Published in: Computer Networks and ISDN Systems, 25 (1992), 23-41

A framework based on implementation relations for implementing LOTOS specifications

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
Tel.: + 32 41 562698; fax: + 32 41 562989; e-mail: u514401@bliulg11.bitnet

Abstract

A framework is developed for studying the implementation process, as a stepwise process in which an abstract specification is successively transformed to reach a final compilable specification adapted to the computer environment. In this context, an *implementation relation* is referred to as the relation which should link any "valid" implementation to its abstract formal specification. In other words, the implementation relation is intended to express formally the notion of validity. Our framework allows the exact characterization of the transformations which may take place at each step for a given implementation relation. This framework is essential for dealing with non-transitive implementation relations. In the second part of the paper, these results are exemplified in LOTOS on some existing relations, and an apparent paradox is presented. Some new results about these relations are also derived.

Keywords
LOTOS, implementation relation, refinement, implementation process, transformation, conformance, specification, implementation, abstraction, FDT, process algebra

1. Introduction

In this paper a general mathematical framework is developed for reasoning on the implementation process. The implementation process is the part of the design process dealing with the implementation or realization of a formal specification, in which an abstract specification is successively transformed into a more implementation-oriented one. This process is composed of synthesis and analysis activities, and involves very likely many intermediate stages before reaching a final (compilable) specification [3, 14, 5]. Moreover, the successive transformations which may take place at each stage are numerous; which implies that many final stages may be derived from the initial specification. However, the intent is to reach a final stage which is considered as a "valid" implementation of the initial abstract specification.

One of the main problems in this context is the formalization of this notion of validity, i.e. the nature of the link which should hold between the allowed final stages and the initial stage of the implementation process.

Validity as an equivalence

The usual view with algebraic techniques is to require that an *equivalence relation* is preserved throughout the whole implementation process. Equivalence relations play a central role in process algebraic languages such as CCS [25], ACP [2] or LOTOS [17, 6] 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 [13]. 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 [13, 29, 21]. Examples of such equivalences are the observation equivalence [26, 25] or the testing equivalences [4, 12, 8, 15].

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 have been identified for different purposes [30]. The implementation process may then be considered as successive transformations of (parts of) the specification from one style to another one [28].

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 (abstract) 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 TCSP, such a relation had already been introduced [4, 16] as a preorder of the failures equivalence. In CCS, other preorders of various testing equivalences were introduced [12]. And finally, in LOTOS, preorders of the testing equivalence, as well as a conformance relation, have been defined [8]. In [21], we have surveyed the main equivalences and their associated preorders, and tried to relate many apparently different views, based on traces, refusals or acceptance sets, and divergences.

The concept of an implementation relation has also been introduced in other formal models. For instance, with logic-based specifications [11], B is usually considered as an implementation of A iff $B \Rightarrow A$, i.e. B satisfies more properties than A. 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 [22, 18, 19, 23, 1, 24]. With modal transition systems (i.e. an extension of a LTS with necessary and admissible transitions) [20], B is considered as an implementation of A iff there exists a refinement relation between B and A.

Content of the paper

In our framework, we consider a generic implementation relation, called a *reference implementation relation*, which is supposed to express formally the notion of validity explained above.

The paper then focuses on the following problem:

> Given this reference implementation relation, what are the induced restrictions on the allowed transformations between two intermediate specification stages in the implementation process? In other words, how can we characterize the *implementer's freedom* at each step as a consequence of the choice of the reference implementation relation?

The framework is then exemplified by instantiating the reference implementation relation with some existing relations. Some new results about these relations are also presented.

We have tried to make as few mathematical hypotheses as possible about the reference implementation relation. For instance, we suppose that it is reflexive, but not necessarily symmetric and transitive. The reflexivity is justified by the fact that a specification is a valid implementation of itself. The symmetry is obviously not required since a specification and an implementation are not interchangeable in general. The transitive character is debatable but, in our opinion, not necessarily required. This point will be further discussed in section 2, and an interesting example of a non-transitive relation, viz. *conf*, will be analysed in section 8.

It is important to note that these implementation relations are studied per se, and that we do not intend to define transformation rules which could be applied in the design process to help in the automatic derivation of a valid implementation. Our results are however intended to help understand what relation should hold, and thus be verified, between two design stages for a given reference implementa-

tion relation. The results have also been extended to allow local transformations, i.e. transformations in any LOTOS context. This leads to consider the monotonicity of the LOTOS operators w.r.t. these relations, or, stated otherwise, the precongruent character of these relations. This framework is essential for dealing with non-transitive implementation relations. In particular, a counter-intuitive or apparently paradoxical result has been discovered, which states that the designer's freedom between two stages may be greater when the reference implementation relation is more restrictive.

2. Implementation relation concept

In this section we reason in a very general way about the concept and the purpose of a formal implementation relation. Such relation, denoted *imp* in the sequel, is intended to express the conditions that an implementation must fulfil in order to be considered as valid with respect to a specification. We will not define *imp* in this section, but rather develop a generic framework where suitable properties of *imp* are discussed when we consider the stepwise derivation of an implementation from a specification.

A specification is an unambiguous description of an object at a relatively high level of abstraction. Very often, either the goal is not to implement this object as such, or it is simply impossible to implement it. There are various reasons such as the level of abstraction, the non-constructive character of the specification or the specification model. This justifies in practice the need for a transformation process starting with a very abstract specification and ending up with an implementation. There may be intermediate stages in this process.

Before going further, let us remark that, as discussed in [7], the term *implementation* does not mean physical implementation, but rather a specification at a very low level of abstraction, i.e. a model of the physical implementation. This makes it possible to include a notion of implementation in a formal theory. The link between a physical reality and its model remains of course informal by nature. Therefore, in this paper, an implementation is a formal description at a very low level of abstraction, which is not supposed to be refined or transformed any further in a formal way. It is thus needed to translate it or compile it into, e.g., a more classical programming language such as C, Pascal or Ada.

The transformations taking place between the different stages of the design preserve some behavioural characteristics while changing others. A less abstract specification is usually required to respect in some sense a more abstract specification, but this does not necessarily mean that they have to exhibit exactly the same behaviour. For example, the functionality may be reduced or extended, non-determinism may be removed partially.

The notion of validity of an implementation w.r.t. a specification is not expressed in the specification itself. This is not surprising and probably suitable because it allows for the definition of several such notions. The same approach has led to the definition of several equivalence relations between specifications. Therefore a formal relation, *imp*, should be defined in order to decide whether or not a given

implementation is valid with respect to the specification. The pair “specification + implementation relation” is a definition, or a compact notation, characterizing in fact a (possibly infinite) set of valid implementations. Again the choice of an equivalence relation is similar in the sense that the pair “specification + equivalence relation” defines a class of specifications, all of them being equivalent according to the relation. We consider however that the notion of formal implementation has a more fundamental nature because it induces naturally, whatever it is (see definition 2.1), an equivalence relation.

Some properties of *imp* can be stated a priori.

Imp must be reflexive because the specification is a valid implementation of itself. From now on, *imp* is therefore reflexive. If we denote *Id* the identity relation which is the smallest reflexive relation, we have $\underline{Id} \subseteq \underline{imp}$.

Moreover *imp* is not required to be symmetric because an implementation and a specification are obviously not interchangeable in general.

What is less evident is the question of the transitivity of *imp*, i.e. whether we require that a valid implementation of a valid implementation is again a valid implementation. Note that if *imp* is not transitive, the valid implementation cannot be used in turn as an intermediate specification; which is not its purpose anyway.

We will show how *imp* induces naturally an equivalence relation, and how *imp* restricts the implementer’s freedom at each design step. This latter point will be achieved by way of the derivation of the weakest relation, denoted *imp-restr*, which must hold between two intermediate design stages. Let us already note that *imp-restr* will naturally be transitive, whatever *imp* is.

The results will also be extended to allow local transformations, i.e. transformations in any LOTOS context. This leads to consider the monotonicity of the LOTOS operators w.r.t. *imp-restr* (and not *imp*), or, stated otherwise, the precongruent character of *imp-restr* in the LOTOS contexts.

We now start the study of the framework by presenting the natural equivalence induced by *imp* and denoted *imp–eq*.

Definition 2.1

$S_1 \; \underline{imp\text{-}eq} \; S_2 \quad iff \quad \{I \mid I \; \underline{imp} \; S_1\} = \{I \mid I \; \underline{imp} \; S_2\}$

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

Intuitively, two specifications are equivalent iff 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.

