Model Generation for Distributed Java Programs

Rabéa Boulifa and Eric Madelaine

INRIA Sophia-Antipolis
(rabea.boulifa, eric.madelaine)@sophia.inria.fr

Abstract

We present techniques for analyzing the source code of distributed Java applications, and building finite models of their behaviour. The models are labelled transition systems, representing the communication events between the distributed objects. When combined with techniques for abstracting the data values used by the programs, and especially values used in the creation of distributed objects, to bounded domains, our construction terminates. We provide models suitable for automatic verification, and typically for model checking. Moreover our models are structured in a compositional way, so that we can use verification techniques that scale up to applications of realistic size.

1 Introduction

The use of formal verification techniques to enhance the reliability of software systems is not yet widely accepted, though there are a number of methods and tools available. Ideally, one should be provided with a development environment including high level languages for the specification of the requirements of the system, and automatic tools for checking the implementation against the specification. At the heart of this is the ability to generate, from the code of an implementation, a model that is both precise enough to encompass the property we want to prove, but small enough to be manageable by the tools (and at least have a finite representation).

The first concern is naturally related to the work on abstract interpretation [8, 10], that can be used to abstract away from concrete data values. It is also related to *slicing* techniques [3, 11], that allows to restrict the analysis to the part of the program effectively related to the property.

A number of tool developments have been done recently, for example in the Slam project [2] for analysis and verification of C programs, or in the Bandera project [7] for sequential or multi-threaded Java programs. Typically this kind of tools use static analysis techniques, coupled with abstraction and slicing, to produce a finite model that is fed to various model-checking tools. The whole process cannot be fully automatic, because the data abstractions are provided by the user, and also because other forms of approximations are needed to deal with the control flow of the program (and potential dynamic creation of objects).

We are developing a similar framework [4], dedicated to distributed Java applications, in which communications between remote objects are done by asynchronous method calls. We are interested in proving global properties of distributed applications, namely temporal properties capturing the significant events of the lifecycle of distributed objects : sending of remote method calls, receiving results from remote computations, selecting and serving requests from the local request queue. Distributed applications fit naturally with compositional models, and we shall take advantage of this for structuring the models, thus keeping them much smaller, and also by using verification tools that make use of this structure.

We rely on a middleware called ProActive [5] is an example that provides the developer with a high-level programming API for building distributed Java applications, ranging from Grid computing to pervasive and mobile applications. ProActive has a formal semantics, that guarantees that programs behave independently of their distribution on the physical network; *active objects* are the basic units of activity, distribution, and mobility used for building these applications.

The verification tools we use to check our models are based on Process Algebra theories: models are communicating labelled transition systems [1], and their semantics is considered modulo bisimulation congruences. Their emphasis is on properties related with bisimulation semantics, including safety and liveness properties in modal branching time logics, and more generally equivalence of models of different level of refinement. They take advantage of congruence properties of the systems to avoid state explosion of the models. This approach allows us to build the models of our applications on a per-object basis, and also to specify their desired behaviour in a component-wise manner.

Our contribution in this paper is a behavioural semantics for Java/ProActive applications, given in the form of SOS rules working on the method call graph [9] of the active objects. Assuming that we have (e.g. by abstract interpretation) a finite enumeration of the active objects created during the application lifetime and a finite set of message labels, these rules give us a procedure for building a finite LTS model for each object class, and a synchronization network [1] representing the application. We guarantee that obtained network is finite and the procedure for computing the behavioural model terminates.

The approach here is to build compositional models to keep individual LTS as small as possible, and to use the compositionality features of the semantics (bisimulation congruences) and of the checking tools to master the state explosion.

2 Background

2.1 ProActive

ProActive [5] is a 100% Java library for concurrent, distributed, and mobile computing whose main features are transparent remote active objects, asynchronous two-way communications with transparent futures, and high-level synchronization mechanisms. *ProActive* is built on top of standard Java APIs and does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor, or modified virtual machine. The target environment can be either a multiprocessor workstation, a pool of machines on a local network, or even a grid.

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities (see Figure 1) called *active objects* which can informally be thought of as "active components". Each active object has one distinguished element, the *root*, which is the only entry point to the active object. All the other objects inside the active object are called *passive objects* and cannot be referenced directly from objects which are outside of the active object.

