Merging Behavior 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

Abstract

This paper describes a method for merging behavior specifications modeled by transition systems. Given two behavior specifications B_1 and B_2, Merge(B_1, B_2) defines a new behavior specification that extends B_1 and B_2. Moreover, provided that a necessary and sufficient condition holds, Merge(B_1, B_2) is a cyclic extension of B_1 and B_2. In other words, Merge(B_1, B_2) extends B_1 and B_2, and any cyclic trace in B_1 or B_2 remains a cyclic in Merge(B_1, B_2). Therefore, in the case of cyclic traces of B_1 or B_2, Merge(B_1, B_2) transforms into Merge(B_1, B_2), and may exhibit, in a recursive manner, behaviors of B_1 and B_2. If Merge(B_1, B_2) is a cyclic extension of B_1 and B_2, then Merge(B_1, B_2) represents the least common cyclic extension of B_1 and B_2. This approach is useful for the extension and integration of system specifications.

1 Introduction

Formal specifications play an important role in the development life cycle of systems. They capture the user requirements. They can be validated against such requirements and used as basis for the design of implementations and test suites. A formal specification represents the reference in each step of the development life cycle of the required system. The design and the verification of the specification of a system is a very complex task. Therefore, methodologies for the design of formal specifications become very important.

Systems may consist of many distinct functions. During the design and the validation of the specification, these functions may be taken into consideration simultaneously. The validation of such specification may be a very complex task. In order to facilitate the design and validation of the

[*] 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.

specification of a multiple-functions system, the divide-and-conquer approach may be very useful. In this case, a specification for each function is designed and analyzed separately. These specifications are then combined to form the required system specification. The combination of these functions specifications should preserve the semantic properties of every single function specification.

From another point of view, system specifications may be enriched by adding new behaviors required by the user, such as adding new functionality to a given system specification. Different system specifications may be integrated. In both cases, the semantic properties of the given system specifications and behaviors should be preserved. Preserving semantic properties may, for instance, mean that the combined specification exhibits at least the behavior of each single specification without introducing additional failures for these behaviors. This is captured by the formal relation between specifications, called extension, introduced in [Brin 86]. Informally, a behavior specification B_2 extends a behavior specification B_1, if and only if, B_2 allows any sequence of actions that B_1 allows, and B_2 can only refuse what B_1 can refuse, after a given sequence of actions allowed by B_1.

Given two behavior specifications B_1 and B_2, we may combine them into a new behavior specification B, such that B extends B_1 and B extends B_2. By definition of the extension relation, B may exhibits behaviors of B_1 (respectively B_2), without any new failure for these behaviors. However, B may exhibits behaviors of B_1 and behaviors of B_2, in an exclusive manner. In other words, B may exhibits only behaviors of B_1 or only behaviors of B_2, once the environment has chosen a behavior of B_1 or a behavior of B_2, respectively.

A behavior specification B may contain certain sequences of actions that may be repeated recursively. Such sequences of actions start from the initial state of B and reach the initial state of B_1. They are called cyclic sequences of actions. We assume that the completion of a cyclic sequence of actions in B corresponds to the completion of B. In other words, we assume that the initial state of B represents the "final" state for the sequences of actions (functionalities) in B. We are interested in combining two behavior specifications B_1 and B_2 into a new specification B, such that, in the case of cyclic sequences of actions of B_1 or B_2, B may exhibit, without any new failure, behaviors in B_1 and behaviors in B_2, in a recursive manner. In other words, B extends B_1 and B_2, and after a cyclic sequence of actions of B_1 or B_2, B transforms into B', with B' extends B_1 and B_2, and after a cyclic sequence of actions of B_1 or B_2, B' transforms into B", with B" extends B_1 and B_2, and so on. This is possible, if B extends B_1 and B_2, and any cyclic sequence of actions in B_1 or B_2

remains cyclic in B. Therefore, after a cyclic sequence of actions of B_1 or B_2, B transforms into B, which extends B_1 and B_2. This new relation between behaviors is called cyclic extension.

In this paper, we describe a formal approach for merging behavior specifications modeled by transition systems. Given two behavior specifications B_1 and B_2, we define a new specification behavior, called Merge(B_1, B_2), which extends B_1 and B_2. Moreover, provided that a necessary and sufficient condition holds, Merge(B_1, B_2) is the least common cyclic extension of B_1 and B_2.

We consider two models of transition systems, the Acceptance Graphs (AGs), which are similar to the Acceptance Trees of Hennessy [Henn 85] and the Tgraphs in [Clea 93], and the Labelled Transition Systems (LTSs) [Kell 76]. The merging of behavior specifications is, first, defined in the AGs model, which is more tractable mathematically than the LTSs model. The merging of LTSs is based on the merging of AGs and relies on a correspondence between LTSs and AGs, which is introduced in this paper.

The remainder of this paper is structured as follows. The next section introduces the LTSs model, some related equivalence relations and preorders and the notions of least common extension and least common cyclic extension. Section 3 introduces the AGs model, the related equivalences and preorders, the notions of least common extension and least common cyclic extension for AGs, and the correspondence between AGs and LTSs. The merging of two AGs G_1 and G_2, Merge(G_1, G_2), is defined in Section 4. Main properties of Merge are listed and an example of application is also provided in Section 4. In Section 5, the merging of LTSs is defined, as well as its properties and an example of application. In Section 6, our approach is compared to the related ones. In Section 7, we conclude. The proofs of the propositions and the theorem stated in this paper are provided in the Appendix.

2 Labelled Transition Systems

2.1 Model

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 S is a quadruple <St, L, T, s_o>, where

- St is a (countable), non-empty set of states.
- L is a (countable) set of observable actions.
- T ⊆ St x (L ∪ {τ}) x St [illegible] transitions, where a transition from a state s_i to state s_j by an action μ ($\mu \in$ L ∪ {τ}) is denoted by $s_i-\mu\rightarrow s_j$. τ represents the internal, nonobservable action ($\tau \notin$ L).
- s_o is the initial state.

An LTS S = <St, L, T, s_o> represents a process interacting, in a synchronous manner, with the environment by executing the actions in L ∪ {τ} following the rules specified by T. More exactly S represents a set of processes. Each state s_i of S corresponds to a process P represented by the LTS <St, L, T, s_i>. In the following, we use the terms process and state as synonyms. We also may refer to an LTS by its initial state. All the definitions on the states are extended to LTSs and processes. The term "interaction" refers to an observable action.

A finite LTS (FLTS for short) is an LTS in which St and L are finite. For the graphic representation of the FLTSs, the initial state will be circled. The notations in Table 1 are used for the LTSs.

$P-\mu_1...\mu_n\rightarrow Q$	$\exists$ P_i ($0 \le i \le n$) such that $P = P_o-\mu_1\rightarrow P_1...P_{n-1}-\mu_n\rightarrow P_n = Q$
$P-\mu_1...\mu_n\rightarrow$	$\exists$ Q such that $P-\mu_1...\mu_n\rightarrow Q$
$P-,/\mu_1...\mu_n\rightarrow$	not ($P-\mu_1...\mu_n\rightarrow$)
$P=\varepsilon\Rightarrow Q$	$P = Q$ or $\exists$ n 1 $P-\tau^n\rightarrow Q$
$P=a\Rightarrow Q$	$\exists P_1, P_2$ such that $P=\varepsilon\Rightarrow P_1-a\rightarrow P_2=\varepsilon\Rightarrow Q$
$P=a_1.a_2..\,a_n\Rightarrow Q$	$\exists$ P_i ($0 \le i \le n$) such that $P = P_o=a_1\Rightarrow P_1=a_2\Rightarrow..a_n\Rightarrow P_n = Q$
$P=\sigma\Rightarrow$	$\exists$Q such that $P=\sigma\Rightarrow Q$
$P\ \sigma\Rightarrow$	not ($P=\sigma\Rightarrow$)
Tr(P)	{$\sigma \in L^*$ | $P=\sigma\Rightarrow$}
out(P)	{$a \in L$ | $P=a\Rightarrow$ }

Notations:
μ, $\mu_i \in$ L ∪ {τ}; 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 events or sequence of events (traces).

Table 1. Notations for LTSs

For a given LTS S = <St, L, T, s_o>, a trace from a given state s_i, is a sequence of interactions that S can perform starting from state s_i. The traces that S can perform from its initial state represent the traces of S. s_i after σ (= {s_j | $s_i=\sigma\Rightarrow s_j$}) denotes the set of all states reachable from s_i by sequence σ. out(s_i, σ) (= $\bigcup_{s_j \in (s_i \text{ after } \sigma)}$ out(s_j)) denotes the set of all possible interactions after σ, starting from state s_i. A trace of S is cyclic, if and only if the set of states reachable by this trace is equal to the set of states reachable by the empty trace from the initial state. An elementary cyclic trace is a cyclic trace that is not prefixed by a nonempty cyclic trace. Note that, any cyclic trace results from the concatenation of elementary cyclic traces.

Definition 2.2 (Cyclic T[redacted]ce for LTS[redacted])
Given an LTS S = <St, L, T[redacted]>, a trace σ[redacted] a cyclic trace in S, iff
(s_o after σ) = {s_i ∈ St such t[redacted]t s_o−ε⇒s_i}.

Definition 2.3 (Elementa[redacted] Cyclic Tr[redacted]e for LTSs)
Given an LTS S = <St, L, T[redacted]$_o$>, a trace σ[redacted] an elementary cyclic trace in S, iff
(1) σ is a cyclic trace, and [redacted]
(2) σ' [redacted] and σ' is a cyclic trace in S.

2.2 Equivalences and P[redacted]orders

Intuitively, different LTSs may describe the same "observable behavior". Different equivalences have been defined corresponding to different notions of "observable behavior" [DeNi 87]. In the case of trace equivalence, two systems are considered equivalent if the set of all possible sequences (traces) of interactions that they may produce are the same.

Finer equivalences are obtained if the refusal (blocking) properties of the systems, which are in general non-deterministic, are also taken into account. P ref A means that P refuses to perform any interaction in A (P a⇒, ∀ a ∈ A). In other words, P deadlocks with any interaction a in A. A is called a refusal for P. Note that if A is a refusal for P, then any subset of A is a refusal for P.

Ref(P, σ) = {X | ∃ Q ∈ (P after σ) such that P ref X} denotes the refusal set of P after σ.
Note that if σ ∉ Tr(P), then Ref(P, σ) = ø.

[redacted]
Two systems are testing equiv[redacted]ace equivalence, they have the same refusal (blocking) properties [Brin 86].
[redacted]

Definition 2.4 (Testing Equivalence for LTSs)
Let S1 and S2 be two LTSs, S1 and S2 are testing equivalent, S1 te S2, iff
(1) Tr(S1)   Tr(S2), and
(2) ∀ σ ∈ L*, Ref(S2, σ)   Ref(S1, σ).

For instance, the LTSs S1, S2 and S3 in Figure 1 can perform the same sequences (**a**, **a.b**, **a.b.c**, **a.b.d**) of interactions (**a**, **b**, **c** and **d**). They have the same set of traces, they are trace equivalent. Moreover, the LTSs S1 and S2 have the same refusal properties. Because of nondeterminism, S1 and S2 may both refuse interaction **c** (respectively **d**) after the sequence of interactions **a.b**. S1 and S2 are not distinguishable by external experiences. They are testing equivalent. However, S3 is not testing equivalent to S1 (and S2). S3 always accept interaction **c** or **d**, after the sequence **a.b**.

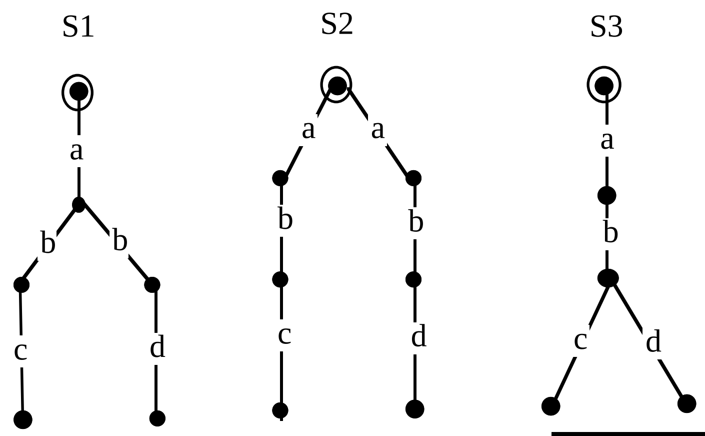

Figure 1. Examples for behavior equivalences.

Instead of considering the sets of interactions that may be refused, we may consider the sets of interactions that may be accepted. The notion of acceptance sets is dual to the notion of refusal sets. If Ref(P, σ) is a refusal set, then the corresponding acceptance set Acc(P, σ), is defined as the complement of the refusals in Ref(P, σ) with respect to out(P, σ).

Acc(P, σ) = {out(P, σ) - X | X ∈ Ref(P, σ)}
= {X | ∃Q ∈ (P after σ) such that out(Q) X out(P, σ)}.

The following properties of Acc(P, σ) can be derived from its definition:

- Acc(P, σ) = ∅ iff σ ∉ Tr(P),
- ∀ A1, A2 ∈ Acc(P, σ), A1 A2 ∈ Acc(P, σ),
- ∀ A1, A2 ∈ Acc(P, σ), if A1 A3 A2, then A3 ∈ Acc(P, σ) .

Intuitively, a set of interactions X belongs to Acc(P, σ), if and only if there is a state Q reachable from P by σ and X includes the set of interactions enabled in this state, but X is included in the set of all possible interactions of (P after σ). This definition corresponds to the acceptance sets definition in [Henn 85].

Condition (2) in Definition 2.4 may be stated in terms of acceptance sets as follows:
∀ σ ∈ L*, Acc(S2, σ) Acc(S1, σ).

Similar testing equivalence relations are defined in [Broo 85, DeNi 84, Henn 88]. They differ from the testing equivalence we consider in this paper, in the way the divergence (possibility of infinite sequence of internal actions) is dealt with.

Finer equivalences, the bisimulation equivalence (strong bisimulation, $\sim$) [Park 81] and the observation equivalence (weak bisimulation, $\approx$) [Miln 89], may be defined if the internal states of the two systems are taken into account. These relations are based on the notions of strong bisimulation [Park 81] and weak bisimulation [Miln 89], respectively.

Definition 2.5 (Strong Bisimulation)
A relation $R \subseteq St \times St$ is a strong bisimulation, if $(s_i, s_j) \in R$ implies that
$\forall a \in (L \cup \{\tau\})$, if $s_i\text{–}a\rightarrow s_k$ then $s_j\text{–}a\rightarrow s_l$ and $(s_k, s_l) \in R$,
if $s_j\text{–}a\rightarrow s_l$ then $s_i\text{–}a\rightarrow s_k$ and $(s_k, s_l) \in R$

Definition 2.6 (Weak Bisimulation)
A relation $R \subseteq St \times St$ is a weak bisimulation, if $(s_i, s_j) \in R$ implies that
$\forall a \in (L \cup \{\varepsilon\})$, if $s_i{=}a{\Rightarrow} s_k$ then $s_j{=}a{\Rightarrow} s_l$ and $(s_k, s_l) \in R$,
if $s_j{=}a{\Rightarrow} s_l$ then $s_i{=}a{\Rightarrow} s_k$ and $(s_k, s_l) \in R$

Two LTSs S1 and S2, with $s1_o$ and $s2_o$ as initial state, respectively, are (strongly) bisimulation equivalent, S1 $\sim$ S2, (respectively observation equivalent, S1 $\approx$ S2), if and only if there is a strong bisimulation R (respectively weak bisimulation R) with $(s1_o, s2_o) \in R$. The observation equivalence of Milner is stronger than the testing equivalence, but weaker than the bisimulation equivalence. Two LTSs S1 and S2, with $s1_o$ and $s2_o$ as initial state, respectively, are isomorphic, if and only if there is a strong bisimulation R, such that $(s1_o, s2_o) \in R$ and each state of S1 is related to one and only one state of S2 and vice et versa.

In addition to the equivalences, many preorders (reflexive and transitive relations) have been defined in the literature [DeNi 87, Henn 85, Brin 86]. The extension preorder defined in [Brin 86] is most appropriate for extending specification behaviors. Informally, S2 extends S1, S2 ext S1, if and only if S2 may perform any sequence of interactions that S1 may perform, and S2 can not refuse what S1 can not refuse after a given sequence of interactions allowed by S1 [Brin 86]. The extension preorder induces the testing equivalence [Brin 86]. In other words, two specifications are testing equivalent if and only if each is the extension of the other. In the following, for a given set X, *P*(X) denotes the power set of X, i.e. the set of subsets of X.

Definition 2.7
Let A, B P(L), B, iff ∀ A_1 ∈ A, ∃ B_1 ∈ B such that B_1 A_1.

The following definition of the extension introduced in [Ledu 90] is equivalent to the original one:

Definition 2.8 (Extension for LTSs)
Let S_1 and S_2 be two LTSs, S_2 ext S_1, iff
(1) Tr(S_1) Tr(S_2), and
(2) ∀ σ ∈ Tr(S_1), Acc(S_2, σ) Acc(S_1, σ).

