

A Component Calculus for Modeling the Olan Configuration Language

Jean-Yves Vion-Dury, Luc Bellissard, Vladimir Marangozov

▶ **To cite this version:**

Jean-Yves Vion-Dury, Luc Bellissard, Vladimir Marangozov. A Component Calculus for Modeling the Olan Configuration Language. [Research Report] RR-3231, INRIA. 1997. inria-00073458

HAL Id: inria-00073458
https://inria.hal.science/inria-00073458

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Component Calculus for Modeling the Olan Configuration Language

Jean-Yves Vion-Dury, Luc Bellissard, Vladimir Marangozov

N ° 3231

August 1997

THÈME 1



ISSN 0249-6399



A Component Calculus for Modeling the Olan Configuration Language

Jean-Yves Vion-Dury,* Luc Bellissard,† Vladimir Marangozov‡

Thème 1 — Réseaux et systèmes
Projet Sirac

Rapport de recherche n ° 3231 — August 1997 — 27 pages

Abstract: Components will certainly become a key concept for the next generation of software architectures because of their impact on effective software reuse, real interoperability and integration. Within the Olan project [18], we face the difficulty of defining an operational semantics able to reflect the diversity of execution models involved in real applications. Existing process calculi offer the required abstractions such as encapsulation and process equivalences, but they rely on the fundamental assumption that agents are active, i.e autonomously able to initiate communication. However, components, viewed as software pieces with explicit interfaces, require a notion of passive composition that allows, for instance, several components to be traversed by a same process. In this paper, we introduce a calculus, named ICCS, which extends the Milner's CCS calculus with (1) an operator for passive composition, and (2) selective interactions. While preserving the powerful theory of process equivalences established for CCS, this calculus provides an operational definition of *passive* components and allows thus to establish the basis of an operational semantics for the Olan Configuration Language.

Key-words: Distributed Application Programming, Architectural Definition Language, operational semantics, process calculus

(Résumé : tsvp)

A short version was published in the Second International Conference on Coordination Models and Languages, Berlin, Germany, September 1-3, 1997.

* Rank Xerox Research Centre, 6 Chemin de Maupertuis, 38240 Meylan, France - Université Joseph Fourier, Grenoble
† INRIA Rhône-Alpes, Projet SIRAC, 655, avenue de l'Europe, 38330 Montbonnot Saint Martin, France - Institut National Polytechnique, Grenoble
‡ INRIA Rhône-Alpes, Projet SIRAC, 655, avenue de l'Europe, 38330 Montbonnot Saint Martin, France - Université de Savoie, Chambéry

Unité de recherche INRIA Rhˆone-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 04 76 61 52 00 – Télécopie : (33) 04 76 61 52 52

Un calcul sur les composants pour le langage de configuration Olan

Résumé : La notion de composants va certainement devenir un des concepts clés pour la nouvelle génération d'architectures logicielles. En effet, les composants et les langages de description d'architecture favorisent la construction d'une application par réutilisation effective de modules logiciels existants en décrivant les interconnexions entre ceux-ci.

Dans le projet Olan [18], nous avons choisi de définir une sémantique opérationnelle pour le langage d'interconnexion, capable de refléter la diversité des différents modèles d'exécution présent dans les applications réelles. Les calculs formels sur les processus existant offrent les abstractions nécessaires, telles que l'encapsulation et l'équivalence de processus, mais ils reposent sur l'hypothèse fondamentale que les agents sont actifs, i.e. que chaque agent est mis en oeuvre par un processus capable d'initialiser une communication de manière autonome. Toutefois, la notion de composants considérés au sens large comme des entitées logicielles dont l'interface est rendue explicite, requière une notion de composition "passive". Cette composition permettrait d'exprimer par exemple, que plusieurs composants sont traversés par un seul et mˆeme processus.

Dans ce rapport, nous introduisons un calcul formel, nommé ICCS, qui d'une part étend CCS de Robin Milner avec un opérateur de composition passive et d'autre part introduit de la sélectivité dans les intéractions. Le calcul formel proposé permet de préserver les résultats sur l'équivalence de processus établis pour CCS, de donner une définition opérationelle des composants *passifs* et ainsi d'établir la base d'une sémantique opérationnelle pour le langage de configuration Olan.

Mots-clé : Contruction d'applications réparties, programmation constructive, sémantique opérationnelle, algèbre de processus

1 Introduction

Configuration languages such as Darwin [19] or Olan [1] aim at using components as the basic structure for programming distributed applications, and thus take full advantage of their properties: reusability of software at minimal cost, explicit rendering of the software architecture and clear separation of the communication code from the application specific code.

The first benefit of such an approach is to ease the distribution of component-based applications over various hardware platforms, and to significantly help the configuration, monitoring and administration tasks.

Other key issues for component technologies are related to software reuse, integration and interoperability. The reuse is addressed through the notion of *interfaces* which define offered services, as well as required services in order to make explicit the dependencies of the component, and thus allowing its use in various contexts. The first theoretical difficulty is to provide enough information at the interface level in order to allow compatibility checking, but to avoid too complicated specifications. The second difficulty, for component-based language designers, is to provide guidelines and methods for encapsulating heterogeneous pieces of code (possibly written in different programming languages) in order to make them work together. The "programmer in the largë [14] should be able to build up new applications from various primitive components, with no deep knowledge of their internal implementations. Here again, the interface definition must be able to provide relevant information concerning the semantics of the component. A third difficulty is related to interoperability, i.e. provide coordination and data transformation in order to adapt entities that were not initially designed to work together. The challenge here is to propose, as automatically as possible, the communication specific code that performs the adaptation [4].

Clearly, theoretical foundations are needed that allow the modeling of the behavior of components and offer effective methods for computing component compatibility as well as adaptations toward ad hoc or generic agents, specialized in communication and coordination tasks.

Process calculi, as CSP [6], CCS [9] and the π-calculus [11] offer formal tools for modeling general concurrent systems through the notion of communicating processes. The CCS calculus appears to be the closest to our requirements because:

1. CCS is based on an observational approach (as opposed to more specification-oriented calculi)
2. Basic hypotheses are weak (for instance processes are not forced to synchronize on communication channels), and also the calculus is general and concise (only one alternation operator, no specific termination actions)
3. Process equivalence theory is particularly developed, and leads to subtle and important criteria for behavioral compatibility. This last point is central to our requirements concerning component composition, because compatibility criteria must be defined, and interfaces must correctly reflect encapsulated behaviors.
4. Numerous extensions have been proposed which are of particular interest in our area, such as distributed real time processes calculi [2] [3].

The key issue addressed in this paper is the use of a formal calculus to model applications constructed and configured with Olan. Olan rely on a component-based model that describes applications as interconnections of components, with no restrictions on the execution model of those components. Indeed, components may encapsulate active entities or software (e.g. a multi-threaded server) as well as passive entities (e.g. a framework of non-active classes or a library). CCS as well as other calculi are all based on the hypothesis that components are processes and the problem of interconnecting components is thus addressed as process composition. The idea of the remaining sections is to adapt the CCS calculus to the Olan Component Model where the information about the execution model is exhibited at the interface level of components, thus allowing totally passive components to be interconnected to any kind of other components. Section 2 presents ICCS, an extension of CCS that allows the composition of active as well as passive entities. Section 3 details the applicability of such a calculus to Olan and finally, we conclude by comparing our proposition with related works and the existing approaches in this area.

2 The Olan Component Model

The Olan Configuration Language (OCL) belongs to the class of Module Interconnection Language(MIL) [14]. Such languages aim at clearly separating the programming phase of individual components from the configuration phase which consists of the description of the whole application. This latter phase aims at assembling and interconnecting components together according to various schemas of interaction [1]. The central elements of OCL are components and connectors. An application is described and configured as a hierarchy of interconnected components, the leaves of which are basic software units encapsulated in so-called *primitive components*. The nodes of this hierarchy are more complex components (namely *composite components*) which are constructed from interconnected components from a lower level of the hierarchy. The object that realizes an interconnection is called a *connector*. Its role is to ensure properties to the behavior of an inter-component communication when the application is executed.

Components A component (see Fig 1) is made of an interface and an implementation. The interface, like in other MIL [15], describes the services that the implementation provides to other components, along with the services it requires at run-time. The implementation, either primitive or composite, fulfills the requirements expressed at the interface level; in other words, it maps the interface to the encapsulated pieces of software.

Interface Interfaces are specified in OCL with a language and a type system derived from CORBA IDL [16]. Interfaces are described as a set of services (plus associated signatures) and attributes. While being a syntactic extension of IDL, it offers richer constructs for the description of services. Indeed, in order to comply to the Olan component model and to faithfully reflect the behavior of a component (and thus the encapsulated pieces of software), various types of services specify the provisions and the requirements of a component, as well as some information about the execution

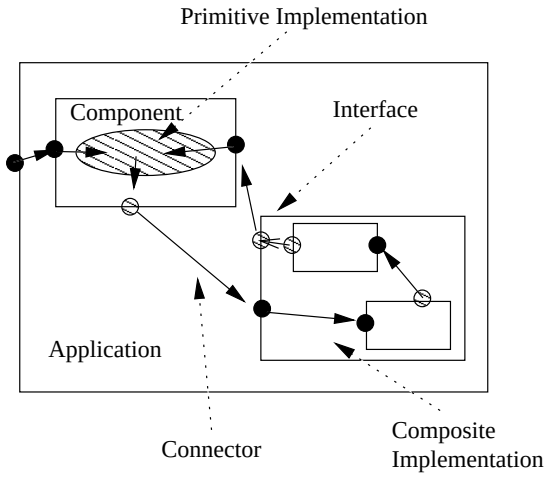


Figure 1: Olan components

Services	Data Flow	**Execution Model**
Provide ●	in/out parameters	No specific constraints (usually synchronously executed, as a procedure)
React ■	in parameters	The encapsulated implementation must contain an execution flow that handles the service processing in an asynchronous way.
Require ○	in/out parameters	The encapsulated implementation must be designed for synchronous external calls (e.g procedure call)
Notify □	in parameters	The encapsulated implementation must be designed for synchronous external calls (e.g procedure call)

Figure 2: Interface services

model of the implementation of the component. Fig 2 shows the current types of services that exist in OCL with their corresponding hypothesis concerning the data-flow and the execution model.

