Incremental Construction Approach for Distributed System Specifications[*]

Ferhat Khendek and Gregor v. Bochmann

Département d'informatique et de recherche opérationnelle

Université de Montréal

C. P. 6128, Succ. A, Montréal, Que H3C 3J7, Canada

E-mail: {khendek, bochmann}@iro.umontreal.ca

Abstract

In this paper, we propose an incremental construction approach for distributed system specifications. These specifications are structured as a parallel composition of subsystem specifications. The approach consists of merging two specifications S_{old} and S_{added} into a new specification S_{new}, such that S_{new} extends S_{old} and S_{new} extends S_{added}. Moreover, in the case of cyclic behaviors, S_{new} offers a choice between behaviors of S_{old} and behaviors of S_{added}, in a recursive manner. The derived specification S_{new} has the same internal structure as S_{old}. Our approach is described in terms of Labelled Transition Systems, and it is applicable for many specification languages.

1 Introduction

The design of a distributed system goes through many phases. The initial phase allows the capturing of functional requirements in a specification with a high level of abstraction. This specification describes the functionalities of the system, but not how to realize them. In the next phases, it is refined into specifications with a lower level of abstraction where some design decisions are taken and a structure is chosen. The specification obtained after each step should remain correct with respect to the initial specification. The service specification and protocol specification for a given OSI layer are typical examples of two different levels of abstraction [Viss 85].

The step-wise refinement approach allows the methodical production of a specification with a low level of abstraction from a specification with a high level of abstraction. The distributed system specification task, however, still remain very complex, particularly when many functions have to be handled simultaneously. A complementary approach to deal with this complexity is the divide-and-conquer methodology. It consists of building specifications for the different features of the required system independently and of combining them to obtain the desired specification. From another point

[*] This research was supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the Government of Canada and by an IBM research fellowship.

of view, this approach allows the enrichment of a system specification by adding new behaviors required by the user, such as adding a new functionality to a given telecommunication system.

The combination should preserve the semantics properties of each single specification. For instance, the addition of a new function to a telephone system specification should not disturb the semantics properties of the telephone system specification and the semantics properties of the new function. In the context of distributed systems, preserving semantic properties may, for instance, mean that the combined specification exhibits at least the behaviors of the original ones without introducing additional failures for these behaviors. This is captured by the formal relation between specifications, called extension, introduced in [Brin 86]. Informally, a specification S_2 extends a specification S_1, if and only if, S_2 allows any sequence of actions that S_1 allows, and S_2 can only refuse what S_1 can refuse, after a given sequence of actions allowed by S_1.

Two specifications S_{old} and S_{added} may be combined in different ways depending on the user requirements. In this paper, we assume that S_{old} and S_{added} have to be combined as alternative behaviors. We propose an incremental specification approach, which consists of merging two specifications S_{old} and S_{added} into a specification S_{new}, such that S_{new} extends S_{old} and S_{new} extends S_{added}. Moreover, in the case of cyclic traces, S_{new} offers a choice between behaviors of S_{old} and S_{added}, in a recursive manner. We consider distributed system specifications, which may consist of a parallel combination of subsystem specifications. The incremental specification approach preserves such structure. Therefore, the designer does not have to redesign it. The approach for merging structured specifications described in this paper, is based on the approach for merging monolithic specifications described in [Khen 92].

The remainder of the paper is structured as follows. Section 2 introduces the labelled transition systems model [Kell 76] and some definitions used in this paper. In Section 3, we summarize the principle and properties of the approach for merging monolithic specifications. In Section 4, our approach for merging structured specifications is described. In Section 5, it is compared to related ones. In Section 6, we conclude.

2 Labelled Transition Systems

We view the specification of a distributed system and its subsystems as processes, which are expressed by labelled transition systems (LTS for short). In this section, we introduce the LTS model [Kell 76] and some definitions, such as the definition of a cyclic trace, a minimal cyclic trace, and the definition of the extension relation [Brin 86].

2.1 Definitions

An LTS is a graph in which nodes represent states, and edges, also called transitions, represent state changes, labelled by actions occurring during the change of state. These actions may be observable or not.

Definition 2.1 [Kell 76]
An LTS TS is a quadruple $<S, L, T, S_o>$, where
- S is a (countable) nonempty set of states;
- L is a (countable) set of observable actions.
- $T: S \times L \cup \{\tau\} \to S$ is a transition relation, where a transition from a state S_i to state S_j by an action μ ($\mu \in L \cup \{\tau\}$) is denoted by $S_i{-}\mu{\rightarrow}S_j$.
  τ represents the internal, nonobservable action.
- S_o is the initial state of TS.

A finite LTS (FLTS for short) is an LTS in which S and L are finite. In the remainder of this paper, we may refer to an LTS by its initial state and vice versa. We may also write act(TS), instead of L, to denote the set of observable actions of TS. Some notations for LTSs are summarized in Table 1.

$P{-}\mu_1...\mu_n{\rightarrow}Q$	$\exists P_i$ ($0 \le i \le n$) such that $P = P_0{-}\mu_1{\rightarrow}P_1...P_{n-1}{-}\mu_n{\rightarrow}P_n = Q$
$P{-}\mu_1... \mu_n{\rightarrow}$	$\exists$ Q such that $P{-}\mu_1...\mu_n{\rightarrow}Q$
$P{=}\varepsilon{\Rightarrow}Q$	$P \equiv Q$ or $\exists n \ge 1$ $P{-}\tau^n{\rightarrow}Q$
$P{=}a{\Rightarrow}Q$	$\exists P_1, P_2$ such that $P{=}\varepsilon{\Rightarrow}P_1{-}a{\rightarrow} P_2{=}\varepsilon{\Rightarrow}Q$
$P{=}a_1... a_n{\Rightarrow}Q$	$\exists P_i$ ($0 \le i \le n$) such that $P = P_0{=}a_1{\Rightarrow}P_1{=}a_1{\Rightarrow}..a_n{\Rightarrow}P_n = Q$
$P{=}\sigma{\Rightarrow}$	$\exists$Q such that $P{=}\sigma{\Rightarrow}Q$
$P{\neq}\sigma{\Rightarrow}$	not ($P{=}\sigma{\Rightarrow}$)
Tr(P)	$\{\sigma \in L^* \mid P{=}\sigma{\Rightarrow}\}$
out(P, σ)	$\{a \in L \mid \sigma.a \in Tr(P)\}$
initials(P)	out(P, ε)
P after σ	$\{Q \mid P{=}\sigma{\Rightarrow}Q\}$
Acc(P, σ)	$\{X \mid \exists Q \in$ (P after σ), such that initials(Q) $\subseteq$ X [illegible]

where μ, $\mu_i \in L \cup \{\tau\}$; $a, a_i \in L$; P, Q, P_i, Q_i represent states; ε represents the empty trace;
$\sigma = a_1.a_2... a_n$, where "." denotes the concatenation of actions or sequence of actions (traces).

Table 1. LTS notations

A trace, of a given state S_i in the LTS TS, is a sequence of actions that TS can perform starting from state S_i. A cyclic trace in TS is a trace of the initial state S_o that reaches only the initial state S_o and the states that can be reached by the empty trace from S_o. In other words, a cyclic trace always brings back TS to its initial state. TS may then move to an other state by the nonobservable action τ. A minimal cyclic trace is a cyclic trace that is not prefixed by a nonempty cyclic trace.

Definitio[redacted] 2.2 (Cy[redacted]ic Trace)[redacted]
Given an [redacted]S TS = <[redacted], T, S_o>, [redacted] trace σ is cyclic, iff
(S_o after [redacted] = {S_o [redacted] S', [redacted]S' is such that ∀ S_i ∈ S', S_o=ε⇒S_i.

Defi[redacted]inimal Cyclic Trace)
Given an [redacted]S TS[redacted] <S, L, T, S_o>, σ is a minimal cyclic trace, iff
σ is a cycl[redacted] trace[redacted]nd
[redacted]≠ ε) such that σ = σ'.σ" and σ' is cyclic trace in TS.

2.2 Operations on Labelled Transition Systems

The specification of a distributed system may be considered as a composition of its subsystem specifications. Among the possible compositions, the parallel composition operator and the action hiding operator are of special interest in this paper. The parallel composition operator (B1 $|_{\{a1, ..., an\}}$ B2) allows one to express the parallel execution of the behaviors B1 and B2. B1 and B2 synchronize on actions in {a1, ..., an} and interleave with respect to other actions. The hiding operator allows the hiding of actions, which then [redacted]ll be considered internal actions. We write B\A to denote the hiding of the actions in A in the [redacted] The inference rules for these operators are as follows (adapted from [ISO 8807]).

Parallel composition: B[redacted]
If B1–a→B1' and a {a[redacted]n B1 $|_{\{a1, ..., an\}}$ B2–a→B1' $|_{\{a1, ..., an\}}$ B2,
If B2–a→B2' and a {a1, ..[redacted]}, then B1 $|_{\{a1, ..., an\}}$ B2–a→B1 $|_{\{a1, ..., an\}}$ B2',
If B2–a→B2' and B1[redacted]nd a ∈ {a1, ..., an}, then B1 $|_{\{a1, ..., an\}}$ B2–a→B1' $|_{\{a1, ..., an\}}$ B2'.

Hiding operator: B\{a1[redacted]
If B–a→B' and a {a1, ...,[redacted]n}, then B\{a1, ..., am}–a→B'\{a1, ..., am},
If B–a→B' and a ∈ {a1, ..., am}, then B\{a1, ..., am}–τ→B'\{a1, ..., am}.

2.3 The extension relation

Intuitively, different LTSs may describe the same observable behavior. Therefore different equivalence relations have been defined based on the notion of observable behavior. They range from the relatively coarse trace equivalence to the much finer strong bisimulation equivalence [DeNi 87]. However, for our considerations, one does not need equivalence relations, but rather ordering relationships. Among them, we note the reduction and extension relation as defined in [Brin 86]. These relations may serve different purposes during the specification life cycle. The extension relation is most appropriate for our purpose of compatible enrichment of specifications. Informally, S2 extends S_1, if an [redacted] llows any sequence of actions that S_1 allows, and S_2 can only refuse what S_1 c [redacted] actions allowed by S_1.

Definition [redacted]

S_2 extends [redacted] 1), iff

(a) $Tr(S_1)$ T [redacted]

(b) $\forall\ \sigma \in Tr(S_1)$, $\forall$ A [redacted]

if [redacted] $\Rightarrow S_2'$ and $S_2' \not= a \Rightarrow$, $\forall\ a \in A$,

then [redacted] $\sigma \Rightarrow S_1'$ and $S_1' \not= a \Rightarrow$, $\forall\ a \in A$.