For instance, the LTSs S_6 and S_7 in Figure 2 extend both of the LTSs S_4 and S_5. S_6 (and S_7) may perform any sequence of interactions that S_4 (respectively S_5) may perform and S_6 can not refuse what S_4 (respectively S_5) may not refuse after a sequence of interactions allowed by S_4 (respectively S_5). However, S_8 does neither extend S_3 nor S_4. Indeed, S_8 may perform any sequence of interactions that S_4 (respectively S_5) may perform, but S_8 may, for instance, refuse interaction **b** (respectively **c**) after sequence **a**, whereas S_4 (respectively S_5) never refuses to interaction **b** (respectively c) after sequence **a**.

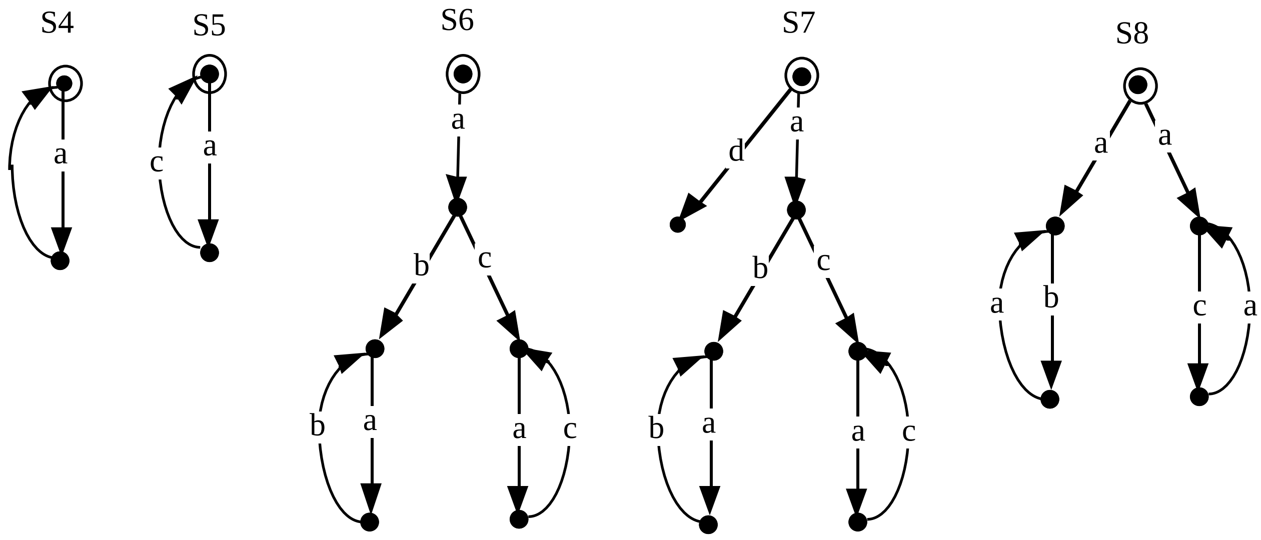

Figure 2. Extension of behaviors.

Among the common extensions of S_4 and S_5, S_6 is the least one. In other words, any common extension of S_4 and S_5 is an extension of S_6. For instance, S_7 extends S_6. The least common extension is unique up to testing equivalence.

Definition 2.9 (Least Common Extension for LTSs)

Given three LTSs S1, S2 and S3, such that S3 ext S1 and S3 ext S2,
S3 is the least common extension of S1 and S2, iff
any common extension of S1 and S2 is also an extension of S3.

As introduced previously, in this paper we assume that the completion of a cyclic sequence of interactions in a given specification S corresponds to the completion of S. For instance, after performing **a.b**, S4 has completed its functionality and may repeat it in a recursive manner. The LTS S6, in Figure 2, extends both S4 and S5. However, S6 may exhibit only behavior **a.b** of S4 in a recursive manner or only behavior **a.c** of S5 in a recursive manner. S6 does not exhibit behaviors of S4 and behaviors of S5, in a recursive manner, contrarily to the LTS S9 in Figure 3. Indeed S9 extends both S4 and S5 and after performing a cyclic sequence of interactions in S4 (respectively S5) S9 transforms into S9 and offers again behaviors of S4 and S5. S9 may exhibit the behaviors **a.b.a.b**...., **a.c.a.c**...., **a.b.a.c.a.b.a.c**, ... etc. A condition for S9 to transform into S9 after any cyclic trace of S4 or S5, is that any cyclic trace in S4 (respectively S5) is a cyclic trace in S9. In this case, S9 is called a cyclic extension of S4 (respectively S5).

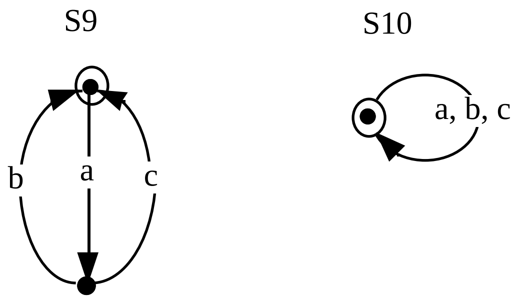

Figure 3. Cyclic extension of behaviors.

Definition 2.10 (Cyclic Extension for LTSs)
Let S1 and S2 be two LTSs. S2 is a cyclic extension of S1, S2 extc S1, iff
(1) S2 ext S1, and
(2) any cyclic trace in S1 is a cyclic trace in S2.

Since any cyclic trace results from the concatenation of elementary cyclic traces, any cyclic trace in S1 is a cyclic trace in S2, if and only if any elementary cyclic trace in S1 is a cyclic trace in S2. Among the common cyclic extensions of S4 and S5 shown in Figure 2, S9 shown in Figure 3 is the least one. In other words, any common cyclic extension of S4 and S5 is a cyclic extension of S9. For instance, S10, a cyclic extension of S4 and S5, is also a cyclic extension of S9. Note that the least common cyclic extension of S4 and S5, S9, extends the least common extension of S4 and S5, S6.

Definition 2.11 (Least Common Cyclic Extension for LTSs)
Given three LTSs S1, S2 and S3, such that S3 extc S1 and S3 extc S2,
S3 is the least common cyclic extension of S1 and S2, iff
any common cyclic extension of S1 and S2 is also a cyclic extension of S3.

The testing equivalence is refined into the cyclic testing equivalence, if the preservation of the cyclic traces is taken into account. Note that the cyclic extension is a preorder and it induces the cyclic testing equivalence.

Definition 2.12 (Cyclic Testing Equivalence for LTSs)
Let S1 and S2 be two LTSs. S2 and S1 are cyclic testing equivalent, S1 tec S2, iff
(1) S1 te S2, and
(2) any cyclic trace in S1 is a cyclic trace in S2 and reciprocally.

S1 and S2 have the same set of cyclic traces, as stated by condition (2) in Definition 2.12, if and only if S1 and S2 have the same set of elementary cyclic traces, since the concatenation of elementary cyclic traces leads a cyclic trace. Similarly to the testing equivalence, the strong bisimulation and the observation equivalence are also refined into the cyclic strong bisimulation (c) and the cyclic observation equivalence (≈c), respectively, when the preservation of the cyclic traces is taken into consideration.

3 Acceptance Graphs

3.1 Model

An AG is a bilabelled graph-structure. An AG is a graph in which nodes represent states, and transitions represent interactions occurring during state changes. Instead of modeling the nondeterminism by the labels of the transitions, the AGs model allows to keep such information in the labels of the states. Each state is labelled by a set of sets of interactions, called acceptance set, that the system may accept (perform) at this state. The outgoing transitions, from a given state, have distinct labels.

Definition 3.1 (Acceptance Graph)
An AG G is 5-tuple <Sg, L, Ac, Tg, g_o>, where
- Sg is a (countable) non empty set of states.
- L is a (countable) set of interactions.

- Ac: Sg $\rightarrow$ *P*(*P*(L)) is a mapping from Sg to a set of subsets of L.
  Ac(g_i) is called the acceptance set of state g_i.
- Tg: Sg x L$\rightarrow$Sg is a transition function, where a transition from state g_i to state g_j by an interaction a (a $\in$ L) is denoted by g_i–a$\rightarrow g_j$.
- g_o is the initial state.

The AGs used in this paper are similar to the Acceptance Trees of Hennessy [Henn 85] and Agraphs in [Clea 93]. However, in our case, we do not distinguish between "closed" and "open" states, since divergence is not considered explicitly as in [Henn 85] or [Clea 93]. In this paper, any state g_i is labelled by an acceptance set, Ac(g_i), which may be infinite or contain some infinite elements in the case where g_i is infinitely branching ({g_j | g_i–a$\rightarrow g_j$ for some a $\in$ L} is infinite). The mapping Ac and the transition function Tg should satisfy the following consistency constraints, which are similar to the consistency constraints defined for the "closed" states in [Henn 85]:

C0: $\forall$ $g_i \in$ Sg, Ac(g_i)   Ø.
C1: $\forall$ $g_i \in$ Sg, A $\in$ Ac(g_i) and a $\in$ A, there is one and only one $g_j \in$ Sg such that g_i–a$\rightarrow g_j$.
C2: $\forall$ $g_i \in$ Sg, if $\exists$ $g_j \in$ Sg, such that g_i–a$\rightarrow g_j$, then $\exists$ A [illegible]
C3: $\forall$ $g_i \in$ Sg, if A1, A2$\in$ Ac(g_i), then A1 [illegible] A2 $\in$ Ac(g_i).
C4: $\forall$ $g_i \in$ Sg, if A1, A2 $\in$ Ac(g_i) and A1 [illegible] A3 [illegible] A2, then A3 $\in$ Ac(g_i).

A finite AG (FAG for short) is an AG in which Sg and L are finite. As for the LTSs, the initial state will be circled for the graphic representation of an FAG. The notations introduced in Table 1 will be used for the AGs with the same meaning as for the LTSs, since leaving the mapping Ac out of account, an AG can be seen as an LTS. In the case of AGs, the notation "g_i after σ" will denote the state g_j such that g_i=σ⇒g_j, instead of set of states in the case of LTSs. The notion of cyclic trace for AGs corres[illegible]f cyclic path in the graph theory. A cyclic trace is a trace, of the initial state, that reaches the initial state. Similarly to the LTSs, an elementary cyclic trace, is a cyclic trace, which does not result from the concatenation of cyclic subtraces. Any cyclic trace results from the concatenation of elementary cyclic traces.

Definition 3.2 (Cyclic Trace for AGs)
Given an AG G = <Sg, L, Ac, Tg, g_o>, a trace σ is a cyclic trace in G iff g_o=σ⇒g_o.

Definition 3.3 (Elementary Cyclic Trace for AGs)
Given an AG G = <Sg, L, Ac, Tg, g_o>, a trace σ is an elementary cyclic trace in G, iff
(1) σ is a cyclic trace, and
(2) σ'([illegible] and σ' is cyclic trace in G.

An AG G may contain certain states that are not reachable (A state g_i is reachable iff $\exists\ \sigma \in$ Tr(G) such that g_o=σ⇒g_i.). The graph defined by the set of reachable states, their acceptance sets and their transitions as defined in G, denoted by reachable(G), is an AG. It is obvious that reachable(G) satisfies all the consistency requirements listed above.

Definition 3.4 (Reachable Part of an AG)
Given an AG G = <Sg, L, Ac, Tg, g_o>, the reachable part of G, reachable(G), is an AG G' = <Sg', L, Ac', Tg', g_o>, where

- Sg' = {$g_i \in$ Sg | $\exists\ \sigma \in$ Tr(G) such g_o=σ⇒g_i}
- $\forall\ g_i \in$ Sg', Ac'(g_i) = Ac(g_i),
- $\forall\ g_i$, $g_j \in$ Sg', g_i–a→$g_j \in$ Tg' iff g_i–a→$g_j \in$ Tg.

3.2 Equivalences and preorders

Similarly to the LTSs, in the case of trace equivalence, two AGs G1 and G2 are considered equivalent, if and only if Tr(G1) = Tr(G2). However, in the case of AGs, the testing equivalence and the observation equivalence coincide with the bisimulation equivalence. The LTS's structure is finer than the AG's structure. In this paper, we define the bisimulation for AGs as an instantiation of the Π-bisimulation introduced in [Clea 93].

Definition 3.5 (Bisimulation)
A relation R $\subseteq$ Sg x Sg is a bisimulation, if (g_i, g_j) $\in$ R implies that

Ac(g_i) = Ac(g_j), and $\forall$ a $\in$ L,
if g_i–a→g_k then g_j–a→g_l and (g_k, g_l) $\in$ R,
if g_j–a→g_l then g_i–a→g_k and (g_k, g_l) $\in$ R

Definition 3.6
Two AGs G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, Ac2, Tg2, $g2_o$> are bisimulation equivalent, G1 $\approx_g$ G2, if and only if there is a bisimulation R such that ($g1_o$, $g2_o$) $\in$ R.

An alternative definition of the bisimulation equivalence for AGs is given by Proposition 3.1.

Proposition 3.1
Given two AGs Gi = <Sgi, Li, Aci, Tgi, gi_o>, i = 1, 2; G1 $\approx_g$ G2 iff
Tr(G1) = Tr(G2) and ($\forall\ \sigma \in$ Tr(G1), Ac1($g1_o$ after σ) = Ac2($g2_o$ after σ)).

Two AGs G1 and G2, with $g1_o$ and $g2_o$ as initial state, respectively, are isomorphic, G1 $=_g$ G2, if and only if there is a bisimulation R, such that $(g1_o, g2_o) \in$ R and each state of G1 is related to one and only one state of G2 and vice et versa.

Similarly to the LTSs, the extension relation is defined as follows:

Definition 3.7 (Extension for AGs)
Let G1 and G2 be two AGs,
G2 extends G1, G2 ext_g G1, iff
(1) Tr(G1) $\subseteq$ Tr(G2), and
(2) $\forall\, \sigma \in$ Tr(G1), Ac2($g2_o$ after σ) $\subset\subset$ Ac1($g1_o$ after σ).

In the case of AGs, the extension is a preorder that induces the bisimulation equivalence. From Proposition 3.1 and Definition 3.6, it is obvious that if G2 ext_g G1 and G1 ext_g G2, then G1 $\approx_g$ G2. If we take into consideration the preservation of the cyclic traces, the extension and the bisimulation equivalence are refined into the cyclic extension and the cyclic bisimulation equivalence. Note that the cyclic extension preorder induces the cyclic bisimulation equivalence. Similarly to the LTSs, the cyclic traces of a given AG are preserved, if and only if its elementary cyclic traces are preserved, at least, as cyclic traces. Two AGs have the same set of cyclic traces, if and only if they have the same set of elementary cyclic traces.

Definition 3.8 (Cyclic Extension for AGs)
Let G1 and G2 be two AGs,
G2 is a cyclic extension of G1, written G2 $extc_g$ G1, iff
(1) G2 ext_g G1, and
(2) any cyclic trace in G1 is a cyclic trace in G2.

Definition 3.9 (Cyclic Bisimulation for AGs)
Let G1 and G2 be two AGs,
G2 and G1 are cyclic bisimulation equivalent, written G1 $\approx c_g$ G2, iff
(1) G1 $\approx_g$ G2, and
(2) any cyclic trace in G1 is a cyclic trace in G2 and reciprocally.

The notions of least common extension and least common cyclic extension for AGs are defined in a similar way as for LTSs.

Definition 3.10 (Least Common Extension)
Given three AGs G_1, G_2 and G_3, such that G_3 ext_g G_1 and G_3 ext_g G_2,
G_3 is the least common extension of G_1 and G_2, iff
any common extension of G_1 and G_2 is also an extension of G_3.

Definition 3.11 (Least Common Cyclic Extension)
Given three AGs G_1, G_2 and G_3, such that G_3 extc_g G_1 and G_3 extc_g G_2,
G_3 is the least common cyclic extension of G_1 and G_2, iff
any common cyclic extension of G_1 and G_2 is also a cyclic extension of G_3.

3.3 Correspondence and transformations between AGs and LTSs

This section aims to define a correspondence between the LTSs and the AGs as well as the constructions for generating AGs from arbitrary LTSs and vice et versa. The correspondence between LTSs and AGs is based on the preservation of the traces, the acceptance sets and the cyclic traces.

Definition 3.12 (Correspondence between LTSs and AGs)
Given an LTS S = <St, L, T, s_o> and an AG G = <Sg, L, Ac, Tg, g_o>,
we say that G is the AG corresponding to S, G = ag(S), iff
(1) Tr(S) = Tr(G),
(2) $\forall \sigma \in$ Tr(G), Ac(g_o after σ) = Acc(s_o, σ),
(3) any cyclic trace in S is a cyclic trace in G, and
(4) any cyclic trace in G is a cyclic trace in S.

Note that, for a given LTS, the corresponding AG is unique up to the cyclic bisimulation equivalence. However, An AG may correspond to more than one LTS. These LTSs are cyclic testing equivalent. The following proposition is straightforward.

Proposition 3.2
Given two LTSs S_1, S_2, and two AGs G_1, G_2,
such that G_1 = ag(S_1) and G_2 = ag(S_2), the following holds:
(1) S_2 ext S_1 iff G_2 ext_g G_1.
(2) any cyclic trace in S_1 is a cyclic trace in S_2 iff
any cyclic trace in G_1 is a cyclic trace in G_2.

Lemma 3.1 follows from Proposition 3.2, since the extension (respectively, the cyclic extension) induces the testing equivalence (respectively, the cyclic testing equivalence) in the case of LTSs and the bisimulation equivalence (respectively, the cyclic bisimulation equivalence) in the case of AGs.

