On Compatibility and Behavioural Substitutability of Component Protocols

Nabil Hameurlain
LIUPPA Laboratory, University of Pau, France
Nabil.hameurlain@univ-pau.fr

Abstract

Component Based Development (CBD) aims to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. Components specification is thus needed to ensure the safety of composing systems from components. This paper focus on component protocols specification and provides a framework for modelling protocols together with their composition. We start by investigating compatibility of component protocols based on service observation. Two compatibility relations together with their characterisation by the preservation to their degree of change property are proposed. Safety and liveness properties such as deadlock-freeness and proper termination of protocols are preserved up to different extents. Then, we propose some behavioural subtyping relations for component protocols related to the principle of substitutability. Finally, we address the soundness of our subtyping relations by showing the existing link between compatibility and substitutability concepts, namely their combination, which have found necessary when dealing with incremental design of components.

1. Introduction

Component Based Development (CBD) [17] aims to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. It is one of the most important technical initiatives in software engineering. In CBD, software systems are built by assembling components which are already developed and prepared for integration. The specification of components provides a definition of its interface that must be precise and complete for users, and an abstract definition of its internal structure for developers. Specification and verification are needed to ensure the safety of composing systems from components. Verification and CBD are synergistic: CBD introduces compositional structures and composition rules to the system being built, whereas specification and verification enable effective development of reliable component-based software systems.

Although there is no unique definition of the component concept, several characteristics are fundamental to it such as explicit interfaces including sufficient information. Beyond classical signature based interfaces (such as CORBA [4], EJB [9]), interfaces should make explicit all the means for using components, such as communications and control between those components. In the last years [5, 14, 20], *protocols* in component interfaces have been proposed to specify the behavioural property of components such as “call sequences accepted” (as specified in a provided interface) and “call sequences required” (as specified in the required interface) by the component. Different approaches for protocols specification have been proposed, ranging from state machine approaches [5, 13], Petri nets [18], predicates [21], to process calculi [1, 2, 3, 11]. In most of those approaches, the semantics used are based upon finite-state automata, which support automatic verification techniques.

This paper integrates into interfaces, by means of protocols, the sequencing constraints that any component should obey when calling the services of another component. Protocols are formalized within component-nets [18], a formalism combining Petri nets and component-based approach. The protocols allow to detect specific component properties when checking two components for compatibility and substitutability. Two properties are of great importance (1) safety property: deadlock-freeness of the protocol, and (2) liveness property: the successful (proper) termination of the protocol.

The contribution of this paper are (a) to provide compatibility relations for component protocols based on service observation together with their characterisation to their degree of change by property preservation, (b) to propose some behavioural

subtyping relations for protocols related to the principle of substitutability [10] and (c) to study the existing link between compatibility and substitutability, namely the soundness of the proposed subtyping relations that is the preservation of compatibility relations by substitutability.

The paper is organized as follows. Section 2 presents the basic definitions of the notions based on action observation. Section 3 describes our component protocol specification model together with component-nets formalism. The semantics, the composition and the properties of protocols are presented. Section 4 provides two compatibility relations and their characterisations by property preservation. Three subtyping relations are proposed and the preservation of components compatibility relations to their degree of change is studied. The compatibility and the subtyping relations are based upon failure and bisimulation semantics, which are considered in the study of concurrent systems and proved to be quite easy to use in practice. Section 5 concludes and presents some related approaches. Due to lack of space, the proofs may be found in the internal report [7].

2. Basic definitions

We start by the definitions of the relevant concepts underlying our component protocol model, compatibility and behavioural subtyping relations. Labelled Petri Nets (PN for short) [12] are used to describe the behaviour of protocols, the services that are invoked on and by component and their order of execution, together with the behavioural type of a component protocol. The semantics of compatibility and behavioural subtyping relations and their soundness are based on failure (of the process algebra CSP), and bisimulation semantics. Let A be a set of methods, that is the alphabet of observable actions, and $\{\lambda,\nu\}$ denotes two special unobservable actions. The symbol λ plays the usual role of an internal action, whose execution is under the control of the net alone; the symbol ν stands for action which is unobservable to a particular client of a server net, but is not under the control of the server alone; it may have to be executed together with another client of the net.

Labelled Petri nets. A marked Petri net $N = (P, T, W, M_N)$ consists of a finite set P of places, a finite set T of transitions where $P \cap T = \varnothing$, a weighting function $W : P \times T \cup T \times P \to \mathbb{N}$, and $M_N : P \longrightarrow \mathbb{N}$ is an initial marking. A transition $t \in T$ is enabled under a marking M, noted $M(t>$, if $W(p, t) \leq M(p)$, for each place p. In this case t may occur, and its occurrence yields the follower marking M', where $M'(p) = M(p) - W(p, t) + W(t, p)$, noted $M(t > M'$. The enabling and the occurrence of a sequence of transitions $\sigma \in T^*$ are defined inductively. The preset of a node $x \in P \cup T$ is defined as ${}^\bullet x = \{y \in P \cup T, W(y, x) \neq 0\}$, and the postset of $x \in P \cup T$ is defined as $x^\bullet = \{y \in P \cup T, W(x, y) \neq 0\}$. We denote as $LN = (P, T, W, M_N, l)$ the (marked, labelled) Petri net (see [12] for further information) in which the events represent actions, which can be observable. It consists of a marked Petri net $N = (P, T, W, M_N)$ with a labelling function $l: T \longrightarrow A \cup \{\lambda,\nu\}$. Let ε be the empty sequence of transitions, l is extended to an homomorphism $l^*: T^* \longrightarrow A^* \cup \{\lambda, \nu\}$ in the following way: $l(\varepsilon) = \lambda$ where ε is the empty string of T^*, and $l^*(\sigma.t) = l^*(\sigma)$ if $l(t) \in \{\lambda, \nu\}$, $l^*(\sigma.t) = l^*(\sigma).l(t)$ if $l(t) \notin \{\lambda, \nu\}$. In the following, we denote l^* by l, LN by (N, l), and if $LN = (P, T, W, M_N, l)$ is a Petri net and l' is another labelling function of N, (N, l') denotes the Petri net (P, T, W, M_N, l'), that is N provided with the labelling l'. A sequence of actions $w \in A^* \cup \{\lambda\}$ is enabled under the marking M and its occurrence yields a marking M', noted $M(w >> M'$, iff either $M = M'$ and $w = \lambda$ or there exists some sequence $\sigma \in T^*$ such that $l(\sigma) = w$ and $M(\sigma> M'$. The first condition accounts for the fact that λ is the label image of the empty sequence of transitions. A marking is stable if no unobservable action λ is enabled: M stable if <u>not</u> $(M(\lambda >>)$. For a marking M, $Reach (N,M) = \{M'; \exists \sigma \in T^*; M(\sigma > M'\}$ is the set of reachable markings of the net N from the marking M.

