Equivalence associée à la relation de conformité, *conf*, et simplification du testeur canonique en LOTOS

G. Leduc

Chercheur qualifié du Fonds national de la recherche scientifique, Université de Liège, Institut d'électricité Montefiore, B 28,
B-4000 Liège 1, Belgique
Tél. : (32) 41.56.26.98 — Fax. : 32.41.56.29.89
u514401@bliulg11.bitnet leduc@montefiore.ulg.ac.be

Résumé. Cet article étudie la relation *conf* qui a été proposée par E. Brinksma pour formaliser la notion de conformité aux tests. On sait que pour tester si une implémentation *I* est conforme à une spécification *S* (i.e. *I conf S*), la technique utilisée consiste à construire, à partir de *S*, un testeur canonique *T (S)* tel que, lorsque *T (S)* est synchronisé avec une implémentation *I*, il atteint toujours un état terminal correct si, et seulement si, *I conf S*. Si *I* n'est pas conforme à *S*, le testeur canonique *T (S)* peut rester bloqué dans un état non terminal où toute interaction avec *I* est impossible. Dans cet article, nous mettons en évidence le rôle de la relation d'équivalence associée à *conf*, appelée *conf-eq*, qui est une équivalence plus faible que l'équivalence de test, *te*. Cette propriété nous permet de définir le testeur canonique *T (S)* modulo *conf-eq*; alors qu'il était défini modulo *te*. Ce nouveau testeur canonique est plus simple que le précédent, en ce sens que, contrairement à celui-ci, il peut contenir moins de traces que la spécification *S*. Nous montrons enfin que, dans le cas de spécifications non auto-bloquantes, les deux testeurs sont *te*-équivalents.
Mots-clés. FDT, LOTOS, algèbre de processus, test, testeur canonique, relation d'implémentation, conformité, spécification, implémentation, abstraction.

1. Introduction

Une version anglaise de cet article a été publiée à PSTV XI [Leduc 91c].

Cet article se rapporte d'une part à la problématique de l'implémentation des spécifications formelles exprimées dans le langage LOTOS [ISO 8807, Bolognesi 87], et d'autre part aux tests de conformité des implémentations. Dans ce contexte, plusieurs problèmes sont actuellement non résolus. Nous allons les résumer avant de situer notre contribution.

La première difficulté est due au fait que le processus d'implémentation fait intervenir d'une part une spécification formelle servant de référence, et d'autre part une réalisation physique de cette spécification. Afin de résoudre ce problème, on ne considère pas la réalisation physique elle-même, mais plutôt un modèle de cette réalisation [Brinksma 86]. Ce modèle, que nous appelerons *spécification d'implémentation* ou simplement *implémentation*, sera dès lors exprimé en LOTOS tout comme la spécification elle-même. Cette façon de faire a l'avantage de traiter le problème de l'implémentation dans un formalisme unique, et de rendre possible l'inclusion d'une notion d'implémentation dans une théorie formelle. Il peut sembler irréaliste d'utiliser LOTOS comme modèle d'une réalisation physique; ce n'est effectivement pas le domaine d'application idéal pour ce langage conçu initialement pour décrire des spécifications abstraites de normes de l'ISO. Toutefois, divers styles de spécifications ont été mis en évidence en LOTOS [Vissers 88] et permettent de moduler le niveau d'abstraction de la spécification. LOTOS peut même être utilisé comme un langage de programmation tel que C [Mañas 90] : la structure de la spécification et son niveau de détail (ou d'abstraction) est alors celui d'un programme C. Le code C généré à partir d'une telle spécification est à peine moins performant qu'un programme de même structure écrit directement en C. Dans cet article, nous travaillerons dès lors avec des modèles de réalisations, i.e. des implémentations, en LOTOS.

Le fait de disposer d'un formalisme unique nous permet d'aborder le deuxième problème, à savoir la nature du lien devant exister entre une implémentation conforme et sa spécification formelle. Le problème est double : il faut d'abord trouver les critères qui permettent de caractériser ce lien, et il faut ensuite les exprimer formellement. Ces formalisations peuvent se classer parmi deux grandes tendances : la conformité est caractérisée soit par une équivalence appropriée [Park 81, Brookes 84, de Nicola 84, de Nicola 87, Brinksma 87b, Hennessy 88, Milner 89], soit par une relation non nécessairement symétrique telle qu'un préordre.

Cette deuxième tendance est, selon nous, plus générale et plus appropriée car elle prend en compte le caractère asymétrique du lien entre implémentation et spécification [Brinksma 86, Brinksma 87b, Leduc 90, Leduc 91a, Leduc 91b]. Au lieu de considérer que l'implémentation doive être équivalente à la spécification, l'idée consiste à définir une relation moins restrictive et habituellement asymétrique. Ces relations sont souvent appelées des *relations d'implémentation*.

Ces relations ont été moins étudiées que les équivalences dans le contexte des techniques algébriques. Il n'existe d'ailleurs pas d'opinion bien établie sur la nature de ces relations d'implémentation, mais quelques tendances existent. Par exemple, il est souvent admis qu'une implémentation puisse être plus déterministe qu'une spécification. Selon cette vision, une relation

d'implémentation serait plutôt considérée comme un préordre (i.e. une relation réflexive et transitive). Un préordre, par son caractère asymétrique, définit un ordre partiel sur les systèmes. Si ce préordre est bien choisi, il peut être interprété comme une *relation d'implémentation*, i.e. si deux systèmes A et B sont tels que B *est inférieur à* A selon cet ordre, alors cela signifie que dans un certain sens B *implémente* A, ou B *est une implémentation conforme de* A. Par exemple, un critère pouvant être exprimé formellement par une telle relation est la réduction du non-déterminisme, i.e. B est une implémentation conforme de A si, et seulement si, B est une transformation de A par laquelle certains choix (volontairement) non déterministes de A ont été résolus.

Quelques relations d'implémentation basées sur cette idée ont été définies. Ce sont en général - mais pas toujours - des préordres (e.g. en CSP [Brookes 84, Hoare 85], en CCS [de Nicola 84], en LOTOS [Brinksma 87b]). C'est sur la relation de conformité, appelée *conf*, présentée dans [Brinksma 87b] que cet article se concentrera. Dans [Leduc 91b], une autre relation d'implémentation, appelée *conf**, est également proposée et étudiée.