Lemma 3.1
Given two LTSs S_1, S_2, and two AGs G_1, G_2,
such that $G_1 = ag(S_1)$ and $G_2 = ag(S_2)$, the following holds:

(1) S_1 te S_2 iff $G_1 \;\; _g \, G_2$,

(2) S_1 tec S_2 iff $G_1 \;\; c_g \, G_2$.

Lemma 3.2 follows from Proposition 3.2 and the definitions of least common extension and least common cyclic extension for LTS and AGs, respectively.

Lemma 3.2
Given three LTSs S_1, S_2, S_3 and three AGs G_1, G_2, G_3, such that $G_1 = ag(S_1)$, $G_2 = ag(S_2)$ and $G_3 = ag(S_3)$, the following holds:

(1) S_3 is the least common extension of S_1 and S_2, iff
G_3 is the least common extension of G_1 and G_2.

(2) S_3 is the least common cyclic extension of S_1 and S_2, iff
G_3 is the least common cyclic extension of G_1 and G_2.

In the following proposition we define for an arbitrary LTS the corresponding AG. The definition of the corresponding AG for an arbitrary LTS is similar to the construction of a Tgraph from an arbitrary LTS in [Clea 93].

Definition 3.13 (ε-closure of a set of states) [Clea 93]
Given an LTS $S = <St, L, T, s_o>$, the ε-closure of a set of states $Qt \in P(St)$, written Qt^{ε},
is defined as follows: $Qt^{\varepsilon} = \{s_j \in St \mid \exists\, s_i \in Qt$ such that $s_i {=}\varepsilon{\Rightarrow} s_j\}$.

Proposition 3.3 (Definition of the AG corresponding to an arbitrary LTS)
Given an LTS $S = <St, L, T, s_o>$, the following AG G is such that $G = ag(S)$:
$G = <Sg, L, Ac, Tg, g_o>$, where

(1) $Sg = \{g_i \in P(St) \mid g_i = g_i^{\varepsilon}\}$,

(2) $g_o = \{s_i \in St \mid s_o{=}\varepsilon{\Rightarrow}s_i\}(= \{s_i \in St \mid s_o{=}\varepsilon{\Rightarrow}s_i\}^{\varepsilon}) \in Sg$,

(3) $\forall\, g_i \in Sg$, $Ac(g_i) = \{X \mid \exists\, s_j \in g_i$ such that $out(s_j) \quad X \quad _{s_k \in g_i} out(s_k)\}$.

(4) $\forall\ g_i \in Sg$, we have $g_i–a \rightarrow g_j$, iff

$a \in A$, $A \in Ac(g_i)$ and $g_j = \{s_k \in St$ such that [redacted] $s_j \in g_i$ with $s_j–a \rightarrow s_k\}^{\varepsilon}$.

An arbitrary AG G corresponds t[redacted]valent LTSs. However, by Proposition 3.4, for an arbitrary AG G, we define a sp[redacted]ial LTS S, w[redacted]tten lts(G), corresponding to G. For that, each state of G is split into a set of S [redacted]tates as show[redacted] in Figure 4. For each non redundant set of interactions A_{ij} [redacted] the acceptance set [redacted] a st[redacted]e g_i in G [redacted]orresponds a state s_{Aij} in St. By a non redundant set of interactions, we denote a s[redacted]her sets of interactions in the [redacted]cceptance set [redacted]nor it includes a [redacted]cceptanc[redacted]et and is included in another one. The correspo[redacted]ing S states, [redacted]or a given G state, a[redacted]ed as fo[redacted]ws:

Definition 3.13 [redacted]**LTS states** [redacted]**orrespond**[redacted]**g t**[redacted] **state)**

Given an AG G = [redacted]Sg, L, Ac, [redacted], g_o> and a sta[redacted] g_i [redacted]e states [redacted]rresponding to g_i in an LTS corresponding to [redacted]re defined [redacted]s follows:

$f(g_i) = \{s_{Aij} \mid Aij$ [redacted]$Ac(g_i)$, and [redacted]

such that [redacted]$ij = Aik$ [redacted] or Ai[redacted]$\}$

Proposition 3.4 [redacted]**Definition** [redacted]**f lts(G) for an arb**[redacted]**ry AG G)**

Given an AG G = [redacted] L, Ac, [redacted]g, g_o>,

the following LTS S, w[redacted]en lts([redacted]), is such that [redacted] ag(S):

S = <St, L, T, s_o>, where

(1) St = $_{g_i \in Sg}$ $(f(g_i)$ [redacted]

(2) $s_i–\tau \rightarrow s_{Aij}$, for each $s_{Aij} \in f(g_i)$, for each s_i in St (see Figure 4),

(3) For each transition $g_i–a \rightarrow g_k$ in G, for each $s_{Aij} \in f(g_i)$, with $a \in A_{ij}$,
there is a transition $s_{Aij}–a \rightarrow s_k$ in S (see Figure 4).

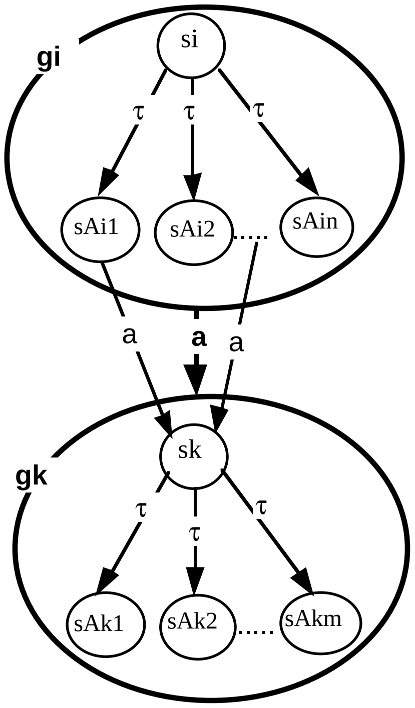

Figure 4. Transformation of the AG G into lts(G).

By definition, for an arbitrary AG G, lts(G) is unique. Due to the special form of LTSs defined by Proposition 3.4, two AGs G_1 and G_2 are (cyclic) bisimulation equivalent, if and only if lts(G_1) and lts(G_2) are (cyclic) strong bisimulation equivalent. Moreover, due to the correspondence between states of an G_1 (resp[redacted] of lts(G_1) [redacted]espectively lts(G_2), G_1 and G_2 are isomorphic, if and [redacted]

Proposition 3.[redacted]

Given two AGs G_1, G_2, and two [redacted]

such that S_1 = lts(G_1) and S_2 = lts(G_2), the following holds:

(1) S_1 S_2 iff G_1 ${}_g$ G_2,

(2) S_1 c S_2 iff G_1 c_g G_2,

(3) lts(G_1) = lts(G_2) iff G_1 ${}_g$ G_2.

For this special form of LTSs, defined in Proposition 3.4, the (cyclic) testing, (cyclic) observation and (cyclic) bisimulation equivalences coincide. Lemma 3.3 follows directly from the facts that G_1 = ag(lts(G_1)), G_2 = ag(lts(G_2)), Lemma 3.1 and Proposition 3.5.

Lemma 3.3

Given two AGs, G_1 and G_2,

(1) the following statements are equivalent:
lts(G1) te lts(G2), lts(G1) ≈ lts(G2), lts(G1) lts(G2), G1 $_g$ G2.

(2) the following statements are equivalent:
lts(G1) tec lts(G2), lts(G1) ≈c lts(G2), lts(G1) c lts(G2) , G1 c_g G2.

Note that similar correspondence between LTSs and Tgraphs is used in [Clea 93] in order to verify the testing equivalence relation between LTSs as defined in [Henn 88] by verifying the bisimulation equivalence between the corresponding Tgraphs. Drira has used similar correspondence between LTS and Refusal Graphs for the same purpose as in [Clea 93]. He also defined a special form of LTSs, called normal form, and proved that the testing, observation and bisimulation equivalences coincide for these LTSs, as we have done in the first part of Lemma 3.3. The form of the LTSs defined by Proposition 3.4 is similar to the normal form defined in [Drir 92], except that in our case each state has, in an exclusive manner, transitions labelled by the silent action or transitions labelled by interactions, whereas in [Drir 92] a state may have both kind of transitions.

4 Merging Acceptance Graphs

In this section, we define the merging of AGs. The AGs are more tractable mathematically than the LTSs, because the outgoing transitions, from a given state, have distinct labels. Given two AGs G1 and G2, we define an operation Merge, such that Merge(G1, G2) extends G1 and G2. Moreover, provided that a necessary and sufficient condition holds, Merge(G1, G2) is the least common cyclic extension of G1 and G2. The main properties of this Merge operation are described and an algorithm for the construction of Merge(G1, G2) in the case of FAGs as well as an example of application are given.

4.1 Definition and Properties of the Merge operation

Informally, given two AGs G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>, we define Merge(G1, G2) to be the reachable part of a graph in which a state g_i is either a pair <$g1_i$, $g2_j$> consisting of a state $g1_i$ from Sg1 and a state $g2_j$ from Sg2 (for instance, the initial state <$g1_o$, $g2_o$>), or a simple state $g1_i$ from Sg1, or a simple state $g2_j$ from Sg2.

The definition of the transitions from a state <$g1_i$, $g2_j$> in Merge(G1, G2) depends on the transitions from $g1_i$ in G1 and from $g2_j$ in G2. For instance, the transition <$g1_i$, $g2_j$>–a→<$g1_k$, $g2_m$> is defined in Merge(G1, G2), if and only if there is a transition $g1_i$–a→$g1_k$ in G1 and a transition $g2_j$–a→$g2_m$ in G2. A transition <$g1_i$, $g2_j$>–a→$g1_k$ is defined in Merge(G1, G2), if and only if there exist a transition $g1_i$–a→$g1_k$ in G1, but there is no transition labeled by a from $g2_j$ in G2. The transitions from a simple state in Merge(G1, G2), such as $g1_k$ for instance, remain the same as defined in G1 or G2 except for the transitions that reach the initial states of G1 or G2, which are replaced by corresponding transitions that reach the initial state <$g1_o$, $g2_o$> of Merge(G1, G2). A complete definition is as follows:

Definition 4.1 (Merge)

Given two AGs, G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and
G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>,
Merge(G1, G2) = reachable(<Sg3, L1 L2, Ac3, Tg3, <$g1_o$, $g2_o$>>), where

(1) Sg3 = {<$g1_i$, $g2_k$> | $g1_i \in$ Sg1 and $g2_k \in$ Sg2} Sg1 Sg2

(2) The mapping Ac3 is defined as follows: For each state g_i in Sg3,
  if g_i = <$g1_i$, $g2_j$>, then Ac3(g_i) = {X1 X2 | X1 $\in$ Ac1($g1_i$) and X2 $\in$ Ac2($g2_j$)},
  if $g_i \in$ Sgx, then Ac3(g_i) = Acx(g_i), where x = 1, 2.

(3) For each state <$g1_j$, $g2_k$> in Sg3,
  3-1. <$g1_j$, $g2_k$>–a→<$g1_l$, $g2_m$> $\in$ Tg3 iff $g1_j$–a→$g1_l$ $\in$ Tg1 and $g2_k$–a→$g2_m$ $\in$ Tg2.
  3-2. <$g1_j$, $g2_k$>–a→<$g1_o$, $g2_o$> $\in$ Tg3 iff ($g1_j$–a→$g1_o$ $\in$ Tg1 and $g2_k$–,/a→ in Tg2)
   or ($g1_j$–,/a→ in Tg1 and $g2_k$–a→$g2_o$ $\in$ Tg2).
  3-3. <$g1_j$, $g2_k$>–a→$g1_l$ $\in$ Tg3 iff $g1_j$–a→$g1_l$ $\in$ Tg1, $g1_l$ $g1_o$, and $g2_k$–,/a→ in Tg2.
  3-4. <$g1_j$, $g2_k$>–a→$g2_m$ $\in$ Tg3 iff $g2_k$–a→$g2_m$ $\in$ Tg2, $g2_m$ $g2_o$, and $g1_j$–,/a→ in Tg1.

(4) For each state gx_j in Sg3, where x = 1, 2,
  4-1. gx_j–a→<$g1_o$, $g2_o$> $\in$ Tg3 iff gx_j–a→gx_o $\in$ Tgx.
  4-2. gx_j–a→gx_l $\in$ Tg3 iff gx_j–a→gx_l $\in$ Tgx, gx_l gx_o.

If we consider, for instance, the AGs G1 and G2 shown in Figure 5, Merge(G1, G2) is described by the reachable part (in bold) of G.

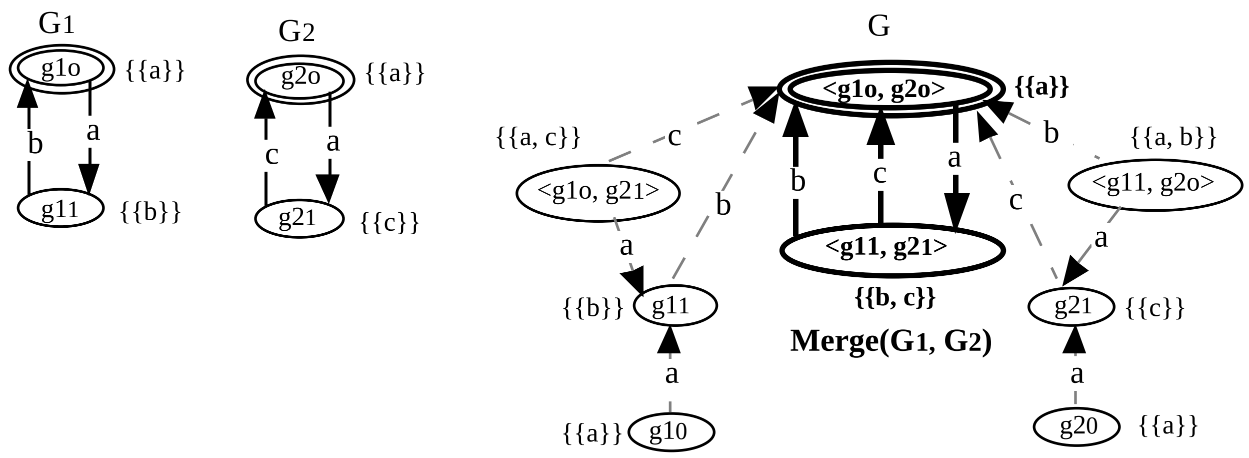

Figure 5. Example of Merge.

Merge(G1, G2) defines an AG. The consistency constraints defined in Section 3.1 are satisfied by Merge(G1, G2) as stated by Proposition 4.1 below. Stated otherwise, given two AGs G1 and G2, Merge(G1, G2), always exists.

Proposition 4.1
Given two AGs, G1 and G2, Merge(G1, G2) is an AG.

The operation Merge is commutative and associative. Therefore, AGs may be combined in an incremental way and in any order.

Proposition 4.2
Given three AGs, G1, G2 and G3, the following holds:

(a) Merge(G1, G2) $=_g$ Merge(G2, G1),

(b) Merge(Merge(G1, G2), G3) $=_g$ Merge(G1, Merge(G2, G3))

In the remainder of this paper, in order to avoid redundancy whenever G1 and G2 play symmetrical roles, we state and prove properties of Merge(G1, G1) relatively to G1 only. Same properties hold with respect to G2, since operation Merge is commutative.

Merge(G1, G2) always extends G1.

Proposition 4.3
Given two AGs, G1 and G2, Merge(G1, G2) ext_g G1.

In order to be a cyclic extension of G1, Merge(G1, G2) should preserve the cyclic traces of G1. Merge(G1, G2) preserves the cyclic traces of G1, if and only if it preserves, at least as cyclic traces, the elementary cyclic traces of G1. However, there is some situation where an elementary cyclic trace in G1 is a noncyclic trace in Merge(G1, G2). Indeed, this is the case when a certain elementary cyclic trace σ in G1 ($g1_o{=}\sigma{\Rightarrow}g1_o$) is a noncyclic trace in G2 ($g2_o{=}\sigma{\Rightarrow}g2_k$ with $g2_k \quad g2_o$). By definition of Merge, after performing σ, Merge(G1, G2) reaches a state $<g1_o, g2_k>$ different from its initial $<g1_o, g2_o>$, since $g2_k \quad g2_o$. Therefore, σ is a noncyclic trace in Merge(G1, G2). The example in Figure 6 illustrates such situations. For instance, **a** is an elementary cyclic trace in G1 ($g1_o{=}\sigma{\Rightarrow}g1_o$), but **a** is a non cyclic trace in G2 ($g2_o{=}\sigma{\Rightarrow}g2_1$ with $g2_1 \quad g2_o$). Therefore, **a** is a non cyclic trace in Merge(G1, G2) ($<g1_o, g2_o>{=}\sigma{\Rightarrow}<g1_o, g2_1>$ with $g2_1 \quad g2_o$). In Proposition 4.4, we state a necessary and sufficient condition for an elementary cyclic trace in G1 to remain a cyclic trace in Merge(G1, G2).