Definition 2.1 (Traces and language)
Let $N = (P, T, W, M_N, l)$ be a labelled net. Then $Tr (N) = \{\sigma \in T^*; M_N(\sigma >\}$is the traces of N, i.e. the set of enabled transition sequences of N. The label image of the traces of N is its language $L(N) = l(Tr(N)) = \{l(\sigma) \in A^*: \exists \sigma \in Tr(N)\}$.

Definition 2.2 (Failures)
Let $N = (P, T, W, M_N, l)$ be a labelled net. Then the *failures* of the net N on T' is $F(N, T') = \{(\sigma, S); \sigma \in T^*$, $S \subseteq T'$, and there exists some marking M such that $M_N(\sigma > M$, and $\forall t \in S$, <u>not</u> $(M(t >)\}$.
The label image of the failures of N is $F(N) = l(F(N, T)) = \{(l(\sigma), X); X \subseteq A$, and $\forall a \in X$, <u>not</u> $(M(a>>)$, for M stable such that $M_N(\sigma > M\}$.

Definition 2.3 (Bisimulation)
Let $N = (P, T, W, M_N, l)$ and $N' = (P', T', W', M_{N'}, l')$ be two labeled nets. We say that N and N' are bisimilar,

noted $N \approx_{BiSim} N'$, iff there exists a bisimulation relation $R \subseteq$ Reach (N, M_N) x Reach $(N', M_{N'})$ such that :
1. $(M_N, M_{N'}) \in R$,
2. $\forall (M_1, M'_1) \in R, \forall a \in A \cup \{\lambda\}, \forall M_2, M_1(a>> M_2 \Rightarrow \exists M'_2, M'_1(a>> M'_2$ and $(M_2, M'_2) \in R$,
3. and vice versa: $\forall a \in A \cup \{\lambda\}, \forall M'_2, M'_1(a>> M'_2 \Rightarrow \exists M_2, M_1(a>> M_2$ and $(M_2, M'_2) \in R$.

3. Component protocol modelling

In this paper we adopt the approach specifying protocols by Petri nets, allowing non-deterministic and concurrency between protocols together with their composition. First, we present the formalism underlying the definition of protocols, called component-nets, and then we give the definition of our component protocol model together with its execution semantics.

3.1. Components nets (C-nets)

3.1.1. Definition and semantics. Component-nets formalism [18] combines Petri nets with the component-based approach; petri nets will be used for concurrency, specification, verification, and refinement of protocols, whereas component-based approach will be used as a high level concept of abstraction which consider a protocol as a collection of sub-protocols, dealing with complex interactions between components. Semantically, a Component-net involves actions, which are observable or not observable together with two special places: the first one is the input place for instance creation of the component; and the second one is the output place for instance completion of the component. A C-net makes some services available to the nets and is capable of rendering these services. Each offered service is associated to one or several transitions, which may be requested by C-nets, and the service is available when one of these transitions, called *accept-transitions*, is enabled. On the other hand it can request services from other C-net transitions, called *request-transitions*, and needs these requests to be fulfilled. Thus, a C-net may be a server (and/ or client) if and only if it accepts (and/ or requests) at least one service.

Definition 3.1 (C-net)
Let $CN = (P \cup \{I, O\}, T, W, M_N, l_{Prov}, l_{Req})$ be a labelled Petri net. CN is a *Component-net* (C-net) if and only if the following conditions are satisfied:
1. The labelling of transitions consists of two labelling functions l_{Prov} and l_{Req}, such that: $l_{Prov} : T \longrightarrow Prov \cup \{\lambda,\nu\}$, where $Prov \subseteq A$ is the set of provided (received) services, and $l_{Req} : T \longrightarrow Req \cup \{\lambda,\nu\}$, where $Req \subseteq A$ is the set of required (requested) services.
2. *Instance creation*: the set of places contains a specific *Input* (source) place I, such that ${}^\bullet I = \varnothing$,
3. *Instance completion*: the set of places contains a specific *Output* place O, such that $O^\bullet = \varnothing$.
4. *Visibility*: for any $t \in T$ such that $t \in \{I^\bullet \cup {}^\bullet O\}$: $l(t) \in A$.

The first requirement allows to focusing either upon the server side of a C-net or its client side. Then, the interface of a C-net is the set of its provided and required services. The last requirement states that all the transitions related to the Input place I, and to the Output place O, are necessarily observable actions. They give input (parameters) and output (results) of the performed net.

Notation. We denote by [I] and [O], the markings of the Input and the Output place of CN, and by Reach (CN, [I]), the set of reachable markings of the component-net CN obtained from its initial marking M_N within one token in its Input place I.

