# Parameterized Models for Distributed Java Objects

Tomás Barros, Rabéa Boulifa, and Eric Madelaine

INRIA Sophia-Antipolis
{tomas.barros,rabea.boulifa,eric.madelaine}@sophia.inria.fr

**Abstract.** Distributed Java applications use remote method invocation as a communication means between distributed objects. The ProActive library provides high level primitives and strong semantic guarantees for programming Java applications with distributed, mobile, secured components. We present a method for building finite, parameterized models capturing the behavioural semantics of ProActive objects. Our models are symbolic networks of labelled transition systems, whose labels represent (abstractions of) remote method calls. In contrast to the usual finite models, they encode naturally and finitely a large class of distributed object-oriented applications. Their finite instantiations can be used in classical model-checkers and equivalence-checkers for checking temporal logic properties in a compositional manner. We are building a software tool set for the analysis of ProActive applications using these methods.

## 1 Introduction

We aim at developing methods for the analysis and verification of behavioural properties of distributed applications, that would be applicable in automatic tools, on a real language. At the heart of such tools lie sophisticated static analysis techniques, and abstraction principles, that enable the generation of finitary models of the behaviour from the source code of the application. A good candidate as a behavioural model would be a process algebra with at least value-passing features, or even encoding dynamic process and channel creation and reconfiguration. Still, despite the very important development in the last 20 years of value-passing and high-order process theories, most of them are just too expressive to be subject to decision procedures, and would not give us models and algorithms usable in practical tools.

At the same time, a number of analysis tools, model-checkers, equivalence checkers have been developed, using input formats, in their respective areas; that have some of the desired features for our work. For example the Promela language, input of the SPIN model-checker, can describe value-passing processes and channels with data values of simple types or the NTIF format [1] that encodes the sophisticated communication between E-LOTOS processes. However, few of them include compositional structures that would allow to take advantage of the congruence properties of process algebra models. Outside the value-passing area, it is worth citing the seminal work by Arnold [2], and the MEC language and analysis tool, that permits a direct and finite representation of the synchronisation constraint between processes.

Our approach aims at combining the value-passing and the synchronisation product approaches. We define a model featuring parameterized processes, value-passing communication, behaviours expressed as symbolic labelled transition systems, and data-values of simple countable types. We have developed a graphical language close to this model, that is powerful and natural enough to serve as a specification language for distributed applications [3]. We argue that the same model is adequate as the target for automatic model generation for distributed applications. As an illustration of the approach, we define the generation procedure [4] for the Java/ProActive framework. One key feature is that the design of the model ensures that it can be automatically and finitely produced from an "abstract" version of the application code, in which data have been abstracted to simple types. Then, given a finite instantiation of the variables in the model, we have an automatic procedure producing a (hierarchical) finite instantiation of the model, suitable for use e.g. in a standard model-checker.

Our method can be applied in the following way: starting with the source code of a real application, the developer would specify an abstraction of its data-types, and transform his code accordingly. The work in the Bandera tool set [5] shows how this step can be largely assisted by the tool. At this level, the design of the abstraction can be tuned specifically to the properties one wishes to verify, in order to reduce the size of the generated models. From this abstract code, static analysis techniques, plus our model generation procedure, produces automatically a parameterized network. Then the developer, for each property he wants to prove, will produce a finite network, using a notion of instantiation that is again an abstract interpretation in the style of [6], before checking the property (or its corresponding instantiation) with a model-checker. The instantiation could even be performed on-the-fly, if the checker offers this possibility. The properties can be themselves specified as parameterized scenarios in our graphical language, or as parameterized formulas in a temporal logics.

In section 2, we define parameterized labelled transition systems and synchronisation networks, their instantiations to pure LTS and networks, and the corresponding synchronisation product. Then we sketch a generic way of defining finite instantiations as abstract interpretations of the parameterized models. In section 3, we specialise this model for representing the behaviours of Java distributed applications built using the ProActive framework, and give an algorithm for computing the models from static analysis of the code. In section 4, we give an example of model generated for a small ProActive application. Finally, we conclude about our work and future research directions.

## 2 Parameterized Models

We give the behavioural semantics of programs in terms of labelled transition systems. We specify the composition of LTSs by synchronisation networks [2], and give their semantics in term of a synchronisation product.

## 2.1 Theoretical Model

We start with an unspecified set of communications **Actions** *Act*, that will be refined later.

We model the behaviour of a process as a Labelled Transition System (LTS) in a classical way [7]. 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* $S$ *is the set of states,* $s_0 \in S$ *is the initial state,* $L \subseteq Act$ *is the set of labels,* $\rightarrow$ *is the set of transitions:* $\rightarrow \subseteq S \times L \times S$. *We write* $s \xrightarrow{\alpha} s'$ *for* $(s, \alpha, s') \in \ \rightarrow$.

Then we define **Nets** in a form inspired by [2], that are used to synchronise a finite number of processes. A Net is a form of generalised parallel operator, and each of its arguments are typed by a **Sort** that is the set of its possible observable actions.

**Definition 2 Sort**. *A Sort is a set* $I \subseteq Act$ *of actions.*

A LTS $(S, s_0, L, \rightarrow)$ can be used as an argument in a Net only if it agrees with the corresponding Sort ($L \subseteq I_i$). In this respect, a Sort characterises a family of LTSs which satisfy this inclusion condition.

