Ann. Telecommun. (2009) 64:25–43
DOI 10.1007/s12243-008-0069-7

Behavioural models for distributed Fractal components

Tomás Barros · Rabéa Ameur-Boulifa · Antonio Cansado · Ludovic Henrio · Eric Madelaine

Received: 30 July 2007 / Accepted: 16 July 2008 / Published online: 10 January 2009
© Institut TELECOM and Springer-Verlag France 2008

Abstract This paper presents a formal behavioural specification framework for specifying and verifying the correct behaviour of distributed Fractal components. The first contribution is a parameterised and hierarchical behavioural model called *pNets* that serves as a low-level semantic framework for expressing the behaviour of various classes of distributed languages and as a common internal format for our tools. Then, we use this model to define the generation of behavioural models for applications ranging from sequential Fractal components, to distributed objects, and finally to 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 (semi)automatically verify properties of his application.

T. Barros
Universidad de Chili, Ejército 441, Santiago, Chile
e-mail: tomas.barros@niclabs.cl

R. Ameur-Boulifa
GET/ENST/LabSoC, Telecom Paristech, BP 193,
06904 Sophia-Antipolis Cedex, France
e-mail: Rabea.Ameur-Boulifa@telecom-paristech.fr

A. Cansado · L. Henrio · E. Madelaine (✉)
INRIA Sophia-Antipolis, CNRS, UNSA,
INRIA, Oasis. 2004, Route des Lucioles, BP 93,
06902 Sophia-Antipolis Cedex, France
e-mail: Eric.Madelaine@sophia.inria.fr

A. Cansado
e-mail: Antonio.Cansado@sophia.inria.fr

L. Henrio
e-mail: Ludovic.Henrio@sophia.inria.fr

Keywords Hierarchical components · Distributed asynchronous components · Formal verification · Behavioural specification · Model-Checking

1 Introduction

Component models provide a structured programming paradigm allowing a better reusability 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* [1] provides the following crucial features: the explicit definition of provided/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 concerns 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)* [2], which extends Fractal by addressing large-scale distributed aspects of components.

Moreover, in a 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-checker 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 [3, 4]), and the synchronisation vectors of [5], to Milner's π-calculus [6]. 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 π-calculus has only been used in a few research prototypes, because its high expressivity comes with a very high complexity of most related algorithms.

Early verification tools were using internal formats with a very simple structure, featuring no data parameters; even intermediate formats used to interface different tools were kept at a very low level. However, introducing data in those languages appeared quickly as being very beneficial both for compactness and for expressiveness.

For example, in the CADP toolbox [7], 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 [8], resembling our pLTS, has been devised as a more structured and compact intermediate form between LOTOS or ELOTOS programs and the CADP engines.

In a similar way, the SPIN model-checker is using PROMELA, a high-level language with data, but data values are instantiated (on bound domains) by the state exploration 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 [9] offers a high-level language for expressing CSP models, and an internal machine-readable dialect of CSP [10] using a specific expression language, more adapted to generate the models needed by the verification engines. The LTSA tool [11] uses Finite State Processes as an intermediate language (with processes and data parameters) for modelling concurrent Java programs. Another example of research showing goals close to ours makes use of Symbolic Transition Systems (STS) [12, 13], which are structures akin to our pNets. In the STSLib toolset, there is a dedicated specification language (with algebraic 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 (parameterised) internal models into finite structures suitable for analysis (internal data structures of the verification engines, typically LTS, BDD, or various classes of automata…)?

Our proposal is different from previous approaches in the sense that we want a low-level model able to express various mechanisms for distributed systems, and that we do not limit ourselves to finite systems: we shall be able to define mappings to various classes of systems, finite or not. At the same time, the structure of our parameterised model is closer to the programming or specification language structure. Consequently, parameterised models are more compact, and easier to produce, than classical internal models.

Typically, our pNets model is lower-level than Lotos and Promela and more flexible for expressing different synchronisation mechanisms. On the other hand, it has no recursive constructs, in order to better control the finiteness of encodings.

Contribution This paper tries to answer these questions in the framework of distributed component systems. Toward this challenging perspective, we

develop a formal and parameterised behavioural model called *pNets*. We use 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 the 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 3, we define formally our basic model, named pNets (this formalisation unifies and extends our previous publications in [14–17]) and recall the main properties of this model. In Section 4, we describe the model construction principles for four successive kinds of applications, namely active objects, hierarchical components, Fractal components with synchronous controllers and asynchronous GCM components with controllers. In Section 5, we describe the Vercors verification platform, and its application to a simple example, starting from the input specifications, through the model generation phase, to the verification of properties. We conclude with an analysis of perspectives of this work.

2 Context

2.1 Fractal, GCM and ProActive

The GCM [2] 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 [1] 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 (client) and gathercast (server). Typically, a multicast interface is bound to the service interfaces of a number of parallel components, and a method call toward this interface is distributed, as well as its parameters, to several or all of them. GCM provides various policies for the request parameters that can be broadcast, or scattered, or distributed in a round-robin fashion; additional policies can be specified by the user. Symmetrically, gathercast interfaces are bound to a number of client components, and various synchronisation policies are provided. This treatment of collective communications provides a clear separation of concern between the programming of each component and the management of the application topology: within a component code, method calls are addressed simply to the component local interfaces. The management of bindings of clients (on a gathercast interface) or services (on a multicast interface) is separated from the functional code.

The GCM also allows the component controllers to be designed 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* 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.

2.2 A GCM reference implementation: GCM/ProActive

A GCM reference implementation is based on ProActive [18], an Open Source middleware implementing the ASP calculus [19, 20]. 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 with respect to the programming model:

- Components communicate through asynchronous method calls with transparent futures (placeholders for promised replies): a method call on a server interface adds a request in the server's *request queue*.
- Communication semantics uses a "rendezvous", 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), and the result is still unavailable.
- 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. Some methods of this active object are exported as the method of the component's interfaces. The active object managing a composite is generic and provided by the GCM/ProActive platform; it forwards the functional requests it receives to its subcomponents. Primitive component functionalities are addressed by the encapsulated active object. For primitive components, it is possible to define the order in which requests are served by writing a specific method called `runActivity()`; we call this the service policy. If no `runActivity()` is given, a default one implements a FIFO policy. Composite components always use a FIFO policy. 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: each primitive component represents the unit of distribution and is managed by a single thread. Composite components are also managed by their own thread and allocated separately, but there is no link between the location of a composite and the location of its subcomponents. 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.

2.2.1 Life-cycle of GCM/ProActive components

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.

