ソフトウェア工学 110－2
（1996. 7. 25）

A Dynamic Extention for the Specifications of Distributed Systems

Issam A. Hamid
[1] Tohoku University of Art & Design
Department of Information Design
200 Kamisakurada, Yamagata, JAPAN

In this paper, we describe an approach for extending distributed system specifications. These specifications are structured as a parallel composition of subsystem specifications. The approach consists of building a new specification Snew by adding a new behavior described by a specification Sadded to a specification Sold with preservation of the properties of Sold and Sadded as well as the structure of Sold. Snew has all the properties of Sold and Sadded, if Snew can perform whatever Sold (and Sadded) can perform, and it does not block where Sold (or Sadded) does not block. We apply our approach for extending the functionality of a basic Automatic Teller Machine.

分散システム仕様のための動的拡張

アイサム. A. ハミド
東北芸術工科大学
情報デザイン学科
山形市上桜田２００番地

本論では、分散システムの仕様を拡張するためのアプローチについて述べる。これらの仕様は、サブシステム仕様の並列的な構成として構造化される。このアプローチでは、仕様Soldに対して、仕様Saddedによって記述される新たな振る舞いを追加することにより、新たな仕様Snewを構築する。仕様Snewでは、SoldとSaddedのプロパティは、Soldの構造と同様に維持される。もし、Sold（かつSadded）が実行可能なもの全てをSnewが実行できるものとし、Sold（またはSadded）がブロック化しない場所で、Snewがブロック化しないとすれば、SnewはSoldとSaddedのプロパティの全てを持つことになる。我々は、上記のアプローチを、基本的なAutomatic Teller Machineの機能性の拡張に適用する。

[1] This research is done in part as international scientific joint research, sponsored by the ministry of Education, Science and Culture of Japan, with the University of Montreal, department of Information and Operational Research.

1 Introduction

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

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

The combination should preserve the semantics properties of each single specification. For instance, the addition of a new function to a telephone system specification should not disturb the semantics properties of the telephone system specification and the semantics properties of the new function. In the context of distributed systems, preserving semantic properties may, for instance, mean that the combined specification exhibits at least the behaviors of the original ones without introducing additional failures for these behaviors. This is captured by the formal relation between specifications, called extension, introduced in [Brin 86]. Informally, a specification S2 extends a specification S1, if and only if, S2 allows any sequence of actions that S1 allows, and S2 can only refuse what S1 can refuse, after a given sequence of actions allowed by S1.

In this paper, we propose an incremental specification approach, which consists of merging two given specifications Sold and Sadded into a specification Snew, such that Snew extends Sold and Snew extends Sadded. Moreover, in the case of minimal cyclic traces of Sold or Sadded, Snew transforms into Snew, and may exhibit, in a recursive manner, behaviors of Sold and Sadded. We consider distributed system specifications, which may consist of a parallel combination of subsystem specifications. The incremental specification approach preserves such structure. Therefore, the designer does not have to redesign it. The approach for merging structured specifications described in this paper, is based on our approach for merging monolithic specifications described in [Hami 95].

The remainder of the paper is structured as follows. Section 2 introduces the labelled transition systems model [Kell 76] and some definitions used in this paper. In Section 3, we summarize the principle and properties of the approach for merging monolithic specifications. In Section 4, our approach for merging structured specifications is described. In Section 5, we apply our approach for extending the functionality of a basic Automatic Teller Machine. In Section 6 our approach is compared to related ones. In Section 7, we conclude.

2 Labelled Transition Systems

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

2.1 Definitions

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

Definition 2.1 [Kell 76]

An LTS TS is a quadruple <S, L, T, So>, where

- S is a (countable) nonempty set of states.
- L is a (countable) set of observable actions.
- T: S x L $\cup\{\tau\} \rightarrow$ S is a transition relation, where a transition from a state Si to state Sj by an action μ ($\mu \in L \cup \{\tau\}$) is denoted by Si-$\mu\rightarrow$Sj.
  τ represents the internal, nonobservable action.
- So is the initial state of TS.

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

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

Definition 2.2 (Cyclic Trace)

Given an LTS TS = <S, L, T, So>, a trace s is cyclic, iff (So after σ) = {Si $\in$ S such So=$\varepsilon \Rightarrow$ Si}.

