Revised version of the paper published in: B. Jonsson, J. Parrow, B. Pehrson, eds., Protocol Specification, Testing and Verification, XI, Elsevier Science Publishers B.V. (North-Holland), Amsterdam (1991), 249-264.

Conformance relation, associated equivalence, and minimum canonical tester in LOTOS

Guy Leduc

Research Associate of the National Fund for Scientific Research (Belgium)
Université de Liège, Institut d'Electricité Montefiore, B 28, B-4000 Liège 1, Belgium

Abstract

We first study the *conf* relation proposed by E. Brinksma and G. Scollo to formalize testing conformance. It is well-known from their work that, in order to test whether an implementation *I* is a valid implementation of a specification *S* (i.e. *I conf S*), it suffices to build, from *S*, a canonical tester *T (S)* such that, when *T (S)* is synchronized with an implementation *I*, it always reaches a correct final state if, and only if, *I conf S*. For instance, if *I* is not a valid implementation of *S*, the canonical tester *T (S)* may deadlock with *I* before reaching a correct final state.

We put into evidence the role of the equivalence relation, *conf-eq*, associated naturally with *conf*. An important result states that if S_1 *conf-eq* S_2, their canonical testers T_1 and T_2 must also satisfy T_1 *conf-eq* T_2, and reversely. Therefore, the best approach is to define the canonical tester modulo *conf-eq*, whereas it is currently defined modulo the testing equivalence *te*. Taking into account that *conf-eq* is weaker than *te*, we were able to propose a *minimum* canonical tester which is simpler than *T (S)*: unlike *T (S)*, it may have fewer traces than the specification *S*. The term minimum means that no trace from this tester can be deleted without losing the exhaustive test property or, stated otherwise, without taking the risk of accepting an invalid implementation (in the *conf* sense).

1. Introduction

This paper deals on the one hand with the implementation of formal specifications expressed in the LOTOS language [ISO 8807, BoB 87], and on the other hand with the conformance testing of implementations. In this context, several problems are still open. We will summarize them before introducing our work.

The first difficulty is due to the fact that the implementation process involves on the one hand a reference formal specification, and on the other hand a physical realization of this specification. In order to solve this problem, one does not consider the physical realization itself, but instead a model of this realization [BrS 86]. We postulate that this model, that we call *implementation specification* or simply *implementation*, will be expressible in LOTOS, like the specification itself. This makes it possible to reason on the implementation process in a single formalism, and to introduce an implementation concept in a formal theory. It may seem unrealistic to use LOTOS as a model of a physical realization; indeed, this is not the ideal application field for this language designed to describe abstract specifications of ISO standards. However, we can substantiate our postulate as follows. Several LOTOS specification styles proposed in [VSv 88] allow an adjustment of the abstraction level of the specification. LOTOS may even be used like a programming language such as C [MaS 90], i.e. the structure and the level of detail (or abstraction) are close to a C program. The generated C code from such a specification is only just less efficient than a program directly written in C with the same structure. In conclusion, we will work with models of realizations, i.e. with implementations, in LOTOS.

Using only one formalism helps in solving the second problem: the nature of the link which should hold between a valid implementation and its formal specification. The problem is twofold: first, we must find the criteria which allow a characterization of this link, and second we have to express them formally. These formalizations may be separated into two categories: either the validity is modelled by an appropriate equivalence, or it is modelled by a non necessarily symmetric relation such as a preorder.

Validity as an equivalence

Equivalence relations play a central role in process algebraic languages such as CCS [Mil 89], ACP [BeK 85] or LOTOS [ISO 8807, BoB 87] for reasoning about systems and analysing their properties. Many different notions of equivalence have been proposed, and this is not surprising since there are many properties which may be relevant in the analysis of distributed systems [dNi 87]. However, these equivalences are almost always based on some observation criterion, i.e. two systems are considered equivalent if, and only if, they are indistinguishable by external observation of a certain kind. The main idea is indeed to discriminate systems on their external behaviour only, and thus abstract away from internal details. However, there remain many reasonable ways to observe systems [dNi 87, vGl 90, Led 90]. Examples of such equivalences are the observation equivalence [Par 81, Mil 89] or the testing equivalences [BHR 84, dNi 84, BSS 87, Hen 88].

This approach ensures that the implementation will behave (externally) exactly as described in the specification. The structure of the specification may of course change during the stepwise process in order to be closer to an implementation structure for instance; however nothing changes externally, i.e. other co-operating or communicating systems are not able to distinguish between any possible final stages of the design. In LOTOS, several specification styles [VSv 88] have been identified which may be used to describe the same system at different stages of its design.

Validity as a non necessarily symmetric relation

Even if this view is attractive, there is in our opinion a more appropriate view which takes into account the asymmetric character of the implementation process. Instead of considering that any final stage should be somehow externally equivalent to the specification, the idea is to define a less restrictive and usually asymmetric relation. These relations are referred to as *implementation relation*s in the sequel.

With process algebraic techniques, such relations have been less studied than equivalences. There is no well-established opinion on the desired nature of these implementation relations, but some trends exist however. For instance, it is usually admitted that an implementation may be more deterministic than the specification. With this view, an implementation relation would be better formalized by a preorder (i.e. a reflexive and transitive relation) than by an equivalence. A preorder, having an asymmetric character, defines an ordering among systems. If this preorder is chosen carefully, it can be interpreted as an *implementation relation*, i.e. if two systems A and B are such that B *is less than* A according to this ordering, then it means that in a certain sense B *implements* A, or B *is a valid implementation of* A. For instance, a criterion which may be formally expressed by such a relation is the reduction of the nondeterminism, i.e. B is a valid implementation of A if, and only if, B is obtained from A by resolving some (voluntarily) open nondeterministic choices of A.

Some implementation relations based on this idea have been defined. They are often (but not always) preorders. In CSP [BHR 84, Hoa 85], such a relation had already been introduced as a preorder of the failures equivalence. In CCS [dNH 84], other preorders of various testing equivalences were introduced. And finally, in LOTOS [BSS 87], preorders of the testing equivalence, as well as a conformance relation, have been defined. This is precisely on this conformance relation, denoted *conf*, that this paper will focus. In [Led 91b], another implementation relation, denoted *conf**, is also proposed and studied.

The concept of an implementation relation has also been introduced in other formal models. The implicit refinement relation of Dijkstra's wp-calculus [Dij 76] is a widespread example of this type of implementation relation. With logic-based specifications, B is usually considered as an implementation of A iff $B \Rightarrow A$, i.e. B satisfies more properties than A [ChM 89]. With state machines or I/O automata, a specification B may be considered as an implementation of the specification A iff there is an appropriate mapping from B to A [LyF 81, Lam 83, LaS 84, LyT 87, AbL 88, Mer 89]. With modal transition systems (i.e. an extension of a LTS with necessary and admissible transitions) [Lar 89], B is considered as an implementation of A iff there exists a refinement relation between B and A.

The third problem related to the implementation of formal specification is conformance testing. A general framework and methodology of conformance testing is studied within ISO [ISO 9646, Ray 87]. The crucial problem which remains unsolved is the generation of an adequate set of testing scenarios from the formal specification [FaL 87, Sar 87, SBC 87, SaD 88, Hog 89, SiL 89, VCI 89, PhG 91].

