Adding Support for Parameterized Virtual Machines to the JR Concurrent Programming Language

Hiu Ning Chan

University of California, Davis

Department of Computer Science

Honors Thesis

March 2004

Contents

List of Figures

Chapter 1

Introduction

"The JR model of computation allows a program to be split into one or more address spaces called virtual machines. Each virtual machine defines an address space on one physical machine. (A JR virtual machine includes a Java virtual machine and an additional layer that supports the JR concurrency extensions.) Virtual machines are created dynamically, in a way similar to the way objects are created. When a virtual machine is created, it can be placed on a specific physical machine." [3]

Distributed programming in JR often involves one or more coordinator processes placing worker processes on multiple virtual machines to divide and speed up the computation. If coordinator processes need to have a node identifier to store information such as machine ID and name for each virtual machine, each process itself either declares variables in the remote object for storage or uses some message passing technique for passing data around. Figure 1.1 illustrates the technique. In that example, n in class Foo is used to store the machine ID for each of its instance created on each virtual machine.

With the extension of parameterized virtual machines in JR, machine specific information can be stored on the virtual machine itself and all remote objects on a single virtual machine can share the information by calling appropriate functions, which saves programmer effort and physical memory. Figure 1.2 is the same as Figure 1.1 except now each machine ID is saved on the parameterized virtual machines object Myvm.

```
public class Main    {
        public static void main(String [] args)      {
                int size = 5;
                vm [] vmref = new vm[size];
                remote Foo [] foo_array = new remote Foo[size];
                op void done();

                for (int i = 0; i < size; i++)   {
                        vmref[i] = new vm();
                        foo_array[i] = new remote Foo(i, done)
                                        on new vmref[i];
                }
                for (int i = 0; i < size * Foo.totProc; i++)   {
                        receive done();
                }
                for (int i = 0; i < size; i++)   {
                        foo_array[i].writex();
                }
        }
}

public class Foo      {
        int n;          // for storing virtual machine ID
        cap void() c;
        public static final int totProc = 4;            // total number of processes

        public Foo(int n, cap void() c)       {
                this.n = n;
                this.c = c;
        }
        private process p((int i = 0; i < totProc; i++))      {
                        send c();
        }
        public op void writex()       {
                // prints out virtual machine ID
                System.out.println(n);
        }
}
```

Figure 1.1: Example Code with Regular Virtual Machines Support

```
public class Main     {
        public static void main(String [] args)      {
                int size = 5;
                vm [] vmref = new vm[size];
                remote Foo [] foo_array = new remote Foo[size];
                op void done();

                for (int i = 0; i < size; i++)    {
                        vmref[i] = new Myvm(i);
                        foo_array[i] = new remote Foo(done)
                                        on new vmref[i];
                }
                for (int i = 0; i < size * Foo.totProc; i++)   {
                        receive done();
                }
                for (int i = 0; i < size; i++)    {
                        foo_array[i].writex();
                }
        }
}

public class Foo      {
        cap void() c;
        public static int totProc = 4;          // total number of processes
        public Foo(cap void() c)     {
                this.c = c;
        }
        private process p((int i = 0; i < totProc; i++))      {
                        send c();
        }
        public op void writex()       {
                System.out.println(((Myvm)(vm.thisvm)).GetID());
        }
}

public class Myvm extends vm     {
        private int id = 0;
        public Myvm(int id)  {
                this.id = id;
        }
        public int GetID()     {
                return id;
        }
}
```

Figure 1.2: Example Code with Parameterized Virtual Machines Support

In this example, Myvm is the extended class of vm that stores the machine ID and provides a GetID() function for remote objects to retrieve the ID. Remote object Foo retrieves the machine ID by calling the GetID() function instead of declaring a variable to store the machine ID. Also if multiple instances of Foo are created on a single virtual machine, all those instances can retrieve the machine ID by calling the GetID() function instead of having separate copies of the same ID in each instance of Foo (Note that operation done can be put in Myvm instead; see similar example in Chapter 3.)

The subsequent chapters in this thesis discuss the syntactic changes; modifications in the implementation; new and related problems; and future work on parameterized virtual machines. To distinguish plain text explanations with keyword from source codes, I typeset example programs, JR implementation code fragments and specific keywords in the Arial typeface. In addition, the term VM will be used as an abbreviation for virtual machine.