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 set of valid implementations. 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 iff each one is a valid implementation of the other one. However, $imp \cap imp^{-1}$ is not necessarily an equivalence.

Proposition 2.2
$imp\text{-}eq \subseteq imp \cap imp^{-1}$

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*.

We now introduce another relation, denoted *imp-restr*, which plays a central role in the stepwise derivation of a valid implementation from a given specification. We consider that each step of this activity can only restrict the set of valid implementations, the final stage being the expected implementation which is therefore valid by construction.

Definition 2.3
S_2 *imp-restr* S_1 *iff* $\{I \mid I \; imp \; S_2\} \subseteq \{I \mid I \; imp \; S_1\}$

The purpose of this relation will become apparent in the sequel. Let us already note that replacing the specification S_1 by another specification S_2 (closer to an implementation) such that S_2 *imp-restr* S_1 guarantees that the remaining valid implementations are valid implementations of S_1.

Propositions 2.4

(i) *imp-restr* is a preorder (i.e. a reflexive and transitive relation)

(ii) $imp\text{-}restr \supseteq imp\text{-}eq$

(iii) $imp\text{-}restr \cap imp\text{-}restr^{-1} = imp\text{-}eq$

The proofs are straightforward.

Now we may come back to the question of the transitivity of *imp*, i.e. should *imp* be transitive? We think that arguments in favour of the transitive nature of *imp* are coming from the confusion between *imp* and *imp-restr*. Remember that *imp* expresses the formal link between two objects, a specification and an implementation; the transitive nature has no sense at this level since there are no notion of composition of this relation. By contrast, *imp-restr* is intrinsically related to the stepwise nature of the implementation process, since *B* *imp-restr* *A* means that B is closer to an implementation than A, or

that B restricts the set of valid implementations more than A does. This is the reason why it is not surprising that *imp-restr* is naturally transitive whatever *imp* is.

The transitive relation *imp-restr* results from the choice of *imp*, which is not required to be transitive. These two relations will be further related by the next propositions.

Proposition 2.5

imp-restr $\subseteq$ *imp*

The proof is easy from the definition of *imp-restr* and the reflexivity of *imp*.

Proposition 2.6

imp o imp-restr = imp

where the symbol 'o' denotes composition of relations.

Proof

The proof follows from the two following arguments

(i) *Id* $\subseteq$ *imp-restr* $\Rightarrow$ *imp* $\subseteq$ *imp o imp-restr*

(ii) *imp o imp-restr* $\subseteq$ *imp*

because if I *imp* $S_2 \wedge S_2$ *imp-restr* S_1 then I *imp* S_1, by definition of *imp-restr*

❐

Let us now consider some properties of the relations stronger than *imp-restr*. These properties will characterize *imp-restr* and outline its fundamental role in the implementation process.

Proposition 2.7

$\forall$ *R, we have R* $\subseteq$ *imp-restr* $\Leftrightarrow$ *imp o R* $\subseteq$ *imp*

Proof

($\Rightarrow$) Directly from the two following facts:

R $\subseteq$ *imp-restr* $\Rightarrow$ *imp o R* $\subseteq$ *imp o imp-restr*, and

imp o imp-restr = imp

($\Leftarrow$) By contradiction, consider any relation *R* such that *R* $\subseteq$ *imp-restr* does not hold. We will prove that *imp o R* $\subseteq$ *imp* does not hold.

By hypothesis, there exist P and Q such that P *R* $Q \wedge \neg (P$ *imp-restr* $Q)$ (*)

$\neg (P$ *imp-restr* $Q) \Rightarrow \exists I$, *such that* I *imp* $P \wedge \neg (I$ *imp* $Q)$ (**)

We deduce from (*) and (**) that $\exists I, P, Q$, *such that* I *imp* $P \wedge P$ *R* $Q \wedge \neg (I$ *imp* $Q)$, and therefore $\neg$ (*imp o R* $\subseteq$ *imp*)

❐

If we restrict ourselves to the *reflexive* relations stronger than *imp-restr*, we get the following result.

Proposition 2.8

$\forall$ *R, we have Id* $\subseteq$ *R* $\Rightarrow$ (*R* $\subseteq$ *imp-restr* $\Leftrightarrow$ *imp o R = imp*)

This is easily derived from proposition 2.7 noting that $\underline{Id} \subseteq \underline{R} \Rightarrow \underline{imp} \subseteq \underline{imp} \; o \; \underline{R}$.

Corollary 2.9

$\underline{imp\text{-}restr}$ may be defined as the least relation $\underline{R}$ such that $\underline{imp} \; o \; \underline{R} = \underline{imp}$

This is a direct consequence of propositions 2.6 and 2.7. This corollary is fundamental; its importance will be apparent in section 3.

If $\underline{imp}$ is transitive, we get the following stronger results.

Proposition 2.10

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

In this case two specifications are equivalent iff each one is a valid implementation of the other one.

Proof

By proposition 2.2 it suffices to prove that $\underline{imp} \cap \underline{imp}^{-1} \subseteq \underline{imp\text{-}eq}$ or, stated otherwise, that $\forall P, Q,$ *we have* $P \; \underline{imp} \; Q \wedge Q \; \underline{imp} \; P \Rightarrow \{I \mid I \; \underline{imp} \; P\} = \{I \mid I \; \underline{imp} \; Q\}$
Consider any I such that $I \; \underline{imp} \; P$. By hypothesis $P \; \underline{imp} \; Q$, which implies by transitivity of $\underline{imp}$ that $I \; \underline{imp} \; Q$. The proof is similar if we consider any I such that $I \; \underline{imp} \; Q$.

❒

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

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

Proposition 2.11

$\underline{imp}$ is transitive iff $\underline{imp\text{-}restr} = \underline{imp}$

Proof

($\Rightarrow$) From proposition 2.5, it suffices to prove that $\underline{imp} \subseteq \underline{imp\text{-}restr}$. This is obvious because if $P \; \underline{imp} \; Q$, for any I such that $I \; \underline{imp} \; P$ we derive $I \; \underline{imp} \; Q$ by transitivity of $\underline{imp}$.

($\Leftarrow$) $\underline{imp} = \underline{imp\text{-}restr}$ implies $\underline{imp} \; o \; \underline{imp} = \underline{imp} \; o \; \underline{imp\text{-}restr}$
Since $\underline{imp} \; o \; \underline{imp\text{-}restr} = \underline{imp}$, by proposition 2.6, we get $\underline{imp} \; o \; \underline{imp} = \underline{imp}$

❒

Proposition 2.12

$\underline{imp}$ is transitive $\Leftrightarrow (\forall \underline{R}$, *we have* $\underline{Id} \subseteq \underline{R} \Rightarrow (\underline{R} \subseteq \underline{imp} \Leftrightarrow \underline{imp} \; o \; \underline{R} = \underline{imp}))$

Proof

($\Rightarrow$) $\underline{imp}$ transitive $\Rightarrow \underline{imp\text{-}restr} = \underline{imp}$, therefore the right-hand side of proposition 2.12 is correct by proposition 2.8.

($\Leftarrow$) Obvious, take $\underline{R} = \underline{imp}$ ❒

3. Allowed transformations w.r.t. an implementation relation

We now consider the activity of deriving an implementation from a specification. This activity is supposed to be a stepwise transformation where each step consists in replacing a specification by another one according to some design criteria. This transformation process is a synthesis activity which is likely to be empirical but which is usually required to preserve some properties expressed by a suitable relation, denoted *R* in the sequel. This means that *S* can be replaced by *S'* in a step iff *S' R S*. We may imagine as many relations *R* as we want if they express some practical criteria. However, if we require that a step can only restrict the valid implementations, the choice of *imp* imposes constraints on the allowed relations *R*.

Definition 3.1

(i) A transformation from *S* to *S'* is an *R*-transformation iff *S' R S*

(ii) An *R*-transformation is allowed by *imp* iff

$\forall$ *S, S', we have* $S' \underline{R} S \Rightarrow \{I \mid I \underline{imp} S'\} \subseteq \{I \mid I \underline{imp} S\}$,

or equivalently, iff $\underline{R} \subseteq \underline{imp\text{-}restr}$, by definition of *imp-restr*.

Intuitively *R* can be used to characterize some transformations allowed on *S* (and denoted *R*-transformations) iff the derived specification *S'* does not allow some new valid implementations (figure 3.1).

