A STEP TOWARD AUTOMATIC DISTRIBUTION OF JAVA PROGRAMS

Isabelle Attali
Denis Caromel
Romain Guider
INRIA - CNRS - I3S - UNSA
BP 93, F-06902 Sophia Antipolis
France
First.Last@inria.fr

Abstract This article investigates the automatic distribution and parallelization of object-oriented programs. We formally define a set of properties that allow to turn standard objects into active objects to be distributed on a network (local or wide area).

We first explain a principle of seamless sequential, multi-threaded and distributed programming using Java, that enhances code reuse and code distribution. We exhibit conditions on the graph of objects to detect activable objects and transform a sequential program into a distributed or parallel program using active objects. Finally, we explain how these properties were implemented in order to check them statically on a given Java program.

Keywords: model for object distribution, formal techniques, program analysis

1. INTRODUCTION

One major issue of object-oriented programming is reusability through inheritance, polymorphism, and dynamic binding. This feature has also been studied and enhanced in the context of parallelization and distribution. Several concurrent object-oriented programming languages have been designed, see for instance Hybrid [30], Pool [3], DROL [34], and more recently Java [23]. Also, many object-oriented languages have been extended to address concurrency, parallelism, and distribution issues, see for instance ConcurrentSmalltalk [35], Distributed Smalltalk [9], Eiffel// [11], ProActive [14].

Most of these works offer a setting for easy parallel, distributed, and concurrent programming, starting from a sequential application, and making it run on a parallel or distributed architecture. Some languages provide a unified syntax (no syntactic extension) for both sequential and parallel versions. This feature is critical for reuse.

The concurrent object-oriented language Eiffel// comes together with a design method for concurrent applications [12]. This method promotes a first phase of sequential design and programming before a second phase that must start with the identification of concurrent activities, and then follows with the creation of active objects (actor-like entities [1]). This method can be applied to Java as well, using the ProActive library [14], which makes it possible to reuse sequential components in a parallel and distributed setting with very limited modifications.

Another problem indeed is to be able to guarantee that the semantics of the original sequential version is preserved in a parallel and distributed setting. This problem has been tackled in various works, using different modelizations: π-calculus [29], operational semantics [27], natural semantics [5], Temporal Logic of Actions (TLA) [7]. These work are to be generalised, and extended to the point where the detection of parallelism is no longer manually achieved, but automatic.

Toward that goal, we present in this article a parallelization and distribution criterion based on properties of the graph of objects. The criterion formally expresses conditions under which objects are *activable*, and so can be transformed into *active* objects of the ProActive library. The library itself takes care of the distribution and parallel executions:

- the active object can be created either within the current VM, or on a remote host;
- an active object has its own thread that executes methods invoked on the object in a default FIFO order;
- the semantics of calls to an active object are transparently asynchronous, with no code modification being required on the caller's side (futures with wait-by-necessity [11]).

We do not address the problem of compiling legacy code for high performance architectures; instead, we want to provide developers of distributed or concurrent applications with a tool that helps them in the process of identification and organization of parallel activities.

The next section of this paper presents the underlying model of distribution used in the ProActive library, and details the transformations between sequential and parallel versions of a program. In Section 3, we

briefly describe semantic domains for an operational semantics of the Java language. Based on these domains, we formally express conditions for detecting activable objects on the graph of objects, and prove that these conditions enforce a tree-like topology of processes. We explain in Section 4 how we can derive an algorithm for checking our criterion, founded on a static analysis of Java programs, and compare this work with existing static analyses. Then we discuss related work on parallelization in Section 5. Finally, we conclude and outline future work.

2. DISTRIBUTION OF OBJECT-ORIENTED PROGRAMS

In this section we give an overview of the ProActive model and library, and explain how one can use it as a target system for distributing applications. We pinpoint some of the crucial features that make it possible, with very limited changes, to turn standard objects into active ones to be distributed. A typical example illustrates that technique (a binary search tree).

The target model of distribution that we use has been studied and improved along several experiments, both practical and more formal [12, 13, 5, 4]. The current experimentation and implementation are done within Java with a library named ProActive [14]. Its main goal is to improve simplicity and reuse in the programming of distributed object systems.

The ProActive model uses by default the following principles and features:

- heterogeneous model with both passive and active objects;
- sequential processes;
- unified syntax between message passing and inter-process communication;
- systematic asynchronous communications towards active objects;
- wait-by-necessity (automatic and transparent futures);
- automatic continuations (a transparent delegation mechanism);
- no shared passive objects (call-by-value between processes);
- centralized and explicit control by default;
- polymorphism between standard objects and remote or thread objects.

A *passive object* is simply a standard Java object that does not have a thread on its own; it is passive in the sense that only calls coming from outside will carry and trigger their execution. An *active object* is, in an actors sense, an object, plus a thread, plus a request line of pending calls. Based on such an heterogeneous model (featuring both active

and passive objects), and thanks to the absence of sharing, a system is always structured as several *subsystems.* Each subsystem is defined as an active root object, and all the standard objects (not active) that it can reach. An *automatic continuation* occurs when the result of an asynchronous call has to be returned to another subsystem: it allows to automatically avoid the blocking of the current subsystem; the binary search tree example below (routine `search`) features such a case.