Chapter 2

Extension for Parameterized Virtual Machines

2.1 Declaration

The syntax for parameterized VMs builds from the syntax for non-parameterized VMs.

The syntax for declaring a non-parameterized VM stays the same as before:

```
vm vm_name = new vm() [on physical_machine];
```

where vm is a keyword.

The syntax for declaring a parameterized VM is:

```
vm vm_name = new pvm({expression}) [on physical_machine];
```

where pvm is the name of a user defined class extended from the jrvm class and the {expression} denotes the actual arguments passed to the subclass constructor.

2.2 Self-Referencing the Current VM

Self-Referencing a current parameterized VM is the same as self-referencing a current non-parameterized VM, which uses the reserved expression:

```
vm.thisvm
```

2.3 Accessing Extended Class Member Fields

Only remote objects can access public member fields and methods of a parameterized VM. The syntax is as follows:

```
((pvm)(vm.thisvm)).member_field;
((pvm)(vm.thisvm)).member_method({expression});
```

where pvm is the name of the jrvm subclass that corresponds to the type of the current JR virtual machine.

2.4 Subclass Definition

Extending jrvm is the same as extending any other classes. Here, the clause "extends vm" must follow the name of the class. For example,

```
public class pvm extends vm       {
        // class body
}
```

There is no further requirement or restriction on the contents of the class body.

2.5 Objects Used inside Parameterized VMs

Objects used as types for member fields in a parameterized VM must be serializable. Here is an example:

```
public class pvm extends vm       {
        Obj myObj = new Obj();
        // rest of class body
}

public class Obj implements java.io.Serializable     {
        public Obj()   {}
        // rest of class body
}
```

Chapter 3

Example Program

The following four figures represent a comprehensive example of using parameterized virtual machines in JR. It demonstrates the correct syntax for defining and accessing fields and methods in an extended vm class. Figure 3.1 shows Myvm, which is an extended class from vm. It stores the machine ID and name for each VM. Figure 3.2 contains MyObj, which is a user-defined class that serves as a member field in Myvm. It shows an example for how user-defined objects are used with parameterized VMs. Figure 3.3 shows the definition of remote object Foo, which demonstrates the retrieval of VM information stored in Myvm. Finally, Figure 3.4 shows the Main class where all virtual machines and remote objects are created.

```
public class Myvm extends vm    {
        private int vm_id;      // test primitive type
        private String vm_name; // test string
        private MyObj vm_obj;   // test arbitrary object
        private cap void (int) vm_cap;  // test capability and operation

        public Myvm(int vm_id, String vm_name, MyObj vm_obj,
                cap void (int) vm_cap)  {
                this.vm_id = vm_id;
                this.vm_name = vm_name;
                this.vm_obj = vm_obj;
                this.vm_cap = vm_cap;
        }

        public int GetID()      {return vm_id;}

        public String GetName() {return vm_name;}

        public int GetObj()     {return vm_obj.GetObjID();}

        public cap void (int) GetCap()  {return vm_cap;}

}
```

Figure 3.1: Myvm.jr

```
public class MyObj implements java.io.Serializable     {
        private int MyObj_id = 10;

        public MyObj(int id)    {MyObj_id = id;}

        public int GetObjID()   {return MyObj_id;}
}
```

Figure 3.2: MyObj.jr

```
public class Foo        {
        private int Foo_id;

        public Foo(int i)       {
                Foo_id = i;
        }

        public op void print()  {
                System.out.println("Foo id is: " + Foo_id);
                System.out.println("vm id is: " + ((Myvm)(vm.thisvm)).GetID());
                System.out.println("vm name is: " + ((Myvm)(vm.thisvm)).GetName());
                System.out.println("vm MyObj is: " + ((Myvm)(vm.thisvm)).GetObj());

                cap void (int) temp = ((Myvm)(vm.thisvm)).GetCap();
                send temp(Foo_id);
        }
}
```