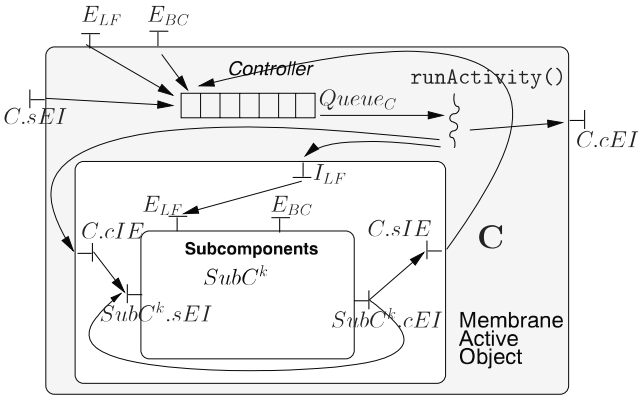


Fig. 1 *ProActive* composite component

Like in Fractal, when a component is stopped, only control requests are served. A component is started by invoking the non-functional request: `start()`. Because threads are non-interruptible in Java, a component necessarily finishes the request it is treating before being stopped. If a `runActivity()` method is specified by the programmer, the stop signal must be taken into account in this method. 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.

3 Theoretical model

In this section, we give the formal definition of our intermediate language that we call *parameterised Networks of Synchronised Automata (pNets)*. This language is not a new *calculus* in the tradition of theoretical computer science that gave birth to λ-calculus, π-calculus or σ-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 generic 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 and Nivat [5] 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 and Nivat.

In the 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 parameterised to encode sets of equivalent agents running in parallel. This leads us to the definition of pNets, that will later appear as a natural model of software systems. Indeed they correspond to the way developers specify or program these systems: the system structure is parameterised 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. In order for this article to be self-contained and with uniform notations, we first define LTSs, Nets and synchronisation product; these definitions are equivalent to those found in the literature. Then, we give the definitions of our parameterised 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 indexed sets. The indexes will usually be integers, bounded or not. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, 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 ($\tilde{A}_J = \langle A_j \rangle_{j\in J}$), and $\tilde{a}_J$ for an indexed multiset of elements ($\tilde{a}_J = \langle a_j \rangle_{j\in J}$), 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 $\langle a.\tilde{a}_J \rangle$ 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 ($\langle x_j = e_j \rangle_{j\in J}$), $e\{\tilde{x}_J \leftarrow \tilde{e}_J\}$ 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:

$$a_0 \in A_0 \wedge \tilde{a}_J \in \tilde{A}_J \Rightarrow \langle a_0.\tilde{a}_J \rangle \in \prod_{j\in\{0\}\cup J} A_j$$

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 includes at least a distinguished *Boolean* type and an *Action* type.

3.1 Networks of synchronised automata

We model the behaviour of a process as a LTS in a classical way [3]. The LTS transitions encode the actions that a process can perform in a given state.

Definition 1 LTS. A LTS 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 and $\rightarrow$ is the set of transitions $\rightarrow \subseteq SxLxS$. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

We define **Nets** in a form inspired by [5], that are used to synchronise a (potentially infinite) number of processes.

Definition 2 Net (Network of LTSs). Let Act be an action set. A Net is a tuple $\langle A_G, J, \tilde{O}_J, T \rangle$ 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 $L_T = \{\overrightarrow{v} = \langle a_g.\tilde{\alpha}_I \rangle.\ a_g \in A_G,\ I \subseteq J \wedge \forall i \in I, \alpha_i \in O_i\}$

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 [22, 23]. A transducer in the Net is encoded as a LTS which labels are synchronisation vectors ($\overrightarrow{v}$), each describing one particular synchronisation between the actions (α_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 state in the transducer 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 A *System* is a tree-like structure in which nodes are Nets and leaves are LTSs. At each node, a partial function maps holes to corresponding subsystems. A system is closed if all holes are mapped and open otherwise.

Definition 4 The *Sort* of a system is the set of actions that can be observed from outside the system. It is determined by its top-level node, L for a LTS, and A_G for a Net:

$$Sort(S, s_0, L, \rightarrow) = L \qquad Sort(\langle A_G, J, \tilde{O}_J, T\rangle) = A_G$$

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 terms 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 [22], but this is not necessary for our goals here.

Definition 5 Synchronisation product. Given an indexed set $\tilde{P}_J$ of LTSs $\tilde{P}_J = (\tilde{S}_J, \tilde{s_0}_J, \tilde{L}_J, \tilde{\rightarrow}_J)$, and a Net $\langle A_G, J, \tilde{O}_J, T = (S_T, s_{0_T}, L_T, \rightarrow_T)\rangle$, such that $\forall j \in J,\ L_j \subseteq O_j$, we construct the product LTS $(S,\ s_0,\ L,\ \rightarrow)$ where $S = \prod_{j\in\{T\}\cup J} S_j$, $s_0 = \langle s_{0_T}.\tilde{s}_{0_J}\rangle$, $L \subseteq A_G$, and the transition relation is defined as:

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

3.2 Parameterised networks of synchronised automata

Next, we enrich the above definitions with parameters in the spirit of Symbolic Transition Graphs [21]. We start by giving the notion of parameterised 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 6 Parameterised actions. Let V be a set of names, $\mathcal{L}_{A,V}$ a term algebra built over V, including the constant action τ. We call $v \in V$ a parameter, and $a \in \mathcal{L}_{A,V}$ a parameterised action, $\mathcal{B}_{A,V}$ the set of boolean expressions (guards) over $\mathcal{L}_{A,V}$.

Example In Milner's *value-passing CCS* [3], the action algebra has constructors "`tau`", "`a`" for input actions, "`'a`" for output actions and "`a(x)`" for parameterised action. Then, "`'out(3)`" is a closed output action term, "`a(x,y)`" an open input action term with parameters `x` and `y` and "`x+y=3`" a guard.

Definition 7 pLTS. A parameterised LTS is a tuple $(V, S, s_0, L, \rightarrow)$ where:

- V is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,V}$.
- S is a 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 $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle$ such that if $s \xrightarrow{l} s'$, then:
  - α is a parameterised 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 *optional* guard.
- The variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by the *optional* expressions $\tilde{e}_{J_{s'}}$

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

Example Figure 2 is based on an implementation of the philosopher problem in ProActive. It represents the pLTS for the body behaviour of a Philo active object (see how we generate active object behaviour models in Section 4.1). The action alphabet used here reflects the active object communication schema: each remote request sent by the body has the form "$!dest.\texttt{request}(f, \mathcal{M}(\tilde{arg}))$", where *dest* is the remote reference, $\mathcal{M}$ is the method name, with parameters $\tilde{arg}$ and f is a future reference. More precisely, f is the identifier of the future proxy instance. Requests that do not require a response do not use a future proxy.