Given a sequential Java program, it takes only minor modifications from the programmer in order to turn it into a multi-threaded, parallel, or distributed one. ProActive actually only requires instantiation code to be modified in order to transform a standard object into an active one. Here is a sample of code with several techniques for turning a passive instance of class A into an active, possibly remote, one.
A creation of a standard object through a statement:

```
A a = new A ("foo", 7) ;
```

can become either:

- instantiation-based:
  ```
  Object[] params={"foo", 7};
  A a =(A) ProActive.newActive ("A",params, Node);
  ```
- class-based:
  ```
  class pA extends A implements Active {}
  Object[] params={"foo", 7};
  A a =(A) ProActive.newActive ("pA", params, Node);
  ```
- object-based:
  ```
  A a = new A ("foo", 7) ; // No change
  a = (A) ProActive.turnActive (a, Node);
  ```

All these techniques create an active object, an instance of class `A` or `pA` on a given node. The node is actually a Java virtual machine that can be running on a remote host; the library transparently takes care of the distribution. The active object just created has its own thread that executes methods invoked on the object in a default FIFO order. The semantics of calls to such an object are transparently asynchronous, with no code modification being required on the caller's side.

Instantiation-based creation is much of a convenience technique. It allows the programmer to create an active instance of `A` with a FIFO behavior without defining any new class. In the class-based creation, given a class `A`, the programmer writes a subclass `pA` that inherits directly from `A` and implements the specific marker interface `Active`. He or she may also provide a live method in class `pA` for giving a specific activity or managing synchronization. The object-based technique enables a programmer to attach an active behavior to an existing object at any

```
package fr.inria.proactive.examples.binarytree;
public class BTree extends Object
{
    protected int key;      // Key for accessing the value contained in this node
    protected Object value; // Actual value contained in this node
    protected BTree leftTree; // The two subtrees
    protected BTree rightTree;

    public BTree () {
        this.value = null;     // On creation, the node does not contain a value
        this.leftTree = null;  // Nor does it have any child
        this.rightTree = null;
    }
    //   Inserts a (key, value) pair in the subtree that has this node as its root
    public void put (int key, Object value) {
        if ((this.leftTree==null) && (this.rightTree==null))) { // Is leaf
            this.key = key; this.value = value;
            this.createChildren ();
            return;
        } else if (key==this.key) {
            this.value = value;
        } else if (key<this.key) {
            this.leftTree.put (key, value);
        } else {
            this.rightTree.put (key, value);
        }
        return;
    }
    //  Retrieve a value from a key in the subtree that has this node as its root
    public Object get (int key) {
        if ((this.leftTree==null) && (this.rightTree==null))) { // Is leaf
            return null;
        } else if (key==this.key) {
            return this.value;
        } else if (key<this.key) {
            return this.leftTree.get (key);
        } else {
            return this.rightTree.get (key);
        }
    }
    // Creates two empty leaves as children
    protected void createChildren () {
        this.leftTree = new BTree ();
        this.rightTree = new BTree ();
        return;
    }
}
```

Figure 1 Sequential Recursive Binary Tree

time after its creation. This is especially useful when one does not have access to the code that creates the standard object to be made active; however, this technique is not used for the distribution transformations described in the current paper.

Once the active object is created, it automatically features the principles previously described. Among them, a few are critical for the goal of this paper: polymorphism between objects and active objects (allows the transformations), sequential processes without shared objects (no interleaving), asynchrony of calls and automatic continuations (avoids

```
package fr.inria.proactive.examples.binarytree;
import fr.inria.proactive.*;

public class ActiveBTree extends BTree implements Active {
   protected void createChildren (){
        this.leftTree = (BTree) ProActive.newActive ("ActiveBTree", null, null);
        this.rightTree = (BTree) ProActive.newActive ("ActiveBTree", null, null);
        return;
}}
```

Figure 2 Active Binary Tree

```
package fr.inria.proactive.examples.binarytree;
import fr.inria.proactive.*;

public class TestBT
{   public static void main (String[] args)
    {   BTree myTree;
        myTree = new BTree (); // Instantiating a standard version
              // To get an active version, just comment the line above,
              // and comment out the line below
        // myTree = (BTree) ProActive.newActive ("ActiveBTree", null, null);
        // * First parameter:  get an active instance of class ActiveBTree
        // * Second ('null'):  instantiates with empty (no-arg) constructor
        //                     'null' is a convenience for 'new Object [0]'
        // * Last ('null'):    instantiates this object on the current host,
        //                     within the current virtual machine
             // Use either standard or active versions through polymorphism
        TestBT.useBTree (myTree);
        return;
    }
// Note that this code is the same for the passive or active version of the tree
    protected static void useBTree (BTree bt)
    {   String s1; String s2;
        bt.put (1, "one");      // We insert 4 elements in the tree, non-blocking
        bt.put (2, "two");
        bt.put (3, "three");
        bt.put (4, "four");
           // Now we get all these 4 elements out of the tree
           // method get in class BTree returns a future object if
           // bt is an active object, but as System.out actually calls toString()
           // on the future, the execution of each of the following 4 calls
           // to System.out blocks until the future object is available.
        System.out.println ("Value associated to key 2 is "+bt.get (2));
        System.out.println ("Value associated to key 1 is "+bt.get (1));
        System.out.println ("Value associated to key 3 is "+bt.get (3));
        System.out.println ("Value associated to key 4 is "+bt.get (4));
           // When using variables, all the instructions are non-blocking
        bt.put (2, "twoBis");
        s2 = bt.get (2); // non-blocking
        bt.put (2, "twoTer");
        s2 = bt.get (2); // non-blocking
        s1 = bt.get (1); // non-blocking
           // Blocking operations
        System.out.println ("Value associated to key 2 is "+ s2 ); // prints "twoTer"
        System.out.println ("Value associated to key 1 is "+ s1 ); // prints "one"
        return;
}}
```