Le concept de relation d'implémentation a aussi été introduit dans d'autres modèles formels (e.g. le wp-calcul [Dijkstra 76], la logique [Chandy 89], les machines à états ou les automates E/S [Lynch 81, Lamport 83, Lam 84, Lynch 87, Abadi 88, Merritt 89], et les systèmes de transitions modaux [Larsen 89]).

Le troisième problème en relation avec l'implémentation des spécifications formelles est celui des tests de conformité. Un cadre général et une méthodologie de test de conformité est étudiée à l'ISO [ISO 9646, Rayner 87]. Le problème crucial qui est actuellement non résolu est celui de la génération d'un ensemble adéquat de scénarios de tests à partir des spécifications formelles [Castanet 87, Favreau 87, Sarikaya 87a, Sarikaya 87b, Bochmann 88, Sabnani 88, Hogrefe 89, Sidhu 89, Vuong 89, Phalippou 91].

En LOTOS, le concept de testeur canonique associé à une spécification a été défini et étudié dans [Brinksma 86, Brinksma 87b, Brinksma 89] et mis en œuvre dans [Alderden 89, Wezeman 89, Wezeman 91]. Le testeur canonique est lui-même une spécification LOTOS qui décrit comment tester les implémentations et déterminer si elles sont conformes ou non à la spécification. Le testeur canonique n'opère aucune sélection sur les séquences de tests; en fait, il est conçu pour tester exhaustivement les implémentations. C'est donc en quelque sorte une borne supérieure théorique sur la façon de tester.

Le testeur canonique est basé sur la relation *conf* proposée pour formaliser la notion de conformité aux tests dans [Brinksma 89]. Cela signifie que la méthode de test proposée, qui se base sur une spécification *S*, rejette toute implémentation *I* qui ne satisfait pas la règle *I conf S*. La technique utilisée consiste à construire, à partir de *S*, un testeur canonique *T (S)* tel que, lorsqu'il est synchronisé avec une implémentation *I*, il atteint **toujours** un état terminal

correct[1] si, et seulement si, *I* *conf* *S*. Si *I* n'est pas conforme à *S*, le testeur canonique *T (S)* **peut** rester bloqué dans un état non terminal où toute interaction avec *I* est impossible.

Cette propriété du testeur canonique montre que le problème du test et le concept de relation d'implémentation (ou de conformité) sont étroitement liés, puisque les définitions de *conf* et de *T* le sont. Ceci est d'ailleurs très logique puisque les tests de conformité ont avant tout comme objectif de tester des implémentations conformes.

Dans cet article, nous montrons que certaines traces du testeur canonique sont inutiles et peuvent être supprimées. Intuitivement, les morceaux de scénarios de test que l'on supprime sont en quelque sorte des tests qui ne peuvent apporter aucune conclusion sur la conformité. Le testeur résultant peut toujours tester exhaustivement les implémentations tout en étant un peu plus simple que le testeur canonique. Les scénarios de test de ce nouveau testeur restent quand même beaucoup trop nombreux, mais nous n'aborderons pas ici le problème d'une sélection adéquate de tests parmi ceux qui sont proposés par le testeur.

Contenu de l'article

Cet article présente d'abord brièvement différentes relations asymétriques existantes [Brinksma 87b], telles que *red*, *ext*, et la relation de conformité, *conf* .

Nous montrons ensuite comment ces relations définissent naturellement des relations d'équivalence. Nous utilisons pour cela une relation générique *imp* pouvant être instanciée par les différentes relations *red*, *ext* et *conf*. L'équivalence associée naturellement à *red* et *ext* est l'équivalence de test, *te* [Brinksma 87b]. L'équivalence associée à *conf* est une équivalence, appelée *conf-eq*, plus faible que *te*.

Le testeur canonique *T (S)* présenté dans [Brinksma 89] est, comme nous l'avons dit, basé sur la relation *conf*. *T (S)* est toutefois défini modulo *te*; ce qui peut sembler peu cohérent étant donné que l'équivalence associée à *conf* est plus faible que *te*.

Partant de cette constatation, nous allons montrer que l'on peut définir *T (S)* modulo *conf-eq*, c'est -à-dire que si *T* est un testeur canonique de la spécification *S*, et que *T'* *conf-eq* *T*, alors *T'* est aussi un testeur canonique : *T'*,

[1] Un état terminal correct de *T (S)* est un état où *T (S)* est supposé s'arrêter, i.e. *T (S)* a été conçu de telle sorte que, après certaines séquences, il puisse se comporter comme *stop*. Quand *T (S)* n'a pas atteint un tel état terminal, mais qu'il reste bloqué dans sa synchronisation avec une implémentation *I*, ceci est considéré comme un état terminal incorrect, i.e. quand le processus composé *I* // *T (S)* est dans un tel état terminal, *T (S)* est encore capable d'exécuter certaines actions.

lorsqu'il est synchronisé avec une implémentation *I*, atteint **toujours** un état terminal correct si, et seulement si, *I* $\underline{conf}$ *S*.

Cette caractérisation modulo $\underline{conf\text{-}eq}$ du testeur canonique a l'avantage de laisser plus de degrés de liberté pour définir *T (S)*. Ce qui nous permet alors de trouver un testeur canonique plus simple, appelé T_m *(S)*, pouvant contenir moins de traces que la spécification *S*; ce qui n'est pas le cas de *T (S)*.

Nous montrerons enfin que pour des spécifications non auto-bloquantes - ce qui est le cas des spécifications se terminant toujours avec succès telles que celles considérées dans [Brinksma 86, Brinksma 89] - T_m et *T* sont $\underline{te}$-équivalents.

2. Définitions et propriétés de quelques relations

En LOTOS, quelques relations asymétriques ont été proposées dans [Brinksma 86, Brinksma 87b] comme candidates au rôle de relation d'implémentation. Nous les rappelons brièvement ci-dessous en utilisant un formalisme de traces et refus similaire à celui de CSP [Hoare 85].

Notations 2.1

L est un alphabet d'actions observables, *i* est l'action interne (i.e. non observable) et δ est l'action de terminaison correcte.