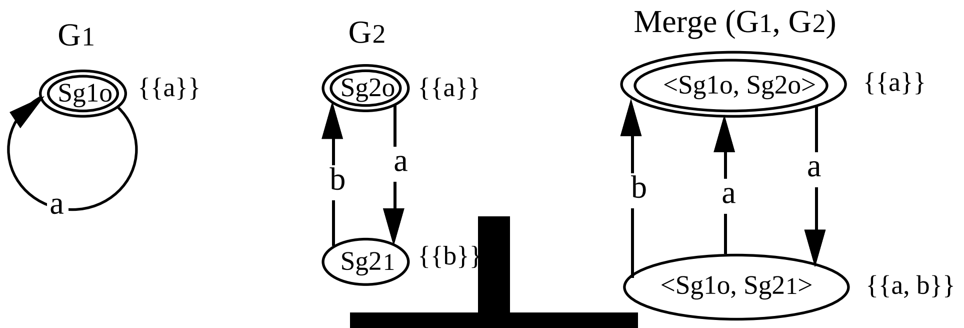

Figure 6. Preservation of the cyclic traces by Merge.

Proposition 4.4
Given two AGs, G1 and G2,
an elementary cyclic trace σ in G1 is a cyclic trace in Merge(G1, G2) iff
(σ is a cyclic trace in G2 or σ Tr(G2)).

From Proposition 4.4, it follows that Merge(G1, G2) preserves the cyclic traces of G1 if and only if any elementary cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G2), which is equivalent to any cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G2) as stated in the following proposition.

Proposition 4.5
Given two AGs, G1 and G2, the following statements are equivalent:

(a) Merge(G1, G2) preserves the cyclic traces of G1,

(b) any elementary cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G2).

(c) any cyclic trace σ in G_1 is a cyclic trace in G_2 or $\sigma \quad Tr(G_2)$.

The conditions (b) (and (c)) in Proposition 4.5 can be stated in terms of states as follows: for any state $<g1_i, g2_j>$ in Merge(G_1, G_2), if $g1_i = g1_o$ then $g2_j = g2_o$. This condition is very easy to verify in the case of FAGs.

In Proposition 4.4, we have stated a sufficient and necessary condition for which an elementary cyclic trace σ in G_1 remains a cyclic trace in Merge(G_1, G_2). Moreover, in this case σ is an elementary cyclic trace in Merge(G_1, G_2). Indeed, if $\sigma = a1.a2...an$ and $g1_o{-}a1{\rightarrow}g1_i$, $g1_i{-}a2{\rightarrow}g1_{i+1}$ ..., $g1_{i+n-2}{-}an{\rightarrow}g1_o$ with $g1_{i+j} \quad g1_o$, for $j = 0, ..., n-2$, and σ is a cyclic trace in Merge(G_1, G_2), then by definition of Merge, $<g1_o, g2_o>{-}a1{\rightarrow}g_i$, $g1_i{-}a2{\rightarrow}g_{i+1}$ ..., $g_{i+n-2}{-}an{\rightarrow}<g1_o, g2_o>$ with $g_{i+j} = g1_{i+j}$ or $<g1_{i+j}, g2_{kj}>$ for some state $g2_{kj}$ in G_2 and $g_{i+j} \quad <g1_o, g2_o>$, since $g1_{i+j} \quad g1_o$, for $j = 0, ..., n-2$. However, an elementary cyclic trace in Merge(G_1, G_2) is not always an elementary cyclic trace in G_1 or G_2. As shown by the example in Figure 6, **a.a** is neither an elementary cyclic trace in G_1 nor in G_2. **a.a** is a cyclic trace in G_1. As stated by Proposition 4.6, any elementary cyclic trace in Merge(G_1, G_2) is a cyclic trace in G_1 or G_2.

Proposition 4.6
Given two AGs, G_1 and G_2,
any elementary cyclic trace in Merge(G_1, G_2) is a cyclic trace in G_1 or G_2.

Any trace in Merge(G_1, G_2) results from the recursive concatenation of cyclic traces of G_1 or G_2, and a certain trace of G_1 or G_2. In other words, Merge(G_1, G_2) may only perform what G_1 or G_2 may perform, in a recursive manner.

Proposition 4.7
Given two AGs, G_1 and G_2,
any trace σ of Merge(G_1, G_2) may be written as $\sigma = \sigma_1.\sigma_2...\sigma_n.\sigma_{n+1}$, with
σ_i as a cyclic trace in G_1 or G_2, for $i = 1, ..., n$, and ($\sigma_{n+1} \in Tr(G_1)$ or $\sigma_{n+1} \in Tr(G_2)$).

In the case where the cyclic traces of G_1 and the cyclic traces of G_2 remain as cyclic traces in Merge(G_1, G_2), Merge(G_1, G_2) represents the least common cyclic extension of G_1 and G_2. The following theorem follows partly from Proposition 4.3 and Proposition 4.5.

Theorem 4.1
Given two AGs, G_1, G_2,

Merge(G_1, G_2) is **the least common cyclic extension** of G_1 and G_2 iff any cyclic trace σ in G_1 is a cyclic trace in G_2 or $\sigma \notin$ Tr(G_2), and reciprocally.

Due to the constraint for the preservation of the cyclic traces of G_1 and G_2 in Merge(G_1, G_2), bisimulation equivalence is not substitutive under the Merge combinator. In other words, the fact that X is bisimulation equivalent to Y does not ensure that Merge(X, Z) is bisimulation equivalent to Merge(Y, Z). The example in Figure 7, for instance, illustrates such situation. We have $G_1 \approx_g G_3$ but Merge(G_1, G_2) and Merge(G_3, G_2) are not bisimulation equivalent. As shown by this example, this is due to the fact that **a** is a cyclic trace in G_1 but not in G_3. The cyclic bisimulation equivalence is substitutive under the Merge combinator. As stated by Theorem 4.2, if X is cyclic bisimulation equivalent to Y then Merge(X, Z) is cyclic bisimulation equivalent to Merge(Y, Z), for any AG Z. Therefore, Merge(X, Z) is bisimulation equivalent to Merge(Y, Z).

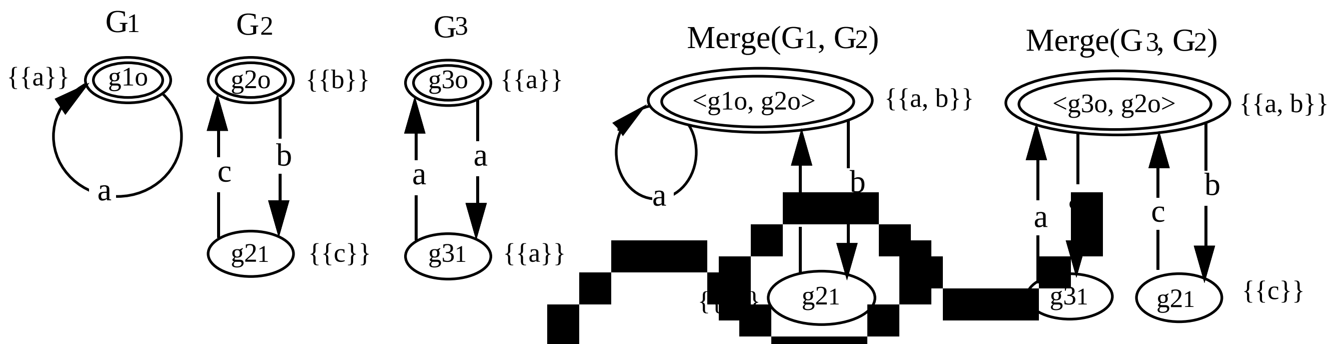

Figure 7. Substitution property of the bisimulation equivalence under Merge.

Theorem 4.2
Given three AGs, G_1, G_2, and G_3, such that $G_1 \approx_{cg} G_3$,
the following holds: Merge(G_1, G_2) $\approx_{cg}$ Merge(G_3, G_2)

4.2 Merging FAGs and Application

In the previous section the Merge combinator has been defined for arbitrary AGs. In the following, we describe an algorithm, also called Merge, for the construction of Merge(G_1, G_2), in the case of FAGs, and we apply it for the combination of two versions of the so-called Daemon Game [ISO 8807]. Notice that, in the case of an FAG G, for any state g_i of G, Ac(g_i) and any element in Ac(g_i) are finite, since Ac(g_i) $\subseteq$ *P(L)* .

Algorithm Merge

Given two AGs, $G1 = \langle Sg1, L1, Ac1, Tg1, g1_o \rangle$ and $G2 = \langle Sg2, L2, Ac2, Tg2, g2_o \rangle$, $Merge(G1, G2) = \langle Sg3, L1 \;\; L2, Ac3, Tg3, \langle g1_o, g2_o \rangle \rangle$, where Sg3, Ac3 and Tg3 are built, recursively, as follows:

Initial step:

$Sg3 = \{\langle g1_o, g2_o \rangle\}$ and $Ac3(\langle g1_o, g2_o \rangle) = \{X1 \;\; X2 \mid X1 \in Ac1(g1_o)$ and $X2 \in Ac2(g2_o)\}$.

Loop:

For each state g_i entered into Sg3 (first for the initial state $\langle g1_o, g2_o \rangle$) repeat the following:

```
if $g_i = \langle g1_j, g2_k \rangle$, then for each $A \in Ac3(\langle g1_j, g2_k \rangle)$ and $a \in A$,
    if $g1_j - a \rightarrow g1_l \in Tg1$ and $g2_k - a \rightarrow g2_m \in Tg2$, then
            $Sg3 = Sg3 \;\; \{\langle g1_l, g2_m \rangle\}$, $\langle g1_j, g2_k \rangle - a \rightarrow \langle g1_l, g2_m \rangle \in Tg3$ and
            $Ac3(\langle g1_l, g2_m \rangle) = \{ X1 \;\; X2 \mid X1 \in Ac1(g1_l)$ and $X2 \in Ac2(g2_m)\}$.
    if $g1_j - a \rightarrow g1_o \in Tg1$ and $g2_k -,/a\rightarrow$ in Tg2, then $\langle g1_j, g2_k \rangle - a \rightarrow \langle g1_o, g2_o \rangle \in Tg3$.
    if $g1_j -,/a\rightarrow$ in Tg1 and $g2_k - a \rightarrow g2_o \in Tg1$, then $\langle g1_j, g2_k \rangle - a \rightarrow \langle g1_o, g2_o \rangle \in Tg3$.
    if $g1_j - a \rightarrow g1_l \in Tg1$, with $g1_l \;\; g1_o$ and $g2_k -,/a\rightarrow$ in Tg2, then
            $Sg3 = Sg3 \;\; \{g1_l\}$, $Ac3(g1_l) = Ac1(g1_l)$ and $\langle g1_j, g2_k \rangle - a \rightarrow g1_l \in Tg3$.
    if $g1_j -,/a\rightarrow$ in Tg1 and $g2_k - a \rightarrow g2_m \in Tg2$, with $g2_m \;\; g2_o$, then
            $Sg3 = Sg3 \;\; \{g2_m\}$, $Ac3(g2_m) = Ac2(g2_m)$ and $\langle g1_j, g2_k \rangle - a \rightarrow g2_m \in Tg3$.
if $g_i = gx_j$, with $x = 1, 2$, then for each $A \in Ac3(gx_j)$ and $a \in A$,
    if $gx_j - a \rightarrow gx_o \in Tgx$, then $gx_j - a \rightarrow \langle g1_o, g2_o \rangle \in Tg3$.
    if $gx_j - a \rightarrow gx_l \in Tgx$, with $gx_l \;\; gx_o$, then
                    $Sg3 = Sg3 \;\; \{gx_l\}$, $Ac3(gx_l) = Acx(gx_l)$, and $gx_j - a \rightarrow gx_l \in Tg3$.
```

Application

As application, we consider two versions of the Daemon game [ISO 8807]. The first game is called Simple Daemon Game. The player may insert a coin, probe the system, then he randomly loses or wins and collects. The behavior of this game is modeled by the FAG G1 in Figure 8 (a). The second game is called Jackpot Daemon Game. The behavior of this second game is as follows: the player has to insert a coin before starting the game. Once the coin has been inserted, the player can 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" and collect it. The behavior of Jackpot Daemon Game is modeled by the FAG G2 in Figure 8 (b).

Assume that we want to combine these two games, in order to describe a new system, called Combined Game, where the player can, alternatively, play the Simple Daemon Game and the Jackpot Daemon Game, without any interference between these two games. Merge(G1, G2), as shown in Figure 9, defines such a combination of the Simple Daemon Game and the Jackpot

Daemon Game. We have Merge(G1, G2) extends G1 and G2. Moreover, any cyclic trace of G1 remains as cyclic trace in Merge(G1, G2), since there is no state $<g1_o, g2_j>$ in Merge(G1, G2) with $g2_j$ $g2_o$. Any cyclic trace of G2 remains as cyclic trace in Merge(G1, G2), since there is no state $<g1_i, g2_o>$ in Merge(G1, G2) with $g1_i$ $g1_o$. Merge(G1, G2) is the least common cyclic extension of G1 and G2. Merge(G1, G2) is able to behave, alternatively, in a recursive manner, as G1 and G2.

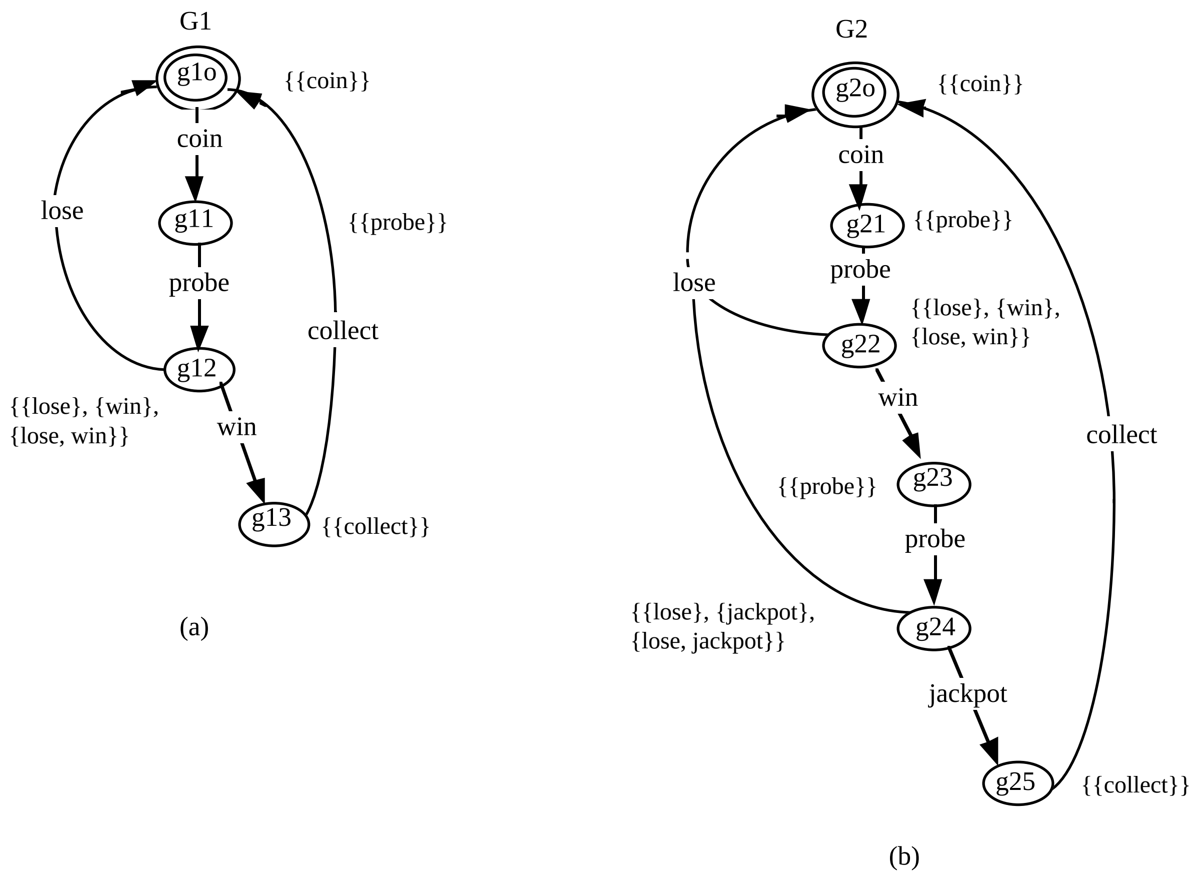

Figure 8. (a) Simple Daemon Game **(b)** Jackpot Game Descriptions.

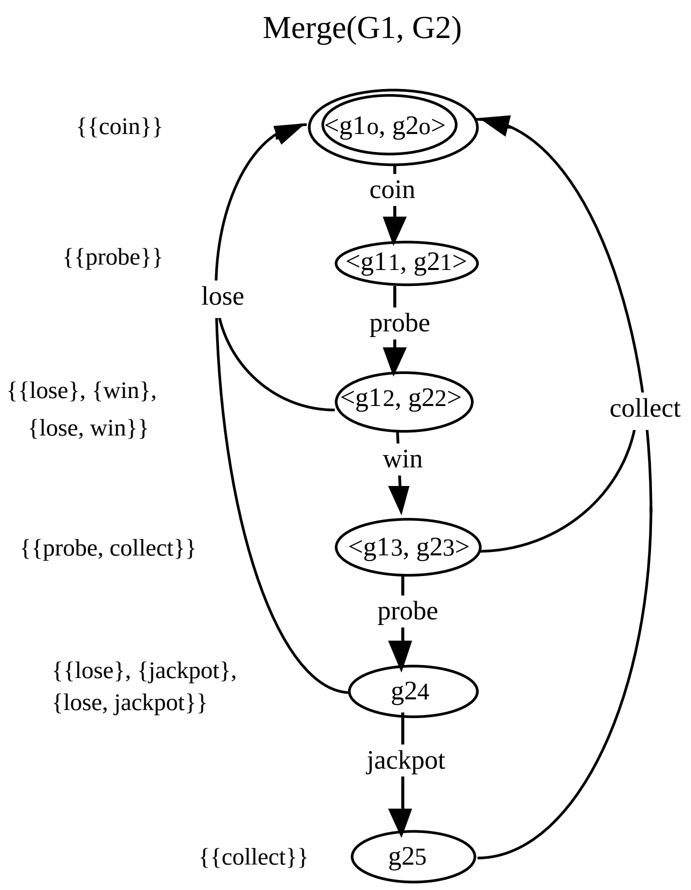

Figure 9. Combined Game Description.