Definition 2.3 (Minimal Cyclic Trace)
Given an LTS TS = <S, L, T, So>, σ is a minimal cyclic trace, iff σ is a cyclic trace, and ∃σ' (≠ ε) and σ'' (≠ e) such that σ = σ'.σ'' and σ' is cyclic trace in TS.

P−μ1...μn→Q: ∃ Pi (0 ≤ i ≤ n) such that
$P = P_0 - \mu_1 \rightarrow P_1 \ldots P_{n-1} - \mu_n \rightarrow P_n = Q$
$P - \mu_1 \ldots \mu_n \rightarrow$: ∃ Q such that $P - \mu_1 \ldots \mu_n \rightarrow Q$
P=ε⇒Q: P ≡ Q or ∃ n ≥ 1 $P - \tau^n \rightarrow Q$
P=a⇒Q: $\exists P_1, P_2$ such that $P = \varepsilon \Rightarrow P_1 - a \rightarrow P_2 = \varepsilon \Rightarrow Q$
$P = a_1 \ldots an \Rightarrow Q$: $\exists P_i$ (0 ≤ i ≤ n) such that
$P = Po = a_1 \Rightarrow P_1 = a_1 \Rightarrow ..an \Rightarrow Pn = Q$
P=σ⇒: ∃Q such that P=σ⇒Q
P≠σ⇒: not (P=σ⇒)
Tr(P): {σ ∈ L* | P=s⇒}
out(P, s): {a ∈ L| σ.a ∈ Tr(P)}
initials(P): out(P, ε)
P after σ: {Q | P=σ⇒Q}
Acc(P, σ): {X | ∃Q ∈ (P after σ), such that
initials(Q) ⊆ X ⊆out(P, σ)}
where μ, μ_i ∪ L {τ}; a, ai ∈ L; P, Q, Pi, Qi represent states; ε represents the empty trace;
σ = a1.a2... an, where "." denotes the concatenation of actions or sequence of actions (traces).

Table 1. LTS notations

2.2 Operations on Labelled Transition Systems

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

Parallel composition: $B1|_{\{a1, \ldots, an\}}B2$
If B1−a→B1' and a { {a1, ..., an}, then B1 |{a1, ..., an} B2−a→B1' |{a1, ..., an} B2,
If B2−a→B2' and a { {a1, ..., an}, then B1 |{a1, ..., an} B2−a→B1 |{a1, ..., an} B2',
If B2−a→B2' and B1−a→B1' and a ∈ {a1, ..., an}, then $B1\,|_{\{a1, \ldots, an\}}\, B2 - a \rightarrow B1'\,|_{\{a1, \ldots, an\}}\, B2'$.
Hiding operator: B\{a1, ..., am}
If B−a→B' and a { {a1, ..., am}, then B\{a1, ..., am}−a→ B'\{a1, ..., am},
If B−a→B' and a ∈ {a1, ..., am}, then B\{a1, ..., am}−τ →B'\{a1, ..., am}.

2.3 The extension relation

Intuitively, different LTSs may describe the same observable behavior. Therefore different equivalence relations have been defined based on the notion of observable behavior. They range from the relatively coarse trace equivalence to the much finer strong bisimulation equivalence [DeNi 87]. However, for our considerations, one does not need equivalence relations, but rather ordering relationships. Among them, we note the reduction and extension relation as defined in [Brin 86]. These relations may serve different purposes during the specification life cycle. The extension relation is most appropriate for our purpose of compatible enrichment of specifications. Informally, S2 extends S1, if and only if, S2 allows any sequence of actions that S1 allows, and S2 can only refuse what S1 can refuse, after a given sequence of actions allowed by S1.

Definition 2.4 [Brin 86]
S2 extends S1, written S2 ext S1, iff
(a) Tr(S1) ⊆ Tr(S2), and
(b) ∀ σ ∈ Tr(S1) , ∀ A ⊆ L,
if S2' such that S2=σ⇒S2' and S2'≠a⇒ , ∀ a ∈ A,
then S1' such that S1=σ⇒S1' and S1'≠a⇒ , ∀ a ∈ A.

3 Merging monolithic specifications

In this section, we consider monolithic specifications [Viss 88]. A monolithic specification has no internal structure and is defined directly in terms of some allowed ordering of actions. A monolithic specification is represented by a single LTS.
Given two LTSs, S1 and S2, we want to construct systematically an LTS S3, such that S3 extends S1, and S3 extends S2. Moreover, in the case of minimal cyclic traces of S1 or S2, S3 transforms into S3, and may exhibit, in a recursive manner, behaviors of S1 and S2. Note that the usual choice operators defined for LOTOS [ISO 8807] and CCS [Miln 89] for instance, do not allow such combination of specifications as shown in Figure 1.