In LOTOS, the concept of a canonical tester associated with a specification has been defined and studied in [BrS 86, Bri 87, Bri 88] and extended and brought into play in [Ald 89, Wez 89, WBL 91]. The canonical tester is itself a LOTOS specification which describes how to test the implementations and find whether they are conforming to the specification. There is no particular selection of test scenario in the design of the canonical tester: it is designed to test the implementations exhaustively; it is somehow a theoretical upper bound on the testing process.

The canonical tester is based on the *conf* relation proposed to formalize conformance testing in [Bri 88]. This means that the proposed test method, which is based on a specification *S*, rejects any implementation *I* which does not satisfy the rule *I conf S*. The proposed technique consists in building, from *S*, a canonical tester *T (S)* such that, when *T (S)* is synchronized with an implementation *I*, it **always** reaches a correct final state[1] if, and only if, *I conf S*. For instance, if *I* is not a valid implementation of *S*, the canonical tester *T (S)* **may** deadlock with *I* before reaching a correct final state.

This property of the canonical tester illustrates that the testing problem and the implementation (or conformance) relation are closely related, since the definitions of *conf* and *T* are. Besides, this is very logical since the conformance tests aim primarily at selecting *valid* implementations.

In this paper, we prove that some traces of the canonical tester are useless and may be deleted. Intuitively, the parts of test scenarios which are deleted always yield inconclusive tests. The resulting tester is simpler than the canonical tester, but may still test exhaustively the implementations. The test scenarios of this new tester are still too numerous, but this problem of an adequate test selection is not dealt with in this paper. The new tester may be considered as the best upper bound on exhaustive testing.

Content of the paper

We first present briefly some existing non-symmetric relations [BSS 87], such as *red*, *ext*, and the conformance relation *conf*.

We then show how these relations define equivalence relations in a very natural way. We use for that purpose a generic relation *imp* which may be instantiated by the various relations *red*, *ext* and *conf*. The equivalence associated in this way with *red* and *ext* is the testing equivalence *te* [BSS 87]. The equivalence associated with *conf* is denoted *conf-eq* and is weaker than *te*.

The canonical tester *T (S)* presented in [Bri 88] is, as already stated, based on the *conf* relation. *T (S)* is however defined modulo *te*; which could seem relatively inconsistent, given that the equivalence associated with *conf* is weaker than *te*.

From this observation, we will prove that we can define *T (S)* modulo *conf-eq*, i.e. if *T* is the canonical tester of the specification *S*, and if *T'* *conf-eq* *T*, then *T'* may also play the role of a canonical tester of *S*: viz. *T'*, when synchronized with an implementation *I*, **always** reaches a correct final state if, and only if, *I conf S*.

This characterization of the canonical tester modulo *conf-eq* has an advantage: it leaves more freedom to define *T (S)*. This allows us to find a minimum canonical tester, denoted $T_m(S)$, which, unlike *T (S)*, may have fewer traces than the specification *S*.

In fact, no trace of $T_m(S)$ may be deleted without losing the certainty of a correct verdict of conformance: deleting a trace from $T_m(S)$ may lead to accept as valid an invalid implementation of *S*.

The definition of $T_m(S)$ in this paper is different from the $T_m(S)$ which has been proposed in [Led 91c]. In [Led 91c], $T_m(S)$ was not the minimum canonical tester, but simply a simplification of *T (S)*. This error has been pointed out to us by Ed Brinksma, and corrected in this version. A consequence of this error is also that the sufficient condition mentioned at the end of [Led 91c] is only valid for the simplification of *T (S)* proposed there; it is no more true for the minimum canonical tester presented here.

[1] A correct final state of *T (S)* is a state where *T (S)* is supposed to stop, i.e. *T (S)* was designed in such a way that, after some sequences, it may behave like *stop*. When *T (S)* has not reached such a final state but is in deadlock with an implementation *I*, this is considered as an invalid final state, i.e. when the composed process *I* || *T (S)* is in such a final state, *T (S)* is still able to execute some actions.

2. Definitions and properties of well-known implementation relations

In LOTOS, some non symmetric relations have been proposed in [BrS 86, BSS 87]. We briefly recall them in a trace-refusal formalism similar to the failures used in CSP [Hoa 85].

Notations 2.1

L is the alphabet of observable actions, i is the internal (i.e. unobservable) action, and δ is the successful termination action.

$P — a \rightarrow P'$ means that process P may engage in action a and, after doing so, behave like process P'.

$P — i^k \rightarrow P'$ means that process P may engage in the sequence of k internal actions and, after doing so, behave like process P'.

$P — a.b \rightarrow P'$ *means* $\exists\, P''$, *such that* $P — a \rightarrow P'' \ \wedge \ P'' — b \rightarrow P'$.

$P = a \Rightarrow P'$ where $a \in L$, means $\exists\, k_0, k_1 \in N$, such that $P — i^{k_0}.a.i^{k_1} \rightarrow P'$.

$P = a \Rightarrow$ where $a \in L$, means that $\exists\, P'$, such that $P = a \Rightarrow P'$, i.e. P may accept the action a.

$P = a \neq >$ where $a \in L$, means $\neg\, (P = a \Rightarrow)$, i.e. P cannot accept (or must refuse) the action a.

$P = \sigma \Rightarrow P'$ means that process P may engage in the sequence of observable actions σ and, after doing so, behave like process P'. More precisely, if $\sigma = a_1 .. a_n$ where $a_1, \ldots a_n \in L$:

$\exists\, k_0, \ldots k_n \in N$, such that $P — i^{k_0}.a_1.i^{k_1}.a_2 \ldots a_n.i^{k_n} \rightarrow P'$.

$P = \sigma \Rightarrow$ means that $\exists\, P'$, such that $P = \sigma \Rightarrow P'$.

P *after* $\sigma = \{P' \mid P = \sigma \Rightarrow P'\}$,

i.e. the set of all behaviour expressions (or states) reachable from P by the sequence σ.

$Tr\ (P)$ is the trace set of P, i.e. $\{\sigma \mid P = \sigma \Rightarrow\}$; $Tr\ (P)$ is a subset of L^*.

$\sigma_1 \leq \sigma_2$ iff $\exists\, \sigma_3 \in L^*$, such that $\sigma_1.\sigma_3 = \sigma_2$ i.e. σ_1 is a prefix of σ_2.

$\sigma_1 < \sigma_2$ iff $\exists\, \sigma_3 \in L^+$, such that $\sigma_1.\sigma_3 = \sigma_2$ i.e. σ_1 is a strict prefix of σ_2.

$Out\ (P, \sigma)$ is the set of possible observable actions after the trace σ,

i.e. $Out\ (P, \sigma) = \{a \in L \mid \sigma.a \in Tr\ (P)\}$.

$Ref\ (P, \sigma)$ is the refusal set of P after the trace σ, i.e.

$Ref\ (P, \sigma) = \{X \subseteq L \mid \exists\, P' \in P$ *after* σ, *such that* $P' = a \neq >, \forall\, a \in X\}$;

$Ref\ (P, \sigma)$ is a set of sets and a subset of $\wp\ (L)$, the power set of L (i.e. the set of subsets of L).

A set $X \subseteq L$ belongs to $Ref\ (P, \sigma)$ iff P may engage in the trace σ and, after doing so, refuse every event of the set X.

Some possible interpretations of the notion of validity have been presented and formalized in [BrS 86, BSS 87] by means of three basic relations, viz. *conf*, *red* and *ext* , and an equivalence, viz. *te* .