4.3 Discussion

The operation Merge defined in Section 4.1 is such that, for given AGs, G1 and G2, in the case of the cyclic traces of G1 or G2, Merge(G1, G2) may exhibit the behaviors of G1 and the behaviors of G2, in a recursive manner, without any new failure for these behaviors. Consider, for instance, the example in Section 4.2, the Combined Game may exhibit the behaviors of the Simple Daemon Game and the behaviors of the Jackpot Daemon Game, in a recursive manner. Each time the Combined Game exhibits a behavior of the Simple Daemon Game or a behavior of the Jackpot Daemon Game, the Combined Game does not block where the Simple Daemon Game or the Jackpot Daemon Game may not block, respectively.

Merge(G1, G2) always extends G1 and G2. Provided that certain necessary and sufficient condition (Theorem 4.1) is satisfied, Merge(G1, G2) is the least common cyclic extension of G1 and G2. In general, Merge(G1, G2) is not the least common extension of G1 and G2. The least common extension of G1 and G2 is defined by the combinator , which is very similar to Merge operation, except for the rules defining the transitions, which are replaced by the following rules:

(3) For each state $<g1_j, g2_k>$ in Sg3,

3-1. $<g1_j, g2_k>-a\rightarrow<g1_l, g2_m> \in$ Tg3 iff $g1_j-a\rightarrow g1_l \in$ Tg1 and $g2_k-a\rightarrow g2_m \in$ Tg2.

3-2. $<g1_j, g2_k>-a\rightarrow g1_l \in$ Tg3 iff $g1_j$ [illegible] 2.

3-3. $<g1_j, g2_k>-a\rightarrow g2_m \in$ Tg3 iff $g2_k-a\rightarrow g2_m \in$ Tg2 and $g1_j$–,/a→ in Tg1.

(4) For each state gx_j in Sg3 where x = 1, 2, gx_j [illegible] $gx_l \in$ Tg3 iff gx_j [illegible] $gx_l \in$ Tgx.

Contrarily to Merge(G1, G2), in G1 [illegible] 1 of G1 (respectively $g2_j$ of G2), that reach the initial state [illegible] of G1 (respectively $g2$ of G2) are preserved without change. For a state [illegible] labelled by a in G[illegible] then the [illegible] defined in G1 [illegible] instead of [illegible] $g2_k>-a\rightarrow<g$[illegible]$_o$, g[illegible] ively. Once the initial state $g1_o$ of G1 (respectively $g2_o$ of G2) is reached, G1 G2 behaves [illegible].

The combinator defines an AG. It is commutative and associative. G1 G2 always extends G1 and G[illegible] **the least common extension of G1 and G2**. By definition, G1 G2 does not preserve the cyclic traces of G1 (respectively G2), except the cyclic traces common to G1 and G2. We have Tr(G1 G2) = Tr(G1) [illegible] Tr(G2). G[illegible] behaviors [illegible] example in Figure 10, which describes the least common [illegible] Jackpot Daemon Game. In this figure, G1 G2 describes a system which may behave only as the Simple [illegible] Jackpot Daemon Game.

The operation preserves the bisimulation equivalence. In other words, if G1 $_g$ G3 then G1 G2 $_g$ G3 G2. Moreover, preserves the cyclic bisimulation equivalence, since only the common cyclic traces of G1 (respectively G3) and G2 are preserved in G1 G2 (respectively G3 G2).

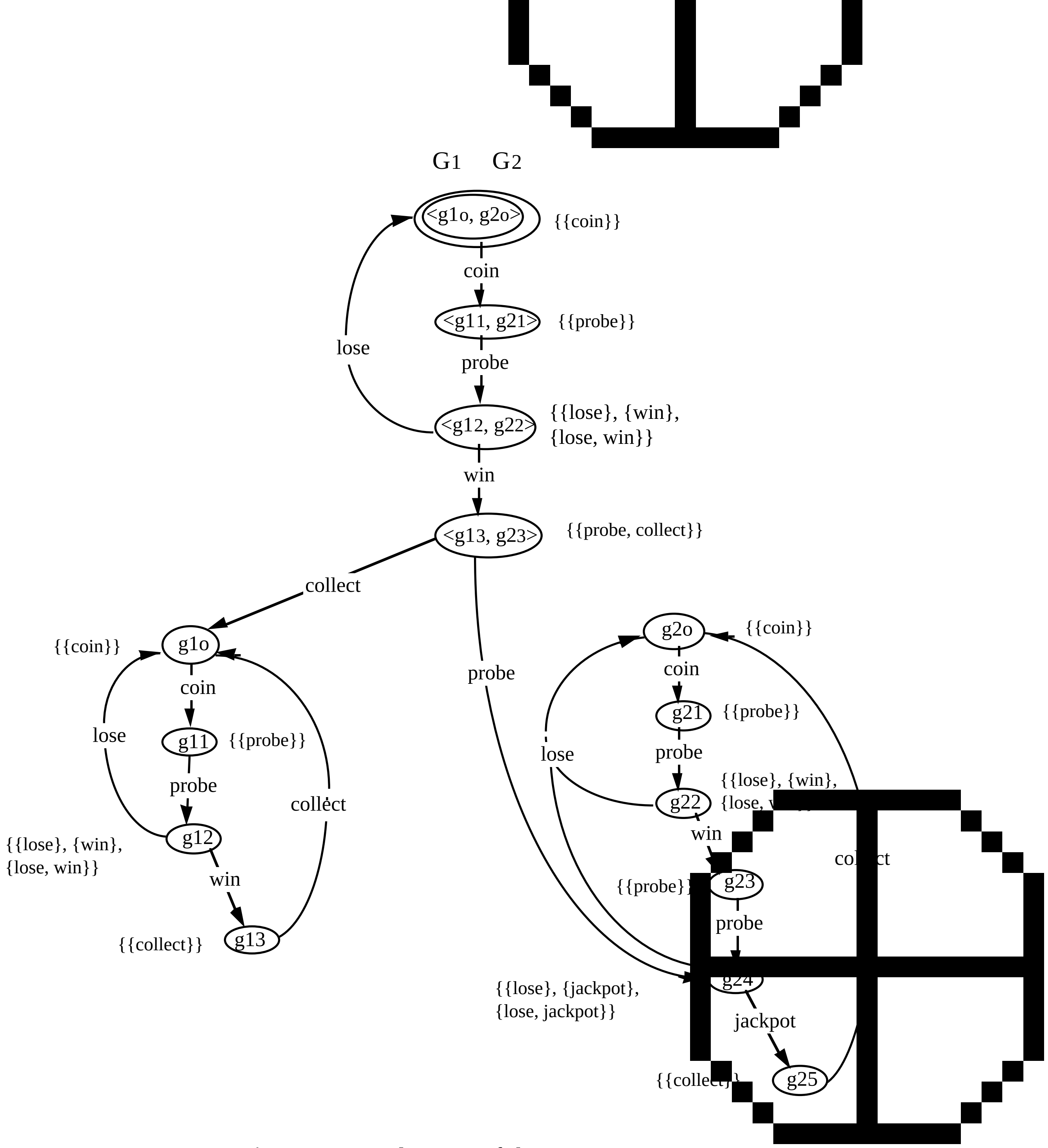

Figure 10. Application of the operation .

5 Merging Labelled Transition Systems

The definition of Merge for LTSs is based on the definition of Merge for AGs and the correspondence between LTSs and AGs.

5.1 Definition and Properties of Merge

Definition 5.1 (Merge for LTSs)
Given two LTSs S_1 and S_2, $\text{Merge}(S_1, S_2) = \text{lts}(\text{Merge}(\text{ag}(S_1), \text{ag}(S_2)))$.

Since for any LTS S, there is one and only one AG G such that G = ag(S), for a[illegible]y AG G there is one and only one LTS such S = lts(G), and for given AGs G_1 and [illegible]lways exists and uniquely defined, then for given LTSs S_1 and S_2, Merge(S_1, S_2), al[illegible]ys, exists and is uniquely defined.

All the propositions, lemmas and Theorem 4.1 stated for Merge in [illegible]e case of A[illegible] holds for Merge in the case of LTSs. For instance, Merge(S_1, S_2) always extends [illegible] [illegible]erge([illegible]) is commutative and associative. Merge(S_1, S_2) is the least common cyclic extensi[illegible] of S_1 and S_2, if and only if any cyclic trace σ in S_1 is a cyclic trace in S_2 or σ Tr(S_2) and recipro[illegible]lly.

By correspondence to the AGs and Theorem 4.2, the testing, observation, strong bisimulation equivalences are not substitutive [illegible]der the [illegible]TSs M[illegible]rg[illegible] com[illegible]nator. H[illegible]vever, the cyclic (testing[illegible] observation, strong bisimulati[illegible]n) equivalenc[illegible]s [illegible]re substitu[illegible]v[illegible] under [illegible]e LTSs [illegible]erge com[illegible]in[illegible]tor. The fact that X and Y are, [illegible]t least, cyclic [illegible]ting equivale[illegible] en[illegible]res that M[illegible]e(X, Z) is [illegible] bisimulation equivalent to [illegible]erge(Y, Z). Ind[illegible]d, if X and Y a[illegible]t least, cycli[illegible]sting equi[illegible] their corresponding AGs ag[illegible] [illegible]ma 3.1 in S[illegible] 3), Merge(ag(X), ag(Z)) is [illegible]clic bisimulati[illegible] equivalent to [illegible] Section 4), and lts(Merge([illegible]g(X), ag(Z))) a[illegible] lts(Merge(ag[illegible] ag(Z))) are [illegible]lic bisimu[illegible] equivalent (Proposition 3.5 in [illegible]ction 3).

Similarly to Merge, S_1 S_2 = lts(ag(S_1) ag(S_2)). By correspondenc[illegible] to the A[illegible]s S_1 [illegible] is the least common extension of S_1 and S_2 and the properties of in the case of AGs hold for in the case of LTSs.

5.2 Merging FLTSs and Application

In the previous section, we defined Merge(S_1, S_2) for arbitrary LTSs. In this section, we describe an algorithm for the construction of Merge(S_1, S_2), for the case where S_1 and S_2 are FLTSs. This algorithm consists of three steps. In the first step, S_1 and S_2 are transformed into FAGs G_1 and G_2, such that $G_1 = \text{ag}(S_1)$ and $G_2 = \text{ag}(S_2)$. In the second step, Merge(G_1, G_2) is constructed following algorithm Merge described in Section 4.2. In the last step, Merge(G_1, G_2) is translated into lts(Merge(G_1, G_2)).

5.2.1 From an FLTS to an FAG

Given an FLTS S = <St, L, T, s_o>, the following algorithm de[illegible] corresponding FAG G = <Sg, L, Ac, Tg, g_o>. It is based on the "subset construction" a[illegible] defined in [Hopc 79].

Step 1: Apply the "subset construction" algorithm [Hopc [illegible]9], [illegible]h transforms a nondeterministic finite state automata to a determ[illegible]e (in our ca[illegible] G). To each state in G corresponds a set of states in S. To th[illegible] g_o, for ins[illegible]nce, corresponds the set of states $\{s_i \in St \mid s_o = \varepsilon \Rightarrow s_i\}$.

Step 2: For each state g_i in G, $Ac(g_i) = \{X \mid out(s_j)$ X $_{k=m, k=1}$ out(s[illegible] m\}, if $\{s_1, s_2, ..., s_m\}$ corresponds to g_i[illegible]

5.2.2 From an FAG to an FLTS

Given an FAG G = <Sg, L, Ac, Tg, g_o>, the fo[illegible]wing al[illegible]m allows t[illegible]he FLTS S = <St, L, T, s_o> = lts(G).

Step 1: (Reduction of the acceptance sets):
$\forall g_i \in Sg$, $Ac'(g_i) = \{X \mid X \in Ac(g_i)$, suc[illegible]that Y [illegible]
and X = Y Z or Y [illegible]

Step 2: Each state g_i is decomposed into k+1 LTS states $s_i, s_{i1}, s_{i2}, ..., s_{ik}$, where k = cardinal($Ac'(g_i)$). s_o represents the initial state of S. Each state s_{ij} corresponds to an element A_{ij} of $Ac'(g_i)$.
The transitions $s_i \xrightarrow{\tau} s_{ij}$ are defined in S, for j =1, ..., k, for each state s_i in S.

Step 3: For each state s_{ij} in St, for each $a \in A_{ij}$, if $g_i \xrightarrow{a} g_m \in Tg$, then $s_{ij} \xrightarrow{a} s_m \in T$.

5.2.3 Application

We consider the same example as in Section 4. The behaviors of the "Simple Daemon Game" and the "Jackpot Daemon Game" are modeled by FLTSs S_1 and S_2 in Figure 11, respectively. Merging S_1

and S_2 yields the FLTS S_3 shown in Figure 11. S_3 extends S_1 and S_2. Moreover, any cyclic trace of S_1 or S_2 remains a cyclic trace in S_3. S_3 is the least common cyclic extension of S_1 and S_2. S_3 may behave, alternatively, in a recursive manner, as S_1 and S_2. Note that S_3 may be reduced with respect to the (cyclic) observation equivalence by removing some internal transitions τ.

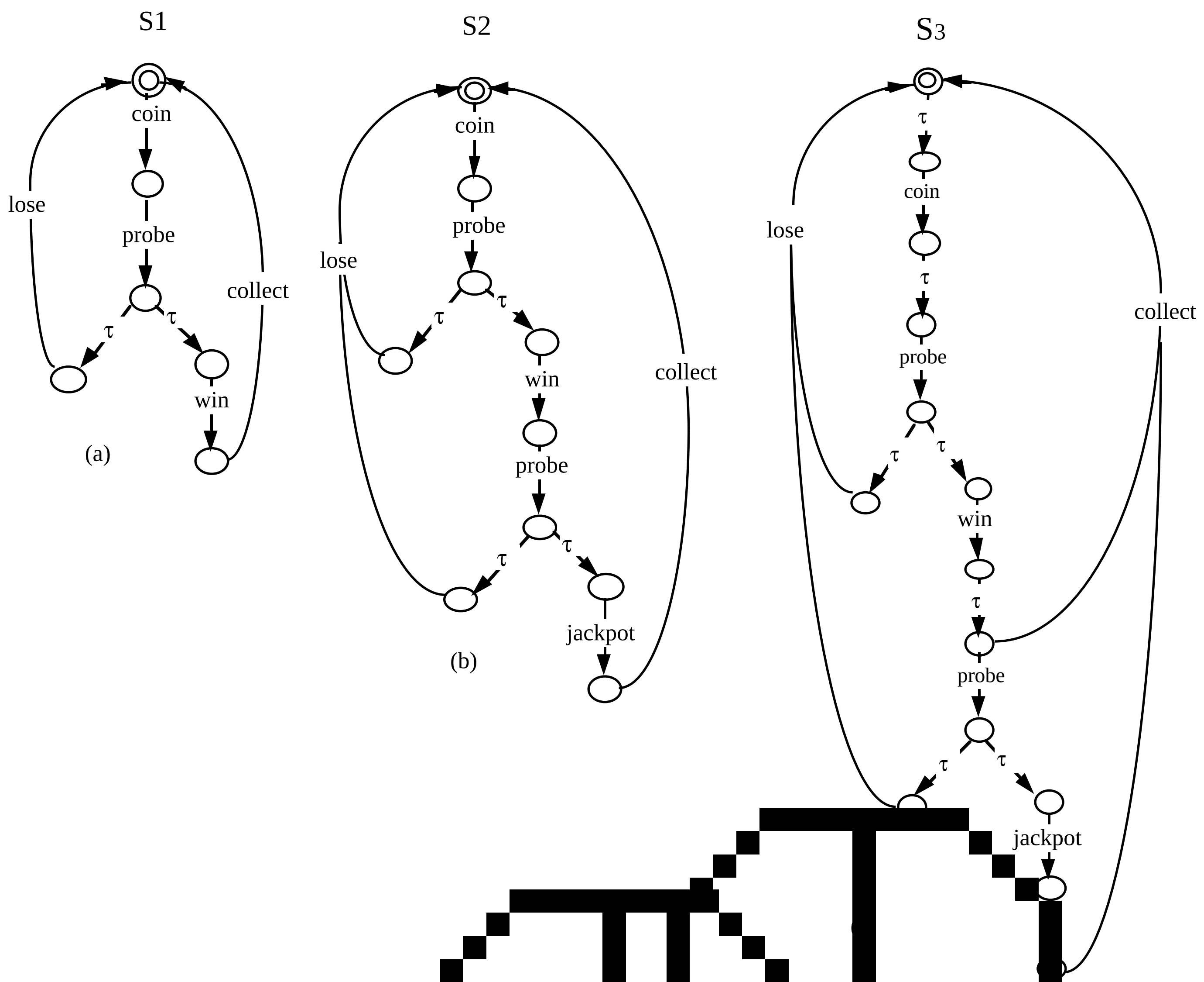

Figure 11. (a) Simple Daemon Ga[illegible] **(b)** Jackpot [illegible]ame [illegible]s.