Figure 3 Binary Tree: example of main program

deadlocks and allows parallelism), wait-by-necessity (automatically respects data dependencies).

We have been studying various case studies of parallelizations, and it appears that, with these features, if the graph of objects at execution is a tree, then we can safely turn the objects of the graph into active ones. This property has been formally studied within various frameworks [4, 5, 7] and demonstrated on either examples or subsets of the model, and we are currently working on its generalization. The goal of this paper is to exploit this property in order to detect the places in a system where we can apply it for the sake of distribution or parallelization. Being a property on dynamic structures, the graph of objects and its topology at execution, it requires static analysis to uncover the places where it holds.

Figures 1, 2, 3 present an example of parallelization: a binary search tree. Applied on this example, the goal that we pursue in this paper is to help the programmer to identify the places where the `BTree` class can be transformed into an active object while the semantics remains constant. In Figure 1, the system will point out that the instantiation of `BTree`:

```
myTree = new BTree ();
```

can be replaced with an active binary tree. So the user, possibly with the help of semi-automatic tools, will define the class `ActiveBTree` and replace the above instantiation with:

```
myTree=(BTree)ProActive.newActive("ActiveBTree",null,Node);
```

The prototype we describe in this paper is able to detect that all instantiation sites of the `Btree` class in the program presented in Figure 1 can be safely turned active, including instantiation sites within the method `createChildren`.

3. A CRITERION FOR DISTRIBUTION OF OBJECT-ORIENTED PROGRAMS

In this section, we give a condition, expressed on object graphs, for the identification of active objects. We show that when all objects that satisfy this condition (activable objects) are activated, then the resulting program is correct with respect to the semantics of the sequential program and to the constraints imposed on the runtime structures within

ProActive. These constraints impose that passive objects are not directly accessible through more than one active object. To guarantee the correctness of the parallel program with respect to the semantics of the sequential one, we show that the condition implies that the topology of processes is a tree. This guarantees that there is no interference between processes.

We formulate the condition on the basis of semantic domains that describe the structure of a Java interpreter. The domains are as follows :

$$\begin{array}{lcl} Env & = & Id \rightarrow Val \\ Obj & = & Ref \times Type \times Env \\ Objs & = & \mathcal{P}(Obj) \\ Act & = & Inst \times Id \times Type \times Env \\ Stack & = & Act^{\star} \\ State & = & Stack \times Objs \end{array}$$

The syntactic domains $Type$, Id and Val are respectively the set of classes, identifiers, and values defined by the program being parallelized.

The interpreter is organized around an execution stack (in the domain $Stack$) that together with a set of object (in the domain Obj) form a state (in the domain $State$). Stack elements are method activations that correspond to the execution of a method call. Activations are built up from all the elements that are needed for a method to execute: a program counter (element of the domain $Inst$), the type of the target object (we omit static method invocation so any activation has a target object) and an environment that binds local variables to their values (local variables are both formal parameters and local variables declared inside the method body).

We currently restrict ourselves to a subset of Java where there are no threads, no class methods nor class variables, and no dynamic class loading. For more details on the description of the operational semantics, the reader should refer to [6].

We first give various definitions (Section 3.1), then a property to be satisfied by an active object (Section 3.2) and finally prove facts about the topology of processes (Section 3.3).

3.1. ACCESSIBILITY OF OBJECTS

We could directly use the domains defined for the interpreter, but we can remark that there are two ways for a method to access an object: through instance variables or through local variables. In the context of parallelization we have to consider all the objects accessed by a process, that is all the objects accessed by the executions of a method invocation on a possibly active object. In ProActive, processes are as-

sociated to objects, so we have to consider that an object o' reachable by a local variable of a method activation on an object o is in the set of objects accessible through o.

In order to reason uniformly about these two ways of accessing an object, we define a graph (the *accessibility graph* of a state) from an interpreter state. This graph represents both relations between objects by identical edges. We introduce a fictive node (r_S) that does not correspond to an object in the object graph. It is used as a root for the object graph and allows us to treat uniformly nodes pointed to by local variables from the base activation (the method `main`) as we treat nodes pointed to by other activations.

Definition 1 *Accessibility Graph*
Given a state $S = \langle s, h \rangle$, we define the accessibility graph associated with S to be $G_S = (V_S, E_S)$, such that:

- *$V_S = h \cup \{r_S\}$, where r_S is a fictive node (does not correspond to a node in h).*
- *$\langle o, o' \rangle \in E_S$ iff:*
  - *the object o points to o' (an attribute of o directly references o').*
  - *o' is the value of a local variable[1] of a method which target object is o (r_S is considered as the target object of the method* `main`*).*

The Figure 4(b) represents the accessibility graph associated to the state represented in Figure 4(a).