Figure 3.3: Foo.jr

```
import edu.ucdavis.jr.JR;

public class Main       {
        public static op void myOp(int x);
        public static cap void (int) myCap;

        public static void main(String [] args) {
            myCap = myOp;

            // test capability
            vm vm1 = new Myvm(10, "vm1", new MyObj(999), myCap);

            // test operation
            vm vm2 = new Myvm(20, "vm2", new MyObj(888), myOp);

            remote Foo f = new remote Foo(100) on vm1;
            remote Foo g = new remote Foo(200) on vm2;

            System.out.println("f: ");
            f.print();
            System.out.println("g: ");
            g.print();

            System.out.println("---------------");

            inni void myOp(int x)   {
                    System.out.println("receive from Foo id: " + x);
            }
            inni void myOp(int x)   {
                    System.out.println("receive from Foo id: " + x);
            }

        }
}
```

Figure 3.4: Main.jr

When the program executes, first it creates two parameterized VM vm1 and vm2 with argument types corresponding to those specified in the constructor of class Myvm. Then it creates two remote objects f and g of type Foo on vm1 and vm2 respectively. After that the print() function of both f and g will be called. Inside the print() function of Foo, there are print statements to print out the values of its private member variable Foo_id and other variables previously stored in the parameterized VM that the remote object is created on, which are vm_id and vm_name. When Foo calls the Myvm function GetObj(), it in turn calls GetObjID() inside MyObj so that MyObj_id will be returned to Foo. Before the print() function exits, it does a send to the operation referenced by the capability stored in the current VM, so that the two inni statements in Main can service those invocations and print out the two acknowledgements. The program's output is shown in Figure 3.5; notice that the output of this program is deterministic because there is no concurrent activity involved in the code.

```
Foo id is: 100
vm id is: 10
vm name is: vm1
vm MyObj is: 999
Foo id is: 200
vm id is: 20
vm name is: vm2
vm MyObj is: 888
---------------
receive from Foo id: 100
receive from Foo id: 200
```

Figure 3.5: Program's Output

Chapter 4

Modifications in the JR Implementation to Incorporate the Extension

4.1 The Overall Idea

"When a user starts execution of a JR program, the RTS creates one JR virtual machine (JRVM) on the local physical machine. After initializing the JRVM, the RTS then executes the program's main method. Each JRVM executes in a single Java virtual machine. JRVMs exchange messages using Java's Remote Method Invocation (RMI). Each JR virtual machine is a small Java program that provides a thin layer over the standard Java virtual machine to allow remote object creation. To create a new virtual machine, a thread must contact a centralized virtual machine manager, called JRX, which plays a role similar to that of SR's SRX. JRX contacts (via rsh or a specified alternate program) the physical host on which the virtual machine is to be created and initiates the execution of the jrvm program. Once created, a virtual machine executes a remote method invocation to contact JRX and register itself as ready to receive requests. A reference to the virtual machine is then returned to the instantiating thread so that remote objects can subsequently be created on the virtual machine. Processes, variables, and operations in an instance exist entirely within a single virtual machine. As in Java, the static parts of a class are created automatically as needed; however, a separate instance of the static parts is created on each virtual machine that needs them." [3]

The previous translator design for non-parameterized VMs is maintained. Normal VMs works the same way as in previous JR. Parameterized VMs are created in the same manner as normal VM. New functions added specifically for parameterized VMs are independent from those for normal VMs so that there will not be any interference. All parameterized virtual

machines classes actually extend from class jrvm. JR continues to treat vm as a keyword instead of a real class. Class JRX_impl is responsible for telling jrvm what vm subtype to instantiate. On the other hand, the translator is responsible for translating user's calls to vm subclass constructor to appropriate createVM() function calls in JRX_impl.

4.2 Specific Changes

Below are changes in the implementation for incorporating parameterized VMs, organized by each type of class (examples presented in Chapter 3) it processes:

4.2.1 Subclass that Extends vm (Myvm in Figure 3.1)

As class Parser parses the clause

```
public class Myvm extends vm
```

vm is changed and stored internally as edu.ucdavis.jr.jrx.jrvm, and the clause becomes

```
public class Myvm extends edu.ucdavis.jr.jrx.jrvm
```

so that everything defined inside this subclass will be semantically checked against the jrvm base class.

Formal parameters for subclass constructor can be of any type including primitives, capabilities, operations and user defined serializable objects.