6 Related work

In [Ichi 90], the problem of incremental specification in the LOTOS specification language is approached. They introduced a new LOTOS operator and defined the corresponding inference rules, called specification merging operator. This approach is restricted to behavior specifications without the internal action τ. B_1 B_2 defines a behavior, which is supposed to be an extension of

B1 and B2. Unfortunately, it is not always the case as shown by the counter-example of Figure 12. For instance, B1 never refuses interaction **c** after trace **a.b**, whereas B1 B2 may refuse interaction **c** after trace **a.b**. Moreover, B1 B2 is not able to behave, alternatively, as B1 and B2. B1 B2 may behave only as B1 or only as B2, once the environment has chosen B1 or B2, respectively. In the case of deterministic LTSs, this combinator leads the same LTS as the combinator (merging without taking into account the preservation of cyclic traces) introduced in this paper.

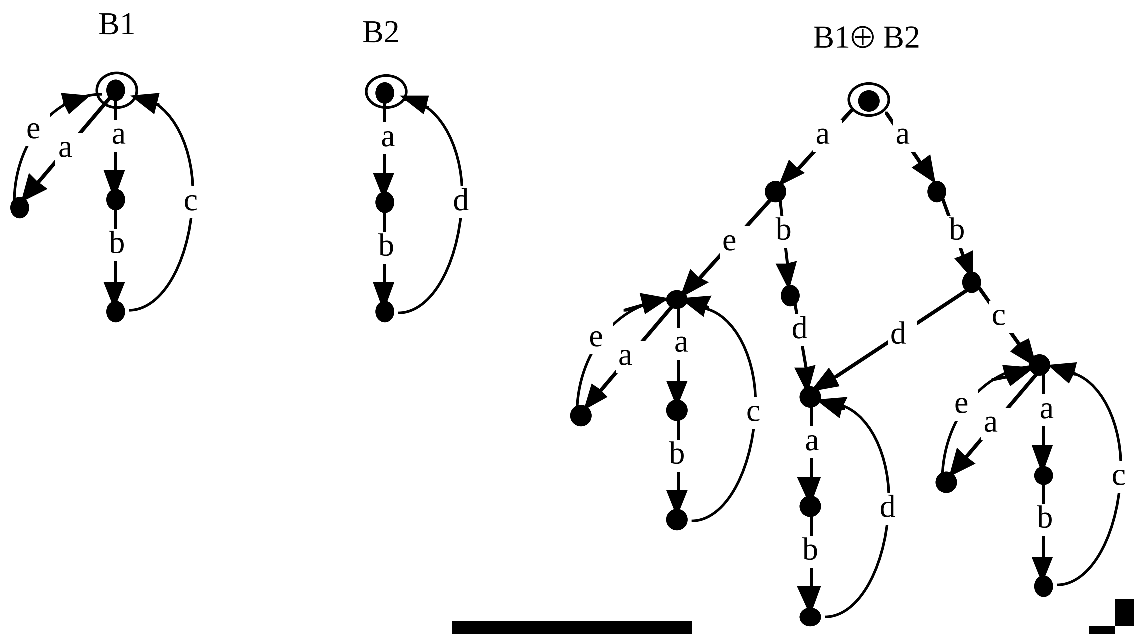

Figure 12. Counter-example for Ichikawa et al. merging operator.

Mayr has considered the choice operator of the LOTOS language for the extension of behavior specifications [Mayr 88]. The extension of a behavior t by a behavior m is denoted by [illegible] [] m. However, strong restrictions [illegible] in order to ensure that s extends t. For instance, the initial interactions of m should be distinct from initial interactions of t.

In [Rudk 91] the notion of inheritance is defined for LOTOS. It is seen as an incremental modification technique. A corresponding operator is introduced and denoted by " ". This operator is defined such that if s = t m , then s extends 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. The specifications B1 and B2 in Figure 14, for instance, do not satisfy such requirements. In order to define a recursive choice between t and m, Rudkin extended the LOTOS language by a new primitive process "self". 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.

Lin has developed an approach for merging alternative protocol functions [Lin 91]. The approach is based on the model of communicating finite state machines. It consists of designing a component protocol for each individual function and then combine them into a single alternating-function protocol. The combination algorithm resolves problems of competition and synchronization between the component protocols, in order to preserve the safety properties (absence deadlock and unspecified receptions) of the component protocols. However, this approach does not take into account the service realized by each protocol component and how this service is preserved in the alternating-function protocol.

7 Conclusion

In this paper, we described an approach for merging behavior specifications. These behav rs are modeled by acceptance graphs or labelled transition systems. Given two behavior specificat and B_2, we defined the merging of B_1 and B_2, written Merge(B_1, B_2). We proved certain pr erties of Merge; for instance, Merge(B_1, B_2) extends B_1 and B_2. Provided that a necessary and su icient condition holds, the cyclic traces in B_1 (respectively B_2) remain cyclic traces in Merge(B_1,). Therefore, Merge(B_1, B_2) is a cyclic extension of B_1 and B_2. Moreover, in this case, Merge(B_1, B_2) is the least common cyclic extension of B_1 and B_2. We defined a second combinator, , which is very similar to Merge, but differs on the treatment of the cyclic traces of B_1 and B_2. The operation always leads the least common extension of B_1 and B_2.

The proposed approach for merging behavior specifications is useful for the construction of multiple-function specifications. Instead of handling all the functions simultaneously, the designer may design and verify one function at a time. The merging approach will then derive the required combined specification. From another point of view, it allows the designer to enrich existing specifications with new behaviors required by the user and to integrate existing system specifications.

The approach introduced in this paper has been extended to structured specifications, i.e. specifications which are modeled as parallel composition of subsystem specifications [Khen 93]. As future development, the application of the extended approach to real case system specifications, such as the telephone system specification, is expected.

The labelled transition systems model used in this paper is the underlying semantic model for many specification languages, such as LOTOS [ISO 8807] and CCS [Miln 89]. The full examination of the algebraic properties of the merging operators Merge and as well as the congruence property of the newly introduced (cyclic) equivalences in the context of these languages is left for future development.

References

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

[Broo 85] S. D. Brookes et A. W. Roscoe, An Improved Failure Model for Communicating Sequential Processes, Proceedings of the NSF-SERC Seminar on Concurrency, Springer-Verlag LNCS 197, 1985.

[DeNi 84] R. De Nicola et M. Hennessy, Testing equivalences for processes, Theo. Comp. Sci. 34, 1984, pp. 83 133.

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

[Clea 93] R. Cleaveland and M. Hennessy, Testing Equivalence as Bisimulation Equivalence, Formal Aspects of Computing, 5, pp. 1-20, 1993.

[Drira 92] K. Drira, Tranformation et composition de graphes de refus: analyse de la testabilité, Doctorat Thesis, Université de Toulouse, 1992.

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

[Henn 88] M . Hennessy, Algebraic Theory for Processes, MIT Press, Cambridge, 1988.

[Hopc 79] J. E. Hopcroft and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979, 418p.

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

[ISO 8807] ISO -Information Processing Systems - Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, DIS 8807, 1987.

[Khen 93] F. Khendek and G.v. Bochmann, Incremental Construction Approach for Distributed System Specifications, Proceedings of the Int. Symp. on Formal Description Techniques, Boston, Mas., 26-29 Oct., 1993.

[Ledu 90] G. Leduc, On the role of Implementation Relations in the Design of Distributed systems using LOTOS, Doctoral Dissertation, Liège, Belgium.

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

[Lin 91] H. A. Lin, Constructing Protocols with Alternative Functions, IEEE Transactions on Computer, Vol. 40, No. 4, April 1991.

[Mayr 88] T. Mayr, Specification of object-oriented systems in LOTOS, FORTE, Stirling, 1988.

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

[Park 81] D. Park, Concurrency and Automata in Infinite Strings, Lecture Notes in Computer Science 104, Springer-Verlag, Berlin, 1981, pp. 167-183.

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

Appendix

Proposition 3.1

Consider two AGs, $G1 = \langle Sg1, L1, Ac1, Tg1, g1_o \rangle$ and $G2 = \langle Sg2, L2, Ac2, Tg2, g2_o \rangle$.

1 - Assume that Tr(G1) = Tr(G2) and (∀ σ ∈ Tr(G1), Ac1(g1$_0$ after σ) = Ac2(g2$_0$ after σ)).
To prove that G1 $_g$ G2, we have to prove that the relation {((g1$_0$ after σ), (g2$_0$ after σ)): σ ∈ Tr(G1)} is a bisimulation. By hypothesis, Ac1(g1$_0$ after σ) = Ac2(g2$_0$ after σ), ∀ σ ∈ Tr(G1).
Consider (g1$_0$ [redacted] g2$_0$ after σ) ∈ [redacted], for some σ ∈ Tr(G1), (g1$_0$ after σ)–a→g1$_i$, if and only if (g2$_0$ after σ)–a→g2$_j$, [redacted]nce Tr(G1) [redacted] Tr(G2). [redacted]e have g1$_i$ = g1$_0$ after σ.a and g2$_j$ = g2$_0$ after σ.a, since the transition relation is a function in the case of AGs. Therefore, (g1$_i$, g2$_j$) ∈ {(g1$_0$ after σ, g2$_0$ after σ): σ ∈ Tr(G1)}, and the relation {((g1$_0$ after σ), (g2$_0$ after σ)): σ ∈ Tr(G1)} is a bisimulation.

2 - G1 $_g$ G2, there is a bisimulation R such that (g1$_0$, g2$_0$) ∈ R, and ∀ (g1$_i$, g2$_j$) ∈ R, Ac1(g1$_i$) = Ac2(g2$_j$). Consider σ, an arbitrary sequence of actions. First case σ = ε, it is obvious that ε ∈ Tr(G1) and ε ∈ Tr(G2). By definition of AGs, g1$_0$ after ε = g1$_0$ and g2$_0$ after ε = g2$_0$. By hypothesis, (g1$_0$, g2$_0$) ∈ R and Ac1(g1$_0$ after ε) = Ac2(g2$_0$ after ε). Second case σ = a1.a2...an, σ ∈ Tr(G1) if and only if g1$_0$–a1→g1$_i$–a2→g1$_{i+1}$...g1$_{i+n-2}$–an→g1$_{i+n-1}$. The transition relations Tg1 and Tg2 are functions and (g1$_0$, g2$_0$) ∈ R. It follows that g1$_0$–a1→g1$_i$–a2→g1$_{i+1}$..[redacted]$_{i+n-2}$–an→g1$_{i+n-1}$ if and only if g2$_0$–a1→g2$_j$–a2→g2$_{j+1}$...g2$_{j+n-2}$–an→g2$_{j+n-1}$ with (g1$_i$, g2$_j$) ∈ R, ([redacted]$_{i+1}$, g2$_{j+1}$) ∈ R, ...[redacted]nd (g1$_{i+n-1}$, g2$_{j+n-1}$) ∈ R. [redacted]onsequently, σ ∈ Tr(G1) if and only if σ ∈ Tr(G2) (T[redacted]G1) =Tr(G2)) and [redacted]c1(g1$_0$ after σ) = Ac2(g2$_0$ af[redacted]σ).

Proposition 3.2
Consider the AGs, G1, G2 and the L[redacted]Ss S1[redacted]with g1$_0$, g2$_0$, s1$_0$, s2$_0$, as in [redacted]
1- First, we have to prove that S2 ext S1 [redacted]
1 - 1 - Prove that S2 ext S1 ⇒ G2 ext$_g$ G1:
1 - 1 - a - Prove that Tr(G1) Tr(G2)[redacted]: G1 = ag(S[redacted]plies that Tr(S1) = [redacted](G1). G2 = ag(S2) impl[redacted] that Tr(S2) =Tr(G2). S2 ext S1 implies that Tr(S1) Tr(S2)[redacted]
1 - 1 - b - ∀ σ ∈ Tr(G1), Ac2(g2$_0$ after σ) Ac1(g1$_0$ after σ): G1 = ag(S1) i[redacted]ies that Ac1(g1$_0$ after σ) = Acc(s1$_0$, σ). G2 = ag(S2) implies that Ac2(g2$_0$ after σ) = Acc(s2$_0$, σ). ∀ σ ∈ Tr(S1)[redacted] σ), because S2 ext S1. It follows that, ∀ σ ∈ Tr(G1), Ac2(g2$_0$ after σ) Ac1(g1$_0$ after σ). Consequently, S2 ext S1 ⇒ G2 ext$_g$ G1.
1 - 2 - The proof for G2 ext$_g$ G1 ⇒ S2 ext S1 is very similar.

2 - Any cyclic trace in S1 is a cyclic trace in S2, iff any cyclic trace in G1 is a cyclic trace in G2 :
2 - 1 - Any cyclic trace in S1 is a cyclic trace in S2 ⇒ any cyclic trace in G1 is a cyclic trace in G2 :
G1 = ag(S1), it follows that any cyclic trace in S1 is a cyclic trace in G1, and reciprocally.
G2 = ag(S2), it follows that any cyclic trace in S2 is a cyclic trace in G2, and reciprocally.
Now, assume that any cyclic trace in S1 is a cyclic trace in S2. It follows that any cyclic trace in G1 is a cyclic trace in S2. We deduce that any cyclic trace in G1 is a cyclic trace in G2, which concludes the

first part of the proof. The proof for any cyclic trace in G1 is a cyclic trace in G2 $\Rightarrow$ any cyclic trace in S1 is a cyclic trace in S2 is similar.

Proposition 3.3

Consider an LTS $S = <St, L, T, s_o>$ and the graph $G = <Sg, L, Ac, Tg, g_o>$ defined by Proposition 3.3. We first have to prove that G is an AG. The constraints Co, C3, C4 are satisfied by definition of $Ac(g_i)$, for each state g_i in Sg. Constraint C2 is satisfied by definition of the transitions in G. We have to prove that G satisfies constraint C1: Given a state g_i, we have to prove that $\forall$ a $\in$ A, A $\in$ $Ac(g_i)$, there is one and only one g_j such that $g_i{-}a{\rightarrow}g_j$: by definition of G, $\forall$ a $\in$ A, and A $\in$ $Ac(g_i)$, $g_i{-}a{\rightarrow}g_j$ iff $g_j = \{s_j \in St \mid \exists s_m \in g_i \text{ such that } s_m{-}a{\rightarrow}s_j\}^{\varepsilon}$. $\forall$ a $\in$ A, and A $\in$ $Ac(g_i)$, g_j always exists, since $\forall$ a $\in$ L, a $\in$ A, and A $\in$ $Ac(g_i)$, if and only if there exists at least one state s_k in g_i such that $s_k{=}a{\Rightarrow}$ (or a state s_m such that $s_m{-}a{\rightarrow}$). $g_j = \{s_j \in St \mid \exists s_m \in g_i \text{ such that } s_m{-}a{\rightarrow}s_j\}^{\varepsilon}$ is unique, because the set $\{s_j \in St \mid \exists s_m \in g_i \text{ such that } s_m{-}a{\rightarrow}s_j\}$ is unique.

The proof of G = ag(S) follows directly from the definition of G, it is clear that $g_o{=}\sigma{\Rightarrow}g_i$, iff g_i = (s_o after σ). It follows that $Tr(g_o) = Tr(s_o)$ and from the definition of Ac for each state in Sg, $\forall$ $\sigma \in Tr(g_o)$, with $g_o{=}\sigma{\Rightarrow}g_i$, Ac(gi) = Acc($s_o$, σ). For the cyclic traces, from the definition of G we have, $\forall$ $\sigma \in Tr(g_o)$, $g_o{=}\sigma{\Rightarrow}g_o$ iff (s_o after σ) = g_o = $\{s_i \in St \text{ such that } s_o{=}\varepsilon{\Rightarrow}s_i\}$, it follows that a trace σ is a cyclic trace in G, iff σ is a cyclic trace in S.

Proposition 3.4

Consider an AG $G = <Sg, L, Ac, Tg, g_o>$ and the LTS $S = <St, L, T, s_o>$ = lts(G) as defined by Prop. 3.4. A trace $\sigma \in Tr(s_o)$ iff there is a state s_i such that $s_o{=}\sigma{\Rightarrow}s_i$. From the definition of S, the state s_i exists iff there is a state g_i in G such that $g_o{=}\sigma{\Rightarrow}g_i$. It follows that Tr(G) = Tr(S).

By definition of S, (s_o after σ) = $\{s_i\}$ [illegible] $f(g_i)$, iff g_o[illegible] [illegible]ows that Acc(s_o, σ) = $Ac(g_i)$.

From the definition of the transitions in S, $s_{Akl}{-}a{\rightarrow}s_o$ iff $g_k{-}a{\rightarrow}g_o$. Moreover, in this case, there is no transition s_{Akl} –a[illegible] s_o) in S. It follows that (s_o after σ) = $\{s_i \mid s_o{=}\varepsilon{\Rightarrow}s_i\}$ iff [illegible] g_o.

Proposition 3.5

Consider the AGs $G1 = <Sg1, L1, Ac1, Tg1, g1_o>$, $G2 = <Sg2, L2, Ac2, Tg2, g2_o>$, and the LTSs $S1 = <S1, L1, T1, s1_o>$, $S2 = <S2, L2, T2, s2_o>$, such that S1 = lts(G1) and S2 = lts(G2).