$P \text{—} a \rightarrow P'$ signifie que le processus *P* peut effectuer l'action *a* et, cette action faite, se comporter ultérieurement comme le processus *P'*.

$P \text{—} i^k \rightarrow P'$ signifie que le processus *P* peut effectuer la séquence de k actions internes et, cette séquence faite, se comporter ultérieurement comme le processus *P'*.

$P \text{—} a.b \rightarrow P'$ signifie $\exists\, P''$, *tel que* $P \text{—} a \rightarrow P'' \ \wedge \ P'' \text{—} b \rightarrow P'$.

$P = a \Rightarrow P'$ où $a \in L$, signifie $\exists\, k_0, k_1 \in N$, tels que $P \text{—} i^{k_0}.a.i^{k_1} \rightarrow P'$

$P = a \Rightarrow$ où $a \in L$, signifie que $\exists\, P'$, tel que $P = a \Rightarrow P'$, i.e. *P* peut accepter l'action *a*.

$P = a \neq >$ où $a \in L$, signifie $\neg\, (P = a \Rightarrow)$, i.e. *P* ne peut pas accepter (ou doit refuser) l'action *a*.

$P = \sigma \Rightarrow P'$ signifie que le processus *P* peut effectuer la séquence d'actions observables σ et, cette séquence faite, se comporter ultérieurement comme le processus *P'*. Plus précisément, si $\sigma = a_1 \ldots a_n$ où $a_1, \ldots a_n \in L$, alors
$\exists\, k_0, \ldots k_n \in N$, tels que $P \text{—} i^{k_0}.a_1.i^{k_1}.a_2 \ldots a_n.i^{k_n} \rightarrow P'$

$P = \sigma \Rightarrow$ signifie que $\exists\, P'$, tel que $P = \sigma \Rightarrow P'$

P *after* $\sigma = \{P' \mid P = \sigma \Rightarrow P'\}$,
i.e. l'ensemble de toutes les expressions de comportement (ou états) accessibles à partir de *P* par la séquence σ.

Tr (P) est l'ensemble des traces de *P*, i.e. $\{\sigma \mid P = \sigma \Rightarrow\}$; *Tr (P)* est un sous-ensemble de L^*.

$\sigma_1 \leq \sigma_2$ ssi $\exists\, \sigma_3 \in L^*$, tel que $\sigma_1.\sigma_3 = \sigma_2$ i.e. σ_1 est un préfixe de σ_2.

$\sigma_1 < \sigma_2$ ssi $\exists\, \sigma_3 \in L^+$, tel que $\sigma_1.\sigma_3 = \sigma_2$ i.e. σ_1 est un préfixe strict de σ_2.

Out (P, σ) est l'ensemble des actions observables possibles après la trace σ, i.e. $Out(P, \sigma) = \{a \in L \mid \sigma.a \in Tr(P)\}$.

Ref (P, σ) est l'ensemble des ensembles de refus de P après la trace σ, i.e. $Ref(P, \sigma) = \{X \subseteq L \mid \exists P' \in P \text{ after } \sigma, \text{ tel que } P' = a \not\Rightarrow, \forall a \in X\}$;
Ref (P, σ) est un ensemble d'ensembles et un sous-ensemble de $\wp(L)$, le "power set" de L, i.e. l'ensemble des sous-ensembles de L. Un ensemble $X \subseteq L$ appartient à *Ref (P, σ)* ssi P peut effectuer la trace σ et, cette séquence faite, refuser toute action de l'ensemble X.

Quelques interprétations possibles de la relation de conformité ont été présentées et formalisées dans [Brinksma 86, Brinksma 87b] par trois relations de base : $\underline{conf}$, $\underline{red}$ et $\underline{ext}$, et une relation d'équivalence : $\underline{te}$. Nous en donnons les définitions précises ci-dessous.

Définitions 2.2
Soient les deux processus P_1 and P_2 .

$P_1 \underline{conf} P_2$ *ssi* $\forall \sigma \in Tr(P_2)$, *on a* $Ref(P_1, \sigma) \subseteq Ref(P_2, \sigma)$
ou de façon équivalente,
ssi $\forall \sigma \in Tr(P_2) \cap Tr(P_1)$, *on a* $Ref(P_1, \sigma) \subseteq Ref(P_2, \sigma)$
puisque si $\sigma \in Tr(P_2) - Tr(P_1)$, alors $Ref(P_1, \sigma) = \varnothing$

Intuitivement, $P_1 \underline{conf} P_2$ ssi, placés dans tout environnement dont les traces sont limitées à celles de P_2, P_1 ne peut pas être bloqué quand P_2 ne peut pas l'être. Autrement dit, P_1 se bloque moins souvent que P_2 dans un tel environnement. $\underline{conf}$ a été choisie pour modéliser formellement la notion de conformité aux tests dans [Brinksma 89], et a été appelée pour cette raison la relation de *conformité*.

$P_1 \underline{red} P_2$ ssi (i) $Tr(P_1) \subseteq Tr(P_2)$, et
(ii) $P_1 \underline{conf} P_2$

Intuitivement, si $P_1 \underline{red} P_2$, P_1 a moins de traces que P_2, mais même dans un environnement dont les traces sont limitées à celles de P_1, P_1 se bloque moins souvent. $\underline{red}$ est la relation de *réduction*.

$P_1 \underline{ext} P_2$ ssi (i) $Tr(P_1) \supseteq Tr(P_2)$, et
(ii) $P_1 \underline{conf} P_2$

Intuitivement, si $P_1 \underline{ext} P_2$, P_1 a plus de traces que P_2, mais dans un environnement dont les traces sont limitées à celles de P_2, il se bloque moins souvent. $\underline{ext}$ est la relation d'*extension*.

$\underline{te} = \underline{red} \cap \underline{red}^{-1} = \underline{ext} \cap \underline{ext}^{-1}$ C'est l'équivalence de test.

