Electronic Notes in Theoretical Computer Science 82 No. 5 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 12 pages

Composition of Structured Process Specifications

Samira Sadaoui [1]

Department of Computer Science
University of Regina
3737 Wascana Parkway
Regina, SK S4S 0A2, Canada

Abstract

This paper provides the definition of an operator that composes two structured process specifications while preserving the original structure in the new specification. On each structuring level, this operator assembles two by two the components of the original processes, and so on until the lower level is reached where the basic components are integrated. The composition is driven by the external gates that are shared between the participating components, and components are assembled if they have the same internal structure. We associate with our operator a set of semantics conditions that ensures the correctness of the composition. The composition operator is progressively introduced with several examples.

1 Introduction

The real benefit of composable software systems lies in their increased flexibility: a system built from components should be easy to recompose to address new requirements [8]. Today, several composition techniques are needed to support different development approaches, including:

- bottom-up development where components from different sources are integrated into the system
- viewpoint oriented development where multiple partial components are composed to produce the final product. These components focus on different aspects of the design [9]
- incremental development where a component is enriched (or composed) with new properties compatible with the existing ones. For example, in an architectural context, the successive evolution of components is useful to conceive first the high level architecture, and then refine the components

[1] Email: sadaouis@cs.uregina.ca

In all cases, the resulting system should preserve, without introducing any errors, the behaviors of the composed components. For instance, the architectural refinement must not disturb neither the global functionality nor the added functionality [3]. Today in software components, the assembled components plug, but they might not play [7].

We examine the composition techniques in the area of process algebra, and in particular with the specification language LOTOS [1]. The operators in LOTOS allow a certain form of composition; for example the parallel operators are used to compose constraints in the constraint-oriented style [11]. However, only the full synchronization operator composes with the trace preorder which is a very weak notion of refinement [10]. Therefore, it is necessary to provide new composition operators that, in one hand, compose components in different ways that lead to different specifications, and in the other hand, preserve after composition important properties such as the deadlock ones. In literature, most of the composition operators involve non structured specifications. These operators are based on the well-known implementation relations of LOTOS, viz *reduction* of the non-determinism, and compatible *extension* of functionalities [2]. Hence, different composition types can be defined, e.g. the composition by reduction allows the partial reuse of components, and the composition by extension, the classical composition in component technology, reuses components without modification ("as-is reuse"). In this paper, we define an operator that composes by extension two structured specifications while preserving the initial structure in the new specification. On each structuring level, the composition is driven by the external gates that can be shared between the components of the original processes, and components are assembled if they have the same internal structure. We associate with our operator a set of semantics conditions that ensures the correctness of the composition.

2 Background

LOTOS is an international formal specification technique for specifying concurrent and distributed systems. It combines a process calculus with an abstract data type language [1]. It models parallel execution and interprocess communication, and supports a practical theory of correctness and refinement. Refinement transforms a specification into a more detailed or structured specification. Processes can be represented in three different ways: by algebraic expressions, labelled transition systems, or by trace and refusal sets. The operational semantics of LOTOS maps each process into a transition system. The syntax and operational semantics of the LOTOS operators involved in our composition operator are shown in table 1; G is the set of all observable actions and **i** is the invisible or internal action; Act = $G \cup \{\mathbf{i}\}$, $G^+ = G \cup \{\delta\}$ and Act^+ = Act $\cup \{\delta\}$; $g \in G$, $g^+ \in G^+$, $\mu \in Act$, $\mu^+ \in Act^+$ and $G \subseteq G$; E is the set of all processes, B, B_1 and B_2 are expression behaviors or processes; $\rightarrow$ denotes a transition relation.