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 open Lotos expressions of [8]. They are encoded as LTSs which labels are synchronisation vectors, each describing one particular synchronisation of the process actions:

**Definition 3 Net**. *A Net is a tuple* $< A_G, I, T >$ *where* $A_G$ *is a set of global actions,* $I$ *is a finite set of Sorts* $I = \{I_i\}_{i=1,\ldots,n}$, *and* $T$ *(the transducer) is a LTS* $(T_T, s_{0_t}, L_T, \rightarrow_T)$, *such that* $\forall \overrightarrow{v} \in L_T$, $\overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >$ *where* $l_t \in A_G$ *and* $\forall i \in [1, n], \alpha_i \in I_i \cup \{idle\}$.

We say that a Net is *static* when its transducer vector contains only one state. Note that a 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.

The semantics of the Net construct is given by the synchronisation product:

**Definition 4 Synchronisation Product**. *Given a set of LTS* $\{LTS_i = (S_i,\ s_{0_i},\ L_i, \rightarrow_i)\}_{i=1\ldots n}$ *and a Net* $< A_G, \{I_i\}_{i=1\ldots n}, (S_T, s_{0_T}, L_T, \rightarrow_T) >$, *such that* $\forall i \in [1, n], L_i \subseteq I_i$, *we construct the product LTS* $(S,\ s_0,\ L,\ \ \rightarrow)$ *where* $S = S_T \times \prod_{i=1}^{n}(S_i)$, $s_0 = s_{0_T} \times \prod_{i=1}^{n}(s_{0_i})$, $L = A_G$, *and the transition relation is defined as:*

$\rightarrow \overset{\text{def}}{=} \{s \xrightarrow{l_t} s' |\ s =< s_t, s_1, \ldots, s_n >, s' =< s'_t, s'_1, \ldots, s'_n >,$
$\exists\ s_t \xrightarrow{\overrightarrow{v}} s'_t \in \rightarrow_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >,\ \forall i \in [1, n],\ (\alpha_i \neq idle \wedge s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i$
$) \vee (\alpha_i = idle \wedge s_i = s'_i)$

Note that the result of the product is a LTS, which in turn can be synchronised with other LTSs in a Net. This property enables us to have different levels of synchronisations, i.e. a hierarchical definition for a system.

Next, we introduce our parameterized systems which are an extension from the above definitions to include parameters. These definitions are connected to the semantics of Symbolic Transition Graph with Assignment (STGA) [9].

Parameterized Actions have a rich structure, for they take care of value passing in the communication actions, of assignment of state variables, and of process parameters. In order to be able to define variable instantiation as an *abstraction* of the data domains (in the style of [6]), we restrict these domains to be **simple (countable) types**, namely: booleans, enumerated sets, integers or intervals over integers and finite records, arrays of simple types.

**Definition 5 Parameterized Actions** *are: $\tau$ the non-observable action, $\mathcal{M}$ encoding an observable local sequential program (with assignment of variables), $?m(P, \overline{x})$ encoding the reception of a call to the method $m$ from the process $P$ ($\overline{x}$ will be affected by the arguments of the call) and $!P.m(\overline{e})$ encoding a call to the method $m$ of a remote process $P$ with arguments $\overline{e}$.*

A parameterized LTS is a LTS with parameterized actions, with a set of parameters (defining a family of similar LTSs) and variables attached to each state. Parameters and variables types are simple. Additionally, the transitions can be guarded and have a resulting expression which assigns the variables associated to the target state:

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

*$K = \{k_i\}$ is a finite set of parameters,*

*$S$ is the set of states, and each state $s \in S$ is associated with a finite set of variables $\overrightarrow{v_s}$,*

*$s_0 \in S$ is the initial state,*

*$L = (b, \alpha(\overrightarrow{x}), \overrightarrow{e})$ is the set of labels (parameterized actions), where $b$ is a boolean expression, $\alpha(\overrightarrow{x})$ is a parameterized action, and $\overrightarrow{e}$ is a finite set of expressions.*

*$\rightarrow \ \subseteq S \times L \times S$ is the set of transitions:*

**Definition 7 Parameterized Sort**. *A Parameterized Sort is a set $pI$ of parameterized actions.*

**Definition 8** *A* **pNet** *is a tuple $< pA_G, H, T >$ where: $pA_G$ is the set of global parameterized actions, $H = \{pI_i, K_i\}_{i=1..n}$ is a finite set of holes (arguments). The transducer $T$ is a pLTS $(K_G, S_T, s_{0_T}, L_T, \rightarrow_T)$, such that $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$ where $l_t \in pA_G$ , $\alpha_i \in pI_i \cup \{idle\}$ and $k_i \in K_i$.*

The $K_G$ of the transducer is the set of global parameters of the pNet. Each hole in the pNet has a sort constraint $pI_i$ and a parameter set $K_i$, expressing that

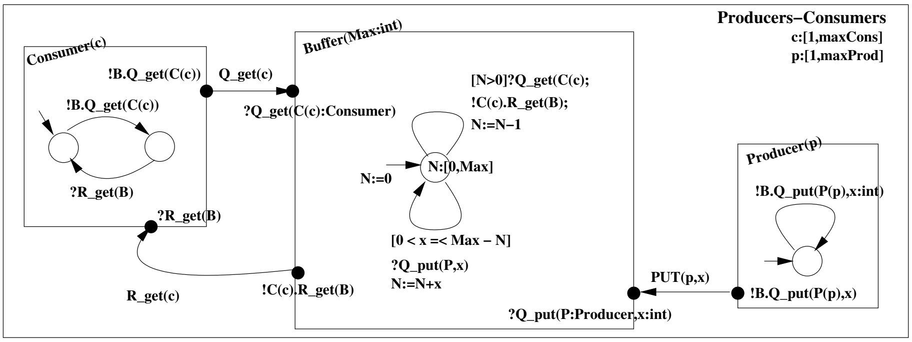


**Fig. 1.** Graphical representation of parameterized networks

this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation. In a synchronisation vector $\overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$, each $\alpha_i^{k_i}$ corresponds to the $\alpha_i$ action of the $k_i$-nth corresponding argument LTS.

