INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Behavioural Models for Distributed Fractal Components*

Tomás Barros — Rabéa Boulifa — Antonio Cansado — Ludovic Henrio — Eric Madelaine

**N° 6491**

Avril 2008

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6491--FR+ENG

# Behavioural Models for Distributed Fractal Components

Tomás Barros[*] , Rabéa Boulifa[*] , Antonio Cansado[*] , Ludovic Henrio[*] , Eric Madelaine[*]

Thème COM — Systèmes communicants
Équipe-Projet Oasis

Rapport de recherche n° 6491 — Avril 2008 — 25 pages

**Abstract:** This paper presents a formal behavioural specification framework together with its applications in different contexts for specifying and verifying the correct behaviour of distributed Fractal components. Our framework allows us to build behavioural models for applications ranging from sequential Fractal components, to distributed objects, and finally distributed components. Our models are able to characterise both functional and non-functional behaviours, and the interaction between the two concerns.

Finally, this work has resulted in the development of tools allowing the non-expert programmer to specify the behaviour of his components, and automatically, or semi-automatically verify properties of his application.

**Key-words:** Behavioural Models, Transition Systems, Distributed Components, Hierarchical Components, Fractal, Verification, Verification Platform

[*] INRIA Sophia-Antipolis, I3S, Univ. de Nice Sophia-Antipolis, CNRS, France, (first.last)@sophia.inria.fr

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

# Modèles comportementaux pour les systèmes de composants répartis Fractal

**Résumé :** Cet article présente un modèle formel pour la spécification comportementale, ainsi que son application à différents contextes, permettant de spécifier et de vérifier le comportement de composants répartis à la Fractal. Notre méthode permet de construire des modèles comportementaux pour des applications allant des composants Fractal sèquentiels, aux objets répartis, jusqu'aux composants répartis. Nos modèles sont capables de représenter à la fois les comportements fonctionnels et les aspects non-fonctionnels, ainsi que l'interaction entre les deux.

Par ailleurs, ce travail a donné lieu au développement d'outils logiciels permettant à un utilisateur non-expert de spécifier le comportement de ses composants, et de vérifier, automatiquement ou semi-automatiquement, leurs propriètès.

**Mots-clés :** Modèles comportementaux, Systèmes de transitions, Composants répartis, Composants hiérarchiques, Fractal, Vérification, Plateforme de vérification

# I Introduction

Component models provide a structured programming paradigm allowing a better re-usability of programs by the fact that both provided/required services and application structure are expressed statically in the composition. This takes even more importance as the structure of distributed components acts as an abstraction for the component distribution. However, this architectural description is not always sufficient. Indeed, in order to be able to safely compose "off the shelf" or even dynamically discovered components, a form of specification language is required. Such a specification can only rely on the existence of some well defined semantics for the underlying programming language or middleware.

Among the existing component models, *Fractal* [10] provides the following crucial features: the explicit definition of provide/required interfaces for expressing dependencies between components; a hierarchical structure allowing to build components by composition of smaller components; and the definition of non-functional features through specific interfaces, providing a clear separation of concern between functional and non-functional aspects.

Globally, our work is placed in the context of large scale distributed applications. This work is strongly related to programming models that aim at easing the programming of distributed applications by providing high level abstractions of distributed features together with an efficient implementation of these features. More precisely, we rely on the *Grid Component Model (GCM)* [16], which extends Fractal by addressing large scale distributed aspects of components.

Moreover, in distributed context, adaptive components are necessary in order to adapt the application to constantly evolving environments, and evolving requirements in terms of quality of services. Our work is intended to be adapted to the verification of autonomous systems adapting and reconfiguring themselves in order to better match dynamic requirements of the application.

Our main objective is to provide tools to the programmer of distributed components in order to verify the correct behaviour of his program. We require those tools to be intuitive and user-friendly for them to be usable by non-experts of formal methods. To this end we build an analysis toolset, including state-of-the-art model-checking tools; at the heart of this platform lie the model generation tools that are the subject of this article. In this context the choice of the behavioural model is crucial: it has to be compact, expressive enough represent the behavioural semantics, but not too much, to allow an easy mapping to the model-checking input format.

**Related work** Historically, models of behaviours were defined in terms of semantic-level calculi, ranging from core Labelled Transition Systems (LTS), from the very beginning of the process algebra era (see [25, 8]), and the synchronisation vectors of [2], to Milner's $\pi$-calculus [24]. LTS is also, without contest, the most often used model for the representing behaviours in analysis and verification toolsets. At the other end of the spectrum, the $\pi$-calculus has only been used in a few research prototypes, because its high expressivity comes with a very high complexity of most related algorithms.

Naturally, tool developers have tried to add data to the internal models, in order to keep them more compact. For example in the CADP toolbox [19], the internal model is a version of Petri nets with data, that can be later unfolded (eventually on-the-fly) into LTSs suitable for model-checking. Recently a new semantic-level format named NTIF [18], resembling our pLTS, has been devised as a more structured and compact intermediate form between LOTOS or ELOTOS programs and the CADP engines.

Many works have been done based on process algebra foundations, and have led to systems with a more developer-oriented specification language. The FDR2 tool [29] offers a high-level language for expressing CSP models, and an internal machine-readable dialect of CSP [30] using a specific expression language, more adapted to generate the models needed by the verification engines. The LTSA tool [23] uses Finite State Processes (FSP) as an intermediate language (with processes and data parameters) for modelling concurrent Java programs. Another example of research showing goals close to ours make use of Symbolic Transition Systems (STS) [28, 27], that are structures akin to our pNets. In the STSLib toolset, there is a dedicated specification language (with abstract data types) for distributed components, that are modelled by STS, themselves mapped to LOTOS programs that can be model-checked with CADP.

In all these cases, two important questions are: 1) how do you relate the programming language (or specification language) semantics with the internal model, and what properties are preserved by this mapping? 2) how do you transform your (parameterized) internal models into finite structures suitable for analysis (usually LTS)?

**Contribution** This paper tries to answer these questions in the framework of distributed component systems. Toward this challenging perspective, we developed a formal and parametric behavioural model called *pNets*. We have used this formalism to express models for ProActive distributed applications, Fractal components, and GCM distributed components. All our distributed models feature asynchronous calls with futures, which lowers latency while preserving a natural, data-flow oriented synchronisation.

One of the strong original aspects of this work is the focus put on non-functional properties, and the results we provide on the interleaving between functional and non-functional concerns. Thus, the programmer should be able to prove the correct behaviour of his distributed component system in presence of evolution (or reconfiguration) of the system.

**Structure of the paper** In the next section we recall the features of Fractal that are the most relevant to this study, describe the extensions proposed by the GCM model, and sketch the informal semantics of the GCM/ProActive implementation. In Section III we define formally our basic model, named pNets (this formalisation unifies and extends our previous publications in [4, 9, 6, 3]) and recall the main properties of this model. In Section IV we describe the model construction principles for 4 successive kinds of applications, namely active objects, hierarchical components, Fractal components with synchronous controllers, and asynchronous GCM components with controllers. In Section V we present the CoCoME case-study, that will be used to illustrate the rest of the paper. In Section VI we describe the Vercors verification platform, and its application to the case-study, from the input specifications, the model generation phase, to the verification of properties. We conclude with an analysis of perspectives of this work.

# II Context

## II.1 Fractal, GCM and ProActive

The Grid Component Model (GCM) [16] is a novel component model being defined by the european Network of Excellence CoreGrid and implemented by the EU project GridCOMP. The GCM is based on the Fractal Component Model [10], and extends it to address Grid concerns.

From Fractal, GCM inherits a hierarchical structure with strong separation of concerns between functional and non-functional behaviours, including for example life-cycle and binding management. GCM also inherits from Fractal introspection of components and reconfiguration capabilities. Grids consider thousands of computers all over the world, for that, GCM extends Fractal using asynchronous method calls for dealing with latency. Grid applications usually have numerous similar components, so the GCM defines collective interfaces which ease design and implementation of such parallel components by providing synchronisation and distribution capacities. There are two kinds of collective interfaces in the GCM: multicast and gathercast. A client interface may be a multicast interface, meaning that a call toward this interface can be distributed, with its parameters, to many server interfaces. Similarly, a server interface may be a gathercast interface, meaning that multiple client calls will be synchronised and their parameters gathered into a single call that will be performed towards the service component. The GCM also allows control of components to be designed itself in the form of components, and benefit from such a design; moreover, the GCM specifies interfaces for the autonomic management and adaptation of components.

The Architecture Description Language (ADL) of both Fractal and the GCM is an XML-based format, that contains both the structural definition of the system components (subcomponents, interfaces and bindings), and some deployment concerns. Deployment relies on *virtual nodes* (VN) that are an abstraction of the physical infrastructure on which the application will be deployed. The ADL only refers to an abstract architecture, and the mapping between the abstract architecture and a real one is given separately as a deployment descriptor.

