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 S1 conf-eq S2, their canonical testers T1 and T2 must also satisfy T1 conf-eq T2, 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 relations 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 ⇒ 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 conformance 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 \( S \). 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 \( S \). 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 \( S \) and \( T(S) \) 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 \( \text{red} \), \( \text{ext} \), and the conformance relation \( S \).

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

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

From this observation, we will prove that we can define \( T(S) \) modulo \( \text{conf-eq} \), i.e. if \( T \) is the canonical tester of the specification S, and if \( T' \text{conf-eq} T \), 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 \( S \).

This characterization of the canonical tester modulo \( \text{conf-eq} \) 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.

\( P \xrightarrow{a} P' \) means that process \( P \) may engage in action \( a \) and, after doing so, behave like process \( P' \).

\( P \xrightarrow{i_0.a_i k_i} P' \) means that process \( P \) may engage in the sequence of \( k \) internal actions and, after doing so, behave like process \( P' \).

\( P \xrightarrow{a} P' \) means \( \exists P'' \), such that \( P \xrightarrow{a} P'' \land P'' \xrightarrow{b} P' \).

\( P = a \Rightarrow P' \) where \( a \in L \), means that \( P \xrightarrow{i_{k_0}.a_i k_i} P' \).

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

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

\[ \exists k_0, ... k_n \in N, \text{ such that } P \xrightarrow{i_{k_0}.a_1 k_1.a_2 ... .k_n} P' \]

\( P \) after \( \sigma = \{ P' | P = \sigma \Rightarrow P' \} \),

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

\( Tr(P) \) is the trace set of \( P \), i.e. \( \{ \sigma | \sigma \Rightarrow \} \); \( Tr(P) \) 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. \( \sigma_1 \) is a prefix of \( \sigma_2 \).

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

\( Out(P, \sigma) \) is the set of possible observable actions after the trace \( \sigma \), i.e. \( \{ a \in L | a \in Tr(P) \} \).

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

\[ Ref(P, \sigma) = \{ X \subseteq L | \exists P' \in P \text{ after } \sigma \text{ such that } P' = a \Rightarrow, \forall a \in X \}; \]

\( Ref(P, \sigma) \) is a set of sets and a subset of \( \emptyset \) (\( L \)), 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 \( \sigma \) 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,

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

because if \( \sigma \in Tr(P_2) - Tr(P_1) \), then \( Ref(P_1, \sigma) = \emptyset \).

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) \( P_1 conf P_2 \)

4
Intuitively, if $P_1 \text{ red } P_2$, $P_1$ 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. \textit{Red} is the \textit{reduction} relation.

\[ P_1 \text{ ext } P_2 \text{ iff (i) } Tr (P_1) \supseteq Tr (P_2), \text{ and (ii) } P_1 \text{ conf } P_2 \]

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

\[ te = \text{ red } \cap \text{ red}^{-1} = \text{ ext } \cap \text{ ext}^{-1} \]

This is the testing equivalence.

\textbf{Propositions 2.3 [BrS 86]}

(i) $\text{conf} \supset \text{red}$

(ii) $\text{conf} \supset \text{ext}$

(iii) $\text{red}$ and $\text{ext}$ are preorders

(iv) $\text{conf}$ is not transitive

\[ S_1 \text{ imp-eq } S_2 \text{ iff } \{I | I \text{ imp } S_1\} = \{I | I \text{ imp } S_2\} \]

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

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

It is obvious that \textit{imp-eq} is reflexive, symmetric and transitive. \textit{imp-eq} is therefore an equivalence relation whatever \textit{imp} is (even if \textit{imp} is not reflexive).

If \textit{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-
Conformance relation, associated equivalence, and minimum canonical tester in LOTOS

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

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

**Proposition 3.2**

\[ \text{imp-eq} \subseteq \text{imp} \cap \text{imp}^{-1} \]

The proof is immediate because \( S_1 \text{imp-eq} S_2 \Rightarrow (S_1 \text{imp} S_2 \land S_2 \text{imp} S_1) \). This is derived from the definition of \( \text{imp-eq} \) and the property of reflexivity of \( \text{imp} \).