Proposition 2.3 [Brinksma 86]
(i) $\underline{conf} \supset \underline{red}$
(ii) $\underline{conf} \supset \underline{ext}$
(iii) $\underline{red}$ et $\underline{ext}$ sont des préordres

(iv) *conf* est non transitive

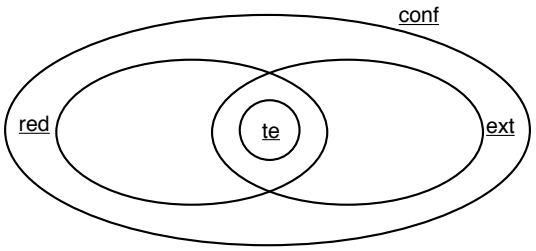

Figure 2.1 : *te, red, ext, conf*

La figure 2.1 illustre les positions relatives de ces différentes relations.

3. Relation d'implémentation et équivalence associée

Dans cette section, nous utilisons une relation générique *imp* afin de modéliser une relation d'implémentation quelconque, à l'exception du fait qu'elle sera réflexive comme nous allons l'expliquer. Rappelons que la relation d'implémentation exprime formellement la notion de conformité à une spécification.

Imp doit être réflexive car la spécification est une implémentation conforme d'elle-même. En conséquence, nous considérerons dans le reste de cet article que *imp* est réflexive.

Par contre, il n'est pas nécessaire que *imp* soit symétrique, puisque l'implémentation et la spécification ne sont pas interchangeables en général.

Un point plus délicat est la transitivité de *imp* : doit-on exiger qu'une implémentation conforme d'une implémentation conforme soit toujours une implémentation conforme ? Si *imp* n'est pas transitive, une implémentation conforme ne peut pas être utilisée comme une spécification intermédiaire; ce qui n'est pas son rôle de toute façon. Nous n'exigerons donc pas de *imp* qu'elle soit transitive. Une étude détaillée de ce problème est présentée dans [Leduc 91a]. La relation *conf* est d'ailleurs un exemple de relation non transitive.

Nous allons montrer comment *imp* induit naturellement une équivalence, appelée *imp–eq*.

Définition 3.1

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

où $\{I \mid I \text{ imp } S\}$ désigne l'ensemble des implémentations I, conformes à S selon la relation *imp*.

Intuitivement, deux spécifications sont équivalentes si, et seulement si, elles déterminent exactement le même ensemble d'implémentations conformes au sens de *imp*.

Il est évident que *imp-eq* est réflexive, symétrique et transitive. *Imp-eq* est donc une équivalence quelle que soit la relation *imp* (même non réflexive).

Si *imp* est considérée comme relation de référence, cette équivalence joue un rôle fondamental en ce sens qu'aucune distinction ne doit être faite entre deux spécifications autorisant les mêmes implémentations conformes.

L'équivalence est déduite naturellement de la relation d'implémentation. Le contraire n'est pas toujours possible.

On pourrait penser que $\underline{imp\text{-}eq}$ est l'équivalence définie par $\underline{imp} \cap \underline{imp}^{-1}$: deux spécifications sont équivalentes si, et seulement si, chacune d'elles est une implémentation conforme de l'autre. Cependant, $\underline{imp} \cap \underline{imp}^{-1}$ n'est pas nécessairement une équivalence.

Proposition 3.2 [Leduc 91a]
$\underline{imp\text{-}eq} \subseteq \underline{imp} \cap \underline{imp}^{-1}$

Propositions 3.3 [Leduc 91a]
$\underline{imp}$ est transitive $\Rightarrow \underline{imp\text{-}eq} = \underline{imp} \cap \underline{imp}^{-1}$

Dans ce cas, deux spécifications sont équivalentes si, et seulement si, chacune d'elles est une implémentation conforme de l'autre.

Ainsi, quand $\underline{imp}$ est transitive, $\underline{imp\text{-}eq}$ peut être définie plus simplement comme suit : $S_1 \ \underline{imp\text{-}eq} \ S_2 \quad ssi \quad S_1 \ \underline{imp} \ S_2 \ \wedge \ S_2 \ \underline{imp} \ S_1$.

4. Relations d'équivalence associées à *red*, *ext* et *conf*

Proposition 4.1 [Brinksma 86]
$\underline{Red\text{-}eq} = \underline{ext\text{-}eq} = \underline{te}$

Définition 4.2
$S_1 \ \underline{conf\text{-}eq} \ S_2 \quad ssi \quad \{I \mid I \ \underline{conf} \ S_1\} = \{I \mid I \ \underline{conf} \ S_2\}$

Cette définition est l'instanciation de la définition 3.1.

Pour étudier la nature de cette équivalence $\underline{conf\text{-}eq}$ associée à $\underline{conf}$, nous allons d'abord rappeler quelques propositions importantes, et nous donnerons ensuite une autre définition de $\underline{conf\text{-}eq}$.

Propositions 4.3 [Leduc 91a]

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

(ii) $\underline{conf\text{-}eq} \supset \underline{te}$ i.e., $\underline{conf\text{-}eq}$ est plus faible que l'équivalence de test

(iii) $\underline{conf\text{-}eq} \cap \underline{trace\text{-}eq} = \underline{conf} \cap \underline{conf}^{-1} \cap \underline{trace\text{-}eq} = \underline{te}$
ou de façon équivalente, $\forall P, Q$, *on a*
$P \ \underline{conf\text{-}eq} \ Q \wedge (Tr(P) = Tr(Q))$
$\Leftrightarrow P \ \underline{conf} \ Q \wedge Q \ \underline{conf} \ P \wedge (Tr(P) = Tr(Q))$
$\Leftrightarrow P \ \underline{te} \ Q$
Pour des processus ayant les mêmes traces, $\underline{conf\text{-}eq}$ et $\underline{te}$ sont identiques.

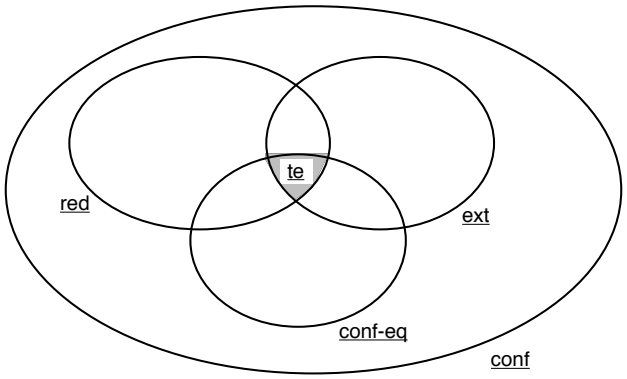

Figure 4.1 : *conf-eq* par rapport aux autres relations

Tous ces résultats sont résumés à la figure 4.1 La partie hachurée est exactement l'équivalence de test. Dans [Leduc 90], des exemples ont été trouvés afin de prouver que les inclusions sont strictes, c'est-à-dire qu'aucune zone de la figure n'est vide.