## II.2 A GCM Reference Implementation: GCM/ProActive

A GCM reference implementation is based on ProActive [11], an Open Source middleware implementing the ASP calculus [12, 13]. In this implementation, an active object is used to implement each primitive component and each composite membrane. Although composite components do not have functional code themselves, they have a membrane that encapsulates controllers, and dispatches functional calls to inner subcomponents. As a consequence, this implementation also inherits some constraints and properties w.r.t. the programming model:

- components communicate through asynchronous method calls with transparent futures (place-holders for promised replies): a method call on a server interface adds a request to the server *request queue*;
- communication semantics uses a “rendez-vous” ensuring the causal ordering of communications;
- synchronisation between components is ensured with a data-flow synchronisation called *wait-by-necessity*: futures are first order objects that can be forwarded to any component in a non-blocking manner, execution is only blocked if the concrete value of the result is needed (accessed);
- there is no shared memory between components, and a single thread is available for each component.

Each primitive component is associated to an active object written by the programmer, whereas the active object managing a composite is generic and provided by the GCM/ProActive platform. In general, composite components simply forward the functional requests it receives to its subcomponents. Primitive component functionalities are addressed by the encapsulated active object. Most of the time, requests are served in a FIFO order but any *service policy* can be specified when programming active objects (for primitive components), by writing a specific method called `runActivity()`. Note that futures create some kinds of implicit return channels, which are only used to return one value to a component that might need it. One particularity of this approach is that it unifies the concept of component with the unit of distribution and parallelism.

One essential property of GCM/ProActive is that the global behaviour of a component system is totally independent of the physical localisation of components on a distributed architecture.

### II.2.1 Life-cycle of GCM/ProActive components

Like in Fractal, when a component is stopped, only control requests are served. A component is started by invoking the non-functional request: `start()`. For composite components and the primitive components that implement a FIFO policy, as soon as a stop request is encountered, the component can be stopped, and then serves (only) the control requests.

GCM/ProActive implements the membrane of a composite as an active object, thus it contains a unique request queue and a single service thread. The requests to its external server interfaces (including control requests) and from its internal client interfaces are dropped to its request queue. A graphical view of a composite is shown in Fig. 1.

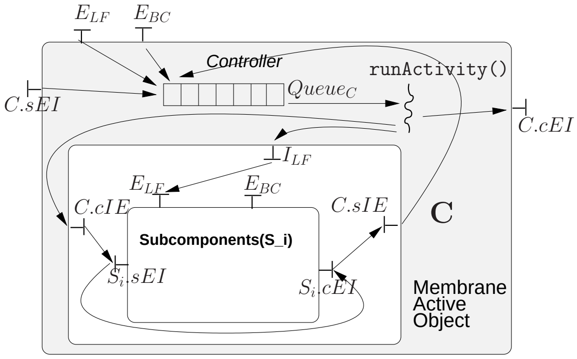


Figure 1: *ProActive* composite component

If a primitive component has a `runActivity()` method, then it is started if and only if it is inside its `runActivity()` method. Since active objects are non-preemptive, the exit from the `runActivity()` method cannot be forced: stop requests are signalled by setting the local variable `isActive` to false; then, the `runActivity()` method should eventually end its execution.

Note that a *stopped* component will not emit functional calls on its required interfaces, even if its subcomponents are active and send requests to its internal interfaces.

# III Theoretical Model

In this section we give the formal definition of our intermediate language that we call *Parameterized Networks of Synchronised Automata (pNets)*. This language is not a new *calculus* in the tradition of theoretical computer science that gave birth to $\lambda$-calculus, $\pi$-calculus, or $\sigma$-calculus, on which we would build new theories or new languages; nor is it a new process algebra endowed with syntax, semantics, and equivalences, that could be used to study new constructs for distributed computing. Rather, pNets give an intermediate and general formalism intended to specify and synchronise the behaviour of a set of automata. We built this model with two goals: give a formal foundation to the model generation principles that we developed for various families of (distributed) component framework; and build a model that would be more machine-oriented, and serve as a versatile internal format for software tools, meaning it must be both expressive (from the universality of synchronised LTSs) and compact (from the conciseness of symbolic graphs).

The synchronisation product introduced by Arnold & Nivat [2] is both simple and powerful, because it directly addresses the core of the problem. One of the main advantages of using its high abstraction level is that almost all parallel operators (or interaction mechanisms) encountered so far in the process algebra literature become particular cases of a very general concept: synchronisation vectors. We structure the synchronisation vectors as parts of a *synchronisation network*. Contrary to synchronisation constraints, the network allows dynamic reconfigurations between different sets of synchronisation vectors through a *transducer* LTS. Our definition of the synchronisation product is semantically equivalent to the one given by Arnold & Nivat.

At a next step, we use Lin's [21] approach for adding parameters in the communications events of both transition systems and synchronisation networks. These communication events can be guarded with conditions on their parameters. Our agents can also be parameterized to encode sets of equivalent agents running in parallel. This leads us to the definition of pNets, that will later appear as being natural models of software systems, because they correspond to the way in which developers specify or program these systems: the system structure is parameterized and described in a finite way (the code is finite), but a specific instance is determined at each execution, or even varies dynamically.

We now give the formal definitions of the model in two steps. We first give our definitions for LTSs, Nets, and synchronisation product; these are equivalent to those found in the literature, but we want this article to be self-contained and with notations coherent with the rest of the model. Then we give the definitions of our parameterized structures (pLTS and pNet), and of their instantiations; their semantics are in terms of standard (infinite) LTS.

**Notations** In the following definitions, we extensively use indexed structures (maps or vectors) over some countable index sets. The indexes will usually be integers, bounded or not. When this is not ambiguous, we shall by abuse use set vocabulary and notations, and typically write "indexed set over J" when formally we should speak of multisets, and still better write "mapping from J to the power set of $\mathcal{A}$".

We use uppercase letters $A, B, I, J, \ldots$ to range over sets, and lowercase letters $a, b, i, j, \ldots$ to range over elements of the sets. We write $\tilde{A}_J$ for an indexed multiset of sets ($\tilde{A}_J =< A_j >_{j\in J}$), and $\tilde{a}_J$ for an indexed multiset of elements ($\tilde{a}_J =< a_j >_{j\in J}$), where $J$ can possibly be infinite. For indexed sets of elements or sets, we say $\tilde{a}_J = \tilde{b}_I \Leftrightarrow J = I \wedge \forall j \in J, a_j = b_j$ (element-wise equality). We write $< a.\tilde{a}_J >$ for the concatenation of an element $a$ at the beginning of an indexed set, $\tilde{x}_J = \tilde{e}_J$ for an indexed set of equations ($< x_j = e_j >_{j\in J}$), $e\{\tilde{x}_J \leftarrow \tilde{e}_J\}$ for the parallel substitution of variables $\tilde{x}_J$ by expressions $\tilde{e}_J$ within expression $e$.

As part of our abusive notation, we extensively, and sometimes implicitly, use the following definition for indexed set membership: $\tilde{a}_J \in \tilde{A}_J \Leftrightarrow \forall j \in J, a_j \in A_j$. Cartesian product is naturally extended to indexed sets so that the following is verified: $a_0 \in A_0 \wedge \tilde{a}_J \in \tilde{A}_J \Rightarrow < a_0.\tilde{a}_J > \in \prod_{j\in\{0\}\cup J} A_j$

We use the usual notions from (typed) term algebras: *operators*, *free variables*, *closed* and *open terms*, etc. Term algebras are endowed with a type system, that include at least a distinguished *Boolean* type and an *Action* type.

## III.1 Networks of Synchronised Automata

We model the behaviour of a process as a Labelled Transition System (**LTS**) in a classical way [25]. The LTS transitions encode the actions that a process can perform in a given state.

**Definition 1 LTS**. *A labelled transition system is a tuple* $(S, s_0, L, \rightarrow)$ *where* $S$ *(possibly infinite) is the set of states,* $s_0 \in S$ *is the initial state,* $L$ *is the set of labels,* $\rightarrow$ *is the set of transitions :* $\rightarrow \subseteq SxLxS$*. We write* $s \overset{\alpha}{\rightarrow} s'$ *for* $(s, \alpha, s') \in \rightarrow$.

We define **Nets** in a form inspired by [2], that are used to synchronise a (potentially infinite) number of processes.

**Definition 2 Network of LTSs**. *Let Act be an action set. A* **Net** *is a tuple* $< A_G, J, \tilde{O}_J, T >$ *where* $A_G \subseteq Act$ *is a set of global actions,* $J$ *is a countable set of argument indexes, each index* $j \in J$ *is called a* hole *and is associated with a* sort $O_j \subset Act$*. The transducer* $T$ *is a LTS* $(S_T, s_{0T}, L_T, \rightarrow_T)$*, and* $L_T = \{\overrightarrow{v} =< a_g.\tilde{\alpha}_I > . a_g \in A_G,\ I \subseteq J \wedge \forall i \in I, \alpha_i \in O_i\}$

**Explanations** Nets describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net. They are *transducers*, in a sense similar to the Lotomaton expressions [20, 26]. A transducer in the Net is encoded as a LTSs which labels are synchronisation vectors ($\overrightarrow{v}$), each describing one particular synchronisation between the actions ($\alpha_I$) of different argument processes, generating a global action $a_g$. Each state of the transducer $T$ corresponds to a given configuration of the network in which a given set of synchronisations is possible. Some of those synchronisations can trigger a change of the transducer's state leading to a new configuration of the network, that is, it encodes a dynamic change on the configuration of the system.

We say that a Net is *static* when its transducer contains only one state. Note that each synchronisation vector can define a synchronisation between one, two or more actions from different arguments of the Net. When the synchronisation vector involves only one argument, its action can occur freely.