Operators	Syntax	Semantics
Inaction	**stop**	
Successful termination	**exit**	$\text{exit} \xrightarrow{\delta} \text{stop}$
Action prefix	$\mu;B$	$\dfrac{}{\mu;B \xrightarrow{\mu} B}$
Choice	$B_1 [\,] B_2$	$\dfrac{B_1 \xrightarrow{\mu^+} B_1'}{B_1 [\,] B_2 \xrightarrow{\mu^+} B_1'} \quad \dfrac{B_2 \xrightarrow{\mu^+} B_2'}{B_1 [\,] B_2 \xrightarrow{\mu^+} B_2'}$
Hiding	**hide** G **in** B	$\dfrac{B \xrightarrow{g} B',\ g \in G}{\textbf{hide } G \textbf{ in } B \xrightarrow{i} \textbf{hide } G \textbf{ in } B'}$ $\dfrac{B \xrightarrow{\mu^+} B',\ g \notin G}{\textbf{hide } G \textbf{ in } B \xrightarrow{\mu^+} \textbf{hide } G \textbf{ in } B'}$
Parallel composition	$B_1 \lvert[\mathbf{G}]\rvert B_2$	$\dfrac{B_1 \xrightarrow{\mu} B_1',\ \mu \notin G}{B_1 \lvert[G]\rvert B_2 \xrightarrow{\mu} B_1' \lvert[G]\rvert B_2} \quad \dfrac{B_2 \xrightarrow{\mu} B_2',\ \mu \notin G}{B_1 \lvert[G]\rvert B_2 \xrightarrow{\mu} B_1 \lvert[G]\rvert B_2'}$ $\dfrac{B_1 \xrightarrow{g^+} B_1',\ B_2 \xrightarrow{g^+} B_2',\ g^+ \in G}{B_1 \lvert[G]\rvert B_2 \xrightarrow{g^+} B_1' \lvert[G]\rvert B_2'}$

Table 1
Operational semantics for LOTOS

Definition 2.1 P $\in E$, L is the alphabet of P, a $\in$ Act an action, s $\in Act^*$ a sequence of actions, and $\varepsilon \in$ Act the empty trace. A trace is a sequence of observable actions. The transition $\Rightarrow$ is used to ignore the internal actions.

- P is stable iff P $\overset{i}{\not\rightarrow}$ (P can not move observably to another state)
- P is deadlock iff $\forall$ a $\in$ Act, $P \overset{a}{\not\rightarrow}$ (no progress is possible)
- Der(P) = $\{ P' \mid \exists s \in Act^*, P \overset{s}{\Rightarrow} P' \}$ (the set of all states that can be reached from P)
- Gates(P) = $\{a \in G \mid \exists P' \in Der(P), P' \overset{a}{\rightarrow}\}$ (set of all external actions of P)
- Hgates(P) = L(P) \ Gates(P) (set of all hidden actions of P)
- $Init(P) = \{a \in G \mid P \overset{a}{\Rightarrow}\}$ (set of all initial actions of P)
- $Tr(P) = \{s \in G^* | P \overset{s}{\Rightarrow}\}$ (set of all traces that can be reached from P)
- $After(P, s) = \{P' | P \overset{s}{\Rightarrow} P'\}$ (set of all states that can be reached from P via the trace s)
- $Ref(P, s) = \{X \mid \exists P' \in$ After(P, s) : $\forall a \in X, P' \overset{a}{\not\Rightarrow}\}$ (refusal set of P after the trace s)
  Ref(P, s) is a set of sets such that Ref(P, s) is included in the set of partitions of L. A set X $\subseteq L$ belongs to Ref(P, s) if and only if P can execute the trace s and then refuses all the actions of X
- $Dash(P, a) = \{P' | P \overset{\varepsilon}{\Rightarrow} \overset{a}{\rightarrow} P'\}$ (set of all states that can be reached from P after the action a eventually preceded by internal actions)

The extension preorder, denoted **ext** [2], allows for the introduction of new traces in an implementation P while preserving the deadlocks properties of the original specification Q. Informally, P extends Q, if P allows any traces that Q allows, and P does not deadlock in a situation where Q would not deadlock (P can only refuse what Q can refuse).

Definition 2.2 P and Q are two processes. P is an extension of Q, P **ext** Q iff:

- $Tr(P) \supseteq Tr(Q)$ (* P performs all the traces of Q *)
- $\forall s \in Tr(Q),\ Ref(P,s) \subseteq Ref(Q,s)$ (*Preserving the deadlock properties in P*)

Example 2.3 Let us consider the following process `P:= a; i; b; stop [] i; a; c; stop.`

- L(P) = {a, b, c}
- P is not stable; P can offer **i** as the first action
- P is deadlock; after the inaction stop, P can not progress
- Gates(P) = {a, b, c}; Hgates (P) = ∅; Init(P) = {a}
- Dash(P, a) = {(i, b, stop), (c, stop)}