**Propositions 3.3**

\( \text{imp} \) is transitive \( \Rightarrow \text{imp-eq} = \text{imp} \cap \text{imp}^{-1} \)

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

So, when \( \text{imp} \) is transitive, \( \text{imp-eq} \) can be defined advantageously as follows:

\[ S_1 \text{imp-eq} S_2 \iff S_1 \text{imp} S_2 \land S_2 \text{imp} S_1. \]

4. Equivalence relations associated with \( \text{red, ext} \) and \( \text{conf} \)

**Proposition 4.1** [BrS 86]

\[ \text{Red-eq} = \text{ext-eq} = \text{te} \]

**Definition 4.2**

\[ S_1 \text{conf-eq} S_2 \iff \{I \mid I \text{conf} S_1\} = \{I \mid I \text{conf} S_2\} \]

This definition is a simple instantiation of definition 3.1.

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

**Propositions 4.3**

(i) \( \text{conf-eq} \subseteq \text{conf} \cap \text{conf}^{-1} \)

(ii) \( \text{conf-eq} \supseteq \text{te} \)

i.e., \( \text{conf-eq} \) is weaker than the testing equivalence

(iii) \( \text{conf-eq} \cap \text{trace-eq} = \text{conf} \cap \text{conf}^{-1} \cap \text{trace-eq} = \text{te} \)

or equivalently,

\[ \forall P, Q, \text{we have} \]

\[ P \text{conf-eq} Q \land (\text{Tr} (P) = \text{Tr} (Q)) \iff P \text{conf} Q \land Q \text{conf} P \land (\text{Tr} (P) = \text{Tr} (Q)) \iff P \text{te} Q \]

For processes with equal trace sets, \( \text{conf-eq} \) and \( \text{te} \) 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 \( \text{conf-eq} \).
**Proposition 4.4** [Led 90]

\[ P \text{ conf-eq } Q \text{ iff } \]

(a) \( P \text{ conf } Q \land Q \text{ conf } P \), and (i.e. \( \forall \sigma \in \text{Tr}(P) \cap \text{Tr}(Q), \text{ Ref}(P, \sigma) = \text{Ref}(Q, \sigma) \))

(b) \( \forall \sigma \in \text{Tr}(P) - \text{Tr}(Q) \), we have \( L \in \text{Ref}(P, \sigma) \), and

(c) \( \forall \sigma \in \text{Tr}(Q) - \text{Tr}(P) \), we have \( L \in \text{Ref}(Q, \sigma) \)

5. **conf-eq versus te**

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

Consider the following example where \( P \text{ conf-eq } Q \), but \( \neg (P \text{ te } Q) \):

\[ P = a; \text{stop} \quad \text{and} \quad Q = (a; \text{stop} \quad [a; \text{b}; \text{stop}) \].

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

Note however that if \( P = a; \text{stop} \) and \( Q = (a; \text{stop} \quad [a; \text{b}; \text{c}; \text{stop}) \), we do not have \( P \text{ conf-eq } Q \) any more, because \( a; \text{b}; \text{stop} \) 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 \( Q \text{ te } T(S) \). 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) \( \text{Tr}(X) = \text{Tr}(S) \)

(ii) \( \forall P, \ P \text{ conf } S \iff (\forall \sigma \in L^*, \text{ we have } (L \in \text{Ref}(P \parallel X, \sigma) \Rightarrow L \in \text{Ref}(X, \sigma)) \)
This definition characterizes $T(S)$ by its traces and refusals\(^1\) after each trace; which defines $T(S)$ modulo $\text{te}$. (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 $P \parallel T(S)$ may engage in a sequence $\sigma$ and then deadlock after $\sigma$, 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 $S := a; b; \text{exit} [] a; c; \text{stop}$. We give without proof that $T(S) := a; (b; \text{exit} [] c; \text{stop}) \mod \text{te}$. The correct final states of $T(S)$ are reached after the sequences $ab\delta$ and $ac$.