Definition 3.2 (completion + reliability = soundness)
Let $CN = (P \cup \{I, O\}, T, W, M_N, l)$ be a *Component-net* (C-net). CN is said to be *sound* if and only if the following conditions are satisfied:
1. *Completion option*: for any reachable marking $M \in$ Reach (CN, [I]), $[O] \in$ Reach(CN, M).
2. *Reliability option*: for any reachable marking $M \in$ Reach (CN, [I]), $M \geq [O]$ implies $M = [O]$.

A *Sound* Component-net has a life-cycle, which satisfies the completion and the reliability options. *Completion* option states that, if starting from the initial state, i.e. activation of the C-net , it is always possible to reach the marking with one token in the output place O. *Reliability* option states that the moment a token is put in the output place O corresponds to the *termination* of a C-net without leaving dangling references.

3.1.2. Operations on C-nets. To define our behavioural subtyping relations, we need three basic operations on the C-nets: abstraction, cancellation of services, and asynchronous composition, used for testing compatibility together with characterizations of type substitutability:
- The *abstraction operator* λ labels as not observable and internal actions, some transitions of a Labelled C-net. It introduces new non-stable states, from which the refusal sets are not taken into account for the failure

semantics. Formally, given a C-net N = (P, T, W, M_N, l), for each H ⊆ A, $\lambda_H(N) = N' = (P, T, W, M_N, l')$ such that l'(t) = l(t) = a, if t ∈ T and a ∈ A \ H, l'(t) = λ else.

- The *cancellation operator* δ labels as not observable, but not internal actions, some transitions of a Labelled net. Cancellation is another kind of abstraction, which does look at the new non-stable states when computing failures. It renames transitions into ν transitions. Formally, given a labelled Petri net N = (P, T, W, M_N, l), for each H ⊆ A, $\delta_H(N) = N' = (P, T, W, M_N, l')$ such that l'(t) = l(t) = a, if t ∈ T and a ∈ A \ H, l'(t) = ν else.

- The *parallel composition* operator ⊕ : C-net × C-net ⟶ C-net computes the set of parallel compositions of traces, interleaving actions. The composition ⊕ is made by communication places allowing interaction through observable services in *asynchronous* way. Given a client C-net and a server C-net, it consists in connecting, through the communication places, the request and accept transitions having the same service names: each accept-transition of the server is provided with an *entry-place* for receiving the requests/replies. Then, the client C-net is connected with the server C-net through this communication place by an arc from each request-transition towards the suitable entry-place and an arc from the suitable entry-place towards each accept-transition. The composition of two C-nets is also a C-net, and this composition is associative. The following definition gives the composition of two C-nets A and B. So, to achieve a syntactically correct compound C-net C = A ⊕ B, it is necessary to add new components for initialization and termination: two new places (an Input and Output places), noted {Ic, Oc}, and two new observable transitions, noted {ti, to}, for interconnecting the input place {Ic} to the original two input places via the first new transition {ti}, and the two original output places to the output place {Oc} via the second new transition {to}.

Definition 3.3 (Composition of C-nets)
Let A = (Pa ∪ {Ia, Oa}, Ta, Wa, Ma, la) and B = (Pb ∪ {Ib, Ob}, Tb, Wb, Mb, lb) be two C-nets such that Pa ∩ Pb = ∅. Let A_A (resp. A_B) be the set of services of A (resp. B). Let {Ic, Oc} ∉ (Pa ∪ Pb) be two new places, and {ti, to} ∉ (Ta ∪ Tb) be two new observable transitions labeled resp. by α and β.
The composed C-net C = A ⊕ B, is given by C = (Pc ∪ {Ic, Oc}, Tc, Wc, Mc, lc), where:

- $A_C = A_A \cup A_B \cup \{\alpha, \beta\}$ the alphabet of observable actions of the C-net C.
- Pc = Pa ∪ Pb ∪ Pint, where Pint ={ps, s∈{$Prov_1$∩ Req_2}∪{$Prov_2$ ∩ Req_1}} is the set of places communication.
- Tc = Ta ∪ Tb ∪{ti, to}.
- lc = $l_{Req} \cup l_{Prov}$, where lc(t) = la(t), if t ∈Ta, lc(t)= lb(t), if t ∈Tb, and lc(ti) = α and lc(to)= β.
- Wc = Wa ∪ Wb ∪ Wint ∪{(Ic, ti), (ti, Ia), (ti, Ib), (to, Oc), (Oa, to), (Ob, to)}, where Wint(t, ps) = 1 if t ∈ Ta ∪ Tb, ps ∈ Pint and $l_{Req}(t) = s$; Wint(ps, t) = 1 if t ∈ Ta ∪ Tb, ps ∈ Pint and $l_{Prov}(t) = s$.
- Mc is such that Mc(ps) = 0, ps ∈ Pint, Mc(Ic) = Mc(Oc) = 0, Mc(p) = Ma(p), if p ∈ Pa, and Mc(p) = Mb(p), if p ∈ Pb.

Example 1 : As an example of the composition of two C-nets, considers A and B shown in figure 1, and the result of their composition in C = A ⊕ B. The ! and ? keywords are the usual sending (required) and receiving (provided) services.