Definition 8 A *pNet* is a tuple $\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$, where: V is a set of parameters, $pA_G \subset \mathcal{L}_{A,V}$ is its set of (parameterised) external actions and 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: $\overrightarrow{v} = \langle a_g, \{\alpha_t\}_{i\in I, t\in B_i}\rangle$ 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 "parameterised 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 parameterised holes express *parameterised topologies* of processes synchronised by a given Net. Each parameterised 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 such graphics in previous publications to represent parameterised processes in the Autograph editor [24].

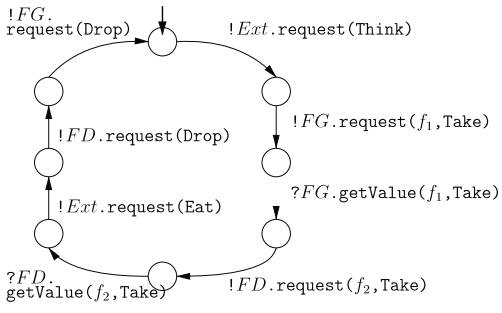
The sorts of our parameterised structures are sets of parameterised actions. This definition extends the simple sorts from Definition 4:

Definition 9 Parameterised sorts:

- The sort of a pLTS: $Sort(V, S, s_0, L, \rightarrow) = \{\alpha \mid \exists l \in L.\, l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle\}$
- The sort of a pNet: $Sort\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle = pA_G$

Example The drawing in Fig. 3 shows a (static) pNet representing the classical philosophers problem, with two parameterised holes (indexed by the same variable k) for philosophers and forks. On the right-hand side are the corresponding elements of the formal pNet, in which we explicitly list the sort of the holes (O_{philo} and O_{Fork}), and where appear synchronisation vectors parameterised over the index k and the future ids f_1 and f_2.

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 parameterisation of arguments: an actual (parameterised) argument of a pNet at position j is a



PhiloRunActivityLTS = $\langle V, S, s_0, L, \rightarrow\rangle$ with:

$V = \{f_1, f_2\}$

$S = \{s_i\}$, `i ∈ [0:7]`

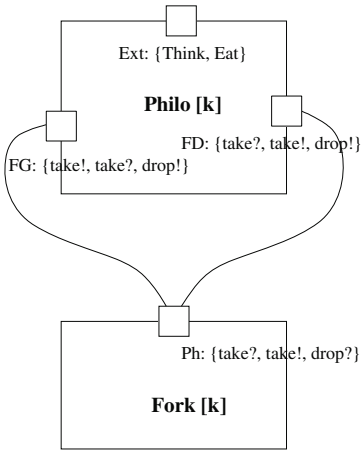
L = { !*Ext*.request(Think), !*Ext*.request(Eat), !*FG*.request(f_1,Take), ?*FG*.getValue(f_1,Take), !*FD*.request(f_2,Take), ?*FD*.getValue(f_2,Take), !*FG*.request(Drop), !*FD*.request(Drop) }

$\rightarrow$ such that:

s_0 : !*Ext*.request(Think) $\rightarrow$ s_1,
s_1 : !*FG*.request(f_1,Take) $\rightarrow$ s_2
...

Fig. 2 Example of pLTS

Fig. 3 Example of pNet



PhiloNet $= \langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$ with:

$V = \{k, f_1, f_2\}$

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*.request(Think), !*Ext*.request(Eat), !*FG*.request(f_1,Take), !*FD*.request(f_2,Take), ?*FG*.getValue(f_1,Take), ?*FD*.getValue(f_2,Take), !*FG*.request(Drop), !*FD*.request(Drop)}

O_{Fork} = {?*Ph*.request(f_1,Take), ?*Ph*.request(f_2,Take), !*Ph*.getValue(f_1,Take), !*Ph*.getValue(f_2,Take), ?*Ph*.request(Drop)}

This pNet is static, T has a unique state, and transitions with the following labels:
L_T = {
⟨Think(k), !Philo[k].*Ext*.request(Think)⟩
⟨Eat(k), !Philo[k].*Ext*.request(Eat)⟩
⟨!TakeG(k), !Philo[k].*FG*.request(f_1,Take), ?Fork[k].*Ph*.request(f_1,Take)⟩
⟨!TakeD(k), !Philo[k].*FD*.request(f_2,Take), ?Fork[k+1].*Ph*.request(f_2,Take)⟩
⟨?TakeG(k), ?Philo[k].*FG*.getValue(f_1,Take), !Fork[k].*Ph*.getValue(f_1,Take)⟩
⟨?TakeD(k), ?Philo[k].*FD*.getValue(f_2,Take), !Fork[k+1].*Ph*.getValue(f_2,Take)⟩
⟨DropG(k), !Philo[k].*FG*.request(Drop), ?Fork[k].*Ph*.request(Drop)⟩
⟨DropD(k), !Philo[k].*FD*.request(Drop), ?Fork[k+1].*Ph*.request(Drop)⟩ }

pair $\langle Sys, \mathcal{D}\rangle$, 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 parameterised LTS and Nets, based on a (possibly 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-parameterised) 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 former, 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 10 pLTS Instantiation. Given a pLTS $P_p = \langle V, S_p, s_{0_p}, L_p, \rightarrow_p\rangle$, 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 ρ_0 for the variables of the initial state s_{0_p}, the instantiation $\Phi(P_p, \mathcal{D}_V)$ is a LTS $P = \langle S, s_0, L, \rightarrow\rangle$ such that:

- $S = \bigcup_{s_p\in S_p} \left\{ s_p\{\tilde{x}_V \leftarrow \tilde{e}_V\} \mid \forall x \in V,\ \forall e_V \in \mathcal{D}(x)\right\}$,
- $s_0 = s_{0_p}\{fv(s_{0_p}) \leftarrow \rho_0(fv(s_0))\}$,
- L is the set of ground actions (i.e. closed terms) of the action algebra $\mathcal{L}_{A,V}$,
- $\rightarrow\ (\subseteq SxLxS) = \bigcup_{t\in\rightarrow_p} \Phi(t)$ is the union of instantiations of the parameterised transitions, built in the following way:
  let $t = s \xrightarrow{l_p=\langle\alpha, e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle} s'_p$ 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