1 - a - S1 S2 implies that S1 te S2. By Lemma 3.1 it follows that G1 $_g$ G2,

since G1 =ag(S1) and G2 = ag(S2), .

1 - b - G1 $_g$ G2: by definition, we have Gi = ag(lts(Si)), i = 1, 2. It follows that Tr(Si) = Tr(Gi), i = 1, 2.

By hypothesis, G1 $_g$ G2, therefore Tr(S1) = Tr(S2) = Tr(G1) = Tr(G2). We have to prove that the following relation R = {($s1_i$, $s2_j$): $s1_o$=σ⇒$s1_i$–τ→, $s2_o$=σ⇒$s2_j$–τ→, σ ∈ Tr(S1)} (= R1) {($s1_{Aik}$, $s2A_{jl}$): $s1_{Aik}$ ∈ f($g1_o$ after σ), $s2A_{jl}$ ∈ f($g2_o$ after σ), Aik = Ajl, and σ ∈ Tr(S1)}(= R2) is a strong bisimulation. Note that ($s1_o$, $s2_o$) ∈ R1.

- Consider an element ($s1_i$, $s2_j$) ∈ R1. By definition of R1, for some σ ∈ Tr(S1), $s1_o$=σ⇒$s1_i$–τ→, $s2_o$=σ⇒$s2_j$–τ→. Assume that $s1_i$–τ→$s1_{Aik}$, (–τ→ is the only kind of transition we have for such states by definition of lts(G) in Proposition 3.4). From Proposition 3.4, we have $s1_{Aik}$ ∈ f($g1_o$ after σ). By hypothesis, G1 $_g$ G2, therefore, ∀ σ ∈ Tr(G1), Ac1($g1_o$ after σ) = Ac2($g2_o$ after σ). It follows that there is a state $s2A_{jl}$ ∈ f($g2_o$ after σ), such that Aik = Ajl, and by definition of lts(G) in Proposition 3.4, $s2_j$–τ→$s2_{Ajl}$. Therefore, ($s1_{Aik}$, $s2_{Ajl}$) ∈ R2. The second part of the proof (assume that $s2_j$–τ→$s2_{Ajl}$ ...) is symmetrical.

- Consider an element ($s1_{Aik}$, $s2_{Ajl}$) ∈ R2. It follows that $s1_{Aik}$ ∈ f($g1_o$ after σ), $s2A_{jl}$ ∈ f($g2_o$ after σ), for some σ ∈ Tr(S1), and Aik = Ajl. Now assume that $s1_{Aik}$–a→$s1_l$, (–a→ is the only kind of transition we have for such states by definition of lts(G) in Proposition 3.4). By definition of lts(G), this is possible if and only if $g1_i$–a→$g1_l$. Since G1 $_g$ G2, then we also have $g2_j$–a→$g2_m$ in G2. Since Aik = Ajl and a ∈ Aik, it follows that a ∈ Ajl. By definition of lts(G), we have $s2_{Ajl}$–a→$s2_m$. We have $s1_o$=σ.a⇒$s1_l$–τ→, $s2_o$=σ.a⇒$s2_m$–τ→, for some σ.a ∈ Tr(S1). Therefore, ($s1_l$, $s2_m$) ∈ R1. The second part of the proof (assume that $s2_{Ajl}$–a→$s2_m$ ...) is identical. We have proved that R is a bisimulation. Therefore, if G1 $_g$ G2 then lts(G1) lts(G2). Consequently G1 $_g$ G2 iff lts(G1) lts(G2).

2 - From Proposition 3.2 and Lemma 3.1, S1 and S2 have the same set of cyclic traces, if and only if G1 and G2 have the set of cyclic traces. From (1), G1 $_g$ G2 iff lts(G1) lts(G2). Therefore, G1 $_{cg}$ G2 iff lts(G1) c lts(G2).

3 - From (1), we know that G1 $_g$ G2 iff lts(G1) lts(G2). Due to the correspondence between states of an G1 (respectively G2) and states of lts(G1) (respectively lts(G2), it is obvious that there is a bisimulation between G1 and G2 where each state of G1 is related to one and only state of G2, if and only if there is a bisimulation between lts(G1) and lts(G2) where each state of lts(G1) is related to one and only state of lts(G2).

Proposition 4.1

Consider the AGs G1 = <Sg1, L1, Ac1, Tg1, $g1_o$>, G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>.

We have to prove that Merge(G1, G2) satisfies the consistency constraints Co, C1, C2, C3, and C4.

For that, we have to pr[illegible]e that $<Sg3, L1$ [illegible], $Ac3$, [illegible]$>>$ as [illegible]efined in Definition 4.1 satisfies these requireme[illegible]s:

- Co: By definit[illegible]n of the [illegible]cceptance sets o[illegible] state[illegible]n Sg3, [illegible]e have [illegible] Sg3, [illegible]$3(g_i)$ [illegible]
  because [illegible] and G2 [illegible]re AGs, $\forall g1_j \in S$[illegible], [illegible]$c1(g1$[illegible]$\emptyset$ and [illegible]$2_k \in S$[illegible], [illegible]$(g2_k)$ [illegible].
- C1 and C2: The [illegible]constrai[illegible]s C1 and C2 are s[illegible]ied b[illegible]efiniti[illegible] of th[illegible]tra[illegible]tion [illegible]nction Tg3 and the fact th[illegible] G1 and [illegible]2 are AGs. For [illegible] state [illegible]n Sg3, [illegible]et A in [illegible]3[illegible] and i[illegible]eractio[illegible] a in A [illegible] there is [illegible] and on[illegible] transition lab[illegible] [illegible]y a fr[illegible] this s[illegible]te. For [illegible]ac[illegible]tate i[illegible] g_i in S[illegible]3, ther[illegible] a transi[illegible] from g_i labelled by inte[illegible]on a on[illegible]f $\exists$ A [illegible] $Ac3(g_i)$ [illegible]at a $\in$ [illegible]
- C3 (closure u[illegible]union): $\forall g_i \in$ [illegible]g3, if $g_i =$ [illegible] $g2_k>$, the[illegible] $Ac3(g_i)$ [illegible] $Ac1(g1$[illegible] x $Ac2(g2_k)$, [illegible] A1, A[illegible] $\in Ac3($[illegible]
  then A1[illegible] A1[illegible] $A2_{k1}$ and A[illegible]$_{j2}$ [illegible]2_{k2}, wher[illegible] $Ac1(g1_j)$ and [illegible] $\in Ac2($[illegible] by defi[illegible]tion of Ac3. Since Ac1 and [illegible]c2 satisfy C3, we have ($A1_{j1}$ [illegible] $\in Ac1(g1_j)$[illegible] ($A2_{k1}$ [illegible]2_{k2}) $\in Ac2($[illegible]ows [illegible]at ($A1_{j1}$ $A1_{j2}$) ($A2$[illegible] cases w[illegible]ere $g_i = g1_j$, or $g_i = g2_k$, the [illegible]roof is obvious since Ac1 and Ac2 satisfy C3 by hypothesis. The proof of satisfaction of C4 is similar to the proof for C3.

$<Sg3, L1$ $L2, Ac3, Tg$[illegible]$>$ is an AG. Consequently, Merge(G1, G2) = reachable($<Sg3, L1$ [illegible]2, Ac3, Tg3, $<g1_o, g2_o>>$) is an AG.

Proposition 4.2

Let G1 = $<Sg1, L1, Ac1, Tg1, g1_o>$, G2 = $<Sg2, L2, Ac2, Tg2, g2_o>$ and G3 = $<Sg3, L3, Ac3, Tg3, g3_o>$.

(a) Merge(G1, G2) $=_g$ Merge(G2, G1):

let Sg4 and Sg5 be the set of states of Merge(G1, G2) an[illegible] Merge(G2, G1) , respectiv[illegible]y. The relation $\{(<g1_i, g2_j>, <g2_j, g1_i>): g1_i \in Sg1, g2_j \in Sg2, <g1_i,$ [illegible]$_j> \in Sg4$ and $<g2_j, g1_i> \in$ [illegible]5$\}$ $\{(g_i, g_i'): g_i \in Sg4, g_i' \in Sg5$, and $g_i = g_i'\}$ is a bisimulation c[illegible]ntaining the pair ($<g1_o, g2$[illegible], $<g2_o, g1_o>$) and each state of Merge(G1, G2) is related to one and only [illegible]ate of Merge(G2, G1) and vi[illegible] et versa. The AGs G1 and G2 have symmetrical roles in the [illegible]efinition of Merge(G1, G2).

(b) Merge(Merge(G1, G2), G3) $=_g$ Merge(G1, Merge(G2, G3)):

let Sg4 and Sg5 be the set of states of Merge(Merge(G1, G2), G[illegible] and Merge(G1, Merge(G[illegible]G3)), respectively. The relation $\{(<<g1_i, g2_j>, g3_k>, <g1_i, <g2_j, g3_k>>)$[illegible]$1_i \in Sg1, g2_j \in S$[illegible]2, $g3_k \in Sg3$, $<<<g1_i, g2_j>, g3_k> \in Sg4$ and $<g1_i, <g2_j, g3_k>> \in Sg5\}$ $\{(g_i, g_i')$: [illegible] Sg5, and $g_i = g_i'\}$ is a bisimulation containing the pair ($<<g1_o, g2_o>, g3_o>, <g1_o, <g2_o, g3_o>>$) and each state in Sg4 is related to one and only state of Sg5 and vice et versa.

Proposition 4.3

Given the AGs G1 = $<Sg1, L1, Ac1, Tg1, g1_o>$, G2 = $<Sg2, L2, Ac2, Tg2, g2_o>$, we have to prove that Merge(G1, G2) ext_g G1:

a - Consider an arbitrary trace σ ∈ G1 with $g1_o$=σ⇒$g1_i$. From definition of Merge(G1, G2), ∃g_i ∈ Sg3 such that <$g1_o$, $g2_o$>=σ⇒g_i, where g_i = $g1_i$ or g_i = <$g1_i$, $g2_j$>, for some state $g2_j$ ∈ Sg2. Consequently, Tr(G1)   Tr(Merge(G1, G2)).

b - From (a) above, if $g1_o$=σ⇒$g1_i$, then ∃g_i ∈ Sg3 such that <$g1_o$, $g2_o$>=σ⇒g_i, where g_i = $g1_i$ or g_i =<$g1_i$, $g2_j$>, for some state $g2_j$ ∈ [illegible] of Merge we have Ac3(g_i) = Ac1($g1_i$), it follows that Ac3(g_i)   Ac1($g1_i$). If g_i = <$g1_i$, $g2_j$> for some $g2_j$ ∈ Sg2, by definition of Merge we have Ac3(g_i) = {X1   X2 |X1 ∈ Ac1($g1_i$) [illegible] Ac2($g2_j$)}. It follows that Ac3(g_i)   Ac1($g1_i$), since for any X∈ Ac3(g_i), there is an X1∈ Ac1($g1_i$) such that X1   X. Consequently, Merge(G1, G2) $ext_σ$ G1.

Proposition 4.4

Let G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>.

Consider an elementary cyclic trace σ = a1.a2...an in G1. It follows that ∃$g1_i$, $g1_{i+1}$,..., $g1_{i+n-2}$ in g1, such that $g1_o$=a1⇒$g1_i$, $g1_i$=a2⇒$g1_{i+1}$,..., $g1_{i+n-2}$=an⇒$g1_o$, with $g1_j$ $g1_o$, for j = i, ..., i+n-2.

Sufficient condition:

σ   Tr(G2), it follows that σ = σ'.aj.σ" and $g2_o$=a1⇒$g2_k$, $g2_k$=a2⇒$g2_{k+1}$, ..., $g2_{k+j-3}$=aj-1⇒$g2_{k+j-2}$, and $g2_{k+j-2}$ aj⇒ for some 1 j n. From the definition of Merge(G1, G2), we have <$g1_o$,$g2_o$>=a1⇒ <$g1_i$,$g2_k$>, <$g1_i$,$g2_k$>=a2⇒<$g1_{i+1}$,$g2_{k+1}$>,..,<$g1_{i+j-3}$,$g2_{k+j-3}$>=aj-1⇒ <$g1_{i+j-2}$,$g2_{k+j-2}$>, <$g1_{i+j-2}$,$g2_{k+j-2}$>=aj⇒$g1_{i+j-1}$, ..., $g1_{i+n-2}$=an⇒<$g1_o$,$g2_o$> in Merge(G1, G2), which means that σ is a cyclic trace in Merge(G1, G2).

σ is a cyclic trace in G2, it follows ∃$g2_k$, $g2_{k+1}$, ..., $g2_{k+n-2}$ in Sg2 such that $g2_o$=a1⇒$g2_k$, $g2_k$=a2⇒ $g2_{k+1}$, ..., $g2_{k+n-2}$=an⇒$g2_o$. From the definition of Merge(G1, G2), we have <$g1_o$,$g2_o$>=a1⇒<$g1_i$,$g2_k$>, <$g1_i$,$g2_k$>=a2⇒<$g1_{i+1}$,$g2_{k+1}$>, ..., and <$g1_{i+n-2}$,$g2_{k+n-2}$>=an-2⇒<$g1_o$,$g2_o$> in Merge(G1, G2), which means that σ is a cyclic trace in Merge(G1, G2).

Necessary Condition:

Assume that σ∈ Tr(G2) and σ is not a cyclic trace in G2. It follows that ∃$g2_k$, such that $g2_o$=σ⇒$g2_k$, with $g2_k$ $g2_o$. By definition of Merge(G1, G2), we have <$g1_o$, $g2_o$>=σ⇒<$g1_o$, $g2_k$>, with <$g1_o$,$g2_k$> <$g1_o$,$g2_o$>. Consequently, σ is not a cyclic in Merge(G1, G2), which ends the proof that (σ   Tr(G2) or σ is a cyclic trace in G2) is a necessary condition.

Proposition 4.5

Let G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>

1 - Equivalence between (a) and (b): we know that Merge(G1, G2) preserves the cyclic traces of G1, iff any elementary cyclic trace in G1 is preserved, as cyclic trace, in Merge(G1, G2). From Proposition 4.4, we know that any elementary cyclic trace σ in G1 is a cyclic trace in Merge(G1, G2), iff σ is a

cyclic trace in G2 or σ Tr(G2). It follows that Merge(G1, G2) preserves the cyclic traces of G1 iff any elementary cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G2).

2 - Equivalence between (b) and (c):

2- 1 - (c) implies (b): obvious since any elementary cyclic trace is a cyclic trace.

2 - 2 - (b) implies (c): assume that any elementary cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G2) and consider an arbitrary cyclic trace σ in G1. Any cyclic trace results from the concatenation of elementary cyclic traces, therefore σ = σ1.σ2...σn, with σi as elementary cyclic trace in G1, for i = 1, ..., n. σi is an elementary cyclic in G1, by hypothesis, it follows that σi is a cyclic trace in G2 or σi Tr(G2), for i = 1, ..., n. Assume that σi is a cyclic trace in G2, for i = 1, ..., n, it follows that σ = σ1.σ2...σn is a cyclic trace in G2 (concatenation of cyclic traces is a cyclic trace). Now assume that σi, for i =1, ...j -1, are cyclic traces in G2 and σj Tr(G2) with 1 j n . It follows that σ1.σ2...σj-1 is a cyclic trace in G2, but σ1.σ2...σj-1.σj Tr(G2), which means that σ Tr(G2). Therefore, (b) implies (c).

Consequently, the statements (a), (b) and (c) in Proposition 4.5 are equivalent.

Proposition 4.6

Let G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>.

Consider σ = a1.a2....an, an arbitrary elementary cyclic trace in Merge(G1, G2). By definition of the elementary cyclic trace, we have <$g1_o$, $g2_o$>=a1⇒g_{i1}=a2⇒g_{i2}...g_{in-1}=an⇒<$g1_o$, $g2_o$> with g_{ij} <$g1_o$, $g2_o$>, for j = 1, ..., n-1. From the Definition of Merge, we have the following three cases:

(a) g_{ij} = <$g1_{ij}$, $g2_{ij}$>, with <$g1_{ij}$, $g2_{ij}$> <$g1_o$, $g2_o$> for j = 1,..., n-1, which implies that $g1_o$=a1⇒$g1_{i1}$=a2⇒$g1_{i2}$...$g1_{in-1}$=an⇒$g1_o$ and $g2_o$=a1⇒$g2_{i1}$=a2⇒$g2_{i2}$... $g2_{in-1}$=an⇒$g2_o$. Therefore, σ is a cyclic trace in G1 and G2.

(b) g_{ij} = <$g1_{ij}$, $g2_{ij}$> with <$g1_{ij}$, $g2_{ij}$> <$g1_o$, $g2_o$> , for j = 1,..., k, (for a certain k) and g_{ij} = $g1_{ij}$ ($g1_o$), for j = k+1,..., n-1, which means that $g1_o$=a1⇒$g1_{i1}$=a2⇒$g1_{i2}$...$g1_{in-1}$=an⇒$g1_o$. Therefore, σ is a cyclic trace in G1.