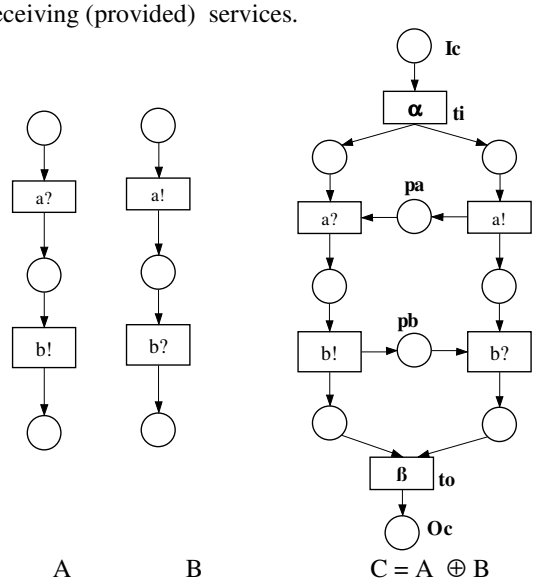

Figure 1. Composition of two C-nets, A ⊕ B.

3.2. Component protocol specification

Component protocols are specified by component-nets formalism allowing to explicitly: (1) identify and characterize elementary services; (2) have compositional semantics in order to deduce emergent interaction among Component protocols. In the following, first we define our Component protocol specification model, which is a template instantiated on C-nets. Then we investigate the composition of component-protocols together with their execution semantics.

Definition 3.4 (Component-protocol)
A Component-protocol CP is a two-tuple, CP = (Behav, Serv), where,

- Behav = (P ∪ {I, O}, T, W, M_N, l) is a C-net describing the life-cycle of CP.
- Serv = (Req, Prov) is an "interface" through which CP interacts with other Component-protocol for instance messaging interface. It is a pair (Req, Prov), where Req is a set of required services, and Prov is the set of provided services by CP, and more precisely by Behav.

A component-protocol is a set of call sequences, that is a set of required and provided services that must be performed by the component; a call is the invocation of a method implemented by the component.

Definition 3.5 (Component protocols composition)
A (Component-) protocol CP = (Behav, Serv), can be composed from a set of primitive protocols, ($Behav_1$, $Serv_1$),…, ($Behav_n$, $Serv_n$), noted CP = $CP_1 \otimes_{...} \otimes CP_n$, as follows:

- Behav is obtained from $Behav_1$, …, $Behav_n$ by connecting $Behav_1$, …, $Behav_n$ through their interfaces in asynchronous way, that is Behav = $Behav_1 \oplus \ldots \oplus Behav_n$.
- Serv = (Req, Prov) is derived from $Serv_i$ = (Req_i, $Prov_i$), i = 1, …, n. Req (or Prov, respectively) is a subset of $\cup Req_i$, i = 1, …, n (or $\cup Prov_i$, i = 1,…, n).

The execution semantics of the Component-protocol is defined according to the interleaving semantics, as follows: when the component executes, then at any given moment exactly one (sub) component-protocol executes. Component protocols interacts with each other through the call of service requests, and messages input or output by a component-protocol are consumed or generated by the component-protocol or its recursively nested sub-component protocol.

4. Compatibility and substitutability

We are now finally ready to define compatibility and behavioral subtyping relations. We show the existing link between compatibility and substitutability concepts, and namely their combination, which seems necessary, when we deal with incremental design of components. First, adequate definitions of protocols compatibility are given together with their characterization by property preservation. Second, useful behavioral subtyping relations related to the principle of substitutability are presented. Finally, the compatibility and the substitutability of component protocols are related to each other with the core theorem. In this paper, among the very numerous semantics, which may be used to compare behaviour of component protocols, only failure and Bisimulation semantics will deal with. The failure semantics involve linear case dealing with deadlock, whereas Bisimulation semantics are the finest and involve the branching case (see [6, 15] for a comparative study of these relations).

4.1. Compatibility and property preservation

In the following we will discuss both compatibility between required and provided interfaces. We call the first compatibility relation, weak compatibility. The choice of the name is due to the fact that weak compatibility relation guarantees safety property.

Definition 4.1 (Weak compatibility)
Let CP_1 = ($Behav_1$, $Serv_1$) and CP_2 = ($Behav_2$, $Serv_2$) be two component protocols. Let CP = $CP_1 \otimes CP_2$ = (Behav, Serv).
CP_1 and CP_2 are Weakly Compatible, noted $CP_1 \approx_{WC} CP_2$, iff $F(\lambda_{Serv2}(Behav)) \subseteq F(Behav_1)$ and $F(\lambda_{Serv1}(Behav)) \subseteq F(Behav_2)$.

The definition of weak compatibility uses failure semantics, and hence reasons about the deadlock. It ensures that the possible failures of CP must be a subset of the corresponding failure of CP_i, i=1, 2.

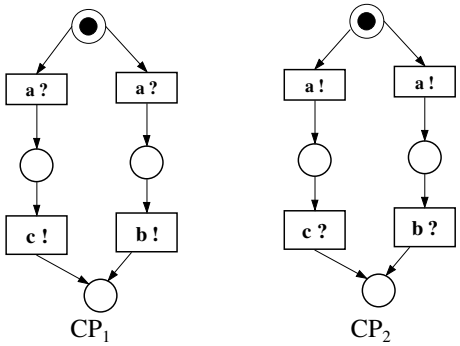

Figure 2. $CP_1 \approx_{WC} CP_2$ does not hold, where $Serv_1$ = ({b, c}, {a}) and $Serv_2$ = ({a}, {b, c}).

Example 2: As an example, it is easy to prove that protocols CP_1 and CP_2 shown in figure 2 are not related by the weak compatibility relation, that is $CP_1 \approx_{WC} CP_2$ does not hold, since (a!.a?.b!, {a!, c?, b?}) ∈ F(CP), whereas (a!, {a!, c?, b?}) ∉ $F(CP_2)$. Further, CP_1 and

CP_2 shown in figure 3 are related by the weak compatibility, that is $CP_1 \approx_{WC} CP_2$ holds.