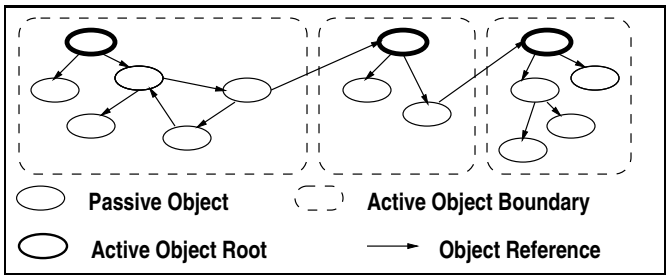

Fig. 1. A typical object graph with active objects

Each active object has its own thread of control and has the ability to decide the order in which to serve incoming method calls that have been dropped into its pending requests queue. Method calls to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism called *wait-by-necessity*. Note that while

asynchronous, there is a guaranty of delivery, and a conservation of order for remote method calls ; this is achieved thanks to a short *rendez-vous* phase at request sending.

ProActive provides primitives for dynamically creating and migrating active objects. In the context of this paper, we shall not elaborate on those dynamic features, but rather concentrate on the case where the active objects are already created and their topology is fixed.

2.2 Method Call Graphs

Our analysis focuses on applications in which the interaction between active objects is done by messages exchange: procedure or method calls (e.g. Figure 2). A model of program capturing these interactions is the method call graph. The latest abstracts away all data flow and focuses on control flow which methods are called during the execution and in what order.

Definition 1 *Starting with the main method of an application, its* **Method Call Graph** *is a tuple* $MCG \stackrel{def}{=} (id, V, \mu, \rightarrow^T, \rightarrow^C)$ *where*

1. id *is a qualified method name in the form Class.identifier or Package.identifier,*
2. V *is a set of nodes, each decorated by a node type in* $\{ent(id),\ call(id),\ rep(id),\ seq,\ ret\}$,
3. $\mu : V \rightharpoonup V$ *is a partial function mapping nodes (representing a program point where a future object is used) to the node where this future was defined,*
4. $\rightarrow^C \subseteq V \times V$ *are the call edges of MCG,*
5. $\rightarrow^T \subseteq V \times V$ *are the (intra-procedural) transfer edges of MCG.*

The node type indicates whether a node is the entry point of method (*ent(id)*), a call to another method (local or remote) (*call(id)*), a reply point to a remote method call (*rep(id)*), a sequence node (representing standard sequential instruction, including branching) (*seq*), or a node in which the execution of the method terminates, either normally or with an exception (*ret*).

The domain of a method named id is the set of its nodes, $\mathcal{D}(id) = V$.

2.3 Labelled Transition Systems

As usual in the setting of process algebras and distributed applications, we give the behavioural semantics of programs in terms of labelled transition systems. The composition of LTSs representing the behaviour of individual active objects, into the global LTS of the application is expressed by synchronization networks [1]. We give here the notations that we use for those notions:

Definition 2 Action. *The alphabet of actions is a set* $A = \{!m, m, ?m\}$ *representing method calls (local and remote), and reception of their results.*

Definition 3 LTS. *A labelled transition system is a tuple* $LTS \stackrel{def}{=} (S, s_0, L, \rightarrow)$ *where* S *is the set of states,* $s_0 \in S$ *is the initial state,* $L \subset A$ *is the set of labels,* $\rightarrow$ *is the set of transitions :* $\rightarrow \subset S \times L \times S$. *We write* $s \stackrel{\alpha}{\rightarrow} s'$ *for* $(s, \alpha, s') \in \ \rightarrow$.

Graphical Nets. The operator for composing LTSs is a synchronization network. We give its definition in a graphical form (as in [12]), before defining its semantics as a synchronization product.

```
public class A {
    ...
  void m0(){
    int i=0;
     B.m1(i);
    }
}
```

```
public class B {
    ...
    void m1(int i){
      i--;
      if (i>0)
        m1(i);
      }
}
```

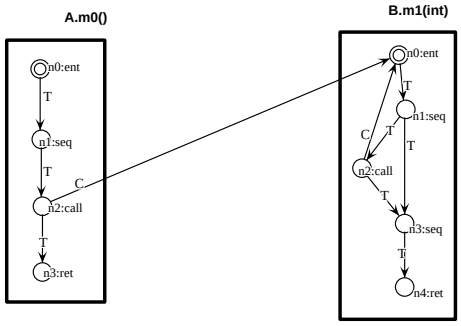

Fig. 2. Java program with the corresponding method call graph

Definition 4 Net. *A net is a pair* $Net \stackrel{def}{=} (\mathcal{B},\ \mathcal{L})$ *where*

- $\mathcal{B}$ *is a set of Boxes, each box has a name* $Name(\mathcal{B})$ *and a set of labels* $Ports(\mathcal{B})$ *labelled by actions in* A,
- $\mathcal{L}$ *is a set of links between ports, each link having the form* $B1.p1 \xrightarrow{l} B2.p2$ *in which* $p1$ *is a port of box* $B1$ *and* $p2$ *is a port of box* $B2$ *and* l *an action in* A.