Connectors Within any composite implementation, the application designer describe the interconnections of components. The interconnections specify the way components should communicate with one another. At first, an interconnection is a binding of the component's requirements (*Require* or *Notify* services) to the component's provisions (*Provide* or *React* services) by the means of a connector. More than just a "binding object" between components, connectors ensure the adaptation between compliant but not necessarily compatible interfaces, the control of the communication (for instance, in terms of synchronization of execution flow) and the effective transport of information through the use of a configurable communication protocol (e.g. TCP or UDP) and/or mechanism (e.g. an ORB or simple sockets).

The benefit of the use of Olan is to provide an overall vision of the application architecture to the designer and the developper. The architecture is here considered as the components of the system as well as the way components communicate with each other. In the current version of OCL, components are described with a pure functionnal interface, thus lacking some kind of information about the internal behavior and semantics of the encapsulated implementation. Obviously, this is a major problem as far as the reusability issue of components is concerned. How can an application designer integrate a component into his system if the only available information is simply the functionnal interface? Even if a documented API is available, errors issued from misunderstanding or ambiguities are still prone to happen. In OCL, the interface although reflects some information about the execution model of the component's implementation. However, there is still the need to show (part of) the internal semantics of outgoing communication. This is why the use of a formal specification of components based on a theoretical calculus, namely ICCS for *Interconnected Component Calculus*, is beneficial for the correct and desired assembling of components when producing an application.

3 ICCS: an Interconnected Component Calculus

3.1 Overview

We propose to refine the CCS calculus into a new calculus able to model the notion of *passive* components as well as *active* components, while preserving the powerful theoretical tools of process equivalence, needed for the definition of component composition. Therefore, we propose a new static composition operator $\triangleleft$ that allows the combination of properly active components with passive components, by expressing the intuitive notion of *activity flow transfer*.

In CCS, the fundamental assumptions are message-passing exchanges (as opposed to shared memory models) and asynchrony across processes. The parallel composition is thus a construction which expresses the potential transfer of data *and* the concurrency among processes. But if more general composition relationships are needed, such as for assembling software pieces, then some different assumptions are required. For instance, a standard procedure call can be viewed as a construction that transfers both values *and* activity flow, since no action can be performed at the upper level by the calling activity.

The most common response to this difficulty is to model the synchronous call by two activities synchronized correctly. We argue that this is possible, but too general, and thus it does not exactly

fulfill the requirements. The following example emulates a synchronous call in CCS:
$(\bar{a}.a.0|a.\bar{b}.b.\bar{a}.0) \setminus \{a\}$, which produces the derivation $\xrightarrow{\tau}\xrightarrow{\bar{b}}\xrightarrow{b}\xrightarrow{\tau}$.

By using the value-passing form of the calculus, it is possible to simulate a synchronous call with input parameter x and output parameter y:
$(a\bar{x}.a(y).0|a(w).b\bar{w}.b(z).\bar{az}.0) \setminus \{a\}$, which produces the derivation $\xrightarrow{\tau}\xrightarrow{b\bar{x}}\xrightarrow{b(z)}\xrightarrow{\tau}$.

This last one is fully representative (it models data transfer), and correct, since the calling activity is suspended during the call. But the following expression is correct too, although not representative of the activity transfer:
$(a\bar{x}.c\bar{y}.a(y).0|a(w).b(z).\bar{az}.0) \setminus \{a\}$, since the calling activity performs an output communication action *c* before the return of the call. Thus, in CCS, there is no *operational* distinction between these two cases: conditions for ensuring the validity of *passive composition* do not appear directly in the calculus. We propose to go even further and to make this difference explicit by proposing a new composition operator $\lhd$ in order to model the activity transfer *and* the value passing.

The idea is to allow output communication actions on the left side of the operator and input actions on the right side, and to commute the operands when actions are performed. Of course, internal τ actions must be possible to express the passing of values between the two operands. Thus, the previous example becomes:

$$\begin{aligned}(\bar{a}.a.0 \lhd a.\bar{b}.b.\bar{a}.0) \setminus \{a\} &\xrightarrow{\tau} (\bar{b}.b.\bar{a}.0 \lhd a.0) \setminus \{a\} \\ &\xrightarrow{\bar{b}} (a.0 \lhd b.\bar{a}.0) \setminus \{a\} \\ &\xrightarrow{b} (\bar{a}.0 \lhd a.0) \setminus \{a\} \\ &\xrightarrow{\tau} (0 \lhd 0) \setminus \{a\}\end{aligned}$$

and, more interesting, the previous "non-desired" example is now discriminated, since the passive composition produces a different derivation tree:

$$\begin{aligned}(\bar{a}.\bar{c}.a.0 \lhd a.\bar{b}.b.\bar{a}.0) \setminus \{a\} &\xrightarrow{\tau} (\bar{b}.b.\bar{a}.0 \lhd \bar{c}.a.0) \setminus \{a\} \\ &\xrightarrow{\bar{b}} (\bar{c}.a.0 \lhd b.\bar{a}.0) \setminus \{a\} \\ &\qquad\xrightarrow{b} (\bar{a}.0 \lhd \bar{c}.a.0) \setminus \{a\} \\ &\text{or} \quad\xrightarrow{\bar{c}} (b.\bar{a}.0 \lhd a.0) \setminus \{a\}\end{aligned}$$

However, this notion we can call *preemption* leads to difficulty if we want to preserve a reasonable level of concurrency.

What does the following composition mean if the entire left subtree is suspended while executing the preemptive communication with the right component:
$(\bar{a}.a.0|(P_1|P_2)) \lhd a.\bar{b}.b.\bar{a}.0$?

In such a case, the concurrent sub-system $(P_1|P_2)$ cannot evolve asynchronously, even if $\bar{a}.a.0$ alone is involved in the passive composition. We propose to tackle this problem (in other terms, the problem of managing the compatibility between the two composition operators) by introducing the

notion of *selective interaction*. The key idea is to refine the component composition by associating *paths* to action labels of the transition system, and to use these paths in order to bring more selectivity to the interaction.

The previous example could become $((\bar{a}.a.0) :: p | (P_1 | P_2)) \triangleleft_p a.\bar{b}.b.\bar{a}.0$, expressing the fact that $(\bar{a}.a.0) :: p$ is concerned by the passive composition, but not $(P_1|P_2)$, which can run asynchronously. The following sub-sections make this precise and demonstrate that this approach solves the compatibility problem. Moreover, it also brings a new way of specifying static component interconnections, as opposed to the dynamic binding capabilities of the π-calculus, and eases the description of configuration languages such as Olan.

3.2 The syntax

The set $\mathcal{P}$ of ICCS formulae, ranged over by P_i or Q_i is defined by two sets of syntax rules.The first one is very similar to the original set proposed by Milner for CCS [9]:

P	$::=$	0	*Nul component: do nothing*
		X	*Component variable*
		$\alpha_p.P$	*Prefix*
		$P + P$	*Alternate choice*
		$P \setminus L : M$	*Restriction*
		$P[f : g]$	*Relabeling*
		$P \mid_{mn} P$	*Concurrent composition*
		$recX : P$	*Recursive operator*

The second set defines the new constructions:

P	$::=$	$P :: p$	*Path nesting*
		$P \triangleleft_{mn} P$	*Passive (or preemptive) composition*

The priority level is given by the following list (binding from the higher to the lower priority): $\alpha_p.P < (P :: p) < (P \triangleleft P) < (P \mid P) < (P + P) < (recX : P) < (P \setminus L : M), P[f : g])$. Thus, $\alpha.P_1 + P_2 \mid P_3 \triangleleft P_4$ means $((\alpha.P_1) + P_2) \mid (P_3 \triangleleft P_4)$.

3.3 Transitional semantics

The transition system is based on the idea of actions and co-actions on the one hand, and the notion of *path* on the other hand. Paths can be viewed as the concatenation of sub-paths which are able to reflect hierarchical structures. *Selectors* are paths associated to composition operators which refine the potential interactions between components. Indeed, the rules that define the evolution of the system use a matching function over paths and selectors in order to evaluate the potential interactions. In CCS, if a process P can interact with a process Q, the expression $(P_1|Q_1|P_2|Q_2)$ allows P_1 to interact either with Q_1 or with Q_2, and also the same for P_2. Within our calculus, the expression $(P_1 \mid_{\circ\circ} Q_2) \mid_{**} (P_2 \mid_{\circ\circ} Q_1)$ prevents all interactions between P_1, Q_2 and P_2, Q_1, because the selector $\circ$ can't match any path while $*$ matches all paths. The following definitions make this precise.

3.3.1 The labeled transition system

The transition system $\xrightarrow{\alpha:p} \subseteq \mathcal{P} \times \mathcal{P}$ is defined over the set $\mathcal{P}$ of ICCS terms, $(\alpha, p) \in Act \times \mathcal{S}$; *Act* is the set of communication actions and $\mathcal{S}$ the set of *paths*.

Communication actions α. $\alpha \in \text{Act} = \mathcal{L} \cup \{\tau\}, \mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$. Here $\mathcal{A} = \{\text{a,b,c,}\dots\}$ is a set of names, $\bar{\mathcal{A}} = \{\bar{a}, \bar{b}, \bar{c}, \dots\}$ is a set of co-names, and $^-$ is a bijection over $\mathcal{A}$ and $\bar{\mathcal{A}}$ such that $\bar{\bar{a}} = a$. As in Milner's CCS, $\mathcal{A}$ represents input communication actions and $\bar{\mathcal{A}}$: output communication actions. τ is a distinguished label that represents internal (unobservable) actions, possibly generated by "hand-shaking" among components.

Paths. Paths, usually written p or q are terms defined by the grammar (1) or are equal to $\circ$. The symbol $\circ$ is used as the void item of a matching function over $\mathcal{S}$.

$$S ::= p_i S \mid * S \mid ?S \mid \varepsilon \tag{1}$$

p_i range over an alphabet $P_s = \{a, b, \dots\}$. For example a, abc, $a?c$, $*c$, ε are all in $\mathcal{S}$.