Our criterion is based on the inspection of the memory regions read and written by method calls. We formally define (Definition 2) these domains using accessibility graphs and we call them accessibilities of objects:

Definition 2 *Accessibility*
For a given interpreter state S, the accessibility of an object o (noted $\mathcal{A}_S(o)$) is the set of objects transitively reachable from o in the graph G_S.

[1]We consider that there is no operation of the form `t.foo(new C( ... ));` in our programs so that any object is at least pointed to by a local variable. Programs can easily be transformed to meet that requirement.

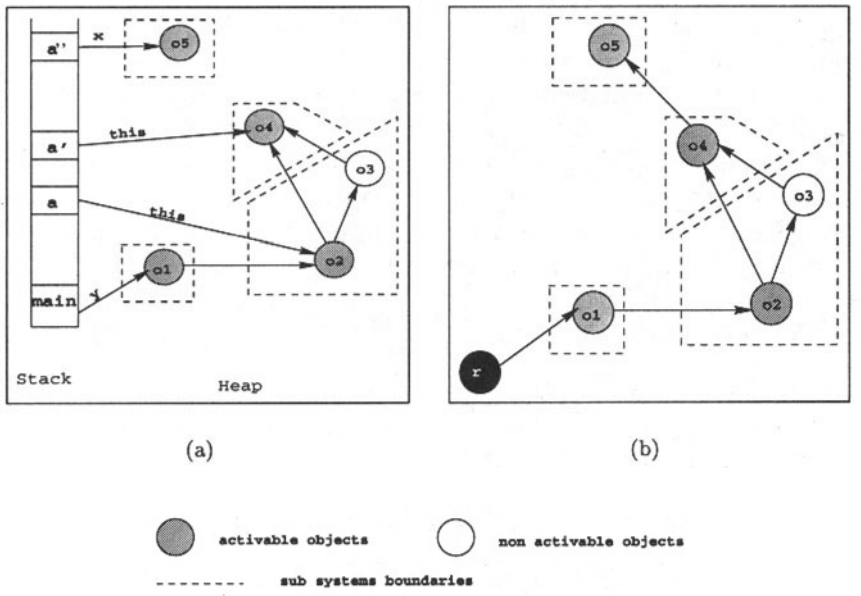

Figure 4 Activable Objects

3.2. ACTIVABLE OBJECTS

Now that we have defined accessibilities, we give a property on them which expresses that they have the same properties as subsystems. This property is stated for a given interpreter state. For an object to be activable, it actually has to verify that property in all the states during its life-time (between the time it is created and the time it is no longer referenced or the program terminates).

Property 1 *Activable*
For a program P, in a given interpreter state S, $Activable_S(o)$ iff $\mathcal{A}_S(o)$ is a disjoint part of the object graph: for all objects $o' \in \mathcal{A}_S(o)$, for all objects o'' either: $o'' \in \mathcal{A}_S(o)$ or o belongs to all the paths from o'' to o' in G_S.

The Property 1 states that an activable object has a self-contained accessibility. Any access to its accessibility must be done through it.

We show in the next section what are the properties of the graph of processes induced by that property.

3.3. TOPOLOGY OF SUBSYSTEMS

We have stated a condition on objects for detecting activable ones. We will now show that objects which satisfy this property are correct candidates for activation. This include three things:

1 We have to show that the parallel program obtained is deadlock-free so that whenever the sequential program terminates, so does the parallel program. We show this by proving that the process topology is cycle-free (Proposition 1).

2 We have to show that if such objects are activated, then the resulting system has the same property as subsystems, that is, there is no shared objects between two subsystems (see [12] for a discussion on the topic and [5] for a formalization of this). This is expressed by Propositions 2 and 3.

3 Last, we have to show that the topology of resulting processes are trees (Proposition 4). This guarantees that the semantics of the sequential program is preserved as we will explain later.

Let's start with a definition of subsystems.

Definition 3 *Subsystems*
Given a state S, and an activable object n, the subsystem associated to n is

$$\mathcal{S}_S(n) = \mathcal{A}_S(n) / \bigcup_{n' \in \mathcal{A}_S(n), n' \neq n, Activable_S(n')} \mathcal{A}_S(n')$$

This definition only formalizes subsystems as they are introduced in [12, 14] and is equivalent to that given in [5] (even though we do not formally show this fact here).

Before we state the propositions that state that activable objects can be activated, we introduce a useful lemma for proving those propositions.