The set of port labels of the box B_i *and the set of actions of links* $\mathcal{L}$ *are respectively denoted* $Actions(B_i)$ *and* $Actions(\mathcal{L})$.

Synchronization Product. Given a Net with n boxes and their corresponding LTSs, we construct a global system for the behaviour of the network, using the synchronization product of [1]. Basically, each of the links of a network is represented by a synchronization vector in Arnold's setting.

Definition 5 Synchronization Product.
Given a Net $N \triangleq (\mathcal{B}_n,\ \mathcal{L})$ *and* n *LTSs* $\{(S_i,\ s_{0i},\ L_i, \rightarrow_i)\}_{i=1..n}$, *we construct the product LTS* $(S,\ s_0,\ L,\ \rightarrow)$ *where* $S \subset \Pi_{i=1..n}(S_i)$, $s_0 = \Pi_{i=1..n}(s_{0i})$, $L \triangleq Actions(\mathcal{L})$,
$\rightarrow \triangleq \{s \xrightarrow{\alpha} s' |\ s =< s_1, \ldots, s_n >, s' =< s'_1, \ldots, s'_n > \text{ and } \exists\ l \in \mathcal{L}.\ l = B_{i_1}.p_{j_1} \xrightarrow{\alpha} B_{i_2}.p_{j_2},\ s_{i_1} \xrightarrow{B_{i_1}.p_{j_1}} s'_{i_1},\ s_{i_2} \xrightarrow{B_{i_2}.p_{j_2}} s'_{i_2}, \forall i \neq i_1, i_2.\ s'_i = s_i\}$

3 Informal semantics of ProActive distributed applications

In the current version of the ProActive middleware, an application is a (flat) collection of *active objects*, that can be dynamically created, migrated, and terminated. We shall associate a *process* to each *active object* of the application, and build a synchronization network to represent the communication between active objects.

We are not interested here in properties related to the location of objects in the physical network, so we can discard all information on location and migration. We are interested in the creation and termination of objects, but we shall limit ourselves in this work to a static

and finite topology of processes, that can be approximated by static analysis techniques [10], based on the primitives for active object creation, e.g. :

$ProActive.newActive\ (Class,\ Args,\ Node)$

Static analysis will also give us, for each process, the set of services it offers (*public methods*) and the set of services it uses (*method calls* to other active objects), allowing us to build the synchronization network between the objects.

The model of an active object is the product of two LTSs, the first encoding (a bounded approximation of) its pending request queue, the other representing its behaviour, that the user has programmed in the special *runActivity* method. The construction of this behaviour includes the unfolding of local method calls (of the active object itself, and of local passive objects) ; the termination of this procedure is obtained by detection of (mutually) recursive method calls modulo a finite abstraction of the call stack (see next section).

We have said that the events of our models are the requests and responses of remote methods. Consider an object *co* (the current active object), calling method *m* (with a non-void return type) of a remote active object *ro* :

$x = ro.m\ (arg1,\ \ldots\ ,\ argn)\ ;$

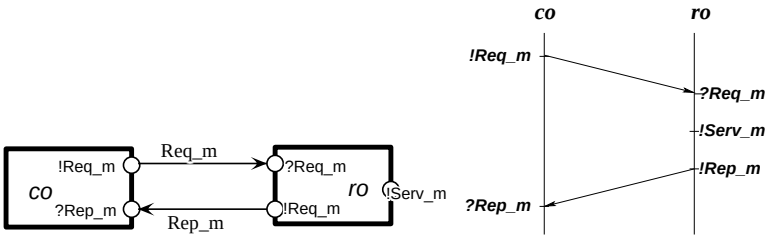

Fig. 3. Message exchanges between the processes co and ro

We model these calls by message exchanges (Figure 3) between the two processes : *co* emits a request $!Req\_m$, that is instantaneously received by *ro*'s queue as $?Req\_m$. The remote object *ro* may eventually decide to serve this request. Starting the execution of the method call within object *ro* is modelled by a message $Serv\_m$ local to *ro*. And finally, when the method returns a result, *ro* emits a reply message $!Rep\_m$, synchronously received by *co* as $?Rep\_m$.