Definition 1 *Concatenation (+)*
$+ : \mathcal{S}^2 \longrightarrow \mathcal{S}$ is such that whatever $s_1, s_2 \in \mathcal{S} - \{\circ\}$, $s_1 + \varepsilon = \varepsilon + s_1 = s_1$ and $s_1 + s_2 = s_1 s_2$. $\mathcal{S}$ is made stable for $+$ by: $s_1 + \circ = \circ + s_1 = \circ = \circ + \circ$
$p_1 + p_2$ is written $p_1 p_2$ for convenience.

Definition 2 *Matching function over S*
We define a matching function $\doteq$: $\mathcal{S} \times \mathcal{S} \longrightarrow \{true, false\}$, by using the table:

$\doteq$	$p_2 S_2$	$*S_2$	ε	$\circ$
$p_1 S_1$	$(p_1{=}p_2 \vee p_1{=}? \vee p_2{=}?) \wedge (S_1 \doteq S_2)$	$(p_1 S_1 \doteq S_2) \vee (S_1 \doteq *S_2)$	$false$	$false$
$*S_1$	$(S_1 \doteq p_2 S_2) \vee (*S_1 \doteq S_2)$	$(S_1 \doteq S_2) \vee (*S_1 \doteq S_2) \vee (S_1 \doteq *S_2)$	$S_1 \doteq \varepsilon$	$false$
ε	$false$	$\varepsilon \doteq S_2$	$true$	$false$
$\circ$	$false$	$false$	$false$	$false$

This definition yields a natural matching function such that $a{*}d \doteq abcd$, $a?c? \doteq abcd$ or even $a{*}d \doteq a{*}?d$, $a?c? \doteq a{*}$
Note that $*^n$, $(n \geq 1)$ matches everything except $\circ$ and that $\circ$ matches nothing.

3.3.2 Inference rules

$$\frac{true}{\alpha_p.P \xrightarrow{\alpha:p} P} \quad \text{[Prefix]} \tag{2}$$

It is the fundamental rule that expresses the sequential behavior of components: $\alpha_p.P$ performs the communication $\alpha : p$ and then behaves like P. In the rest of the paper, α_ε is written α for convenience.

$$\frac{P \xrightarrow{\alpha:p} P'}{P + Q \xrightarrow{\alpha:p} P'}\text{[Alt1]} \qquad \frac{Q \xrightarrow{\beta:q} Q'}{P + Q \xrightarrow{\beta:q} Q'}\text{[Alt2]}$$

The rules [Alt1] and [Alt2] describe the alternative, which can be indeterministic when both P and Q are able to perform transitions (the choice can be either the left operand or the right one).

$$\frac{P \xrightarrow{\alpha:p} P'}{P :: q \xrightarrow{\alpha:pq} P' :: q}\text{[Nesting]} \qquad \frac{P \xrightarrow{\alpha:p} P'}{P[f:g] \xrightarrow{f(\alpha,p):g(\alpha,p)} P'[f:g]}\text{[Relab]}$$

[Nesting] allows expansion by concatenation of the path associated to the initial transition. This can be seen as a way of coding some structural information into the communication itself. [Relab] is designed for allowing the relabeling of both action labels and paths. Note that the function f is such that f :*Act*$\times\mathcal{S} \longrightarrow$*Act* and g is such that g :*Act*$\times\mathcal{S} \longrightarrow \mathcal{S}$.

$$\frac{P \xrightarrow{\alpha:p} P' \ , \ \alpha \notin L \cup \bar{L}, p \notin M}{P \setminus L:M \xrightarrow{\alpha:p} P' \setminus L:M}\text{[Restr]} \qquad \frac{P\{recX : P \ / \ X\} \xrightarrow{\alpha:p} P'}{recX : P \xrightarrow{\alpha:p} recX : P'}\text{[Rec]}$$

[Restr] is the restriction operator, very similar to the CCS one, except that it can use a set of paths. It forces internal communication by forbidding any communication actions (L and M specify the forbidden sets), excepted τ, which is actually an unobservable action. It is also the way of forcing synchronization among components. Restrictions of the form $P \setminus \{a, b, \dots\} : \emptyset$ will sometimes be written $P \setminus \{a, b, \dots\}$ for simplification.

[Rec] is a recursive operator such that $recX : P$ behaves like $recX : P'$ if $P\{(recX : P)/X\}$ becomes P' after the transition. The notation $A\{x/y\}$ defines a component built by substituting all occurrences of y in A by x (for instance, if $A \equiv a.b.X$, then $A\{Y/X\} \equiv a.b.Y$).[1]

The concurrent composition (with selective interaction) is defined toward [Par1], [Par2] and [Par3]:

$$\frac{P \xrightarrow{\alpha:p} P'}{P \mid_{mn} Q \xrightarrow{\alpha:p} P' \mid_{mn} Q}\text{[Par1]} \qquad \frac{Q \xrightarrow{\beta:q} Q'}{P \mid_{mn} Q \xrightarrow{\beta:q} P \mid_{mn} Q'}\text{[Par2]}$$

$$\frac{P \xrightarrow{\alpha:p} P' \ , \ Q \xrightarrow{\bar{\alpha}:q} Q' \ , \ p \doteq m \ , \ q \doteq n}{P \mid_{mn} Q \xrightarrow{\tau:\varepsilon} P' \mid_{mn} Q'}\quad\text{[Par3]} \tag{3}$$

The following equations define the (selective) preemptive operator. Note that in [Preemp1b], [Preemp2b] and [Preemp3] a, b and $\bar{a}$ (respectively $\in \mathcal{A}$ and $\bar{\mathcal{A}}$) are used instead of $\alpha, \bar{\alpha}$ (both in *Act*), because only matching co-labels emitted by the left operand (and also meeting matching labels from the right operand) can trigger the transition. The $\lhd_{mn}$ notation is used for making explicit this notion of orientation. Intuitively, this expresses that outgoing calls and returns are able to transfer the activity flow.

$$\frac{P \xrightarrow{\alpha:p} P' \ , \ p \neq m \text{ or } \alpha = \tau}{P \lhd_{mn} Q \xrightarrow{\alpha:p} P' \lhd_{mn} Q}\text{[Preemp1a]} \quad \frac{P \xrightarrow{\bar{a}:p} P' \ , \ p \doteq m, \bar{a} \in \bar{\mathcal{A}}}{P \lhd_{mn} Q \xrightarrow{\bar{a}:p} Q \lhd_{nm} P'}\text{[Preemp1b]}$$

$$\frac{Q \xrightarrow{\beta:q} Q' \ , \ q \neq n}{P \lhd_{mn} Q \xrightarrow{\beta:q} P \lhd_{mn} Q'}\text{[Preemp2a]} \qquad \frac{Q \xrightarrow{b:q} Q' \ , \ q \doteq n, b \in \mathcal{A}}{P \lhd_{mn} Q \xrightarrow{b:q} Q' \lhd_{nm} P}\text{[Preemp2b]}$$

[1] We consider in our language only guarded recursions, i.e recursions $recX : P$ were any occurrence of X in P is within some sub-expression $\alpha.X$ ($recX : X + E$ is an example of unguarded recursion).

$$\frac{P \xrightarrow{\bar{a}:p} P' \ , \ Q \xrightarrow{a:q} Q' \ , \ p \doteq m \ , \ q \doteq n}{P \lhd_{mn} Q \xrightarrow{\tau:\varepsilon} Q' \lhd_{nm} P'} \quad \text{[Preemp3]} \tag{4}$$

Note that in (4), both operands P', Q' and selectors m, n are exchanged after the τ transition. The following derivation illustrates the functioning of the $\lhd$ operator:

$$\begin{aligned}
(\bar{a}.a.0 \lhd_{**} a.\bar{b}.b.\bar{a}.0) \setminus \{a\} &\xrightarrow{\tau:\varepsilon} (\bar{b}.b.\bar{a}.0 \lhd_{**} a.0) \setminus \{a\} && \text{[Preemp3]} \\
&\xrightarrow{\bar{b}:\varepsilon} (a.0 \lhd_{**} b.\bar{a}.0) \setminus \{a\} && \text{[Preemp1b]} \\
&\xrightarrow{b:\varepsilon} (\bar{a}.0 \lhd_{**} a.0) \setminus \{a\} && \text{[Preemp2b]} \\
&\xrightarrow{\tau:\varepsilon} (0 \lhd_{**} 0) \setminus \{a\} && \text{[Preemp3]}
\end{aligned}$$

The reader shall consider the following derivation produced by preemptive composition of a component which is not passive (P):

$$\begin{aligned}
(\bar{a}.a.0 \lhd_{**} a.\bar{b}.\bar{c}.b.\bar{a}.0) \setminus \{a\} &\xrightarrow{\tau:\varepsilon} (\bar{b}.\bar{c}.b.\bar{a}.0 \lhd_{**} a.0) \setminus \{a\} && \text{[Preemp3]} \\
&\xrightarrow{\bar{b}:\varepsilon} (a.0 \lhd_{**} \bar{c}.b.\bar{a}.0) \setminus \{a\} && \text{[Preemp1b]}
\end{aligned}$$

The derivation ends at this point. The composition $(a.\bar{b}.\bar{c}.b.\bar{a}.0 \mid \bar{a}.a.0) \setminus \{a\}$ would produce a different (successful, and thus not representative) derivation tree:
$\xrightarrow{\tau:\varepsilon}\xrightarrow{\bar{b}:\varepsilon}\xrightarrow{\bar{c}:\varepsilon}\xrightarrow{b:\varepsilon}\xrightarrow{\tau:\varepsilon}$.

3.4 Bisimulations and observation equivalences

The definition of observation equivalences is very important for characterizing components. For a composite component, it establishes the validity of a specification (obtained by composition of subcomponents) with respect to the behavorial description of its interface. It also opens perspectives for reusing existing components by finding equivalent components in a repository indexed by interface descriptions. This section shows that the labeled transition system of ICCS allows us to define the notions of strong and weak bisimulation, thus allowing the same definitions of observation equivalences as in CCS. Moreover, it proposes more specific bisimulations able to capture the features of ICCS, related to selective interactions.