Definitions 2.2

Let P_1 and P_2 be processes.

P_1 *conf* P_2 *iff* $\forall\ \sigma \in Tr\ (P_2)$, *we have* $Ref\ (P_1, \sigma) \subseteq Ref\ (P_2, \sigma)$ or equivalently,

iff $\forall\ \sigma \in Tr\ (P_2) \cap Tr\ (P_1)$, *we have* $Ref\ (P_1, \sigma) \subseteq Ref\ (P_2, \sigma)$

because if $\sigma \in Tr\ (P_2) - Tr\ (P_1)$, then $Ref\ (P_1, \sigma) = \varnothing$

Intuitively, P_1 *conf* P_2 iff, placed in any environment whose traces are limited to those of P_2, P_1 cannot deadlock when P_2 cannot deadlock. Stated otherwise, P_1 deadlocks less often than P_2 in such an environment. This relation has been taken as the formal basis of conformance testing in [Bri 88], and is denoted the *conformance* relation.

P_1 *red* P_2 iff (i) $Tr\ (P_1) \subseteq Tr\ (P_2)$, and
(ii) P_1 *conf* P_2

Intuitively, if P_1 *red* P_2, P_1 has less traces than P_2, but even in an environment whose traces are limited to those of P_1, P_1 deadlocks less often. *Red* is the *reduction* relation

P_1 *ext* P_2 iff (i) $Tr(P_1) \supseteq Tr(P_2)$, and
(ii) P_1 *conf* P_2

Intuitively, if P_1 *ext* P_2, P_1 has more traces than P_2, but in an environment whose traces are limited to those of P_2, it deadlocks less often. *Ext* is the *extension* relation.

te = *red* $\cap$ *red* $^{-1}$ = *ext* $\cap$ *ext* $^{-1}$ This is the testing equivalence.

Propositions 2.3 [BrS 86]

(i) *conf* $\supset$ *red*
(ii) *conf* $\supset$ *ext*
(iii) *red* and *ext* are preorders
(iv) *conf* is not transitive

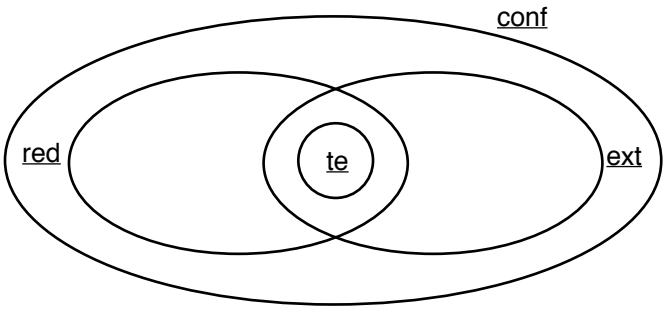

Figure 2.1: *te*, *red*, *ext*, *conf*

Figure 2.1 illustrates the links between these relations.

3. Implementation relation and associated equivalence

In this section, we use a generic relation *imp* in order to model any implementation relation, except that it should be reflexive as explained hereafter. Remember that an implementation relation expresses formally the notion of validity w.r.t. a specification.

Imp must be reflexive because the specification is a valid implementation of itself. Therefore, in the sequel we consider that *imp* is reflexive.

By contrast, *imp* is not required to be symmetric, since the implementation and the specification are not interchangeable in general.

The transitivity of *imp* is more debatable: should one require that a valid implementation of a valid implementation be again a valid implementation? If *imp* is not transitive, a valid implementation cannot be used as an intermediate specification; but it is not its role anyway, since by definition the implementation is the last formal stage of the implementation process. Therefore, we will not restrict ourselves to transitive implementation relations. A more detailed study of this problem is presented in [Led 91a]. Besides, let us note that the *conf* relation is not transitive.

We will show how *imp* induces naturally an equivalence, denoted *imp-eq*.

Definition 3.1

S_1 *imp-eq* S_2 *iff* $\{I \mid I \text{ imp } S_1\} = \{I \mid I \text{ imp } S_2\}$

where $\{I \mid I \text{ imp } S\}$ denotes the set of processes I which are valid implementations of S according to the relation *imp*.

Intuitively, two specifications are equivalent if, and only if, they determine exactly the same set of valid implementations in the sense of *imp*.

It is obvious that *imp-eq* is reflexive, symmetric and transitive. *Imp-eq* is therefore an equivalence relation whatever *imp* is (even if *imp* is not reflexive).

If *imp* is considered as the reference relation, this equivalence has a fundamental nature in the sense that no distinction should be made between two specifications allowing the same valid imple-

mentations. The equivalence relation is derived naturally from the implementation relation. The contrary is not always possible.

One might think that *imp-eq* is the equivalence defined by $imp \cap imp^{-1}$, i.e. two specifications are equivalent if, and only if, each one is a valid implementation of the other one. However, $imp \cap imp^{-1}$ is not necessarily an equivalence.

Proposition 3.2

$imp\text{-}eq \subseteq imp \cap imp^{-1}$

The proof is immediate because S_1 *imp-eq* $S_2 \Rightarrow (S_1$ *imp* $S_2 \wedge S_2$ *imp* $S_1)$. This is derived from the definition of *imp-eq* and the property of reflexivity of *imp*.

Propositions 3.3

imp is transitive $\Rightarrow$ *imp-eq* = $imp \cap imp^{-1}$

In this case, two specifications are equivalent if, and only if, each one is a valid implementation of the other one.

So, when *imp* is transitive, *imp-eq* can be defined advantageously as follows:

$$S_1 \text{ imp-eq } S_2 \quad \textit{iff} \quad S_1 \text{ imp } S_2 \ \wedge \ S_2 \text{ imp } S_1.$$

4. Equivalence relations associated with *red*, *ext* and *conf*

Proposition 4.1 [BrS 86]

Red-eq = *ext-eq* = *te*

Definition 4.2

S_1 *conf-eq* S_2 *iff* $\{I \mid I \text{ conf } S_1\} = \{I \mid I \text{ conf } S_2\}$

This definition is a simple instantiation of definition 3.1.

In order to study the nature of the *conf-eq* equivalence associated with *conf*, we will first instantiate some results from section 3, and then propose another definition of *conf-eq*.

Propositions 4.3

(i) *conf-eq* $\subset$ *conf* $\cap$ *conf* $^{-1}$

(ii) *conf-eq* $\supset$ *te*
i.e., *conf-eq* is weaker than the testing equivalence

(iii) *conf-eq* $\cap$ *trace-eq* = *conf* $\cap$ *conf* $^{-1}$ $\cap$ *trace-eq* = *te*
or equivalently,
$\forall P, Q$, *we have*
P *conf-eq* $Q \wedge (Tr(P) = Tr(Q)) \Leftrightarrow P$ *conf* $Q \wedge Q$ *conf* $P \wedge (Tr(P) = Tr(Q)) \Leftrightarrow P$ *te* Q
For processes with equal trace sets, *conf-eq* and *te* are equal.

(i) is directly derived from proposition 3.2, and the proofs of (ii) and (iii) are given in [Led 91a]

All these results are summarized in figure 4.1. The shaded area is exactly the testing equivalence. In [Led 90], examples are provided to prove that all the inclusions are strict, i.e. no area in the figure is empty.

We give now a more useful definition of *conf-eq*.

