignment *assig*. Test cases whose verdicts do not follow the test suite verdict strategy w.r.t. $Spec' = SpecPlus(assig)$ are not *selected* for execution. This is often the case where the test case 'concentrates on' a special DND-alternative, but the examinee is allowed to avoid this alternative, even according to *assig*.

With the appropriate procedural measures, additional parameter assignments might even be permitted to vary from test case to test case, but we omit the details here.

3.7 Correctness criteria for test cases

We summarize some of the mentioned or implicit criteria that a test case *TC* must meet in order to be considered correct. We assume that a verdict strategy has been defined for the entire test suite. We consider *Spec* and *CS* as given. We add corresponding informal requirements in parentheses.

- The signatures of *Spec*, *CS* and *TC* must be compatible. (In protocol testing: *TC* must use the right set of service primitives.)
- The parameter set of *TC* must be a (parameter) subset of the parameter set of *Spec*, if applicable, united with the standardized additional parameter set. (*TC* must not have any undefined parameters.)
- The verdict strategy must be acceptable and strict. *Verdi*(*TC*) must comply with the verdict strategy. (*TC* must deliver valid verdicts.)

4 CONCLUSION AND OUTLOOK

We have defined a theoretical framework for conformance testing, giving precise and consistent meanings to many practically motivated CT notions. More of them will be formalized in forthcoming papers. Moreover, we have introduced formal correctness criteria for test cases. We have contrasted our approach with comparable existing approaches.

A clean and comprehensive mathematical framework for behaviour testing is the necessary foundation for

- the generation of correct test cases,
- the validation of existing test cases,
- coverage optimization, and
- protocol design rules for testability.

In this paper, we have contributed to the first two objectives. We have neither dealt with the last two, nor with the topics of

- conformance requirements and test purposes,
- alternatives between regular and exceptional behaviour,
- concurrent TTCN,
- probability in conformance testing.

We expect to do so in the future.

5 ACKNOWLEDGEMENTS

The author gratefully acknowledges numerous hints and improvements suggested by Helmut Wiland and the anonymous referees, as well as fruitful discussions with Alfred Giessler, Christa Paule, Gunther Gattung, and Olaf Henniger.

6 REFERENCES

[AD94] R. Alur, D.L. Dill: A theory of timed automata, *Theoretical Computer Science* 126, 1994, 183-235

[Bae94] U. Bär: *OSI-Konformitätstests: Validierung und qualitative Bewertung*, VDI Verlag, 1994

[Bau90] B. Baumgarten: *Petri-Netze, Grundlagen und Anwendungen*, B.I. Wissenschaftsverlag, 1990

[Bau94] B. Baumgarten: Open Issues in Conformance Test Specification, *7th IFIP WG6.1 International Workshop on Protocol Test Systems*, Proceedings, Tokyo, 1994

[BG94] B. Baumgarten, A. Giessler: *OSI Conformance Testing Methodology and TTCN*, North-Holland, 1994

[Bri89] E. Brinksma et al.: A Formal Approach to Conformance Testing, *Protocol Test Systems, North-Holland*, 1989

[BW95] B. Baumgarten, H. Wiland: What is a correct test case? Elements of a Petri net oriented theory of protocol testing, *Petri Nets applied to Protocols*, Proceedings of a Workshop of the 16th ICATPN, Torino, 1995

[CL93] S. T. Chanson, Qin Li: On Inconclusive Verdict in Conformance Testing, *Protocol Test Systems, V*, North-Holland, 1993, 81-92

[EM85] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specifications, Vol. 1*, Springer, 1985

[ISO91] ISO/IEC IS 9646: Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework, 5 parts, 1991/2

[ISO92] ISO/IEC DIS 7498-1: Information Technology – Open Systems Interconnection – Reference Model, Part 1: Basic Reference Model, 1992 (Revision of 1984 IS 7498)

[ISO94] ISO/IEC JTC 1, IS 10731: Information Technology – Open Systems Interconnection – Basic Reference Model – Conventions for the definition of OSI services , 1994

[ISO94a] ISO/IEC IS 9646: Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework, 7 parts, 1994–19XX

[ISO95] ISO/IEC JTC1/SC21/P.54.1: Framework: Formal Methods in Conformance Testing, Interim version, 1995

[Kni93] K. Knightson: *OSI Protocol Conformance Testing*, McGraw-Hill, 1993

[Mer74] P.M. Merlin: A Study of the Recoverability of Computing Systems, Irvine; Univ. California, Dept. of Information and Computer Science, TR 58, 1974

[Pha94] M. Phalippou: *Relations d'implantation et hypothèses de test sur des automates à entrées et sorties*, Ph.D. thesis, Université de Bordeaux, no. d'ordre 1112, 1994

[Pha94a] M. Phalippou: Executable testers, *Protocol Test Systems, VI*, North-Holland, 1994

[Ram74] C. Ramchandani: *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, MIT, Project MAC, Technical Report 120, Feb. 1974

[Tre92] J.G. Tretmans: *A Formal Approach to Conformance Testing*, CIP - Gegevens Koninklijke Bibliotheek, Den Haag, 1992

[Tre94] J.G. Tretmans: A Formal Approach to Conformance Testing, *Protocol Test Systems, VI*, North-Holland, 1994

7 BIOGRAPHY

B. Baumgarten received the degree in mathematics in 1973 and the doctorate in 1976 at the Technical University of Darmstadt, Germany. In 1977, he joined GMD – German National Research Center for Information Technology, where he has taken part in several research and development projects in the fields of distributed systems, formal description techniques, and testing. Since 1988, he has been involved in OSI conformance testing standardization. He is author and co-author of books on Petri nets and on conformance testing.