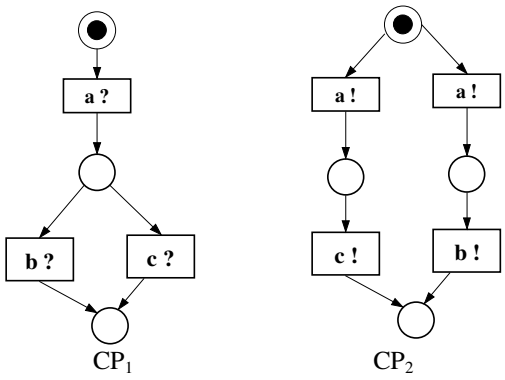

Figure 3. $CP_1 \approx_{WC} CP_2$, where $Serv_1 = (\varnothing, \{a, b, c\})$ and $Serv_2 = (\{a, b, c\}, \varnothing)$.

According to the above example together with theorem 4.1 (see below), weak compatibility relation is a very powerful way to guaranty the correctness of the protocol when reasoning about safety property like the deadlock-freeness. Sometimes this is not enough, and we want to claim that some liveness properties are preserved by the Protocol's composition like the proper (or successful) termination. This is the aim of strong compatibility relation, which is based on branching bisimulation semantics.

Definition 4.2 (Strong compatibility)
Let $CP_1 = (Behav_1, Serv_1)$ and $CP_2 = (Behav_2, Serv_2)$ be two component protocols. Let $CP = CP_1 \otimes CP_2 = (Behav, Serv)$.
CP_1 and CP_2 are Strongly Compatible, noted $CP_1 \approx_{SC} CP_2$, iff $\lambda_{Serv2}(Behav) \approx_{BiSim} Behav_1$, and $\lambda_{Serv1}(Behav) \approx_{BiSim} Behav_2$.

Strong compatibility of two protocols preserves the behaviour of each protocol in the compound protocol according to the branching bisimulation semantics.

Example 3: As an example**,** it is easy to prove that protocols CP_1 and CP_2 shown in figure 3 are not related by the strong compatibility, that is $CP_1 \approx_{SC} CP_2$ does not hold, since the (branching) semantics of CP_1 are not preserved in the compound CP, whereas the two protocols shown in figure 4 are related by the strong compatibility, that is $CP_1 \approx_{SC} CP_2$ holds.

Last but not least, Strong Compatibility always implies Weak Compatibility, and this last is finer than weak Compatibility:

Property 4.1 (Hierarchy of compatibility relations)
The compatibility relations form a hierarchy: $\approx_{SC} \Rightarrow \approx_{WC}$.

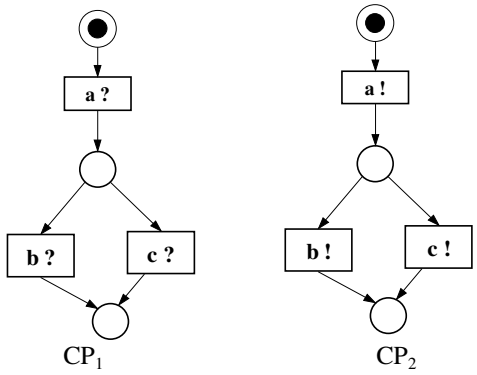

Figure 4. $CP_1 \approx_{SC} CP_2$, where $Serv_1 = (\varnothing, \{a, b, c\})$ and $Serv_2 = (\{a, b, c\}, \varnothing)$.

We have defined compatibilities between protocols. Now, we investigate properties preservation such as, the safety property: no deadlock between protocols will occur, and the liveness property: the successful termination of a protocol. These two properties are respectively related to the completion and the soundness of the underlying C-net, describing the life-cycle of the component. First, we give the definition of these two protocol's properties, and then the core theorem characterising the weak and the strong compatibility by property preservation is given.

Definition 4.3 (Safety and liveness property)
Let CP = (Behav, Serv) be a protocol.
1.CP is deadlock-free iff Behav satisfies the completion option . (def. 3.2).
2. CP terminates successfully iff Behav is sound.

Theorem 4.1 (Property preservation)
Let CP_1, CP_2 be two component protocols, and $CP = CP_1 \otimes CP_2$.
1. $\otimes$ preserves deadlock-freeness for protocols related by weak compatibility: $CP_1 \approx_{WC} CP_2$ and CP_i, i=1,2, are deadlock-free $\Rightarrow$ CP is deadlock-free.
3. $\otimes$ preserves successful termination for protocols related by strong compatibility: $CP_1 \approx_{SC} CP_2$ and CP_i, i=1,2, terminates successfully $\Rightarrow$ CP terminates successfully.

4.2. Substitutability of protocols

Substitutability of protocols is the capacity to replace one protocol by another one without losing behaviours. Our main interest is to define behavioural subtyping relations capturing the principle of substitutability [10]. In this paper we propose to base subtyping relations on the preorder which have been introduced to compare the behaviour of concurrent systems, such as the failure and bisimulation semantics. The first proposed subtyping relation, called weak subtyping, deals with refusals (failures) services (provided and required) by the component protocol, and is adapted for a single access component. Instead, the second one, called optimal subtyping is adapted for shared components. The third one, called strong subtyping, which is more restrictive than the weak subtyping, is based on bisimulation semantics dealing with the branching case and is adapted for a single access component protocol. In our context, there are two possibilities to treat old and new services : we hide them (abstraction) or we cancel them (cancellation).