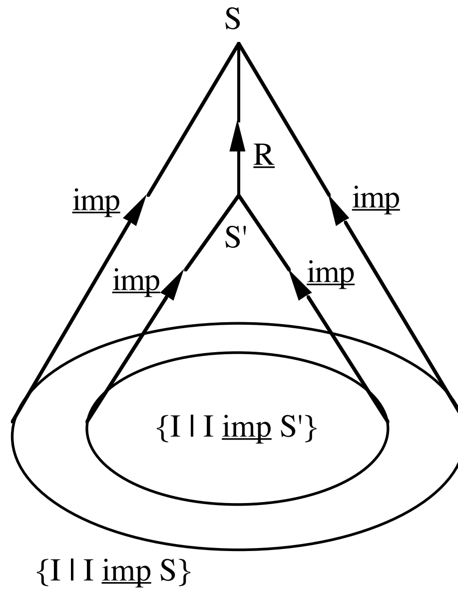

Figure 3.1: a transformation step

The next proposition shows another way of characterizing the allowed *R*-transformations.

Proposition 3.2

$\underline{R} \subseteq \underline{imp\text{-}restr} \Leftrightarrow \underline{imp} \circ \underline{R} \subseteq \underline{imp}$

The proof is immediate by proposition 2.7.

If *R* is reflexive, we have the following stronger result.

Proposition 3.3

$\underline{Id} \subseteq \underline{R} \Rightarrow (\underline{R} \subseteq \underline{imp\text{-}restr} \Leftrightarrow \underline{imp}\ o\ \underline{R} = \underline{imp})$

The proof is immediate from proposition 2.8

4. Link with the OSI Reference Model

Now, we apply this framework to study the relations which should hold between a service specification, a protocol specification, any intermediate specification and the final implementation.

The relation $\underline{R}_2$ (figure 4.1) should characterize some allowed transformations with respect to *imp* in order to ensure that the implementation is valid w.r.t. the protocol. The requirement is of course the same for $\underline{R}_1$. At this stage it does not seem mandatory that the service and the protocol be equivalent in some sense, but we will come back to this problem later on.

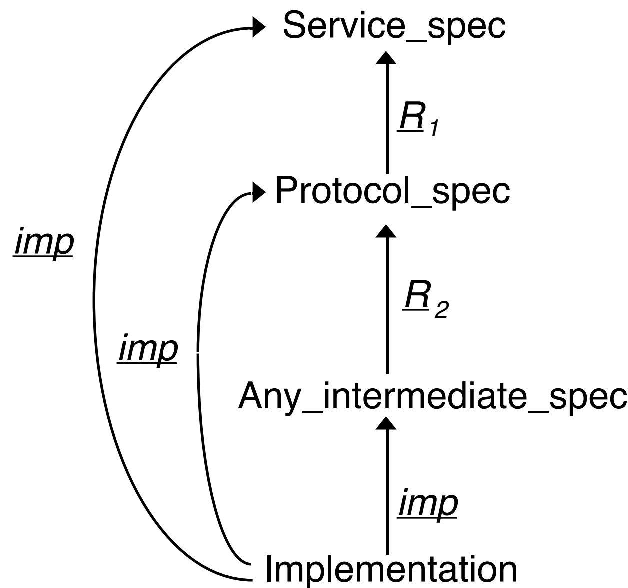

Figure 4.1: link with the OSI RM

The only requirement which arises from this study of the design process is therefore that all the transformation relations ($\underline{R}_1$, $\underline{R}_2$, …) be stronger than or equal to *imp-restr*. More precisely, it suffices to verify the following (we suppose that there is at least one intermediate step):

(i) *protocol_spec* *imp-restr* *service_spec*

(ii) *first_intermediate_spec* *imp-restr* *protocol_spec*

(iii) *any_intermediate_spec_but_the_first* *imp-restr* *previous_intermediate_spec*

(iv) *implementation* *imp* *last_intermediate_spec*.

Imp-restr may be called a *consistency* relation [7], because if *protocol_spec* *imp-restr* *service_spec*, for example, then any valid implementation of the protocol specification is also a valid implementa-

tion of the service specification. This property is in our opinion the weakest link required to hold between a protocol and a service, once the reference implementation relation, imp, has been adopted.

We may also require that the protocol specification should be *complete* [7] with respect to the service specification, i.e. *protocol_spec* imp-restr^{-1} *service_spec*. The argument being that if this does not hold, this complicates unnecessarily the layer above which has been designed to work with the service and not the protocol. If the protocol specification is not complete with respect to the service specification, we are sure that no valid implementation will be complete either. If it is considered as unacceptable, either the protocol or the service specification should be redesigned in order to get *protocol_spec* imp-eq *service_spec* (imp-eq = imp-restr $\cap$ imp-restr^{-1}).

A last property that imp-restr must fulfil arises when we consider the protocol and the service in the context of their upper layer. Imagine that we replace, in the specification of the protocol (N+1), the service (N) by the protocol (N), and suppose that *protocol (N)* imp-restr *service (N)* as required by the design process. It is important to ensure that, in so doing, we preserve the consistency of the protocol (N+1) with respect to the service (N+1), i.e. *protocol (N+1)* imp-restr *service (N+1)* still holds.

Formally, let us define these objects in LOTOS (figure 4.2):

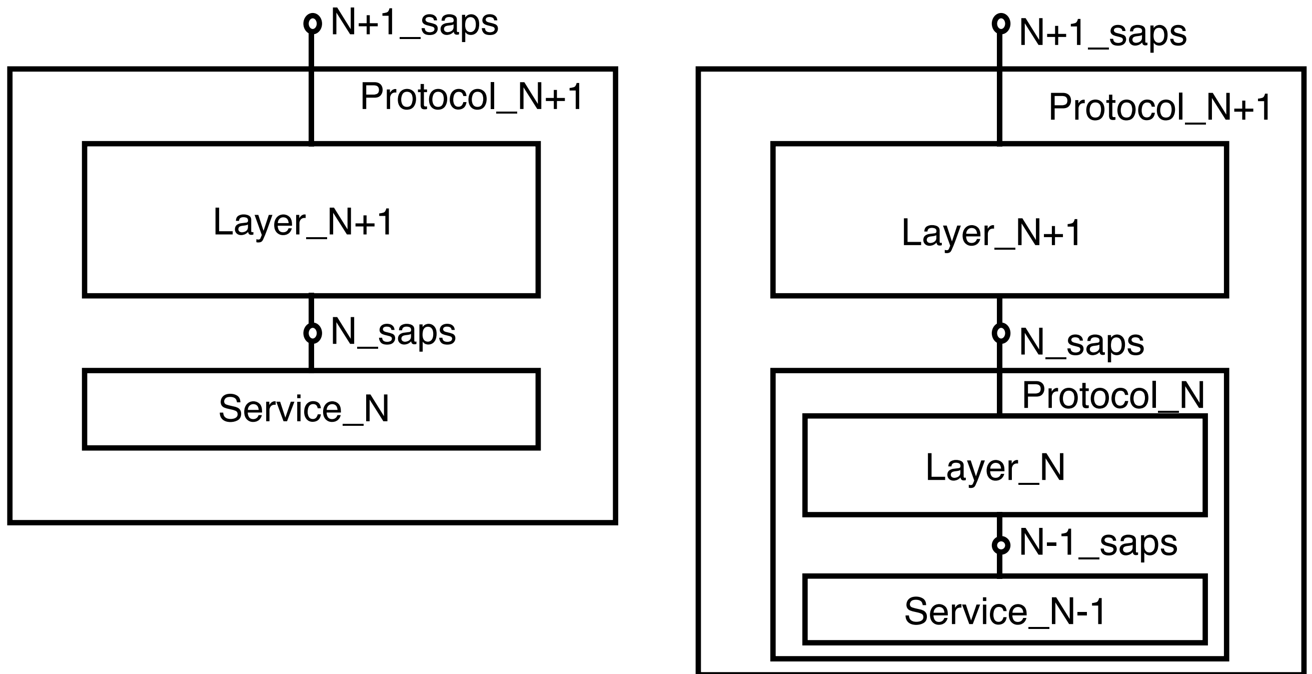

Figure 4.2: transformations within the OSI framework

protocol_N+1 [N+1_saps] := **hide** *N_saps* **in**
layer_N+1 [N+1_saps, N_saps]
|[N_saps]|
service_N [N_saps]

where *protocol_N [N_saps]* imp-restr *service_N [N_saps]*

This property can be stated as follows: it is necessary that

protocol_N+1 [N+1_saps] <u>*imp-restr*</u> *service_N+1 [N+1_saps]*

or stated otherwise, that

hide *N_saps* ***in*** *(layer_N+1 [N+1_saps, N_saps] |[N_saps]| protocol_N [N_saps])*
<u>*imp-restr*</u> *service_N+1 [N+1_saps].*

A similar case occurs when a layer (N) or an entity (N), say *X*, is replaced by another one, say *Y*, such that *Y* <u>*imp-restr*</u> *X*.