Nous donnons maintenant une définition plus classique de *conf-eq*.

Proposition 4.4 [Leduc 90]

P conf-eq Q ssi

(a) $P \; conf \; Q \wedge Q \; conf \; P$, *et*

(i.e. $\forall \sigma \in Tr(P) \cap Tr(Q)$, *on a* $Ref(P, \sigma) = Ref(Q, \sigma)$)

(b) $\forall \sigma \in Tr(P) - Tr(Q)$, *on a* $L \in Ref(P, \sigma)$, *et*

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

5. *conf-eq* contre *te*

Si *conf* est la relation d'implémentation de référence, l'équivalence de test n'est pas l'équivalence qui convient car, comme l'indique la proposition 4.3 (ii), elle est plus forte que *conf-eq* : certains processus qui ne sont pas *te-équivalents* définissent pourtant les mêmes ensembles d'implémentations conformes.

Considérons l'exemple suivant où $P \; conf\text{-}eq \; Q$, mais $\neg (P \; te \; Q)$:

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

Si *P* et *Q* sont deux spécifications, elles définissent exactement les mêmes ensembles d'implémentations conformes (au sens de *conf*); en particulier $P \; conf \; Q$ et $Q \; conf \; P$.

Notons cependant que si *P = a; stop* et *Q = (a; stop [] a; b; c; stop),* nous n'avons plus $P \; conf\text{-}eq \; Q$ car *a; b; stop* est une implémentation conforme à *P* mais pas à *Q*. Cela peut s'expliquer intuitivement par le fait qu'une implémentation conforme à *Q*, peut ou non accepter *b* après *a*; mais si elle l'accepte, alors elle ne peut refuser *c* juste après, contrairement aux implémentations conformes à *P*.

6. Simplification du testeur canonique

Dans [Brinksma 87a], le concept de testeur canonique d'une spécification *S* a été introduit et appelé *T (S)*. Ce *T (S)* est la spécification d'un testeur qui,

lorsqu'il est synchronisé avec un processus quelconque *P*, n'est bloqué dans son exécution que si *P* n'est pas conforme à *S*. Il a été prouvé que *T (S)*, tel que défini dans [Brinksma 87a] (voir définition 6.1), existe toujours et est unique modulo l'équivalence de test, c'est-à-dire que si un autre processus *Q* répond aussi aux critères pour être le testeur canonique de *S*, alors *Q* $\underline{te}$ *T (S)*. Cette section a pour but d'étendre ces résultats.

Définition 6.1 (le testeur canonique) [Brinksma 87a]

Soit une spécification *S* , *T (S)* est défini implicitement comme la solution *X* satisfaisant les deux équations suivantes :

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

(ii) $\forall P, P\ \underline{conf}\ S \Leftrightarrow (\forall \sigma \in L^*, on\ a\ (L \in Ref(P||X,\sigma) \Rightarrow L \in Ref(X,\sigma)))$

Cette définition caractérise *T (S)* uniquement par ses traces et refus[1] après chaque trace; ce qui définit *T (S)* modulo $\underline{te}$. (i) indique que le testeur canonique est défini de façon à pouvoir tester toutes les traces de *S*. De plus, (ii) exprime que si le testeur canonique est synchronisé avec une implémentation *P* non conforme à *S*, alors le couple *P* || *T (S)* **peut** effectuer une séquence σ et puis se bloquer après σ, alors que le testeur offre encore des possibilités d'interaction. Inversement, si le testeur canonique est synchronisé avec une implémentation conforme *P* de *S*, il ne peut y avoir de blocage que si le testeur est arrivé dans un état terminal normal (qui est aussi, formellement parlant, un état de blocage).

Exemple de testeur canonique

Soit *S := a; b; exit [] a; c; stop*.

On donne sans preuve que *T (S) := a; (b; exit [] c; stop)* modulo $\underline{te}$.

Les états terminaux corrects de *T (S)* sont atteints après les traces $ab\delta$ et ac.

Considérons les implémentations suivantes à titre d'illustration :

I_1 := *a; b; exit* est une implémentation conforme à *S*, puisque I_1 || *T (S)* ne peut se bloquer qu'après la séquence $ab\delta$ qui conduit aussi à un blocage de *T (S)*

I_2 := *a; b; stop* n'est pas une implémentation conforme à *S*, puisque I_2 || *T (S)* peut se bloquer après la séquence *ab* qui ne conduit pas à un blocage de *T (S)*: *T (S)* offre δ.

---

[1] Cette caractérisation d'un processus par ses traces et ses refus a d'abord été proposée en CSP par Hoare dans [Hoare 85] et en LOTOS par Brinksma dans [Brinksma 87a] où ce modèle est appelé "Failure Tree". Un modèle semblable mais étendu est également développé dans [Leduc 90]. Il faut noter que les traces et les refus satisfont certaines propriétés générales présentées dans les travaux cités (e.g. tout préfixe d'une trace est une trace; si X $\in$ Ref (P, σ) et si X' $\subseteq$ X, alors X' $\in$ Ref (P, σ)). Ceci nous oblige, lorsque nous travaillons dans ce modèle, à vérifier que les ensembles de traces et de refus sont bien formés sous peine de traiter des processus irréels. Ces vérifications sont effectuées implicitement dans ce papier.

$I_3 := S$ est une implémentation conforme à S, puisque $I_3 \,//\, T(S)$ ne peut se bloquer qu'après les séquences $ab\delta$ ou ac qui conduisent aussi à un blocage de $T(S)$

$I_4 := a; exit$ n'est pas une implémentation conforme à S, puisque $I_4 \,//\, T(S)$ peut se bloquer après la séquence a qui ne conduit pas à un blocage de $T(S)$: $T(S)$ offre les actions b et c.