Lemma 1 *Given a state S, $\forall\ n_1, n_2$ such that $Activable_S(n_1)$:*
$\mathcal{A}_S(n_1) \subseteq \mathcal{A}_S(n_2)$ *or* $\mathcal{A}_S(n_2) \subseteq \mathcal{A}_S(n_1)$ *or* $\mathcal{A}_S(n_2) \cap \mathcal{A}_S(n_1) = \emptyset$

Proof. First if $n_1 = n_2$ then the proposition is trivially true because in that case $\mathcal{A}_S(n_1) = \mathcal{A}_S(n_2)$. If $n_1 \neq n_2$ then assume $\mathcal{A}_S(n_1) \not\subseteq \mathcal{A}_S(n_2)$ and $\mathcal{A}_S(n_2) \not\subseteq \mathcal{A}_S(n_1)$ and $\mathcal{A}_S(n_1) \cap \mathcal{A}_S(n_2) = I \neq \emptyset$. We will show that this leads to a contradiction with the hypothesis. From Definition 2, we have $\forall n, n' \in V_S,\ n \in \mathcal{A}_S(n') \Rightarrow \mathcal{A}_S(n) \subseteq \mathcal{A}_S(n')$. This together with $\mathcal{A}_S(n_1) \not\subseteq \mathcal{A}_S(n_2)$ implies $n_1 \notin \mathcal{A}_S(n_2)$ (symmetrically, we prove $n_2 \notin \mathcal{A}_S(n_1)$). Let n_I be a node in I then $n_I \neq n_1$ and $n_I \neq n_2$.

Moreover, $n_I \in \mathcal{A}_S(n_2)$ implies there is a path from n_2 to n_I in G_S; since $n_2 \notin \mathcal{A}_S(n_1)$ and there is no path from n_2 to n_1, we can then conclude that $notActivables_S(n_1)$, which is in contradiction with the hypothesis. □

The next proposition states that two activable nodes can never be on the same cycle. One important corollary of this is that the graph of processes of the parallel program is acyclic. This guarantees deadlock freeness.

Proposition 1 *In an interpreter state S, if two nodes n_1 and n_2 of V_S are in a cycle in G_S, then one at most is activable.*

Proof. The node r_S is such that it is not accessible from any node in G_S; r_S is not in the accessibility of any node. Plus, in a given interpreter state, all objects that are not candidates for garbage collection are reachable through a path rooted at a local variable so that, according to the definition of G_S, any node in an accessibility graph is reachable from r_S.

Consider two nodes n_1 and n_2 in S such that there is a path from n_1 to n_2 and a path from n_2 to n_1 in G_S and a shortest path P from r_S to n_1 in G_S. Either P passes through n_2 or not. If it does, then the subpath of P that ends at n_2 does not contain n_1; otherwise P would not be a shortest path from r_S to n_1. From this, we conclude that there is a path in G_S from r_S to n_2 that does not contain n_1 or a path from r_S to n_1 that does notcontain n_2. In the first case, there is a path from $r_S \notin \mathcal{A}_S(n_1)$ to $n_2 \in \mathcal{A}_S(n_1)$ that does not pass through n_1. From this we can conclude that n_1 is not activable. In the second case, we can conduct the same reasoning replacing n_1 by n_2 and conversely conclude that n_2 is not activable. In both cases, one of the two nodes is not activable that is, one at most is activable. □

Proposition 2 *Let $n_1, n_2, n_1 \neq n_2$ be two activable objects in a state S, then $\mathcal{S}_S(n_1) \cap \mathcal{S}_S(n_2) = \emptyset$*

Proof. From Lemma 1, there are three cases to consider:

(i) $\mathcal{A}_S(n_1) \cap \mathcal{A}_S(n_2) = \emptyset$. From Definition 3 we get that for all activable object n, $\mathcal{S}_S(n) \subseteq \mathcal{A}_S(n)$ and we can conclude that $\mathcal{S}_S(n_1) \cap \mathcal{S}_S(n_2) = \emptyset$.

(ii) $\mathcal{A}_S(n_1) \subseteq \mathcal{A}_S(n_2)$ then $n_1 \in \mathcal{A}_S(n_2)$ and consequently,

$$\mathcal{A}_S(n_1) \subseteq \bigcup_{n' \in \mathcal{A}_S(n_2), Activable_S(n')} \mathcal{A}_S(n')$$

From this, we conclude that $\mathcal{A}_S(n_1) \cap \mathcal{S}_S(n_2) = \emptyset$ and finally that $\mathcal{S}_S(n_1) \cap \mathcal{S}_S(n_2) = \emptyset$ because $\mathcal{S}_S(n_1) \subseteq \mathcal{A}_S(n_1)$.

(iii) $\mathcal{A}_S(n_2) \subseteq \mathcal{A}_S(n_1)$. This is the symmetric case of the previous one. □

The next proposition states that all incoming edges of a non-activable node (a passive object) are from nodes in the same subsystem. A consequence is that a non-activable node can only be reached by nodes of its subsystem or through the active node of its own subsystem.

Proposition 3 *In a state S, given a subsystem $\mathcal{S}_S(n)$ such that n is activable, $\forall n' \in \mathcal{S}_S(n), \forall \langle n'', n' \rangle \in E_S,\ n' \neq n \Rightarrow n'' \in \mathcal{S}_S(n)$*

Proof. Assume $n \neq n'$ and $n'' \notin \mathcal{S}_S(n)$. We will show that this leads to a contradiction. There are two cases to consider:

(i) $n'' \notin \mathcal{A}_S(n)$. In that case, because $\langle n'', n' \rangle \in E_S$, there is a path from a node outside $\mathcal{A}_S(n)$ such that n does not belong to that path. This is in contradiction with $Activables_S(n)$.