More generally, consider two processes *Q* and *S* synchronized through the gate set γ (i.e. *Q |[*γ*]| S*), and a process *P* such that *P* <u>*imp-restr*</u> *Q*.
It is necessary that *(hide* γ *in (P |[*γ*]| S))* <u>*imp-restr*</u> *(hide* γ *in (Q |[*γ*]| S))*

Two general properties that <u>*imp-restr*</u> must fulfil for the previous formula to hold, are:

(i) $\forall$ *P, Q, S, we require* *P* <u>*imp-restr*</u> *Q* $\Rightarrow$ $\forall\gamma$, *(P |[*γ*]| S)* <u>*imp-restr*</u> *(Q |[*γ*]| S)*
(ii) $\forall$ *P, Q, we require* *P* <u>*imp-restr*</u> *Q* $\Rightarrow$ $\forall\gamma$, *(hide* γ *in P)* <u>*imp-restr*</u> *(hide* γ *in Q)*

These properties are called substitution properties with respect to the parallel composition operator and the hiding operator. The substitution property in other contexts (such as action-prefix or choice) does not seem so important, as they have no associated architectural significance. However, we also often require these substitution properties in order to allow the replacement of <u>any</u> part *Q* of a specification by another part *P* such that *P* <u>*imp-restr*</u> *Q* and thereby guarantee that no new implementation would be allowed. If any of these substitution properties is not satisfied, <u>*imp-restr*</u> should be replaced by a stronger relation which satisfies them.

This stronger relation is precisely the least precongruence stronger than <u>*imp-restr*</u>, which will be defined after the notion of context.

Definitions 4.1
A *context C [.]* is a behaviour expression with a formal parameter '[.]', called a hole.
C [B] is *C [.]* where all occurrences of '[.]' have been replaced by *B*.
For example, if *C [.] = hide a in (A || [.])*, then *C [B] = hide a in (A || B)*.

Definition 4.2
The least precongruence stronger than <u>*imp-restr*</u> is denoted <u>*cimp-restr*</u> and defined by:
$\forall$ *P, Q, we have P* <u>*cimp-restr*</u> *Q iff* $\forall$ *context C [.], we have C [P]* <u>*imp-restr*</u> *C [Q].*

5. Another viewpoint on this implementation process

Now we take the problem the other way round and consider the possible reference implementation relations which are preserved when changes in the specification respect some well-known relation <u>*R*</u>.

This relation represents the degree of freedom of the designer, it means that (s)he may replace any part *Q* of an intermediate specification by *P*, provided that *P* R *Q*.

Definition 5.1

Let R be a reflexive relation,
imp is a relation preserved by R-transformations, iff
$\forall P, Q$, *we have* $P \; \underline{R} \; Q \Rightarrow \forall C[.], C[P] \; \underline{imp\text{-}restr} \; C[Q]$.

Informally, imp is a relation preserved by R-transformations, iff when we limit the replacements of any part *Q* of a specification *C [Q]* by processes *P* such that *P* R *Q*, then the new specification *C [P]* has fewer valid implementations (in the imp sense) than *C [Q]*.

Note that it is imp-restr, and not imp, which is needed in this definition: as explained in section 3, if imp is the relation which should hold between the first and the last stages of the design process, imp-restr should hold between any two intermediate stages.

Proposition 5.2

Let R be a reflexive relation,
imp is a relation preserved by R-transformations, iff $\underline{cimp\text{-}restr} \supseteq R$

Proof

(⇒) Since imp is a relation preserved by R-transformations, we get by definition 5.1 that $\forall C[.]$, *we have* $C[P] \; \underline{imp\text{-}restr} \; C[Q]$ and then by definition 4.2:
$P \; \underline{cimp\text{-}restr} \; Q$

(⇐) Suppose $P \; \underline{R} \; Q$.
From $\underline{R} \subseteq \underline{cimp\text{-}restr}$, we get $P \; \underline{cimp\text{-}restr} \; Q$.
Then by definition 4.2, $\forall C[.]$, *we have* $C[P] \; \underline{imp\text{-}restr} \; C[Q]$.

❐

Proposition 5.3

Let R be a precongruence,
imp is a relation preserved by R-transformations, iff $\underline{imp\text{-}restr} \supseteq \underline{R}$.

Proof

(⇒) Trivial by proposition 5.2

(⇐) If $\underline{R} \subseteq \underline{imp\text{-}restr}$ and R is a precongruence, then $\underline{R} \subseteq \underline{cimp\text{-}restr}$. The result then follows from proposition 5.2.

❐

Note again that we get $\underline{imp\text{-}restr} \supseteq \underline{R}$ and not $\underline{imp} \supseteq \underline{R}$.

Proposition 5.4

Let R be a precongruence,
imp is a relation preserved by R-transformations, iff $\underline{imp} \; o \; \underline{R} = \underline{imp}$

The proof is immediately derived from proposition 2.8.

Informally, if we limit the replacements of any part *Q* of a specification *C [Q]* by processes *P* such that *P* $\underline{R}$ *Q* (where $\underline{R}$ is a precongruence), then for any reference implementation relation $\underline{imp}$ such that $\underline{imp} \circ \underline{R} = \underline{imp}$, the new specification *C [P]* has fewer valid implementations (in the $\underline{imp}$ sense) than *C [Q]*.

We finally conclude this framework with some propositions indicating that things are not always as intuitive as we would think.

Proposition 5.5

Let $\underline{R_1}$ and $\underline{R_2}$ be two reflexive relations.

If $\underline{R_1} \subseteq \underline{R_2}$ and $\underline{imp}$ is a relation preserved by $\underline{R_2}$-transformations

then $\underline{imp}$ is a relation preserved by $\underline{R_1}$-transformations.

Proof

From $\underline{R_2} \subseteq \underline{cimp\text{-}restr}$ and $\underline{R_1} \subseteq \underline{R_2}$, we get immediately $\underline{R_1} \subseteq \underline{cimp\text{-}restr}$. ❒

This proposition is intuitively obvious, because if you restrict the freedom of the designer, then more implementation relations will be preserved. The next proposition however hurts intuition because it is very close to the previous one and seems to contradict it.

Proposition 5.6

Let $\underline{R}$ be a reflexive relation.

If $\underline{imp_1} \subseteq \underline{imp_2}$ and $\underline{imp_1}$ is a relation preserved by $\underline{R}$-transformations

then **it is not necessarily true** that $\underline{imp_2}$ is a relation preserved by $\underline{R}$-transformations.

We show an interesting counter-example in section 9. This proposition seems paradoxical. It means that if you choose a more generous implementation relation (i.e. a relation which allows more valid implementations), then it may happen that, for a given degree of freedom of the designer (made precise by $\underline{R}$), you may produce an implementation which was valid w.r.t. the former and no more valid w.r.t. the latter !

Of course, if we replace $\underline{imp_i}$ by $\underline{imp_i\text{-}restr}$ in this proposition, the result holds, viz.

if $\underline{imp_1\text{-}restr} \subseteq \underline{imp_2\text{-}restr}$ and $\underline{imp_1\text{-}restr}$ is a relation preserved by $\underline{R}$-transformations

then $\underline{imp_2\text{-}restr}$ is a relation preserved by $\underline{R}$-transformations.

But, the point remains that the $\underline{imp_i\text{-}restr}$ are NOT the implementation relations, they are only the weakest relations which should hold between any two intermediate stages in the design process in order to preserve $\underline{imp_i}$ at the end.

The next proposition states a sufficient condition on $\underline{imp_2}$ to solve this problem.

Proposition 5.7

Let $\underline{R}$ be a reflexive relation.

If $\underline{imp_1} \subseteq \underline{imp_2}$ and $\underline{imp_1}$ is a relation preserved by $\underline{R}$-transformations and $\underline{imp_2}$ is transitive
then $\underline{imp_2}$ is a relation preserved by $\underline{R}$-transformations.