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

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 11 pNet Instantiation. Given a pNet $N_p = \langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$, 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 $N = \langle A'_G, J', \tilde{O}'_{J'}, T'\rangle$, with $T' = \langle S_{T'}, s_{0T'}, L_{T'}, T_{T'}\rangle$ constructed in the following way:

1. Expand the parameterised holes: $J' = \Phi(J) = \biguplus_{j\in J}\mathcal{D}(p_j)$ where $\biguplus$ 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:
   $S_{T'} = S_T$
   $s_{0T'} = s_{0T}$
   $L_{T'} = \bigcup_{\vec{v}\in L_T}\{\Phi(\vec{v})\}$ the expansion of the synchronisation vectors
   $T_{T'} = \bigcup_{(s,\vec{v},s')\in T_T}\{(s, a, s'), a \in \Phi(\vec{v})\}$ the expansion of the transition relation
   with $\Phi(\vec{v})$ computed by :

   let $\vec{v} = \langle a_g, \{\alpha_{i,t}\}_{i\in I, t\in B_i}\rangle$,
   let $V = fv(\vec{v})$,
   and $\mathcal{D}_V$ their instantiation domains,
   for each possible valuation $\tilde{e}_V$ of $\tilde{x} \in V$,
   (let $\phi = \{\tilde{x}_V \leftarrow \tilde{e}_V\}$ be the corresponding instantiation function,
   expand each parameterised action by
   $\Phi(\alpha_{j,t}) =$
   if $j \notin I$ then $\langle *, ..., *\rangle$
   else $\langle x_1, ..., x_{|J'_i|}\rangle$,
   with $x_k = *$ if $k \notin B_i$, else $\phi(\alpha_{j,t})$,
   build $\Phi(\phi, \vec{v})$ as a vector of cardinality $|J'|$
   as the concatenation of subvectors $x \in \Phi(\alpha_{j,t})$ for each hole $j \in J$),
   return $\Phi(\vec{v}) = \{\Phi(\phi, \vec{v})\}_{\{\tilde{e}_V\}}$

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 We give here a small instantiation of the philosopher system from Fig. 3:

$\Phi\left(PhiloNet, \mathcal{D}(k) = \{1, 2\}, \mathcal{D}(f_1) = \{1\}, \mathcal{D}(f_2) = \{2\}\right)$
=
$\langle A'_G, J', \tilde{O}'_{J'}, T'\rangle$ with:
$A'_G = \{$`Think(1), Think(2), Eat(1), !TakeG(1),...`$\}$
$J' = \{$`Philo, Philo, Fork, Fork`$\}$
$O\prime_{Philo^{(1)}} = \{$`!`$Ext$`.request(Think), !`$Ext$`.request(Eat), !`$FG$`.request(1,Take), ...`$\}$
$O\prime_{Philo^{(2)}} = \{$`!`$Ext$`.request(Think), !`$Ext$`.request(Eat), !`$FG$`.request(1,Take), ...`$\}$
...
$L_{T'} = \{$
`⟨ Think(1), !`Ext`.request(Think), *, *, * ⟩`
`⟨ Think(2), *, !`Ext`.request(Think), *, * ⟩`
...
`⟨ !takeG(1), !`FG`.request(1,Take), *, ?`Ph`.request(1,Take), * ⟩`
`⟨ !takeD(1), !`FD`.request(2,Take), *, *, ?`Ph`.request(2,Take) ⟩`
... $\}$

Expressivity In [14], 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. However, one should note that this expressivity is gained from the properties of the indexes domains (here, integers with standard arithmetic): the pNets formal definition is (on purpose) separated from the data domain definition and does not provide by itself any formal expressivity result.

Another aspect of expressivity is the representation of classical patterns of distributed systems. We claim that pNets, used with simple (first-order) parameter domains, provide powerful and easy representations for our needs, including two-way or multi-way synchronisation, dynamic composition operators or dynamic creation/activation/orchestration of indexed families of processes, as will be exemplified in the following sections.

3.3 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 [25] to justify the use of finite model-checking on our instantiated model:

Property 1 Let *Sys* be a closed pNet system, with parameters in V, (concrete) parameter domains $\mathcal{D}_V$ and abstract parameter domains $\mathcal{A}_V$, with the following hypothesis:

- 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* [25], or the better-known *testing preorder* [26], is 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 relative 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.

Unfortunately, the examples from this paper are too simple for giving a significant example of abstraction. Rather, let us use an example extracted from a previous case-study of our team modelling the Chilean electronic tax systems [27]. There we were manipulating invoice documents that could typically be described as structures $doct = \langle vendorid, invoiceid, date, content\rangle$ that would be checked by government services against $\langle vendorid, invoiceid\rangle$ records. In the case-study, we were using the abstract domain $doct = \langle vendorid \in [0..2], invoiceid \in [0..2]\rangle$ as an abstract interpretation preserving all safety properties involving, at most, two invoice documents.

In cases where the instantiated variables are parameters of the system topology, then the previous result does not apply. However, 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. As an example, one could check safety properties involving Philo [1] and Fork [2] in the philosopher system, using an abstract domain for indexes defined as $\{\{1\}, \{2\}, \{others\}\}$. However, this will not prove that such a property holds for a system with an arbitrary number of philosophers.

[1] Cleaveland and Riely [25] was using a slightly relaxed condition called "galois insertions".

4 Behavioural models for distributed applications

In this section, we apply the pNets model to four examples, starting with distributed active objects. Then, we successively define a hierarchical component model and enrich it with non-functional controllers. We finally merge the previous concepts to get a modelisation of GCM/ProActive distributed components.

4.1 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 [14, 15] 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 special attention paid 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 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.

Figure 4 illustrates the structure of the pNets expressing an asynchronous communication between two 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 from 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.

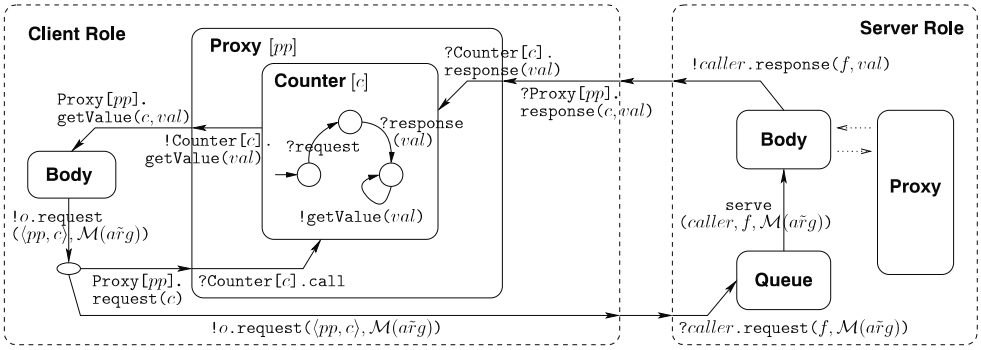


Fig. 4 Communication between two active objects

Nevertheless, for a reasonable subset of ProActive programs, we have the following result [15]:

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

More precisely, this result applies to most standard ProActive programs, with either FIFO or user-defined request selection policies, but with no usage of reflexivity in the Java code. It does not handle first-class futures, nor group communication, but extensions are currently studied. The strongest limitations come from the imprecision of the static analysis mentioned above, and from some difficulties when dealing with some of the Java constructs, like arrays of active objects.