Definition 4.4 (Weak subtyping)
Let $CP_1 = (Behav_1, Serv_1)$ and $CP_2 = (Behav_2, Serv_2)$ be two component protocols such that $Serv_i = (Req_i, Prov_i)$, i=1,2, $Prov_1 \subseteq Prov_2$ and $Req_2 \subseteq Req_1$. Let $G = Prov_2 \setminus Prov_1$ and $H = Req_1 \setminus Req_2$.
CP_2 is less equal to CP_1 w.r.t Weak Substitutability, denoted $CP_2 \leq_{WS} CP_1$, iff $F(\lambda_G(Behav_2)) = F(\lambda_H(Behav_1))$.

If CP_2 is less or equal to CP_1 w.r.t *Weak Substitutability,* then the protocol CP_1 can be substituted by a protocol CP_2 and the client-component will not be able to notice the difference since the new provided services added in the sub-protocol CP_2 are considered unobservable, through the abstraction operator λ_G, and the two protocols are failure equivalent on the super-protocol's provided services as well as on the sub-protocol's required services.

Example 4: As an example, consider the protocols CP_1 and CP_2 shown in figure 5. It is easy to prove that $CP_2 \leq_{WS} CP_1$ holds, since for G = {c, d} and H = ∅, we have $F(\lambda_G(Behav_2)) = F(\lambda_H(Behav_1))$.

The above example shows why we are still not at the end of defining behavioural substitutability relations based on failure semantics. The protocol CP_2 provides services {b, c, d}, where the new services to be added are {c, d}. So, the client of the old service {b}, might indeed notice the differences, if for instance another client is requesting the new service {c}. This extension is however weak, and is adapted for a single access protocol. Instead, for shared component protocol, the second type of substitutability, optimal subtyping, which is more restrictive than weak substitutability, is necessary to capture the desired substitutability.

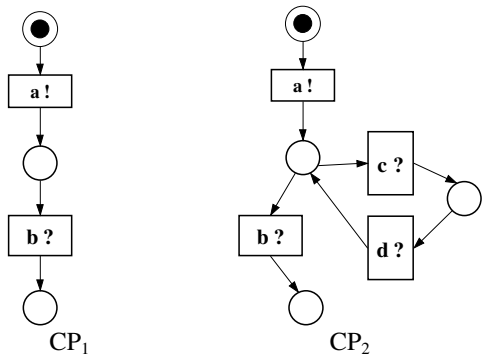

Figure 5. $CP_2 \leq_{WS} CP_1$, where $Serv_1 = (\{a\}, \{b\})$ and $Serv_2 = (\{a\}, \{b, c, d\})$.

Definition 4.5 (Optimal subtyping)
Let $CP_1 = (Behav_1, Serv_1)$ and $CP_2 = (Behav_2, Serv_2)$ be two component protocols such that $Serv_i = (Req_i, Prov_i)$, i=1,2, $Prov_1 \subseteq Prov_2$ and $Req_2 \subseteq Req_1$. Let $G = Prov_2 \setminus Prov_1$ and $H = Req_1 \setminus Req_2$.
CP_2 is less equal to CP_1 w.r.t Optimal Substitutability, denoted $CP_2 \leq_{OS} CP_1$, iff $F(\delta_G(Behav_2)) = F(\lambda_H(Behav_1))$.

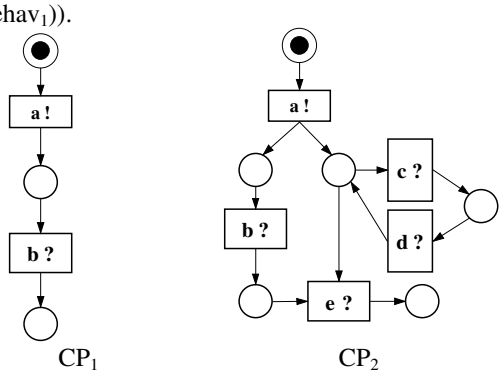

Figure 6. $CP_2 \leq_{OS} CP_1$, where $Serv_1 = (\{a\}, \{b\})$ and $Serv_2 = (\{a\}, \{b, c, d, e\})$.

Example 5: As an example, consider the protocols CP_1 and CP_2 shown in figure 6. It is easy to prove that $CP_2 \leq_{OS} CP_1$ holds, since for G = {c, d, e} and H = ∅, we have $F(\delta_G(Behav_2)) = F(\lambda_H(Behav_1))$. The new provided services {c, d} can be concurrently executed with the old one {b}. For the client which requests the

{b} service, feature the same service as before is now possible.

Definition 4.6 (Strong subtyping)
Let $CP_1 = (Behav_1, Serv_1)$ and $CP_2 = (Behav_2, Serv_2)$ be two component protocols such that $Serv_i = (Req_i, Prov_i)$, i=1,2, $Prov_1 \subseteq Prov_2$ and $Req_2 \subseteq Req_1$. Let $G = Prov_2 \setminus Prov_1$ and $H = Req_1 \setminus Req_2$.
CP_2 is less equal to and CP_1 w.r.t Optimal Substitutability, denoted $CP_2 \leq_{SS} CP_1$, iff $\lambda_G(Behav_2) \equiv_{BiSim} \lambda_H (Behav_1)$.

If CP_2 is less or equal to CP_1 w.r.t *Strong Substitutability,* then the protocol CP_1 can be substituted by a protocol CP_2 and the client-component will not be able to notice the difference since the new provided services added in the protocol CP_2 are considered unobservable, through the abstraction operator λ_G, and the two protocols are bisimilar on the super-protocol's provided services as well as on the sub-protocol's required services.

Example 6: last but not least, consider the protocols CP_1 and CP_2 shown in figure 7. It is easy to prove that $CP_2 \leq_{SS} CP_1$ holds, since for G = {d} and H = {b}, we have $\lambda_G(Behav_2) \approx_{BiSim} \lambda_H (Behav_1)$.