**Definition 3** *The* **Sort** *of a system is the set of actions that can be observed from outside the system. It is determined directly by its top-level structure, L for a LTS:* $Sort(S, s_0, L, \rightarrow) = L$*, and* $A_G$ *for a Net:* $Sort(< A_G, J, \tilde{O}_J, T >) = A_G$.

As this is often the case in process algebras, sorts here are determined statically, and are upper approximations of the set of actions that the system can effectively perform. The precision of this approximation depends naturally on the specific model generation procedure, but in most cases an exact computation is not possible.

**Building hierarchical Nets** A Net is a generalised parallel operator. Complex systems are built by combining LTSs in a hierarchical manner using Nets at each level. There is a natural typing compatibility constraint for this construction, in term of the sorts of the formal and actual parameters. The standard compatibility relation is Sort inclusion: a system $Sys$ can be used as an actual argument of a Net at position $j$ only if it agrees with the sort of the hole $O_j$ ($Sort(Sys) \subseteq O_j$). Here also, the compatibility relation may depend on the language or formalism that is modelled; for example if actions represent Java-like method calls, the compatibility could take into account sub-typing.

Our behavioural objects being LTSs, and Nets being operators over LTSs, it is natural to give their semantics in terms of products over LTSs. The definition of the *synchronisation product* below defines the LTS representing any closed Net expression, computed in a bottom-up manner. It would be also possible to define a *symbolic* product over Nets that would reduce any *open* Net expression to a single Net, in the spirit of [20], but this is not necessary for our goals here.

**Definition 4** **Synchronisation Product**. *Given an indexed set* $\tilde{P}_J$ *of LTSs* $\tilde{P}_J = (\tilde{S}_J, \tilde{s}_{0J}, \tilde{L}_J, \tilde{\rightarrow}_J)$*, and a Net* $< A_G, J, \tilde{O}_J, T = (S_T, s_{0_T}, L_T, \rightarrow_T) >$*, such that* $\forall j \in J,\ L_j \subseteq O_j$*, we construct the product LTS* $(S,\ s_0,\ L,\ \rightarrow)$ *where* $S = \prod_{j \in \{T\} \cup J} S_j$*,* $s_0 =< s_{0_T}.\tilde{s}_{0_J} >$*,* $L \subseteq A_G$*, and the transition relation is defined as:*

$$s \xrightarrow{l_t} s' \ \Leftrightarrow\ \begin{pmatrix} s =< s_t.\tilde{s}_J > \wedge\ s' =< s'_t.\tilde{s}'_J > \ \wedge \\ \exists s_t \xrightarrow{<l_t.\tilde{\alpha}_I>} s'_t\ \in \rightarrow_T,\ I \subseteq J\ \wedge\ \forall i \in I, s_i \xrightarrow{\alpha_i} s'_i \in\ \ \rightarrow_i\ \ \wedge\ \forall j \in J \backslash I, s_j = s'_j \end{pmatrix}$$

## III.2 Parameterized Networks of Synchronised Automata

Next we enrich the above definitions with parameters in the spirit of [21]. We start by giving the notion of parameterized actions. We leave unspecified here the constructors and operators of the action algebra, they will be defined together with the mapping of some specific formalism to pNets.

**Definition 5 Parameterized Actions.** *Let $V$ be a set of names, $\mathcal{L}_{A,V}$ a term algebra built over $V$, including the constant action $\tau$. We call $v \in V$ a parameter, and $a \in \mathcal{L}_{A,V}$ a parameterized action, $\mathcal{B}_{A,V}$ the set of boolean expressions (guards) over $\mathcal{L}_{A,V}$.*

**Definition 6 pLTS**. *A parameterized labelled transition system is a tuple $pLTS = (V, S, s_0, L, \rightarrow)$ where:*

- *$V$ is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,V}$,*
- *$S$ is a finite set of states; to each state $s \in S$ is associated a finite indexed set of free variables $fv(s) = \tilde{x}_{J_s} \subseteq V$,*
- *$s_0 \in S$ is the initial state,*
- *$L$ is the set of labels, $\rightarrow$ the transition relation $\rightarrow \subset S \times L \times S$*
- *Labels have the form $l =< \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} >$ such that if $s \xrightarrow{l} s'$, then:*

  *$\alpha$ is a parameterized action, expressing a combination of inputs $iv(\alpha) \subseteq V$ (defining new variables) and outputs $oe(\alpha)$ (using action expressions),*

  *$e_b \in \mathcal{B}_{A,V}$ is the guard,*

  *the variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by expressions $\tilde{e}_{J_{s'}}$*

  *with the constraints: $fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$ and $fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$.*

**Definition 7** *A* **pNet** *is a tuple $< V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T >$ where: $V$ is a set of parameters, $pA_G \subset \mathcal{L}_{A,V}$ is its set of (parameterized) external actions, $J$ is a finite set of holes, each hole $j$ being associated with (at most) a parameter $p_j \in V$ and with a sort $O_j \subset \mathcal{L}_{A,V}$. The transducer $T$ is a LTS $(S_T, s_{0T}, L_T, T_T)$, which transition labels $(\overrightarrow{v} \in L_T)$ are synchronisation vectors of the form: $\overrightarrow{v} =< l_g, \{\alpha_t\}_{i \in I, t \in B_i} >$ such that: $I \subseteq J \wedge B_i \subseteq \mathcal{D}om(p_i) \wedge \alpha_i \in O_i \wedge fv(\alpha_i) \subseteq V$*

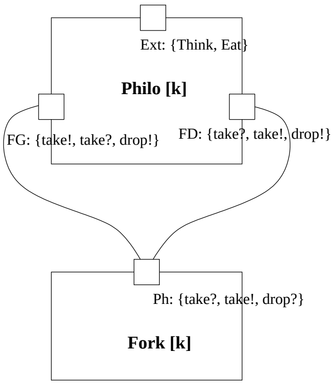
**Explanations** Each hole in the pNet has a parameter $p_j$, expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation of its parameter (we could have, without changing the expressivity, several parameters per hole). In other words, the parameterized holes express *parameterized topologies* of processes synchronised by a given Net. Each parameterized synchronisation vector in the transducer expresses a synchronisation between some instances $(\{t\}_{t \in B_i})$ of some of the pNet holes ($I \subseteq J$). The hole parameters being part of the variables of the action algebra, they can be used in communication and synchronisation between the processes.

A *static* pNet has a unique state, but it has state variables that encode some notion of internal memory that can influence the synchronisation. Static pNets have the nice property that they can be easily represented graphically. We have used them in previous publications to represent them in the Autograph editor [22].

The sorts of our parameterized structures are sets of parameterized actions:

**Definition 8** *Parameterized sorts:*
$Sort(V, S, s_0, L, \rightarrow) = \{\alpha \mid \exists l \in L.\ l = \ < \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} >\}$
$Sort < V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T >= pA_G$

**Example** The drawing in Fig. 2 shows a (static) pNet representing a philosopher problem table, with 2 parameterized holes (indexed by the same variable $k$) for philosophers and forks. On the right side are the corresponding elements of the formal pNet, in which we use the syntax `i[s].a` to denote the action set $\{a_t\}, t \in \{s\}$.


Ext: {Think, Eat}
Philo [k]
FG: {take!, take?, drop!}
FD: {take?, take!, drop!}
Ph: {take?, take!, drop?}
Fork [k]

$PhiloNet = \; < V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T >$ with:
$V = \{k\}$
$pA_G = \{$`Think!k, Eat!k, takeG!k, takeD!k, takeG?k, takeD?k, dropG!k, dropD!k`$\}$
$J = \{$`Philo, Fork`$\}$
$p_{Philo} = k, p_{Fork} = k$
$O_{Philo} = \{$`Ext.Think, Ext.Eat, FD.take!, FG.take!, FD.take?, FD.drop!, FG.drop!`$\}$
$O_{Fork} = \{$`Ph.take?, Ph.take!, Ph.drop?`$\}$
$T$ has a unique state, and transitions with the following labels:
$L_T = \{$

```
<Think!k, Philo[k].Think>
<Eat!k, Philo[k].Eat>
<takeG!k, Philo[k].FG.take!, Fork[k].Ph.take?>
<takeD!k, Philo[k].FD.take!, Fork[k+1].Ph.take?>
<takeG?k, Philo[k].FG.take?, Fork[k].Ph.take!>
<takeD?k, Philo[k].FD.take?, Fork[k+1].Ph.take!>
<dropG!k, Philo[k].FG.drop!, Fork[k].Ph.drop?>
<dropD!k, Philo[k].FD.drop!, Fork[k+1].Ph.drop?> }
```

Figure 2: Example of pNet

**Building hierarchical pNets** Except from the occurrence of parameters in the structure of labels, the rest of the construction of complex systems as hierarchical pNet expressions is similar to the previous section, with the additional parameterization of arguments: an actual (parameterized) argument of a pNet at position $j$ is a pair $< Sys, \mathcal{D} >$, where $Sys$ is a pNet (or pLTS) that agrees with the sort of the hole ($Sort(Sys) \subset O_j$), and $\mathcal{D}$ is the actual domain for the hole parameter $p_j$, i.e. denotes the set of similar arguments inserted in this hole.

We do not define a synchronisation product for pLTS that would give some kind of “early” or “symbolic” semantics of our generalised pNets. Instead, we define instantiations of the parameterized LTS and Nets, based on a (eventually infinite) domain for each variable.