In the framework of this paper, we do not want to give a more precise definition of the language of parameterized actions, and we shall not try to give a direct definition of the synchronisation product of pNets/pLTSs. Instead, we shall instantiate separately a pNet and its argument pLTSs (abstracting the domains of their parameters and variables to finite domains, before instantiating for all possible values of those abstract domains), then use the non-parameterized synchronisation product (Definition 4). This is known as the early approach to value-passing systems [7, 10].

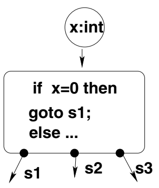
## 2.2 Graphical Language

We provide a graphical syntax for representing *static* Parameterized Networks, that is a compromise between expressiveness and user-friendliness. We use a graphical syntax similar to the Autograph editor [11], augmented by elements for parameters and variables: a *pLTS* is drawn as a set of circles representing states and edges representing transitions, where the states are labelled with their set of variables ($\overrightarrow{v_s}$) and the edges are labelled by $[b]\ \alpha(\overrightarrow{x}) \rightarrow \overrightarrow{e}$ (see Definition 6).

An *static pNet* is represented by a set of boxes, each one encoding a particular Sort of the pNet. These boxes can be filled with a pLTS satisfying the Sort inclusion condition. Each box has ports on the border, represented by bullets, each one encoding a particular parameterized action of the Sort.

Fig. 1 shows an example of such a parameterized system. It is a simple consumer-producer system with a single buffer and an arbitrary number of consumers and producers. In Fig. 1, the right-most link is a communication name **Q_put** from process **Producer(p)** to the buffer **B**, carrying a value **x:int** that the developer has chosen to observe as the event **PUT(p,x)**.

The edges between ports in Figure 1 are called links. Links express synchronisation between internal boxes or to external processes. Each link encodes a transition in the Transducer LTS of the *pNet*.



The sequential code encoding the control and data-flow within a process is carried by macro-transitions, with multiple output states. We restrict them to sequential programs without communication events. This way, we avoid duplicating code in sequential transitions and at the same time avoid the extra interleaving that would be created by macro-transitions containing visible events.

We have used this language extensively in [3] to specify and verify a large distributed system from a realistic case study.

### 2.3 Instantiations as Abstractions

From a parameterized network, we want to construct abstract models, with parameters in abstract domains simpler than the original (concrete) domains. Ultimately the parameter domains should be finite, allowing us to use standard model-checking tools on the abstract model. And we want this abstraction to be consistent, in the sense that some families of properties (typically reachability) are preserved by the abstraction. Thus from the reachability of some abstract event in the abstract domain, we can conclude to the reachability of some concrete representative of this event in the original model.

In a slightly different settings, [6] have shown how to define abstractions on value domains, in such a way that they induce safe abstractions on value-passing processes (preserving safety and liveness properties). We shall use a similar construction to define instantiations as safe abstractions of our simple data types: an instantiation is a partition (a total subjection) from a simple data type onto an abstract domain; lifting the instantiation to sets of values yields a Galois connection.

## 3 Application: Models for Distributed Active Objects

We now specialise our parameterized models, for representing the behaviour of distributed applications. We choose a specific framework providing high-level distribution and communication primitives for distributed objects, namely the ProActive library. ProActive is also endowed with a formal semantics, and the library services provide strong guarantees on the communication mechanism, that helps a lot in defining our model generation method.

It should be clear that our parameterized models could also be used for other languages or other frameworks. However, providing a similar work for languages with weaker semantical properties (like Java with standard RMI, or C with basic sockets) would definitely be more difficult, and the various properties of our approach (finiteness, abstraction, compositionality) would not be guaranteed.

### 3.1 Java and ProActive

*ProActive* [12] is a pure Java implementation of distributed active objects with asynchronous remote method calls and replies by means of future references. A distributed application built using ProActive is composed of a quantity of active objects (or activities), each one having a distinguished entry point, the *root*, accessible from anywhere. All the other members of an active object (they are called *passive objects*) can not be referenced directly from outside. Each active object owns its own and unique thread of control and the programmer decides the order to serve (or not) incoming calls to its methods. Each active object has a pending queue where are dropped the incoming requests to be served by the control thread. The requests are done via a *rendez-vous* phase so there is a guaranty of delivery and a conservation of the order of incoming calls. The responses (when relevant) are always asynchronous with replies by means of future references; their synchronisation is done by a mechanism of *wait-by-necessity*.

*ProActive* provides primitives to dynamically create and migrate active objects. Dynamic creation is naturally represented in our parameterized models. Migration is not treated in this work: the semantics of *ProActive* ensures transparent active object migration and remote creation.

### 3.2 Data Abstraction

The aim in this work is to generate parameterized models encoding the behaviour of ProActive distributed objects. The events that we want to observe in these models are naturally the communication between activities, plus eventually specific local events that the user will specify.