Proof

We know that $\underline{imp_1\text{-}restr} \subseteq \underline{imp_1}$, by proposition 2.5
$\subseteq \underline{imp_2}$, by hypothesis
$= \underline{imp_2\text{-}restr}$, by proposition 2.11

Therefore, $\underline{cimp_1\text{-}restr} \subseteq \underline{cimp_2\text{-}restr}$,
directly from definition 4.2 and $\underline{imp_1\text{-}restr} \subseteq \underline{imp_2\text{-}restr}$

Finally, $\underline{R} \subseteq \underline{cimp_1\text{-}restr}$, by hypothesis
$\subseteq \underline{cimp_2\text{-}restr}$, just derived

❒

More generally, the next proposition states the fundamental result.

Proposition 5.8

If $\underline{cimp_1\text{-}restr} \subseteq \underline{cimp_2\text{-}restr}$ and $\underline{imp_1}$ is a relation preserved by $\underline{R}$-transformations
then $\underline{imp_2}$ is a relation preserved by $\underline{R}$-transformations.

The proof is simply the last part of the proof of proposition 5.7.

6. Discussion of the framework

One may argue that an implementation process where the specification phase and the implementation phase are clearly separated is overly naive and does not match reality [27], specification and implementation being respectively the already-fixed and the yet-to-be-done portions of a multi-step system development. In our framework, each stage of the implementation process should be a valid realization of the specification. By valid we mean that behaviours specified by the intermediate specification are a subset of those defined by the specification. It is not sure that this approach matches current practice at least in an environment where the initial specification is not a standard. It seems that, in actual practice, the intermediate steps may violate the validity condition, i.e. rather than implementing the specification, the designer knowingly redefines the specification. This may arise for several reasons among which we find the impossibility to foresee all the implications of specification choices, some of them leading to undesirable effects. One is thus forced to consider these steps as being part of the specification process and not part of the implementation process. The problem is to fix the point behind which all the steps must satisfy the validity condition. This point is then considered as the reference specification.

Another way to allow some violation of the validity condition is to consider that any intermediate stage should be valid w.r.t. the *initial* specification. This is less restrictive than being valid w.r.t. the previous intermediate stage. For example, one may have cases where

$interm_1$ imp-restr *spec*,
but $\neg$ ($interm_2$ imp-restr $interm_1$)
whereas $interm_2$ imp-restr *spec*.

Clearly, $interm_1$ is a valid transformation of the initial specification, *spec*, and $interm_2$ is not a valid transformation of $interm_1$ but is valid w.r.t. *spec*.

This does not change fundamentally the framework since in such cases, it suffices to discard $interm_1$ and consider that $interm_2$ is the first valid intermediate stage; the process may then continue this way from $interm_2$.

Finally, let us remark that many implementation relations, or notions of validity, may coexist and thus many equivalence relations too. The framework presented in this section makes no assumption on the reference implementation relation, except that it must be reflexive. In particular, the framework will be very useful to deal with the non-transitive relation conf in the next sections where we discuss the properties of well-known relations with respect to this framework. These relations are based on traces and refusals, but other kinds of relations could be considered similarly.

7. Definitions and properties of well-known implementation relations

Some implementation relations have been proposed in [7, 8]. We briefly recall them in a trace-refusal formalism.

Notations 7.1

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

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

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

$P\text{—}a.b\text{→}P'$ means $\exists\, P''$, such that $P\text{—}a\text{→}P'' \;\wedge\; P''\text{—}b\text{→}P'$.

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

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

$P{=}a{\not\Rightarrow}$ 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 $\sigma = a_1\ldots a_n$ where $a_1, \ldots a_n \in L$:
$\exists\, k_0, \ldots k_n \in N$, such that $P\text{—}i^{k_0}.a_1.i^{k_1}.a_2 \ldots a_n.i^{k_n}\text{→}P'$

$P{=}\sigma{\Rightarrow}$ 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^*.

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

$Ref\ (P, \sigma) = \{X \mid \exists P' \in P\ after\ \sigma,\ such\ that\ P'{=}a{\not\Rightarrow},\ \forall a \in X\};$

$Ref\ (P, \sigma)$ 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 [7, 8] by means of three basic relations, viz. <u>conf</u>, <u>red</u> and <u>ext</u> , and two related ones, viz. <u>cred</u> and <u>cext</u>. Some related equivalences have also been proposed, viz. <u>te</u> and <u>tc</u>.

Definitions 7.2

Let P_1 and P_2 be processes.

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

iff $\forall\ \sigma \in Tr\ (P_1) \cap Tr\ (P_2)$, *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 <u>conf</u> 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 [10], and is denoted the *conformance* relation.

Example: *A [a, b] := a; b; stop*

B [a, b, c] := i; a; stop [] b; c; stop

We get *A* <u>conf</u> *B*:

Let us consider all the traces σ of B: $\{\varepsilon, a, b, bc\}$; either the trace is not a trace of A (for example, b or bc), or $\forall A' \in A\ after\ \sigma$, if A' refuses any event in a set K, then $\exists B' \in B\ after\ \sigma,\ such\ that\ B'$ refuses any event in the same set K. For example, let $A' \in A\ after\ a$, i.e. $A' = b; stop$, which thus refuses any event in $\{a, c\}$; now, $\exists B' \in B\ after\ a$, viz. $B' = stop$ which also refuses any event in $\{a, c\}$.

P_1 <u>red</u> P_2 iff (i) $Tr\ (P_1) \subseteq Tr\ (P_2)$, and

(ii) P_1 <u>conf</u> P_2

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

Example: *A [a] := a; stop*

B [a, b, c] := i; a; stop [] b; c; stop

We get *A* <u>red</u> *B*

P_1 <u>ext</u> P_2 iff (i) $Tr\ (P_1) \supseteq Tr\ (P_2)$, and

(ii) P_1 <u>conf</u> P_2

Intuitively, if P_1 <u>ext</u> 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. <u>Ext</u> is the *extension* relation.

Example: *A [a, b, c] := a; b; stop [] b; c; stop*

B [a, b] := a; b; stop

We get *A* ext *B*

P_1 cred P_2 iff (i) P_1 red P_2

(ii) $Stable\ (P_2) \Rightarrow Stable\ (P_1)$

where a process is stable when it cannot perform an initial internal action;

more formally, *Stable (P)* iff $\neg \exists P'$, such that $P{-}i{\rightarrow}P'$;

cred is the least precongruence stronger than red [7] (see note hereafter).

cext = ext ∩ cred This is the least precongruence stronger than ext [7] (see note hereafter).

te = red ∩ red^{-1} = ext ∩ ext^{-1} This is the testing equivalence.

tc = cred ∩ $cred^{-1}$ = cext ∩ $cext^{-1}$

This is the least congruence stronger than te [7] (see note hereafter).

Note: cred, cext and tc are (pre)congruences only if we exclude hiding contexts creating divergence, i.e. an infinite sequence of internal actions. This is studied in detail in [21].

Properties of conf

(i) conf ⊃ red

(ii) conf ⊃ ext

(iii) conf = red o ext

Properties (i) and (ii) are directly derived from the definitions.
Property (iii) has been first established in [7]; another proof is given in [21].

In the last part of this section 7, we present new results or generalize previous results about these relations.

Proposition 7.3

conf ⊂ ext o red

This generalizes the result in [7] where it was proved by a counter-example that ext o red ⊆ conf does not hold. This counter-example allows us to only prove the weaker proposition conf ⊆ ext o red. The proof is provided in [21].

In the sequel, we compose the relations defined above in order to identify some new relations.

Proposition 7.4

ext o conf = ext o red

Proof

From conf ⊂ ext o red, we get by left composition with ext: ext o conf ⊆ ext o ext o red
Now, by transitivity and reflexivity of ext , we have ext o ext o red = ext o red, thus leading to ext o conf ⊆ ext o red (*)

On the other hand, from *red* $\subset$ *conf*, we have by left composition with *ext*:
ext o *red* $\subseteq$ *ext* o *conf*. (**)
The result follows from (*) and (**).