3 Merging monolithic specifications

In this section, we consider monolithic specifications [Viss 88]. A monolithic specification has no internal structure and is defined directly in terms of some allowed ordering of actions. A monolithic specification is represented by a single LTS.

Given two LTSs, S_1 and S_2, we want to construct systematically an LTS S_3, such that S_3 extends S_1, and S_3 extends S_2. Moreover, in the case of cyclic traces, S_3 should offer a choice between behaviors of S_1 and behaviors of S_2, in a recursive manner. Note that the usual choice operators defined for LOTOS [ISO 8807] and CCS [Miln 89] for instance, do not allow such combination of specifications as shown in Figure 1.

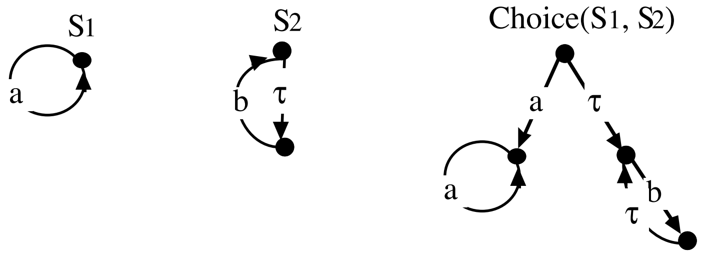

Figure 1. LOTOS, CCS choice operator

We assume that the LTSs are finite. Our FLTSs merging algorithm, called Merge, uses an intermediate representation, the Acceptance Graphs (AGs for short).

Definition 3.1

An AG G is 5-tuple <Sg, L, Ac, Tg, Sg_0>, where

- Sg is a (countable) nonempty set of states.
- L is a (countable) nonempty set of events.

- Ac: Sg $\rightarrow$ *P*(*P*(L)) is a mapping from Sg to sets of subsets of L. Ac(Sg_i) is called the acceptance set of Sg_i.
- Tg: Sg x L $\rightarrow$ Sg is a transition function, where a transition from state Sg_i to state Sg_j by an action a (a $\in$ L) is denoted by Sg_i–a$\rightarrow$$Sg_j$.
- Sg_o is the initial state of G.

The mappings Ac and Tg should satisfy the consistency constraints defined for Acceptance Trees in [Henn 85]. A finite AG (FAG for short) is an AG in which Sg and L are finite. The LTS notations in Table 1 remain valid for the AGs. A cyclic trace for an AG G = <Sg, L, Ac, Tg, Sg_o>, is a trace of the initial state Sg_o that reaches the initial state Sg_o. As for an LTS, a minimal cyclic trace for an AG is a cyclic trace that is not prefixed by a nonempty cyclic trace. In the following, we define a relation AGR between AGs and LTSs.

Definition 3.2

Given an AG G = <Sg, L, Ac, Tg, Sg_o> and an LTS S = <St, L, T, S_o>, we note G = AGR(S), iff

- Tr(G) = Tr(S),
- $\forall$ $\sigma \in$ Tr(S), if Sg_o=$\sigma$$\Rightarrow$$Sg_i$, then Ac($Sg_i$) = Acc($S_o$, σ),
- Any minimal cyclic trace in S is a minimal cyclic trace in G, and
- Any minimal cyclic trace in G is a minimal cyclic trace in S.

Given two FLTSs S1 = <St1, L1, T1, $S1_o$> and S2 = <St2, L2, T2, $S2_o$>, the algorithm Merge consists, first, of transforming the FLTSs S1 and S2 into FAGs G1=<Sg1, L1, Ac1, Tg1, $Sg1_o$> and G2= <Sg2, L2, Ac2, Tg2, $Sg2_o$>, respectively, such that Sg1 $\cap$ Sg2 = Ø and G1 = AGR(S1) and G2 = AGR(S2). The FAGs G1 and G2 are then merged by an FAG merging algorithm into the FAG G3 = <Sg3, L1 $\cup$ L2, Ac3, Tg3, <$Sg1_o$, $Sg2_o$>>, which is transformed back to an FLTS S3 such that G3 = AGR(S3).

The algorithm for the transformation of an FLTS to an FAG is similar to the "subset construction" algorithm defined in [Aho 79]. The transformation of an FAG to an FLTS, in the last step, is the converse transformation. This transformation eliminates the information redundancy concerning the failure possibilities. The FLTS generated by this transformation is the canonical representative of a class of testing equivalent LTSs with the same set of minimal cyclic traces. In the following, we describe, informally, the FAG merging algorithm. A more formal treatment of these issues can be found in [Khen 92].

A state Sg_i in Sg3 may be either a tuple <$Sg1_i$, $Sg2_j$> consisting of state $Sg1_i$ from Sg1 and $Sg2_j$ from Sg2 (as for the initial state <$Sg1_o$, $Sg2_o$>), or a simple state $Sg1_i$ from Sg1, or a simple state $Sg2_j$ from

$Sg2$. These states and the transitions which reach them are added by the FAG merging algorithm step by step into $Sg3$ and $Tg3$, respectively, except for the two initial states $Sg1_o$ and $Sg2_o$, each of these is replaced by the initial state $<Sg1_o, Sg2_o>$ of $G3$.

Initially, $Sg3$ contains only the initial state $<Sg1_o, Sg2_o>$. The definition of the transitions from state $<Sg1_i, Sg2_j>$ in $Sg3$ depends on the transitions from $Sg1_i$ in $Sg1$ and from $Sg2_j$ in $Sg2$. For instance, for a given state $<Sg1_i, Sg2_j>$, if there is a transition $Sg1_i – a \rightarrow Sg1_k$ in $Tg1$ and a transition $Sg2_j – a \rightarrow Sg2_m$ in $Tg2$, then the state $<Sg1_k, Sg2_m>$ is added into $Sg3$ and the two transitions are combined into one transition $<Sg1_i, Sg2_j> – a \rightarrow <Sg1_k, Sg2_m>$ in $Tg3$. This is the situation when $G1$ and $G2$ have a common trace from their initial state to $Sg1_k$ and $Sg2_m$, respectively.

Another case of this construction, if for a given state $<Sg1_i, Sg2_j>$, there exists a transition $Sg1_i – a \rightarrow Sg1_k$ in $Tg1$, with $Sg1_k \neq Sg1_o$, but there is no transition labelled by a from $Sg2_j$ in $Tg2$, then the state $Sg1_k$ is added into $Sg3$ and the transition $Sg1_i – a \rightarrow Sg1_k$ in $Tg1$ yields the transition $<Sg1_i, Sg2_j> – a \rightarrow Sg1_k$ in $Tg3$. Reciprocally, if there exists a transition $Sg2_j – a \rightarrow Sg2_m$ in $Tg2$, with $Sg2_m \neq Sg2_o$, but there is no transition labelled by a from $Sg1_i$ in $Tg1$, then the state $Sg2_m$ is added into $Sg3$ and the transition $Sg2_j – a \rightarrow Sg2_m$ in $Tg2$ yields the transition $<Sg1_i, Sg2_j> – a \rightarrow Sg2_m$ in $Tg3$. In the case where $Sg1_k = Sg1_o$ (respectively $Sg2_m = Sg2_o$), instead of the transition $<Sg1_i, Sg2_j> – a \rightarrow Sg1_o$ (respectively $<Sg1_i, Sg2_j> – a \rightarrow Sg2_m$), the transition $<Sg1_i, Sg2_j> – a \rightarrow <Sg1_o, Sg2_o>$ is added into $Tg3$.

The transitions from a simple state in $Sg3$, like state $Sg1_k$ or $Sg2_m$ above, for instance, remain the same as defined in $G1$ and $G2$, respectively. The states reached by these transitions are added into $Sg3$, except for the two initial states $Sg1_o$ and $Sg2_o$, each of these is replaced by the initial state $<Sg1_o, Sg2_o>$ of $G3$.

The mapping $Ac3$ is defined as follows: For every state Sg_i in $Sg3$, we have:

- if $Sg_i = <Sg1_i, Sg2_j>$, then $Ac3(Sg_i) = \{X1 \cup X2 \mid X1 \in Ac1(Sg1_i)$ and $X2 \in Ac2(Sg2_j)\}$,
- if $Sg_i = Sg1_i$, with $Sg1_i \in Sg1$, then $Ac3(Sg_i) = Ac1(Sg1_i)$,
- if $Sg_i = Sg2_j$, with $Sg2_j \in Sg2$, then $Ac3(Sg_i) = Ac2(Sg2_j)$.

Given the FLTSs $S1$, $S2$, the following propositions have been proved in [Khen 92] concerning the FLTS $S3$ constructed by the algorithm Merge:

Proposition 1
$S3$ extends $S1$ and $S3$ extends $S2$.

Merge satisfies our first requirement as stated above in Proposition 1. However, the second requirement about the recursive choice between behaviors of $S1$ and behaviors of $S2$, in the case of cyclic behaviors in $S1$ and $S2$, is not always satisfied. This requirement may be satisfied, if all the cyclic traces in S_1 and all the cyclic traces in S_2 remain cyclic traces in S_3. For that, all the minimal cyclic traces in S_1 and all the minimal cyclic traces in S_2 should remain minimal cyclic traces in S_3. Unfortunately, there are some situations where a minimal cyclic trace in S_1 (respectively S_2) does not remain a minimal cyclic trace in S_3. This is the case, when a given trace σ is a minimal cyclic trace in S_1 (respectively S_2), but σ is a noncyclic trace in S_2 (respectively S_1). After executing such a minimal cyclic trace, S_3 reaches a state, which is different from its initial state. Therefore, it does not offer again a choice between the behaviors of S_1 and the behaviors of S_2. Figure 2 illustrates such kind of situations. After performing a, which is a minimal cyclic trace in S_1, S_3 does not offer a choice between behaviors in S_1 and behaviors in S_2, because the trace a belongs to S_2 and it is not a cyclic trace in S_2. However, the minimal cyclic trace a.b in S_2 remains minimal cyclic trace in S_3. In Proposition 2, we determined a sufficient condition, for which a minimal cyclic trace in S_1 (respectively S_2) remains a minimal cyclic trace in S_3.