From the *current object* point of view, the response is asynchronous : the *wait by necessity* semantics of *ProActive* says that the reply message can be received at any time between emission of the request and the first effective usage of the returned object, i.e. the access to a field or method of this object ; or an explicit awaiting of this result. We model this mechanism by interleaving the *response* message with the behaviour of *co* between these two points.

The *Request Queue* of an active object deals with arrival, service and disposal of requests. It is always ready to accept incoming messages. The synchronization with the *runActivity* LTS is done through the service primitives of ProActive, that allow the user to inspect and select specific requests in the queue.

4 Model construction

An application is a set of active objects. We model the system by LTS in a compositional way. Active objects are connected by synchronized network whose links represent the exchange of messages.

We use static analysis techniques to establish for each application :

– an enumeration of its active objects creation points and an enumeration of the links (the set of incoming and outgoing requests) between the objects,
– its method call graph,
– predicate $Active(O)$ indicating for each object O belonging to this application whether O is a ProActive or a Java (passive) object.

Besides given a property to be proved, we require the user to provide an abstract interpretation for all data types in its application. This abstraction will retain only data information (active objects and messages parameters) relevant to this property. In practice, tools like the Bandera abstract specification language can be used.
Property : Whenever all domains of this abstraction are finite, we obtain a network with finite topology, and the procedure for computing the behavioural model is guaranteed to terminate.

4.1 Network construction

We suppose that we have a function enumerating all active object creations in a ProActive application. This set in principle need not be finite, because some applications would create active objects dynamically. In practice, most interesting properties would involve only finitely many instantiations of any given active object class. So, we need to consider only a finite number of objects. Depending on the topology of the application, the enumeration may be obtained by static analysis, or directly provided by the user.

The result of the active object enumeration is expressed as :

1. $\mathcal{O} = \{O_i\}$ a finite number of active object classes.
2. A finite number of instantiations for each class, depending on the possible values of the arguments passed at their creation, denoted $Dom(O_i)$.

For each such object, we build a Box $B(O_i, k)$, named by the object class O_i and an index k in $Dom(O_i)$.

Then for each active object class, we analyze the relevant source code (starting with the *runActivity* method of the object main class), 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 $?Req\_m$ of its box.
2. We then identify in the code the variables carrying "active objects" values, using usual data flow analysis techniques, and taking into account their creation parameters. Amongst those objects, one is identified as the "current object", the others are "remote objects".
3. For each call to a method of the current active object, we add a "local" port m to the current box.
4. For each call to a method of a remote object, we add a "send request" port $!Req\_m$ to the current box, linked to the corresponding port of the remote object box, and if this method expects a result, a "response" port $!Rep\_m$.

In the previous procedures, we can choose, depending on the properties we want to prove, to tune the precision of the model construction : the parameters (other than those already used in the boxes enumeration) of local or remote method calls can be taken into account if needed : in this case the corresponding ports and links will be indexed by the possible values of those parameters. This is similar in spirit to abstraction or slicing techniques used in other frameworks.

The behaviour of an active object is programmed in its *runActivity* method ; it specifies how the requests arriving in the object *request queue* are served, and the requests that the object sends to others. The box of an active object is in turn structured in a runActivity LTS and a request queue LTS connected by a synchronization network. We build a box $B(\mathcal{A})$ for

the runActivity LTS and a box $B(\mathcal{Q})$ for the queue. The ports and links computed between these boxes are determined by the set of public methods of this active object:

For each method m, a "send" port $!Serv\_m$ and a "receive" port $?Serv\_m$ are added on $B(\mathcal{A})$ and $B(\mathcal{Q})$ respectively, and a link between them. If m expects a non-void result, a "send" port $!Rep\_m$ is added on $B(\mathcal{A})$. An example of a generated Net is displayed in Figure 5.

The request queue of an active object runs independently of its body, and is always accepting arriving requests (of correct type). It is synchronized with the object body through the $Serv\_m$ actions, and with remote objects by their Req_m actions. The model of the queue is a LTS $\mathcal{Q}$ constructed in generic way by ad graph grammar from the set of method coming in it and its bound.