Being interested in automatic procedure for generating finitely representations of the behaviours, and working with a real language, we have a problem with the representation of potentially infinite data objects (including user-defined classes). So we require that the source code be first transformed by abstraction of the data-types of the application into the "simple types" of our model.

This transformation cannot be fully automatic, and it will require some input from the user to specify the abstraction of all types in the code (Fig. 2). Furthermore, it will be interesting at this step to abstract away from any data information that would not be significant for the properties that the user wants to prove. It has been shown, e.g. in the Bandera tool [5], how such data abstraction can be implemented as code transformation, either at source code or intermediate code level.

### 3.3 Code Analysis

The generation of our behavioural models from the source code requires sophisticated analysis, starting with usual static analysis functions. Class analysis determines the possible class(es) of each variable in the program, and the possible

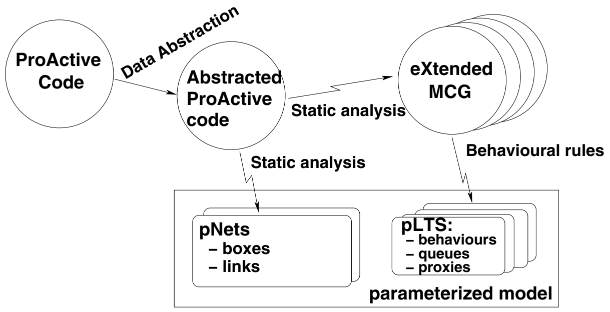


**Fig. 2.** Building models of distributed active objects

identiti(es) of the method called at each invocation point. Then we use Control Flow analysis to compute the Method Call Graph, and Data Flow analysis to track the values of parameters. The adaptation of these methods to ProActive code are not trivial, in particular because the proxy mechanism of the library include a lot of dynamic code generation, and we must emulate its effects during static analysis.

*Language restrictions :* For the sake of this paper, we shall not consider the treatment of exceptions and arrays. Other aspects, such as Java concurrency features (threads, monitors), reflection and dynamic class loading, will not be allowed. These features are important indeed in the implementation of the library, but are not needed for the library user.

The rest of this section describes the steps of the model construction. Starting from the Abstract ProActive code, we build a (static and finite) network description, an eXtended Method Call Graph (XMCG) for each active object class, a local pNet for each activity, and finally a pLTS for each method.

## 3.4 Step 1: Topology and Communication, Extraction of the Global Network

*Static Topology:* In general the topology of a distributed ProActive application is dynamic and unbounded, because active objects can be created dynamically. We compute a static approximation of this topology, in the form of a parameterized network based on: the (finite) set of active object classes, the (finite) set of active object creation points, and the (finite) set of program points where an active object emits a remote request.

*Boxes:* Given a set of active object classes, a set of creation points, we obtain a set of (parameterized) active objects $\{O_i\}$. For each active object creation point, we build a Box $B(O_i(params))$.

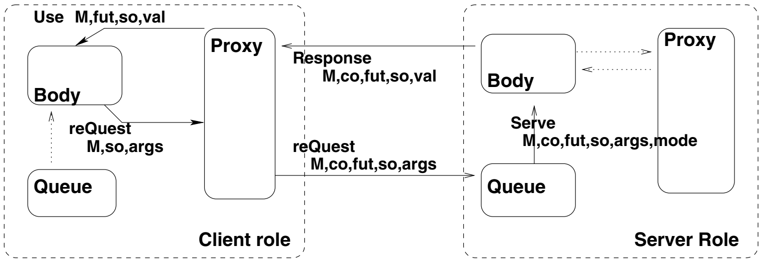


**Fig. 3.** Communication between two activities

*Communication Protocol:* Fig. 3 illustrates the communication corresponding to a request addressed to a remote activity, its processing and the return of its result. 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. It also contains a deep copy of the method's arguments, because there is no sharing between remote activities. Later, the request may eventually be served, and its result value will be sent back and used to update the future value.

*Building the Communication Links:* In the following we denote $m$ a message containing: the (fully qualified) method name, references to the caller, future, callee (parameterized) objects, and either the parameters or the return value of the message.

For each active object class, we analyse the relevant source code to compute its ports and the corresponding links:

1. The set of public methods of the active object main class gives us the set of "receive request" ports $?Q\_m$ of its box, and their response ports (when applicable) $!o.R\_m$.
2. We then identify in the code the variables carrying "active objects" values, using data flow analysis techniques, and taking into account their creation parameters.
3. For each call to a method of a remote object, we add a "send request" port $!o.Q\_m$ to the current box, linked to the corresponding port of the remote object box, and if this method expects a result, a "response" port $?R\_m$.
4. For each local event that we want to observe (e.g. some local method call), we add a "local" port $Call\_m$ to the current box.
5. For each pair of opposite actions such as $?Q\_m$ - $!o.Q\_m$, we build a link (in this case labelled $Q\_m$).

Fig. 4 gives an example of such a pNet, computed from the ProActive code presented in section 4.