Since the jrvm constructor throws a RemoteException, it would be an error if the subclass constructor does not throw this exception. As the translator parses the constructor, class SourceMember checks if it throws a java.rmi.RemoteException, it will insert one if it does not.

At this point, the subclass constructor looks like the following internally for the given example:

```
public Myvm(int vm_id, String vm_name, MyObj vm_obj,
  cap void (int) vm_cap) throws java.rmi.RemoteException {
        super();
        this.vm_id = vm_id;
        this.vm_name = vm_name;
        this.vm_obj = vm_obj;
        this.vm_cap = vm_cap;
}
```

The existing jrvm constructor has been changed from private access to protected so that it can be called by derived classes. Although the translator will add two arguments to super() inside the subclass constructor when it does code generation (described later), at this point all it sees is a call to the default constructor. On the other hand, since the jrvm constructor takes two parameters, the translator will complain that the call to super() inside the subclass constructor has no argument (the call to super() is automatically generated by the translator when the programmer does not provide one). In order to suppress this error, a dummy constructor that takes no parameters and has an empty body is put in jrvm, and it will never get called.

During code generation, SourceMember inserts two strings as the first and second argument to the subclass constructor to represent the source and destination host, and MethodExpression in turn inserts these two strings as arguments to super. For the above example,

public Myvm(int vm_id, String vm_name, MyObj vm_obj, cap void(int) vm_cap)

becomes

public Myvm(String JRname, String JRhost, int vm_id, String vm_name, MyObj vm_obj, cap void (int) vm_cap)

and the call to the jrvm constructor changes from

super();

to

```
super(JRname, JRhost);
```

so that instances of vm subclass will be created properly by invoking the correct base class constructor. JRname and JRhost are chosen here since the "JR" prefix has been reserved for internal identifiers, so as to avoid naming conflicts with other formal parameters.

These two parameters are inserted during code generation instead of during parsing to prevent the translator from complaining about the user's call for the subclass constructor such as

```
vm vm1 = new Myvm(10, "vm1", new MyObj(999), myCap);
```

has insufficient parameters and parameter types mismatch.

4.2.2 User Defined Object Used as Member Fields in Extended vm Class (MyObj in Figure 3.2)

There is no internal change in the translator, however, this class must be serializable as mentioned before.

4.2.3 Class that Creates Extended vm (Main in Figure 3.4)

During type checking, whenever the translator gets a call to a vm subclass constructor such as

```
vm vm1 = new Myvm({expressions});
```

NewInstanceExpression temporary changes the type of the right hand side to VM and performs other necessary semantic checks. The type is changed back to the original before the check function quits. This temporary change is necessary to prevent type mismatch since the left hand side is of type VM[1], and Myvm extends from jrvm instead of VM.

During code generation, the right hand side of the above expression is translated to a call for the overloaded function createVM() in JRX_impl. The newly added createVM() function

[1] Note: VM is a class defined in jrx.

takes three additional arguments at the end: a string representing the name of the vm subclass, an array of the subclass constructor argument types, and their initial values. The parameters and their types supplied by the programmer is made into two arrays so that they can be passed to createVM(). For example,

```
vm vm1 = new Myvm(10, "vm1", new MyObj(999), myCap);
```

is translated to a call to createVM with the last three parameters as

```
Myvm
new Class [] {Integer.TYPE, Class.forName("java.lang.String"),
Class.forName("MyObj"), Class.forName("Cap_intTovoid")}
new Object [] {new Integer(10), "vm1", new MyObj(999), cap_myCap_intTovoid}
```

JRX_impl is modified to include two member fields to store the two arrays passed in from the newly added createVM(). After saving the two arrays, createVM() will then create new jrvm by calling the main function in jrvm using command line format. The name of the vm subclass is passed as the eighth argument to main. In order to classify a regular VM with a parameterized VM, the string "VM" is passed in if it is a regular VM or the name of the subclass is passed if it is an extended VM. The Main function of jrvm will ask JRX_impl for parameter types and initial values for the subclass constructor by calling functions getParamType() and getParam(). Then with the name of the subclass, parameter types and initial values, the appropriate constructor will be called to create a new jrvm.