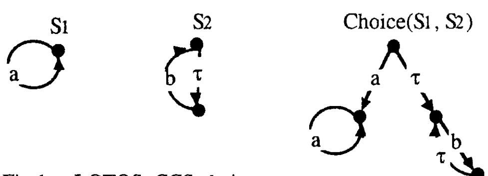


Fig 1. LOTOS, CCS choice operator

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

Definition 3.1
An AG G is 5-tuple <Sg, L, Ac, Tg, Sgo>, where
- Sg is a (countable) nonempty set of states.
- L is a (countable) nonempty set of events.
- Ac: Sg → P(P(L)) is a mapping from Sg to sets of subsets of L.
  Ac(Sgi) is called the acceptance set of Sgi.
- Tg: Sg x L → Sg is a transition function, where a

transition from

state Sgi to state Sgj by an action a ($a \in L$) is denoted by Sgi–a→Sgj.

- Sgo is the initial state of G.

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

Definition 3.2

Given an AG G = <Sg, L, Ac, Tg, Sgo> and an LTS S = <St, L, T, So>, we say that G is a corresponding AG of S, written <G, S> Œ AGR, iff

- Tr(G) = Tr(S),
- $\forall \sigma \in$ Tr(S), if Sgo=σ⇒Sgi, then Ac(Sgi) = Acc(So, σ),
- Any minimal cyclic trace in S is a minimal cyclic trace in G, and
- Any minimal cyclic trace in G is a minimal cyclic trace in S.

Given two FLTSs S1 = <St1, L1, T1, S1o> and S2 = <St2, L2, T2, S2o>, the algorithm Merge consists, first, of transforming the FLTSs S1 and S2 into FAGs G1=<Sg1, L1, Ac1, Tg1, Sg1o> and G2= <Sg2, L2, Ac2, Tg2, Sg2o>, respectively, such that Sg1 $\cap$ Sg2 = $\emptyset$ and <G1, S1> $\in$ AGR and <G2, S2> $\in$ AGR. The FAGs G1 and G2 are then merged by an FAG merging algorithm into the FAG G3 = <Sg3, L1 L2, Ac3, Tg3, <Sg1o, Sg2o>>, which is transformed back to an FLTS S3 such that <G3, S3> $\in$ AGR.

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

A state Sgi in Sg3 may be either a tuple <Sg1i, Sg2j> consisting of state Sg1i from Sg1 and Sg2j from Sg2 (as for the initial state <Sg1o, Sg2o>), or a simple state Sg1i from Sg1, or a simple state Sg2j from Sg2. These states and the transitions which reach them are added by the FAG merging algorithm step by step into Sg3 and Tg3, respectively, except for the two initial states Sg1o and Sg2o, each of these is replaced by the initial state <Sg1o, Sg2o> of G3.

Initially, Sg3 contains only the initial state <Sg1o, Sg2o>. The definition of the transitions from state <Sg1i, Sg2j> in Sg3 depends on the transitions from Sg1i in Sg1 and from Sg2j in Sg2. For instance, for a given state <Sg1i, Sg2j>, if there is a transition Sg1i–a→Sg1k in Tg1 and a transition Sg2j–a→Sg2m in Tg2, then the state <Sg1k, Sg2m> is added into Sg3 and the two transitions are combined into one transition <Sg1i, Sg2j>–a→<Sg1k, Sg2m> in Tg3. This is the situation when G1 and G2 have a common trace from their initial state to Sg1k and Sg2m, respectively.

Another case of this construction, if for a given state <Sg1i, Sg2j>, there exists a transition Sg1i–a→Sg1k in Tg1, with Sg1k≠ Sg1o, but there is no transition labelled by a from Sg2j in Tg2, then the state Sg1k is added into Sg3 and the transition Sg1i-a→Sg1k in Tg1 yields the transition <Sg1i, Sg2j>-a→Sg1k in Tg3. Reciprocally, if there exists a transition Sg2j–a→Sg2m in Tg2, with Sg2m≠ Sg2o, but there is no transition labelled by a from Sg1i in Tg1, then the state Sg2m is added into Sg3 and the transition Sg2j–a → Sg2m in Tg2 yields the transition <Sg1i, Sg2j>–a→Sg2m in Tg3. In the case where Sg1k = Sg1o (respectively Sg2m = Sg2o), instead of the transition <Sg1i, Sg2j>-a → Sg1o (respectively <Sg1i, Sg2j>-a→Sg2m), the transition <Sg1i, Sg2j>-a →<Sg1o, Sg2o> is added into Tg3.