Definition 3 *Derivatives.*
Let $t = (\alpha_1 : p_1) \cdots (\alpha_n : p_n) \in Act^*$. *Then*
(1) $\xrightarrow{t} \stackrel{def}{=} \xrightarrow{\alpha_1:p_1} \cdots \xrightarrow{\alpha_n:p_n}$;
(2) $\hat{t} \in Act^*$ *is the result of removing all* $(\tau : p_i)$*'s from* t;
(3) $\stackrel{t}{\Longrightarrow} \stackrel{def}{=} (\xrightarrow{\tau:p_i})^* \xrightarrow{\alpha_1:p_1} (\xrightarrow{\tau:p_j})^* \cdots \xrightarrow{\alpha_n:p_n} (\xrightarrow{\tau:p_k})^*$.

The first bisimulation is the *strong bisimulation*, that allows to state $a.(b.0 + b.0) \sim a.b.0 + a.b.0$, for instance, but also $a.\tau.b.0 \not\sim a.b.0$. It is the strongest equivalence proposed.

Definition 4 *Strong bisimulation.*
A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a strong bisimulation if $(P_1, P_2) \in \mathcal{R} \Rightarrow \forall(\alpha, p) \in Act \times \mathcal{S}$:
(1) Whenever $P_1 \xrightarrow{\alpha:p} P_1'$ *then, for some* P_2', $P_2 \xrightarrow{\alpha:p} P_2'$ *and* $(P_1', P_2') \in \mathcal{R}$
(2) Whenever $P_2 \xrightarrow{\alpha:p} P_2'$ *then, for some* P_1', $P_1 \xrightarrow{\alpha:p} P_1'$ *and* $(P_1', P_2') \in \mathcal{R}$

Definition 5 *Strong equivalence.*
P_1 and P_2 are strongly equivalent, written $P_1 \sim P_2$ if $(P_1, P_2) \in \mathcal{R}$ for some strong bisimulation $\mathcal{R}$. This can be expressed as:

$$\sim = \bigcup\{\mathcal{R} : \mathcal{R} \textit{ is a strong bisimulation }\}$$

Definition 6 *Weak bisimulation.*
A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a weak bisimulation if $(P_1, P_2) \in \mathcal{P} \Rightarrow \forall(\alpha, p) \in Act \times \mathcal{S}$:
(1) Whenever $P_1 \xrightarrow{\alpha:p} P_1'$ *then, for some* P_2', $P_2 \stackrel{\hat{\alpha}:p}{\Longrightarrow} P_2'$ *and* $(P_1', P_2') \in \mathcal{R}$
(2) Whenever $P_2 \xrightarrow{\alpha:p} P_2'$ *then, for some* P_1', $P_1 \stackrel{\hat{\alpha}:p}{\Longrightarrow} P_1' and (P_1', P_2') \in \mathcal{R}$

Definition 7 *Weak equivalence.*
P_1 is weakly equivalent to P_2 , written $P_1 \approx P_2$ if $(P_1, P_2) \in \mathcal{R}$ for some weak simulation $\mathcal{R}$. This can be expressed as:

$$\approx = \bigcup\{\mathcal{R} : \mathcal{R} \textit{ is a weak bisimulation }\}$$

The weak equivalence equates processes by ignoring τ actions. For instance, $\tau.a.b.0 + c.\tau.0 \approx (a.\tau.b.0 + c.0)$. But $\approx$ is not substitutive for $+$: $b.0 \approx \tau.b.0$, but $a.0 + b.0 \not\approx a.0 + \tau.b.0$.

As for CCS, the observation equality is the better equivalence of ICCS, because it equates more components than $\sim$, and because it is a congruence relation, i.e substitutive for all operators , $\lhd$ excepted (for instance $P = Q \Rightarrow (P \mid_{mn} R) = (Q \mid_{mn} R)$). In fact, for the operator $\lhd$, $=$ is not substitutive in the general case, due to the different way of processing matching τ actions at the left and the right side: $a.\bar{b}.\tau.b.\bar{a}.0 = a.\bar{b}.b.\bar{a}$, but $(0 \lhd a.\bar{b}.\tau.b.\bar{a}.0) \neq (0 \lhd a.\bar{b}.b.\bar{a})$. The full substitution is verified only if both components are in the same category, i.e *passive* or not *passive*. In the sub-section 3.7, definition (13) and proposition (3) will give the formal definition of *passive* components.

Definition 8 *Observation equality.*
P_1 and P_2 are equal, or observation congruent, written $P_1 = P_2$ if $\Rightarrow \forall(\alpha, p) \in Act \times \mathcal{S}$:
(1) Whenever $P_1 \xrightarrow{\alpha:p} P_1'$ *then, for some* P_2', $P_2 \stackrel{\alpha:p}{\Longrightarrow} P_2'$ *and* $P_1' \approx P_2'$
(2) Whenever $P_2 \xrightarrow{\alpha:p} P_2'$ *then, for some* P_1', $P_1 \stackrel{\alpha:p}{\Longrightarrow} P_1' and P_1' \approx P_2'$

As an example, we have $a.\tau.b.0 \not\sim a.b.0$, but $a.\tau.b.0 = a.b.0$, and $(\tau.a.0 + b.0) \neq (a.0 + b.0)$, but $(\tau.a.0 + b.0) \approx (a.0 + b.0)$.
The important point for equivalence laws is that $(P \sim Q) \Rightarrow (P = Q) \Rightarrow (P \approx Q)$. This is a general result that comes from definitions of the different equivalences. The following bisimulations take into account the richer composition possibilities of ICCS. The reader may consider for instance $A \equiv a.b.0$ and $B \equiv a.(b.0) :: p$, which are such that $A \not\sim B$, but $(A \mid \bar{b}.0) \sim (B \mid \bar{b}.0)$.

Definition 9 *m-selective strong bisimulation.*
A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a m-selective strong bisimulation if $(P_1, P_2) \in \mathcal{R} \Rightarrow \forall(\alpha, p) \in Act \times \mathcal{S}$:
(1) Whenever $P_1 \xrightarrow{\alpha:p} P_1'$ *then, for some* P_2', $P_2 \xrightarrow{\alpha:q} P_2'$, $(p \doteq m \Rightarrow q \doteq m)$ *and* $(P_1', P_2') \in \mathcal{R}$
(2) Whenever $P_2 \xrightarrow{\alpha:q} P_2'$ *then, for some* P_1', $P_1 \xrightarrow{\alpha:p} P_1'$, $(q \doteq m \Rightarrow p \doteq m)$ *and* $(P_1', P_2') \in \mathcal{R}$

The corresponding m-selective equivalences, written $\sim_m$, $\approx_m$ and $=_m$ are defined in a similar way. The example given previously now becomes: $A \sim_* B$ and $(A \mid_{**} \bar{b}.0) \sim (B \mid_{**} \bar{b}.0)$. A law that generalizes this proposition will be proposed at the end of the next sub-section.

3.5 Equational laws.

We outline hereafter some equational laws, obtained by establishing strong bisimilarity. Other laws could be produced by specifying conditions on selectors and paths. Basic laws established for CCS are reusable due to the similarity of the transition system. In fact, the rule [Preemp3] shows that $P \mid_{**} Q$ is semantically equivalent to the original operator $P|Q$ of CCS (as long that $\varphi(P) \neq \{\circ\}$ and $\varphi(Q) \neq \{\circ\}$). This means that, in association with the semantic definition of the restriction [Restr] and the relabeling [Relab], ICCS can be viewed as a superset of CCS (however, some laws might be restricted by the path condition).

Definition 10 *Sorts.*
The (syntactic) sort $\phi(P) \subseteq \mathcal{L}$ of a component P, is a set defined recursively over the syntactic structure of P:

$\phi(\alpha_p.P) = \phi(P) \cup \{\alpha\} - \{\tau\}$
$\phi(P_1 \mid_{mn} P_2) = \phi(P_1) \cup \phi(P_2)$
$\phi(P[f:g]) = f(\phi(P))$
$\phi(P :: p) = \phi(P)$
$\phi(P_1 + P_2) = \phi(P_1) \cup \phi(P_2)$
$\phi(P \setminus L:M) = \phi(P) - L \cup \bar{L}$
$\phi(recX : P) = \phi(P)$
$\phi(P_1 \lhd_{mn} P_2) = \phi(P_1) \cup \phi(P_2)$

For example, $\phi(a.\bar{c}.\tau.0 + b.0) = \{a, b, \bar{c}\}$.

Definition 11 *Path sorts.*
The (syntactic) path sort $\varphi(P) \subseteq \mathcal{S}$ of a component P, is the set built over P from the following definition:

$\varphi(0) = \emptyset$
$\varphi(\alpha_p.P) = \{p\} \cup \varphi(P)$
$\varphi(P_1 \mid_{mn} P_2) = \varphi(P_1) \cup \varphi(P_2)$
$\varphi(P[f:g]) = g(\phi(P) \times \varphi(P))$
$\varphi(P :: p) = \oplus(p, \varphi(P))$
$\varphi(X) = \emptyset$
$\varphi(P_1 + P_2) = \varphi(P_1) \cup \varphi(P_2)$
$\varphi(P \setminus L:M) = \varphi(P) - M$
$\varphi(recX : P) = \varphi(P)$
$\varphi(P_1 \lhd_{mn} P_2) = \varphi(P_1) \cup \varphi(P_2)$

Here, $\oplus$ is the function:
$\oplus : \mathcal{S} \times \{p_i \in \mathcal{S}\} \rightarrow \{q_i \in \mathcal{S}\}$ such that $\forall s \in \mathcal{S}$, $\oplus(s, p_i) = q_i = p_i + s$

For example, if $P \equiv a.(b.\tau.0 : p_1)$, then $\varphi(P :: p_2) = \{p_2, p_1\, p_2\}$.