Let us consider another process `Q:= a; b; stop`. P **ext** Q since:

- Tr(P) = {ε, a, ab, ac} ⊇ Tr(Q) = {ε, a, ab}
- Ref(P, ε) = {∅, {b, c} } ⊆ Ref(Q, ε) = {∅, {b, c}}
- Ref(P, a) = {∅, {a, c}, {a, b}} ⊆ Ref(Q, a) = {∅, {a, c}, {a, b}}
- Ref(P, ab) = {∅, {a, b, c}} ⊆ Ref(Q, ab) = {∅, {a, b, c}}

3 Composition of Basic Components

3.1 Composition by Extension

A process can be basic (an alternative ordering of actions), or structured (parallel composition of components). The composition by extension, called $Comp_{ext}$, is a function that takes two basic processes, P and Q, and produces a common extension S of P and Q. In definition 3.1, the traces of S include the traces of both P and Q, and after a trace that P (or Q) may do, S may refuse what P (or Q) can refuse. To produce the biggest composition S, the set inclusions should be replaced by the set equalities [10].

Definition 3.1 S:= $Comp_{ext}$(P, Q) is a composition by extension iff:

- Tr(S) ⊇ Tr(P) ∪ Tr(Q)
- ∀ s ∈ Tr(P) ∩ Tr(Q), Ref(S, s) ⊆ Ref(P,s) ∩ Ref(Q,s)
- ∀ s ∈ Tr(P) \ Tr(Q), Ref(S, s) ⊆ Ref(P,s)

- $\forall$ s $\in$ Tr(Q) $\setminus$ Tr(P), Ref(S, s) $\subseteq$ Ref(Q,s).

A successful composition depends on the consistency of P and Q. Two processes are consistent with each other whether it is possible to find at least one implementation S that refines both specifications [10]. In [5], it has been proved that from two transition systems, we can always build a transition system which is an extension of the two others. Since LOTOS uses the same semantics model, thus the result is applicable to the processes i.e., $\forall$ P $\in E$, $\forall$ Q $\in E$, $\exists$ S $\in E$, S **ext** P and S **ext** Q (the set of compositions is not empty).

Some composition-by-extension operators have been defined. In [3], the composition $\oplus$ supports the incremental development of systems in an architectural context. In [10], the author provides the operator $\bowtie$ which is an improvement of $\oplus$ to take into account the hiding operator and the action **i**. In [4], the operator *Merge*, extension of $\oplus$ to take into account the hiding operator, preserves the cyclic traces in the original processes. This operator applies to the transition systems but uses acceptance tree as an intermediate model.

3.2 The Union Operator

The union operator $\bowtie$ is defined with an operational semantics [10]. This operator merges those behaviors that the two processes have in common, and then provides a choice between the two behaviors when they start to differ. Its resolves the non-determinism and removes all the internal actions from the overlapping behavior of both operands. We associate with the operator $\bowtie$ the algorithm given below; Σ denotes the generalization of the choice operator, i.e. Σ(stop)= stop, Σ({P})= P, Σ({P,Q}) = P [ ] Q, and Σ({P} $\cup$ E)= P [ ] Σ(E).

Algorithm 1

$\bowtie$*(P, Q):=*
if Init(P) $\cap$ *Init(Q)* = {ai_1, ..., ai_n } *and n* $\geq$ 1 *then*
(ai_1; $\bowtie$(Σ(*Dash(P,* ai_1)), Σ(*Dash(Q,* ai_1))) **[ ]** ... **[ ]** ai_n; $\bowtie$(Σ(*Dash(P,* ai_n)), Σ(*Dash(Q,* ai_n))))
else **stop**
[ ]
if Init(P) $\setminus$ *Init(Q)* = {ap_1, ..., ap_m } *and m* $\geq$ 1 *then*
(ap_1 ; (Σ(*Dash(P,* ap_1))) **[ ]** ... **[ ]** ap_m; (Σ(*Dash(P,* ap_m))))
else **stop**
[ ]
if Init(Q) $\setminus$ *Init(P)* = {aq_1, ..., aq_r } *and r* $\geq$ 1 *then*
(aq_1; (Σ(*Dash(Q,* aq_1))) **[ ]** ... **[ ]** aq_r ; (Σ(*Dash(Q,* aq_r))))
else **stop**

Example 3.2 We consider here two vending machines VM1 and VM2 defined below:

- `VM1:= Insert_Coins; (Select_Coffee; stop [] Select_Juice; stop)`
- `VM2:= Insert_Coins; (Select_Tea; stop [] Select_Coffee; stop)`

The specification VM, composition by extension of VM1 and VM2, is obtained as fellows:

- `VM:= VM1 ⋈ VM2`

```
VM:= Insert_Coins; (Select_Coffee; stop [] Select_Juice; stop
     [] Select_Tea; stop)
```

We can easily check that VM **ext** VM1 and VM **ext** VM2 using the definition 3.1.