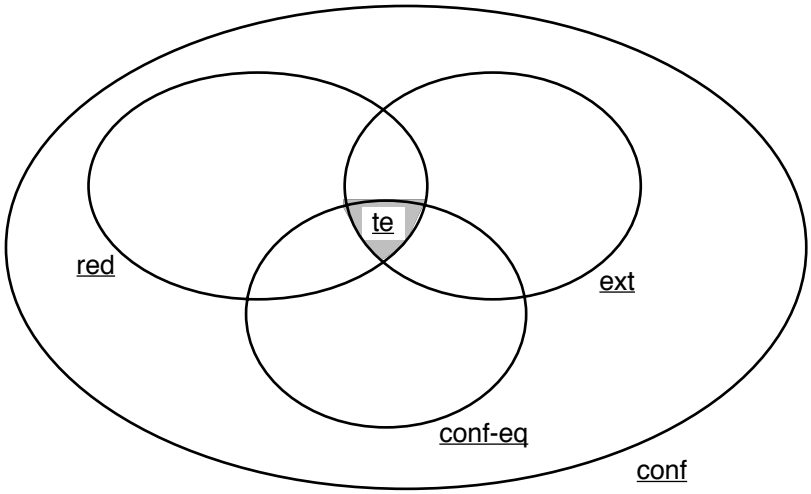

Figure 4.1: *conf-eq* w.r.t. other relations

Proposition 4.4 [Led 90]
P conf-eq Q iff

(a) P conf Q ∧ Q conf P, and (i.e. $\forall \sigma \in Tr(P) \cap Tr(Q),\ Ref(P, \sigma) = Ref(Q, \sigma)$)

(b) $\forall \sigma \in Tr(P) - Tr(Q)$, *we have* $L \in Ref(P, \sigma)$, *and*

(c) $\forall \sigma \in Tr(Q) - Tr(P)$, *we have* $L \in Ref(Q, \sigma)$

5. *conf-eq* versus *te*

If *conf* is adopted as reference implementation relation, the testing equivalence is not adequate because, as stated in proposition 4.3 (ii), it is stronger than *conf-eq*: some processes which are not testing equivalent may define exactly the same set of conforming implementations.

Consider the following example where *P conf-eq Q*, but ¬ *(P te Q)*:

P = a; stop and *Q = (a; stop [] a; b; stop)*.

If *P* and *Q* are two specifications, they define exactly the same set of valid implementations (in the *conf* sense), in particular *P conf Q* and *Q conf P*.

Note however that if *P = a; stop* and *Q = (a; stop [] a; b; c; stop)*, we do not have *P conf-eq Q* any more, because *a; b; stop* is a conforming implementation of *P* but not of *Q*. This can be explained intuitively as follows: a valid implementation of *Q* may, or may not, accept *b* after *a*, but if *b* has been accepted, then, unlike valid implementations of *P*, it cannot refuse *c* just after.

6. A minimum canonical tester

In [Bri 87] the concept of a canonical tester of a specification *S* has been introduced and denoted *T (S)*. This *T (S)* is the specification of a tester which, when synchronized with an implementation *I*, may deadlock with *I* before reaching a correct final state if and only if *I* does not conform to *S*. It has been proved that *T (S)*, as defined in [Bri 87] (see definition 6.1 below), always exists and is unique up to testing equivalence. That is, if another process *Q* satisfies the criteria to be a canonical tester of *S*, then *Q te T (S)*. The purpose of this section is an extension of these results.

Definition 6.1 (the canonical tester) [Bri 87]
Let *S* be a specification, *T (S)* is defined implicitly as a solution *X* satisfying the two following equations:

(i) $Tr(X) = Tr(S)$

(ii) $\forall P,\ P \ conf\ S \ \Leftrightarrow \ (\forall \sigma \in L^*$, *we have* $(L \in Ref(P \,||\, X, \sigma) \Rightarrow L \in Ref(X, \sigma))\ )$

This definition characterizes $T(S)$ by its traces and refusals[1] after each trace; which defines $T(S)$ modulo $\underline{te}$. (i) indicates that $T(S)$ is defined so that it can test all the traces of S. Moreover, (ii) expresses that if the canonical tester is synchronized with an implementation P not conforming to S, then $P \,||\, T(S)$ **may** engage in a sequence σ and then deadlock after σ, whereas the tester is still offering interactions. Reversely, if the canonical tester is synchronized with an implementation P conforming to S, then a deadlock is only possible if the tester has reached a correct final state (which is, formally speaking, a deadlock of $T(S)$).

Example of a canonical tester
Let $S := a; b; exit \,[]\, a; c; stop$.
We give without proof that $T(S) := a; (b; exit \,[]\, c; stop)$ modulo $\underline{te}$.
The correct final states of $T(S)$ are reached after the sequences $ab\delta$ and ac.
Consider the following implementations for illustration:

$I_1 := a; c; stop$	is a valid implementation of S, since $I_1 \,||\, T(S)$ may only deadlock after the sequence ac which also leads to a deadlock of $T(S)$
$I_2 := a; b; exit$	is a valid implementation of S, since $I_2 \,||\, T(S)$ may only deadlock after the sequence $ab\delta$ which also leads to a deadlock of $T(S)$
$I_3 := a; b; stop$	is not a valid implementation of S, since $I_3 \,||\, T(S)$ may deadlock after the sequence ab which does not lead to a deadlock of $T(S)$: $T(S)$ offers δ.
$I_4 := T(S)$	is a valid implementation of S, since $I_4 \,||\, T(S)$ may only deadlock after the sequences $ab\delta$ or ac which also lead to deadlocks of $T(S)$
$I_5 := S$	is a valid implementation of S, since $I_5 \,||\, T(S)$ may only deadlock after the sequences $ab\delta$ or ac which also lead to deadlocks of $T(S)$
$I_6 := a; stop$	is not a valid implementation of S, since $I_6 \,||\, T(S)$ may deadlock after the sequence a which does not lead to a deadlock of $T(S)$: $T(S)$ offers actions b and c.
$I_7 := a; exit$	is not a valid implementation of S, since $I_7 \,||\, T(S)$ may deadlock after the sequence a which does not lead to a deadlock of $T(S)$: $T(S)$ offers actions b and c.

This definition does not require the introduction of a special action, say ω, to report the success of a test, like in [dNH 84]. The introduction of this special action may however always be done, but is considered here as an implementation matter of a practical tester, e.g. it suffices to add this special action ω in $T(S)$ before it reaches stop. Note that δ cannot be used as action ω since the parallel composition always enforces a synchronization on δ; which is to constraining.

Proposition 6.2 [Bri 87, Bri 88, Led 90]
Let S be a specification, $T(S)$ is defined by
(i) $Tr(T(S)) = Tr(S)$
(ii) $\forall\ \sigma \in Tr(S),\ \forall A \subseteq L$, *we have*

$$A \in Ref(T(S), \sigma) \quad iff \quad (L \in Ref(S, \sigma) \ \Leftrightarrow\ L - A \in Ref(S, \sigma)\ )$$ [2]

This proposition gives a method of construction of $T(S)$ from S. Informally, $T(S)$ has the same traces as S; moreover if, after a sequence σ, S may refuse any interaction from the set $A \subset L$, then

---

[1] This characterization of a process by its traces and refusals has been first proposed in CSP [BHR 84, Hoa 85], and then in LOTOS [Bri 87] where this model is denoted "Failure Tree". A similar but extended model is also developed in [Led 90]. We must note that the traces and the refusals satisfy some general properties presented in the aforementioned works (e.g. the trace sets are prefix-closed, the refusal sets are subset-closed, …). These properties must hold for any process in these models, otherwise we do not characterize real processes. These verifications are made implicitly throughout this paper.