Definition 12 *Path quotient.*
The quotient of a path sort $\varphi(P)$ by a path q, written $\varphi(P)/q$ is the set $\{p_i \in \varphi(P) : p_i \doteq q\}$

Proposition 1 *Laws and relations for $|_{mn}$, $\lhd_{mn}$, $\sim$ and $\sim_k$.*
We shall use:
(a) $\|$ *instead of* $|_{\circ\circ}$ *and* $\lhd_{\circ\circ}$; **(b)** $|$ *instead of* $|_{**}$; **(c)** $\lhd$ *instead of* $\lhd_{**}$.

monoïd laws	
1. $(P+Q) \sim (Q+P)$	**2.** $((P+Q)+R) \sim (P+(Q+R))$
3. $(P+P) \sim P$	**4.** $(P+0) \sim P$
static laws	
1. $(P \mid_{m\circ} Q) \sim (P \mid_{\circ m} Q) \sim (P \lhd_{\circ m} Q) \sim (P \lhd_{m\circ} Q) \sim (P \lhd_{\circ\circ} Q) \sim (P \mid_{\circ\circ} Q)$	
$\forall m, n, m', n' \in \mathcal{S}$	**2.** $P \mid_{mn} Q \sim Q \mid_{nm} P$
3. $P \mid_{mn} (Q \mid_{m'n'} R) \sim (P \mid_{nm} Q) \mid_{m'n'} R)$	**4.** $P \mid_{mn} 0 \sim P$
5. $P \setminus L{:}M \sim P$ if $\phi(P) \cap (L \cup \bar{L}) = \emptyset$ and $\varphi(P) \cap M = \emptyset$	
6. $P \setminus L{:}M \sim 0$ if $\phi(P) \cap (L \cup \bar{L}) = \phi(P)$ or $\varphi(P) \cap M = \varphi(P)$	
7. $P \setminus L{:}M \setminus K{:}N \sim P \setminus L \cup K{:}M \cup N$	
8. $(P \mid_{mn} Q) \setminus L{:}M \sim (P \setminus L{:}M) \mid_{mn} (Q \setminus L{:}M)$	if $\phi(P) \cap \overline{\phi(Q)} \cap (L \cup \bar{L}) = \emptyset$ or $\varphi(P)/m \cap \varphi(Q)/n \cap M = \emptyset$
9. $(P \lhd_{mn} Q) \setminus L{:}M \sim (P \setminus L{:}M) \lhd_{mn} (Q \setminus L{:}M)$	if $\phi(P) \cap \overline{\phi(Q)} \cap (L \cup \bar{L}) = \emptyset$ or $\varphi(P)/m \cap \varphi(Q)/n \cap M = \emptyset$
10. $0 :: p \sim 0$	**11.** $P :: \varepsilon \sim P$
12. $P :: p_1 :: p_2 \sim P :: p_1 p_2$	**13.** $(P+Q) :: p_1 \sim P :: p_1 + Q :: p_1$
14a. $(P \mid_{mn} Q) :: p \sim (P :: p \mid_{mp\ np} Q :: p)[(\tau/\tau : \varepsilon) : (p/\tau : \varepsilon)]$	
14b. $(P \lhd_{mn} Q) :: p \sim (P :: p \lhd_{mp\ np} Q :: p)[(\tau/\tau : \varepsilon) : (p/\tau : \varepsilon)]$	
15. $\varphi(P)/\circ = \emptyset$	**16.** $\varphi(P :: \circ)/* = \emptyset$
17. $\varphi(P :: \circ) = \{\circ\}$	**18.** $0[f:g] \sim 0$
19. $P[f:g][f':g'] \sim P[f' \circ (f:g) : g' \circ (f:g)]$	
equational laws	
1. $P_1 \lhd_{mn} (P_2 \| P_3) \sim (P_1 \lhd_{mn} P_2) \| P_3$	if $\varphi(P_3)/n = \emptyset$
2. $(P_1 \| P_2) \lhd_{mn} P_3) \sim (P_1 \lhd_{mn} P_3) \| P_2$	if $\varphi(P_2)/m = \emptyset$
3. $((P_1 + P_2) \lhd_{mn} P_3) \sim ((P_1 \lhd_{mn} P_3) + P_2)$	if $\varphi(P_2)/m = \emptyset$
4. $(P_1 \mid_{mn} P_2) \sim (P_1 \| P_2)$	if $\varphi(P_1)/m = \emptyset$ or $\varphi(P_2)/n = \emptyset$
5. $(P_1 \mid_{mn} P_2) \sim (P_1 \mid P_2)$	if $\varphi(P_1)/m = \varphi(P_1)$ and $\varphi(P_2)/n = \varphi(P_2)$
6. $(P \| Q) \setminus L : M \sim (P \setminus L : M) \| (Q \setminus L : M)$	**7.** $(P \| Q) :: p \sim (P :: p) \| (Q :: p)$
$\sim$ is a congruence relation	
1. $P \sim Q \Rightarrow (P \mid_{mn} R) \sim (Q \mid_{mn} R)$	**2.** $P \sim Q \Rightarrow (P \lhd_{mn} R) \sim (Q \lhd_{mn} R)$
3. $P \sim Q \Rightarrow (R \lhd_{mn} P) \sim (R \lhd_{mn} Q)$	**4.** $P \sim Q \Rightarrow (P :: p) \sim (Q :: p)$
$\sim_k$ and composition	
1. $P \sim_k Q \Rightarrow (P \mid_{kn} R) \sim (Q \mid_{kn} R)$	**2.** $P \sim_k Q \Rightarrow (P \lhd_{kn} R) \sim (Q \lhd_{kn} R)$
3. $P \sim_k Q \Rightarrow (R \lhd_{mk} P) \sim (R \lhd_{nk} Q)$	
4. $P \sim Q \Rightarrow P \sim_k Q$ for some selector k	**5.** $P \sim_k Q \Rightarrow (P :: p) \sim_{kp} (Q :: p)$

3.6 Expansion Laws

Providing an expansion law, i.e a law that relates static constructors of ICCS, such as [Relab], [Restr], [Preemp] and [Par] to dynamic constructors ([Prefix] and [Alt]) is an important step for establishing an axiomatization of the process equality, as Robin Milner did for CCS restricted to finite state agents [9].

ICCS does not yet provide a notion equivalent to the *concurrent normal form* of CCS. A set of transformation laws could be investigated in order to define a normal form able to include the two composition operators. However, we propose four laws for all static operators, which can be recursively applied to any sequential component (without recursive operators), in order to bring them into standard form (prefix and summation):

Proposition 2 *Expansion laws for* $\lhd_{mn}$, $|_{mn}$, $::$, $[f:g]$.

expansion law for $\lhd_{mn}$	
	If $R \equiv (P \lhd_{mn} Q) \setminus L:M$, then
$R \sim$	$\sum\{a_p.(Q \lhd_{nm} P') \setminus L:M$ if $P \xrightarrow{a:p} P', a \in \bar{\mathcal{A}}, p \doteq m, a \notin \bar{L}, p \notin M\}+$
	$\sum\{b_q.(Q' \lhd_{nm} P) \setminus L:M$ if $Q \xrightarrow{b:q} Q', b \in \mathcal{A}, q \doteq n, b \notin L, q \notin M\}+$
	$\sum\{\tau.(Q' \lhd_{nm} P') \setminus L:M$ if $P \xrightarrow{\bar{a}:p} P', Q \xrightarrow{a:q} Q', a \in \mathcal{A}, p \doteq m, q \doteq n\}+$
	$\sum\{\beta_q.(P \lhd_{mn} Q') \setminus L:M$ if $Q \xrightarrow{\beta:q} Q', q \neq n, \beta \notin L \cap \bar{L}, q \notin M\}+$
	$\sum\{\alpha_p.(P' \lhd_{mn} Q) \setminus L:M$ if $P \xrightarrow{\alpha:p} P', (p \neq m, \alpha \notin L \cap \bar{L}, p \notin M)$ or $(\alpha = \tau)\}$
expansion law for $\lvert_{mn}$	
	If $R \equiv (P \lvert_{mn} Q) \setminus L:M$, then
$R \sim$	$\sum\{\tau.(P' \lvert_{mn} Q') \setminus L:M$ such that $P \xrightarrow{\alpha:p} P', Q \xrightarrow{\bar{\alpha}:q} Q', p \doteq m, q \doteq n\}+$
	$\sum\{\beta_q.(P \lvert_{mn} Q') \setminus L:M$ such that $Q \xrightarrow{\beta:q} Q', \beta \notin L \cap \bar{L}, q \notin M\}+$
	$\sum\{\alpha_p.(P' \lvert_{mn} Q) \setminus L:M$ such that $P \xrightarrow{\alpha:p} P', \alpha \notin L \cap \bar{L}, p \notin M\}$
expansion law for $::$	
	$(P :: q) \sim \sum\{\alpha_{pq}.(P' :: q)$ such that $P \xrightarrow{\alpha:p} P'\}$
expansion law for $[f:g]$	
	$P[f:g] \sim \sum\{f(\alpha,p)_{g(\alpha,p)}.(P'[f:g])$ such that $P \xrightarrow{\alpha:p} P'\}$

Proofs: by developing the derivation graph given by the formal definitions of the operators, and then by exhibiting appropriate strong bisimulations.

As an illustration, these laws allow the following: if $P \equiv \bar{a}.a.0 \lhd a.\bar{c}.(b.\bar{a}.0 + c.\bar{a}.0)$ then $P \setminus \{a\}:\emptyset \sim (\tau.\bar{c}.(b.\tau.(0 \lhd 0) + c.\tau.(0 \lhd 0)))$ (after removing identities in relabeling), and, by applying appropriate equational laws: $P \sim (\tau.\bar{c}.(b.\tau.0 + c.\tau.0))$. The direct application of expansion laws is to put ICCS expressions in full standard form (only prefix and alternation constructors) in order to reduce and compare them on the basis of a few primitive laws.

3.7 More on passive components and passive composition

This sub-section proposes a formal definition of passive components and gives examples that illustrate the concept.

Definition 13 *Passive components (behavorial definition)*
A component $P \in \mathcal{P}$ is passive iff:
for all derivative P' of P such that $P \xrightarrow{a:p} (\xrightarrow{\tau:p_i})^* P'$, then $((\exists P'' s.t. P' \xrightarrow{\bar{b}:q} P'') \wedge (a, b \in \mathcal{A}) \wedge (P''$ is passive$))$.

The following components are also passive: $0, a.\bar{a}.0, (a.\bar{b}.c.\bar{d}.0 + e.\bar{e}.0), a.\tau.recX : (\bar{b}.c.X + \bar{a}.0)$, as are the following: $(a.b.\bar{c}.0|\bar{b}.c.\bar{a}.0) \setminus \{b, c\} : \emptyset$ which can be equated to $a.\tau.\tau.\bar{a}.0$ by using the expansion and static laws. The following components are not passive: $\bar{a}.a.0, a.\bar{b}.\tau.b.\bar{a}.0$. Note that internal actions are allowed only after an input action and before an output action (this corresponds with the presence of the activity flow into the component).