(ii) $n'' \in \mathcal{A}_S(n)$. Then, because $n'' \notin \mathcal{S}_S(n)$, and using Definition 3, n' is in the subsystem of another node. So, there is a node $n''' \in V_S$ such that $n''' \in \mathcal{A}_S(n)$ and $Activables_S(n''')$ and $n'' \in \mathcal{A}_S(n''')$.

From this and $\langle n'', n' \rangle \in E_S$, we get $n' \in \mathcal{A}_S(n''')$. From that last fact and Definition 3 we get $n' \notin \mathcal{S}_S(n)$ which is in contradiction with $n' \in \mathcal{S}_S(n)$. □

Definition 4 *In a state $S = \langle s, h \rangle$, the graph of subsystems is a graph $G^\star_S = (V^\star_S, E^\star_S)$ such that:*

- $V^\star_S = \{n \in h | Activables_S(n)\}$
- $\langle n, n' \rangle \in E^\star_S \Leftrightarrow \exists o \in \mathcal{S}_S(n) \wedge \langle o, n' \rangle \in E_S$

Propositions 2 and 3 together guarantee that any object in a subsystem is only accessible from the root of the subsystem. Because active objects serve one request at a time, this guarantees that the parallel system obtained is race condition free.

Proposition 1 states that two objects that form a cycle can not be both activable. According to Definition 4, this guarantees that the process topology is a DAG, and so the resulting parallel system is deadlock-free.

All this, together with Proposition 1 guarantees that the parallel system obtained by activating activable objects is sane in some sense but it is not enough to guarantee that it is equivalent to the sequential system in term of results.

To guarantee this, we prove a last proposition that states that process graphs are trees. This ensures a stronger form of Bernstein's conditions: read/write domains of any two processes are disjoint.

Proposition 4 *The graph of subsystems is a tree.*

Proof. Formally the proposition is expressed as follows: in an interpreter state S let n_1, n_2 and n_3 by activable nodes then $\langle n_1, n_3\rangle \in E_S^\star$ and $\langle n_2, n_3\rangle \in E_S^\star \Rightarrow n_1 = n_2$. Consider that $n_1 \neq n_2$, we will show that this leads to a contradiction with $Activable_S(n_1) \wedge Activable_S(n_2) \wedge Activable_S(n_3)$. From $\langle n_1, n_3\rangle, \langle n_2, n_3\rangle \in E_S^\star$, we get that there are two nodes $o_1 \in \mathcal{S}_S(n_1)$ and $o_2 \in \mathcal{S}_S(n_2)$ and two edges $\langle o_1, n_3\rangle$ and $\langle o_2, n_3\rangle$ in E_S. From $o_1 \in \mathcal{S}_S(n_1)$ we get that there is a path from n_1 to o_1 in G_S entirely contained in $\mathcal{S}_S(n_1)$. Identically, there is a path from n_2 to o_2 entirely contained in $\mathcal{S}_S(n_2)$. From the last fact, we can conclude that $\mathcal{A}_S(n_3) \subseteq \mathcal{A}_S(n_1)$ and $\mathcal{A}_S(n_3) \subseteq \mathcal{A}_S(n_2)$. Now from Lemma 1 we get three cases to consider:

(i) $\mathcal{A}_S(n_1) \cap \mathcal{A}_S(n_2) = \emptyset$. This hypothesis is in contradiction with $\mathcal{A}_S(n_3) \subseteq \mathcal{A}_S(n_1)$ and $\mathcal{A}_S(n_3) \subseteq \mathcal{A}_S(n_2)$.

(ii) $\mathcal{A}_S(n_2) \subseteq \mathcal{A}_S(n_1)$. We have $\mathcal{A}_S(n_3) \subseteq \mathcal{A}_S(n_2)$ that is $n_3 \in \mathcal{A}_S(n_2)$. We also have a path from n_1 to o_1 entirely contained in $\mathcal{S}_S(n_1)$ and an edge from o_1 to n_3 that is a path from a node that is not in $\mathcal{A}_S(n_2)$ (because there is no cycle between two activable nodes) to a node that is in $\mathcal{A}_S(n_2)$ (this node is n_3) and that does not pass through n_2. This is in contradiction with $Activable_S(n_2)$.

(iii) $\mathcal{A}_S(n_1) \subseteq \mathcal{A}_S(n_2)$. This case is the same as the previous one, inverting n_1 and n_2. □

In the case where asynchronous calls performed by a method execution are not in a loop, because the topology of processes is a tree, the various parallel activities triggered have disjoint read/write domains. They satisfy the Bernstein's conditions [10]. Actually, they verify a stronger version of Bernstein's conditions where the union of read and write domains of each process are disjoint.

Because subsystems are disjoint (Propositions 2 and 3), a subsystem and the subsystems that it refers to have disjoint read/write domains so any operation performed by a subsystem and any operation performed by a refered subsystem satisfy Bernstein's condition.

If an asynchronous call lies in a loop, the successive execution of that call may not be independent. Because execution of subsystems is sequen-

tial and processes achieve a FIFO service of request[2], the successive calls are executed one after another, in the order they have been requested by the referencing subsystem so that the possible dependence between successive executions of a method are respected.