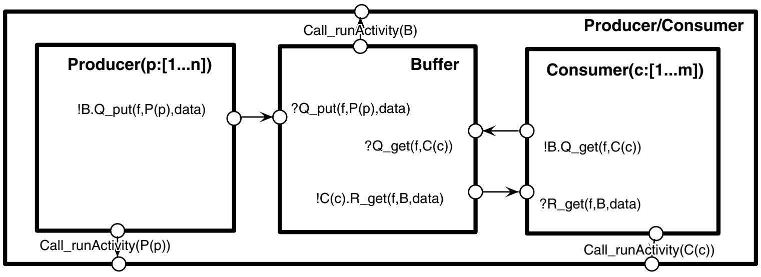


**Fig. 4.** Producers/Consumers: global network

### 3.5 Step 2: eXtended Call Graphs

For each active object class, we define an eXtended Method Call Graph (XMCG) structure containing the results of usual class and control flow analysis (on all classes used by this activity), sequential code encoding the data-flow, and specific constructs relative to the ProActive features namely: active objects, future objects, remote requests and responses, mechanism for the selection of requests in a request queue.

An Extended Method Call Graph is a tuple: $\left\langle M, m_0, V, \xrightarrow{calls}_C, \xrightarrow{succs}_T \right\rangle$ where $M$ is a set of fully qualified methods names, $m_0 \in M$ is the initial method name, $V$ is a set of nodes, and the two transition relations are respectively the inter-procedural (method calls) and intra-procedural (sequential control) transfer relations.

The nodes in $V$ are typed as:

- $ent\ (c, m, args)$ the entry node of method $m \in M$, called by object $c$,
- $call\ (calls)$ encoding method calls (local or remote),
- $pp\ (lab)$ encoding an arbitrary program point with label $lab$,
- $ret\ (val)$ encoding the return node of a method with result value $val$,
- $serve\ (calls, mset, pred, mode)$ encoding the selection of the request $m \in mset$ from the local request queue,
- $use\ (fut, val)$ encoding the point of utilisation of a future value.

All nodes have at most one outgoing transfer edge $< succs(n) = MT, N >$, with $< n, MT, N > \in \xrightarrow{succs}_T$ in which the meta-transition $MT$ is a sequential program with a non-empty set of resulting states $N$.

Call and Serve nodes have a set of nondeterministic outgoing method call edges, $calls(n)$, with $\forall c$ in $calls(n), \exists n'. < n, c, n' > \in \xrightarrow{calls}_C$, each call being either:

- $Remote\ (o.m, args, var, fut)$ for a call to method $m$ of a remote object $o$ through the proxy $fut$,
- $Local\ (o.m, args, var)$ for a call to method $m$ of a local object $o$,
- $Unknown\ (o.m, args, var, fut)$ when it cannot be decided statically whether the call is local or remote.

## 3.6 Step 3: A pNet for the Behaviour of Each Active Object Class

An activity is composed of a body (itself decomposed as we shall see later), a model of its queue, and a model of the proxies (future objects) for each remote method call in its code. The activity model is a pNet synchronising these 3 parts.

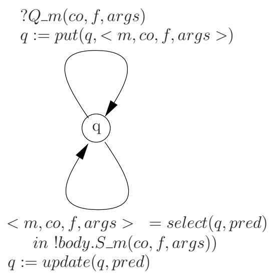
*Methods structure* The essential choice for modelling the behaviour of programs is to get a finite, parameterized representation that take into account the parameters of recursive methods, and the representation of objects in the store. We give the rational for these two points, before describing the procedure for building the model of an activity behaviour.

We choose here to consider each method as a parameterized process, method calls being local synchronisation between specific instantiations of the processes. This simple scheme trivially ensures that we get a finite parameterized network. For each method call and for each return from a call, we generate in the activity pNet a synchronisation event between the 2 processes involved.

*Objects and Stores* There is one common store for each activity. Each object creation point in the code corresponds to a number of objects in the store. If the static analysis can determine precisely this number, we shall use it, otherwise, we index the object by an integer denoting its creation rank.

*Queues* The request queue of an active object runs independently of its body, and is always accepting arriving requests (of correct type) encoded by $Q\_m$ actions. It is synchronised with the object body through the services ($S\_m$) actions produced by rule DO-SERVE.

There are several primitives for selecting requests from the queue. The most frequent way to filter the requests is by the request name, but the programmer can also build more complex selection filters, using the request arguments and/or the sender identity. He can also decide to serve the requests in various order or to do some global treatment on the queue after selection e.g.: *serve Oldest* $(foo)$, *serve Oldest* $(foo(i), i < 10)$, *serve Newest* $(foo, bar)$ or *serve flushNewest* $(Move(x, y))$.



$?Q\_m(co, f, args)$
$q := put(q, < m, co, f, args >)$
q
$< m, co, f, args > \ = select(q, pred)$
$in\ !body.S\_m(co, f, args))$
$q := update(q, pred)$

The various primitives used in a given active object define statically a finite partition of the requests domain. We also collect all selection/operation modes used in the active object code, within: $Modes = \{serve, serveAndFlush\} \times \{Oldest, Newest, Nth\}$.

The idea is that we can now model the queue as a product of independent processes, each encoding one set in the partition, and implementing the relevant operation modes. The model for each part is built in a generic way, as an instantiation of the figure above, in which $m, args, pred$ must be replaced by the corresponding possible values.

The most beneficial optimisation comes from the factorisation in separate queues, and is computed from static analysis by collecting all service primitives used in a specific active object.

Those queues will be coordinated by the automaton encoding the activity behaviour. The benefits come from the fact that we avoid to compute this interleaved product independently from its context.