The following proposition relates the preemptive composition operator to passive components:

Proposition 3 *Passive components (equational characterization)*
A component P ($\varphi(P) \neq \{\circ\}$) is passive iff $(0 \triangleleft P) \sim P$

Proof:
$\Longrightarrow$: by showing (without difficulties) that if P is passive, then there exists a strong bisimulation $\mathcal{R}$ such that $P\mathcal{R}(0 \triangleleft P)$ (by using semantic definitions of [Preemp1b-2b-3]). $\Longleftarrow$: by developing all possible transitions of P such that $(0 \triangleleft P) \sim P$, and showing that by induction, the derivation tree complies with the definition (13).

The full modeling of any passive component requires the consideration of reentrance, which allows the modeling of the interleaving of activity flows inside a passive component, and thus concurrency conflicts or ordered schedules toward semaphores.

Consider the following example:
$A \equiv a.\bar{b}.b.\bar{a}.0$ and $B \equiv \bar{a}.a.0$. Then $((B \| B) \triangleleft A) \setminus \{a\} \sim (\tau.\bar{b}.b.\tau.0 + \tau.\bar{b}.b.\tau.0)$, by applying the expansion law. This does not correspond with a concurrent execution of A. Consider instead $A' \equiv recX : a.(\bar{b}.b.\bar{a}.0 \| X)$.

Now, the reader might verify (by developing both derivation trees) that $((B \| B) \triangleleft A) \setminus \{a\} \approx (\bar{b}.b.0 \| \bar{b}.b.0)$, which corresponds to the full activity flow interleaving that was expected. The following definition makes this precise, for components that can be expressed in a standard form $\sum \alpha_i.P_i$

Definition 14 *Reentrant form of passive components*
A passive component $P \equiv \sum \alpha_i.P_i$ is set in reentrant form P_r by:
$P_r \equiv recX : \sum \alpha_i.(P_i \| X)$

Although this definition is required for modeling general Olan components, it raises two problems: the component must be expressed in a standard form, and the recursive definition produces components which are not finite-state. The first hypothesis is acceptable in our context, where all component interfaces can effectively be rewritten in standard form (see section 4). For the second one, Robin Milner demonstrated that the process equality of infinite-state agents is not decidable in the general case. Anyway, some solutions exist for restricted cases: it is easy to show that $(P \sim Q) \Rightarrow (P_r \sim Q_r)$, and thus, assuming the constraint that any reentrant component must be composed of reentrant components, the component equality becomes decidable by using the initial standard form.

4 Applying ICCS to Olan

This section develops tools more specially suited to the Olan framework. The application designer which uses the Olan configuration language needs to specify the behavior of composite components, starting from a behavioral description of reusable sub-components (interfaces). At this level, the interface description needs some level of non-determinism in order to describe a large frame of possibilities: $a.P + \bar{b}.Q$ describes a component able to perform P after receiving a, or to perform the output action $\bar{b}$ and then Q, without providing any information concerning the internal or external mechanisms issuing the decision. As opposed to this descriptive level, the designer is expected to specify the behavior of connectors in a full deterministic way: behavioral variations of the new system must only depend on behavorial variations of sub-systems, otherwise, the designer will increase entropy rather than controlling the new functionalities.

4.1 Weak and strong determinism

This subsection introduces two notions of determinism that will be used respectively for interface description and for connector specifications. The first one, called weak determinism, allows branching as long as it does not occur on the same communication action. This allows branching such as $a.0 + \bar{b}.0$ or $a.\tau.0 + b.0$, but prohibits $a.b.0 + a.c.0$, $\bar{a}.0 + \bar{b}.0$ or $\tau.a.0 + b.0$.

Definition 15 *Weak determinism.*
A component P is weakly deterministic if, for all derivatives P' of P:
$P' \xrightarrow{\alpha:p} Q$ and $P' \xrightarrow{\beta:q} Q'$ implies $(\alpha : p \neq \beta : q$ and $\alpha, \beta \neq \tau)$.

Definition 16 *Strong determinism.*
An component P is strongly deterministic if, for all derivatives P' of P:
$P' \xrightarrow{\alpha:p} Q$ and $P' \xrightarrow{\beta:q} Q'$ implies $(\alpha : p \neq \beta : q)$ and $\alpha, \beta \in \mathcal{A}$.

This means that branching is allowed only for distinct input labels. This specifies components for which outputs are determined by inputs only, and not by internal decisions. It is obvious that if P is strongly deterministic, then P is weakly deterministic.

4.2 Component interfaces

These first describe basic services, synchronous (**provide**, **require**) or asynchronous (**react**, **notify**); synchronous services can accept or raise a set of exceptions. Moreover, provided services can be protected against concurrent accessing toward semaphores of the form $Sem_i \equiv recX : u_i.\bar{u_i}.d_i.\bar{d_i}.X$.

The following table summarizes all possibilities:

services	ICCS form
provide p	$p.\bar{p}.P$
provide p †	$(p.\bar{u_p}.u_p.\bar{p_0}.p_0.\bar{d_p}.d_p.\bar{p}.P_1) \triangleleft (Sem_p \| p_0.\bar{p_0}.P_0)$
provide p ‡	$p.(\bar{p}.P_1 + \sum e\bar{x}_i.P_i)$
provide p †‡	$(p.\bar{u_p}.u_p.\bar{p_0}.(p_0.\bar{d_p}.d_p.\bar{p}.P_1 + \sum(e\bar{x}_i.\bar{d_p}.d_p.P_i))) \triangleleft (Sem_p \| p_0.\bar{p_0}.P_0)$
require r	$\bar{r}.r.P_1$
require r ‡	$\bar{r}.(r.P_1 + \sum ex_i.P_i)$
react r	$r.P$
notify n	$\bar{n}.P$

† : with synchronization for exclusive calls

‡ : with exceptions $\sum ex_i$

Interface descriptions contain a behavior description specified by finite state, weakly deterministic expressions, built over the following sub-language:

$$P ::= 0 \quad | \quad X \quad | \quad \alpha.P \quad | \quad P + P \quad | \quad recX : P$$

where α_i are taken from the set of services defined.

All specifications S can be equated to some $\sum \alpha_i.P_i$ by using expansion laws. This last expression is then translated to $recX : \sum \alpha_i.(P_i \| X)$, if the component is declared as passive and reentrant; for active or passive components, the behavorial specification is translated to $recX : \sum \alpha_i.Q_i$, where $Q_i \equiv P_i\{X/0\}$.

4.3 Connectors

The connector functionalities are specified by finite and strongly deterministic components, either passive or active, of the form $\sum \alpha_i.P_i$. This means that any connector is specified through the transfer of input sequences into output sequences. The following sub-language covers the requirements: $P ::= 0|X|\alpha.P|P + P$.[2] The connector itself is finally defined by $recX : \sum \alpha_i.Q_i$, where $Q_i \equiv P_i\{X/0\}$, for active or passive connectors, and by $recX : \sum \alpha_i.(P_i \| X)$ for passive *and* reentrant connectors.

4.4 Specifi cation correctness

The definition of correctness criteria, i.e. a set of theoretical tools that assert whether an interface specification or a composite implementation is correct or not, is a central issue in architectural description languages ([7], [4], [17]), because it permits taking full advantage of the explicit architectural specifications and to compensate the specification overhead. The following paragraphs detail the main criteria proposed here.

Interfaces In order to describe passive or reentrant components, an interface specification I_{itf} must verify $(0 \triangleleft I_{itf}) \sim I_{itf}$. It must always be weakly deterministic, and this can be checked by a simple syntactic analysis of branching.

[2] The composition operators might be usable

Connectors Connectors can be either active or passive, and this can be checked in the same way as interfaces. Connectors must be strongly deterministic, and this can be verified by a static analysis of the syntax tree (branching).

Composite implementation This is the most important point. Here, the problem is to define the conformity of the implementation with respect to the interface specification. The component equality brings a powerful response: For active components, the specification S and the interface I_{itf} must verify $S =_k I_{itf}$ for some selector k such that $\varphi(S)/k \neq \emptyset$. This means that the implementation must present the same behavior as the interface, up to a (significant) selector k. For passive components, it is slightly different, due to the fact that $\lhd$ is not associative: $P \lhd (Q \star R)$ is not equal to $(P \lhd Q) \star R$ in the general case ($\star$ stands for $\lhd$ or $|$). Notably, this means that the specification must use a component variable E_{ext} for representing the external context, and a selector variable to parameter the left interaction. For instance: $((E_{ext} \lhd_{mn} P) \star Q \cdots) \setminus A \cup E_A$ (P, Q are not detailed here), and the effective behavior is given by a system of the form:

$$\begin{cases} (E_{ext} \lhd_{mn} P) \star Q \cdots) \setminus A \cup E_A \\ m =? \\ E_{ext} = R \\ E_A = \{\dots\} \end{cases}$$

In order to check the conformity, the system must be:

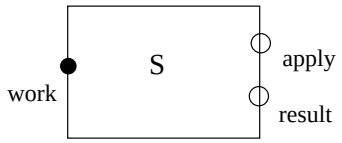
$$S \equiv \begin{cases} ((E_{ext} \lhd_{mn} P) \star Q \cdots) \setminus A \cup E_A \\ m = * \\ E_{ext} = 0 \\ E_A = \emptyset \end{cases}$$

Now, as for active composite components, the equation $S =_k I_{itf}$ must hold for some significant selector k.

This equality is fine, because it covers very different cases. First, as the component equality is fully substitutive, it is ensured that an implementation will behave like its interface through any kind of composition. Secondly, it will check the divergences in the implementation, as well as deadlocks that might be introduced. For instance, if $I_{itf} \equiv a.\bar{b}.b.\bar{a}.0$, then $S \equiv (a.\bar{c}.d.0 \mid \bar{d}.c.\bar{b}.b.\bar{a}.0) \setminus c, d$ (deadlock on c, d) will not equate I_{itf}. Of course, the implementation cannot be correct if the interface is not so. But by minimizing the interface specification expressiveness (no composition), we make things easier in this direction. However, properties of component composition should be studied more deeply, in order to take full benefits of the hypothesis concerning determinism for interfaces and connectors, and in order to make weaker the constraints concerning the composition of reentrant components.