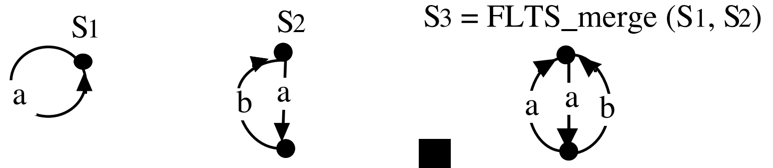

Figure 2. Counterexample for the minimal cyclic traces

Proposition 2

- For any minimal cyclic trace σ in S_1, if $\sigma \notin Tr(S_2)$ or σ is a cyclic trace in S_2, then σ is a minimal cyclic trace in S_3.
- Reciprocally, for any minimal cyclic trace σ in S_2.

Any trace of S_3 is either a trace of S_1, or a trace of S_2, or results from the concatenation of traces of S_1 and S_2. The following proposition shows how a trace $\sigma.a$ of S_3 may be decomposed into its subtraces in S_1 and S_2 when σ is a trace of S_1 (respectively S_2).

Proposition 3

$\forall a \in L_1 \cup L_2$, if $\sigma \in Tr(S_1)$ and $\sigma.a \in Tr(S_3)$,

then $\sigma.a \in Tr(S_1)$, or $\sigma.a \in Tr(S_2)$, or

($\exists\, \sigma_1, \sigma_2$ such that $\sigma = \sigma_1.\sigma_2$, $S_1 {=}\sigma_1{\Rightarrow} S_1$, $S_1 {=}\sigma_2{\Rightarrow} S_1' {\neq}a{\Rightarrow}$, $S_2 {=}\sigma_2{\Rightarrow} S_2' {=}a{\Rightarrow}$).

Reciprocally, for $\sigma \in Tr(S_2)$ and $\sigma.a \in Tr(S_3)$.

4 Merging Structured Specifications

In this section, we consider distributed system specifications, which consist of a parallel composition of subsystem specifications as shown in Figure 3. Such specifications have the following form: S = (S1 $|_A$S2) \ B, where A and B represent sets of actions. The subsystem specifications S1 and S2 may also have the same form as S and so on, until a level where the specifications have no structure and are defined directly in terms of some allowed ordering of actions as monolithic specifications. These specifications are called basic components, they may be nondeterministic, but are assumed to be finite. For instance, these specifications are represented by the streaked boxes in Figure 3.

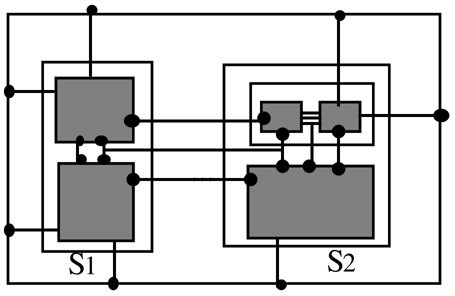

Figure 3. Structure of a Distributed System Specification

Given a distributed system specification S_{old}, which consists of a parallel composition of subsystem specifications and so on until the basic components, and a specification S_{added}, we want to construct a specification S_{new}, such that S_{new} extends S_{old}, and S_{new} extends S_{added}. S_{new} should have the same structure as S_{old}. As for the merging of monolithic specifications, in the case of cyclic traces, S_{new} should offer a choice between behaviors of S_{old} and behaviors of S_{added}, in a recursive manner.

4.1 Identical Structure for S_{old} and S_{added}

We assume that the specifications S_{old} and S_{added} are both structured according to the form (S1 $|_A$S2)\B described above, and S1 and S2 are either basic components or again structured by parallel composition. Moreover, we assume that S_{old} and S_{added} have an identical structure. In other words, the form of the expression S_{old} is identical to the form of the expression S_{added}. To every subsystem specification in S_{old} corresponds a subsystem specification in S_{added} and vice versa. To every basic component Ci_{old} in S_{old}, corresponds a basic component Ci_{added} in S_{added} and vice versa.

The following algorithm for merging structured specifications, called Structured_Merge, is recursive over the structure of S_{old} and S_{added}. It is based on the algorithm Merge, for merging monolithic specifications, described in Section 3.

Merging Algorithm for Structured Specifications

```
Structured_Merge(S1, S2) =
if  S1 = (S11 $|_A$ S12)\B, S2 = (S21 $|_C$ S22)\D,
        then  (Structured_Merge(S11, S21) $|_{(A \cup C)}$ Structured_Merge(S12, S22)) \ (B ∪ D)
        else   Merge(S1, S2)  (*  S1 and S2 are basic components *)
```

S_{new}, obtained by Structured_Merge(S_{old}, S_{added}), has a structure identical to the structure of S_{old} and S_{added}. As basic component, instead of Ci_{old} or Ci_{added}, it has Ci_{new} which results from the merging of Ci_{old} and Ci_{added} by the algorithm Merge.

Unfortunately, S_{new} does not always extend S_{old} and S_{added}. The extension of the basic components of S_{old} and S_{added} is not sufficient to insure the extension of S_{old} and S_{added}, respectively. Consider the counterexample in Figure 4, where $S_{old} = (C1_{old} \,|_{\{g1\}}\, C2_{old})\backslash\{g1\}$, $S_{added} = (C1_{added} \,|_{\{g2\}}\, C2_{added})\backslash\{g2\}$. The structure of the specification S_{new} is identical to the structure of S_{old} and S_{added}, but S_{new} does neither extend S_{old} nor S_{added}. Indeed, S_{old} never refuses the action b after trace a, whereas S_{new} may refuse action b after trace a. The same observation holds for action c after trace a. The trace a is common for $C1_{old}$ and $C1_{added}$ and it is followed by a hidden action g1 in $C1_{old}$ and g2 in $C1_{added}$. The merging of $C1_{old}$ and $C1_{added}$ leads to a choice between the two hidden actions g1 and g2 after the trace a, in $C1_{new}$. The components $C1_{new}$ and $C2_{new}$ may, internally, choose to synchronize on action g1 or g2, after a trace a, and offer only action b or only action c, respectively.

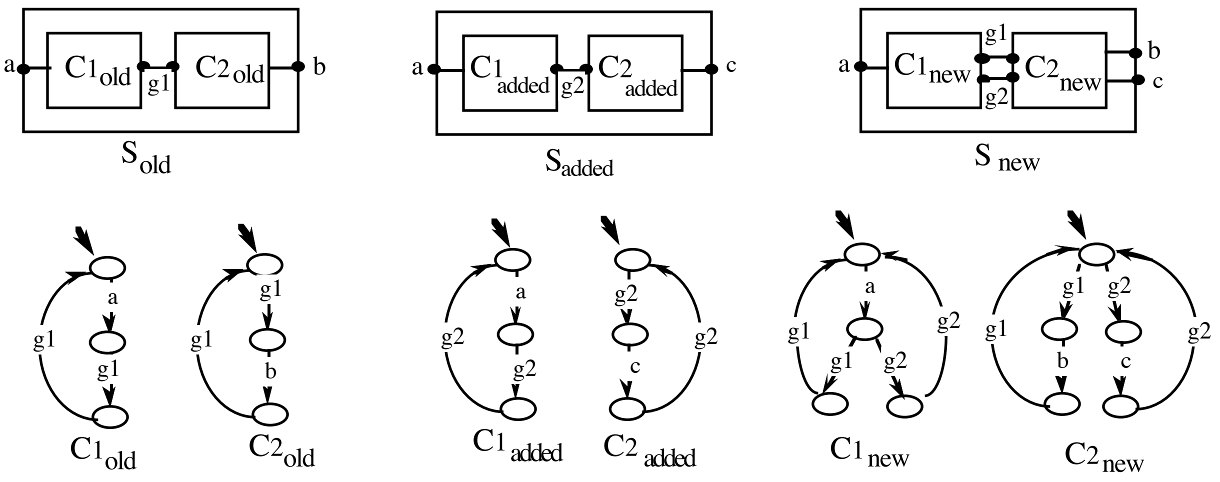

Figure 4. Counterexample

In Theorem 1, we have stated sufficient conditions for S_{old} and S_{added} such that S_{new} extends S_{old} and S_{new} extends S_{added}. We denote by HG_{old} the set of hidden action names in S_{old}, and by HG_{added} the set of hidden action names in S_{added}. The proof of Theorem 1 is given in the Appendix.

Theorem 1

Given S_{old} in the form of a hierarchical structure with the basic components $C1_{old}$, $C2_{old}$, ..., Cn_{old},

S_{added} with an identical structu[redacted]nd the basic components $C1_{added}$, [redacted], ..., Cn_{added}, and S_{new} = Structured_Merge(S_{old}, [redacted]) as [redacted]efined by the merging a[redacted]orithm [redacted]efined above,

we have that S_{new} ext S_{old} [redacted] if th[redacted]llowin[redacted] co[redacted]ions [redacted]re sa[redacted]fied:

(a) $\forall i$, i = 1,..., n, act(Ci_{old}) [redacted] act(C[redacted]) [redacted] G[redacted] = Ø,
(b) $\forall i, j, i \neq j$, (act(Ci_{old}) a[redacted]old) [redacted]ct([redacted])) [redacted]ct(S_{old}) a[redacted])) = Ø,
(c) For x = old, added, [redacted] that f[redacted], $g \in Tr(Ci_x)$ and $g \in Tr(Cj_x)$,
(d) $\forall i$, i = 1, ..., n,
 1 - $\forall\ \sigma \in Tr(Ci_{old})$ - { }, [redacted] with [redacted] $\in$ H[redacted]dded, and reciprocally,
 2 - $\forall\ a \in act(S_{old})$, if $a \in Tr(Ci_{old})$, [redacted]en [redacted], unless σ is cyclic in Ci_{added}, and reciprocally.

Condition (a) says that the names of hidden actions in S_{added} should not conflict with the names of observable or hidden actions in S_{old}. Reciprocally, the names of hidden actions in S_{old} should not conflict with the names of observable or hidden actions in S_{added}. Note that the names of the hidden actions in both specifications are not important. These actions may be renamed without any observable effect, in order to satisfy this condition.

Condition (b) says that there is no observable action of S_{old} and S_{added} shared by two (or more) basic components of S_{old} (respectively S_{added}). A basic component Ci_{old} in S_{old} may have common observable actions only with the corresponding basic component Ci_{added} in S_{added}, and reciprocally. Consider the example in Figure 5, where $C1_{old}$ and $C2_{added}$ have the action a in common, but they are not merged together. $C1_{new}$ = Merge($C1_{old}$, $C1_{added}$), $C2_{new}$ = Merge($C2_{old}$, $C2_{added}$), $C1_{new}$ extends $C1_{old}$ and $C1_{added}$, and $C2_{new}$ extends $C2_{old}$ and $C2_{added}$. The constructed specification S_{new} may refuse action b or action c, after trace a, whereas S_{old} and S_{added} never refuses b or c after a, respectively. S_{new} does neither extend S_{old} nor S_{added}. In order to prevent such situations, for each observable action, we may assign a "place" and the components with common observable actions have to be merged together, as stated by Condition (b).