The transitions from a simple state in Sg3, like state Sg1k or Sg2m above, for instance, remain the same as defined in G1 and G2, respectively. The states reached by these transitions are added into Sg3, except for the two initial states Sg1o and Sg2o, each of these is replaced by the initial state <Sg1o, Sg2o> of G3.

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

- if Sgi = <Sg1i, Sg2j>, then Ac3(Sgi) = {X1 $\cup$ X2 | X1 $\in$ Ac1(Sg1i) and X2 $\in$ Ac2(Sg2j)},
- if Sgi = Sg1i, with Sg1i $\in$ Sg1, then Ac3(Sgi) = Ac1(Sg1i),
- if Sgi = Sg2j, with Sg2j $\in$ Sg2, then Ac3(Sgi) = Ac2(Sg2j).

Given the FLTSs S1, S2, the following propositions have been proved in [Hami 95] concerning the FLTS S3 constructed by the algorithm Merge:

Proposition 1

S3 extends S1 and S3 extends S2.

Merge satisfies our first requirement as stated above in Proposition 1. However, the second requirement about the recursive exhibition of behaviors of S1 and behaviors of S2, in the case of minimal cyclic traces of S1 or S2, is not always satisfied. This requirement is satisfied, if and only if all the minimal cyclic traces in S1 and all the minimal cyclic traces in S2 remain minimal cyclic traces in S3. Unfortunately, there are some situations where a minimal cyclic trace in S1 (respectively S2) does not remain a minimal cyclic trace in S3. This is the case, when a given trace s is a minimal cyclic trace in S1 (respectively S2), but s is a noncyclic trace in S2 (respectively S1). After executing such a minimal cyclic trace, S3 reaches a state, which is different from its initial state. Therefore, after performing such a minimal cyclic trace, S3 does not transform into S3, and S3 may not exhibit again the behaviors of S1 and the behaviors of S2. Figure 2 illustrates such kind of situations. After performing a, which is a minimal cyclic trace in S1, S3 does not transform into S3, because the trace a belongs to S2 and it is not a cyclic trace in S2. S3 does not offer the behavior a.b of S2, after the minimal cyclic trace a. Note that, the minimal cyclic trace a.b in S2 remains a minimal cyclic trace in S3. In Proposition 2, we determined a necessary and sufficient condition, for which a minimal cyclic trace in S1 (respectively S2) remains a minimal cyclic trace in S3.

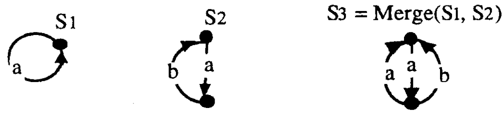


Fig 2. Counterexample for the minimal cyclic traces

Proposition 2

- A minimal cyclic trace s in S1, is a minimal cyclic in S3, iff (s { Tr(S2) or s is cyclic in S2).
- Reciprocally, for a minimal cyclic trace s in S2.

Any trace of S3 is either a trace of S1, or a trace of S2, or results from the concatenation of traces of S1 and S2. The following proposition shows how a trace s.a of S3 may be decomposed into its subtraces in S1 and S2, when s is a trace of S1 (respectively S2).

Proposition 3

$\forall$ a $\in$ L1 $\cup$ L2, if s $\in$ Tr(S1) and s.a $\in$ Tr(S3),
then σ.a $\in$ Tr(S1), or σ.a $\in$ Tr(S2), or
($\exists$ σ1, σ2 such that $\sigma = \sigma1.\sigma2$, S1=σ1$\Rightarrow$S1, S1=σ2$\Rightarrow$ S1'$\neq$a$\Rightarrow$, S2=s2$\Rightarrow$S2'=a$\Rightarrow$).
Reciprocally, for s Œ Tr(S2) and s.a Œ Tr(S3).

4 Merging Structured Specifications

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

Given a distributed system specification Sold, which consists of a parallel composition of subsystem specifications and so on until the basic components, and a specification Sadded, we want to construct a specification Snew, such that Snew extends Sold, and Snew extends Sadded.