Given a hierarchical pNet expression, and instantiation domains for all parameters in this expression, the definitions below allow us to construct a (non parameterized) Net expression, by applying instantiation separately on each pLTS and each pNet in the expression. This can be performed both for closed or open pNet expressions, the result being, respectively, closed or open Net expressions. In the first case, closed Net expressions can then be reduced to a single LTS (expressing the global behaviour) using the synchronous products in a bottom-up way.

**Definition 9 pLTS Instantiation** *Given a pLTS $P_p =< V, S_p, s_{0_p}, L_p, \rightarrow_p>$, with $V = \tilde{x}_V$ and given a countable domain for each variable $\mathcal{D}_V = \{\mathcal{D}(x)\}_{x \in V}$, and an initial assignment $\rho_0$ for the variables of the initial state $s_{0_p}$, the instantiation $\Phi(P_p, \mathcal{D}_V)$ is a LTS $P =< S, s_0, L, \rightarrow>$*
*such that:*

- $S = \bigcup_{s_p \in S_p} \left\{ s_p\{\tilde{x}_V \leftarrow \tilde{e}_V\} | \forall x \in V, \; \forall e_V \in \mathcal{D}(x) \right\}$,
- $s_0 = s_{0_p}\{fv(s_{0_p}) \leftarrow \rho_0(fv(s_0))\}$,
- *$L$ is the set of ground actions (i.e. closed terms) of the action algebra $\mathcal{L}_{A,V}$,*
- $\rightarrow \; (\subseteq S x L x S) = \bigcup_{t \; \in \rightarrow_p} \Phi(t)$ *is the union of instantiations the of parameterized transitions, built in the following way:*

*let* $t = s \xrightarrow{l_p = <e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}},\alpha>} s'_p$ *be a transition,*

*let* $V_t = fv(s) \cup fv(\alpha) \cup fv(s')$ *the free variables of* $t$*, and* $\mathcal{D}_{V_t}$ *their instantiation domains, then*

$$\Phi(t) = \bigcup_{\tilde{e}_{V_t} \in \mathcal{D}_{V_t}} \left\{ \begin{array}{l} if\ e_b\{\tilde{x}_{V_t} \leftarrow \tilde{e}_{V_t}\} == False\ \ then\ \phi \\ otherwise \\ \ \ let\ \psi = \{\tilde{x}_{V_t} \leftarrow \tilde{e}_{V_t}\} \\ \ \ in\ \left\{ \psi(s) \xrightarrow{\psi(\alpha)} s'\{if\ \exists j \in J_{s'}, x = x_j, then\ x \leftarrow \psi(e_j)^{(*)}\ else\ x \leftarrow e_x\} \right\} \end{array} \right\}$$

Apart from the proliferation of indexes, this definition is quite natural and straightforward; only the case when variables of the target state are assigned during the transition needs care (see (*) in the equation), because the assigned open expressions $\tilde{e}_{J_{s'}}$ need themselves to be instantiated.

This operation has an upper-bound complexity that is exponential in the cardinality of the instantiation domains, in number of states and transitions.

**Definition 10 pNet Instantiation** *Given a pNet* $N_p =< V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T >$*, with the transducer* $T = (S_T, s_{0T}, L_T, T_T)$*, and given domains* $\mathcal{D}_V$ *for variables in* $V$*, the instantiation* $\Phi(N_p, \mathcal{D}_V)$ *is a Net* $N =< A'_G, J', \tilde{O}'_{J'}, T' >$*, with* $T' =< S_{T'}, s_{0T'}, L_{T'}, T_{T'} >$ *constructed in the following way:*

1) *expand the parameterized holes:* $J' = \Phi(J) = \Cup_{j \in J} \mathcal{D}(p_j)$ *where* $\Cup$ *is a disjoint union (or concatenation) of sets; let* $J'_j \subset J'$ *be the part of* $J'$ *corresponding to the expansion of hole number* $j$*;*

2) *instantiate the sort of holes and the global sort:*
   *for* $i \in J'_j$*, build* $\tilde{O}'_i = \bigcup_{a \in O_j} \Phi(a)$
   $A'_G = \bigcup_{a \in pA_G} \Phi(a)$

3) *instantiate the transducer:*

```
$S_{T'} = S_T$
$s_{0T'} = s_{0T}$
$L_{T'} = \bigcup_{\overrightarrow{v} \in L_T} \{\Phi(\overrightarrow{v})\}$ the expansion of the synchronisation vectors:
.  for each $\overrightarrow{v} =< l_g, \{\alpha_{i,t}\}_{i \in I, t \in B_i} >$ let $V = fv(\overrightarrow{v})$, $\mathcal{D}_V$ their instantiation domains,
.     for each possible valuation $\tilde{e}_V$ of the variables in V,
.       let $\phi = \{\tilde{x}_V \leftarrow \tilde{e}_V\}$ the corresponding instantiation function,
.       expand each parameterized action by $\Phi(\alpha_{j,t})$ = if $j \notin I$ then $< *, ..., * >$
.         else $< x_1, ..., x_{|J'_i|} >$, with $x_k = *$ if $k \notin B_i$, $\phi(\alpha_{j,t})$ otherwise,
.       build $\Phi(\phi, \overrightarrow{v})$ as a vector of cardinality $|J'|$ as the concatenation of subvectors
.       $x \in \Phi(\alpha_{j,t})$ for each hole $j \in J$,
.   $\Phi(\overrightarrow{v}) = \{\Phi(\phi, \overrightarrow{v})\}_\phi$
$T_{T'} = \bigcup_{(s, \overrightarrow{v}, s') \in T_T} \{(s, a, s'), a \in \Phi(\overrightarrow{v})\}$
```

Naturally, even if the above definition does not suppose finiteness of the parameter domains, it will be used in practice with finite instantiation domains, and finite vectors.

**Example** Small instantiation of the philosopher system in Fig. 2:

$\Phi(PhiloNet, \mathcal{D}(k) = \{1, 2\}) = < A'_G, J', \tilde{O}'_{J'}, T' >$ with:
$A'_G = \{\texttt{Think!1, Think!2, Eat!1, ...}\}$
$J' = \{\texttt{Philo, Philo, Fork, Fork}\}$
$O\prime_{Philo^{(1)}} = \{\texttt{Ext.Think, Ext.Eat, FD.take!, ...}\}$
$O\prime_{Philo^{(2)}} = \{\texttt{Ext.Think, Ext.Eat, FD.take!, ...}\}$
...
$L_T\prime = \{$

```
<Think!1, Think, *, *, *>
<Think!2, Think, *, *, *>
...
<takeG!1, FG.take!, *, Ph.take?, *>
<takeD!1, FG.take!, *, *, Ph.take?>
...  }
```

**Expressivity** In [4], we gave examples of pNets representing various kinds of recursive functions: The "data flow" within an index family of pLTSs is expressed by an adequate indexing within the synchronisation vectors. A similar construction could be used to show that the model is Turing-expressive.

In practice, and in this paper, we are more interested in expressing specific patterns of parallelism and synchronisation.

## III.3 Data Abstraction

The main interest of the instantiation mechanism defined so far, is the ability to build specific domain instantiations with specific properties. In particular, if the instantiation domains are finite, and are built in such a way that they constitute abstract interpretations of the initial parameter domains, then the instantiated Net is finite. Moreover if parameters were only used as value-passing variables in the original pNet (by contrast with parameters of the system topology), then we can apply a result from Cleaveland and Riely [15] to justify the use of finite model-checking on our instantiated model:

**Property 1** *Let $Sys$ be a (closed) pNet expression, with parameters in $V$, (concrete) parameter domains $\mathcal{D}_V$, and abstract parameter domains $\mathcal{A}_V$, with the following hypotheses:*
*- each $\mathcal{A}_v$ is an abstract interpretation*[1] *of the corresponding concrete domain $\mathcal{D}_v$;*
*- the domains of pNet holes parameters in $Sys$ are unchanged by the abstraction;*
*then the abstraction preserves the specification preorder.*

The *specification preorder* [15], or the better known *testing preorder* [14] are closely related to safety and liveness properties. Given a system and a specification (set of properties), one can build a "most abstract" (finite) value interpretation relatively to the specification, and try to establish its satisfaction. If this succeeds, the result is valid also for the concrete (potentially infinite) system; if it fails, one can select a more concrete (= more values) interpretation and repeat the analysis.

In cases where the instantiated variables are parameters of the system topology, then the previous result does not apply. But the same procedure can be used to build a finite model for one or more finite abstractions of the value domains. Even if this does not provide a proof of validity on the original system, it is still a valuable debugging tool.

[1] [15] was using a slightly relaxed condition called "galois insertions"

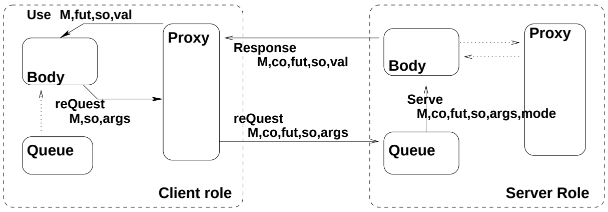


Figure 3: Communication between two Active Objects

# IV Behavioural Models for distributed Applications

## IV.1 Active Objects

The first application of pNets that we have published was for ProActive distributed applications, based on active objects, before the introduction of components. In [4, 9] we presented a methodology for generating behavioural models for ProActive, based on static analysis of the Java/ProActive code. This method is composed of two steps: first the source code is analysed by classical compilation techniques, with a special attention to tracking references to remote objects in the code, and identifying remote method calls. This analysis produces a graph including the method call graph and some data-flow information. The second step consists in applying a set of structured operational semantics (SOS) rules to the graph, computing the states and transitions of the behavioural model. The pNets model fits well in this context, and allows us to build compact models, with a natural relation to the code structure: we associate a hierarchical pNet to each active object of the application, and build a synchronisation network to represent the communication between them.