4.2 Behaviour computation

In the following section we give a construction function $\Longrightarrow_A$ of active objects activity describing a traversal of the method call graph of an active object class, and the corresponding building of the LTS. This function is specified in terms of a set of transition rules in SOS style :

$$\frac{\{Premise\}*}{< v = pattern,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \quad \Longrightarrow_A \quad < v',\ n',\ \mathcal{A}',\ \mathcal{M}',\ \mathcal{S}'_n,\ \mathcal{S}'_m >}$$

where

- $v = pattern$ is the current analyzed MCG node, together with its value,
- n is the last LTS node created,
- $\mathcal{A}$ the currently constructed LTS. The empty LTS is denoted by $\emptyset$, and the LTS reduced to a single node is denoted by this node itself,
- $\mathcal{M}$ is a mapping from MCG nodes already visited to LTS nodes,
- $\mathcal{S}_n$ a stack of MCG nodes; empty stack is denoted by [ ] and push on top of the stack is denoted by $v : \mathcal{S}$,
- $\mathcal{S}_m$ a method calls stack.

We need some auxiliary constructions, to be used in the rules.

Definition 6 Sub-LTS. *A sub-LTS, of the LTS $\mathcal{A} = (S,\ s_0,\ L\ , \rightarrow)$, denoted $\mathcal{A}^m_n$, gives LTS $\mathcal{A}'$ defined by $S' \subseteq S$, $L' \subseteq L$, $s'_0 = n$, $\rightarrow' = \{s_1 \rightarrow s_2 \in \rightarrow\ | \exists x \in L^*.\ n \xrightarrow{x} s_1 \in \rightarrow$ and $\exists x' \in L^*.\ s_2 \xrightarrow{x'} m \in \rightarrow\}$*

Definition 7 Interleaving operator. *Interleave a sub-LTS $\mathcal{A}^{s_0}_{s_k}$ with an action α, denoted $\mathcal{A}^{s_0}_{s_k} \parallel \xrightarrow{\alpha}$, gives the pair a LTS and a node, $[\mathcal{A}', s']$ defined by $S' = \{s_l | s \in S\} \cup \{s_r | s \in S\}$, $L' = L \cup \{\alpha\}$, $s'_0 = s_0$, $\rightarrow' = \{s_{1l} \xrightarrow{x} s_{2l}$ and $s_{1r} \xrightarrow{x} s_{2r} | s_1 \xrightarrow{x} s_2 \in \rightarrow\} \cup \{s_{1l} \xrightarrow{\alpha} s_{1r} | s_1 \in S\}$, $s' = s_{kr}$*

Definition 8 Connection operator. *The connection of LTSs $\mathcal{A}$ and $\mathcal{A}'$ as $s'_0 \in S$, denoted $\mathcal{A} \triangleleft \mathcal{A}'$, gives $\mathcal{A}''$ defined by $S'' = S \cup S'$, $L'' = L \cup L'$, $s''_0 = s_0$, $\rightarrow'' = (\rightarrow \cup \rightarrow')$*

Definition 9 Substitution. *Operation consisting in replacing a node n by a node m, i.e replace all $n \xrightarrow{x} n'$ by $m \xrightarrow{x} n'$ and all $n' \xrightarrow{x} n$ by $n' \xrightarrow{x} m$, is denoted $\mathcal{A}[m/n]$*

We are ready now to describe the construction rules of the LTS corresponding to the active object's behaviour from the *runActivity* method MCG which encapsulates this behaviour. At the beginning we have

$$< v = ent(runActivity),\ \emptyset,\ \emptyset,\ \mathcal{M},\ [\,],\ [\,] >$$

Note that each MCG node visited is mapped to a corresponding node in the LTS and added to $\mathcal{M}$. In the rules description the unused fields by construction are grayed.

$$\frac{v_1 \rightarrow^T v_2 \quad fresh(init(n))}{< v_1 = ent(runActivity),\ \emptyset,\ \emptyset,\ \mathcal{M},\ [\,],\ [\,] > \Longrightarrow_A < v_2,\ n,\ n,\ \mathcal{M} \cup \{v_1 \mapsto n\},\ [\,],\ runActivity : [\,] >} \text{(Ent\_Run)}$$

$$\frac{v_1 \rightarrow^C v_2 \quad v_1 \rightarrow^T v_2' \quad Passive(O) \quad fresh(n')}{< v_1 = call(O.m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ v_2' : \mathcal{S}_n,\ \mathcal{S}_m >} \text{(L\_Call)}$$

The first method analyzed is the method named *runActivity*; at the entry point the initial LTS node is then created (Ent_Run rule). For a call to a local method we construct a transition labelled with the name of this called method (L_Call rule) ; afterwards this method is inlined (its MCG is analyzed). The continuation node v_2' is save on the node stack for later analysis.