## 3.7 Step 4: A Model for the Activity Behaviour

The procedure for building the model of a (parameterized) activity is:

1. Compute the set of required static object classes, the XMCG, and the set of object instances in the store (static object creation points with their parameters).
2. Build the activity pNet, with one box for each method in the XMCG, and one-to-one links for method calls. The activity behaviour is functional, because there is a single thread of execution in the activity; this means that only one of those boxes has an initial transition that can be fired alone while others will have to wait to be called.
3. For each method $m$ in the activity, use the Procedure `Method-Behav (m, n, XMCG)`, where $n$ is the entry node of $m$ in the XMCG, to compute the corresponding parameterized LTS.

```
1  Method-Behav ($m,\ n,\ < M,\ V,\ \xrightarrow{calls}_C,\ \xrightarrow{succs}_T >$) :
2    Aut.init = $\{fresh\ s_0\}$; Map = $\emptyset$; $Caller$ = $\emptyset$; ToDo = $\{< n,\ s_0 >\}$
3    while ToDo $\neq\ \emptyset$
4      ToDo.choose $< n,\ s >$
5      if Map($n$) then DO-LOOP-JOIN
6      else
7        select˜$n$ in
8          Ent(c,m,args)                 : DO-ENTRY
9          Call($calls(n)$)                : DO-CALL
10         PP(lab)                       : DO-PP
11         Serve($calls(n)$,mset,pred,mode): DO-SERVE
12         Use(fut,val)                  : DO-FUTURE
13         Ret(val)                      : DO-RETURN
14     unless˜$n$=Ret
15        let $MT, N$ = $succs(n)$ in
16        foreach˜$n_i$ in˜$N$ do
17           fresh˜$s_i$; ToDo.Add $< n_i,\ s_i >$
18        Aut.add˜$s_1 \xrightarrow{MT)} \mathcal{S} = \{s_i\}_i$
```

The `ToDo` set collects all pending MCG nodes, that need to be processed later, with the corresponding LTS node. Map is the mapping between nodes of the XMCG, and the corresponding nodes in the created LTS. For all nodes, MT is a meta-transition encoding the sequential intra-procedural flow. It carries a piece of sequential program (possibly empty) and has a number ($\geq 1$) of target nodes $\mathcal{N}$, from which we create an equal number of LTS nodes $\mathcal{S}$. The $s_1$ LTS node in line 18 is the terminal node created by each of the specific `DO-*` procedures (joining all branches created by the procedure when necessary).

Each of the following node-specific procedures sets the $s_1$ for branching the subsequent transitions, and updates the mapping Map.

*Initialisation :* The *Caller* object is memorised and will be used by the return nodes.

```
DO-ENTRY (c, m, args) =
   $fresh\ s_1$; $Caller$ =~$c$, Map = Map $\cup \{n \mapsto s_1\}$
   Aut.add $s \xrightarrow{?Call\_m(c,args)} s_1$
```

*Sequential nodes :* PP nodes in the XMCG correspond to program points, e.g. a label in the source code corresponding to a loop or a join in the program structure, or a specific passage point that the user has designated. This event can be made visible in the LTS if we need it for a given proof.

```
DO-PP (lab) =
   if $observable$(lab) then Aut.add $s \xrightarrow{Obs(lab)} (fresh\ s_1)$, Map = Map $\cup \{n \mapsto s_1\}$
   else~$s_1$ =~$s$
```

*Call nodes :* A call node has one or more call transitions, each of them can be remote or local, and each of them have an optional [Mix] guard, meaning that its true (remote or local) nature will only be determined at instantiation time.

```
DO-CALL ($calls(n)$) =
  fresh~$s_1$, Map = Map $\cup \{n \mapsto s_1\}$
  foreach $call$ in $calls(n)$
    match $call$ with
      "Remote(o.m,args,var,fut)":  Aut.add $s \xrightarrow{!fut.Q\_m(o,args)} s_1$
      "Local(o.m,args,var)":       Aut.add $s \xrightarrow{!o.Call\_m(args)} (fresh\ s_2)$
      "Unknown(o.m,args,var,fut)": Aut.add $s \xrightarrow{[Mix]!fut.Q\_m(o,args)} s_1$
                                   Aut.add $s \xrightarrow{[Mix]!o.Call\_m(args)} (fresh\ s_2)$
  if local-or-unknown $\cap$ non-void-result($m$) then
     Aut.add~$s_2 \xrightarrow{?Ret\_m(o,val)} (fresh\ s_3) \xrightarrow{var:=\ val} s_1$
```

*Return nodes :* Return nodes are not marked in the node mapping: each return node of a method is treated separately, and generates its own `!Ret` action. The return value *val* is absent for void-result methods.

```
DO-RETURN (val) =
   Aut.add $s \xrightarrow{!Caller.Ret\_m(val)} (fresh\ s_1)$
```

*Request Service nodes :* Serving a request from the local queue is similar to calling a method, but we have to encode the request selection mechanism. Call arcs from a serve node are only of Local type, and for each request $m$ in *mset*, we have such one call arc, expressing one of the possible selection in the queue. The activity model is synchronised with the queue model through the $?S\_m$ message (with guard *pred* if needed); then the method $m$ is started with the arguments gathered from the queue, it waits for the computation to terminate and if necessary sends back the return value to the caller (object $o$, proxy $f$, that were stored with the request).