Consider the following implementations for illustration:

- $I_1 := a; c; \text{stop}$ is a valid implementation of $S$, since $I_1 \parallel T(S)$ may only deadlock after the sequence $ac$ which leads to a deadlock of $T(S)$.
- $I_2 := a; b; \text{exit}$ is a valid implementation of $S$, since $I_2 \parallel T(S)$ may only deadlock after the sequence $ab\delta$ which also leads to a deadlock of $T(S)$.
- $I_3 := a; b; \text{stop}$ is not a valid implementation of $S$, since $I_3 \parallel T(S)$ may deadlock after the sequence $ab$ which does not lead to a deadlock of $T(S)$: $T(S)$ offers $\delta$.
- $I_4 := T(S)$ is a valid implementation of $S$, since $I_4 \parallel T(S)$ may only deadlock after the sequences $ab\delta$ or $ac$ which also leads to deadlocks of $T(S)$.
- $I_5 := S$ is a valid implementation of $S$, since $I_5 \parallel T(S)$ may only deadlock after the sequences $ab\delta$ or $ac$ which also lead to deadlocks of $T(S)$.
- $I_6 := a; \text{stop}$ is not a valid implementation of $S$, since $I_6 \parallel T(S)$ may deadlock after the sequence $a$ which does not lead to a deadlock of $T(S)$: $T(S)$ offers actions $b$ and $c$.
- $I_7 := a; \text{exit}$ is not a valid implementation of $S$, since $I_7 \parallel T(S)$ may deadlock after the sequence $a$ which leads 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 $\omega$, 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 to the tester before it reaches stop. Note that $\delta$ cannot be used as action in $T(S)$ since the parallel composition always enforces a synchronization on $\delta$, which is to constraining.

**Proposition 6.2 [Bri 87, Bri 88, Led 90]**

Let $S$ be a specification, $T(S)$ is defined by

1. $\forall \sigma \in \text{Tr}(S), \forall A \subseteq L$, we have
   \[ A \in \text{Ref}(T(S), \sigma) \iff (L \in \text{Ref}(S, \sigma) \implies L - A \in \text{Ref}(S, \sigma)) \]

   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 $\sigma$, $S$ may refuse any interaction from the set $A \subseteq L$, 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. $L \in \text{Ref}(S, \sigma) \implies L - A \in \text{Ref}(S, \sigma)$:

   $\forall \sigma \in \text{Tr}(S), \forall A \subseteq L$, we have
   \[ A \in \text{Ref}(T(S), \sigma) \iff (L - A \in \text{Ref}(S, \sigma) \implies L \in \text{Ref}(S, \sigma)) \]
$T (S)$ must accept at least one interaction of the set $L - A$ after $\sigma$. In the particular case where $S$ may deadlock after $\sigma$ (i.e. refuses $L$, or in other words may reach a final state), $T (S)$ must also have a reachable final state after $\sigma$.

**Propositions 6.3 [Bri 87]**

(i) $\forall \sigma \in L^*, \text{we have } L \in \text{Ref} (S, \sigma) \iff L \in \text{Ref} (T (S), \sigma)$

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

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

Proposition 6.3 (iii) is the best result we may have since $T (S)$ has been defined modulo $\text{te}$.

Before extending the results of [Bri 87, Bri 88], we need some additional propositions.

**Propositions 6.4**

(i) $\forall P, Q$, we have $P \text{ conf } Q \iff Q \text{ conf } P \implies T (P) \text{ conf } T (Q) \iff T (Q) \text{ conf } T (P)$

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

(iii) $\forall P, Q$, we have $P \text{ te } Q \iff T (P) \text{ te } T (Q)$

**Proofs**

(i) We first prove $\implies$.

$P \text{ conf } Q \wedge Q \text{ conf } P \iff \forall \sigma \in \text{Tr} (P) \cap \text{Tr} (Q)$, we have $\text{Ref} (P, \sigma) = \text{Ref} (Q, \sigma)$.

We must prove that

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

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

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

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

Case (a) is immediate:

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

whence $\text{Ref} (T (P), \sigma) = \text{Ref} (T (Q), \sigma) = \emptyset (L)$.

Case (b): we get successively

$\forall A \subseteq L$, we have $A \in \text{Ref} (P, \sigma) \iff A \in \text{Ref} (Q, \sigma)$, by hypothesis

$\forall A \subseteq L$, we have $L - A \notin \text{Ref} (T (P), \sigma) \iff L - A \notin \text{Ref} (T (Q), \sigma)$, by definition of $T$

$\forall A \subseteq L$, we have $A \notin \text{Ref} (T (P), \sigma) \iff A \notin \text{Ref} (T (Q), \sigma)$, directly

$\forall A \subseteq L$, we have $A \in \text{Ref} (T (P), \sigma) \iff A \in \text{Ref} (T (Q), \sigma)$, directly

We now prove $\iff$.