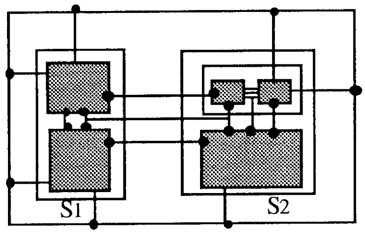


Fig 3. Structure of a Distributed System Specification

The specification Snew should preserve the internal structure of Sold. As for the merging of monolithic specifications, in the case of minimal cyclic traces of Sold or Sadded, Snew transforms into Snew, and may exhibit, in a recursive manner, behaviors of Sold and Sadded.

4.1 Identical Structure for Sold and Sadded

We assume that the specifications Sold and Sadded are both structured according to the form (S1 |AS2)\B described above, and S1 and S2 are either basic components or again structured by parallel composition. Moreover, we assume that Sold and Sadded have an identical structure. In other words, the form of the expression Sold is identical to the form of the expression Sadded. To every subsystem specification in Sold corresponds a subsystem specification in Sadded and vice versa. To every basic component Ciold in Sold, corresponds a basic component Ciadded in Sadded and vice versa.

The following algorithm for merging structured specifications, called Structured_Merge, is recursive over the structure of Sold and Sadded. It is based on the algorithm Merge, for merging monolithic specifications, described in Section 3.

Merging Algorithm for Structured Specifications

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

Snew, obtained by Structured_Merge(Sold, Sadded), has a structure identical to the structure of Sold and Sadded. As basic component, instead of Ciold or Ciadded, it has Cinew which results from the merging of Ciold and Ciadded by the algorithm Merge.

Unfortunately, Snew does not always extend Sold and Sadded. The extension of the basic components of Sold and Sadded is not sufficient to insure the extension of Sold and Sadded, respectively. Consider the counterexample in Figure 4, where Sold = (C1old $|_{\{g1\}}$ C2old)\{g1}, Sadded

= (C1added $|_{\{g2\}}$ C2added)\{g2}. The structure of the specification Snew is identical to the structure of Sold and Sadded, but Snew does neither extend Sold nor Sadded. Indeed, Sold never refuses the action b after trace a, whereas Snew may refuse action b after trace a. The same observation holds for action c after trace a. The trace a is common for C1old and C1added and it is followed by a hidden action g1 in C1old and g2 in C1added. The merging of C1old and C1added leads to a choice between the two hidden actions g1 and g2 after the trace a, in C1new. The components C1new and C2new may, internally, choose to synchronize on action g1 or g2, after a trace a, and offer only action b or only action c, respectively.

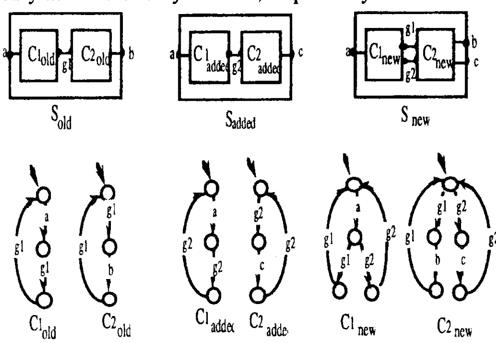


Fig 4. Counterexample

In Theorem 1, we have stated sufficient conditions for Sold and Sadded such that Snew extends Sold and Snew extends Sadded. We denote by HGold the set of hidden action names in Sold, and by HGadded the set of hidden action names in Sadded. The proof of Theorem 1 is given in the Appendix.

Theorem 1

Given Sold in the form of a hierarchical structure with the basic components C1old, C2old, ..., Cnold,
Sadded with an identical structure and the basic components C1added, C2added, ..., Cnadded, and
Snew = Structured_Merge(Sold, Sadded) as defined by the merging algorithm defined above,
we have that Snew ext Sold and Snew ext Sadded, if the following conditions are satisfied:

(a) $\forall i$, i = 1,..., n, act(Ciold) $\cap$ (HGadded = $\emptyset$, and act(Ciadded) $\cap$ (HGold = $\emptyset$,

(b) $\forall i, j, i \neq j$, (act(Ciold) $\cup$ act(Ciadded)) $\cap$ ((act(Cjold) $\cup$ act(Cjadded)) $\cap$ ((act(Sold) $\cup$ act(Sadded)) =$\emptyset$,