4 Composition of Structured Specifications

4.1 First Version

We present here an operator called *CompSt* that composes by extension two architectures P and Q, and preserves the original structure in the new specification. The basic components in P and Q are integrated with any composition-by-extension operator $Comp_{ext}$. Building structured specifications is important for distributed software engineering. We consider here the specifications with the following form S: **hide** HG **in** $(C_1 \mid [HG] \mid C_2)$ such that:

- if $HG = \emptyset$ then C_1 and C_2 are independent components
- if $HG \neq \emptyset$ then C_1 and C_2 are components that communicate on hidden gates

The two components C_1 and C_2 are both structured according to S, and so on. We also note that any specification can be re-structured with respect to S. We give here the first version of the operator *CompSt* which is a generalization of the composition algorithm given in [4] where the monolithic components (basic components but without the action **i**) are assembled with the operator *Merge*.

Algorithm 2

CompSt(P, Q):=
if P = **hide** HG_P **in** $(C_{1P} \mid [HG_P] \mid C_{2P})$
and Q = **hide** HG_Q **in** $(C_{1Q} \mid [HG_Q] \mid C_{2Q})$ *(*structured processes*)*
then **hide** HG_P, HG_Q **in**
$(CompSt(C_{iP}, C_{jQ}) \mid [HG_P, HG_Q] \mid CompSt(C_{i'P}, C_{j'Q}))$
such that $i, i', j, j' \in 1..2$ *and* $i \neq i'$ *and* $j \neq j'$
*(*composition of the structuring levels*)*
else $Comp_{ext}(P, Q)$ *(*composition of basic processes*)*

Example 4.1 Our aim is to compose two structured vending machines VM1 and

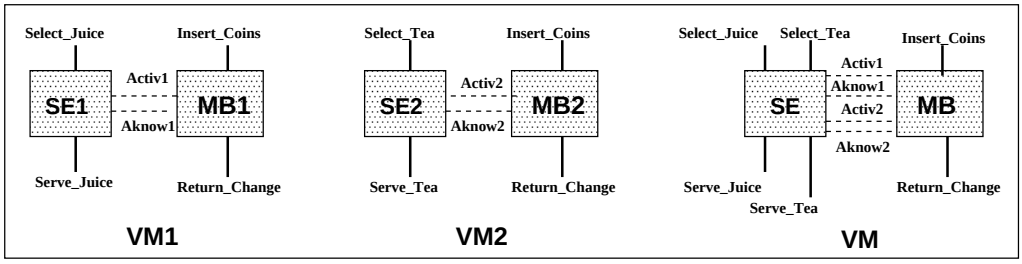

Fig. 1. Composition of 2 vending machines

VM2 illustrated in figure 1. A vending machine is the parallel composition of two processes: a money box and a selector. The hidden gates are represented in dotted lines. The basic components in VM1 and VM2 are defined below:

- SE1 := Activ1; Select_Juice; Serve_Juice; Aknow1; SE1
- MB1 := Insert_Coins; Activ1; Aknow1; Return_Change; MB1
- SE2 := Activ2; Select_Tea; Serve_Tea; Aknow2; SE2
- MB2 := Insert_Coins; Activ2; Aknow2; Return_Change; MB2

The structure of the composition VM, given in figure 1, is as fellows:

- VM:= *CompSt*(VM1, VM2)

  VM:= **hide** Activ1,Aknow1,Activ2,Aknow2 **in**

  (SE |[Activ1,Aknow1,Activ2,Aknow2]| MB)

In algorithm 2, the composition $Comp_{ext}$ is instantiated with the union operator $\bowtie$ as fellows:

- SE:= SE1 $\bowtie$ SE2

  SE:= (Activ1; Select_Juice; Serve_Juice; Aknow1;SE) [] (Activ2; Select_Tea; Serve_Tea; Aknow2; SE)
- MB:= MB1 $\bowtie$ MB2

  MB:= Insert_Coins; (Activ1; Aknow1; Return_Change; MB [] Activ2; Aknow2; Return_Change; MB)

4.2 Conditions of the Composition

We associate with the operator *CompSt* the following conditions: P and Q must have the same internal structure, and the extension relation must be preserved on each structuring level.