[2] Part (ii) of this proposition may be simplified as follows by taking account of the subset closedness property of refusal sets, i.e. $L \in Ref(S, \sigma) \Rightarrow L - A \in Ref(S, \sigma)$:

$\forall\ \sigma \in Tr(S),\ \forall A \subseteq L$, *we have*

$$A \in Ref(T(S), \sigma) \quad iff \quad (L - A \in Ref(S, \sigma) \ \Rightarrow L \in Ref(S, \sigma))$$

T (S) must accept at least one interaction of the set $L - A$ after σ. In the particular case where *S* may deadlock after σ (i.e. refuses *L*, or in other words may reach a final state), *T (S)* must also have a reachable final state after σ.

Propositions 6.3 [Bri 87]

(i) $\forall \sigma \in L^*$, *we have* $L \in Ref(S, \sigma) \Leftrightarrow L \in Ref(T(S), \sigma)$

(ii) $\forall$ *S, T (S)* exists and is unique up to $\underline{te}$.

(iii) $\forall$ *S, we have* $T(T(S)) \ \underline{te}\ S$

Proposition 6.3 (iii) is the best result we may have since *T (S)* has been defined modulo $\underline{te}$.
Before extending the results of [Bri 87, Bri 88], we need some additional propositions.

Propositions 6.4

(i) $\forall P, Q$, *we have* $P \ \underline{conf}\ Q \wedge Q \ \underline{conf}\ P \Leftrightarrow T(P) \ \underline{conf}\ T(Q) \wedge T(Q) \ \underline{conf}\ T(P)$

(ii) $\forall P, Q$, *we have* $P \ \underline{conf\text{-}eq}\ Q \Leftrightarrow T(P) \ \underline{conf\text{-}eq}\ T(Q)$

(iii) $\forall P, Q$, *we have* $P \ \underline{te}\ Q \Leftrightarrow T(P) \ \underline{te}\ T(Q)$

Proofs

(i) We first prove $\Rightarrow$.

$P \ \underline{conf}\ Q \wedge Q \ \underline{conf}\ P$ *iff* $\forall \sigma \in Tr(P) \cap Tr(Q)$, *we have* $Ref(P, \sigma) = Ref(Q, \sigma)$.
We must prove that

$\forall \sigma \in Tr(P) \cap Tr(Q)$, *we have* $Ref(T(P), \sigma) = Ref(T(Q), \sigma)$.

But by hypothesis, $\forall \sigma \in Tr(P) \cap Tr(Q)$, we may restrict the proof to the following two cases:

(a) $L \in Ref(P, \sigma) \wedge L \in Ref(Q, \sigma)$

(b) $L \notin Ref(P, \sigma) \wedge L \notin Ref(Q, \sigma)$

Case (a) is immediate:

by proposition 6.3 (i), we get $L \in Ref(T(P), \sigma) \wedge L \in Ref(T(Q), \sigma)$,

whence $Ref(T(P), \sigma) = Ref(T(Q), \sigma) = \wp(L)$.

Case (b): we get successively

$\forall A \subseteq L$, *we have* $A \in Ref(P, \sigma) \Leftrightarrow A \in Ref(Q, \sigma)$, by hypothesis

$\forall A \subseteq L$, *we have* $L - A \notin Ref(T(P), \sigma) \Leftrightarrow L - A \notin Ref(T(Q), \sigma)$, by definition of *T*

$\forall A \subseteq L$, *we have* $A \notin Ref(T(P), \sigma) \Leftrightarrow A \notin Ref(T(Q), \sigma)$, directly

$\forall A \subseteq L$, *we have* $A \in Ref(T(P), \sigma) \Leftrightarrow A \in Ref(T(Q), \sigma)$, directly

We now prove $\Leftarrow$.

By applying $\Rightarrow$ above with *P* (resp. *Q*) replaced by *T (P)* (resp. *T (Q)*), we get immediately:

$T(P) \ \underline{conf}\ T(Q) \wedge T(Q) \ \underline{conf}\ T(P) \Rightarrow T(T(P)) \ \underline{conf}\ T(T(Q)) \wedge T(T(Q)) \ \underline{conf}\ T(T(P))$

The result then follows from proposition 6.3 (iii) and $\underline{conf} \circ \underline{te} = \underline{te} \circ \underline{conf} = \underline{conf}$.

(ii) We first prove $\Rightarrow$.

We will decompose the proof into three parts:

(a) $P \ \underline{conf\text{-}eq}\ Q \Rightarrow P \ \underline{conf}\ Q \wedge Q \ \underline{conf}\ P$

$P \ \underline{conf}\ Q \wedge Q \ \underline{conf}\ P \Rightarrow T(P) \ \underline{conf}\ T(Q) \wedge T(Q) \ \underline{conf}\ T(P)$, by proposition 6.4 (i).

(b) On the other hand, $P \ \underline{conf\text{-}eq}\ Q \Rightarrow \forall \sigma \in Tr(P) - Tr(Q)$, *we have* $L \in Ref(P, \sigma)$, by proposition 4.4.

Consequently, $\forall \sigma \in Tr(T(P)) - Tr(T(Q))$, *we have* $L \in Ref(T(P), \sigma)$, by definition of *T* and proposition 6.3 (i).

(c) Similarly, $P \ \underline{conf\text{-}eq}\ Q \Rightarrow \forall \sigma \in Tr(T(Q)) - Tr(T(P))$, *we have* $L \in Ref(T(Q), \sigma)$.

These three results prove the first part of the proposition, thanks to proposition 4.4.

We now prove $\Leftarrow$.

By applying $\Rightarrow$ above with P (resp. Q) replaced by $T(P)$ (resp. $T(Q)$), we get immediately:

$T(P)$ conf-eq $T(Q) \Rightarrow T(T(P))$ conf-eq $T(T(Q))$.

The result then follows from propositions 6.3 (iii) and 4.3 (ii).

(iii) This result is immediate by definition of T modulo te, and by proposition 6.3 (iii).

❐

Proposition 6.4 (ii) generalizes 6.4 (iii), and is a key proposition of this paper. Indeed, we know that if two specifications S_1 and S_2 are conf-equivalent (i.e. S_1 conf-eq S_2), their canonical testers must be interchangeable, since, by definition of conf-eq, they must give the same verdicts of conformance for the same implementations. From proposition 6.4 (ii), we get that this interchangeability is characterized by conf-eq, and not te. Therefore, we must define $T(S)$ modulo conf-eq, instead of modulo te. We will see that this will allow us to find a canonical tester $T_m(S)$ simpler than $T(S)$, and unique up to conf-eq.

The next proposition is the key of this paper. It proves that all the solutions X of equation 6.1 (ii) are such that X conf-eq $T(S)$, and reversely.

Proposition 6.5

Let S be a specification, and $T(S)$ its canonical tester.