Fig. 3 illustrates the structure of the pNets expressing an asynchronous communication between 2 active objects. A method call to a remote activity goes through a proxy, that locally creates a “future” object, while the request goes to the remote request queue. The request arguments include the references to the caller and callee objects, but also to the future. Later, the request may eventually be served, and its result value will be sent back and used to update the future value.

The construction of the extended graphs by static analysis is technically difficult, and fundamentally imprecise. Imprecision comes from classical reasons (having only static information about variables, types, etc), but also for specific sources: it may not be decidable statically whether a variable references a local or a remote object. Furthermore, the middleware libraries include a lot of dynamic code generation, and the analysis would not be possible for code relying on reflexivity, classically used to manage some types of “dynamic topologies” in ProActive.

Nevertheless, for a reasonable subset of ProActive programs, we have the following result [9]:

**Theorem 1 Finite pNet Construction:** *The analysis terminates, and (up to abstraction during analysis) each active object is modelled by a finite pNet hierarchy.*

## IV.2 Hierarchical Components

Going from active objects to distributed and hierarchical components allows us to gain precision in the generated models. The most significant difference is that required interfaces are explicitly declared, and are local in the component code, so we always know whether a method call is local or remote. Moreover, the pNets’s formalism expresses naturally the hierarchical structure of components.

To formalise the model generation for components, we give a definition of the structural information that is usually given through Architecture and Interface definition languages (ADL and IDL resp.). This definition extends slightly those used in Fractal or in the GCM.

**Definition 11 Component Structure:**

*A Component $C$ is a tuple $< V, \Sigma_V, \tilde{EI}_C, \xi_C >$, where $V$ is a set of parameters, $\Sigma_V$ a term algebra, $\tilde{EI}_C$ is the set of external interfaces of $C$, and $\xi_C$ the content.*

*An Interface type $Ity = \ < M >$ is a set of methods $m = \ < T, name, \tilde{A} >$ with $T$ its return type, and each $A = \ < T_A, name >$ a typed argument.*

*An Interface is a tuple $Itf = \ < name, Ity, \kappa, \nu, \rho >$, where $Ity$ is its interface type, $\kappa$ is the Fractal contingency (mandatory or optional), $\nu$ is the interface multiplicity, and $\rho$ the interface role (either required or provided).*

*The Content of a composite component is a tuple $\xi_C = \ < I\tilde{It}f, S\tilde{C}, \tilde{B} >$, where $I\tilde{It}f$ is the set of internal interfaces, $\tilde{B}$ the set of bindings. $S\tilde{C}$ is the set of parameterized subcomponents $SC =< v, C >$, with $v \in V$ a parameter, and $C$ a component.*

*A Binding $B$ is a pair $< C_1.cItf, C_2.sItf >$ with $C_i = \ self \mid subC[expr \in \Sigma_V]$ identifies either the composite itself or one instance of a subcomponent, and $cItf$ is a client interface and $sItf$ is a server interface.*

Note that we leave here undefined the content of a primitive component. It will depend on the framework, and be used to generate a pLTS representing the primitive behaviour. We also leave undefined the algebra $\Sigma_V$, that is used to build expressions for specifying indexes within the parameterized structure; it will depend on the domains used for the parameters $V$ in a specific language.

From the information in a Component structure, it is straightforward to generate a pNet representing the communication between the interfaces and the subcomponents, from the following elements:

- the pNet has one hole for each (parametric) subcomponent;
- the pNet global actions $pA_G$ and hole sorts $\tilde{O}_J$ are sets of actions of the form $C_i.Itf[!|?]m(\tilde{arg})$ for performing / serving a method $m$ with each argument $arg \in \Sigma_{T_{arg}, V}$,
- its transducer has one parameterized synchronisation vector for each binding in $\tilde{B}$.

We have shown examples of proofs using such models in [5].

From now on, we have achieved a natural model generation for (parametric) hierarchical systems, that can be compared with existing methods of other verification frameworks, e.g. CADP, $\mu$CRL, or $\pi$ADL. One important difference is that we have explicitly limited ourselves to (countable) static systems, and use a property-preserving abstraction mechanism. Now we build on this result to introduce some management and reconfiguration mechanisms in such a way that our verification methods still apply.

## IV.3 Hierarchical Components + Management Interfaces = Fractal

In the Fractal model, and in Fractal implementations, the ADL describes a static view of the architecture, and non-functional (NF) interfaces are used to control dynamically the evolution of the system. In this section we define models for the Life-Cycle Controller (LF) and the Binding Controller (BC), in terms of pLTS generated from the Component structure of the previous section.

Stopping a component in Fractal means that its functional activity is detained, while NF calls are still allowed in order to allow reconfiguring the component. This is modelled with an interceptor of all incoming calls. Then, depending in the components life-cycle (started or stopped), functional calls are allowed or not. Similarly, we only allow rebinding interfaces when the component is stopped.

A LF pLTS (see Fig. 4) is attached to each component. Control actions (start/stop) are synchronised with the parent component and with all of its subcomponents (note that this will not be the case for the asynchronous version); and status actions (started/stopped) are synchronised with the component's functional behaviour and with the BC, because the BC may only allow rebinding of interfaces when stopped.

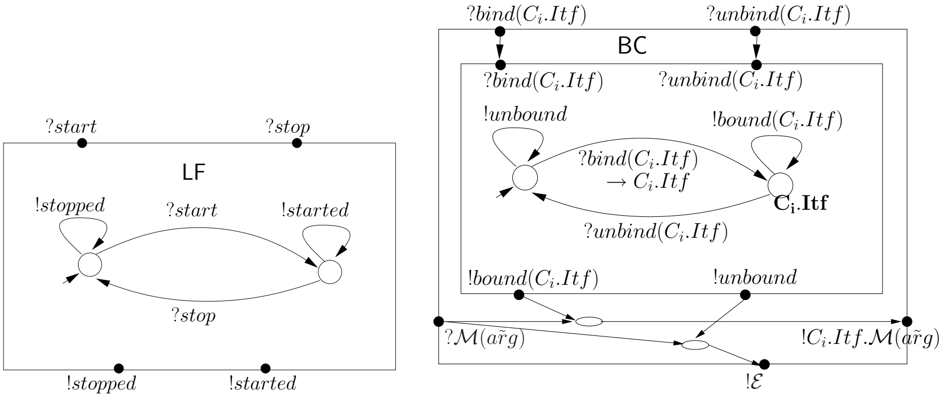


Figure 4: pLTS of Fractal Life Cycle and Binding Controllers

A BC pLTS (see Fig. 4) is attached to each interface. Control actions (bind/unbind) are synchronised up to the higher level (Fractal defines a white-box definition for NF actions) and with the affected interface; status actions (bound/unbound) are used to allow method calls $\mathcal{M}(\tilde{arg})$, to forward the call to the appropriate bound interface and to signal errors. The latter is a distinguished action $\mathcal{E}(unbound, C, Itf)$, visible to the higher level of hierarchy, and triggered whenever a method call is performed over an unbound interface.

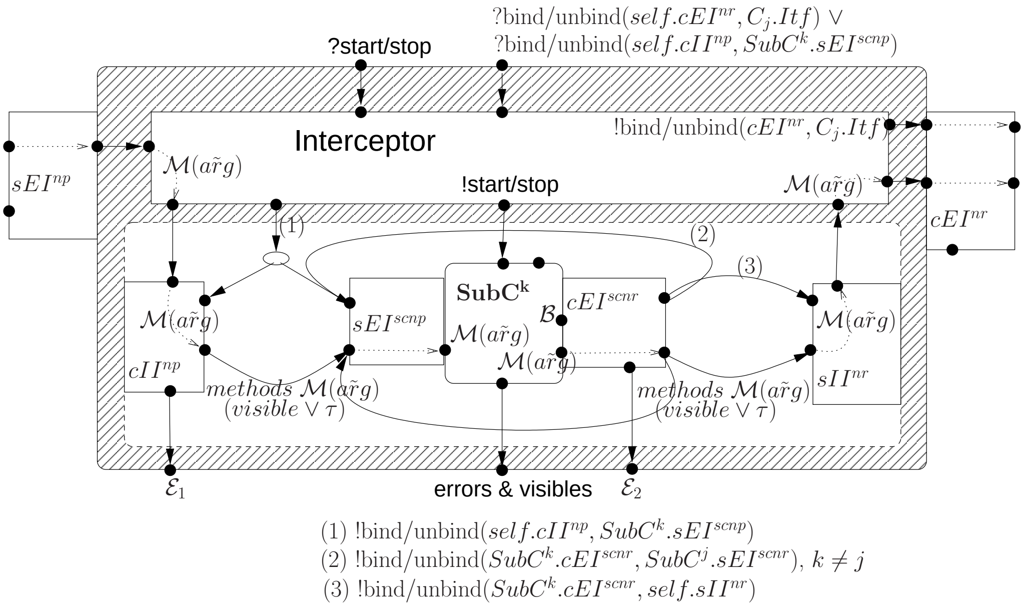
Note that we put external interface automata of a component in the next level of the hierarchy. This enables us to calculate the *controller* automaton of a component before knowing its environment. Thus, all the properties not involving external interfaces can be verified in a fully compositional manner.