❐

Proposition 7.5

conf o *conf* = *conf* o *red*

Proof

From *ext* o *red* = *ext* o *conf*, we get by left composition with *red*: *red* o *ext* o *red* = *red* o *ext* o *conf*, which is equivalent to *conf* o *red* = *conf* o *conf* , because *conf* = *red* o *ext*

❐

Proposition 7.6

conf o *red* = *ext* o *red*

Proof

From *conf* $\subset$ *ext* o *red* , we derive successively

red o *ext* $\subset$ *ext* o *red*,	by definition of *conf*
red o *ext* o *red* $\subseteq$ *ext* o *red* o *red*	by right composition with *red*
conf o *red* $\subseteq$ *ext* o *red* o *red*,	by definition of *conf*
conf o *red* $\subseteq$ *ext* o *red*,	by transitivity and reflexivity of red (*)

On the other hand, from *conf* $\supset$ *ext*, we derive by right composition with *red*:
conf o *red* $\supseteq$ *ext* o *red* (**)
The result follows from (*) and (**).

❐

Proposition 7.7

*conf** = *conf* o *conf*

where *conf** denotes the transitive closure of *conf* (informally, *conf* o *conf* o *conf* o …).

In other words, *conf* o *conf* is the transitive closure of *conf*, which is the strongest transitive relation weaker than *conf*.

Proof

It suffices to prove that *conf* o *conf* o *conf* = *conf* o *conf*.
conf o *conf* o *conf*

= *conf* o *conf* o *red*	by proposition 7.5
= *conf* o *red* o *red*	by proposition 7.5
= *conf* o *red*	by transitivity and reflexivity of *red*
= *conf* o *conf*	by proposition 7.5

❐

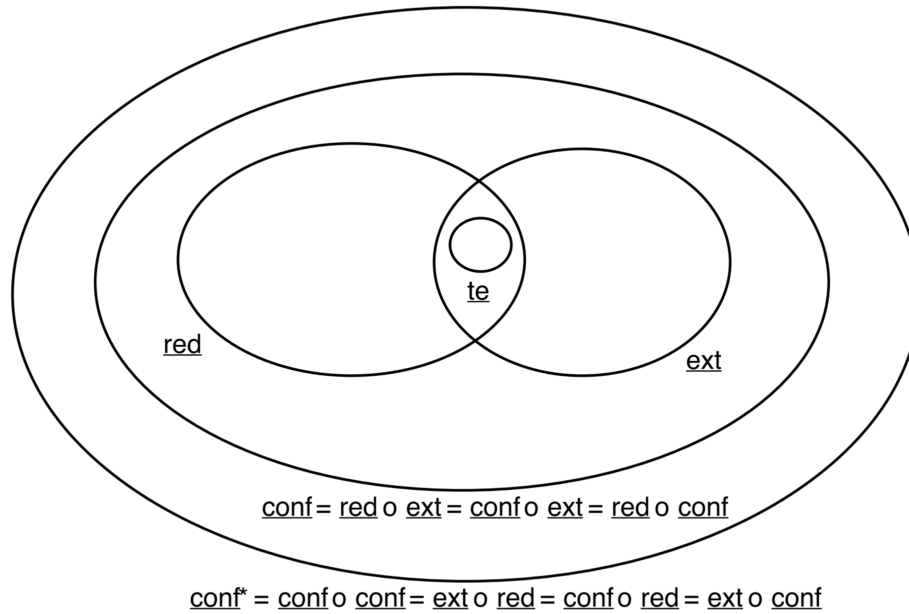

Figure 7.1: relations between *te*, *red*, *ext*, *conf*, *conf**

Proposition 7.8

*conf** = *ext* o *red*

The proof follows directly from propositions 7.7, 7.5 and 7.6.

All these results are summarized in figure 7.1.

Figure 7.2 shows how *cred*, *cext* and *tc* are related to these relations.

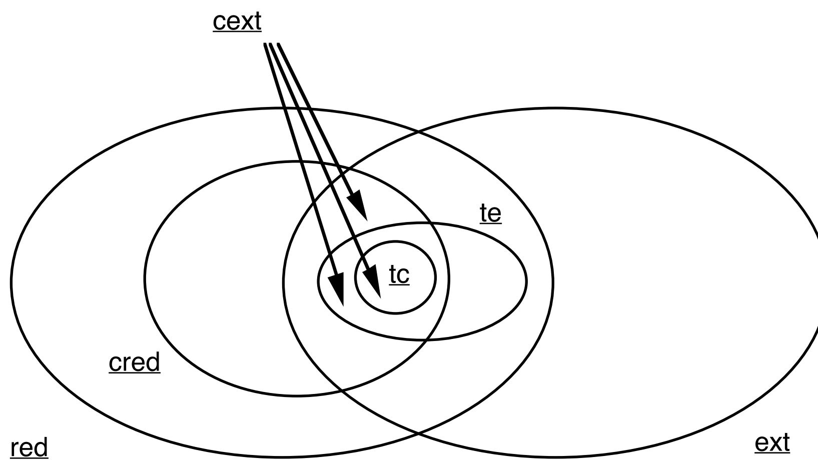

Figure 7.2: relations between *tc*, *te*, *cred*, *red*, *cext*, *ext*

8. *Conf* as reference implementation relation

Conf has been taken as the formal basis of conformance testing in [10]. Since conformance testing is primarily concerned with the testing of valid implementations, *conf* is a good candidate as reference implementation relation. Moreover, *conf* is an interesting example of non-transitive relation.

This section studies the implementation framework when *imp* has been instantiated by *conf*. Consequences and perspectives of this choice are also analysed. *Conf* is reflexive and thus satisfies the only prerequisite, but *conf* is not transitive. Consider the following example from [7]:

Example: *b; stop [] i; a; stop* *conf* *i; a; stop* *conf* *b; c; stop [] i; a; stop*
but ¬ *(b; stop [] i; a; stop* *conf* *b; c; stop [] i; a; stop)*

A first interesting question about *conf* is the nature of the preorder *conf-restr* and the equivalence *conf-eq* which are related to *conf*. We first instantiate some general results stated in section 2, and then propose other equivalent definitions of *conf-restr* and *conf-eq*.

Proposition 8.1
conf-eq ⊂ *conf-restr* ⊂ *conf*

Derived immediately from propositions 2.4 and 2.5, since *conf* is not transitive.

Proposition 8.2
conf-eq ⊂ *conf* ∩ $conf^{-1}$

Proof
From proposition 2.2, we derive *conf-eq* ⊆ *conf* ∩ $conf^{-1}$
Moreover, there exist *P* and *Q* such that *P conf Q* and *Q conf P* but ¬ *(P conf-eq Q)*, as shown hereafter:
Let *P = a; stop* and *Q = (a; stop [] a; b; c; stop)*, we get *P conf Q* and *Q conf P*, but
¬ *(P conf-eq Q)* because if we take *I = a; b; stop*, we have *I conf P* but ¬ *(I conf Q)*.

❐

Proposition 8.3
conf-eq ⊃ *te*

i.e., *conf-eq* is weaker than the testing equivalence

Proof
We have to prove that *P te Q* ∧ *I conf P* ⇒ *I conf Q*.
We first note that
(a) *P te Q* ⇒ *P ext Q* by definition of *te*
(b) *I conf P* ∧ *P ext Q* ⇒ *I conf o ext Q* by definition of composition
We know by proposition 7.2 (iii) and transitivity of *ext* that *conf o ext = conf*,
therefore *I conf Q* by (b).
Moreover there exist two processes *P* and *Q* such that *P conf-eq Q* but ¬ *(P te Q)*.
For example, *P = a; stop* *Q = i; a; stop [] b; stop*

¬ (P ext Q), so ¬ (P te Q),
It can be shown very easily that *P* and *Q* have exactly the same sets of conforming implementations, so *P conf-eq Q*

❐

Proposition 8.4

$\underline{conf\text{-}eq} \cap \underline{trace\text{-}eq} = \underline{conf} \cap \underline{conf}^{-1} \cap \underline{trace\text{-}eq} = \underline{te}$

or equivalently,

∀ P, Q, we have

$P\ \underline{conf\text{-}eq}\ Q \wedge (Tr(P) = Tr(Q)) \Leftrightarrow P\ \underline{conf}\ Q \wedge Q\ \underline{conf}\ P \wedge (Tr(P) = Tr(Q)) \Leftrightarrow P\ \underline{te}\ Q$

For processes with equal trace sets, $\underline{conf\text{-}eq}$, $\underline{conf} \cap \underline{conf}^{-1}$ and $\underline{te}$ are identical.