Cette définition ne requiert pas l'introduction d'une action spéciale, ω, pour rapporter le succès d'un test, comme dans [de Nicola 84]. L'introduction de cette action spéciale peut toutefois toujours être réalisée, mais nous considérons que ce n'est pas requis car cela relève purement de considérations pratiques sur l'implémentation d'un testeur, e.g. il suffit d'ajouter une action spéciale ω dans $T(S)$ avant qu'il n'atteigne *stop*. Remarquons que δ ne peut pas être utilisée comme action ω car la composition parallèle impose toujours une synchronisation sur l'action δ; ce qui est trop contraignant.

Proposition 6.2 [Brinksma 87a,Brinksma 89,Leduc 90,Leduc 91c]
Soit une spécification S , $T(S)$ est défini par
(i) $Tr(T(S)) = Tr(S)$
(ii) $\forall\ \sigma \in Tr(S),\ \forall A \subseteq L$, on a

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

Cette proposition donne une méthode de construction de $T(S)$ à partir de S. Informellement, $T(S)$ peut effectuer les mêmes séquences d'actions que S; cependant si, après une séquence σ, S peut refuser toute interaction de l'ensemble $A \subset L$, alors $T(S)$ doit accepter au moins une interaction de l'ensemble $L - A$ après la séquence σ. Dans le cas particulier où S peut être bloquée après la séquence σ (i.e. elle peut refuser L, c'est-à-dire qu'elle peut accéder à un état terminal), $T(S)$ doit aussi avoir un état terminal accessible après la séquence σ.

Propositions 6.3 [Brinksma 87a]
(i) $\forall \sigma \in L^*$, on a $L \in Ref(S, \sigma) \quad \Leftrightarrow \quad L \in Ref(T(S), \sigma)$
(ii) $\forall S$, $T(S)$ existe et est unique modulo l'équivalence de test, $\underline{te}$.
(iii) $\forall S$, on a $T(T(S)) \ \underline{te}\ S$

La proposition 6.3 (iii) est le meilleur résultat que l'on puisse atteindre puisque $T(S)$ a été défini modulo $\underline{te}$.

Avant de généraliser les résultats de [Brinksma 87a, Brinksma 89], nous avons besoin de quelques propositions supplémentaires.

Propositions 6.4 [Leduc 91c]
(i) $\forall P, Q$, on a $P \ \underline{conf}\ Q \wedge Q \ \underline{conf}\ P \Leftrightarrow T(P) \ \underline{conf}\ T(Q) \wedge T(Q) \ \underline{conf}\ T(P)$
(ii) $\forall P, Q$, on a $P \ \underline{conf\text{-}eq}\ Q \Leftrightarrow T(P) \ \underline{conf\text{-}eq}\ T(Q)$
(iii) $\forall P, Q$, on a $P \ \underline{te}\ Q \Leftrightarrow T(P) \ \underline{te}\ T(Q)$

La proposition 6.4 (ii) est une généralisation de 6.4 (iii), et est une proposition clé de cet article. En effet, on sait que si deux spécifications S_1 et S_2 sont

conf-équivalentes (i.e. S_1 *conf-eq* S_2), les testeurs canoniques de S_1 et S_2 sont interchangeables, puisqu'ils doivent donner les mêmes verdicts de conformité pour les mêmes implémentations. Grâce à la proposition 6.4 (ii), on voit que c'est *conf-eq*, et non *te*, qui caractérise cette interchangeabilité. Dès lors, c'est modulo *conf-eq*, et non modulo *te*, qu'il faut définir *T (S)*. Nous allons voir que cela va nous permettre de trouver un testeur canonique T_m *(S)* plus simple que *T (S)*, et unique modulo *conf-eq*.

La proposition suivante est la clé de cet article. Elle prouve que toutes les solutions *X* de l'équation 6.1 (ii) sont telles que *X conf-eq T (S)*, et inversement.

Proposition 6.5 [Leduc 91c]
Soient *S* une spécification, et *T (S)* son testeur canonique.
$\forall X$, *(X conf-eq T (S)* $\Leftrightarrow$
$(\forall P, P \text{ conf } S \Leftrightarrow (\forall \sigma \in L^*, \text{on a } (L \in Ref(P||X,\sigma) \Rightarrow L \in Ref(X,\sigma)))))$

Si nous reprenons la définition 6.1 du testeur canonique, nous voyons que 6.1 (ii) définit *T (S)* modulo *conf-eq*, et que 6.1 (i) ajoute une contrainte qui fixe *T (S)* modulo *te*. En effet, *conf-eq* $\cap$ *trace-eq* = *te* (cf. prop. 4.3 (iii)).

Cette contrainte 6.1 (i) sur les traces est tout à fait arbitraire et ne se justifie ni en pratique, ni en théorie. Seule la deuxième contrainte 6.1 (ii) a un sens pour le test des implémentations. Elle permet à elle seule de définir un testeur qui fera dans tous les cas la distinction entre les implémentations conformes et les autres. Le fait d'enlever la contrainte 6.1 (i) permet en plus, comme nous allons le voir, de trouver un testeur canonique qui a en général moins de traces que *T (S)*.

Proposition 6.6
Soient *S* une spécification et *T(S)* son testeur canonique,
X conf-eq T (S) $\Leftrightarrow$
(i) $\forall \sigma \in Tr(X) \cap Tr(S)$, on a $Ref(X, \sigma) = Ref(T(S), \sigma)$, et
(ii) $\forall \sigma \in Tr(X) - Tr(S)$, on a $L \in Ref(X, \sigma)$, et
(iii) $\forall \sigma \in Tr(S) - Tr(X)$, on a $L \in Ref(S, \sigma)$.

Ceci découle directement des définitions 4.4 et 6.1(i), et de la prop. 6.3(i).

Il est évident que tout ensemble de processus peut être ordonné selon l'ordre partiel "a ses traces incluses dans les traces de". En particulier, l'ensemble des solutions *X* de l'équation 6.1 (ii) peut être ordonné selon cette relation. Par les propositions 6.5 et 6.6, cet ensemble a un élément minimal *X* qui possède moins de traces que tous les autres. Nous allons formaliser cela en détail.