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

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

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

(ii) We first prove $\Rightarrow$.

We will decompose the proof into three parts:

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

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

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

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

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

These three results prove the first part of the proposition, thanks to proposition 4.4.
We now prove $\equiv$.

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

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

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 $\text{conf-equiv}$ (i.e. $S_1 \text{ conf-eq } S_2$), their canonical testers must be interchangeable, since, by definition of $\text{conf-eq}$, 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 $\text{conf-eq}$, and not $te$. Therefore, we must define $T(S)$ modulo $\text{conf-eq}$, 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 $\text{conf-eq}$.

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

**Proposition 6.5**

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

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

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

**Proof**

The following propositions are equivalent:

$X \text{ conf-eq } T(S)$,

$$T(X) \text{ conf-eq } T(T(S)),$$

by proposition 6.4 (ii),

and

$$T(X) \text{ conf-eq } S,$$

by propositions 6.3 (iii) and 4.3 (ii),

$$\forall P, \text{ P conf } S \Leftrightarrow \text{ P conf } T(X),$$

by definition 4.2 of $\text{conf-eq}$.

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

$$P \text{ conf } T(X),$$

$$\forall \sigma \in L^*, \text{ we have } (L \in \text{Ref (P || T (T(X)), } \sigma) \Rightarrow L \in \text{Ref (T (T(X)), } \sigma)),$$

by definition 6.1 (ii),

and

$$\forall \sigma \in L^*, \text{ we have } (L \in \text{Ref (P || X, } \sigma) \Rightarrow L \in \text{Ref (X, } \sigma)),$$

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 $\text{conf-eq}$, and that 6.1 (i) adds a constraint which fixes $T(S)$ modulo $te$.

Indeed, $\text{conf-eq} \cap \text{trace-eq} = te$ (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,

$$X \text{ conf-eq } T(S) \Leftrightarrow$$

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

(ii) $\forall \sigma \in Tr(X) \setminus Tr(S)$, we have $L \in \text{Ref (X, } \sigma)$, and

(iii) $\forall \sigma \in Tr(S) \setminus Tr(X)$, we have $L \in \text{Ref (S, } \sigma)$.

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) – {σ | (∀ σ’ ≥ σ, we have σ’ ∈ Tr (S) ⇒ L ∈ Ref (S, σ’)) ∧

∃ a, σ’ such that (σ = σ’ .a ∧

∀ X ∈ Ref (S, σ’), X ∪ {a} ∈ Ref (S, σ’))}

(ii) ∀ σ ∈ Tr (Min (S)), Ref (Min (S), σ) = Ref (S, σ)

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. ∀ X, X conf-eq S ⇒ Tr (Min (S)) ⊆ Tr (X)

Proofs

(i) Obviously, Tr (Min (S)) ⊆ Tr (S)

By definition, we also have that

∀ σ ∈ Tr (Min(S)), Ref (Min (S), σ) = Ref (S, σ)

It remains to prove that

∀ σ ∈ Tr (S) – Tr (Min (S)), we have L ∈ Ref (S, σ)

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 Min (S), and

(b) ∃ u ∈ 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 ∈ Ref (Min (S), u).

Similarly, ∀ σ’ ≥ u, we have σ’ ∈ Tr (Min (S)) ⇒ L ∈ Ref (Min (S), σ’),

since obviously σ’ ∉ Tr (P)

Which is equivalent to ∀ σ’ ≥ u, we have σ’ ∈ Tr (S) ⇒ L ∈ Ref (S, σ’) (*)&

∃ a, σ”, such that u = σ”.a, since u cannot be ε.

a) Suppose σ” ∈ Tr (P),

then Ref (P, σ”) = Ref (Min (S), σ”) since P conf-eq Min(S)

Moreover, ∀ X ∈ Ref (P, σ”), X ∪ {a} ∈ Ref (P, σ”), since σ” .a ∉ Tr (P)