4.2.4 **Class for Remote Object** (Foo in Figure 3.3)

The syntax for referencing subclass member fields is reproduced here:

```
((subclass)(vm.thisvm)).fieldname
```

Since vm.thisvm returns type VM, in order to make the above casting legal, a condition is added in class Environment to allow castings from type VM to any types extended from jrvm.

During code generation, the above expression becomes

((subclass) VM.currentVM().jrVM).fieldName

where the string ".jrVM" is inserted by CastExpression so that the entire casting will be performed on jrvm instead of VM.

4.3 Restrictions

The above changes include two minor restrictions on using parameterized VMs, as summarized below:

1. Formal parameters for vm subclass constructor cannot be named JRname or JRhost. (Section 4.2.1)
2. Objects that serve as member fields in vm subclasses must be serializable (as originally discussed in Section 2.5).

Chapter 5

A Problem with Parameterized Virtual Machines

5.1 The Problem

In earlier versions of JR, users can create remote objects either on the main VM that is created at the very beginning of program execution, a different VM that is on the same physical machine as the main VM, or a VM that is on another physical machine. However, in the new version, remote objects that have member functions dealing with parameterized VMs will cause a run time error if any of their instances is created on the main VM since it is of type jrvm. The following example illustrates the problem.

Consider the program in Chapter 3 but with a minor change in Main (Figure 3.4), so that the line that declares the remote object f becomes:

```
remote Foo f = new remote Foo(100);
```

In this case f is created on the main VM while g is created on another VM of type Myvm. When Main calls f.print() it will generate a run time error as it executes the line

```
System.out.println("vm id is: " + ((Myvm)(vm.thisvm)).GetID());
```

since the casting is invalid. That is, the main VM is not a Myvm.

5.2 Possible Solutions

One easy way to solve the problem is to have each JR program take a command line argument that specifies the type of the main VM. The translator will create the main VM with the given type. However, it will not work if more than one vm subtype is being used in a single program since the type of main VM is fixed to one subtype. For example, if the user specifies a command line argument Myvm1, the translator will create the main VM of type Myvm1. Then

the following program will cause an error when foo2.print() is called since the main VM is of type Myvm1 instead of Myvm2.

```
public class Main     {
        public static void main(String [] args)      {
                remote Foo1 foo1 = new remote Foo1();
                remote Foo2 foo2 = new remote Foo2();
                foo1.print();
                foo2.print();
        }
}

public class Foo1     {
        public op void print()          {
                System.out.println("id is: " + ((Myvm1)(vm.thisvm)).GetID());
        }
}

public class Foo2     {
        public op void print()          {
                System.out.println("id is: " + ((Myvm2)(vm.thisvm)).GetID());
        }
}

public class Myvm1 extends vm    {
        private int vm1_id;
        public Myvm1(int vm1_id)  {
                this.vm1_id = vm1_id;
        }
        public int GetID()     {return vm1_id;}
}

public class Myvm2 extends vm    {
        private int vm2_id;
        public Myvm2(int vm2_id)  {
                this.vm2_id = vm2_id;
        }
        public int GetID()     {return vm2_id;}
}
```

Figure 5.1 Program that Causes Problem with Fixed Main VM Type

This observation suggests that the only solution is to allow the main VM to change its type to any of its subclasses and be able to change back to the base type. Here is the proposed syntax:

```
change CurrentVM to Myvm(1);
```

or

```
vm.change(new Myvm(1));
```

for changing the main VM to its subtype, and

```
change CurrentVM to vm;
```

for changing the main VM back to the base type. The second version of the planned syntax for changing the main VM to its subtype is analogous to the syntax for obtaining a reference for the current VM which is the following reserved expression:

```
vm.thisvm
```

One restriction with this is that the argument to the change function must be a call to a subclass constructor and it cannot be a reference to another existing VM of any type.

The change in the translator will be minimal. Whenever the type of the main VM needs to be changed from base type to one of its subtypes, a new instance of the subclass will be created with the provided parameters and assigned to the main VM. The new instance will inherit from `jrvm` by invoking a copy constructor, which copies all the main VM data such as global and static variables. When the main VM changes from a subtype back to the base type, a new instance of the base `vm` class will be created and assigned to the main VM by calling the copy constructor of `jrvm` to copy data from the `vm` subtype. Data copying is preferred instead of just having an object reference inside the subclass that points to the old VM because the program might need to reference those global or static data after main VM has changed to a subtype. An

object reference within a subclass would require extra member field de-referencing. Although the translator can be made to handle extra de-referencing at compile time, the program will only run correctly when it does not have any concurrent activity that might cause race conditions. When the program is run concurrently with two or more processes where the order of program execution is non-deterministic, static de-referencing will fail.