(c) For x = old, added,
Ci_x and Cj_x, with $i \neq j$, such that for some $g \in$ HGx, $g \in$ initials(Cix) and $g \in$ initials(Cj_x),

(d) For i = 1, ..., n,

1 - $\forall$ $\sigma \in$ Tr(Ciold), if s.g $\in$ Tr(Ciadded) with $g \in$ HGadded, then σ is cyclic in Ciold and Ciadded, and reciprocally,

2 - $\forall$ a $\in$ (act(Sold) (initials(Ciold)), if σ.a $\in$ Tr(Ciadded) for some σ, then σ is cyclic in Ciadded, and reciprocally.

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

Condition (b) says that there is no observable action of Sold and Sadded shared by two (or more) basic components of Sold (respectively Sadded). A basic component Ciold in Sold may have common observable actions only with the corresponding basic component Ciadded in Sadded, and reciprocally. Consider the example in Figure 5, where C1old and C2added have the action a in common, but they are not merged together. C1new = Merge(C1old, C1added), C2new = Merge(C2old, C2added), C1new extends C1old and C1added , and C2new extends C2old and C2added. The constructed specification Snew may refuse action b or action c, after trace a, whereas Sold and Sadded never refuses b or c after a, respectively. Snew does neither extend Sold nor Sadded. In order to prevent such situations, for each observable action, we may assign a "place" and the components with common observable actions have to be merged together, as stated by Condition (b).

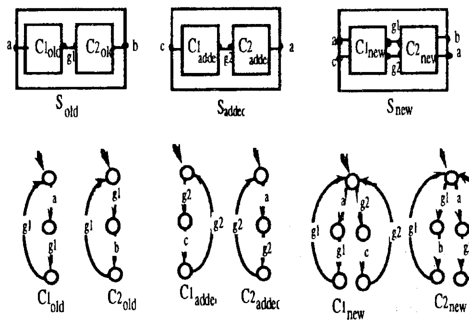


Fig 5. An illustration for Condition (b)

Condition (c) prevents Sold and Sadded from performing a hidden action from HGold or from HGadded, respectively, before interacting with the environment. Consider the example in Figure 6, in which C1new = Merge(C1old, C1added), C2new = Merge(C2old, C2added), C1new extends C1old and C1added, and C2new extends C2old and C2added. However Snew does not extend Sadded. After an internal move by executing the hidden action g1, it refuses the action a, whereas Sadded never refuses action a after an empty trace.

Condition (d-1) prevents from any new nondeterminism which may be introduced by the hidden actions in HGadded with respect to behavior in Sold and reciprocally, as shown in Figure 4. For a given pair of basic components Ciold and Ciadded, a common trace, which is not cyclic in both components, should not be followed by hidden actions from HGold or HGadded.

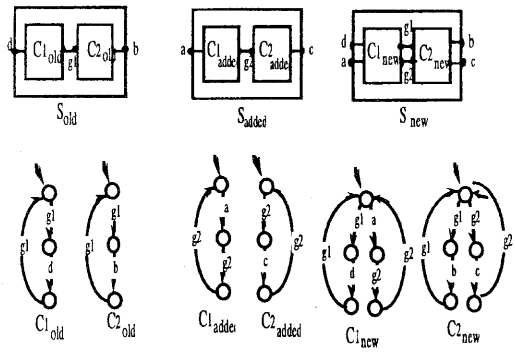


Figure 6. An illustration for Condition (c)

Condition (d-2) is introduced in order to prevent situations similar to the one shown in Figure 7. Assume that Sold = (C1old $|_{\{g1,\ g2\}}$ C2old)\{g1, g2} and Sadded = (C1added $|_{\emptyset}$ stop)\ $\emptyset$. The merging algorithm for structured specifications leads to Snew = (C1new $|_{\{g1,\ g2\}}$ C2new) \{g1, g2}, where C1new is shown in Figure 7 and C2new = C2old. We have C1new ext C1old and C1new ext C1added as well as C2new ext C2old and C2new ext C2added. However, Snew does not extend Sold. For instance, after the trace f.a.b.c, Snew may refuse to perform action d, whereas Sold never refuses to perform action d after trace f.a.b.c. This is due to the fact that we have two traces s1 = a.g1.b and s2 = a.g2.b in C1old, such that s1 ≠ s2, s1\HGold = s2\HGold, s1 is cyclic, s2 is not cyclic, s2.c is a trace in C1old, and c is a trace in C1added. It is possible to prevent such situations with a weaker condition than Condition (d-2) as explained in this example. However the verification of such conditions may be complex, whereas Condition (d-2) can be checked very easily.