4.2 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 active objects are statically identified by components, 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, respectively). This definition extends slightly those used in Fractal or in the GCM.

Definition 12 Component structure:

- A component C is a tuple $\langle V, \Sigma_V, \tilde{EI}, \xi\xi\rangle$, where V is a set of parameters, Σ_V a term algebra, $\tilde{EI}$ is the set of external interfaces of C, and ξ the content.
- An interface type $Ity = \langle\tilde{\mathcal{M}}\rangle$ is a set of methods $\mathcal{M} = \langle T, name, \tilde{A}\rangle$ with T its return type, and each $A = \langle T_A, name\rangle$ a typed argument.
- An interface is a tuple $Itf = \langle name, Ity, \kappa, \nu, \rho\rangle$, where Ity is its interface type, κ is the Fractal contingency (mandatory or optional), ν is the interface multiplicity, and ρ the interface role (either required or provided).
- The content of a composite component is a tuple $\xi = \langle\tilde{IItf}, \tilde{SubC}, \tilde{B}\rangle$, where $\tilde{IItf}$ is the set of internal interfaces, $\tilde{B}$ the set of bindings. $\tilde{SubC}$ is the set of parameterised subcomponents $SubC = \langle v, C\rangle$, with $v \in V$ a parameter and C a component.
- A binding B is a pair $\langle C_1.cItf, C_2.sItf\rangle$ with $C_i = \ self \mid sub\, C\left[expr \in \Sigma_V\right]$ 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 will be used to generate a pLTS representing the primitive behaviour. We also leave undefined the algebra Σ_V, which is used to build expressions for specifying indexes within the parameterised structure; it will depend on the domains used for the parameter 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 global actions pA_G and hole sorts $\tilde{O}_J$ of the pNets are sets of actions of the form $[!|?]\, C_i.Itf.\mathcal{M}(a\tilde{r}g)$ for invoking/serving a method $\mathcal{M}$ with each argument $arg \in \Sigma_{T_{arg},V}$.
- It has one parameterised synchronisation vector for each binding in $\tilde{B}$.

We have shown examples of proofs using such models in [28]. 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, μCRL or πADL. 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.

4.3 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 on 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. 5) 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.

A BC pLTS (see Fig. 5) 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.

Alternatively, this could have been encoded using one state in the pNet transducer for each configuration of the bindings. However, this would require many transducer states, corresponding to all combinations of states of all controllers. Our approach is equivalent and more modular.

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. 6. 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. 7).

In this drawing, the behaviour of subcomponents is represented by the box named $SubC^{\mathbf{k}}$. 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 dotted 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 model construction is applied bottom-up through the hierarchy. The generated model is powerful enough to prove

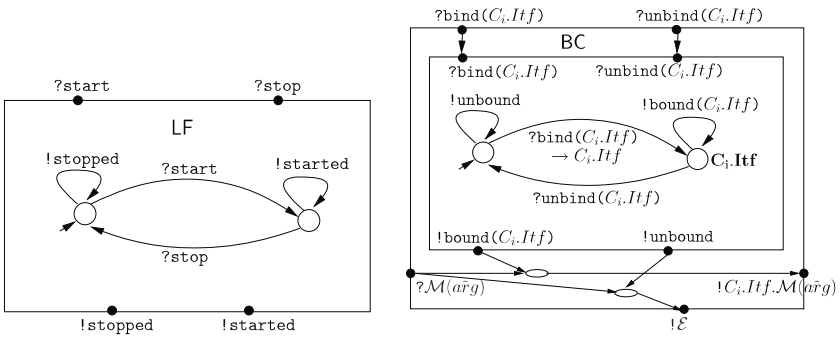


Fig. 5 pLTS of fractal life cycle and binding controllers

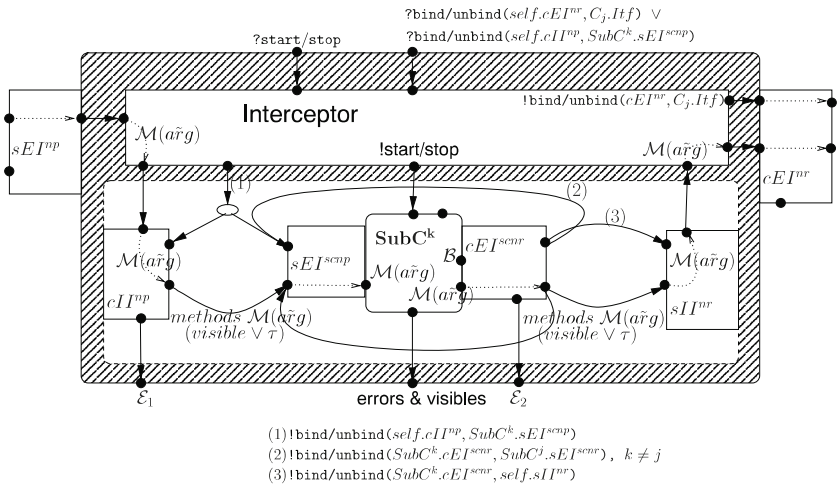


Fig. 6 Synchronisation pNet for a Fractal composite component

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 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 $\surd$. We build a *non-deployed 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 non-deployed model with the deployment sequences. It allows us to check for correctness of deployment specifications, which is characterised by reachability of $\surd$.

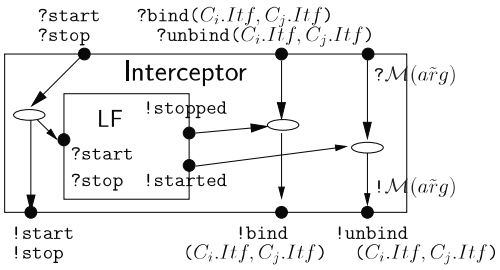


Fig. 7 Interceptor for synchronous Fractal components

- [*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.