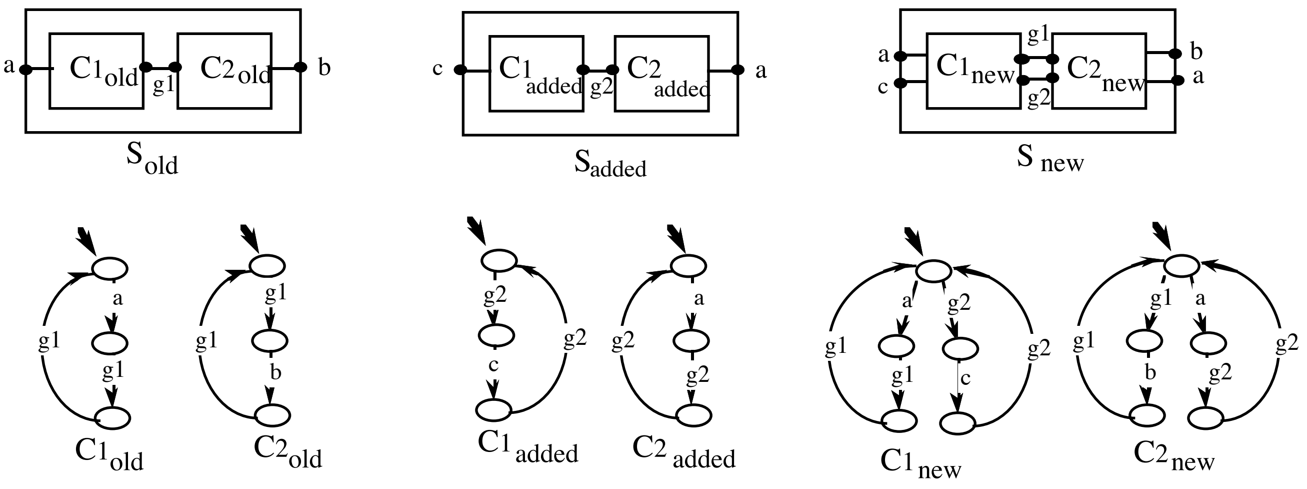

Figure 5. An illustration for Condition (b)

Condition (c) states that S_{old} and S_{added} should not be able to perform an action from HG_{old} or from HG_{added}, respectively, before interacting with the environment. Consider the example in Figure 6, $C1_{new}$ = Merge($C1_{old}$, $C1_{added}$), $C2_{new}$ = Merge($C2_{old}$, $C2_{added}$), $C1_{new}$ extends $C1_{old}$ and $C1_{added}$, and $C2_{new}$ extends $C2_{old}$ and $C2_{added}$. However S_{new} does not extend S_{added}. After an internal move by executing the hidden action g1, it refuses the action a, whereas S_{added} never refuses action a after an empty trace.

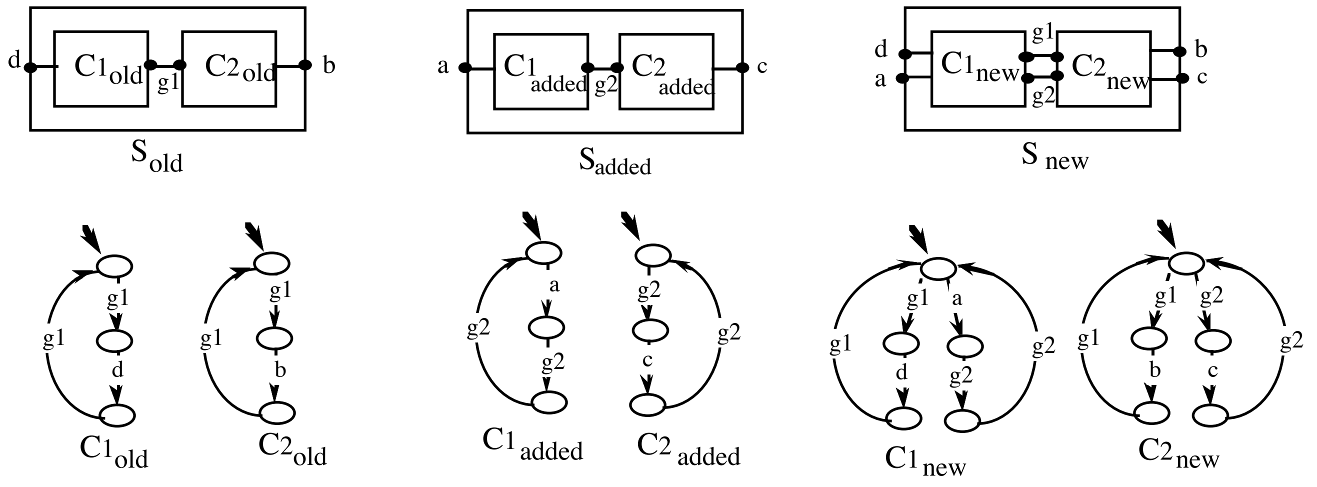

Figure 6. An illustration for Conditio[redacted]

Condition (d-1) prevents from any new nondeterminism which m[redacted]introduced by the hidden actions in HG_{added} with respect to behavior in S_{old} and reciprocally, [redacted] shown in Figure 4. For a given pair of basic components Ci_{old} and Ci_{added}, a common trace σ ($\neq$) sh[redacted]t be followed by hidden actions from HG_{old} or HG_{added}.

Condition (d-2) is introduced in order to prevent situations similar to the one shown in Figure 7. Assume that S_{old} = $(C1_{old} |_{\{g1, g2\}} C2_{old})\backslash\{g1, g2\}$ and $S_{added} = (C1_{added} |_{\emptyset}$ stop$)\backslash\emptyset$. The merging algorithm for structured specifications leads to S_{new} = $(C1_{new} |_{\{g1, g2\}} C2_{new}) \backslash\{g1, g2\}$, where $C1_{new}$ is shown in Figure 7 and $C2_{new} = C2_{old}$. We have $C1_{new}$ ext $C1_{old}$ and $C1_{new}$ ext $C1_{added}$ as well as $C2_{new}$ ext $C2_{old}$ and $C2_{new}$ ext $C2_{added}$. However, S_{new} does not extend S_{old}. For instance, after the trace f.a.b.c, S_{new} may refuse to perform action d, whereas S_{old} never refuses to perform action d after trace f.a.b.c. This is due to the fact that we have two traces $\sigma1$ = a.g1.b and $\sigma2$ = a.g2.b in $C1_{old}$, such that $\sigma1 \neq \sigma2$, $\sigma1\backslash HG_{old} = \sigma2\backslash HG_{old}$, $\sigma1$ is cyclic, $\sigma2$ is not cyclic, $\sigma2$.c is a trace in $C1_{old}$, and c is a trace in $C1_{added}$. It is possible to prevent such situations with a weaker condition than Condition (d-2) as explained in this example. However the verification of such conditions may be complex, whereas Condition (d-2) can be checked very easily.

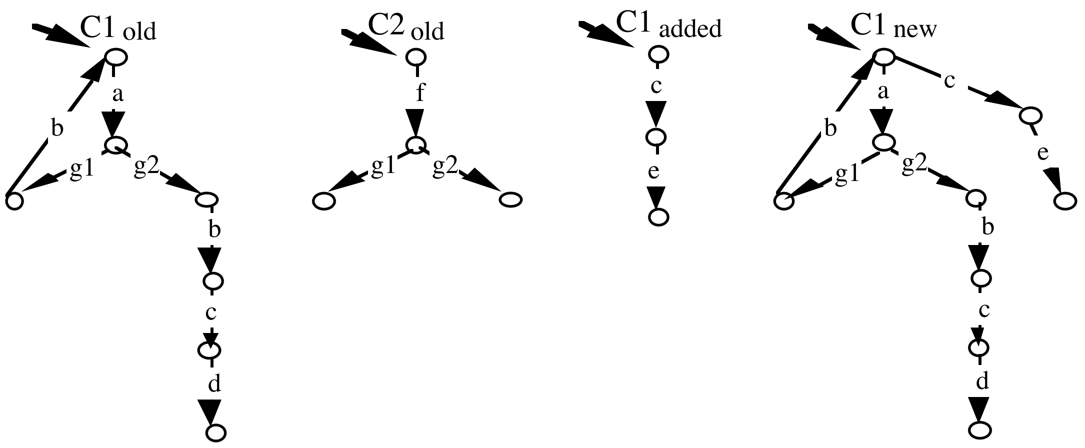

Figure 7. Illustration for Condition (d-2).

Theorem 2 states that under certain conditions on the basic components of S_{old} and S_{added}, a minimal cyclic trace σ in S_{old} (respectively S_{added}) remains cyclic in S_{new}. Therefore, after performing σ, S_{new} reaches its initial state, and offers again a choice between behaviors in S_{old} and behaviors in S_{added}. The proof of Theorem 2 is given in the Appendix.

Theorem [redacted]

Gi[redacted]n spe[redacted]ications S_{old}, S_{added}, and S_{new} as in Theorem 1, and
ass[redacted]nditions of Theorem 1 are satisfied, we have
- For[redacted]l cyclic trace σ in S_{old}, if for i = 1,..., n, σi is a minimal cyclic trace in Ci_{old} and (σi
Tr(Ci_{ac}[redacted]d) or σi is a cyclic trace in Ci_{added})), where σi represents the sequence of actions performed[redacted]y Ci_{old}, when S_{old} performs the trace σ, then σ is a cyclic trace in S_{new}.
- Reciprocally, for any minimal cyclic trace σ in S_{added}.

Example

In the following, we will illustrate our approach by an example. We use variations of the Daemon game [ISO 8807]. We assume a simple game description, noted "Simple Daemon Game" (SDG for short). The player may insert a coin, start the game, probe the system, then he randomly loses or wins. The inserted coin may be refused, the user has to recollect his coin and insert it again until accepted by the system before he can start the game. We have, arbitrarily, structured this system as follows: SDG = (P1 $|_{\{g1\}}$ P2)\{g1}. The processes P1 and P2 synchronize through action g1. The structure of SDG and the processes P1 and P2 are drawn in Figure 8.