By lack of space, we do not give here the detailed definition of the pNet expressing the synchronisation of the LF/BC controllers of a component with its functional behaviour, but we sketch its structure in Fig. 5. For synchronous Fractal components, the role of the interceptor is to synchronise incoming requests with the life-cycle state (either started or stopped actions) in order to restrict the allowed requests; allowed requests are synchronised with the inner part of the component (see Fig. 6).

In this drawing, the behaviour of subcomponents is represented by the box named $\mathbf{SubC^k}$. For each interface defined in the component's ADL description, a box encoding the behaviour of its internal ($cII$ and $sII$) and external ($cEI$ and $sEI$) views is incorporated. The doted edges inside the boxes indicate a causality relation induced by the data flow through the box. Primitive components have a similar automaton without subcomponents and internal interfaces.

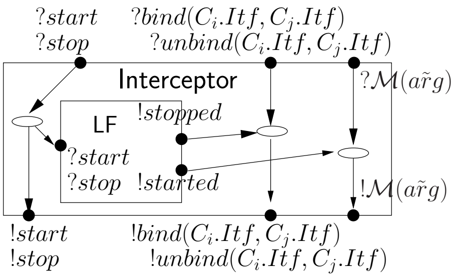
**Building and using variants of this model** The previous model construction is applied bottom-up through the hierarchy. The generated model is powerful enough to prove properties about deployment, normal behaviour, or reconfiguration, of a whole system. For pragmatic reasons, it is interesting to distinguish variants of this model in which only selected management actions are visible or authorised. We define the following variants:

- [*Static Automaton*] This is the model in which all controllers are initialised in a "started" state, and all control actions are hidden. If the ADL was correct, then it should be equivalent (up to weak



$?\mathrm{bind/unbind}(self.cEI^{nr}, C_j.Itf) \vee$
$?\mathrm{bind/unbind}(self.cII^{np}, SubC^k.sEI^{scnp})$
?start/stop
Interceptor
$!\mathrm{bind/unbind}(cEI^{nr}, C_j.Itf)$
$sEI^{np}$
$\mathcal{M}(\tilde{arg})$
!start/stop
$\mathcal{M}(\tilde{arg})$
$cEI^{nr}$
(1)
(2)
(3)
SubC$^k$
$\mathcal{M}(\tilde{arg})$
$cII^{np}$
$sEI^{scnp}$
$\mathcal{M}(\tilde{arg})$
$\mathcal{B}$
$cEI^{scnr}$
$\mathcal{M}(\tilde{arg})$
$\mathcal{M}(\tilde{arg})$
$sII^{nr}$
$methods\ \mathcal{M}(\tilde{arg})$
$(visible \vee \tau)$
$methods\ \mathcal{M}(\tilde{arg})$
$(visible \vee \tau)$
$\mathcal{E}_1$
errors & visibles
$\mathcal{E}_2$
(1) $!\mathrm{bind/unbind}(self.cII^{np}, SubC^k.sEI^{scnp})$
(2) $!\mathrm{bind/unbind}(SubC^k.cEI^{scnr}, SubC^j.sEI^{scnr})$, $k \neq j$
(3) $!\mathrm{bind/unbind}(SubC^k.cEI^{scnr}, self.sII^{nr})$

Figure 5: Synchronisation pNet for a Fractal Composite Component



$?start$
$?stop$
$?bind(C_i.Itf, C_j.Itf)$
$?unbind(C_i.Itf, C_j.Itf)$
Interceptor
$?\mathcal{M}(\tilde{arg})$
LF
$!stopped$
$?start$
$?stop$
$!started$
$!\mathcal{M}(\tilde{arg})$
$!start$
$!stop$
$!bind(C_i.Itf, C_j.Itf)$
$!unbind(C_i.Itf, C_j.Itf)$

Figure 6: Interceptor for Synchronous Fractal Components

bisimulation) to the hierarchical component model (without controllers) from the previous section, otherwise, there will typically be reachable “unbound interface errors”. It is used to check the normal behaviour of the system.

- [*Deployment Automaton*] We define a *deployment sequence* for each composite as a sequence of control operations, expressed by an automaton, ending with a distinguished successful action $\surd$. And we build an *undeployed model* similar to the static model, but with controllers initialised in their `unbound` resp. `stopped` states. Then the *deployment automaton* is the product of the undeployed model with the deployment sequences. It allows to check for correctness of deployment specifications, which is characterised by reachability of $\surd$.

- [*Reconfiguration Models*] If we build the full model, then we can check properties relative to reconfiguration. This can be very costly because of the size of the action alphabet, so it can be refined by only keeping visible selected sets of control actions.

## IV.4 Distributed Components: GCM/ProActive

In the subsection IV.1 above we have shown how to build the behaviour of ProActive activities; this corresponds exactly to the functional part of the behaviour of primitive components in our distributed implementation of Fractal. We now extend the model of Section IV.3 with this communication protocol in order to model GCM/ProActive components.

**Primitive Components** Let us recall the principle of asynchronous communication between two GCM/ProActive primitive components, inherited from ProActive (see Fig.3). There, a method call on a client interface goes through a proxy, that locally creates a "future" object, while the request goes to the request queue of the affected component. The request arguments include a reference to the future, together with a deep copy of the method's arguments; this is because there is no sharing between components. Later, the request may eventually be served, and its result value will be sent back to the future reference.

The **Body** box in Fig. 3 represents the component's functional behaviour, and is itself modelled by a synchronisation network made from the synchronisation product of the `runActivity()` method's pLTS – ProActive's service policy – with the behaviour of service methods (methods defined by provided interfaces).

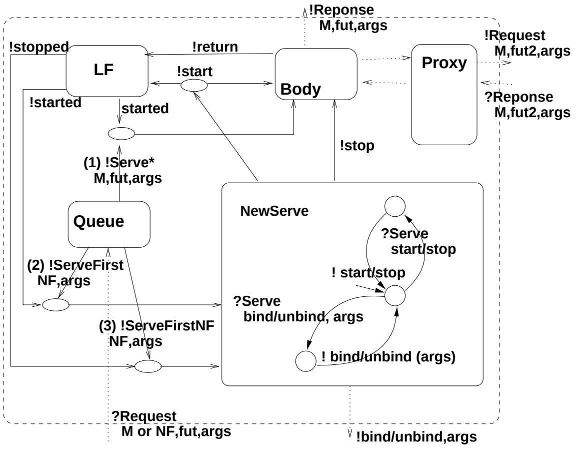


Figure 7: Behaviour model for a GCM/ProActive Primitive

In the model of a GCM/ProActive primitive component we enrich the controller of the active object by adding two extra boxes, **LF** and **NewServe**, which correspond to the Interceptor in Fig. 5. The resulting pNet is drawn in Fig. 7. The **Body** box is the only part that cannot be generated automatically from the ADL; it comes from the user-provided behaviour specification of the primitive (though its sort is fully specified).

**NewServe** implements the treatment of control requests. The action "start" fires the process representing the method `runActivity()` in the **Body**. "stop" triggers the `!stop` synchronisation with **Body** (Fig.7). This synchronisation should eventually lead to the termination of the `runActivity()` method (`!return` synchronisation). In the GCM/ProActive implementation, this is done through setting the state variable `isActive` to false, which should eventually cause the `runActivity()` method to finish, only then the component is considered to be stopped. Note that this may depend on the programmer's implementation of the `runActivity()` method, so it is worth verifying in the generated model!

The **Queue** box can perform three actions: (1) serve the first functional method corresponding to the `Serve` API primitive used in the body code, (2) serve a control method only at the head of the queue, and (3) serve only control methods in FIFO order, bypassing the functional ones.

**Composites Components** A composite membrane in GCM/ProActive is an active object. When started, it serves functional or control methods in FIFO order, forwarding method calls between internal and external functional interfaces. When stopped it serves only control requests.

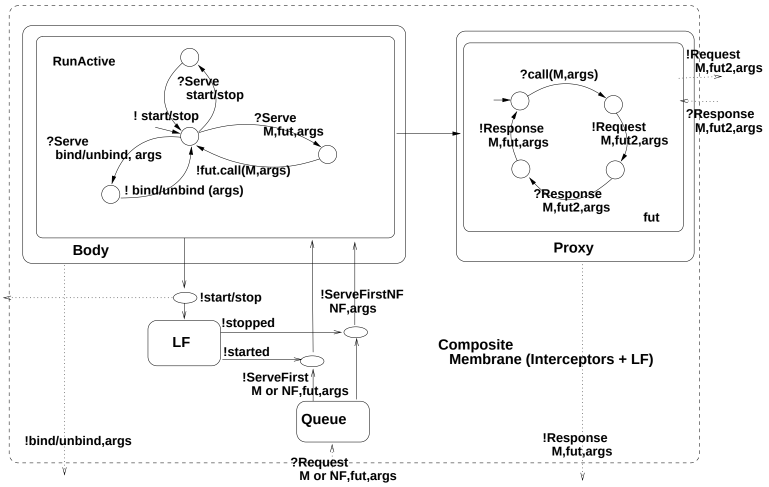


Figure 8: Behaviour of a composite membrane

Fig. 8 shows the model of the membrane, that is similar to the *interceptor* from Fig. 5, though more complex. The membrane model is created from the description of the composite (given by the ADL). Note that the future references (**Proxy** box in Fig. 8) are updated in a chain following the membranes from the primitive serving the method to the caller primitive. Since the method calls include the reference of the future in the arguments, future updates can be addressed directly to the caller immediately before in the chain. Consequently, like in the implementation, the future update would not be affected in case of a rebinding or a change in the life-cycle status of the components. Our model is expressive enough to reflect this property.