Proof

We know that $\underline{conf} \cap \underline{conf}^{-1} \cap \underline{trace\text{-}eq} = \underline{te}$, by definition of $\underline{te}$
So, $\underline{conf\text{-}eq} \cap \underline{trace\text{-}eq} \subseteq \underline{te}$, because $\underline{conf\text{-}eq} \subset \underline{conf} \cap \underline{conf}^{-1}$
It remains to prove that $\underline{te} \subseteq \underline{conf\text{-}eq} \cap \underline{trace\text{-}eq}$.
This is easily deduced from the two arguments:
(i) $P\ \underline{te}\ Q \Rightarrow Tr(P) = Tr(Q)$, or equivalently, $\underline{te} \subseteq \underline{trace\text{-}eq}$, by definition
(ii) $P\ \underline{te}\ Q \Rightarrow P\ \underline{conf\text{-}eq}\ Q$, or equivalently $\underline{te} \subseteq \underline{conf\text{-}eq}$, by the previous proposition.

❐

Proposition 8.5

$\underline{ext} \subset \underline{conf\text{-}restr}$

Proof

We know that $\underline{conf} \circ \underline{ext} = \underline{conf}$ by proposition 7.2 (iii) and transitivity of $\underline{conf}$.
By proposition 2.7, this implies $\underline{ext} \subseteq \underline{conf\text{-}restr}$
Moreover there exist *P* and *Q* such that *P conf-restr Q* but *¬ (P ext Q)*, see example of the proof of proposition 8.3.

❐

Note that by contrast, $\neg (\underline{red} \subseteq \underline{conf\text{-}restr})$. This will have some consequences which will be discussed later on.

All these results are summarized in figure 8.1. The shaded area is exactly the testing equivalence. Examples are provided in [21] to prove that the inclusions are strict and that no area is empty.

To gain a more intuitive understanding of *conf-restr* and *conf-eq*, we provide other equivalent definitions.

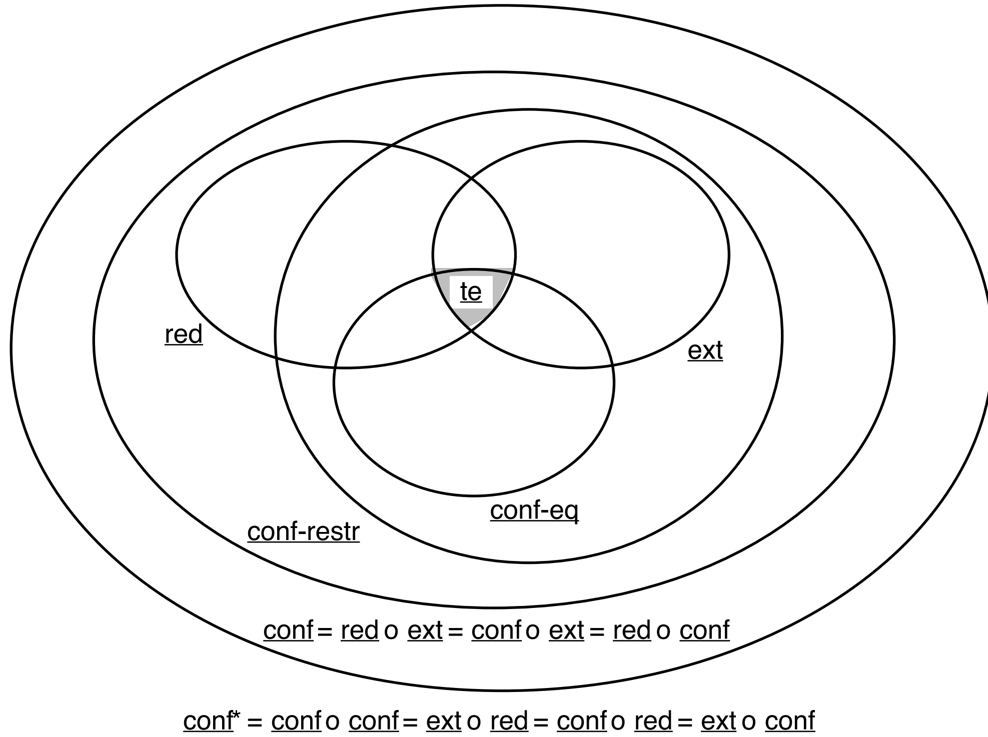

Figure 8.1: links between the main relations

Proposition 8.6

P conf-restr Q iff

(i) *P conf Q*

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

The proof is provided in [21].

Proposition 8.7

P conf-eq Q iff

(i) *P conf Q* ∧ *Q conf P*, i.e. $\forall\, \sigma \in Tr(P) \cap Tr(Q)$, *we have* $Ref(P, \sigma) = Ref(Q, \sigma)$

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

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

This follows directly from *conf-eq* = *conf-restr* ∩ $\textit{conf-restr}^{-1}$.

Note on *conf-eq* versus *te*

If *conf* is adopted as reference implementation relation, the testing equivalence is too strong, i.e. 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*:

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*. Therefore no distinction can be made between them by testing, provided that the test is based on the *conf* relation, which is for instance the case of the canonical tester [9].

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 fact may be explained intuitively as follows: a conforming implementation of *Q* may, or may not, accept *b* after *a*; but if it accepts *b*, then, unlike conforming implementations of *P*, it cannot refuse *c* afterwards.

On the allowed transformations w.r.t. *conf*

Proposition 8.5 states that *ext* ⊆ *conf-restr*, which means that *ext*-transformations are allowed by *conf*. More precisely, one may replace a specification by an extension of it, this transformation does not allow new conforming implementations; in fact this transformation generally reduces the set of valid implementations.

By contrast, it is very interesting to remark that *red* ⊆ *conf-restr* does not hold. The consequence is that *red*-transformations are not allowed by *conf*: reducing the nondeterminism of a specification *S* to obtain the specification *S'* may in general have the drawback of accepting as valid (i.e. conforming) implementations some processes which were not conforming implementations of *S*. The following example illustrates this problem:

let *S := i; a ; stop [] b; c; stop,*
S' := a; stop, *S' red S*
I := a; stop [] b; stop, *I conf S',* but ¬ *(I conf S).*

However *red* ∩ *ext* ⊂ *conf-restr,* which means that reducing the nondeterminism while preserving the traces has not the above-mentioned drawback.

Furthermore, *cred* ⊆ *conf-restr* does not hold either.

Conf seems to be a strange implementation relation, viz. when one reduces the nondeterminism of a specification, new conforming implementations may be allowed as a side effect, whereas when one extends the functionality, the set of conforming implementations is reduced.

Local transformations

In sections 4 and 5, we discussed the way to characterize the conditions that local transformations must satisfy in order to ensure that the resulting transformation of the global specification should be allowed w.r.t. to *imp*. This has led to the notion of the least precongruence stronger than *imp-restr*. The next question that we investigate is thus the nature of the least precongruence stronger than *conf-restr*.

Proposition 8.8

The least precongruence (see note after 7.2) stronger than conf-restr is cext,
i.e. *P cext Q iff ∀ context C [.], we have C [P] conf-restr C [Q].*

The proof is provided in [21].

Proposition 8.8 guarantees that if any part *Q* of a specification *C [Q]* is replaced by a process *P* such that *P cext Q*, we get a specification *C [P]* with *C [P] conf-restr C [Q]* and thus the set of implementations conforming to *C [P]* is included in (or equal to) the set of implementations conforming to *C [Q]*.

What is very interesting to note is that the least precongruence stronger than conf is cred (⊇ cext). But the non-transitive character of conf forced us to restrict the relation which should hold between intermediate specifications to a stronger transitive relation conf-restr, and this relation was no more weaker than cred.

Stated otherwise, if conf is the reference implementation relation, any design step consisting in replacing a process *Q* by a process P in any context[1] is guaranteed to preserve or restrict the conforming implementations, provided that *P cext Q*.

Finally, from proposition 8.8, we derive immediately that the least congruence (see note after 7.2) stronger than conf-eq is again tc.

9. An apparent paradox

Let us consider red as the reference implementation relation.

Red is reflexive and transitive, so by simple particularization of propositions 2.10, 2.11, 2.12, and 3.2, we get the next propositions.

Propositions 9.1

(i) *red-restr = red*

(ii) *red-eq = red* $\cap$ *red*$^{-1}$ *= te*

(iii) *∀ R, we have Id ⊆ R ⊆ red ⇒ red o R = red* , and

(iv) *R* -transformations are allowed by *red* iff *R ⊆ red*

The next proposition now recalls a well-known result (see [7]).