4.4 Distributed components: GCM/ProActive

In the Section 4.1 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 4.3 with this communication protocol in order to model GCM/ProActive components.

4.4.1 Primitive components

Let us recall the principle of asynchronous communication between two GCM/ProActive primitive components, inherited from ProActive (see Fig. 4). 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. 4 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).

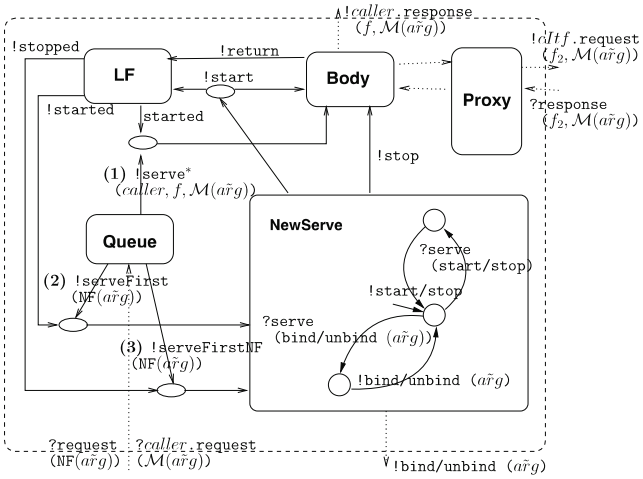


Fig. 8 Behaviour model for a GCM/ProActive primitive

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. 6. The resulting pNet is drawn in Fig. 8. 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. 8). 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.

4.4.2 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 9 shows the model of the membrane, that is similar to the *interceptor* from Fig. 6, though more complex. The membrane model is created from the description of the composite (given by the ADL). The proxy in Fig. 9 is the same as the one presented in Fig. 4. In this case, the proxy is in charge of forwarding the value of the future by receiving the value in action `?response(..)` and forwarding in action `!response(..)`. 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

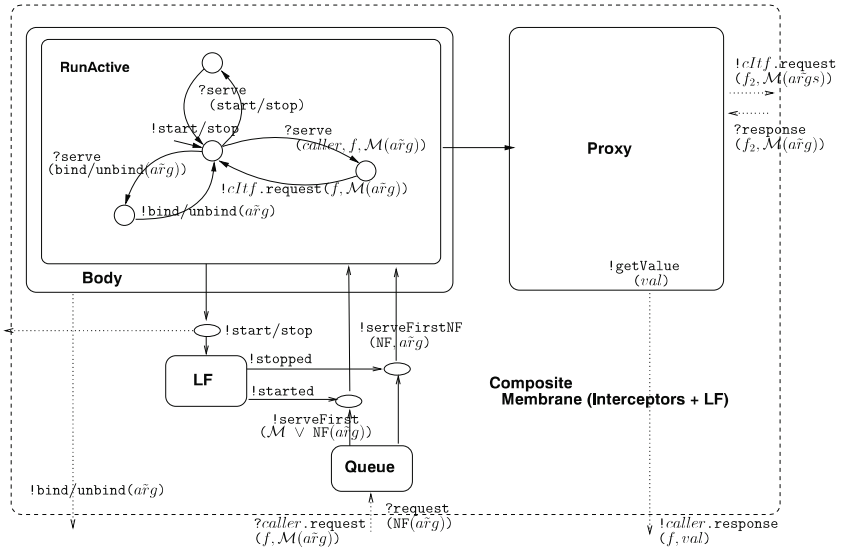


Fig. 9 Behaviour of a composite membrane

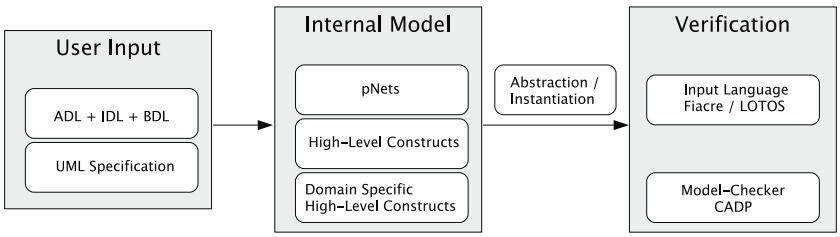


Fig. 10 The VERCORS architecture

of the components. Our model is expressive enough to reflect this property.

The modelisation here does not handle the mechanisms for *first-class futures*, which require specific controllers for storing and updating chains of future proxies through several components. We are working on this extension. This is important both for reflecting realistic applications that use this mechanism for efficiency and because it has significant behavioural impact: deadlocks may be different when you allow first class futures.

5 Platform overview

We present below a high-level view of the Vercors platform and the properties we are able to verify; the interested reader could refer to [17] for further details. 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 4. The platform is presented through the classical problem of a bound buffer with one consumer and one producer.

Figure 10 gives a snapshot of the platform. In the next subsections, we shall describe in detail its three parts: the input from the user (Section 5.1), the behavioural model (Section 5.2), and the verification of properties (Section 5.3). We illustrate our platform through the formal verification of the previously outlined case-study.

5.1 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).

Figure 11 shows an example of a producer consumer system. Both the producer and the consumer produce/consume one element at a time. Additionally, the buffer emits an alarm through its interface I_{alarm}, when the buffer is full.

The XML description of the ADL of the producer–consumer example is shown in Fig. 12. It specifies that the system is composed of the composite BufferSystem (line 6), itself described in a separate file (`components/BufferSystem.fractal`), and the primitive Alarm, the implementation of which is the Java class `components.Alarm` (line 15). The BufferSystem receives a paramater (three in our example, line 7) used to initialise the component with the maximal size of the buffer. The BufferSystem also requires an interface named `alarm` of type `components.AlarmInterface` (lines 8 and 9). Alarm provides an interface named `alarm` of type `components.AlarmInterface` (lines 13