Définition 6.7
Min (S) est la spécification obtenue à partir de *S* en supprimant de *S* un ensemble adéquat de traces selon la règle suivante :
si $\forall \sigma' \geq \sigma$, *on a* $L \in Ref(S, \sigma')$, alors enlever toute séquence $\sigma' > \sigma$.

On a donc directement

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

(ii) $\forall \sigma \in Tr\ (S) - Tr\ (Min\ (S))$, *on a* $L \in Ref\ (S, \sigma)$

Proposition 6.8

Min (S) est la solution *X* de l'équation "*X* <u>*conf-eq*</u> *S* " ayant le moins de traces. Ceci découle directement des définitions 6.7 et 4.4 de <u>*conf-eq*</u>.

Définition 6.9

Le testeur canonique minimum de *S*, appelé $T_m(S)$, est défini par *Min (T(S))*.

Propositions 6.10 [Leduc 91c]

(i) $T_m\ (S)$ <u>*conf-eq*</u> $T\ (S)$

(ii) $T_m\ (S)$ <u>*te*</u> $T\ (Min\ (S))$

$T_m\ (S)$ est donc déduit de *S* en deux étapes : 1) Construire *S' := Min (S)*,
2) Construire $T_m\ (S) := T\ (S')$.

$T_m\ (S)$ a l'avantage d'être plus simple que *T (S)*, comme le montre l'exemple suivant.

Exemple

Soient *P* et *Q* les processus suivants :

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

Nous pouvons prouver que *P* <u>*conf-eq*</u> *Q*, mais que ¬ *(P* <u>*te*</u> *Q)*.

Les testeurs canoniques sont définis (modulo <u>*te*</u>) comme suit :

T (P) := i; a; stop [] i; b; stop

T (Q) := i; a; (b; stop [] i; stop) [] i; b; stop

$T_m\ (P) = T\ (P)$

$T_m\ (Q) = T\ (P)$ est plus simple que *T (Q)*.

Un autre exemple très semblable où T_m et *T* sont <u>*te*</u>-équivalents

Supposons que *Q* soit défini comme suit :

Q = (a; stop [] b; stop [] a; b; c; stop).

Les testeurs canoniques sont définis (modulo <u>*te*</u>) comme suit :

T (Q) := i; a; (b; c; stop [] i; stop) [] i; b; stop

$T_m\ (Q)$ *:= i; a; (b; c; stop [] i; stop) [] i; b; stop*

Considérons les trois implémentations suivantes :

I_1 *:= b; stop [] a; stop* est une implémentation conforme à *Q*, puisque I_1 *|| T (Q)* peut seulement se bloquer après les séquences *a* ou *b* qui conduisent aussi à des blocages de *T (Q)*

I_2 *:= b; stop [] a; b; stop* n'est pas une implémentation conforme à *Q*, puisque si *T (Q)* choisit d'exécuter *b* après *a*, I_2 *|| T (Q)* se bloque alors que *T (Q)* offre l'action *c*

I_3 *:= b; stop [] a; b; c; stop* est une implémentation conforme à *Q*, puisque I_3 *|| T (Q)* peut seulement se bloquer après les séquences *a*, *abc* ou *b* qui conduisent aussi à des blocages de *T (Q)*

La raison pour laquelle $T(Q)$ ne peut pas être simplifié dans ce cas peut être comprise intuitivement sur les trois implémentations ci-dessus : si la branche *i; stop* était supprimée dans $T(Q)$, alors I_1 serait considérée comme une implémentation non conforme; si la branche *b; c; stop* était supprimée dans $T(Q)$, alors I_2 serait considérée comme une implémentation conforme.

Le dernier exemple ci-dessus illustre aussi que l'on **ne peut pas** changer la règle de la définition 6.7 pour écrire : $\forall \sigma$ *tel que* $L \in Ref(S, \sigma)$, alors supprimer toute séquence $\sigma' > \sigma$.

La proposition suivante est le résultat principal de cet article. Il prouve que si S est une spécification, $T_m(S)$ est le processus minimal (en termes de traces) qui peut tester (exhaustivement) toute implémentation et donner un verdict correct de conformité.

Proposition 6.11 [Leduc 91c]
Soit une spécification S.
$T_m(S)$ est la solution minimale X (c'est-à-dire qui a moins de traces que toutes les autres) satisfaisant l'équation :

$\forall P, P \underline{conf} S \Leftrightarrow \forall \sigma \in L^*$, *on a* $(L \in Ref(P \,||\, X, \sigma) \Rightarrow L \in Ref(X, \sigma))$.

Notons que si on se limite aux spécifications S ne pouvant se terminer qu'avec succès (ce qui est le cas des spécifications traitées dans [Brinksma 86, Brinksma 89]), nous allons montrer que $T_m(S)$ $\underline{te}$ $T(S)$; ce qui signifie que $T(S)$ est le testeur canonique minimum dans ce cas. Ces résultats sont formalisés ci-dessous.

Définitions 6.12
(i) S est une spécification ne pouvant se terminer qu'avec succès si, et seulement si, $\forall \sigma \in Tr(S)$, *on a* $(L \in Ref(S, \sigma) \Leftrightarrow \sigma = \sigma'.\delta)$
où δ est l'action spéciale indiquant en LOTOS une terminaison correcte.
(ii) Une spécification S est *auto-bloquante* si, et seulement si,
$\exists \sigma \in Tr(S)$, *telle que* $L \in Ref(S, \sigma) \wedge Out(S,\sigma) \neq \varnothing$

Informellement, une spécification ne pouvant se terminer qu'avec succès est une spécification qui ne peut refuser L qu'après l'action δ. Une spécification est auto-bloquante si, et seulement si, elle peut refuser L après une certaine trace σ alors qu'elle peut accepter au moins une action après σ.

Proposition 6.13 [Leduc 91c]
Si S est une spécification ne pouvant se terminer qu'avec succès, alors S n'est pas auto-bloquante.

On pourrait penser que si S_1 et S_2 ne sont pas des spécifications auto-bloquantes, alors S_1 $\underline{te}$ $S_2 \Leftrightarrow S_1$ $\underline{conf\text{-}eq}$ S_2, mais ceci est faux comme l'illustre le contre-exemple suivant.