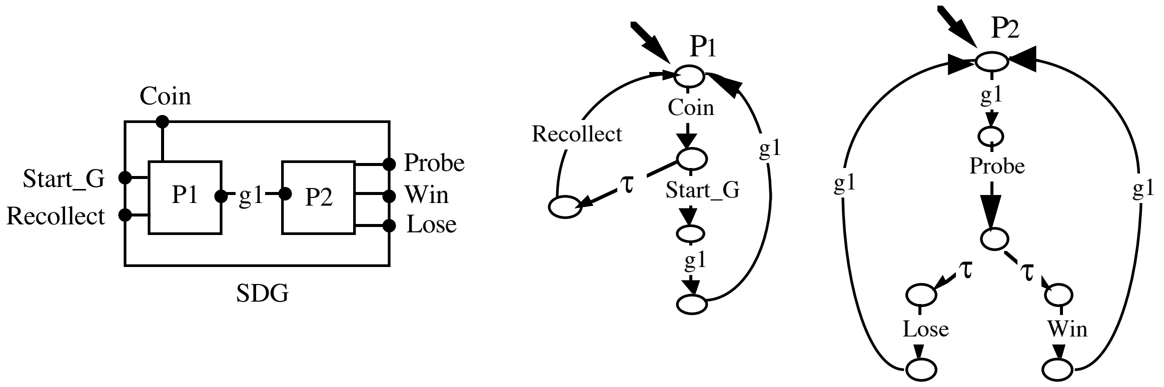

Figure 8. Simple Daemon Game Description

Assume that we want to enrich the specification above, in order to describe a new system ("Combined Game", or CG for short), where the player can play, alternatively, the simple game and a sophisticated game, called "Jackpot Daemon Game". As for the "Simple Daemon Game", the player has to insert a coin before starting the game. This coins may be refused. Once the coin has been accepted, the player can start the game, probe, then he randomly loses or wins. If he wins, the game continues. He can probe again, then he randomly loses or get the "Jackpot". The specification of this sophisticated game is given as follows: JDG = (P3 $|_{\{g2\}}$ P4)\{g2}. The structure of this specification is identical to the structure of SDG. The structure of JDG and the processes P3 and P4 are drawn in Figure 9.

These specifications (games) have many interactions in common. SDG and JDG satisfy the sufficient conditions of Theorem 1. Applying the algorithm Structured_Merge leads to: CG = (P13 $|_{\{g1, g2\}}$P24)\{g1, g2}, where P13 and P24 are described in Figure 10. P13 results from the merging of P1 and P3 by the algorithm Merge. P24 results from the merging of P2 and P4 by the algorithm Merge. The processes P1, P2, P3, and P4 are assumed to be basic components. By construction, we have P13 ext P1, P13 ext P3, P24 ext P2, P24 ext P4, CG ext DG and CG ext JDG. In this example, it is easy to verify that each minimal cyclic trace in SDG (respectively JDG) remains cyclic in CG

(Theorem 2). Therefore, CG describes a new system where the user may always alternate between the "Simple Daemon Game" and the "Jackpot Daemon Game".

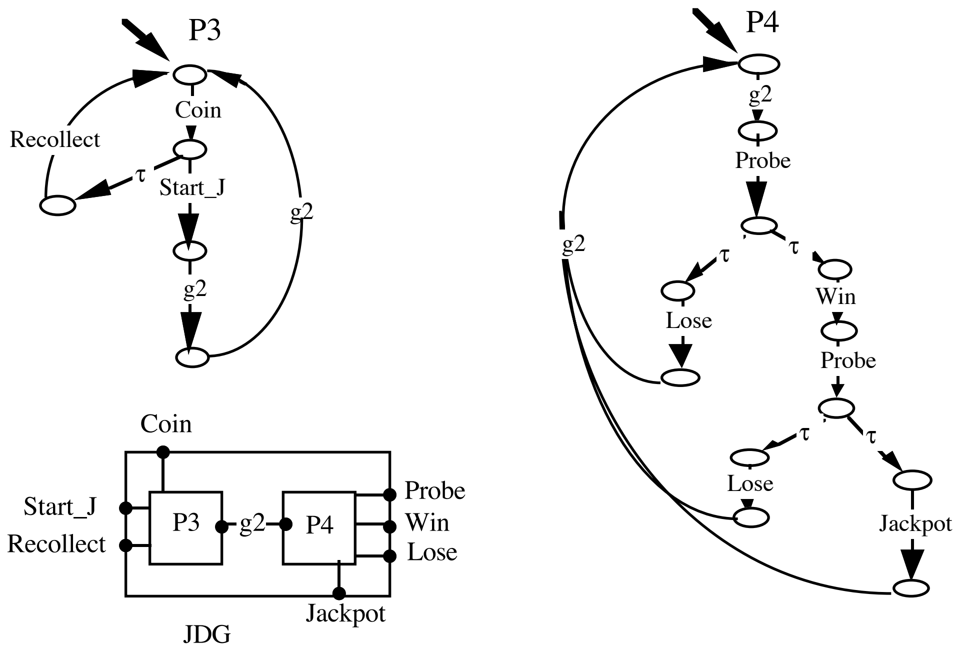

Figure 9. Jackpot Daemon Game Description

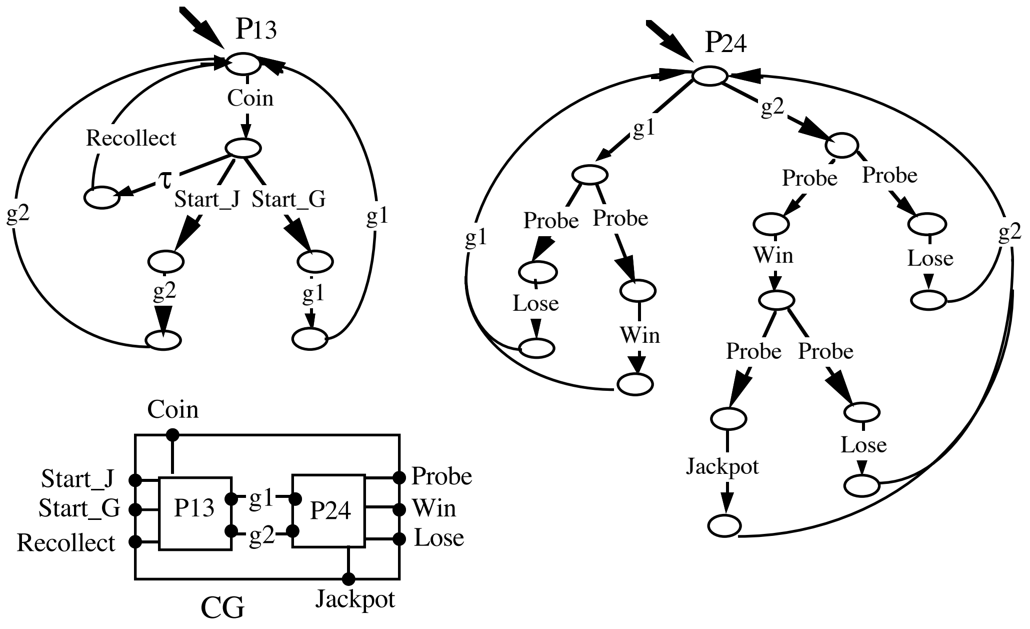

Figure 10. Combined Game Description

4.2 Nonidentical Structure for S_{old} and S_{added}

We assume now that the specifications S_{old} and S_{added} are constructed through the combination of the parallel and hiding operators as previously, but their structures are not identical. For instance, the structures of $S_{old} = (C1_{old} |_A C2_{old}) \backslash B$ and $S_{added} = (S1_{added} |_C C3_{added}) \backslash D$ where $S1_{added} = (C1_{added} |_E C2_{added}) \backslash F$, are not identical. There is no one to one correspondence between the subexpressions of S_{old} and the subexpressions of S_{added}. Before applying the merging algorithm Structured_Merge, S_{old} and S_{added} are transformed into strongly bisimilar specifications S_{old}' and S_{added}', respectively, such that the structures of S_{old}' and S_{added}' are identical. This transformation may be done by the procedure Transform described below. This procedure is given in a style similar to a Prolog program. In order to determine S_{old}' and S_{added}', it may be called by Transform(S_{old}, S_{added}, S_{old}', S_{added}'). Procedure Transform consists of 4 rules applicable to the different forms of the expressions to be transformed.

```
Transform((S11 $|_A$ S12)\ B,  (S21 $|_C$ S22) \D,  (S11' $|_A$ S12')\ B,  (S21' $|_C$ S22')\D) =
        Transform (S11, S21, S11', S21') , Transform (S12, S22, S12', S22').
Transform(S1,  (S21 $|_C$ S22) \D,  (S11' $|_\emptyset$ S12'),  (S21' $|_C$ S22')\D) =
        Transform (S1, S21, S11', S21') , Transform (stop, S22, S12', S22').
Transform((S11 $|_A$ S12)\ B, S2,  (S11' $|_A$ S12')\ B,  (S21' $|_\emptyset$  S22')) =
        Transform (S11, S2, S11', S21') , Transform (S12, stop, S12', S22').
Transform(S1, S2,  S1,  S2).
```

Note that we have introduced a dummy process stop, which is a process that does nothing [ISO 8807]. S_{old}' (respectively S_{added}') is strongly bisimilar to S_{old} (respectively S_{added}). It is deduced from the fact that $S \sim (S |_\emptyset \text{ stop})$, and $(S1 |_A S2)\backslash B \sim (S1' |_A S2)\backslash B$ if $S1 \sim S1'$ [Miln 89]. S_{old}' and S_{added}' are merged into S_{new}, using the algorithm Structured_Merge introduced in the previous subsection. If the sufficient conditions of Theorem 1 are satisfied by S_{old}' and S_{added}', then S_{new} ext S_{old}' and S_{added}'. Since S_{old}' (respectively S_{added}') is strongly bisimilar to S_{old} (respectively S_{added}), it follows that S_{new} ext S_{old} and S_{added}. Same observation for Theorem 2.

4.3 Discussion

(a) Avoiding the conditions of Theorem 1: Note that, whenever the sufficient conditions of Theorem 1 are not satisfied by the basic components of S_{old} and S_{added}, we may consider the processes at the next higher level as monolithic and apply algorithm Merge to them. The internal structure of such processes will be lost and we will have to redesign it after the merging.