Theorem 2 states that under certain conditions on the basic components of Sold and Sadded, a minimal cyclic trace σ in Sold (respectively Sadded) remains cyclic in Snew. Therefore, after performing σ, Snew reaches its initial state, and may exhibit again behaviors of Sold and behaviors of Sadded, without any new failure for these behaviors, since Snew extends Sold and Sadded.

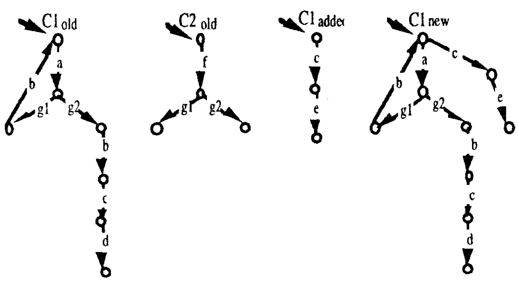


Fig 7. Illustration for Condition (d-2).

Theorem 2

Given specifications Sold, Sadded, and Snew as in Theorem 1, and assume that the conditions of Theorem 1 are satisfied, we have

- For any minimal cyclic trace σ in Sold, if for i = 1,..., n, σi is a minimal cyclic trace in Ci_{old} and (σi Tr(Ci_{added}) or σi is a cyclic trace in Ci_{added})), where σi represents the sequence of actions performed by Ciold, when Sold performs the trace σ, then σ is a cyclic trace in Snew.
- Reciprocally, for any minimal cyclic trace σ in Sadded.

5 Application

In the following, we illustrate our approach by an application. We start with a basic Automatic Teller Machine (ATM) which provides only the withdrawal function as described by the LTS in Figure 8. After inserting his card a customer is prompted for the Personal Identification Number (PIN) which may be valid or invalid. In case of invalid PIN, the card is rejected and the customer can retry again. If the PIN is valid, the customer can ask for a certain amount. The transaction is refused if the amount is higher than the balance. The customer can try with another amount or end the process and get back his card. In the other case, the money is delivered and the card is rejected. This function is implemented by three components as shown in Figure 9. The composition of these components, using the LOTOS parallel and hiding operators, yields the LTS in Figure 8.

We want to enrich the basic ATM with a new function, money deposit described by the LTS in Figure 10. This function allows the customer to deposit money into his account.

Similarly to the decomposition of the withdrawal function, the deposit function is implemented by three components as shown in Figure 11. The behavior of each of these three components is described by an LTS.

In order to obtain a new ATM providing both of the above functions without any interference among them and preserving the structure of the basic ATM, we apply our algorithm for merging structured specifications. This

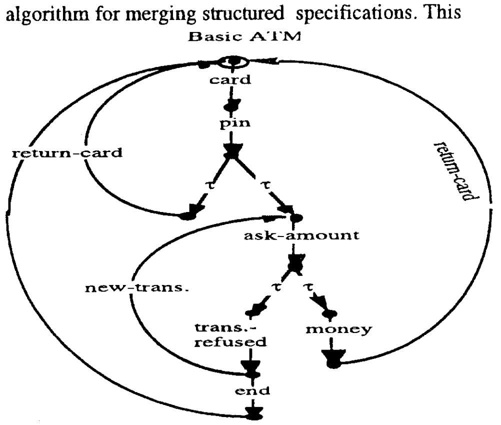
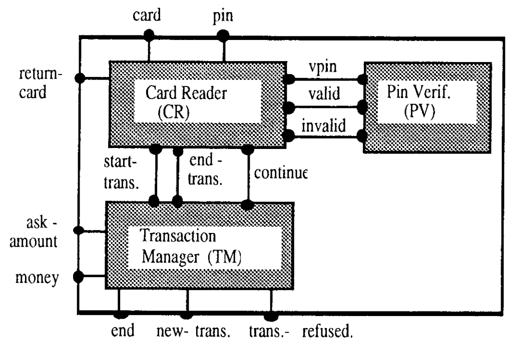
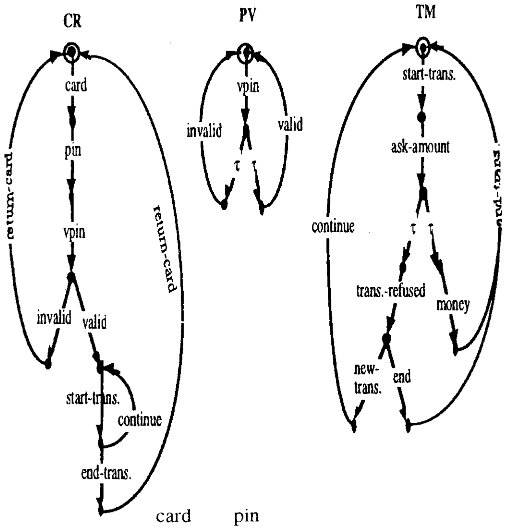