Soient S_1 := *a; stop* et S_2 := *i; a; stop [] b; stop*.

Il est clair que S_1 et S_2 ne sont pas auto-bloquantes, et que S_1 $\underline{conf\text{-}eq}$ S_2, mais $\neg$ (S_1 $\underline{te}$ S_2).

On pourrait alors penser que si S_1 et S_2 sont des spécifications ne pouvant se terminer qu'avec succès, alors S_1 $\underline{te}$ S_2 $\Leftrightarrow$ S_1 $\underline{conf\text{-}eq}$ S_2, mais ceci est également faux comme le montre le contre-exemple suivant.

Soient S_1 := *a; exit* et S_2 := *i; a; exit [] exit*

Il est clair que S_1 et S_2 sont des spécifications ne pouvant se terminer qu'avec succès, et que S_1 $\underline{conf\text{-}eq}$ S_2, mais $\neg$ (S_1 $\underline{te}$ S_2).

Ces résultats nous empêchent d'obtenir directement la condition sous laquelle T_m et T sont $\underline{te}$-équivalents. Néanmoins, ce résultat final est présenté dans la prochaine proposition.

Proposition 6.14 [Leduc 91c]
Si S n'est pas auto-bloquante, alors $T_m(S)$ $\underline{te}$ $T(S)$.

Remarquons que cette condition n'est bien qu'une condition suffisante. Le second exemple suivant les propositions 6.10 montre qu'elle n'est pas nécessaire en général : Q est auto-bloquante, mais $T_m(Q)$ $\underline{te}$ $T(Q)$.

7. Conclusion

Nous avons insisté sur le rôle d'une relation d'implémentation pour formaliser le lien entre une implémentation conforme et une spécification. La définition d'une relation d'implémentation (ou de conformité) est un préalable essentiel à la définition de toute forme de testeur (ou ensemble de scénarios de tests) devant permettre de certifier la conformité d'une implémentation.

Partant d'une relation d'implémentation, nous avons montré qu'il est toujours possible d'en déduire une relation d'équivalence associée, même si la relation d'implémentation n'est pas transitive.

Nous avons ensuite adopté la relation $\underline{conf}$ comme relation d'implémentation, et nous avons montré que la relation d'équivalence associée est l'équivalence $\underline{conf\text{-}eq}$ plus faible que $\underline{te}$.

Ces résultats nous ont permis de montrer qu'il est possible de définir un testeur canonique plus simple que celui défini dans [Brinksma 87a, Brinksma 89], en ce sens que, contrairement à celui-ci, il peut avoir moins de traces que la spécification.

Dans le cas de spécifications non auto-bloquantes - ce qui est le cas des spécifications se terminant toujours avec succès telles que celles considérées dans [Brinksma 86, Brinksma 89] - les deux testeurs T_m et T sont $\underline{te}$-équivalents.

Finalement, soulignons que la méthodologie utilisée dans cet article, qui consiste à définir le testeur canonique modulo une équivalence associée à une relation d'implémentation adéquate, est très importante et nouvelle. Elle peut être utilisée de façon similaire pour d'autres relations d'implémentation. Plutôt que de déduire de rien un nouveau testeur canonique basé uniquement sur la définition 6.1 et la proposition 6.5, nous avons choisi de nous appuyer sur la

théorie du test de E. Brinksma et de montrer comment celle-ci peut être légèrement adaptée. Ceci a donné lieu à une définition de T_m en tant que simplification de T (cf. définition 6.9 associée à la proposition 6.11).

Références

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

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

[Bochmann 88] G. Bochmann, R. Dssouli, B. Sarikaya, *Méthodes de test de protocoles : Architectures et sélection de tests,* Congrès CFIP'88 (Eyrolles) 337-363.

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

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

[Brinksma 87a] 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.

[Brinksma 87b] 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) 349-360.

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

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

[Castanet 87] R. Castanet, R. Sijelmassi, *Génération automatique de séquences pour la préparation au test réparti de protocoles,* Deuxième colloque C^3, Angoulème, mai 1987.

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

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

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

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

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

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

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

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

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

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

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

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

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

[Leduc 90] G. Leduc, *On the role of implementation relations in the design of distributed systems,* Thèse d'Agrégation de l'enseignement supérieur, Université de Liège, Dept. Systèmes et Automatique, B28, Liège, Belgique, 1990.

[Leduc 91a] G. Leduc, *A Framework based on implementation relations for implementing LOTOS specifications,* à paraître dans : Computer Networks & ISDN Systems, 1991.

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

[Leduc 91c] G. Leduc, *Conformance relation, associated equivalence, and new canonical tester in LOTOS,* A paraître dans : B. Pehrson, J. Parrow, B. Johnson, eds., 11th Symposium on Protocol Specification, Testing and Verification, Stockholm, Sweden, 1991.

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

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

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

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

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

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

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

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

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

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

[Sarikaya 87b] B. Sarikaya, G. Bochmann, E. Cerny, *A Test Design Methodology for Protocol Testing,* IEEE Tr. on Soft. Eng., Vol. 13, No. 5, 1987, 518-531.

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

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

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

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

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

Guy Leduc est ingénieur civil électricien (électronique) (1983), et agrégé de l'enseignement supérieur en Sciences Appliquées (1991) de l'Université de Liège. Depuis 1983, il travaille pour le Fonds National belge de la Recherche Scientifique (F.N.R.S.) à l'Université de Liège, dans le service du Professeur A. Danthine. Aspirant du F.N.R.S. de 1983 à 1988, chargé de recherches de 1988 à 1990, il est chercheur qualifié depuis 1990. Ses activités et intérêts de recherche incluent les techniques formelles de description (FDT) et les réseaux de communication. Depuis 1985, ses recherches se sont focalisées sur LOTOS : en dehors de sa thèse, il a appliqué cette FDT dans divers projets européens, tels que ESPRIT/BWN sur la conception et la réalisation d'un réseau à large bande sur site étendu, et ESPRIT/APHRODITE sur le système d'exploitation distribué CHORUS. Il applique actuellement LOTOS dans le cadre du projet ESPRIT II/OSI95 sur la conception de protocoles à hautes performances.