With this new change, whenever users need to deal with remote objects that manipulate parameterized VMs members, they will have to change the type of the main VM to the appropriate subtype prior to accessing those members and change back to the base type when they are done. For code that runs concurrently, users will need to apply mutual exclusion mechanisms to prevent race conditions from occurring during the type change. The following program provides an example where the use of a semaphore is necessary for it to run correctly:

```
public class Main    {
        private static sem mutex = 1;
        private process p1   {
                remote Foo1 foo1 = new remote Foo1();
                P(mutex);
                change CurrentVM to Myvm1(1);
                foo1.print();
                change CurrentVM to vm;
                V(mutex);
        }
        private process p2   {
                remote Foo2 foo2 = new remote Foo2();
                P(mutex);
                change CurrentVM to Myvm2(2);
                foo2.print();
                change CurrentVM to vm;
                V(mutex);
        }
        public static void main(String [] args)      { }
}

public class Foo1    {
        public op void print()         {
```

```
                System.out.println("id is: " + ((Myvm1)(vm.thisvm)).GetID());
        }
}

public class Foo2    {
        public op void print()        {
                System.out.println("id is: " + ((Myvm2)(vm.thisvm)).GetID());
        }
}

public class Myvm1 extends vm   {
        private int vm1_id;
        public Myvm1(int vm1_id)  {
                this.vm1_id = vm1_id;
        }
        public int GetID()     {return vm1_id;}
}

public class Myvm2 extends vm   {
        private int vm2_id;
        public Myvm2(int vm2_id)  {
                this.vm2_id = vm2_id;
        }
        public int GetID()     {return vm2_id;}
}
```

Figure 5.2 Example Program for Changing Main VM Type

If the use of semaphore is omitted above, race conditions can happen between the following two fragments within process p1 and p2:

```
change CurrentVM to Myvm1(1);
Foo1.print();
change CurrentVM to vm;

change CurrentVM to Myvm2(2);
Foo2.print();
change CurrentVM to vm;
```

If foo1.print() and foo2.print() are called one after the other, it will reproduce the problem that was caused by the code in Figure 5.1. On the other hand, a run time error will happen if change CurrentVM to Myvm1(1) and change CurrentVM to Myvm2(2) are called one after the other since those two lines only take care of changing the main VM from base type

to subtype but not from one subtype to another subtype. Therefore, a semaphore is needed to ensure that no context switch occurs during the execution of those two blocks of code.

Comparing between declaring remote objects on appropriate type of parameterized VMs and being able to change the main VM type after its creation, the latter seems to bring too much trouble to users and greatly degrades run time efficiency since it needs to create a new instance of VM each time main VM changes its type. In addition, users are likely not to need this feature as frequently as other features in JR; therefore, the proposed solution is not implemented.

Chapter 6

Related Work

The idea of virtual machine parameterization is new compared to what is found in common concurrent programming tools and standards.

MPI, which stands for Message Passing Interface, is a library specification for message passing [2]. In achieving parallelism, applications start new processes through some spawning routines, which take the name of the program, arguments to the command, maximum number of processes to start and other system related parameters to execute the argumented program on a new virtual machine. There is a group, rank pair to identify each process and processes communicate with each other through the use of intercommunicators returned after each process creation.

On the other hand, PVM, which stands for Parallel Virtual Machine, is a software package that allows a number of computers running on possibly different platforms to be used as a single large parallel computer [4]. An integer task identifier (TID) is used to distinguish each PVM task and for sending or receiving messages. However, users are not allowed to pick their own TID since it must be unique for each task. Nevertheless, users are allowed to group similar tasks together to be used for broadcasting messages.

Both mechanisms above provide ways to execute processes across multiple virtual machines. However, users are at most given the flexibility to identify similar processes as a group, they are not allowed to store machine dependent attributes on each virtual machine.