In papers [7, 5] we have shown some preliminary results of analysis performed using this model. However, as will be discussed in the next section, an automatic tool support is not yet available for the full GCM/ProActive model generation.

# V Description of the CoCoME Case Study

As a matter of testing the behavioural model above, we modelled a a full-fledged case-study called Common Component Modelling Example (CoCoME [17]). As its name suggests, CoCoME is a joint collaboration leaded by the GI-Dagstuhl Research Seminar for defining a common component example to serve as basis for comparing different component models. The system consists in a Point-Of-Sale (POS) industrial application. It is made of a `CashDeskLine` component, and an `Inventory` component. The `CashDeskLine` deals with sales whereas the `Inventory` is in charge of the database and of administrative management.

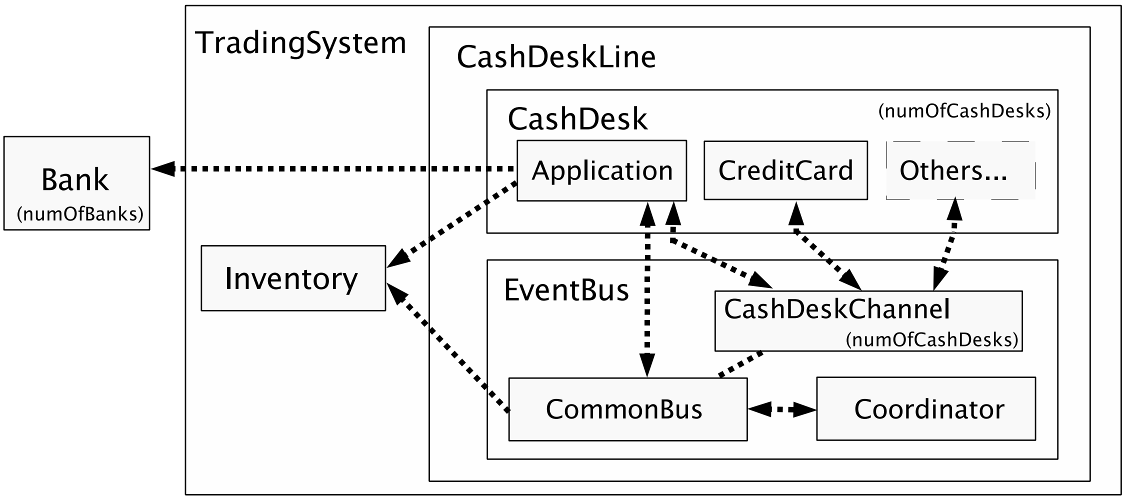


Figure 9: The CoCoME overview

Multiple `CashDesk`s are found within the `CashDeskLine`, each of them connected to a common bus (modelled as a component as well). The `CashDesk`'s peripherals, such as creditcard readers and printers, are controlled by dedicated components that bridge the middleware with the hardware. An outline of the system can be seen in Fig. 9.

The example shows off much of pNets's expressivity: (1) Components have non-trivial functional behaviour. pLTSs allow us to keep any functional behaviour affecting the application control flow including some data flow. (2) There are multiple – similar – components such as `CashDesk`s. These are expressed as families of processes in pNets allowing a generic (and condensed) representation. Arguments in method calls can be used to address a specific component within the family. (3) There are 5 layers of composition wherein pNets' hierarchical structure fits in.

# VI Platform Overview

Our platform comprises several tools for assisting the verification process. Rather than creating a new model-checker, we implement our model-generation methods in a way that they efficiently integrate with existing state-of-the-art tools for checking component specifications based on the models of Section IV.

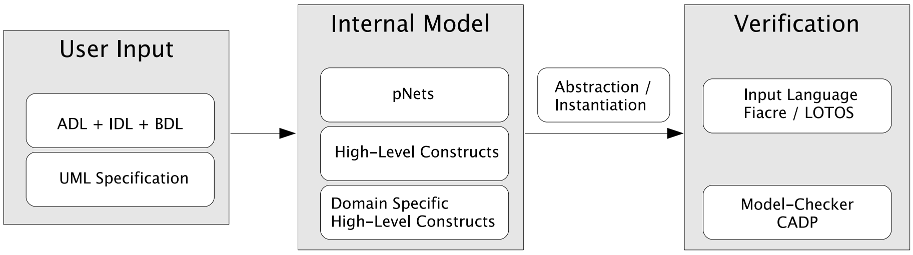


Figure 10: The VERCORS architecture

Fig. 10 gives a snapshot of the platform. In the next subsections we shall describe in details its three parts: the input from the user (VI.1), the behavioural model (VI.2), and the verification of properties (VI.3). We illustrate our platform through the formal verification of the previously outlined case-study.

## VI.1 User Input

For automatically building the behavioural model we take a two-fold approach: (1) the architecture and hierarchy information are extracted from the ADL (and IDL); and (2) each of the primitive component's functional behaviour is specified by the user in an automata-based language which we call Behavioural Description Language (BDL).

The architecture shown in Fig. 9 is specified in a XML file using the Fractal ADL. This file specifies the deployment topology of the component system. The ADL for the CashDesk is defined as:

```
<component name="CashDesk">
  <!-- interfaces -->
   <interface signature="CashDeskLine.if.CashBoxEventIf" role="client" name="cashBoxEventIf"/>
   <interface signature="CashDeskLine.if.ScannerEventIf" role="client" name="scannerEventIf"/>
  ... other interfaces
  <!-- subcomponents -->
  <component name="ScannerController">
  ... other subcomponents
  <!-- bindings -->
  <binding client="ScannerController.scannerEventIf" server="this.scannerEventIf"/>
  ...
```

Then, interface signatures are given with the Fractal Interface Definition Language (IDL). In the implementations we consider, this definition is given by Java interfaces describing the signatures of the methods of each component interface. With this analysis, we are able to fill in the model of Section IV.2 and IV.3.

Finally, the functional behaviour is given by a BDL. We need a behaviour language expressing transition systems with data, but much more abstract and user-friendly than pNets, and that will be easily related with the component structure (ADL and IDL) In our current prototype, we have used LOTOS, that is a natural choice for interfacing with the CADP toolset, but needs a complicated mapping with the IDL objects. Recently, we have also developed a tool called CTTool [1], using UML2 statemachines diagrams to express pLTSs, and a variant of UML2 component structures to specify the system architecture (but only in the static case). We also plan to provide a textual specification language that would integrate smoothly architecture and behaviour specifications for GCM applications, but this is still in progress.

## VI.2 Internal Model

We start the analysis of the input files mentioned above for automatically building the behavioural model in pNets seen in Section IV. This is done by `ADL2N`, which is a tool written in Java for generating the behavioural models of Fractal components by analysing the system's ADL and IDL (see Section IV.2).

Similar to a Fractal implementation, the use of BC and LF controllers allows one to model the deployment of the system as well as to do basic reconfiguration within the system. In our case checking the safeness of these can be done statically by building the *Static*, *Deployment* or *Reconfiguration* automata of Section IV.3.

In practice the user of `ADL2N` will use the tool GUI to specify at the same time the methods that will be visible, the arguments that are significant, and the finite instantiations of those parameters. The visibility of methods and the abstraction (see Section III.3) depend on the formulas to be checked. Although it should be possible to infer safe abstractions given a set of formulas, for the moment it is up to the user to provide finite abstractions of the data domains.

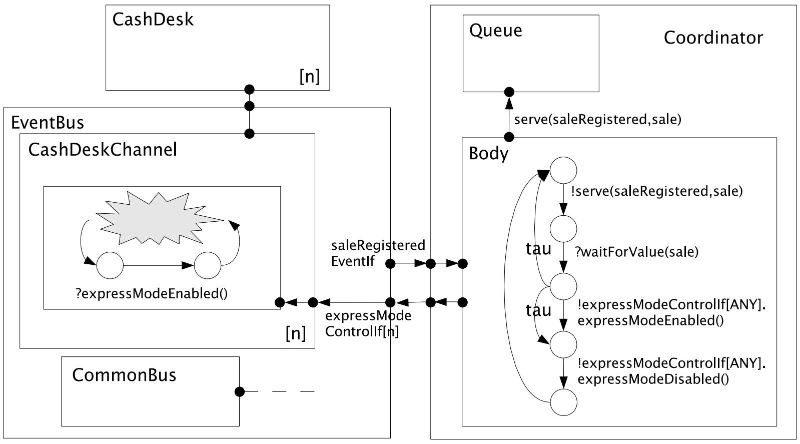


Figure 11: A partial pNets model of CoCoME

The output of ADL2N is the pNets behavioural model of Section IV.3 with the above abstractions and with the selected actions hidden. In Fig. 11 we include a sketch of a pNets model for the CoCoME. For CoCoME, specifying instantiations in such a way that we could check 6 formulas (expressing various usage scenarios), the generated model had 81 distinct transition labels (instances of communication events). Its size before reduction was approx. 1.25 million states / 3 million transitions, and after reduction by branching bisimulation only 9800 states / 33 000 transitions.