Same internal structure.

P and Q must have the same internal structure (for each component C_{iP} in P must correspond a component C_{jQ} in Q). P and Q should at least have the same number of structuring levels and the same number of basic components. The fol-

lowing algorithm *Unifstr* checks whether P and Q have the same structure.

Algorithm 3

Unifstr(P, Q) =
if P = **hide** HG_P **in** $C_{1P} \,|[HG_P]|\, C_{2P}$
and Q = **hide** HG_Q **in** $C_{1Q} \,|[HG_Q]|\, C_{2Q}$
then if Unifstr(C_{1P}, C_{1Q}) *and Unifstr*(C_{2P}, C_{2Q}) *then true*
else if Unifstr(C_{1P}, C_{2Q}) *and Unifstr*(C_{2P}, C_{1Q}) *then true*
else false
else if P and Q are basic components then true
else false.

We notice that if P and Q do not have the same structure, we can transform P into an equivalent process P', and Q into an equivalent process Q' such that P' and Q' have the same parallel structure.

Preserving the extension.

In general when composing two architectures P and Q, S is not always an extension of P and Q. The extension of the basic components is not enough to ensure the extension of the global specification. A set of sufficient conditions is defined in [4] to preserve the extension relation, including:

- the hidden gates in P must not conflict with the gates in Q, and vice versa. To satisfy this condition, we just have to rename the hidden gates because the renaming does not disturb the observable behavior of the process
- P and Q should be stable
- all the synchronization gates should be hidden
- the external gates in each process must not be shared by two or more of its components
- a common trace in P and Q that is not cyclic must not be followed by hidden actions in P and in Q.

If the conditions above are satisfied then *CompSt*(P, Q) **ext** P and *CompSt*(P, Q) **ext** Q. If these conditions are not satisfied, we can transform P and Q into non structured specifications, and then compose them using any operator $Comp_{ext}$. We have then to restructure the resulting composition.

Example 4.2 In the example 4.1, VM **ext** VM1 and VM **ext** VM2 since:

- SE **ext** SE1 and SE **ext** SE2 by construction with the operator $\bowtie$
- MB **ext** MB1 and MB **ext** MB2 by construction with the operator $\bowtie$
- VM **ext** VM1 and VM **ext** VM2 because the extension-preserving conditions defined in [4] are satisfied.

4.3 Detailed Version

The algorithm *CompSt* does not specify how to compose two by two the internal components of P and Q; for any component C_{iP} in P, how to find its corresponding C_{jQ} in Q ? On each structuring level, the component interface provides the connectivity ports:

- first we compose the components that share common actions i.e., $Gates(C_{iP}) \cap Gates(C_{jQ}) \neq \emptyset$.
- after that we compose randomly the other components.

In this new version, the composition is driven by the external gates that can be shared between the internal components of P and Q. On each structuring level, we also take into account that two components are composed if they have the same structure. We notice that even P and Q have the same internal structure, P and Q can not be composed because there exists two components in P and Q that have the same external gates but do not have the same structure.