Proposition 9.2

The least precongruence (see note after 7.2) stronger than red is cred,
i.e. *P cred Q* iff *∀ context C [.], we have C [P] red C [Q].*

Proposition 9.2 guarantees that if any part *Q* of a specification *C [Q]* is replaced by a process *P* such that *P cred Q*, we get a specification *C [P]* with *C [P] red C [Q]* and thus the set of reductions of *C [P]* is included in (or equal to) the set of reductions of *C [Q]*.

If we consider the possible reference implementation relations which are preserved when the local changes in the specification respect some well-known precongruences, and compare the results of sections 5.6 and 5.7, we are facing some kind of paradox. Indeed, it appears that the freedom of the designer in changing parts of a specification without allowing new valid implementations may be greater when the reference implementation is stronger. If we compare the results derived from the two cases where *conf* and *red* have been chosen as reference implementation relations (see table 9.1), the degrees of freedom, modelled by the least precongruences stronger than respectively *conf-restr* and *red*, are respectively *cext* and *cred*. And of course *red* $\subset$ *conf* but *cred* $\supset$ *cext*. This is the counter-example mentioned in proposition 5.6. This is due to the fact that *conf o red* $\subseteq$ *conf* does not hold: *conf o red* = *conf**.

imp	*cimp-restr*	
conf	*cext*	*conf* $\supset$ *red*,
red	*cred*	whereas *cext* $\subset$ *cred*

Table 9.1: examples of imp and related cimp-restr relations

Consider *S := i; (i; a; stop [] c; stop) [] b; c; stop*
And *S' := i; a; stop [] c; stop* the result of a transformation of *S*.
S' red S
Let *I := a; stop [] b; stop, we get I conf S'* but ¬ *(I conf S)*
Conversely, $\forall$ *I , we have I red S'* $\Rightarrow$ *I red S.*

The following proposition summarizes the results.

Proposition 9.3
(i) Any reference implementation relation *imp* such that *imp o cext* = *imp* is preserved by *cext*-transformations.
(ii) Any reference implementation relation *imp* such that *imp o cred* = *imp* is preserved by *cred*-transformations.

The proofs follow directly from proposition 5.2

Examples 9.4
Examples of implementation relations preserved by *cext*-transformations:
cext, ext, cred, red, conf, conf-restr, conf*
Examples of implementation relations preserved by *cred*-transformations: *cred, red, conf**

Conf is not preserved by *cred*-transformations.

10. Conclusion

In this paper, our interest has been focused on so-called implementation relations, and their role in the design process has been underlined. We think that an equivalence is not necessarily the right relation to be proved or to be preserved throughout the implementation process. An equivalence is sometimes too restrictive to capture the link which should hold between two different specifications of a system. For instance, it seems reasonable to require that the service specification and the protocol specification should be trace equivalent, but not necessarily that they should be testing equivalent. One may accept for instance that the protocol could be more deterministic than the service.

Formally: (i) *protocol trace-eq service, and*
(ii) *protocol conf service*
i.e. *protocol* $\underline{\textit{trace-eq}} \cap \underline{\textit{conf}}$ *service*,
or equivalently, *protocol* $\underline{\textit{red}} \cap \underline{\textit{ext}}$ *service*.

In order to ensure that the service may be replaced by the protocol in any LOTOS context (see note after 7.2), one would require a little bit more, viz. *protocol* $\underline{\textit{cred}} \cap \underline{\textit{cext}}$ *service*,
or equivalently, *protocol cext service*.

The problem is similar in the design process where the initial formal specification is transformed several times before reaching a more implementable specification. Again, what is required is not necessarily the equivalence between the initial specification and the implementation specification. A more general view is to verify that the implementation specification is among the valid implementations of the initial specification where the notion of validity is formally expressed by an asymmetric relation. In this case, the successive transformations which take place in the design process are not restricted to equivalence preserving steps. There remain of course restrictions which are related to the notion of validity just mentioned. The link between these restrictions and the notion of validity has been investigated.

More precisely, we have introduced the basic notion of a reference implementation relation *imp*, and shown how an associated equivalence relation *imp-eq* is naturally deduced. The stepwise transformation process leading from the specification to a valid implementation has been discussed with respect to this implementation relation, i.e. a transformation may only restrict the set of valid implementations. The allowed transformations have been characterized by another relation denoted *imp-restr* and related to *imp*. Finally, the transformation relations have been discussed in the LOTOS operator contexts, in order to see how local transformations in a specification should be further restricted in particular contexts. This has led to consider the monotonicity of the LOTOS operators w.r.t. *imp-restr*, or more precisely, to define the least precongruence stronger than *imp-restr*.

In the second part, we have studied the well-known conf relation which may play the role of a reference implementation relation. We have derived the associated conf-restr and the least precongruence stronger than conf-restr which somehow model the freedom of the designer at each transformation. This study has put in evidence that the link between this freedom and the reference implementation relation is not necessarily as simple as we would think when the reference implementation relation in not transitive, which is the case of conf. This may lead to some apparent paradoxes as discussed in section 9.

Finally, there is an interesting line of research which we have only touched upon lightly in this paper, and which may be summarized by the following question:

what do we mean by a *valid* implementation of a specification?

or, what should be the nature of the link existing between the abstract formal specification and the final (compilable) stage of design modelling the implementation?

Some criteria exist and may be split into two categories:

- the intuitive criteria: e.g. an implementation may be more deterministic or cannot be more divergent than its specification;
- the technical or mathematical criteria: e.g. the LOTOS operators should be monotonic or even continuous w.r.t the implementation relation, or one of its associated relations.

Are there other interesting criteria which could be formalized in process algebraic techniques ?

Or, are there interesting extensions of existing models, such as the modal transition systems of [20], where more interesting implementation relations may be defined?

Acknowledgements

I thank very much Professor Ed Brinksma for his precise and judicious remarks on a preliminary version of my thesis from which the present paper is extracted.

References

[1] M. Abadi, L. Lamport,
The Existence of Refinement Mappings
in: Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland, July 88, 165-175.

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

[3] D. Blyth, E. Dubuis, H. Hansson, G. Juanole, M. Kapus-Kolar, H. Kerner, G. Leduc, G. Le Moli, A. Lombardo, S. Marchena, W. Orth, J. Pavon, B. Pehrson, M. Tienari, F. Vogt,
Architectural and Behavioural Modelling in Computer Communication,
in: M. H. Barton, E. L. Dagless, G. L. Reijns, eds., Distributed Processing (North-Holland, Amsterdam, 1988, ISBN 0-444-70419-1), 53-70.

[4] 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.

[5] K. Bogaards,
LOTOS supported system development,
in: K.J. Turner, ed., Formal Description Techniques (North-Holland, Amsterdam, 1989, ISBN: 0-444-87126-8) 279-294.

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

[7] 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, December 1986.

[8] 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.

[9] 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.

[10] 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).

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

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

[13] R. De Nicola,
Extensional Equivalences for Transition Systems,
Acta Informatica 24 (1987) 211-237.

[14] E. Dubuis, R. Gotzhein, H. Hansson, G. Juanole, H. Kerner, P. Lahtinen, G. Leduc, A. Lombardo, S. Marchena, W. Orth, S. Palazzo, J. Pavon, U. Thalmann, M. Tienari, I. Tvrdy,
A Framework for the Taxonomy of Synthesis and Analysis Activities in Distributed System Design
in: R. Speth, ed., Research into Networks and Distributed Applications (North-Holland, Amsterdam, 1988, ISBN 0-444-70428-0), 859-871.

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

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

[17] 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.

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

[19] S.S. Lam, A. U. Shankar,
Protocol verification via projections,
IEEE Transactions on Software Engineering, SE-10 (4) 325-342 (1984).

[20] 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.

[21] G. Leduc,
On the role of implementation relations in the design of distributed systems,
Agrégation dissertation, University of Liège, Dept. Systèmes et Automatique, B28, Liège, Belgium, July 1990.

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

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

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

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

[26] 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.

[27] W. Swartout, R. Balzer,
On the Inevitable Intertwining of Specification and Implementation,
Communications of the ACM, Vol. 25, No. 7, July 1982, 438-440.

[28] P. van Eijk,
Tools for LOTOS Specification Style Transformations,
in: S. T. Vuong, ed., FORTE ’89, Vancouver, Canada, Dec. 89 (North-Holland, Amsterdam, 1990).

[29] 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, ISBN 3-540-53048-7), 278-297.

[30] 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).