(b) Extra behavior: In the merging of structured specifications, S_{new} may contain certain extra behaviors allowed neither by S_{old} nor by S_{added}. This kind of side effect happens when alternative behaviors from S_{old} and S_{added} involve different components. In this case, these alternative behaviors may be interleaved as shown by the example in Figure 11, in which $S_{old} = (C1_{old} |_{\{g1\}} C2_{old})\backslash\{g1\}$, $S_{added} = (C1_{added} |_{\{g2\}} C2_{added})\backslash\{g2\}$, $C1_{new}$ = Merge($C1_{old}$, $C1_{added}$), $C2_{new}$ = Merge($C2_{old}$, $C2_{added}$), and $S_{new} = (C1_{new} |_{\{g1, g2\}} C2_{new})\backslash\{g1, g2\}$. S_{new} extends S_{old} and S_{new} extends S_{added}. However S_{new} allows more than what is allowed by S_{old} and S_{added}, such as the sequences of actions a.c or c.a.

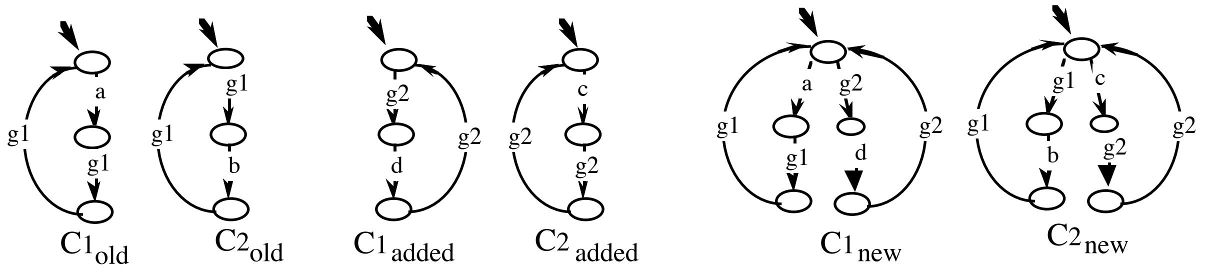

Figure 11. Extra behaviors

(c) Improved procedure Transform: An improved procedure Transform may be used, in order to produce, S_{old}' and S_{added}', which satisfy, systematically, Condition (b) of Theorem 1. Using this improved procedure, if, for instance, S11 and S22 have some observable actions from (act(S_{old}) act(S_{added})) in common, and S11 and S21 do not have observable actions in common, then S11 is associated with S22, instead of S21, for further transformations, and the expression $(S21' |_C S22')\backslash D$ is changed to $(S22' |_C S21')\backslash D$. Note that it may happen that S11 has common observable actions with S21 and with S22. In this case, the specifications S_{old}' and S_{added}' produced by the procedure Transform described in the previous subsection do not satisfy Condition (b) of Theorem 1. The improved procedure Transform will not be able to transform S_{old} and S_{added}, because of this "incompatible distribution" of observable actions. The specification S_{added} should be redesigned using, for instance, the functionality decomposition algorithm described in [Lang 90]. Using this algorithm, the distribution of the common observable actions over the subexpressions of S_{added} should be guided by the distribution of these actions in S_{old}. The observable actions of S_{added}, which do not belong to S_{old}, can be distributed randomly. Such an algorithm can also be used, if S_{old} is given according to the form $(S1 |_A S2)\backslash B$, but S_{added} is given in a high level form, as a monolithic specification, for instance.

(d) Substitution of a system component: The sufficient conditions in Theorem 1 may be adapted as sufficient conditions for the substitution of a component X in a system SYS by a component Y, with the confidence that the new system SYS' obtained by this substitution satisfies SYS' ext SYS, if Y ext X. For this purpose, we assume that SYS consists of a parallel composition of subsystem specifications and so on until the basic components, X is a basic component in SYS, and Y may be written as Y = Merge(X, X') with a certain X'. SYS represents S_{old}. X' represents S_{added}, which is transformed by the procedure Transform described in the previous subsection into S_{added}'. S_{added}' is strongly bisimilar to S_{added} and for each basic component $Z \neq X$ in S_{old} corresponds a basic component $Z' =$ stop in S_{added}'. To the basic component X in S_{old} corresponds the basic component X' in S_{added}'. S_{new} obtained by merging S_{old} and S_{added}' using the algorithm Structured_Merge represents SYS'. Therefore, SYS' extends SYS if the conditions in Theorem 1 are satisfied.

5. Related Work

In [Ichi 90], the problem of incremental specification in the LOTOS language is approached in the following way: Given the processes B_{old} = C[B1] and B_{added}, deduce B_{new} = C[B2], such that B_{new} ext B_{old}, B_{new} ext B_{added} [illegible] B1. C[] represents a process expression context.

A new LOTOS operator [illegible], called specification merging operator is introduced and the corresponding inference rules are defined. This approach is restricted to specification behaviors without the internal action τ. B1 [illegible] B2 [illegible] behavior, which is supposed to be an extension of B1 and B2. Unfortunately this is not always the case, as shown by the counterexample in Figure 12. B1 never refuses the action c after trace a.b, whereas B1 B2 may refuse the action c after trace a.b. Moreover, B1 B2 is [illegible] to behave, alternatively, as B1 and B2, in a recursive [illegible] behavior where the environment has to choose behavior B1 or behavior B2 once and for all.

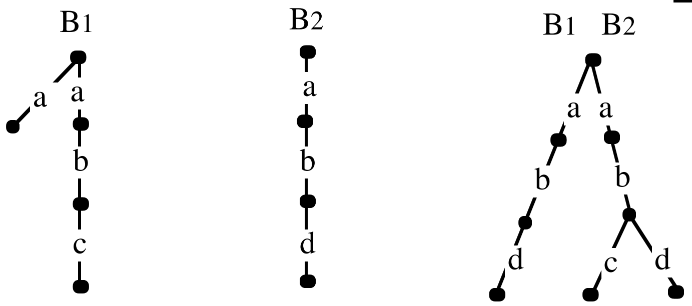

Figure 12. Counterexample for Ichikawa et al. approach

There is no systematic approach to deduce B_{new} from B_{old} and B_{added} with considerations of the structure of these specifications. They considered the basic LOTOS operators and investigated their properties w. r. t. the extension relation. The combination of the hiding operator (hide G in B) and the parallel operator (B1 |[G]| B2) was not considered formally. We note that Proposition 2 in [Ichi 90], which states that (B3 |[G]| B2) ext (B1 |[G]| B2), if B3 ext B1 and out(B3) [illegible] out(B2) [illegible] G, [illegible] hold. We may consider the following counterexample:

B1 = a ; b ; stop

B2 = a ; c ; b; stop

B3 = a ; (b ; stop [] c ; stop)

G = {a , b}

It is clear that B[illegible]ext [illegible] and [illegible]out(B_3) [illegible]t(B_2)) [illegible] does n[illegible] exten[illegible]B_1|[a, b]|B_2.

In [Rudk 91] [illegible]ance is defined for LOTOS. It is seen [illegible]an in[illegible]emen[illegible]modification technique. A [illegible]ing [illegible]rator is introduced and denoted by "   ". T[illegible]r is defined such that if s = t   m , [illegible]nds t and any recursive call in t or m is redirected to s. However strong restrictions are imposed on t and m, such that m should be stable (no internal transition as first event), the initial events of m should be unique and distinct from initial events of t, and so on. There is no requirement such that s should also extend m, and no considerations to the structure of t or how this modification m is propagated to the processes in t.

6. Conclusion

In this paper, we have proposed an incremental construction approach for distributed system specifications. Given two specifications S_{old} and S_{added}, we construct a specification S_{new}, which extends S_{old} and S_{added}, if some sufficient conditions stated in Theorem 1 are satisfied. S_{new} has the same structure as S_{old}. Therefore the designer will not have to redesign this structure. In the case of cyclic behaviors of S_{old} and S_{added}, provided that certain sufficient conditions stated in Theorem 2 are satisfied, S_{new} offers a choice between behaviors in S_{old} and S_{added}, in a recursive manner. Note that in the case of merging monolithic specifications, the more simple propositions 1 and 2 of section 3 apply.

The labelled transition systems model is the underlying semantical model for many specification languages, such as, LOTOS [ISO 8807], CCS [Miln 89]. Therefore, the approach described in this paper is applicable for specifications written in these languages.

The proposed incremental specification approach is useful for dealing with multiple-function specifications. Instead of handling all the functions simultaneously, it allows one to focus on one function at a time for the design and verification. The merging approach will derive, whenever possible, the required combined specification. From another point of view, it allows one to extend existing specifications with new behaviors required by the user.

The approach proposed in this paper may promote the reusability of specifications. Once a function specification has been constructed and verified, for example, it may be used in many system specifications where it is required.

In this paper, we determined sufficient conditions, for which the combined specification S_{new} extends the specifications S_{old} and S_{added}. As future work, it will be interesting to study the necessity of each condition. More complex applications of our approach are also expected.

References

[Aho 79] A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison-Wesley, 1979.

[Brin 86] E. Brinksma, G. Scollo and S. Steenbergen, LOTOS specifications, their implementations and their tests, Protocol Specification, testing and verification, VI, Montréal, Canada, 1986, Sarikaya and Bochmann (eds.).

[DeNi 87] R. De Nicola, Extensional Equivalences for Transition Systems, Acta Informatica, 24 1987, pp. 211 - 237.

[Henn 85] M. Hennessy, Acceptances Trees, J. of ACM, Vol.32, No. 4, Oct. 1985, pp. 896 - 928.

[Ichi 90] H. Ichikawa, K. Yamanaka and J. Kato, Incremental Specification in LOTOS, Protocol Specification, Testing and Verification X (1990), Ottawa, Canada, Logrippo, Probert and Ural (eds.), pp. 185 - 200

[ISO 8807] ISO - Information Processing Systems - Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, Feb. 1989.

[Kell 76] R. Keller, Formal verification of parallel programs, Communication of the ACM 19 July 1976, pp. 371-384.

[Khen 92] F. Khendek and G. v. Bochmann, Merging specification behaviors, submitted for publication.

[Lang 90] R. Langerak, Decomposition of functionality : a correctness-preserving LOTOS transformation, Protocol Specification, Testing and Verification X, 1990, Ottawa, Canada, Logrippo, Probert and Ural (eds.), pp. 203 - 218.

[Miln 89] R. Milner, Communication and Concurrency, Prentice-Hall International, 1989.

[Rudk 91] S. Rudkin, Inheritance in Lotos, Formal description technique (FORTE), Sydney, Australia, 1991, pp. 415 - 430.

[Viss 85] C. A. Vissers and L. Logrippo, The importance of the service concept in the design of data communications protocols, Protocol Specification, Testing, and Verification, V, Toulouse-Moissac, France, June 10 - 13, 1985, M. Diaz (ed.).