Algorithm 4

*Init(k) (*initialize k to 0*)*
CompSt(P, Q):=
  if P = **hide** HG_P **in** $C_{1P} \,|[HG_P]|\, C_{2P}$
  and Q = **hide** HG_Q **in** $C_{1Q} \,|[HG_Q]|\, C_{2Q}$
  then
    if $\exists C_{iP} \in P, \exists C_{jQ} \in Q$ *such that* $Gates(C_{iP}) \cap Gates(C_{jQ}) \neq \emptyset$
    then
      if Unifstr(C_{iP}, C_{jQ}) *and Unifstr*$(C_{i'P}, C_{j'Q})$ *and* $i \neq i'$ *and* $j \neq j'$
      then
        { *Incr(k) (*increment k by one*)*
          hide $Hgates(P), Hgates(Q)$ **in**
          (C_{kS} *Gates(*C_{kS}*)* $\mid [Hgates(P), Hgates(Q)] \mid$ $C_{(k+1)S}$ *Gates(*$C_{(k+1)S}$*)*
          such that C_{kS} *:= CompSt(*C_{iP}*,* C_{jQ}*) and* $C_{(k+1)S}$ *:= CompSt(*$C_{i'P}$*,* $C_{j'Q}$*)*
        }
      else stop
    else
      if $\exists C_{iP} \in P, \exists C_{jQ} \in Q$ *such that Unifstr(*C_{iP}*,* C_{jQ}*) and Unifstr(*$C_{i'P}$*,* $C_{j'Q}$*)*
      and $i \neq i'$ *and* $j \neq j'$
      then
        { *Incr(k) (*increment k by one*)*
          hide $Hgates(P), Hgates(Q)$ **in**
          (C_{kS} *Gates(*C_{kS}*)* $\mid [Hgates(P), Hgates(Q)] \mid$ $C_{(k+1)S}$ *Gates(*$C_{(k+1)S}$*)*
          such that C_{kS} *:= CompSt(*C_{iP}*,* C_{jQ}*) and* $C_{(k+1)S}$ *:= CompSt(*$C_{i'P}$*,* $C_{j'Q}$*)*
        }
      else stop
  else $Comp_{ext}$*(P, Q)*

For the automation of the composition operator (including the two algorithms 1 and 4, and the conditions defined in subsection 4.2), the different semantics properties defined in Section 2 can be algebraically computed according to LOTOS operators. For instance, the trace and refusal sets have been computed and proved correct in [6].

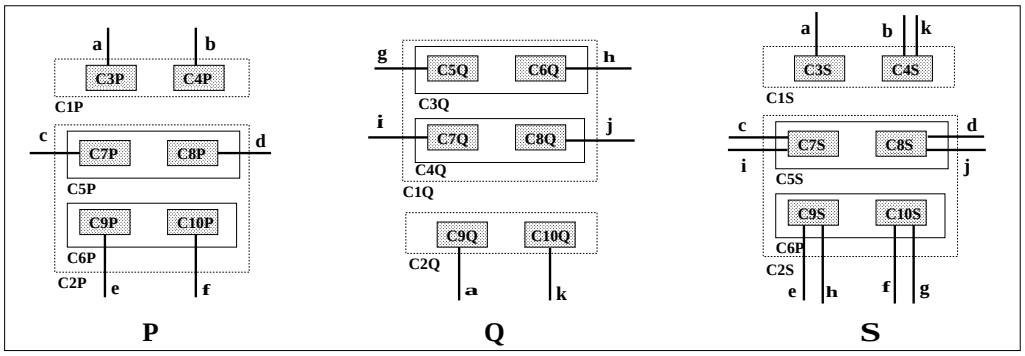

Fig. 2. Processes P, Q, and their composition S

Example 4.3 Table 2 explains how two structured processes P and Q are composed using the algorithm 4. The structures of P and Q are illustrated in figure 2 by abstracting the hidden gates.

In figure 3, we summarize the composition of structured process specifications.