5 Example

This section illustrate how ICCS can be used for modeling an Olan composite component S that performs transformations over data previously stored. This component requires external processing of data units, and delivers results to any external participant connected to it.



The expected behavior of S is expressed in ICCS by:

$$S_{itf} \equiv work.recX : (\overline{apply}.apply.\overline{result}.result.X + \overline{work}.0)$$

This specification becomes:

$$S_{itf} \equiv recX_s : work.(recX : \overline{apply}.apply.\overline{result}.result.X + \overline{work}.X_s)$$

5.1 The primitive components

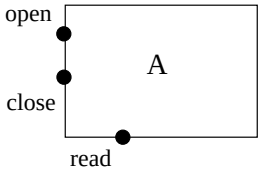


Figure 3: file handler A

A file handler A (fig. 3) The first (passive) sub-component A is a file handler that offers standard basic services: *open,close, read.*
The interface which specifies the correct behavior is first specified by (5), and after transformation, by (6):

$$A_{itf} \equiv open.(\overline{open}.recX : (read.\overline{read}.X + close.\overline{close}.0)) \quad (5)$$

$$A_{itf} \equiv recX_A : open.\overline{open}.recX : (read.\overline{read}.X + close.\overline{close}.X_A) \quad (6)$$

This behavorial description verifies $0 \lhd A_{itf} \sim A_{itf}$.

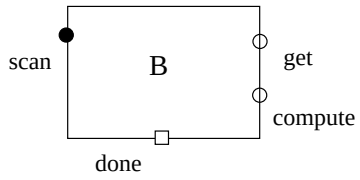


Figure 4: Scanner B

A scanner B (fig. 4) The second component is a scanner that performs successive data inputs toward a required service *get*. Each successful input is followed by a computation request *compute*, then by the notification of the result toward the *done* port. If the return of the *get* call is not valid, then the processing ends and scan returns. This behavior is formally specified by (7) and after transformation (8):

$$B_{itf} \equiv scan.recY : \overline{get}.get.(\overline{compute}.compute.\overline{done}.Y + \overline{scan}.0) \quad (7)$$

$$B_{itf} \equiv recX_B : scan.recY : \overline{get}.get.(\overline{compute}.compute.\overline{done}.Y + \overline{scan}.X_B) \quad (8)$$

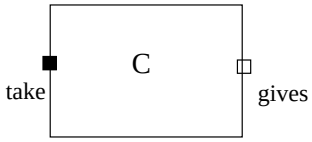


Figure 5: data-flow converter C

A data-flow converter C (fig. 5) Finally, the sub-component C performs data conversion on an input/output data flow through *take* and *gives* ports. The behavior of C is expressed by (9) and becomes (10):

$$C_{itf} \equiv take.\overline{gives}.0 \quad (9)$$

$$C_{itf} \equiv recX : take.\overline{gives}.X \quad (10)$$

5.2 The connectors

The specification of the composite component S uses a set of connectors in order to provide the new functionalities:

$$\begin{aligned}
c_0 &\equiv in_0.\overline{out_0}.out_0.\overline{out_1}.out_1.\overline{out_2}.out_2.\overline{out_3}.out_3.\overline{in_0}.0\\
c_1 &\equiv in_0.\overline{out_0}.out_0.\overline{out_1}.out_1.\overline{in_0}.0\\
c_2 &\equiv (in_0.\overline{out_0}.out_0.\overline{in_0}.in_1.\overline{out_1}.out_1.0 + in_2.\overline{in_2}.0)\\
c_3 &\equiv in_0.\overline{out_0}.0
\end{aligned}$$

After transformation, it becomes:

$$\begin{aligned}
c_0 &\equiv recX : in_0.\overline{out_0}.out_0.\overline{out_1}.out_1.\overline{out_2}.out_2.\overline{out_3}.out_3.\overline{in_0}.X\\
c_1 &\equiv recX : in_0.\overline{out_0}.out_0.\overline{out_1}.out_1.\overline{in_0}.X\\
c_2 &\equiv recX : (in_0.\overline{out_0}.out_0.\overline{in_0}.in_1.\overline{out_1}.out_1.X + in_2.\overline{in_2}.X\\
c_3 &\equiv recX : in_0.\overline{out_0}.X
\end{aligned}$$

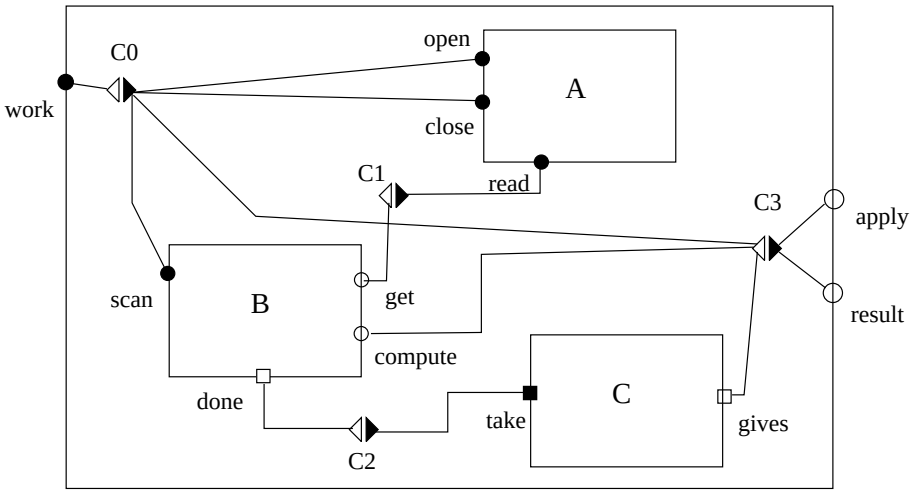
Note that if c_0, c_1, c_2 are "standard" connectors, c_2 is more sophisticated, and performs more than sequencing or message passing: c_2 forces a correct order relationship between out_0 and out_1, performs an adaptation of service types from *notify* into *require* (ports in_1, out_1) and offers the possibility of synchronization by using in_2.
The interconnections between components and connectors are specified through relabeling (I_g is the identity for path):

$$\begin{aligned}
C_0 &\equiv c_0[work/in_0, \overline{work}/\overline{in_0}, open/out_0, \overline{open}/\overline{out_0}, sync/out_3, \overline{sync}/\overline{out_3},\\
&\quad scan/out_1, \overline{scan}/\overline{out_1}, close/out_2, \overline{close}/\overline{out_2}\\
&\quad : a/in_0 : \varepsilon, a/\ in_0 : \varepsilon]\\
C_1 &\equiv c_1[get/in_0, \overline{get}/\overline{in_0}, read/out_0, \overline{read}/\overline{out_0} : I_g]\\
C_2 &\equiv c_2[compute/in_0, \overline{compute}/\overline{in_0}, gives/in_1, sync/in_2, \overline{sync}/\overline{in_2},\\
&\quad apply/out_0, \overline{apply}/\overline{out_0}, result/out_1, \overline{result}/\overline{out_1}\\
&\quad : a/out_0 : \varepsilon, a/\ out_0 : \varepsilon, a/out_1 : \varepsilon, a/\ out_1 : \varepsilon]\\
C_3 &\equiv c_3[done/in_0, \overline{take}/\overline{out_0} : I_g]
\end{aligned}$$

Note that the ports which represent input and output channels with the "outer world" (in_0 of C_0 and out_0, out_1 of C_2 are tagged with the path a, for ensuring the right interaction when composed toward the $\triangleleft$ operator (see the specification of the composite component S below).

5.3 The composite component

The new component, graphically represented by:



has a global behavior expressed by:

$$S_{spec} \equiv ((E_{ext} \lhd_{ma} (C_0 :: b \mid B_{itf} \mid (C_1 :: b \| (C_2 \mid C_3 \mid C_{itf})))) \lhd_{b\varepsilon} A_{itf}) \backslash (\{open, read, scan, done, take, compute, gives, sync\} \cup E_A) : \emptyset$$

Here, E_{ext} is the reference of an external ICCS component potentially connected to S_{spec}. Path a allows selective interaction of E_{ext} with C_0, C_2 and path b selects interaction between C_0, C_1 and the passive sub-component A. Note that the variable name must be bound for developing the derivation graph. The correctness of the specification is expressed by the system:

$$\begin{cases} 0 \lhd S_{itf} \sim S_{itf} & & \text{(passive component)} \\ E_{ext} \equiv 0 & E_A = \emptyset \quad m = * & \\ S_{spec} =_k S_{itf} & & \text{for some selector } k \text{ s.t. } \varphi(S_{spec})/k \neq \emptyset \end{cases}$$

In this case, the system is true for $k = a*$. The figure (5.3) shows the state graphs built from the corresponding derivation trees. The reader might verify, by comparing the two graphs, that the system specification fulfills the correctness criterions. If the synchronization between the connector C_3 and C_0 was omitted, the specification would diverge in some case, when terminating before the invocation of *result*. As an example, the following derivation could occur:
$\xrightarrow{\overline{apply}:a} \xrightarrow{apply:a} (\xrightarrow{\tau p_i})^n \xrightarrow{\overline{work}:ab}$

6 Conclusion and future work

The paper presents a calculus which preserves most of the theoretical contribution of CCS, while providing new possibilities for modeling *passive* components, of which we gave a formal definition (see def. 13).