[Viss 88] C. A. Vissers, G. Scollo and M. v. Sinderen, Architecture ans Specification Style in Formal Descriptions of Distributed Systems, Protocol Specification, Testing, and Verification, VIII, North-Holland, 1988, pp. 189 - 204.

Appendix

For the needs of the proofs in this appendix, we use the following notations:
act(σ) : the set of action names in trace σ,
σ\X: the projection of σ to act(σ) - X,
Comp(old, σ1, σ2, ..., σn) : represents the set of possible traces obtained by composition of σ1, σ2, ..., σn in S_{old} structure with the hidden gates of S_{old}.
Comp(new, σ1, σ2, ..., σn) : represents the set of possible traces obtained by composition of σ1, σ2, ..., σn in S_{new} structure (which is the same than S_{old} structure) with the hidden gates of S_{new}.

Proof of Theorem 1

We will prove that S_{new} ext S_{old}. The proof for S_{ne}[illegible]xt S_{added} is very similar.

a - First, we have to prove that any trace σ of S_{old} [illegible] also a trace of S_{new}:

let σ ∈ $Tr(S_{old})$, it implies that $\exists$ [illegible] for i = 1, ..., n, such that σ ∈ Comp(old, $σ1_σ$, $σ2_σ$, ..., $σn_σ$). From Proposition 1, we have Ci_{new} ext Ci_{old}. It follows that, for i = 1, .., n , $σi_σ$ ∈ $Tr(Ci_{new})$. By Condition (a), we deduce that σ ∈ Comp(new, $σ1_σ$, $σ2_σ$, ..., $σn_σ$). Therefore, σ ∈ $Tr(S_{new})$.

b - In a[illegible]ave to prove that S_{new} w[illegible] not block where S_{old} does not block:

We have to prove th[illegible] $\forall$ σ ∈ $Tr(S_{old})$ and A $\subseteq$ a[illegible],
if $\exists$ [illegible]$_{w}$ =σ⇒S_{new}' ≠a⇒, ∀ a ∈ A,
then $\exists$ [illegible]$_{ld}$ =σ⇒S_{old}' ≠a⇒, ∀ a ∈ A,

Let $\sigma \in Tr(S_{old})$, A [illegible] a[illegible]$_{new}$' such that $S_{new}=\sigma\Rightarrow S_{n}$[illegible]'$\neq a\Rightarrow$, $\forall$ a $\in$ A, it implies [illegible]$_{new}$), and Ci_{new}' for i = 1,...,n, such that [illegible] Comp(ne[illegible], $\sigma 1_\sigma$, $\sigma 2_\sigma$,...,σn_σ) and $Ci_{new}=\sigma i_\sigma\Rightarrow Ci_{new}'\neq a\Rightarrow$, $\forall$ a $\in$ A, since A [illegible] a[illegible]S_{old}) [illegible] (HG_{old} HG_{added}) = Ø.

First, we have to [illegible]at $\sigma i_\sigma \in Tr(Ci_{old})$, for i = 1, ..., n.

We distinguish tw[illegible]s:

b - 1: σ =

From Propo[illegible]ion 3, we [illegible]duce that, for a gi[illegible] a ∈ (act([illegible]$_{old}$) act([illegible])), if a ∈ Tr(Ci_n[illegible]) [illegible] a ∈ Tr(Ci_{old}) [illegible]r a ∈ Tr(C[illegible]$_{dded}$). By Conditi[illegible] (c) which [illegible]te[illegible]hat S_{old} (S[illegible]$_{ded}$) should not be abl[illegible]o perform an [illegible]tion from [illegible]G_{old} (HG_{added}) [illegible] interact[illegible] [illegible]h the envi[illegible]nment. By Condition [illegible]), act(Ci_{old}) [illegible]G_{added} = Ø [illegible]nd act(Ci_{adde}[illegible] [illegible]G_{old} = Ø, [illegible] i [illegible] 1, ..., n. It [illegible]llows that [illegible] Cj_{new}, with i≠j, s[illegible] some g ∈ (HG_{old} [illegible]G_{added}), g ∈ Tr(Ci_{new}) and g ∈ Tr(Cj_{new}). It foll[illegible]s that S_{new} is n[illegible] able to perform an a[illegible]on from (HG_{old} HG_{added}) before inte[illegible]cting w[illegible] the environment. W[illegible]uce that σi_σ = and [illegible] Tr(Ci_{old}), for i = 1, ..., n.

b - 2: σ ≠

From (b-1) above, we know that S_{ne}[illegible]s not able to perform [illegible] action fr[illegible]m (HG_{old} HG_{added}) before interacting with the envi[illegible]. Therefore, σ = a.σ' with a [illegible] act(S_{old}) and [illegible]$_{new}$), such that σl_σ = a.σl'.

Now, assume that [illegible]$_{new}$), but σk_σ Tr(Ck_{o}[illegible]. Mo[illegible]ifically, σk_σ can be written in the form of σk'.μ.σk'', with σk' ∈ Tr(Ck_{old}), but σk'.μ Tr(Ck_{o}[illegible]. μ may be an action from HG_{old}, or from HG_{added}, [illegible]bservable action from act(S_{old}).

We distinguish two sub-[illegible]:

b - 2 - 1: σk' =

b - 2 - 1 - 1: μ ∈ HG_{old}

We have μ ∈ Tr(Ck_{new}), with [illegible] ∈ HG_{old}. [illegible]om Proposition 3, [illegible] ∈ Tr(Ck_{old}) or μ ∈ Tr(Ck_{added}), since Ck_{new} resul[illegible] from the m[illegible]rging of Ck_{old} and C[illegible] [illegible]the [illegible]lgorithm Merge. By Condition (a), act(Ck_{added}) [illegible]G_{old} = Ø, [illegible] follows that μ Tr(Ck_{ad}[illegible]). W[illegible]ave μ ∈ Tr(Ck_{old}), which contradicts our hypothesis above. Consequently, [illegible]$_{new}$), such that σk_σ Tr(Ck_{old}).

b - 2 - 1 - 2: $\mu \in$ **act(S_{old})** (μ is an observable action)

By Proposition 3, we deduce again that $\mu \in Tr(Ck_{old})$ or $\mu \in Tr(Ck_{added})$. Assu[illegible] that μ $Tr(Ck_{old})$, and $\mu \in Tr(Ck_{added})$. We have [illegible] $Tr(S_{old})$, it follows that [illegible], such that $\sigma k_{old} = s.\mu.t$, because of the distribution of [illegible]servable actions over the basic components of S_{old} and S_{added} expressed by Condition (b). [illegible]e hav[illegible] $\mu \in Tr(Ck_{added})$ and $s.\mu \in Tr(Ck_{old})$, by Condition (d-2), it follows that Ck_{old}=s$\Rightarrow Ck_{old}$ and [illegible] follo[illegible]s that $\mu \in Tr(Ck_{old})$, which contradicts our assumption. Consequently, $\mu \in$ Tr[illegible] con[illegible]dicts our original hypothesis. Therefore, $\sigma k_{\sigma} \in Tr(Ck_{new})$, such [illegible](Ck$_{ol}$[illegible]

b [illegible]2 - 1 - 3: [illegible] $\in$ **HG$_{added}$**

W[illegible] deduce th[illegible] [illegible]$_{new}$), such that σm[illegible] = σm[illegible].σm''. F[illegible]n (b-1) above, we know th[illegible] S_{new} is n[illegible] able to perform an action from (HG[illegible] [illegible]) be[illegible]nteracting with the en[illegible]ronment. [illegible] other words, we know that [illegible] with i[illegible]uch for some g $\in$ (HG_{old} HG[illegible]) g $\in Tr(Ci_{new})$ and g $\in Tr(Cj_{new})$. It f[illegible]

We distinguish two cases, σm'$\in Tr(Cm_{old})$ [illegible]d σm[illegible] Tr(Cm$_{o}$[illegible]:

- σm'$\in Tr(Cm_{old})$: we have [illegible]'.$\mu \in$ Tr(Cm[illegible] Tr(Cm[illegible]), since $\mu \in$ [illegible]G$_{added}$. By Condition (d-1), we also ha[illegible]'.μ Tr(C[illegible]). By Proposition 3, it follo[illegible] that σm[illegible] σm1'.σm2' with Cm_{old}=σm[illegible]$\Rightarrow Cm_{old}$, Cm_{old}=[illegible] Cm_{old}', Cm_{added}=[illegible]2'$\Rightarrow$[illegible]$_{added}$' a[illegible] Cm_{added}'=$\mu\Rightarrow$. [illegible] σm2'$\neq$, w[illegible]σm2'($\neq$) $\in$ [illegible]$_{old}$) and σm2'.μ Tr([illegible] contradiction [illegible]h Condition (d-1). If σm2'= , w[illegible] $\mu \in Tr(Cm_{added})$. [illegible] Tr(Ck$_{ne}$[illegible]), by [illegible]oposition 3, it follows that $\mu \in Tr(Ck_{added})$, since μ Tr(Ck$_{o}$[illegible] contradi[illegible]dition (c). We have reached a contradiction again, it fo[illegible] that [illegible]

- σm' Tr(Cm[illegible]): we deduce that σm'= s.μ'.t, where s $\in$ Tr([illegible]m$_{old}$) but [illegible]). Depending on s and μ', we proceed recursive[illegible]. In the case w[illegible]add[illegible], we reach, immediately, a contradiction as shown in the othe[illegible]subcases. In the case [illegible]added, we proceed recursively, until σl_{σ} = a.μ*.σl'' w[illegible]old), but a.μ* [illegible]r(Cl$_{ol}$[illegible]. Since $\sigma \in Tr(S_{old})$, Condition (c) states that S_{old} is[illegible]t able to perfor[illegible]tion g from [illegible]$_{old}$ before interacting with the environment and Condition (b) [illegible]r the distributio[illegible] of [illegible]serva[illegible]e acti[illegible]s over the basic components of S_{old} and S_{added}, it follo[illegible]s that [illegible] s[illegible] that [illegible]. If $\mu^* \in HG_{old}$, or $\mu^* \in$ act(S_{old}), it is solve[illegible]ses (b-[illegible] we reach in both cases a contradiction. If $\mu^* \in$ H[illegible] [illegible]ndition (d[illegible] Tr(Cl$_{add}$[illegible]). We have a.$\mu^* \in Tr(Cl_{new})$, a $\in Tr(Cl_{old})$, a.μ* Tr(Cl$_{ol}$[illegible]nd a. μ* Tr(Cl$_{ad}$[illegible]), by Proposition 3, it follows Ci_{old}=a$\Rightarrow Ci_{old}$ and $\mu^* \in Tr(Cl_{added})$. This is in Contradiction with Condition (c), because we have $\mu^* \in Tr(Ck_{added})$

and $\mu^* \in Tr(Cl_{added})$. [illegible]ecursively, each assumption is contradicted until the first one: μ [illegible] HG_{added}. Consequently, we can n[illegible]ve $\mu \in HG_{added}$.