The activable property as expressed so far is quite restrictive. However it is possible to relax it in various ways. First, in the second part of Definition 1, we can ignore dead variables because objects they point to will not be accessed through those variables. Second, we can allow in the sequential side non-mutable objects (`Integer`, `String` in the Java core classes) to be shared by several subsystems. In the parallel version, those shared objects are copied on asynchronous calls so that they are not actually shared and because they are not mutable the subsystems still verify the Bernstein's conditions.

4. STATIC CHECKING OF THE CRITERION AND IMPLEMENTATION

A condition for parallelization and distribution expressed on states of an interpreter is useful to understand the model of distribution and to guarantee the correctness of program transformations. However it is far from being sufficient because it is not possible to directly use it to find activable objects in programs because the execution traces of programs may be infinite.

We have designed and implemented a static analysis of Java programs that computes an approximated representation of object graphs associated to program points. It is written in Java itself.

This analysis is an inter-procedural extension of the shape analysis of Sagiv et al. [33] that applies to an object-oriented language and includes abstract garbage collection capabilities (abstract garbage collection makes it possible to detect tree/list shape invariants when removing operations are performed on such structures). We have also designed a test [6] that detects, on the results of this static analysis, instantiation points where created instances fulfill the condition for the parallelization described in Section 3.

In the general case, we cannot find all of the instantiation points that could be transformed to create active instances because static analysis only provides an approximated representation of the object graphs. However, we get a conservative algorithm that computes a sufficient con-

[2]this is the default service policy in ProActive.

	PPro(199Mhz)	PII (300Mhz)	Ultra5 (300Mhz)
Btree	2,4	1,6	1,0
List	1,4	0,9	0,6
Sieve	2,1	1,5	0,9
RecursiveBtree	6,4	4,0	2,3

Table 1 CPU Time for the analysis (in seconds)

dition for the condition given in Section 3 to be satisfied. Because we have designed the analysis by abstract interpretation [16] of our operational semantics of Java programs [6], we can formally relate a test on abstract representation of states to the properties we give in Section 3 through an implication relation.

On the example from Figure 1, our prototype could detect that all instantiation sites of the `Btree` class can be safely turned active, including the sites within the method `createChildren`.

Shape analysis is a generic name for analysis that aims at statically inferring properties on dynamically allocated structures. One trend of work in that area consists in using a finite naming scheme for objects or equivalently partitioning objects into a finite number of equivalence classes. Several analysis algorithms use a finite naming scheme that induces a partition of nodes. Identifiers of *abstract nodes* can be the classes of objects. Chase, Wegman and Zadeck [15] use allocation sites to further partition the set of concrete nodes. However, properties that they can extract from programs are not powerful enough for us to check the Property 1. On the other hand, the algorithm of Sagiv et al. is powerful enough to precisely represent tree structures and cyclic lists among other things. The key features of this static analysis is to use sets of referring variables for naming abstract objects and a boolean attribute that explicitly represents sharing of objects. These features allow the algorithm to precisely represent operations that delete references into the heap (operations of the form `x.s = y`) which no other known method can do systematically.

A few experiments have been conducted on the prototype analyser we have developed. The prototype first computes a system of equations that represents the program to analyse, and then iteratively solve the system of equations. Table 1 summarises the results of a few measures we have done on various typical programs, running on a Java Virtual Machine with a JIT.

`Btree` is an iterative implementation of binary search trees (with insertion only). `List` is an iterative implementation of singly linked lists. `Sieve` is a recursive prime number generator that builds up a singly linked list of prime numbers. Finally, `RecursiveBtree` is the recursive implementation of binary search trees as given in Figure 1.

From those first figures, one can notice (not surprisingly) that recursive programs are inherently more complex to analyse.

In addition to those small programs, we conducted some experiments on a bigger one (in the range of 550 lines of Java code). The program automatically derives and simplifies arithmetic expressions using some rewriting rules. It is based on the visitor pattern [20] and is massively recursive and polymorphic. As such, it represents some sort of stress test for our analysis. We currently manage to analyse it within 10 minutes or so, and the memory consumption stays within reasonable values (20 MB).

5. RELATED WORK

The seminal paper of Bernstein [10] introduced the conditions known as *Bernstein's conditions* for (semi-)automated parallelization. These conditions state that two segments of code that have disjoint write domains and such that the read domain of one is disjoint from the write domain of the other one, can be executed concurrently. Then, a transformed program where these segments are executed concurrently produces the same result as the sequential one and terminates if and only if the sequential program terminates.

Number of works use these conditions to establish the correctness of automatic parallelization for various languages. Among others, Hendren and Nicolau [26] parallelize C programs with dynamically allocated objects (with the limitation of no cyclic structures). This work has been pursued by Hendren [25], and gave rise to multiple analyses of dynamically allocated structures and pointers [22, 17, 21] that are the basis of a checking algorithm for Bernstein's conditions. Other authors [24, 28] have also investigated the use of shape analysis for the detection of parallelism. These works differ from ours for two main reasons: the parallelization happens at the procedure level, not at the object level and distribution issues are not addressed.