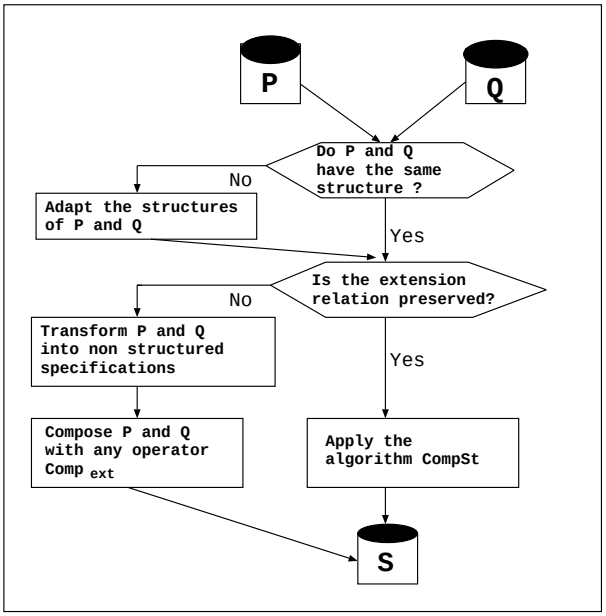

Fig. 3. Composition of structured specifications

Composition of components	**Conditions of composition**
S := *CompSt*(P, Q)	Unifstr(P, Q) = true
First level	
C_{1S} := *CompSt*(C_{1P}, C_{2Q})	Gates(C_{1P}) ∩ Gates(C_{2Q}) = {a} ≠ ∅
	Unifstr(C_{1P}, C_{2Q}) = true
C_{2S} := *CompSt*(C_{2P}, C_{1Q})	Unifstr(C_{2P}, C_{1Q}) = true
Second level	
C_{3S} := C_{3P} ⋈ C_{9Q}	Gates(C_{3P}) ∩ Gates(C_{9Q}) = {a} ≠ ∅
	C_{3P} and C_{9Q} are basic components
C_{4S} := C_{4P} ⋈ C_{10Q}	C_{4P} and C_{10Q} are basic components
random composition	
C_{5S} := *CompSt*(C_{5P}, C_{4Q})	Unifstr(C_{5P}, C_{4Q}) = true
C_{6S} := *CompSt*(C_{6P}, C_{3Q})	Unifstr(C_{6P}, C_{3Q}) = true
Third level	
random composition	
C_{7S} := C_{7P} ⋈ C_{8Q}	
C_{8S} := C_{8P} ⋈ C_{7Q}	basic components
C_{9S} := C_{9P} ⋈ C_{6Q}	
C_{10S} := C_{10P} ⋈ C_{5Q}	

Table 2
Composition of structuring levels

5 Conclusion and future work

The specification language LOTOS with its structuring capabilities and strong theory is suitable for the composition of specifications. The goal of the composition is to find a specification that is a common implementation of the composed processes. For each type of composition (by extension or by reduction), we can propose different operators producing different new specifications. In this paper, we have focused our composition on process algebra and without considering the data part. Our future work consists of including the data types first in the the union operator ⋈, and second in the operator *CompSt*.

To build a new specification from existing components, we can combine several composition operators. The different combinations of the operators lead to different specifications. The major difficulty is to identify the best combination that produces the specification with the desirable behaviors.

References

[1] Bolognesi, T., and E. Brinksma, "Introduction to the ISO specification language LOTOS", In P.H.J. van Eijkand, C.A. Vissers and M. Diaz, eds., The Formal Description Technique LOTOS (North-Holland, Amsterdam) 303-326, 1989.

[2] Brinksma, E., G. Scollo, and C. Steenbergen, *LOTOS Specifications, Their Implementations and Their Tests*, Protocol Specification, Testing and Verification, VI, IFIP, 1987.

[3] Ichikawa, H., K. Yamanaka, and J. Kato, *Incremental Specification in LOTOS*, In L. Logrippo, R.L. Probert, and H. Ural, editors, Protocol Specification, Testing and Verification X, 183–196, Ottawa, Canada, 1990.

[4] Khendek, F., and G. von Bochmann, *Incremental Construction Approach for Distributed System Specification*, Proc. Int. Symp. on Formal Description Techniques, Boston, 1993.

[5] Khendek, F., and G. von Bochmann, *Merging behavior specification*, Journal of Formal Methods in System Design, **6(3)**, (1995), 259–294.

[6] Leduc, G., "On the Role of Implementation Relations in the Design of Distributed Systems using LOTOS", PhD thesis, University of Liège, Belgium, June, 1991.

[7] Michiels, B., and B. Wydaeghe, *Component composition*, ICSE'00, ACM Computing Surveys, 771–771, 2000.

[8] Nierstrasz, O., and T.D. Meijler, *Research direction in software composition*, ACM Computing Surveys, **27(2)**, (1995), 263–264.

[9] Steen, M. W. A., "Consistency and Composition of Process Specifications", PhD thesis, University of Kent at Canterbury, May 1998.

[10] Steen, M. W. A., H. Bowman, and J. Derrick, *Composition of LOTOS specifications*, In P. Dembinski and M. Sredniawa, editors, Protocol Specification, Testing and Verification, Chapman & Hall, 87–102, 1995.

[11] Turner, K. J., *Incremental requirements specification and constraint-oriented style in LOTOS*, Technical Report, Department of Computing Science, University Stirling, UK, Avril 1996.