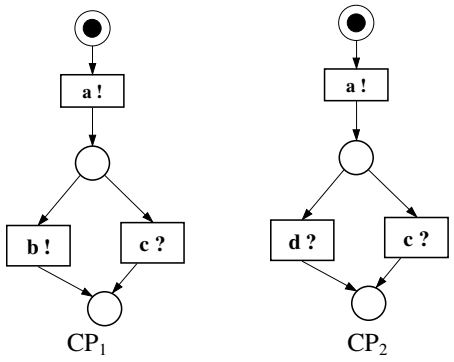

Figure 7. $CP_2 \leq_{SS} CP_1$, where $Serv_1$ = ({a,b}, {c}) and $Serv_2$ = ({a}, {c, d}).

Property 4.2 (Hierarchy of subtyping relations)
1. The subtyping relations $\leq_H$, H ∈ {WS, OS, SS}, are reflexive and transitive.
2. The subtyping relations form a hierarchy: $\leq_{OS} \Rightarrow \leq_{WS}$ and $\leq_{SS} \Rightarrow \leq_{WS}$.

As expected, the $\leq_H$ subtyping relations, where H∈ {OS, SS, WS} are compositional for the composition operator ⊗; for instance, extending (resp. reducing) the provided (resp. required) services of a protocol also extends (resp. reduces) the provided (resp. required) services of its composition with any client/server – component protocol.

Property 4.3 (Subtyping are compositional)
Let CP_1, CP_2 be two component protocols such that $CP_2 \leq_H CP_1$ where H ∈ {WS, OS, SS}. Then for any component protocol CP, we have $CP \otimes CP_2 \leq_H CP \otimes CP_1$.

4.3. Compatibility and substitutability

Substitutability guarantees the transparency of changes of protocols to clients. Namely, the compatibility between components should not be affected by these changes. The following theorem study the preservation of compatibility by substitutability, dealing with the two compatibility relations together with the three subtyping relations given in this paper.

Theorem 4.2 (Soundness of subtyping relations)
Let CP_1 and CP_2 be two component protocols.
1. $CP_2 \leq_{WS} CP_1$ iff $(\forall CP, CP \approx_{WC} CP_1 \Rightarrow CP \approx_{WC} CP_2)$
2. $CP_2 \leq_{SS} CP_1$ iff $(\forall CP, CP \approx_{SC} CP_1 \Rightarrow CP \approx_{SC} CP_2)$.

We have proposed three relations which are suitable as behavioural subtyping relations for component protocols: weak subtyping is the right relation if we would like to preserve weak compatibility, and guarantees the preservation of the deadlock-freeness, optimal subtyping is useful to preserve weak compatibility also for shared components. The third subtyping relation, strong subtyping, preserves strong compatibility, and then guarantees the preservation of the liveness property, which is the successful termination of protocols. The weak and the strong subtyping are only suitable for components with single access. The results can be applied in an incremental design with behavioural subtyping by checking what kind of subtype a sub-protocol is, and then deduce what properties of the super-protocol are preserved. The table 1 summarises the results for theorem 4.2.

Table 1. Preservation of Compatibility by Substitutability.

Subtyping	***Compatibility***		*Component Access*	
	Weak	Strong	Single	Shared
Strong	√	√	√	
Optimal	√		√	√
Weak	√		√	

5. Discussion and related work

The aim of this paper is to integrate specification and verification methods into the Component Based Development of protocols. The specification of protocols is based on Petri nets. Each protocol has an interface and an internal process, specified by a component-net, allowing to specify behavioural property of components such as call sequences accepted (as specified in a provided interface) and call sequences required (as specified in the required interface) by the component. To study compatibility of components, two notions of compatibility and three subtyping relations between protocols are proposed together with the property preservation of the proposed compatibility relations to the degree of change. We furthermore studied the interconnection between compatibility and substitutability, and investigated the characterisations of compatibility by behavioural subtyping.

The approach presented in this paper leads to define compatibility relations together with subtyping relations for component protocols having good properties. Our behavioural subtyping relations take into account the non-deterministic, the composition mechanism of protocols, component's access (single or shared), and determine automatically the compatibility, which is preserved. The next step for this work is to explore the notion of parametric contracts [16] in the definition of protocols compatibility and substitutability. Parametric contracts link the provided- and required interfaces of the same protocol, and seems to be interesting in the re-use of protocols, in different environment, when the required interfaces are not fully meet, but the component can still offer part of its provided interface. Our aim is to define flexible compatibility and substitutability of protocols, depending on the context of use of components.

Related Work. There are many approaches to the specification of protocols in components or object-oriented systems, ranging from state machine based approaches via Petri nets and logic predicates to process calculus. The use of state machines to specify protocols and to check their compatibility is the well known approach. In [14], authors propose an enhanced architectural description language for component behaviour with protocols specified by regular expressions. However, they do not consider property preservation of protocols composition. Nierstrasz [13] uses regular types to investigate service availability of active objects. He defines notions of compatibility and substitutability of protocol-enhanced objects, and only describes the provided interface using finite state machines. In our previous work [8], we studied property preservation by substitutability in this setting. The work presented in this paper can be seen as an extension to components of the previous one, since we take into account both the provided and required component interfaces. In addition, compatibility and substitutability of protocols for non-deterministic systems is studied. In [19], substitutability of active objects is studied for non-deterministic systems. Our approach can be seen as an extension of this work since, in addition, it deals with compatibility and substitutability of components together with their combination. In [18], Petri nets are used for modelling components within software architectures. This work is close to ours, since it uses component-nets formalism to model the life-cycle of components and to check their substitutability. The main contribution of this work is to propose a framework to model software architectures, and address consistency (will a component "fit" or not?) at the level of a single component and at the level of a system architecture. Authors prove that consistency implies the correct behaviour of the overall system, i.e., the system is free of deadlocks; this result is based on the fact that the proposed behavioural inheritance relations are compositional. Nevertheless provided and required interface are not distinguished in the specification of components, and then compatibility of components is not addressed. In contrast, our work specify compatibility of component protocols w.r.t the provided and required interfaces, their characterisations by property (safety and liveness) preservation, and finally to study the preservation of compatibility by substitutability. In [1], authors differentiate between components, described by a set of ports and connectors, for glueing components, described by a set of roles. In our approach, we do not distinguish at the specification level of components protocols between these categories, and both components and connectors are called components. Further, in that work, the notion of compatibility (of a port with a role) is only based on the deadlock-freeness, whereas in our work the compatibility between components protocols is related both to safety (deadlock-freeness) and liveness (proper termination) property. In [2], authors present a relation of compatibility in the context of pi-calculus which formalizes the notion of conformance of behaviour between software components. This approach is enhanced with the definition of a relation of inheritance among processes. This relation, based on (bi) simulation between process, preserves compatibility and indicates whether a process can be considered as an extension of another one. This work is close to ours,