b - 2 - 2: $\sigma k' \neq$ [illegible]

b - 2 - 2 - 1: μ [illegible] HG_{old}

We have $\sigma k'.\mu$ [illegible] $Tr(Ck_{ne}$[illegible], σk[illegible] $Tr($[illegible]$k_{old})$, and $\mu \in$ [illegible] Condition ([illegible], we [illegible]ave $act(Ck_{added})$ [illegible]$G_{old} = \emptyset$ [illegible]at $\sigma k'.\mu$ $Tr(C$[illegible]$ed)$ [illegible]nd [illegible]hat $\sigma k'=\sigma k1'.\sigma k2'$, $Ck_{old}=\sigma k1'\Rightarrow Ck$[illegible]$d$, C[illegible]$d=\sigma k2'\Rightarrow Ck_{ol}$[illegible]$k2'\Rightarrow C$[illegible]$dded'$ and $Ck_{added}'=\mu\Rightarrow$. By Proposition 3, it [illegible]ows t[illegible]t $\sigma k'.\mu \in Tr(Ck_{old})$, [illegible] contradiction with our hypothesis. Consequently [illegible] $_{new}$), such that σk_σ $Tr(Ck_{ol}$[illegible]

b - 2 - 2 - 2: $\mu \in act(S_{old})$:

$\sigma k'.\mu \in Tr(Ck_{new})$, $\sigma k' \in Tr(Ck_o$[illegible]): By Proposit[illegible]n 3, w[illegible] have $\sigma k'$[illegible] $\in Tr(Ck_{old})$, or $\sigma k'.\mu \in Tr(Ck_{added})$, or [illegible]hat $\sigma k'=\sigma$[illegible]1[illegible]$k2$[illegible] Ck[illegible]$=\sigma k1'\Rightarrow$[illegible]$_{old}$, $Ck_{old}=\sigma k2'\Rightarrow Ck_{old}'$, $Ck_{added}=\sigma k2'\Rightarrow$[illegible]$k_{added}'$ and [illegible]$k_{added}'=\mu\Rightarrow$.

- $\sigma k'.\mu \in Tr(C$[illegible]$dded)$: we d[illegible]uce that $act(\sigma$[illegible]) [illegible]G_{ol}[illegible] H[illegible]$= \emptyset$, because $\sigma k' \in Tr(C$[illegible]) $act(Ck_{added})$ [illegible]$G_{old} = \emptyset$ a[illegible]d $act(Ck_{old})$ [illegible]$G_{added} =$ [illegible] (Condition (a)). We write $\sigma k' = a1.a2...an$, with $ai \in act(S_{old})$, for $i = 1, ..., n$. Because of Condition (b), for the distribution of actions over the basic components of S_{old} and S_{added}, and the fact that $\sigma \in Tr(S_{old})$, it [illegible]s that [illegible] $Tr(Ck_{old})$ such that $\sigma k_{old} = \sigma k_{old}'.\sigma k1_{old}''$, where $\sigma k_{old}'\backslash HG_{old} = \sigma k'.$[illegible]$= a1.a2...an$[illegible]. If $\sigma k_{old}'= a1.\ \sigma k_{old}'''$ with $\sigma k_{old}'''\backslash HG_{old} = a2...an.\mu$, the[illegible]$' = a2$[illegible] bec[illegible]of Cond[illegible] (d-1), we can not have hidden actions (from HG_{old}) [illegible]$\sigma k_{old}'''$. It f[illegible]lows that [illegible]$\mu \in Tr(Ck$[illegible]). If $\sigma k_{old}'= \sigma k1_{old}'.a1.\ \sigma k2_{old}'$ with $\sigma k2_{old}'\backslash HG_{old} = a2...an$[illegible]$k1_{old}'\backslash$[illegible]$=$, [illegible] an[illegible]d' is not cyclic in Ck_{old}, we are in contradiction wi[illegible] Condition[illegible]-2). If $\sigma k_{old}'= \sigma k1_{old}'.a1.\ \sigma k2_{old}'$ with $\sigma k2_{old}'\backslash HG_{old} =$[illegible]$...an.\mu$, $\sigma k1_{old}'\backslash HG_{old} =$, [illegible] an[illegible]d' is cyclic in Ck_{old}, it follows that $a1.\ \sigma k2_{o}$[illegible]$\in$ [illegible](Ck_{old}). Because of Condition (d-1), we can not have hidden actions (from HG_{old}) in $\sigma k2_{old}'$[illegible]nd σk[illegible]$ld' = a2...an.\mu$. It follows that $\sigma k'.\mu \in Tr(Ck_{old})$, which is in contradiction with our hypot[illegible]

- $\sigma k'.\mu$ $Tr(Ck_a$[illegible]$d)$: it follows that $\sigma k'= \sigma k1'.\sigma k2'$ [illegible]th $Ck_{old}=\sigma k1'\Rightarrow Ck_{old}$, $Ck_{old}=\sigma k2'\Rightarrow Ck_{old}'$, $Ck_{added}=\sigma k2'\Rightarrow Ck_{added}'$ and $Ck_{added}'=\mu\Rightarrow$. As above, b[illegible]ndition (b) for the distributio[illegible]ons over the basic components of S_{old} and S_{added}, and the [illegible]t that $\sigma \in Tr(S_{old})$, it follows t[illegible] $Tr(Ck_{old})$ such that $\sigma k_{old} = s.\mu.\sigma k1_{old}''$. if $\sigma k2' =$, σk[illegible]. By Condition (d-2), s [illegible]clic in Ck_{old} and $Ck_{old}=\mu\Rightarrow$. It follows that $Ck_{old}=\sigma k'\Rightarrow Ck_{old}$ and $Ck_{old}=\mu\Rightarrow$. We have [illegible]duced that $\sigma k'.\mu \in Tr(Ck_{old})$, which is in contradiction with our hypothesis. If $\sigma k2' \neq$, t[illegible]$2' =$

a1.a2...an, with ai $\in$ act(S_{old}), for i = 1, ..., n, since σk2' is a common trace for Ck_{old} and Ck_{added}. It follows that s = s1.a1.s2.μ.σ$k1_{old}$, such that s1\$HG_{old}$ = σk1'\$HG_{old}$ and s2\$HG_{old}$ = a2...an. We have Ck_{added}=a1⇒, and s1.a1 $\in$ Tr(Ck_{old}), by Condition (d-2), it follows that s1 is cyclic in Ck_{old} and a1 $\in$ Tr(Ck_{old}). We have a1.s2\$HG_{old}$.μ.= a1.a2...an.μ. By Condition (d-1), we can not have hidden action (from HG_{old}) in s2. It follows that s2 = a2...an. We have Ck_{old}=σk1'⇒C[illegible]ld, and σk2'.μ $\in$ Tr(Ck_{old}), it follows that σk'.μ [illegible]k$_{old}$), which is i[illegible]contradiction with o[illegible]thesis.

b - 2 - 2 - 3: μ $\in$ HG_{added}

We have σk'$\in$ Tr(Ck_{old}) and σk'≠ [illegible] (d-1), [illegible] that σk[illegible] [illegible]d). We have σk' $\in$ Tr(Ck_{old}), σk'.μ T[illegible]Ck[illegible]), σk' [illegible] Tr(C[illegible]ed) and σk'.[illegible]Tr(Ck_{new}). By Proposition 3, it follo[illegible]ith C[illegible]d=σk1'⇒Ck_{old} [illegible]k2'⇒Ck_{old}', Ck_{added}=σk2'⇒C[illegible] nd Ck$_{add}$[illegible]'=μ⇒. If σk2'≠ , w[illegible]k2'(≠) $\in$ [illegible]ld) and σk2'.μ Tr(Ck_{added}), we have re[illegible]hed a [illegible]o[illegible]adiction with Condi[illegible]n (d-1). If σk2'= , it [illegible] that μ $\in$ Tr(Ck_{added}) and [illegible]w), such that σm_σ = [illegible]m'.μ.σm''. Now, we are in the same situation as case b [illegible] h is solved recurs[illegible]ly an[illegible]eaches a contradiction in all cases. We can not have μ $\in$ H[illegible]

Conseq[illegible] such that σk_σ Tr(Ck$_{ol}$[illegible]
By the alg[illegible]roposition 1) [illegible]e have Ci_{new} ext C[illegible], for i = 1, ..., n.
It follows that, for i [illegible]..., n, [illegible] σi $\in$ T[illegible]Ci_{old}) and A a[illegible]
if [illegible]ne[illegible]≠a⇒, ∀ a $\in$ A,
then [illegible]⇒Ci_{old} [illegible]a⇒, ∀ a $\in$ A,
σ $\in$ Comp(new, σ1_σ, ..., σn_σ)[illegible]nd for i [illegible] 1, ..., n, σi_σ $\in$ Tr(Ci_{old}), we deduce that σ $\in$ Comp(old, σ1_σ, ..., σn_σ). Since [illegible] Ci_{old} =σi_σ⇒Ci_{old}' ≠a⇒, ∀ a $\in$ A, for i =1 , ..., n, it follows that [illegible]ld =σ⇒S_{old}' ≠a⇒, ∀ a $\in$ A.

Proo[illegible]n 2

Consider σ $\in$ Tr([illegible]ld), such that σ is a[illegible]c trace in S_{old}. It follows that, for i = 1, ..., n, [illegible]uch that σ $\in$ Comp([illegible] ..., σn). Assume that for i = 1, ..., n, σi is a minimal cyclic trace in Ci_{old}, and (σi Tr(Ci$_{add}$[illegible]) or σi is a cyclic trace in Ci_{added}). From Proposition 2, it follows that, for i = 1, ..., n, σi is a minimal cyclic trace in Ci_{new}. By Condition (a) of Theorem 1, we know that σ $\in$ Comp(new, σ1, σ2, ..., σn). Since the initial state of the structured specification S_{new} is composed by the initial states of all its components, we deduce that σ is a cyclic trace in S_{new}.

The proof for the second part of the theorem is similar.