For the moment, our tools are only generating the synchronous models discussed on Section IV. Although limiting, it allowed us to find some interesting properties of the case-study discussed in the following.

## VI.3 Verification

In the current toolset, we only interface with finite-state model-checkers, and namely with the Evaluator model-checker from the CADP toolset, that feature a very efficient check of branching-time logics, together with on-the-fly generation, cluster-based distributed state-generation, tau-confluence reduction, etc.

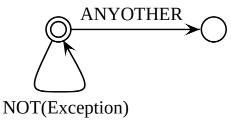
We give here verification examples of various usage scenarios. There are many ways of encoding formulas. Some of them are very powerful as $\mu$-calculus, but at the same time hardly usable by non-experts. We propose to write formulas using extended automata. Their transitions contain predicates with logic quantifiers, and naturally the same data-types than the system specification. Their states can be marked as either acceptance or rejection. An automaton may change to any state whose transition predicates are satisfied. If a final state is unreachable, the formula is false. Moreover, there are special predicates:

- `NOT(i), (i AND j), (i OR j)` with their usual meaning,
- and `ANYOTHER` as a shortcut meaning that all labels not satisfying other transitions from the state satisfies the predicate.

### VI.3.1 Absence of Deadlocks

There are basic formulas that can be proved, the most common being the absence of deadlocks. In the case of our CoCoME specification, this ends-up being trivially false because of two reasons:

- the presence of exceptions: in our specification, raising an exception blocks the system. So we should rather search for deadlocks that are not following an exception;
- the synchronous semantics of Fractal components used in the current state of tools: our components are mono-threaded, and communications are synchronous. As a result, the system deadlocks due to race conditions over the EventBus.

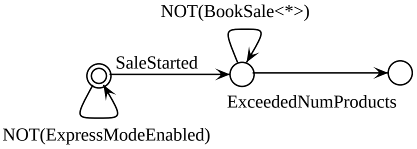


To show this, we write a formula expressing that all deadlocks are the consequence of an exception, and model-check this formula. More precisely we write the negation, i.e. that any transition is followed by some other transition as long as an exception has not been raised. The answer we get when we evaluate the former formula is “false” (the formula does not stand). The diagnostic trace shows two controllers on a race condition over the EventBus. They form a dependency-cycle and thus the system deadlocks.

Note that these kind of scenarios would not be present in a distributed version using GCM / ProActive because of the asynchronous method calls. These requests are buffered in the queues. Therefore, we have more deadlocks in a synchronous implementation of the system than those we would have with ProActive.

Nevertheless, using the CTTool specification we were able to prove some interesting scenarios, and to find some errors (or underspecifications) within the reference CoCoME specification.

### VI.3.2 Safety of the Express Mode



An unspecified scenario was found relating a Use Case from CoCoME. There is nothing within the reference specification that states when a CashDesk may switch from/to an *Express Mode*. In fact, the system ends-up in an inconsistent state if an express mode signal is triggered during an ongoing sale.

This scenario can be found using the formula on the left. A sale starts within the *Normal Mode*, and before a sale is booked an exception unique to the *Express Mode* is raised.

# VII Conclusion and Perspectives

This article defines the pNet model, a powerful extension of labelled transition systems, that features more structuring in terms of hierarchical synchronisation networks, and more expressivity through the use of parameters at both LTS and Networks levels. This model is used for representing the behaviour semantics of distributed systems, starting with a basic active objects model, then introducing step by step a hierarchical component structure, Fractal non-functional controllers, and finally the GCM/Proactive distributed implementation of Fractal.

This kind of semantic-level model is widely used inside analysis and verification toolsets, because it provides a compact and well-defined intermediate format for connecting code analysers or code generators with model-checking or equivalence engines. When dealing with concurrent or distributed systems, intermediate models often make strong hypotheses on the type of synchronisation and communication mechanisms addressed, for example LOTOS-like parallelism in CADP, channels in Promela, or Petri nets in other cases. Our choice with the pNet model is to have low-level primitives (LTS + synchronisation vectors) that are able to represent many possible mechanisms. Another important trade-off is between parameterized representations (close to developers code) and lower-level explicit-state encodings that are required by the model-checkers.

We argue that the pNets model allows for finite and compact representation of systems, expressive enough to capture a large family of behavioural properties of both synchronous and asynchronous applications.

Our definition of model-generation algorithms for distributed component models is part of a larger project, and we have stressed that an ambitious goal of this project is to make these tools available to (non-specialist) developers. The last section of this article sketches the current state of our verification platform, and results of model construction and analysis for a middle-size case-study. The tools currently allow to build behavioural models for synchronous Fractal components with partial support for non-functional controllers. The case-study shows that it scales up well.

The Vercors platform (generation, instantiation and conversion tools) and the CTTool editor, as well as the CocoME case-study, are available at our website [2].

We are currently working on the controller generation for the GCM/Proactive asynchronous components. Encoding their request queues brutally with pNets is possible, but can be very expensive in term of state/transition complexity. Possible solutions use either dedicated algorithms or on-the-fly techniques for model generation, or specific parametric representations (and specialised "infinite-state" engines).

There is an open problem for properly integrating the behaviour description language with the rest of the component descriptions. This will be still more important when dealing with reconfiguration specifications. We are working on a specification language integrating architectural and behavioural views, with high-level constructs for system reconfiguration, and for Grid specific features like collective interface policies. Concretely, this will be a Java-like language that takes architectural aspects as primitives within the language, and complex communication primitives for dealing with multiple components, asynchronous method calls, and data distribution. This language can be used as an input for the Vercors platform, but also for tools that will generate Java code-skeletons with strong guarantees.

# References

[1] S. Ahumada, L. Apvrille, T. Barros, A. Cansado, E. Madelaine, and E. Salageanu. Specifying Fractal and GCM Components With UML. In *proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, Iquique, Chile, November 2007. IEEE.

[2] A. Arnold. *Finite transition systems. Semantics of communicating sytems*. Prentice-Hall, 1994.

[3] T. Barros. *Formal specification and verification of distributed component systems*. PhD thesis, Université de Nice - INRIA Sophia Antipolis, November 2005.

[4] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, volume LNCS 3235, Madrid, September 2004. Spinger Verlag.

[5] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components : The Vercors platform. In *3rd workshop on Formal Aspects of Component Systems*, Prague, Czech Republic, Sep 2006. ENTCS.

[6] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In Patrice Godefroid, editor, *Model Checking Software, 12th Int. SPIN Workshop*, San Francisco, CA, USA, August 2005. LNCS 3639, Springer.

[2] `http://www-sop.inria.fr/oasis/Vercors`

[7] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. ENTCS.

[8] J.A. Bergstra, A. Pose, and S.A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[9] Rabéa Boulifa. *Génération de modèles comportementaux des applications réparties*. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences, December 2004.

[10] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. An open component model and its support in java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, may 2004.

[11] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[12] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[13] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[14] R. Cleaveland and M.C.B. Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5:1-20, 1993.

[15] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *CONCUR'94*. LNCS 836, Springer, 1994.

[16] CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed). Technical report, 2006. Deliverable D.PM.04, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[17] Dagstuhl Research Seminar. The Common Component Modeling Example: Comparing Software Component Models, to appear 2007. http://agrausch.informatik.uni-kl.de/CoCoME.

[18] H. Garavel and F. Lang. NTIF: A general symbolic model for communicating sequential processes with data. In *Proceedings of FORTE'02 (Houston)*. LNCS 2529, November 2002.

[19] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[20] A. Lakas. *Les Transformations Lotomaton : une contribution à la pré-implémentation des systèmes Lotos*. PhD thesis, Univ. Paris VI, june 1996.

[21] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, 26–29 August 1996. LNCS 1119.

[22] E. Madelaine. Verification tools from the CONCUR project. *EATCS Bull.*, 47, 1992.

[23] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs, 2nd Edition*. John Wiley & Sons, 2006.

[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1), 1992.

[25] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[26] E. Najm, A. Lakas, A. Serouchni, E. Madelaine, and R. de Simone. ALTO: an interactive transformation tool for LOTOS and LOTOMATON. In T. Bolognesi, E. Brinksma, and C. Vissers, editors, *Third Lotosphere Workshop and Seminar*, Pisa, September 1992.

[27] P. Poizat and J.C. Royer. A Formal Architectural Description Language based on Transition Systems and Modal Logic. *Journal of Universal Computer Science*, 12(12), 2006.

[28] P. Poizat, J.C. Royer, and G. Salaun. Bounded Analysis and Decomposition for Behavioural Descriptions of Components. In *FMOODS, LNCS 4037*, 2006.

[29] A.W. Roscoe. Model-Checking CSP. In Prentice-Hall, editor, *A Classical Mind, Essays in honour of C.A.R. Hoare*, 1994.

[30] J.B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, 1998.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex

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