$$\frac{m \neq runActivity \quad Rec(v_1)}{< v_1 = ent(m),\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v,\ n',\ \mathcal{A}[n'/n],\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{(Ent1)}$$

$$\frac{\neg Rec(v_1) \quad v_1 \rightarrow^T v_2}{< v_1 = ent(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v_2,\ n,\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_n,\ m : \mathcal{S}_m >} \text{(Ent2)}$$

where $Rec(v_1)$ is the predicate: $\mathcal{M}(v_1) = n' \wedge \exists name \in \mathcal{S}_m.\ v_1 \in \mathcal{D}(name)$

The next set of rules deals with the structure of local method (method of passive objects of the current active object) invocation. Rule L_Call has created a visible event corresponding to a method call, but we must now recognize recursive calls, using the mapping $\mathcal{M}$, and a finite abstraction of the method call stack (a method name appears only once in the stack). This is the role of the $Rec(v)$ predicate : if an entry node v has already been visited the $\mathcal{M}(v)$ is defined. And if its method name is present in the call stack, then we have detected a recursive call, and we build the corresponding cycle in the generated LTS. Otherwise the name of the method is pushed on the call stack, and we continue analyzing this method.

$$\frac{v_1 \notin \mathcal{M} \quad v_1 \rightarrow^T v_2 \quad v_1 \rightarrow^T v_3 \quad \ldots \quad v_1 \rightarrow^T v_n}{< v_1 = seq,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v_2,\ n,\ \mathcal{A},\ \mathcal{M} \cup_i \{v_i \mapsto n\},\ v_3 : \ldots : v_n : \mathcal{S}_n,\ \mathcal{S}_m >} \text{(Seq)}$$

$$\frac{\mathcal{M}(v_1) = n'}{< v_1,\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v,\ n',\ \mathcal{A}[n'/n],\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{(Loop-Join)}$$

Sequence nodes represent sequential instructions and branching. Thus no transition is constructed. In the case of branching, the alternative nodes are pushed onto the stack nodes for later visit (Seq rule).

The (LOOP-JOIN rule) applies to all type of nodes that already have been visited. Then corresponding LTS node is substituted to the current LTS node, eventually creating loops or joins in the LTS.

$$\frac{\mathcal{M}(v) = n'}{< v_1 = ret,\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ m : \mathcal{S}_m > \Longrightarrow_A < v,\ n',\ \mathcal{A},\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_n,\ m : \mathcal{S}_m >} \text{ (RET1)}$$

$$\frac{v \notin \mathcal{D}(m)}{< v_1 = ret,\ n,\ \mathcal{A},\ \mathcal{M},\ v : \mathcal{S}_n,\ m : \mathcal{S}_m > \Longrightarrow_A < v,\ n,\ \mathcal{A},\ \mathcal{M} \backslash \mathcal{D}(m),\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{ (RET2)}$$

At a return node, we check whether the nodes waiting on the node stack. If they belong to the same method we go and process them (RET1 rule). If not, we pop this method from the call stack and we remove its nodes from the mapping before continuing the analysis (RET2 rule).

Note that those rules will not apply if the node stack is empty ; this case will terminate the LTS construction procedure.

$$\frac{v_1 \rightarrow^T v_2 \quad v_2 = call(m) \quad fresh(n')}{\begin{array}{l} < v_1 = call(serve),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \\ \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Serv\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_n,\ \mathcal{S}_m > \end{array}} \text{ (SERV)}$$

This is a call to one of the *service* primitives of the library, e.g. *service.blockingServeOldest(m)*. The service primitives allow to select a request in the queue and run it. The call graph constructed in this case has a first *call(serve)* node without a call edge (we do not represent the serve primitive itself), immediately followed by a *call(m)* node. So we simply construct a transition labelled $!Serv\_m$, and go to the next node.

$$\frac{v_1 \rightarrow^T v_2 \quad fresh(n')}{\begin{array}{l} < v_1 = rep(m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \\ \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \triangleleft (n \xrightarrow{!Rep\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n\},\ \mathcal{S}_n,\ \mathcal{S}_m > \end{array}} \text{ (REP)}$$

When a request returns a non-void result, the control graph will have a special *rep* node following the *call(serve); call(m)* sequence. At this point we construct a transition labelled $!Rep\_m$, the emission of response to the remote caller (REP rule).

$$\frac{v_1 \rightarrow^T v_2 \quad Active(O) \quad fresh(n')}{< v_1 = call(O.m),\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v_2,\ n',\ \mathcal{A} \lhd (n \xrightarrow{!Req\_m} n'),\ \mathcal{M} \cup \{v_1 \mapsto n'\},\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{ (R\_CALL)}$$