since the compatibility relation is based on failure semantics, and then related to the deadlock-freeness property. Our approach can be seen as an extension of this work since, in addition, we deal with compatibility related to the liveness property and its preservation by behavioural together with substitutability of protocols for shared components

In [21], predicate approaches are used for specifying protocols, providing a higher modelling power than the approaches given above. The idea is that formal specifications are pre- and postconditions written as predicates in first order logic; nevertheless checking protocols compatibility and substitutability remains very complex and non-computable. In [3], an approach using process calculus based on the concept of "contract" is proposed; in this approach, modalities on the sequences of actions to be performed by interfaces are introduced in the definition of compatibilities between interfaces, and sound composition of components is studied w.r.t property preservation such as external deadlock-freeness and message consumption. This approach is close to ours, since the compatibility rules are based upon bisimulation semantics, like those used in our strong compatibility and substitutability relations, together with their characterisations by property preservation. Nevertheless, it is based on a logic based calculi, and has a similar drawbacks as predicate based approaches [21], that is the complexity to checks compatibility together with safety and liveness property.

References

[1] R.Allen, and D.Garlan. Formalizing architectural connection. In Proc. CSE'94, pages 71-80, Sorrento(Italy), May 1994.

[2] C. Canal, E. Pimentel, and J.M. Troya. "Compatibility and inheritance in software architectures" Science of Computer Programming, 41(2):105-138. 2001.

[3] C. Carrez, A. Fantechi, E. Najm. "Behavioural Contracts for a Sound Assembly of Components", FORTE 2003, LNCS 2767, pp 111-126, 2003.

[4] OMG. "The Common Object Request Broker: Architecture and Specifications", Report V2.4, OMG, 2000.

[5] L. De Alfaro, T.A. Henzinger. "Interface automata", In proc. of ESEC/FSE, volume 26, 5 of Software Engineering Notes, ACM (2001).

[6] R. Van Glabbeek, U. Goltz. "Equivalence Notions for Concurrent Systems and Refinement of Actions". In MFCS 89, LNCS 379, Springer-Verlag 1989.

[7] N. Hameurlain. "On Compatibility and Behavioural Substitutability of Component Protocols". Internal report, University of Pau (F), 2005. Available at http://www.univ-pau.fr/~hameur.

[8] N. Hameurlain. "Behavioural Subtyping and Property Preservation for Active Objects", Fifth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS'02, pp 95-110, Kluwer 2002.

[9] Sun Microsystems. "JavaBeans 1.01 Specification", Available at http://java.sun.com/beans.

[10] B. H. Liskov, J. M; Wing. "A Behavioral notion of Subtyping", ACM Transactions on Programming Languages and Systems, 16 (6): 1811-1841, November 1994.

[11] R. Milner. "A Calculus of Communicating Systems", LNCS, 92, 1980.

[12] T. Murata. "Petri Nets: Properties, Analysis and Applications". In Proc. of the IEEE, vol. 77, N° 4, pp. 541-580, April 1989.

[13] O. Nierstrasz. "Regular types for Active Objects". In ACM SIGPLAN Notices, 28 (10); Proceedings of OOPSLA'93, Washington DC, pp. 1-15, 1993.

[14] F. Plasil, S. Visnovsky. "Behaviour Protocols for Software Components". In Transactions on Software Engineering, IEEE (2002).

[15] L. Pomello, G. Rozenberg, C. Simone. "A Survey of Equivalence Notions for Net Based System". Advances in Petri Nets 1992; G. Rozenberg Ed., LNCS 609, Springer-Verlag 1992.

[16] R.H. Reussner, J. Happe, A. Habel. "Modelling Parametric Component Contracts and the State Space of Composite components by Graph Grammars". In ETAPS/FASE 2005, LNCS 3442, pp 80-90, Springer.

[17] C. Szyperski, "Component Software-Beyond Object-Oriented Programming", Addison-Wesley, 2002.

[18] W.M.P. Van der Aalst, k.M. van Hee, R.A. van der Toorn. "Component-Based Software Architectures: A framework Based on inheritance of Behaviour". Working Paper Series 45, Eindhoven University of Technology, 2000.

[19] H. Wehrheim. "Behavioural Subtyping relations for Active Objects", Formal Methods in System Design, 23:143-170, 2003.

[20] D.M. Yellin, R.E. Strom. "Protocol Specification and Component Adaptors", ACM TPLS 19: 292-333, 1997.

[21] A. M. Zaremski, J. M. Wing. "Specification Matching of Software Components". ACM Transactions of Software Engineering and Methodology, 6 (4): 333-369, (1997).