$\forall X, (X$ conf-eq $T(S)$

$\Leftrightarrow (\forall P,\ P$ conf $S \Leftrightarrow (\forall \sigma \in L^*$, we have $(L \in Ref(P \,||\, X, \sigma) \Rightarrow L \in Ref(X, \sigma)))))$

Proof

The following propositions are equivalent:

X conf-eq $T(S)$,

$T(X)$ conf-eq $T(T(S))$, by proposition 6.4 (ii),

$T(X)$ conf-eq S, by propositions 6.3 (iii) and 4.3 (ii),

$\forall P,\ P$ conf $S \Leftrightarrow P$ conf $T(X)$, by definition 4.2 of conf-eq,

It remains to transform the second member. Again the following propositions are equivalent:

P conf $T(X)$,

$\forall \sigma \in L^*$, we have $(L \in Ref(P \,||\, T(T(X)), \sigma) \Rightarrow L \in Ref(T(T(X)), \sigma))$,
by definition 6.1 (ii),

$\forall \sigma \in L^*$, we have $(L \in Ref(P \,||\, X, \sigma) \Rightarrow L \in Ref(X, \sigma))$,
by proposition 6.3 (iii) and the congruence of te in the || - context (see [BrS 86]).

❐

Let us come back to definition 6.1 of the canonical tester, we see that 6.1 (ii) defines $T(S)$ modulo conf-eq, and that 6.1 (i) adds a constraint which fixes $T(S)$ modulo te.

Indeed, conf-eq $\cap$ trace-eq = te (see proposition 4.3 (iii)).

This constraint 6.1 (i) on the traces is totally arbitrary and cannot be justified neither practically, nor theoretically. Only the second constraint 6.1 (ii) is justifiable for the testing of implementations. It allows on its own a definition of a tester which, in all cases, will make the distinction between valid and invalid implementations. Moreover, we will see that the withdrawal of constraint 6.1 (i) allows us to find a canonical tester which has in general fewer traces than $T(S)$.

Proposition 6.6

Let S be a specification and $T(S)$ its canonical tester,

X conf-eq $T(S) \Leftrightarrow$

(i) $\forall \sigma \in Tr(X) \cap Tr(S)$, we have $Ref(X, \sigma) = Ref(T(S), \sigma)$, and

(ii) $\forall \sigma \in Tr(X) - Tr(S)$, we have $L \in Ref(X, \sigma)$, and

(iii) $\forall \sigma \in Tr(S) - Tr(X)$, we have $L \in Ref(S, \sigma)$.

This proposition is directly derived from definitions 4.4, 6.1 (i) and proposition 6.3 (i).

Obviously, any set of processes may be ordered w.r.t. a partial order relation "has its traces included in the traces of". In particular, the set of solutions X of equation 6.1 (ii) may be so ordered. It will be proved that this set has a minimum element X, i.e. an X which has fewer traces than any other in the set. We will formalize that in detail.

Definition 6.7

Min (S) is the specification obtained from *S* by deleting from *S* an adequate set of traces, while preserving the refusal sets for the remaining traces, i.e.

(i) $Tr\ (Min\ (S)) = Tr\ (S) - \{\sigma \mid (\forall\ \sigma' \geq \sigma$, we have $\sigma' \in Tr\ (S) \Rightarrow L \in Ref\ (S, \sigma'))$

$\wedge$

$\exists\ a, \sigma''$ such that $(\sigma = \sigma''.a \quad \wedge$

$\forall\ X \in Ref\ (S,\sigma''), X \cup \{a\} \in Ref\ (S,\sigma''))\}$

(ii) $\forall\ \sigma \in Tr\ (Min\ (S)), Ref\ (Min\ (S), \sigma) = Ref\ (S, \sigma)$

It can be shown that *Min (S)* is a well-defined process (modulo te).

Propositions 6.8

(i) *Min (S)* conf-eq *S*

(ii) *Min (S)* is the solution *X* of the equation "*X* conf-eq *S* " which has the smallest set of traces, i.e. $\forall\ X,\ X$ conf-eq $S \Rightarrow Tr\ (Min\ (S)) \subseteq Tr\ (X)$

Proofs

(i) Obviously, $Tr\ (Min\ (S)) \subseteq Tr\ (S)$

By definition, we also have that

$\forall\ \sigma \in Tr\ (Min(S)), Ref\ (Min\ (S), \sigma) = Ref\ (S, \sigma)$

It remains to prove that

$\forall\ \sigma \in Tr\ (S) - Tr\ (Min\ (S))$, we have $L \in Ref\ (S, \sigma)$

This is also obvious from the first line of the definition of $Tr\ (Min\ (S))$.

(ii) By contradiction, let *P* be a solution *X* such that

(a) *P* conf-eq *S*, and

(b) $\exists\ u \in Tr\ (Min\ (S))\ -\ Tr\ (P)$

(a) may be rewritten directly as *P* conf-eq *Min (S)* by proposition 6.8 (i)

Therefore, from that and the definition of *u* and conf-eq, we get $L \in Ref\ (Min\ (S), u)$.

Similarly, $\forall\ \sigma' \geq u$, we have $\sigma' \in Tr\ (Min\ (S)) \Rightarrow L \in Ref\ (Min\ (S), \sigma')$,

since obviously $\sigma' \notin Tr\ (P)$

Which is equivalent to $\forall\ \sigma' \geq u$, we have $\sigma' \in Tr\ (S) \Rightarrow L \in Ref\ (S, \sigma')$ (*)

$\exists$ a, σ", such that $u = \sigma''.a$, since *u* cannot be ε.

a) Suppose $\sigma'' \in Tr\ (P)$,

then $Ref\ (P, \sigma'') = Ref\ (Min\ (S), \sigma'')$ since *P* conf-eq *Min(S)*

Moreover, $\forall\ X \in Ref\ (P, \sigma''), X \cup \{a\} \in Ref\ (P, \sigma'')$, since $\sigma''.a \notin Tr\ (P)$

Finally, $\forall\ X \in Ref\ (Min\ (S), \sigma''), X \cup \{a\} \in Ref\ (Min\ (S), \sigma'')$,

from the previous line, since *Min (S)* conf-eq *P*

Which is equivalent to $\forall\ X \in Ref\ (S, \sigma''), X \cup \{a\} \in Ref\ (S, \sigma'')$, (**)

b) Suppose $\sigma'' \notin Tr\ (P)$,

then $L \in Ref\ (Min\ (S), \sigma'')$ like for *u*

And then obviously, $\forall\ X \in Ref\ (Min\ (S), \sigma''), X \cup \{a\} \in Ref\ (Min\ (S), \sigma'')$

Which is equivalent to $\forall\ X \in Ref\ (S, \sigma''), X \cup \{a\} \in Ref\ (S, \sigma'')$, (***)

(*) is the first condition for *u* not being a trace of *Min (S)*, see def. 6.7.

(**) and (***) are identical, and are the second condition for *u* not being a trace of *Min (S)*

Therefore, *u* cannot be a trace of *Min (S)* which is in contradiction with our hypothesis.

Definition 6.9

The minimum canonical tester of *S*, denoted $T_m\ (S)$, is defined by *Min (T (S))*.

Propositions 6.10
(i) $T_m\ (S)$ conf-eq $T\ (S)$
(ii) $T_m\ (S)$ te $T\ (Min\ (S))$

Proofs
(i) By definition of *Min*.
(ii) Directly from $Min\ (T\ (S))$ te $T\ (Min\ (S))$, which can be proved directly from definition 6.1 (i) and proposition 6.3 (i).