Finally, ∀ X ∈ Ref (Min (S), σ”), X ∪ {a} ∈ Ref (Min (S), σ”),

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

Which is equivalent to ∀ X ∈ Ref (S, σ”), X ∪ {a} ∈ Ref (S, σ”), (**) (***)

b) Suppose σ” ∉ Tr (P),

then L ∈ Ref (Min (S), σ”) like for u

And then obviously, ∀ X ∈ Ref (Min (S), σ”), X ∪ {a} ∈ Ref (Min (S), σ”)

Which is equivalent to ∀ X ∈ Ref (S, σ”), X ∪ {a} ∈ Ref (S, σ”), (**) 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 Tm (S), is defined by Min (T (S)).
Propositions 6.10

(i) \( T_m(S) \) is equivalent to \( T(S) \)

(ii) \( T_m(S) \) is the minimum tester of \( S \)

Proofs

(i) By definition of \( T_m \).

(ii) Directly from \( \exists T (\text{Min}(S)) \), 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 \( S' := \text{Min}(S) \),
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:
\[ P = a; \text{stop} [ ] b; \text{stop} \quad \text{and} \quad Q = (a; \text{stop} [ ] b; \text{stop} [ ] a; b; \text{stop}) . \]

We may prove that \( P \) is equivalent to \( Q \), but not \( T_m(P) \) is equivalent to \( T(Q) \).

The canonical testers are defined (modulo \( T_m \)) as follows:
\[ T(P) := i; a; \text{stop} [ ] i; b; \text{stop} \]
\[ T(Q) := i; a; (b; \text{stop} [ ] i; \text{stop}) [ ] i; b; \text{stop} \]
\[ T_m(P) = T(P) \]
\[ T_m(Q) = T(P) \]

Another slightly different example where \( T_m \) and \( T \) are testing equivalent

Suppose that \( Q \) is defined as follows:
\[ Q = (a; \text{stop} [ ] b; \text{stop} [ ] a; b; c; \text{stop}). \]

The canonical testers are defined (modulo \( T_m \)) as follows:
\[ T(Q) := i; a; (b; c; \text{stop} [ ] i; \text{stop}) [ ] i; b; \text{stop} \]
\[ T_m(Q) := i; a; (b; c; \text{stop} [ ] i; \text{stop}) [ ] i; b; \text{stop} \]

Consider the three following implementations:

\( I_1 := b; \text{stop} [ ] a; \text{stop} \) is a valid implementation of \( Q \), since \( I_1 \) may only deadlock after the sequences \( a \) or \( b \) which also lead to deadlocks of \( T(Q) \)

\( I_2 := b; \text{stop} [ ] a; b; \text{stop} \) is not a valid implementation of \( Q \), since if \( T(Q) \) chooses to execute \( b \) after \( a \), \( I_2 \) may only deadlock whereas \( T(Q) \) offers action \( c \)

\( I_3 := b; \text{stop} [ ] a; b; c; \text{stop} \) is a valid implementation of \( Q \), since \( I_3 \) 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 \( i; \text{stop} \) was deleted in \( T(Q) \), then \( I_1 \) would be considered as an invalid implementation; if the branch \( b; c; \text{stop} \) 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:
\[ P := a; \text{stop} [ ] i; \text{stop} \]

Its canonical tester is defined (modulo \( T_m \)) as follows:
\[ T(P) := a; \text{stop} [ ] i; b; \text{stop} \]

The minimum canonical tester is defined (modulo \( T_m \)) by:
\[ T_m(P) = b; \text{stop} \]

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:
\[ \forall P, P \text{ conf } S \iff \forall \sigma \in L^*, \text{ we have } (L \in \text{Ref}(P || X, \sigma) \Rightarrow L \in \text{Ref}(X, \sigma)) . \]

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,

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

**Counter-example**
Let $P$ be the following successfully terminating process:

$$P := \text{exit} \{ i; a; \text{exit}\}$$

Then $T(P)$ and $T_m(P)$ are defined as follows (modulo $te$):

$$T(P) := \text{exit} \{ i; a; \text{exit}\}$$

$$T_m(P) := a; \text{exit}$$

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


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


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