In the case of the call to a remote method, the call graph has no call edge (because it was built separately for each active object). We construct a transition holding the label $!Req\_m$, emission of the request named by the method name towards this remote object (R_CALL rule).

$$\frac{\mu(v_1) = v_2 \quad n' = \mathcal{M}(v_2) \quad (n' \xrightarrow{!Req\_m} n'') \in \rightarrow \quad [\mathcal{A}', n'''] = (\mathcal{A}^n_{n''} \parallel \xrightarrow{?Rep\_m}) \quad v_1 \rightarrow^T v_2'}{< v_1,\ n,\ \mathcal{A},\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m > \Longrightarrow_A < v_2',\ n''',\ \mathcal{A} \lhd \mathcal{A}',\ \mathcal{M},\ \mathcal{S}_n,\ \mathcal{S}_m >} \text{ (FUT)}$$

One interesting feature of *ProActive* is the notion of the wait-by-necessity : it is only when the result of a method call (future value) is used that this result is awaited; synchronization by *rendez-vous*. In our model we preserve this feature. The MCG is marked by the μ mapping, that identifies the point of creation of a given future, for each of its first points of utilisation. The effective return of the future value (the transition labelled $?Rep\_m$), is interleaved with the part of the LTS comprised between its definition point and its first utilisation points (FUT rule).

5 Example

We illustrate our work on an example borrowed from work made with the Chilean administration to provide electronic procedures for their sales and taxes system. It involves software pieces that will run at different locations (government and companies) and communicate through the Internet. The specification involves both safety (protocols) and security (authentification, secrecy, ...) aspects.

The whole system is huge in size, and we shall only describe here a small part of the *Vendor* processes, in charge of managing the invoice's *Stamps* delivered by the administration (SII) to a given Vendor. This Vendor subsystem is composed of 4 components; we concentrate here on the dialog between the "StampStock" component and the corresponding SII component : the vendor requires a number k of "stamps", and the SII will eventually answer by sending an authorization code for a number x of stamps. This specification is depicted in a parametrised manner in Figure 4.

We have proved a number of temporal properties on this system, for example that 'it's not possible to cancel an invoice which has been never emitted" or " Every buyer which makes a purchase will eventually receive an invoice".

Now we give the code of a ProActive implementation of a StampStock object. It receives external *Stamp* requests. It is more specific than the specification : it asks for more stamps immediately after having received the previous stamps. The response from SII can arrive at any time, but will only be used when the stock is down to 0 (last `else` branch).

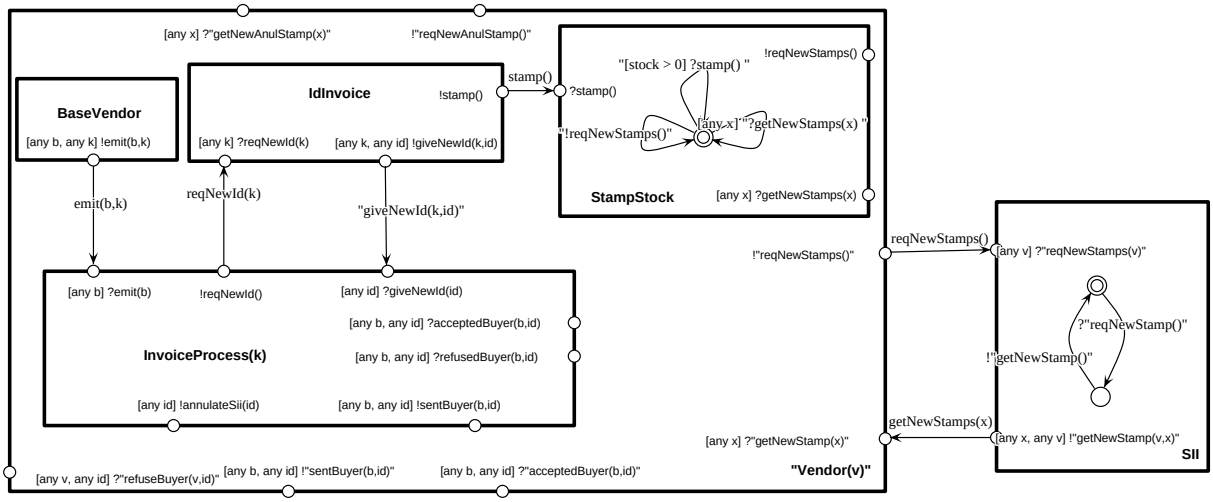

Fig. 4. Specification of a Vendor (partial)

```
public class StampStock implements RunActive {
 protected SII sii; ...
 public Boolean stamp() {
                stock--;
                return (new Boolean(true)); }
 public void runActivity(Body body) {
                Service service = new Service(body);
                boolean stampsRequested = false;
                Integer newStamps = new Integer(0);
                while (body.isActive()) {
                      if (!stampsRequested) {
                                  newStamps = sii.NewStamps();
                                  stampsRequested = true; }
                      else if (stock > 0)
                                  service.blockingServeOldest();
                      else {int ns = newStamps.intValue();
                            stock = stock + ns;
                            stampsRequested = false; }}}}
```