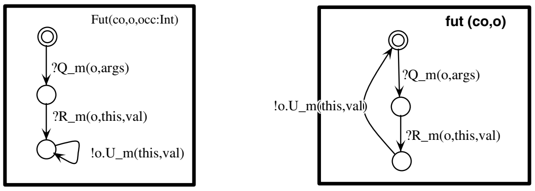


**Fig. 5.** Automata for futures' proxies (without or with recycling)

```
DO-SERVE (calls(n),mset,pred,mode) =
 fresh~$s_1$, Map $=$ Map $\cup \{n \mapsto s_1\}$
 foreach $call$ in $calls(n)$
   match $call$ with "Local(o.m,args,var)"
   if $m \in mset$ do
     fresh~$s_2$, $s_3$
     Aut.add $s \xrightarrow{[pred]?S\_m(f,o,args,mode)} s_2 \xrightarrow{!body.Call\_m(this,args)} s_3$
     if null-result($m$) then Aut.add~$s_3 \xrightarrow{?Ret\_m(val)} s_1$
     else                    Aut.add~$s_3 \xrightarrow{?Ret\_m(val)} (fresh\ s_4) \xrightarrow{!o.R\_m(f,this,val)} s_1$
```

*Loops :* The (Loop-Join rule) applies to all types of nodes that already have been visited. Then the corresponding LTS node is substituted to the current LTS node, eventually creating loops or joins in the LTS.

```
DO-JOIN-LOOP () = $s_1 =$ Map$(n)$
    $Aut.replace(s, s_1)$
```

*Future values and utilisation :* We create a future object at each remote invocation point with a non-void result type. This future object provides the value to its potential use points. Thereby, we have as many "future objects" automatas as invocation points, and we synchronise those with their use points in the future rule. There are cases when static information garanties that a future value is consumed at a particular point, in which case we can recycle the corresponding future object (then a single automaton can be used, instead of a family indexed by its occurence in the store, see Fig. 5).

```
DO-FUTURE (fut,val) =
  Aut.add $s \xrightarrow{?U\_m(fut,val)} (fresh\ s_1)$
  Map $=$ Map $\cup \{n \mapsto s_1\}$
```

## 4 Example

We use here a part of the Producer/Consumer example from section 1 to illustrate the model generation.

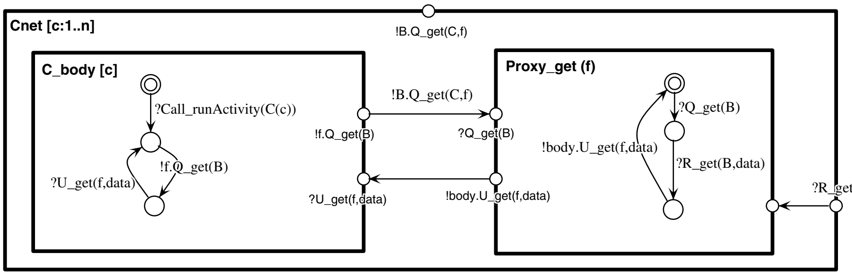


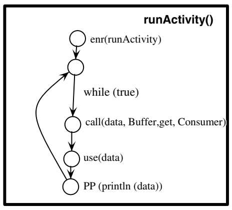
**Fig. 6.** Resulting Consumer model

### 4.1 ProActive Code and Extended MCG

The consumer and corresponding XMCG:

```
    public void runActivity(Body myBody) {

      { ...
        while (true) {
        Type data = Buffer.get();
        System.out.println("The value is " + data);
        }
      }
    }
```

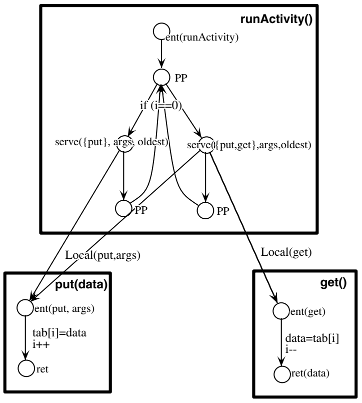


The buffer and corresponding XMCG:

```
    public void runActivity(Body body) {
      int bound;
      Type [] tab;
      {
        while (true) {
        if (bound==0)
                service.serveOldest("put");
         else
                service.serveOldest();
    }}}

    void put (Type data) {
        tab[bound]=data;
        bound++;
         }

    Type get(){
      return(tab[bound--]);}
```



### 4.2 The Generated Nets

We illustrate the model construction with two examples: the Buffer pNet in Fig. 7 illustrates the model of local method calls, and its interaction with the queue, while the Consumer pNet in Fig. 6 illustrates the interaction with a proxy. For each of the methods LTSs, we have applied a simple optimisation after completion of the `Method-Behav` procedure, removing all empty transitions that where not part of a non-deterministic choice (removal of tau-prefix).