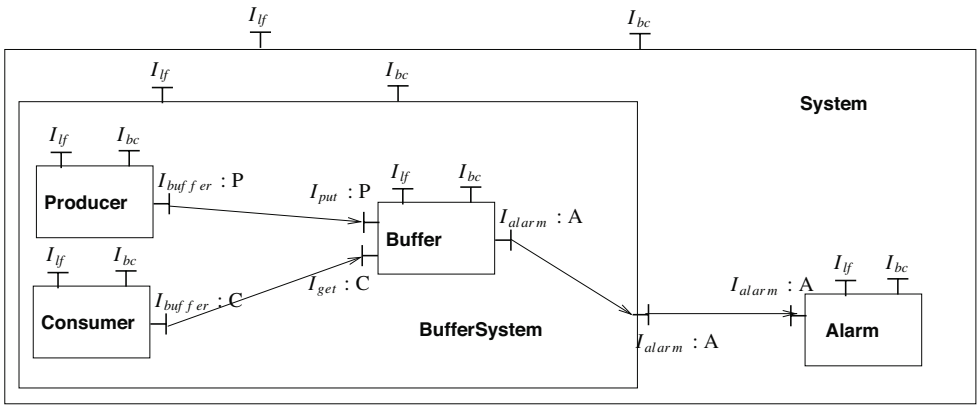


Fig. 11 Consumer–producer example

System.fractal

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE .... >

<definition name="components.System">

   <component name="BufferSystem"
         definition="components.BufferSystem(3)">
      <interface name="alarm" role="client"
            signature="components.AlarmInterface"/>
   </component>

   <component name="Alarm">
      <interface name="alarm" role="server"
            signature="components.AlarmInterface"/>
      <content class="components.Alarm">
         <behaviour file=''AlarmBehav''
               format=''FC2Param''/>
      </content>
   </component>