❒

$T_m\ (S)$ is thus derived from S in two steps:
1) Build $S' := Min\ (S)$,
2) Build $T_m\ (S) := T\ (S')$.
$T_m\ (S)$ has the advantage of being simpler than $T\ (S)$, as shown on the following example.

Examples
Let P and Q be the following processes:
$P = a;\ stop\ []\ b;\ stop$ and $Q = (a;\ stop\ []\ b;\ stop\ []\ a;\ b;\ stop)$.
We may prove that P conf-eq Q, but $\neg\ (P$ te $Q)$.
The canonical testers are defined (modulo te) as follows:
$T\ (P) := i;\ a;\ stop\ []\ i;\ b;\ stop$
$T\ (Q) := i;\ a;\ (b;\ stop\ []\ i;\ stop)\ []\ i;\ b;\ stop$
$T_m\ (P) = T\ (P)$
$T_m\ (Q) = T\ (P)$ is simpler than $T\ (Q)$.

Another slightly different example where T_m and T are testing equivalent
Suppose that Q is defined as follows:
$Q = (a;\ stop\ []\ b;\ stop\ []\ a;\ b;\ c;\ stop)$.
The canonical testers are defined (modulo te) as follows:
$T\ (Q) := i;\ a;\ (b;\ c;\ stop\ []\ i;\ stop)\ []\ i;\ b;\ stop$
$T_m\ (Q) := i;\ a;\ (b;\ c;\ stop\ []\ i;\ stop)\ []\ i;\ b;\ stop$
Consider the three following implementations:

I_1 := b; stop [] a; stop	is a valid implementation of Q, since $I_1 \parallel T\ (Q)$ may only deadlock after the sequences a or b which also lead to deadlocks of $T\ (Q)$
I_2 := b; stop [] a; b; stop	is not a valid implementation of Q, since if $T\ (Q)$ chooses to execute b after a, $I_2 \parallel T\ (Q)$ deadlocks whereas $T\ (Q)$ offers action c
I_3 := b; stop [] a; b; c; stop	is a valid implementation of Q, since $I_3 \parallel T\ (Q)$ may only deadlock after the sequences a, abc or b which also lead to deadlocks of $T\ (Q)$

The reason why $T\ (Q)$ cannot be simplified in this case may be informally understood on the three implementations above: if the branch $i;\ stop$ was deleted in $T\ (Q)$, then I_1 would be considered as an invalid implementation; if the branch $b;\ c;\ stop$ was deleted in $T\ (Q)$, then I_2 would be considered as a valid implementation.

Additional example (proposed by Ed Brinksma)
Let P be the following process: $P := a;\ stop\ []\ i;\ b;\ stop$
Its canonical tester is defined (modulo te) as follows: $T\ (P) := a;\ stop\ []\ i;\ b;\ stop$
The *minimum* canonical tester is defined (modulo te) by: $T_m\ (P) = b;\ stop$
which is simpler than $T\ (P)$.

The next proposition is the main result of this paper. It proves that if S is a specification, $T_m\ (S)$ is the minimum process (in terms of traces) which can test (exhaustively) any implementation and give a correct verdict of conformance.

Proposition 6.11
Let S be a specification.
$T_m\ (S)$ is the smallest solution X (in terms of traces) satisfying the equation:

$\forall\ P,\ P$ conf $S\ \Leftrightarrow\ \forall\ \sigma \in L^*$, *we have* $(L \in Ref\ (P \parallel X,\ \sigma) \Rightarrow L \in Ref\ (X,\ \sigma))$.

Proof
$T_m\ (S)$ is a solution of the equation by propositions 6.10 (i) and 6.5.

T_m *(S)* is the smallest solution by definition 6.9 and proposition 6.8. ❒

Note that even if we restrict ourselves to successfully terminating specifications *S* (like those in [BrS 86, Bri 88]), T_m *(S)* may have fewer traces than *T (S)*; as shown on the counter-example following the formal definition of a successfully terminating process.

Definitions 6.12
A specification *S* is *successfully terminating* if, and only if,

$\forall \sigma \in Tr(S)$, *we have* $(L \in Ref(S, \sigma) \Leftrightarrow \exists \sigma', \sigma = \sigma'.\delta)$ where δ is the special action which indicates a successful termination in LOTOS.

Counter-example
Let *P* be the following successfully terminating process:
P := exit [] i; a; exit
Then *T (P) and* T_m *(P)* are defined as follows (modulo *te*):
T (P) := exit [] i; a; exit
T_m *(P) := a; exit*

7. Conclusion

We have outlined the role of an implementation relation to formalize the link between a valid implementation and a specification. The definition of an implementation (or conformance) relation is an essential prerequisite to the definition of any kind of conformance tester (or set of conformance test scenarios).

We have shown that, from an implementation relation, it is always possible to deduce an associated equivalence, even if the implementation relation is not transitive.

Then we have selected the *conf* relation as implementation relation, and shown that the associated equivalence relation, *conf-eq*, is weaker than *te*.

These results allowed us to show that it is possible to define a minimum canonical tester which is simpler than the *T (S)* defined in [Bri 87, Bri 88]: unlike *T (S)*, it may have fewer traces than the specification.

Finally, let us outline that the methodology used in this paper, which consists in defining the canonical tester modulo an equivalence relation associated with an adequate implementation relation, is very important and new. It may be used in the same way for other implementation relation candidates. Rather than deriving from scratch a new canonical tester based solely on definition 6.1 and proposition 6.5, we have chosen to build on Brinksma's testing theory and show how it may be slightly adapted. This has led to the definition of T_m as a kind of simplification of *T* (see definition 6.9 together with proposition 6.11).

Acknowledgements

I would like to thank Ed Brinksma for having pointed out an error (see end of section 1) in a previous version of this paper [Led 91c].

8. References

[AbL 88] M. Abadi, L. Lamport, *The Existence of Refinement Mappings*, in: Third Annual Symposium on Logic in Computer Science, IEEE Computer Society, Edinburgh, Scotland, July 88, 165-175. To appear in: Theoretical Computer Science.

[Ald 89] R. Alderden, *COOPER, the compositional construction of a canonical tester,* in: S. T. Vuong, ed., FORTE '89 (North-Holland, Amsterdam, 1990).

[BeK 85] J.A. Bergstra, J. W. Klop, *Algebra of Communicating Processes with Abstraction,* Theoretical Computer Science 37 (1985) 77-121 (North-Holland, Amsterdam).

[BHR 84] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, *A theory of Communicating Sequential Processes,* Journ. ACM, Vol. 31, No. 3, July 1984, 560-599.

[BoB 87] T. Bolognesi, E. Brinksma, *Introduction to the ISO Specification Language LOTOS,* Computer Networks and ISDN Systems 14 (1) 25-59 (1987).

[Bri 87] E. Brinksma, *On the existence of canonical testers,* Rept. No. INF-87-5, Twente University of Technology, Department of Informatics, Enschede, The Netherlands, January 1987.

[Bri 88] E. Brinksma, *A Theory for the Derivation of Tests,* in: S. Aggarwal, K. Sabnani, eds., Protocol Specification, Testing and Verification, VIII (North-Holland, Amsterdam, 1988, ISBN 0-444-70542-2) 63-74.

[BrS 86] E. Brinksma, G. Scollo, *Formal notions of implementation and conformance in LOTOS,* Rept. No. INF-86-13, Twente University of Technology, Department of Informatics, Enschede, The Netherlands, Dec. 1986.