We also have to cite the corpus of work devoted to automatic parallelization of Fortran programs [8, 18, 19] which, in a large part, consists in the definition of data dependence tests, with the goal of targeting distributed memory architectures. Detection of parallelism and automatic distribution are traditionally two different tasks in compil-

ers/parallelizers since data structures, usually arrays of numeric data, are not amenable for distribution. In contrast, thanks to the object data structure and specific features of ProActive, we can detect parallelism and possibility for distribution with the same algorithm.

Rinard and Diniz [31] present a method for the detection of parallelism into object-oriented programs. Beside the fact that they do not address distribution, the way they detect parallelism is radically different from other works: instead of testing Bernstein's conditions, they look for operations that commute (for instance, associative operations).

Almeida presented in [2] a type-system that relies on an abstract interpretation of object-oriented programs as a checking mechanism. This type system classifies objects in two categories, standard types and *balloon types* where objects of the second type must have a self-contained accessibility and must not be shared. With our notion of subsystems, we only impose that activable objects have a self-contained accessibility.

6. CONCLUSION

This article intends to make some contribution to the automatic distribution of object-oriented programs. We presented a criterion for the detection of distribution in Java programs: a set of properties that allow to turn standard objects into active objects to be distributed. This criterion can be used in the context of ProActive, a Java library for distributed, concurrent and parallel programming.

One contribution of this work can also be stated as it follows: given a graph of objects corresponding to a sequential program at execution, how can we partition it into a tree-shaped quotient graph that can be easily distributed ? To statically achieve such a partitioning, we have designed, and implemented in Java a prototype system that analyses sequential programs. It is an extension of the shape analysis proposed by Sagiv et al. [33].

The work presented here has to be extended in various ways. First, we want to further investigate the extensions of the criterion, in order to provide more flexibility. In particular, we want to find an extended criterion that allows the distribution of some programs for which the topology of processes forms a DAG instead of a tree. We will also investigate the possibility of designing a static analysis, derived from the shape analysis of Sagiv et al., that directly finds activable objects instead of relying on an algorithm that finds in shape graphs activable objects (this is what we currently do). The parametric view of shape analysis presented in [32] provides a good starting point for this.

References

[1] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] P. S. Almeida. Balloon types: controlling sharing of states in data types. In *Proc. ECOOP)*, LNCS 1241, pages 32–59. Springer Verlag, 1997.

[3] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proc. ECOOP '87*, LNCS 276, pages 234–242, Paris, France, June 1987.

[4] I. Attali, D. Caromel, and S. O. Ehmety. About the automatic continuations in the Eiffel// Model. In *Proc. of the 1998 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.

[5] I. Attali, D. Caromel, and S. O. Ehmety. Formal Properties of the Eiffel// Model. In *Parallel and Distributed Objects.* Hermes Science Publications, 1999.

[6] I. Attali, D. Caromel, and R. Guider. Static analysis of Java for distributed and parallel programming. rapport de recherche 3634, INRIA, March 1999.

[7] I. Attali, D. Caromel, and S. Lippi. From a specification to an equivalence proof in object-oriented parallelism. In *FMPPTA'99: Modeling and Proving*, volume 1586. Springer, LNCS, 1999.

[8] Utpal Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic Publishers, Boston, MA, USA, 1988.

[9] J. K. Bennett. The design and implementation of Distributed Smalltalk. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 318–330, December 1987.

[10] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15:757–762, October 1966.

[11] D. Caromel. Concurrency and reusability: From sequential to parallel. *Journal of Object-Oriented Programming, 3(3)*, 1990.

[12] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[13] D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996.

[14] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[15] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointer and structures. In *Proc. PLDI90*, volume 25(6), pages 296–310. ACM, june 1990.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. POPL77*, pages 238–252. ACM press, 1977.

[17] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. PLDI94*, pages 242–256, 1994.

[18] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[19] P. Feautrier. Toward automatic partitioning of arrays for distributed memory computers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[21] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *Lecture Notes in Computer Science*, 1033, 1996.

[22] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proc. POPL'96*, pages 1–15, jan. 1996.

[23] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[24] V. A. Guarna. A technique for analysing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 212–220, 1988.

[25] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proc. of the 5th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 757, 1993.

[26] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[27] S. Hodges and C. B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In *Object Orientation with Parallelism and Persistance.* Kluwer Academic Publishers, 1996.

[28] J.R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. PLDI88*, pages 21–34, june 1988.

[29] X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proc. TAPSOFT95 6th International Joint Conference CAAP/FASE*, 1995.

[30] O. Nierstrasz. Active objects in hybrid. In *Proc. OOPSLA '87, ACM SIGPLAN Notices 22 (12)*, pages 243–253, 1987.

[31] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *TOPLAS*, 19(6):942–991, 1997.

[32] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. POPL99*, pages pp. 105–118, San Antonio, TX, Jan. 1999.

[33] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, January 1998.

[34] M. Tokoro and K. Takashio. Toward languages and formal systems for distributed computing. In *Proc. of the ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 93–110, 1994.

[35] Y. Yokote and M. Tokoro. The design and implementation of Concurrent Smalltalk. In *Proc. OOPSLA '86, ACM SIGPLAN Notices, 21 (11)*, pages 331–340, November 1986.