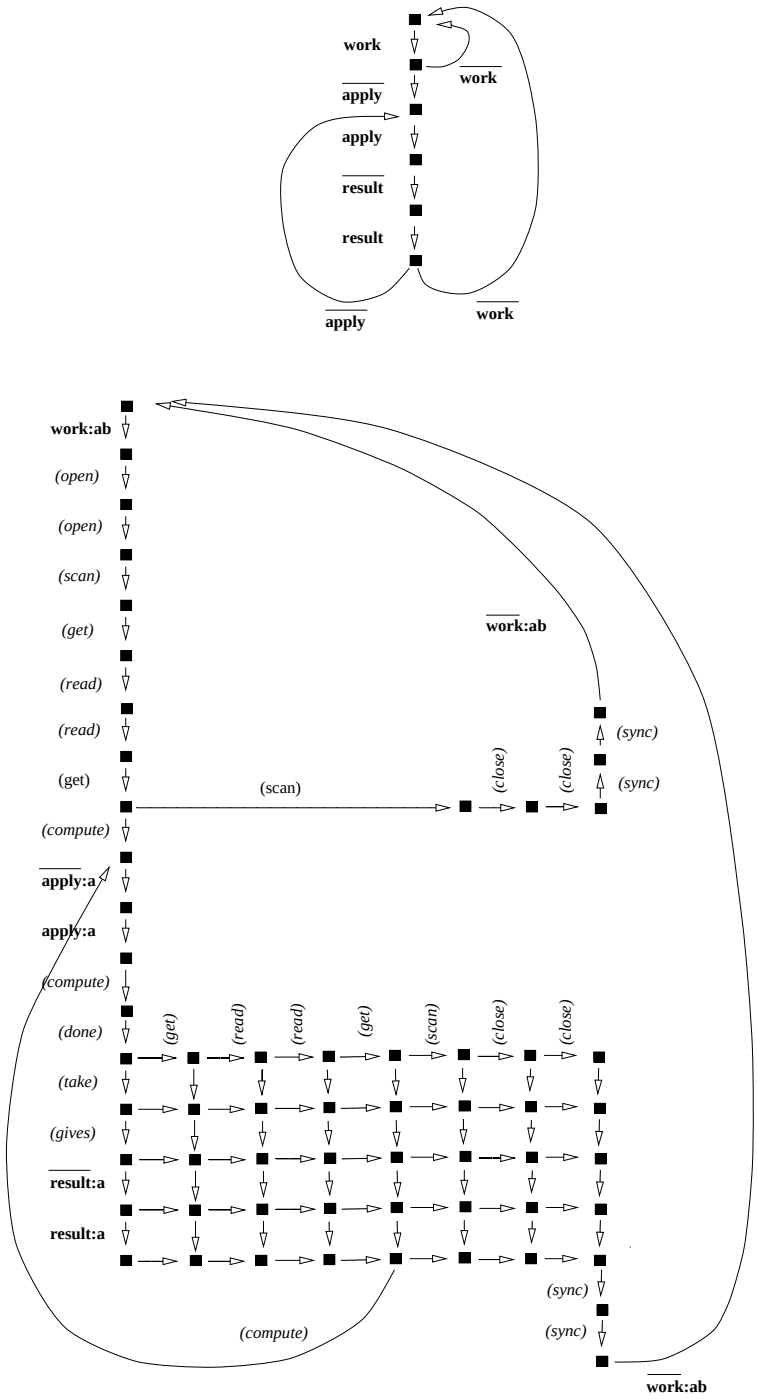


Figure 6: The state graph of S_{itf} and S_{spec} with $E_{ext} \equiv 0$. (Internal communication on port a is written *(a)*)

The difficulty of combining the new operator for passive composition with the standard one is tackled toward the concept of selective interaction, which led as a side effect to refining the expressiveness concerning communication channels. This last issue is important for interconnection languages. Formal developments of section 3 show that the complexity overheads remain at an acceptable cost, and that the theory provides tractable issues. Moreover, selective interactions produce an elegant refinement of the notion of concurrency, making the repertoire of parallel composition richer by providing the $\|$ operator (which semantics expresses parallelism without interaction), and the $|$ operator (which semantics expresses parallelism with full interaction). Both of them appear natural to use within specifications.

Some calculi, such as OC [5], define a composition operator able to perform functional calls as well as remote calls (inter-processes). But the notion of passive composition we propose in this paper is different and more general, allowing the coordination of passive components owning a state, as well as the modeling of functional calls.

Authors such as Yellin and Strom [4] have shown the interest of behavorial specification at the interface level for automating the synthesis of software adaptors. But the formalism presented is not very tractable, and expressiveness of finite-state automatons is poor and probably too verbose for realistic applications. Moreover, this work does not clearly establish the relationship between the finite-state automaton model and the hypothesis concerning execution models.

Robert Allen and David Garlan have proposed in [7] the formalization of architectural interconnections toward ports and connectors, specified by using a subset of CSP [6]. Their work has put the emphasis on connectors which centralize most of the knowledge concerning the interconnection (using *roles* and *glue*), as opposed to the Olan approach, where behavioral specifications are shared between interfaces and connectors. They define compatibility checking toward deadlock free and conservative connectors, i.e. connectors that preserve the deadlock free property toward composition with any specified role. But their correctness criterion seems more complex, less concise and not stronger than ours. Moreover, here again the expressiveness of connectors and role specification is not addressed regarding the basic hypothesis concerning the execution model (connectors and ports are activities).

Magee, Eisenbach and Kramer [8] propose modeling the Darwin configuration language in the π-calculus. Darwin is close to Olan regarding the component model, but proposes few abstractions for capturing the primitive component behaviors. They argue that dynamic aspects of Darwin justify the choice of the π-calculus (which is more complex than CCS), but it is not clear whether dynamic component instantiations might be modeled in CCS or not. But the important point is that by attempting to clearly separate the operational behavior of Darwin architectures from the implementation behaviors, without any information on primitive behaviors, they only define the semantics of communication. Thus no architectural analysis is proposed, and the benefits of the architectural description seem restricted to easing the distribution.

It would be interesting to relate our work to some results in the software engineering field, such as the I-composition of Lam and Shankar [17], and to compare our correctness criteria, based on bisimulation, to their theorem for characterizing the Interface/Implementation correctness by using set oriented operations. However, the calculus developed in this paper seems more adapted to define operational semantics of interconnected systems.

ICCS will be used precisely for defining the operational semantics of the Olan Configuration Language, and for specifying specialized connectors. Moreover, by associating behavorial specifications to the Interface Description Language, it could be applied to a simulation tool (integrated within the Olan development environment), and to the generation of test sets from middle-level specifications. This last issue could take advantage of the modal logic proposed by Robin Milner [9][12], quite applicable to the transition system of ICCS. A small compiler and a simulator have been realized [13] which can be used for exploratory work.

Acknowledgements

the authors are thankfull to all members of the Olan/Sirac Project and its contributors: Roland Balter, Sacha Krakowiak, Michel Riveill, Marie-Claude Pellegrini and Fabienne Boyer. Special thanks are due to Michel Riveill who gave the initial impulsion to this work and followed its evolution, to Irene Maxwell who read various versions of the paper and to Jean-Marc Andréoli for some fruitful exchanges on the passive composition operator.

References

[1] L. Bellissard, S. Ben Atallah, F. Boyer, M. Riveill, "Distributed Application Configuration", *Proc. 16th International Conference on Distributed Computing Systems,* pp. 579-585, IEEE Computer Society, Hong-Kong, May 1996.

[2] Ichiro Satoh and Mario Tokoro, "A Timed Calculus for Distributed Object with Clocks" *Proceedings of the 7th European Conference on Object Oriented Programming,* pp. 326-345, Springer-Verlag, Kaiserslautern, Germany, July 1993.

[3] Ichiro Satoh and Mario Tokoro, "A Formalism for Distributed Real-Time Processes with Temporal Uncertainties", *Proceedings of OOPSLA'92*, ACM, pp.315-326, October 1994.

[4] Daniel M. Yellin and Robert E. Strom "Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors", *Proceedings of OOPSLA'94 (Portland, Oregon USA)*, ACM, pp.176-190, October 1994.

[5] Oscar Nierstrasz, "Toward an Object Calculus", *Proceedings of ECOOP'91 Workshop on Object-Based Concurrent Computing - Geneva*, LNCS, pp 1-20, July 1991.

[6] Hoare C.A.R., "Communicating Sequential Processes", *Communications of ACM*, 21(8), 1978.

[7] Robert Allen and David Garlan, "Formal Connectors", *Technical Report CMU-CS-94-115*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994.

[8] Jef Magee, Susan Eisenbach and Jeff Kramer, "Modelling Darwin in the π-calculus", *Theory & Practice in Distributed systems, Dagstuhl Castle, Germany*, Lecture Notes in computer Sciences 938, September 1994.

[9] Robin Milner, *Communication and Concurrency,* Prentice Hall international, 66 Wood Lane End, Hemel Hempstead Hertfordshire, HP2 4RG UK, 1989.

[10] Robin Milner, "Operational and Algebraic Semantic of Concurrent Processes", *Handbook of Theoretical Computer Science,* Vol. 2, Chapter 19, 1990.

[11] Robin Milner, Joachim Parrow and David Walker, "A Calculus of Mobile Processes (Part 1 & 2)", *LFCS Laboratory for Foundation of Computer Sciences, University of Edinburgh,* June 1989.

[12] Robin Milner, Joachim Parrow and David Walker, "Modal Logics for Mobile Processes", *LFCS Laboratory for Foundation of Computer Sciences, University of Edinburgh,* April 1991.

[13] Jean-Yves Vion-Dury, 'http://pukapuka.inrialpes.fr/ICCS/', *Exploratory Compiler and simulator for ICCS,* January 1997.

[14] De Remer F. and Kron H. "Programming-in-the-large versus Programming-in-the-small", *IEEE Transactions on Software Engineering,* Vol. 2 (No. 2), pp. 80-87, June 1976.

[15] Purtilo J.M. "The POLYLITH Software Bus", *ACM TOPLAS,* Vol. 16 (No. 1), pp. 151-174, January 1994.

[16] Object Management Group. "The Common Object Request Broker: Architecture and Specification", Revision 2.0, July 1995.

[17] Simon S. Lam, A.Udaya Shankar, "A Theory of Interfaces and Modules I-Composition Theorem" *IEEE Transactions on Software Engineering,* Vol. 20, No. 1, January 1994.

[18] Bellissard L., Ben Atallah S., Kerbrat A., and Riveill M. "Comonent-based Programming and Application Management with Olan", *Object-Based Parallel and Distributed Computation France-Japan Workshop, OBPDC'95,* Briot J.P., Geib J.M., Yonezawa A. (Eds.), LNCS, Vol. 1107, Springer-Verlag, 1996.

[19] Magee J., Dulay N. and Kramer J. "A Constructive Development Environment for Parallel and Distributed Programs", *Proceedings of the International Conference on Configurable Distributed Systems,* Pittsburgh, PA, March 1994.



Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399