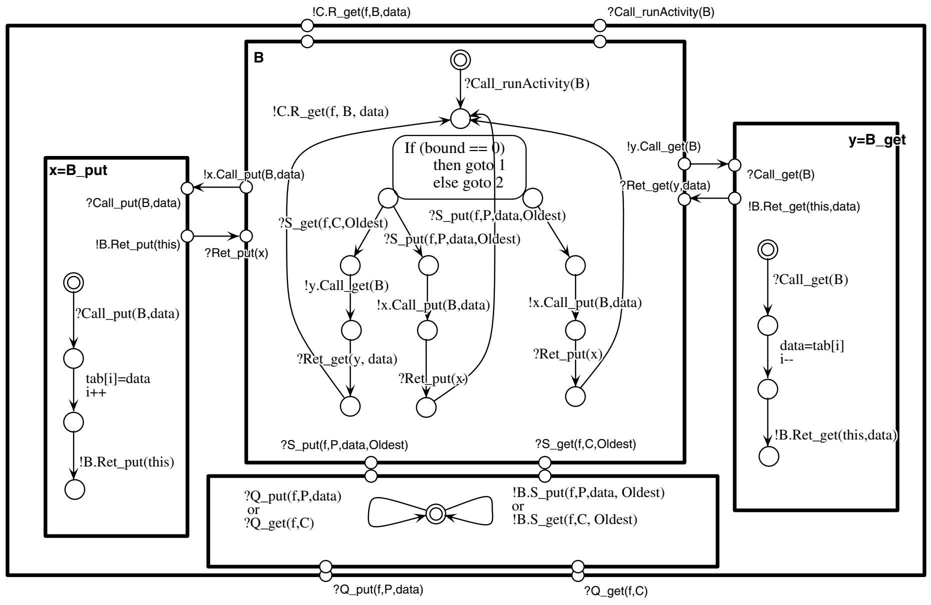


**Fig. 7.** Resulting Buffer model

Fig. 6 shows the pNet modelling the Consumer behaviour. Note that the data accessed by the Consumer is immediately consumed (the `Use` node in the MCG follows immediately the `Request` node). This implies that there can be only one "get" future active at any time, so we use a single future in the proxy instead of an indexed family of futures.

An interesting feature is that the `Q_get` synchronisation between the consumer root and its proxy is directly visible as a message addressed to the buffer process (thanks to the expressivity of synchronisation vectors); this technique enables us to avoid an explicit encoding of a rendez-vous protocol, that would introduce unnecessary interleavings.

## 5 Conclusion and Directions

We have introduced a language to describe the behaviour of communicating distributed systems using parameterized models. The parameters in our model are variables that encode both: data value (such as in the theories of value-passing systems), and process identifiers (such as in the theories of channel-passing systems). We argue that our models are suitable as a specification language for distributed systems behaviour, and for models resulting from static analysis of source code. We also gave a graphical representation of those models that aims to be used by non-specialist in formal methods; we have shown in [3] how our graphical models can be used to specify and verify large distributed applications.

Our models enable us to have a finite representation of infinite systems. They naturally encode the semantics of languages for distributed applications. In fact, we have sketched a method for constructing parameterized models for distributed applications built with the ProActive library. This method has been described in terms of algorithms, and use an extension of method call graphs obtained by flow analysis. Our methodology was illustrated guided by a Producer-Consumer system.

We have developed a tool that makes automatic instantiations of our parameterized models, we have developed a prototype of a graphical editor to design parameterized systems and we will integrate, in a short-term, these parameterized systems to on-the-fly model checking tools.

Having a specification and the models generated from the source code, we want to check the correctness of the implementation. This check will need a refinement pre-order, which allows the implementation to make some choices amongst the possibilities left by the specification, and it should be compatible with the composition by synchronisation networks.

We shall also extend the approach to take into account other features of the middleware, and in particular the primitives for group communication, and for specifying distributed security policies. Last but not least, ProActive active objects are also viewed as *distributed components* in a component framework. In the next version, it will be possible to assemble distributed objects to form more complex components. This will increase the impact of the compositionality of our model, and the importance of being able to prove that a component agrees with its (behavioural) specification.

## References

[1] Garavel, H., Lang, F.: NTIF: A general symbolic model for communicating sequential processes with data. In: Proceedings of FORTE'02 (Houston), LNCS 2529 (2002) 43

[2] Arnold, A.: Finite transition systems. Semantics of communicating sytems. Prentice-Hall (1994) 43, 44, 45

[3] Barros, T., Madelaine, E.: Formalisation and proofs of the chilean electronic invoices system. Technical Report RR-5217, INRIA (2004) 44, 48, 58

[4] Boulifa, R., Madelaine, E.: Model generation for distributed Java programs. In N. Guelfi, E. A., Reggio, G., eds.: Workshop on scientiFic engIneering of Distributed Java applIcations, Luxembourg, Springer-Verlag, LNCS 2952 (2003) 44

[5] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Laubach, S., Zheng, H.: Bandera: Extracting finite-state models from java source code. Int. Conference on Software Engineering (ICSE) (2000) 44, 49

[6] Cleaveland, R., Riely, J.: Testing-based abstractions for value-passing systems. In: International Conference on Concurrency Theory. (1994) 417–432 44, 46, 48

[7] Milner, R.: Communication and Concurrency. Prentice Hall (1989) ISBN 0-13-114984-9. 45, 47

[8] Lakas, A.: Les Transformations Lotomaton: une contribution à la pré-implémentation des systémes Lotos. PhD thesis (1996) 45

[9] Lin, H.: Symbolic transition graph with assignment. In Montanari, U., Sassone, V., eds.: CONCUR '96, Pisa, Italy, LNCS 1119 (1996) 46

[10] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation **100** (1992) 47

[11] Bouali, A., Ressouche, A., Roy, V., de Simone, R.: The fc2tools set. In Dill, D., ed.: Computer Aided Verification (CAV'94), Standford, Springer-Verlag, LNCS (1994) 47

[12] Caromel, D., Klauser, W., Vayssière, J.: Towards seamless computing and metacomputing in Java. Concurrency Practice and Experience **10** (1998) 1043–1061 49