Figure 8. Basic Automatic Teller Machine.



algorithm will couple and merge the card readers together, the PIN verifiers together and the transaction managers together. The structure of the enriched ATM as well the behavior of the new components are shown in Figure 12. The behavior of the new ATM is described by the LTS in Figure 13. The sufficient conditions of Theorem 1 and Theorem 2 are satisfied. The behavior of the new ATM is an extension of the basic ATM and *deposit* function, and the new ATM is able to provide, alternatively, the *deposit* and the *withdrawal* function.



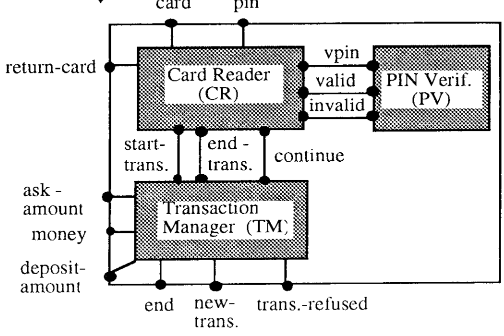


Fig 9. Structure of the Basic Automatic Teller Machine.

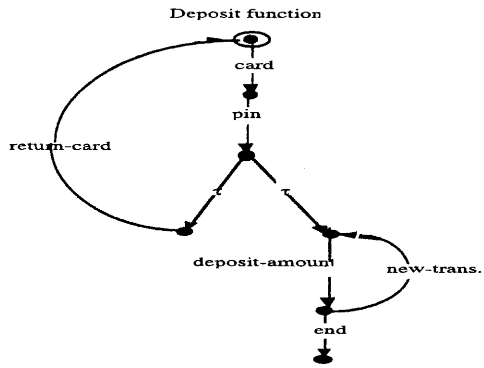


Fig 10. Deposit function.

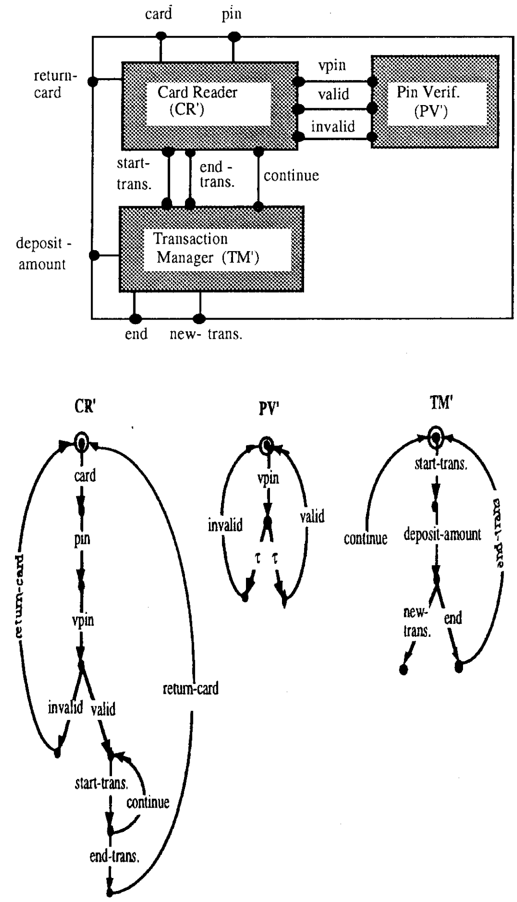


Fig11. Decomposition of deposit function.

References

[Brin 86] E. Brinksma, G. Scollo and S. Steenbergen, LOTOS specifications, their implementations and their tests, Protocol Specification, testing and verification, VI.

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