(c) g_{ij} = <$g1_{ij}$, $g2_{ij}$> with <$g1_{ij}$, $g2_{ij}$> <$g1_o$, $g2_o$> , for j = 1,..., k, (for a certain k) and g_{ij} = $g2_{ij}$ ($g2_o$), for j = k+1,..., n-1, which means that $g2_o$=a1⇒$g2_{i1}$=a2⇒$g2_{i2}$...$g2_{in-1}$=an⇒$g2_o$. Therefore σ is a cyclic trace in G2.

Consequently, σ is a cyclic trace in G1 or G2.

Proposition 4.7

Let G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, Ac2, Tg2, $g2_o$>.

(a) σ is a cyclic in Merge(G1, G2): $\sigma = \sigma1.\sigma2...\sigma n.\sigma n+1$, with σi as elementary cyclic trace in Merge(G1, G2), for i =1, ..., n+1, for a certain integer n. From Proposition 4.6, σi as a cyclic trace in G1 or G2, for i =1, ..., n+1. Therefore, σi is a cyclic trace in G1 or G2, for i=1,..., n, and ($\sigma n+1 \in Tr(G1)$ or $\sigma n+1 \in Tr(G2)$).

(b) σ is a noncyclic in Merge(G1, G2): $\sigma = \sigma'.a1.a2...am$ with $<g1_o, g2_o>=\sigma' \Rightarrow g1_o, g2_o>=a1 \Rightarrow g_{i1}=a2 \Rightarrow g_{i2} ... g_{im-1}=an \Rightarrow g_{im}$ with g_{ij} $<g1_o, g2_o>$, for j = 1, ..., m. σ' is a cyclic trace in Merge(G1, G2). Therefore, $\sigma' = \sigma'1.\sigma'2...\sigma'n$, with $\sigma'i$ as elementary cyclic trace in Merge(G1, G2), for i =1, ..., n, for a certain integer n. From Proposition 4.6, $\sigma'i$ as a cyclic trace in G1 or G2, for i =1, ..., n.

We have $<g1_o, g2_o> =a1 \Rightarrow g_{i1}=a2 \Rightarrow g_{i2} ...\ g_{im-1}=an \Rightarrow g_{im}$ with g_{ij} $<g1_o, g2_o>$, for j = 1, ..., m. From the definition of Merge, we have the following three cases:

(a) $g_{ij} = <g1_{ij}, g2_{ij}>$, with $<g1_{ij}, g2_{ij}>$ $<g1_o, g2_o>$ for j = 1,..., m, which means that $g1_o=a1 \Rightarrow g1_{i1}=a2 \Rightarrow g1_{i2}...g1_{im-1}=am \Rightarrow g1_m$ and $g2_o=a1 \Rightarrow g2_{i1}=a2 \Rightarrow g2_{i2}...g2_{im-1}=am \Rightarrow g2_m$. Therefore, $a1.a2...am \in Tr(G1)$ and $a1.a2...am \in Tr(G2)$.

(b) $g_{ij} = <g1_{ij}, g2_{ij}>$ with $<g1_{ij}, g2_{ij}>$ $<g1_o, g2_o>$, for j = 1,..., k, (for a certain k) and $g_{ij} = g1_{ij}$ ($g1_o$), for j = k+1,..., n-1, which means that $g1_o=a1 \Rightarrow g1_{i1}=a2 \Rightarrow g1_{i2}...g1_{im-1}=am \Rightarrow g1_m$. Therefore, $a1.a2...am \in Tr(G1)$.

(c) $g_{ij} = <g1_{ij}, g2_{ij}>$ with $<g1_{ij}, g2_{ij}>$ $<g1_o, g2_o>$, for j = 1,..., k, (for a certain k) and $g_{ij} = g2_{ij}$ ($g2_o$), for j = k+1,..., n-1, which means that $g2_o=a1 \Rightarrow g2_{i1}=a2 \Rightarrow g2_{i2}...g2_{im-1}=am \Rightarrow g2_m$. Therefore, $a1.a2...am \in Tr(G2)$.

Consequently, any trace σ of Merge(G1, G2) may be written as σ = [illegible].$\sigma2...\sigma n.\sigma n+1$, with σi as a cyclic trace in G1 or G2, for i =1, ..., n, and ($\sigma n+1 \in$ [illegible] Tr(G2)).

Theorem 4.1

Let G1 = <Sg1, L1, Ac1, Tg1, $g1_o$> and G2 = <Sg2, L2, A[illegible]g2, $g2_o$>.
From Proposition 4.3, we have Merge(G1, G2) ext_g Gi, i[illegible]2.
From Proposition 4.5, Merge(G1, G2) preserves the cyc[illegible]
any cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G[illegible]).
It follows that Merge(G1, G2) is a cyclic extension of G1 an[illegible]
any cyclic trace σ in G1 is a cyclic trace in G2 or σ Tr(G2), and reci[illegible]ocally.

Now, we have to prove that Merge(G1, G2) is the least common cyclic extension of G1 and G2. For that, we consider an arbitrary AG G4 = <Sg4, L4, Ac4, Tg4, $g4_o$> such that G4 $extc_g$ G1, G4 $extc_g$ G2 and we will prove that G4 $extc_g$ Merge(G1, G2).

First, we have to prove that any cyclic trace in Merge(G1, G2) is a cyclic trace in G4. Consider a cyclic trace σ in Merge(G1, G2). $\sigma = \sigma1.\sigma2...\sigma n$ with $\sigma1, \sigma2, ..., \sigma n$ as elementary cyclic traces in Merge(G1, G2).

By Proposition 4.6, it follows that σi is a cyclic trace in G1 or G2, for i = 1, ..., n. We have σi as a cyclic trace in G1 or G2, for i = 1, ..., n. It follows that σi is a cyclic trace in G4, for i = 1, ..., n, since G4 is a cyclic extension of G1 and G2. Consequently, σ is a cyclic trace in G4 (concatenation of cyclic traces is a cyclic trace)

Secondly, we have to prove that G4 ext_g Merge(G1, G2):

(1) Consider an arbitrary trace σ in Merge(G1, G2). The trace σ can be written as σ = σ1.σ2...σn-1.σn with σi as cyclic trace in G1 or G2, for i = 1, ..., n-1, and σn ∈ Tr(G1) or σn ∈ Tr(G2). G4 extc_g G1 and G4 extc_g G2, it follows that any trace of G1 (respectively G2) is a trace of G4, and any cyclic trace in G1 (respectively G2) is a cyclic trace in G4, it follows that σi is a cyclic trace in G4, for i = 1, ..., n-1, and σn ∈ Tr(G4). We deduce that σ = σ1.σ2...σn-1.σn ∈ Tr(G4).

(2) Consider an arbitrary trace σ in Merge(G1, G2): as previously, t[illegible]ace σ can be written as σ [illegible] σ1.σ2...σn-1.σn with σi as cycli[illegible]race in G1 or G2, for i = 1[illegible], and σn ∈ Tr(G1) or σn ∈ T[illegible] We have deduced that σi is a c[illegible]lic trace in G4, for i = 1,...[illegible], and σn∈ Tr(G4). σ∈ Tr(Merge([illegible])), it follows that ∃g_i in [illegible]erge(G1, [illegible] Since [illegible] σ2,..., σn-1 are (ele[illegible]entary) cy[illegible]c traces in Merge(G1, G2), i[illegible]llow[illegible]hat <$g1_o$, $g2_o$>=σn⇒g_i. S[illegible] reasoning for G4, [illegible]4_j in [illegible] such that $g4_o$=σ⇒$g4_j$ and $g4_o$=σn⇒g[illegible]n ∈ Tr(G1) and σn [illegible] Tr(G2) [illegible] deduce that ∃$g1_i$ in G1 su[illegible]ha[illegible]1_o=σn⇒$g1_i$, and by definitio[illegible] Merge, [illegible] Ac1($g1_i$). We have G4 ex[illegible] at Ac4($g4_j$) [illegible] Ac[illegible] = Ac1($g1_i$). Reciprocally, if σn [illegible] Tr(G2) and σn [illegible] Tr(G1). If [illegible](G1) and σn ∈ Tr(G2), ∃$g1_i$ in G1 [illegible] ∃$g2_j$ in G2 such that $g1_o$=σn⇒$g1_i$, and $g2_o$=σn⇒$g2_j$, [illegible]d by definition of Merge, g_i = <$g1$[illegible]> and Ac3(g_i) = {X1 [illegible] X2 | X1 ∈ Ac1($g1_i$) and X2 ∈ Ac2($g2_j$)}. [illegible] Ac1($g1_i$) and Ac4($g4_j$) [illegible] Ac2($g2_j$). It follows that A[illegible]4($g4_j$) [illegible]3(g_i), which e[illegible]s the second part of the proof G4 ext_g Merge(G1, G2).

Consequently, G4 [illegible] Merge(G1, G2) and Merge(an arbitrary el[illegible]entary cycli[illegible] trace in Merge(G1, G
2
). By
t i o n of a elementary cyclic tr a [illegible], we ha[illegible] <g1o,
..g4in-1=an <g1o, g2o> with g4ij [illegible]<g1o, [illegible]o>, f o [illegible] j =
1, ..., n-1. From the Defi ition of Merge, we have the
lowing three cases:
(a g4ij = <g1ij, g2 i j>, w i t h
<g1

ij, g2ij>   <g1o, g2o>, for j = 1,..., n-1, it follows that <g3o,   g2o>=a1ﬁg3i1, g2i1>=a2ﬁg3i2, g2i2>..<g3in-1, g2in-1>=anﬁ<g3o, g2o> in Merge(G3, G2) with   <g3ij, g2ij> <g1o, g2o> for j = 1,..., n-1, since an arbitrary elementary cyclic trace in Merge(G1, G2). By definition of an elementary cyclic trace, we have $<g1_o, g2_o>=a1 \Rightarrow g4_{i1}=a2 \Rightarrow g4_{i2}...g4_{in-1}=an \Rightarrow <g1_o, g2_o>$ with $g4_{ij}$ $<g1_o, g2_o>$, for j = 1, ..., n-1. From the Definition of Merge, we have the following three cases:

(a) $g4_{ij} = <g1_{ij}, g2_{ij}>$, with $<g1_{ij}, g2_{ij}>$ $<g1_o, g2_o>$, for j = 1,..., n-1, it follows that $<g3_o, g2_o>=a1 \Rightarrow g3_{i1}, g2_{i1}>=a2 \Rightarrow g3_{i2}, g2_{i2}>..<g3_{in-1}, g2_{in-1}>=an \Rightarrow <g3_o, g2_o>$ in Merge(G3, G2) with $<g3_{ij}, g2_{ij}>$ $<g1_o, g2_o>$ for j = 1,..., n-1, since $g3_{ij} = g3_o$ iff $g1_{ij} = g1_o$, for j = 1,..., n-1 (G1 and G3 have the same cyclic traces). Therefore, σ is an elementary cyclic in Merge(G3, G2).

(b) $g4_{ij} =$ $<g1_{ij}, g2_{ij}>$ with $<g1_{ij}, g2_{ij}>$ $<g1_o, g2_o>$, for j = 1,..., k, (for a certain k) and $g4_{ij} = g1_{ij}$ ($g1_o$), for j = k+1,..., n-1, it follows that $<g3_o, g2_o>=a1 \Rightarrow <g3_{i1}, g2_{i1}>...<g3_{ik-1}, g2_{ik-1}>=ak \Rightarrow <g3_{ik}, g2_{ik}>=ak+1 \Rightarrow g3_{ik+1}...g3_{in-1}=an \Rightarrow <g3_o, g2_o>$in Merge(G3, G2) with $<g3_{ij}, g2_{ij}>$ $<g3_o, g2_o>$ for j = 1, ..., k, and $g3_{ij}$ $g3_o$, for j = k+1, ..., n-1, since $g3_{ij} = g3_o$ iff $g1_{ij} = g1_o$, for j = 1, ..., n-1 ((G1 and G3 have the same cyclic traces)). Therefore, $<g3_o, g2_o>=a1 \Rightarrow g5_{i1}=a2 \Rightarrow$ $g5_{i2}$ ... $g5_{in-1}=an \Rightarrow <g3_o, g2_o>$ with $g5_{ij}$ $<g1_o, g2_o>$, for j = 1, ..., n-1, which means that σ is an elementary cyclic in Merge(G3, G2).

(c) $g4_{ij} =$ $<g1_{ij}, g2_{ij}>$ with $<g1_{ij}, g2_{ij}>$ $<g1_o, g2_o>$, for j = 1,..., k, (for a certain k) and $g4_{ij} = g2_{ij}$ ($g2_o$), for j = k+1,..., n-1, it follows that $<g3_o, g2_o>=a1 \Rightarrow <g3_{i1}, g2_{i1}>...$ $<g3_{ik-1}, g2_{ik-1}>=ak \Rightarrow <g3_{ik}, g2_{ik}>=ak+1 \Rightarrow g2_{ik+1}...g2_{in-1}=an \Rightarrow <g3_o, g2_o>$ in Merge(G3, G2) with $<g3_{ij}, g2_{ij}>$ $<g3_o, g2_o>$ for j = 1,..., k, since $g3_{ij} = g3_o$ iff $g1_{ij} = g1_o$ for j = 1,..., k ((G1 and G3 have the same cyclic traces)) and $g2_{ij}$ $g2_o$, for j = k+1,..., n-1. Therefore, $<g3_o, g2_o>=a1 \Rightarrow g5_{i1}=a2 \Rightarrow g5_{i2}...$ $g5_{in-1}=an \Rightarrow <g3_o, g2_o>$ with $g5_{ij}$ $<g1_o, g2_o>$, for j = 1, ..., n-1, which means that σ is an elementary cyclic in Merge(G3, G2).

The proof for any elementary cyclic trace in Merge(G3, G2) is an elementary cyclic trace in Merge(G1, G2) is symmetrical. Consequently, Merge(G1, G2) and Merge(G3, G2) have the same set of (elementary) cyclic traces.

2 - Merge(G1, G2) $_g$ Merge(G3, G2):

2 - 1 - Tr(Merge(G1, G2)) = Tr(Merge(G3, G2)):

Consider a trace σ ∈ Tr(Merge(G1, G2)). σ = σ1.σ2...σn.σn+1, with σi as elementary cyclic trace in Merge(G1, G2), for i =1, ..., n, and (σn+1 ∈ Tr(G1) or σn+1 ∈ Tr(G2)). It follows, from (1) above, that σi is an elementary cyclic trace in Merge(G3, G2), for i =1, ..., n. Merge(G3, G2) ext_g G3 and G2 and G1 c_g G3, we deduce that (σn+1 ∈ Tr(G3) or σn+1 ∈ Tr(G2)). Therefore, σ = σ1.σ2...σn.σn+1 ∈

Tr(Merge(G3, G2). The proof for any trace σ of Merge(G3, G2) is a trace of Merge(G3, G2) is symmetrical.

2 - 2 - ∀ σ ∈ Tr(Merge(G1, G2)), Ac4(<$g1_o$, $g2_o$> after σ) = Ac5(<$g3_o$, $g2_o$> after σ):

Consider a trace σ ∈ Tr(Merge(G1, G2)). σ = σ1.σ2...σn.σn+[illegible]σi as elementar[illegible]cyclic t[illegible] in Merge(G1, G2) and Merge(G3, G2), for i =1, ..., n, and (σ[illegible]+1 ∈ Tr(G1) (a[illegible]σn+1 ∈ T[illegible]G3)) or σn[illegible] Tr(G2)). Therefore, <$g1_o$, $g2_o$> after σ = <$g1_o$, $g2_o$> a[illegible]r σn+1 and <$g3_o$,[illegible]ter σ = <$g3_o$, [illegible]> after σn+1. Ac4(<$g1_o$, $g2_o$> after σ) = Ac5(<$g1_o$, $g2_o$> after σ), iff Ac4(<$g1_o$, $g2_o$> after σn+1) = Ac5(<$g3_o$, $g2_o$> after σn+1). We have three cases:

- σn+1 ∈ Tr(G1) (σn+1 ∈ Tr(G3)): Ac4(<$g1_o$, $g2_o$> after σn+1) = Ac1($g1_o$ aft[illegible]) = Ac3($g3_o$ [illegible] σn+1) = Ac5(<$g3_o$, $g2_o$> after σn+1), since G1 c_g G3.
- σn+1 ∈ Tr(G2): Ac4(<$g1_o$, $g2_o$> after σn+1) = Ac2($g2_o$ after σn+1) = Ac5(<$g3_o$, $g2_o$> after σn+1).
- σn+1 ∈ Tr([illegible]+1 ∈ Tr(G3)) a[illegible] σn+1 ∈ Tr(G2): Ac4(<$g1_o$, $g2_o$> after σn+1) = {X1 X2 | X1 ∈ Ac1($g1_o$ [illegible]er σn+1) an[illegible]X2 ∈[illegible]$_o$ after σn+1)}[illegible]nd Ac5(<$g3_o$, $g2_o$> after σn+1) = {X3 X2 | X3 ∈ Ac3($g3_o$ after σn+1) and[illegible]2 ∈ Ac2(g2[illegible]fter σn[illegible])}. Since G1 c_g G3, Ac1($g1_o$ after σn+1) = Ac3($g3_o$ after σn+1). It follows that Ac4(<$g1_o$, $g2_o$> after σn+1) = Ac5(<$g3_o$, $g2_o$> after σn+1).

Merge(G1, G2) $_g$ Merge(G3, G2) and a trace σ is cyclic in Merge(G1, G2) iff σ is cyclic in Merge(G3, G2). Consequently, Merge(G1, G2) c_g Merge(G3, G2).