   <binding client="BufferSystem.alarm"
        server="Alarm.alarm"/>
</definition>
```

Fig. 12 System ADL

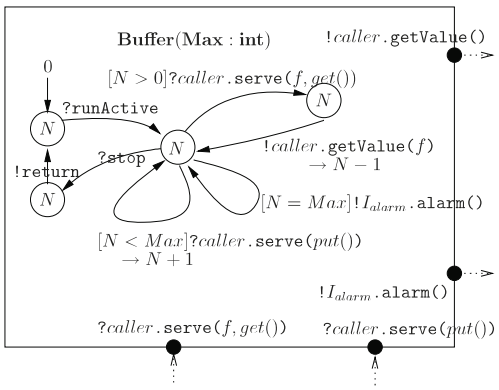


Fig. 13 Buffer behaviour (provided by user)

and 14). 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. Analysing the ADL and the IDL, we are able to build the behavioural model with asynchronous and non-functional controllers of Section 4.4.

Finally, the functional behaviour is given by a BDL, in this case in pNets. An example of a behavioural specification of the `Buffer` is given in Fig. 13. The abstract specification does not consider the values of the elements, but only the amount of elements stored. Therefore, the parameterised automaton has a variable N representing the number of elements stored in the buffer, and the transitions have guards with expressions related to this variable and to a constant Max. In the example, the buffer is instantiated with $Max = 3$ as set in line 7, Fig. 12. Other parameters are: *caller*, representing the reference to the activity (component) that invoked the method call, and f, representing the identifier of the future that is used to send back the response. The buffer also invokes methods on its client interface I_{alarm} in case the buffer is full (action $!I_{alarm}.\texttt{alarm}()$).

Although pNets can be used as a BDL, it is convenient to give a higher-level language to non-expert users. In this vein, we also developed a tool called CTTool [29], 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 smoothly integrate architecture and behaviour specifications for GCM applications, but this is still in progress.

5.2 Internal model

We first automatically build the behavioural model in pNets seen in Section 4. 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 4.2).

We also specified a model for Fractal's binding and life-cycle controllers. Those two controllers allow us to model the deployment and some basic reconfigurations of the system. In our case, checking the safeness of these can be done statically by building the *Static*, *Deployment* or *Reconfiguration* automata of Section 4.3.

In practice, the user of `ADL2N` uses a GUI to specify at the same time the methods that will be visible, the arguments that are significant and their finite instantiation. The visibility of methods and the abstraction (see Section 3.3) 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. The output of `ADL2N` is the pNets behavioural model of Section 4.3 with the above abstractions and with the selected actions hidden.

5.3 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 features 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 examples of verification for various usage scenarios.

Deployment In GCM/ProActive, method calls are asynchronous, and there may be delays between the request for a non-functional method and its treatment. So checking the execution of a control operation must be based on the observation of its application on the component, rather than the arrival of the request.

One of the interesting properties is that the start operation, which is hierarchical, occurs during the deployment; i.e. that the component and all its subcomponents are at some point started. This property can be expressed as the (inevitable) reachability of the start signal in the static automaton of `System`, for all the possible executions, where `name = {System, BufferSystem, Alarm, Buffer, Consumer, Producer}`. This can be translated into a μ-calculus formula and verified in CADP.

Pure-functional properties The classical interesting properties concern the behaviour of the system after its deployment, at least while there are no reconfigurations. For instance, in the example, we would like to prove that a request for an element from the queue is eventually served, i.e. that the element is eventually obtained. This is proved to be true in CADP by model-checking the global state-space.

Functional properties under reconfigurations Our approach enables the verification of properties not only after a correct deployment, but also after and during reconfigurations. For instance, the pure-functional property above becomes false if we stop the producer since, at some point, the buffer will be empty, and the consumer will be blocked waiting for an element. However, if the producer is restarted, the consumer will eventually receive an element and the property should become true again. In other words, we can check that, if the consumer requests an element, and then the producer is stopped, if the producer is started again, the consumer will get the element requested.

For proving this kind of property, the static automaton is not sufficient; we need a behavioural model containing the required reconfiguration operations. We add to the component network a *reconfiguration controller* (Fig. 14): its initial state corresponds to the deployment phase and the next state corresponds to the rest of the life-cycle in which reconfigurations are enabled. This state change is fired by the successful termination of the deployment ($\surd$). For the property stated above, the reconfigurations `?stop(Producer)` and `?start(Producer)` are left visible.

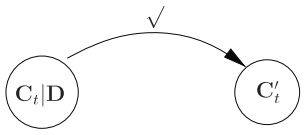


Fig. 14 Synchronisation product supporting further reconfigurations

Asynchronous behaviour properties Let us now focus on a property specific to the asynchronous aspect of the component model. The communication mechanism in GCM/ProActive allows any future, once obtained, to be updated with the associated value, provided that the corresponding method is served and terminates correctly; binds, unbinds or stops operation cannot prevent this. For example, if the consumer is unbound after a request, it gets anyway the response, even if the link is then unbound or the component stopped. We are able to verify this in our behavioural models.

6 Conclusion and perspectives

This article defines the pNets formalism, a parameterised and hierarchical extension of LTSs. pNets have a tree-structure in terms of networks of synchronisation vectors, and a very high expressivity through the use of parameters at both LTS and network levels. This formalism is used to represent the behavioural semantics of distributed systems. It provides a compact and well-defined intermediate format for connecting code analysers or code generators with model-checking or equivalence engines.

In addition to the formal definition of pNets, our contribution is:

- Four scenarios demonstrating the usage of pNets. We generate behavioural models for active objects, hierarchical components, hierarchical components with non-functional controllers and finally asynchronous hierarchical components with non-functional controllers.
- A short description of our verification platform Vercors, in which we use pNets as the pivot format for analysis, abstraction, verification and code-generation tools. We show the results of model construction and analysis of temporal logic properties for a simple case-study.

The pNets format is lower-level, and more versatile, than other models used in existing verification toolsets. Many tools rely on specific synchronisation and communication mechanisms, like the LOTOS-like parallelism in the CADP toolset, channels in Promela or Petri nets in other cases. In contrast, the low-level

primitives of pNets (LTS + synchronisation vectors) are able to represent many possible mechanisms, as demonstrated by the four applications in this article.

Another important trade-off is between parameterised representations (close to developers code) and lower-level explicit-state encodings that are required by 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.

The Vercors platform (editor, generation, instantiation and conversion tools) and a large-scale case-study are available at our website.[2] These tools currently allow to build behavioural models for synchronous Fractal components with partial support for non-functional controllers. They are interfaced with the explicit-state verification toolset CADP.

We are currently working on the controller generation for the GCM/ProActive asynchronous components, including the handling of multicast/gathercast communications, of transparent futures and of component reconfiguration. A specific concern is the encoding of request queues; a direct representation with pNets is possible but would be very expensive in term of state/transition complexity. We are looking for a specific parametric representation coupled with a specialised "infinite-state" engine.

Our main application context is the GCM component model and its reference implementation within the Java/ProActive library. However, static analysis of Java/ProActive code is intrinsically imprecise, making the generation of pNet models difficult. 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 and parameterised component topologies. This language will be used as an input for the Vercors platform, but also for tools that will generate "correct by construction" Java code.

References

1. Bruneton E, Coupaye T, Leclercp M, Quema V, Stefani J (2004) An open component model and its support in java. In: 7th int symp on component-based software engineering (CBSE-7), LNCS, vol 3054. Springer
2. CoreGRID, Programming Model Institute (2006) Basic features of the grid component model (assessed). Technical report, Deliverable D.PM.04. http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf
3. Milner R (1989) Communication and concurrency. Prentice Hall, Englewood Cliffs ISBN 0-13-114984-9
4. Bergstra J, Pose A, Smolka S (2001) Handbook of process algebra. North-Holland, Amsterdam
5. Arnold A (1994) Finite transition systems. Semantics of communicating sytems. Prentice-Hall, Englewood Cliffs
6. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes. Inf Comput 100(1):1–77
7. Garavel H, Lang F, Mateescu R, Serve W (2007) CADP 2006: a toolbox for the construction and analysis of distrbuted processes. In: CAV 2007 conference. Berlin, Germany
8. Garavel H, Lang F (2002) NTIF: a general symbolic model for communicating sequential processes with data. In: Proceedings of FORTE'02 (Houston), LNCS, vol 2529. Springer
9. Roscoe A (1994) Model-checking CSP. In: A classical mind, essays in honour of C.A.R. Hoare. Prentice-Hall, Englewood Cliffs
10. Scattergood J (1998) The semantics and implementation of machine-readable CSP. PhD thesis, Oxford Un. Computing Laboratory
11. Magee J, Kramer J (2006) Concurrency: state models and java programs, 2nd edn. Wiley, New York
12. Poizat P, Royer J, Salaun G (2006) Bounded analysis and decomposition for behavioural descriptions of components. In: FMOODS, LNCS, vol 4037. Springer
13. Poizat P, Royer J (2006) A formal architectural description language based on transition systems and modal logic. J Univers Comput Sci 12(12):1741–1782
14. Barros T, Boulifa R, Madelaine E (2004) Parameterized models for distributed Java objects. In: Forte'04 conference. LNCS, vol 3235. Springer, Madrid
15. Boulifa R (2004) Génération de modèles comportementaux des applications réparties. PhD thesis, University of Nice - Sophia Antipolis – UFR Sciences
16. Barros T, Henrio L, Madelaine E (2005) Behavioural models for hierarchical components. In: Godefroid P (ed) Model checking software, 12th int SPIN workshop, LNCS, vol 3639. Springer, San Francisco
17. Barros T (2005) Formal specification and verification of distributed component systems. PhD thesis, University of Nice - Sophia Antipolis
18. Caromel D, Delbé C, di Costanzo A, Leyton M (2006) ProActive: an integrated platform for programming and running applications on grids and P2P systems. Comput Methods Sci Technol 12(1):69–77
19. Caromel D, Henrio L, Serpette B (2004) Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, New York, pp 123–134
20. Caromel D, Henrio L (2005) A theory of distributed object. Springer, Heidelberg
21. Lin H (1996) Symbolic transition graph with assignment. In: Montanari U, Sassone V (eds) CONCUR '96, LNCS, vol 1119. Pisa, Italy
22. Lakas A (1996) Les Transformations Lotomaton: une contribution à la pré-implémentation des systèmes Lotos. Ph.D. thesis, Univ. Paris VI
23. Najm E, Lakas A, Serouchni A, Madelaine E, de Simone R (1992) ALTO: an interactive transformation tool for LOTOS and LOTOMATON. In: Bolognesi T, Brinksma E, Vissers C (eds) Third lotosphere workshop and seminar, Pisa
24. Madelaine E (1992) Verification tools from the CONCUR project. In: Rozenberg G (ed) EATCS Bull, vol 47. B. Rovan, Bratislava

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

25. Cleaveland R, Riely J (1994) Testing-based abstractions for value-passing systems. In: CONCUR'94, LNCS, vol 836. Springer, Heidelberg
26. Cleaveland R, Hennessy M (1993) Testing equivalence as a bisimulation equivalence. Form Asp Comput 5:1–20
27. Attali I, Barros T, Madelaine E (2004) Formalisation and proofs of the chilean electronic invoices system. In: Proc. of the XXIV international conference of the Chilean computer science society (SCCC'04). IEEE, Arica
28. Barros T, Cansado A, Madelaine E, Rivera M (2006) Model checking distributed components: the Vercors platform. In: 3rd workshop on formal aspects of component systems. ENTCS, Prague
29. Ahumada S, Apvrille L, Barros T, Cansado A, Madelaine E, Salageanu E (2007) Specifying fractal and GCM components With UML. In: Proc of the XXVI international conference of the Chilean computer science society (SCCC'07). IEEE, Iquique