The model generated for the StampStock and SII classes, for a small instantiation of the object parameters (stock <= 2), is shown in Figure 5. The queue of StampStock has been arbitrarily bounded to 2, while the queue of SII can be provably bounded to 1. It can be shown that this implementation is correct versus its specification.

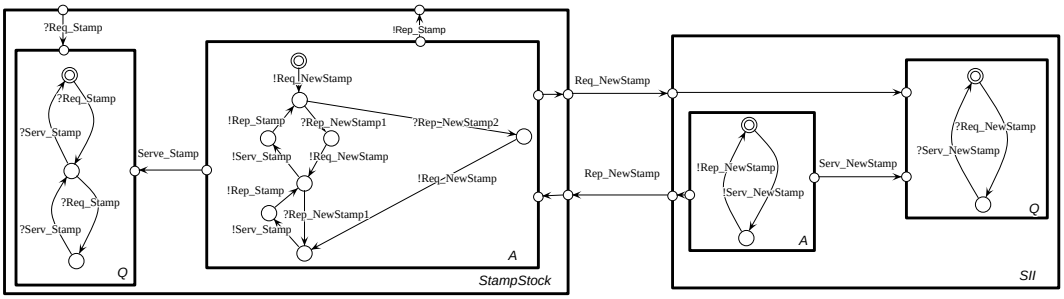

Fig. 5. Constructed models for StampStock and SII

6 Conclusion

We have sketched a method for constructing finite asynchronous models for distributed Java applications built with the *ProActive* library. The models are hierarchical labelled transition systems, suitable for analysis with verification tools based on bisimulation semantics. The behaviour of distributed objects is obtained by SOS-style rules, from the method call graph of the application.

We have illustrated this approach on a distributed implementation of the Chilean invoice and tax declaration system and we have checked some properties. We are working on relating our behavioural semantics with the operational model of ProActive [6], to get correctness results.

One of the crucial hypotheses used in this work is the ability to enumerate by static analysis a finite number of distributed objects. This allows us in practice to prove many useful properties. But there are now a number of verification tools, either automatic or partially automatic (e.g. Moped, Trex, or Harvey), that can deal with some kind of finite representations of infinite systems, or with parametrised systems. We are working on an extension of our intermediate model (LTSs and Nets) that supports parametrised systems; the models will be more compact than in the fully instantiated version, and may be directly user with this new generation of tools.

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. A. Arnold. *Finite transition systems. Semantics of communicating sytems.* Prentice-Hall, 1994.
2. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop)*, LNCS 2057. Springer-Verlag, 2001.
3. D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43:1-50, 1996.
4. R. Boulifa and E. Madelaine. Finite model generation for distributed Java programs. In *Workshop on Model-Checking for Dependable Software-Intensive Systems*, San-Francisco, June 2003. North-Holland.
5. D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.
6. Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*. ACM Press, 2004. To appear.
7. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. *Int. Conference on Software Engineering (ICSE)*, 2000.
8. P. Cousot and R. Cousot. Software analysis and model checking. In E. Brinksma and K.G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002*, Copenhagen, Denmark, LNCS 2404, pages 37–56. Springer-Verlag Berlin Heidelberg, 27–31 July 2002.
9. J. Dean D. Grove, G. DeFouw and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented*, pages 108–124, 1997.
10. J. Feret. Dependency analysis of mobile systems. In *European Symposium on Programming (ESOP'02)*, number 2305 in LNCS. Springer-Verlag, 2002. © Springer-Verlag.
11. J. Hatcliff M. Dwyer and H. Zheng. Slicing software for model construction. *Journal of High-order and Symbolic Computation*, 2000.
12. V. Roy and R. de Simone. Auto and autograph. In *Workshop on Computer Aided Verification, New-Brunswick*, pages 65–75. LNCS 531, Spring-Verlag, June 1990. also available as RR-4460.