[BSS 87] E. Brinksma, G. Scollo, C. Steenbergen, *Process specification, their implementations and their tests,* in: G.v. Bochmann, B. Sarikaya, eds., Protocol Specification, Testing and Verification, VI (North-Holland, Amsterdam, 1987, ISBN 0-444-70126-5) 349-360.

[ChM 89] K. M. Chandy, J. Misra, *Parallel Program Design - A Foundation* (Addison-Wesley, 1989, ISBN 0-201-05866-9).

[Dij 86] E. W. Dijkstra, *A discipline of programming* (Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1976, ISBN 0-13-215871-X).

[dNH 84] R. De Nicola, M.C.B. Hennessy, *Testing equivalences for processes,* Theoretical Computer Science 34 (1984) 83-133 (North-Holland, Amsterdam).

[dNi 87] R. De Nicola, *Extensional Equivalences for Transition Systems,* Acta Informatica 24 (1987) 211-237 (Springer - Verlag, Berlin Heidelberg).

[FaL 87] J.P. Favreau, R. Linn, *Automatic generation of tests scenario skeletons from protocol specifications written in Estelle,* in: G.v. Bochmann, B. Sarikaya, eds., Protocol Specification, Testing and Verification, VI (North-Holland, Amsterdam, 1987, ISBN 0-444-70126-5) 191-202.

[Hen 88] M. Hennessy, *Algebraic Theory of Processes* (MIT Press, Cambridge, London, 1988, ISBN 0-262-08171-7).

[Hoa 85] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall International, London, 1985, ISBN 0-13-153271-5).

[Hog 89] D. Hogrefe, *Automatic generation of test cases from SDL specifications,* SDL newsletter, No. 12, June 1988.

[ISO 8807] ISO/IEC-JTC1/SC21/WG1/FDT/C, *Information Processing Systems - Open Systems Interconnection - LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, IS 8807, February 1989.

[ISO 9646] ISO/IEC-JTC1/SC21, *Conformance Testing Methodology and Framework*, DIS 9646, April 1989.

[Lam 83] L. Lamport, *Specifying concurrent program modules,* ACM Transactions on Programming Languages and Systems 5 (2) 190-222 (1983).

[Lar 89] K. G. Larsen, *Modal Specifications,* in: J. Sifakis, ed., Automatic Verification Methods for Finite State Systems, LNCS 407 (Springer - Verlag, Berlin Heidelberg New York, 1990, ISBN 3-540-52148-8) 232-246.

[LaS 84] S.S. Lam, A. U. Shankar, *Protocol verification via projections,* IEEE Transactions on Software Engineering, Vol. 10, No. 4, April 1984, 325-342.

[Led 90] G. Leduc, *On the role of implementation relations in the design of distributed systems,* Agrégation dissertation, Université de Liège, Dept. Systèmes et Automatique, B28, Liège, Belgium, 1990.

[Led 91a] G. Leduc, *A Framework based on implementation relations for implementing LOTOS specifications,* To appear in a special issue of: Computer Networks & ISDN Systems, 1991.

[Led 91b] G. Leduc, *Relations d'implémentation et transformations autorisées d'une spécification LOTOS,* Réseaux et informatique répartie, Vol. 1 - n°1/1991 (Editions Hermès) 59-86.

[Led 91c] G. Leduc, *Conformance relation, associated equivalence, and new canonical tester in LOTOS,* in: B. Jonsson, J. Parrow, B. Pehrson, eds., Protocol Specification, Testing and Verification, XI (to be published by North-Holland, Amsterdam, 1991).

[LyF 81] N. Lynch, M. Fisher, *On describing the behavior and implementation of distributed systems,* Theoretical Computer Science 13 (1981) 17-43 (North-Holland, Amsterdam).

[LyT 87] N. Lynch, M. Tuttle, *Hierarchical correctness proofs for distributed algorithms,* in: Proc. of the 6th ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada , Aug. 87, 137 - 151.

[MaS 90] J.A. Mañas, J. Salvachúa, *The Sieve of Eratosthenes - An evaluation of Compilation Performance,* Rept. No. Lo/WP2/T2.2/UPM/N0015/V01, ESPRIT II/Lotosphere project, University of Madrid, Spain, Sept. 90.

[Mer 89] M. Merritt, *Completeness Theorems for Automata,* Rept. AT&T Bell Laboratories, Murray Hill, NJ, May 1989.

[Mil 89] R. Milner, *Communication and Concurrency* (Prentice-Hall International, London, 1989, ISBN 0-13-114984-9).

[Par 81] D. Park, *Concurrency and Automata on Infinite Sequences,* in: Theoretical Computer Science, LNCS 104 (Springer-Verlag, Berlin Heidelberg New York, 1981, ISBN 3-540-10576-X) 167-183.

[PhG 91] M. Phalippou, R. Groz, *From Estelle Specifications to Industrial Test Suites, using an Empirical Approach,* in: J. Quemada, J. Mañas, E. Vazquez, eds., FORTE '90 (to be published by North-Holland, Amsterdam, 1991).

[Ray 87] D. Rayner, *OSI Conformance Testing,* Computer Networks and ISDN Systems, 14, 1987, 79-98.

[SaD 88] K. Sabnani, A. Dahbura, *A Protocol Test Generation Procedure,* Computer Networks and ISDN Systems, 15, 1988, 285-297.

[Sar 87] B. Sarikaya, *Conformance Testing: Architectures and Test Sequences,* Computer Networks and ISDN Systems, 17, 1989, 111-126.

[SBC 87] B. Sarikaya, G. Bochmann, E. Cerny, *A Test Design Methodology for Protocol Testing,* IEEE Transactions on Software Engineering, Vol. 13, No. 5, 1987, 518-531.

[SiL 89] D. Sidhu, T. Leung, *Formal Methods for Protocol Testing: A Detailed Study,* IEEE Transactions on Software Engineering, Vol. 15, No. 4, April 1989, 413-426.

[VCI 89] S. Vuong, W. Chan, R. ITO, *The UIO-method for protocol test sequence,* 2nd International Workshop on Protocol Test Systems, Berlin, October 1989, 203-225.

[vGl 90] R.J. van Glabeek, *The Linear Time - Branching Time Spectrum,* in: J.C.M. Baeten, J.W. Klop, eds., CONCUR '90, Theories of Concurrency: Unification and Extension, LNCS 458 (Springer - Verlag, Berlin Heidelberg New York, 1990, ISBN 3-540-53048-7) 278-297.

[VSv 88] C.A. Vissers, G. Scollo, M. van Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems,* in: S. Aggarwal, K. Sabnani, eds., Protocol Specification, Testing and Verification, VIII (North-Holland, Amsterdam, 1988, ISBN 0-444-70542-2) 189-204.

[WBL 91] C.D. Wezeman, S. Batley, J.A. Lynch, *Formal Methods to Assist Conformance Testing - A case study,* in: J. Quemada, J. Mañas, E. Vazquez, eds., FORTE '90 (to be published by North-Holland, Amsterdam, 1991).

[Wez 89] C.D. Wezeman, *The CO-OP method for compositional derivation of conformance testers,* in: E. Brinksma, G. Scollo, C. A. Vissers, eds., Protocol Specification, Testing, and Verification, IX (North-Holland, Amsterdam, 1989).