Chapter 7

Conclusion

With parameterized virtual machines support in the JR concurrent programming language, virtual machine specific data can be stored in a VM object instead of storing a copy of the data in each remote object created on one VM. The detail specification of parameterized VMs is provided in Chapter 2 and the example program in Chapter 3 demonstrates an application of parameterized virtual machines. Implementation particulars can be found in Chapter 4.

So far the extension on virtual machines only saves programmer effort and physical memory, it does nothing to shorten the execution time. JR virtual machines would be able to speed up the execution process if JR were to provide some form of quantified virtual machines, where multiple virtual machines can be created at the same time. In current JR, users will need to do something similar to the following to create multiple virtual machines:

```
vm vm_name[2][2] = new vm[2][2];
for (int i = 0; i < 2; i++)       {
  for (int j = 0; j < 2; j++)
    vm_name[i][j] = new vm() on pc11;
}
```

With quantified virtual machines, the above code fragment can be written as:

```
vm [] vm_name = new vm((int i = 0; i < 2; i++), (int j = 0; j < 2; j++)) on pc11;
```

When parameterized VMs are used, declaration of multiple parameterized VMs such as the following in current JR

```
vm vm_name[2][2] = new vm[2][2];
for (int i = 0; i < 2; i++)       {
  for (int j = 0; j < 2; j++)
    vm_name[i][j] = new pvm(i, j) on pc11;
}
```

can be written as

```
vm [] vm_name = new pvm(((int i = 0; i < 2; i++), (int j = 0; j < 2; j++)), (i, j)) on pc11;
```

with quantified virtual machines.

During execution, instead of creating each individual virtual machine sequentially, multiple VMs will be created concurrently. This will speed up the performance of instantiating VMs especially when they are created on different physical machines.

The implementation of quantified virtual machines creation can make use of the JR concurrent invocation statement. The translator takes the following declaration of quantified VM:

```
vm [] vm_name = new vm((int i = 0; i < 2; i++), (int j = 0; j < 2; j++));
```

and makes it into a co-statement with the instantiation of each element of the vm array to be a co-arm with non-blocking call to the vm constructor so that different VM instances can be created at the same time. The following is the generated code for the previous regular VM declaration statement:

```
vm vm_name[2][2] = new vm[2][2];
co vm_name[0][0] = new vm();
[] vm_name[0][1] = new vm();
[] vm_name[1][0] = new vm();
[] vm_name[1][1] = new vm();
```

Quantified co-statement can also be used in this case to shorten the length of the generated code. The following code operates the same as the one above but it is much shorter:

```
vm vm_name[2][2] = new vm[2][2];
co ((int i = 0; i < 2; i++), (int j = 0; j < 2; j++)) vm_name[i][j] = new vm();
```

If we need to create multiple parameterized VMs, such as the example we had before (reproduced below):

```
vm [] vm_name = new pvm(((int i = 0; i < 2; i++), (int j = 0; j < 2; j++)), (i, j));
```

It will be translated into the following:

```
vm vm_name[2][2] = new vm[2][2];
co vm_name[0][0] = new pvm(0, 0);
[] vm_name[0][1] = new pvm(0, 1);
[] vm_name[1][0] = new pvm(1, 0);
[] vm_name[1][1] = new pvm(1, 1);
```

or the following with quantified co-statement:

```
vm vm_name[2][2] = new vm[2][2];
co ((int i = 0; i < 2; i++), (int j = 0; j < 2; j++)) vm_name[i][j] = new pvm(i, j);
```

At this stage the design of co-statement is only capable of handling multiple operation invocations. In order to use it to simplify the implementation of quantified virtual machines, co-statement needs to incorporate multiple new object instantiations. However, the co-statement in JR has not been fully developed, we will put this proposal on hold and come back to it when the co-statement is ready for further modification.

References

[1] JR Implementation <http://www.cs.ucdavis.edu/~olsson/research/jr/>

[2] MPI Standard 2.0 of July 18, 1997. Argonne National Laboratory. September 10, 2001 <http://www-unix.mcs.anl.gov/mpi/mpich/>

[3] Olsson, Ronald and Keen, Aaron. The JR Programming Language: Concurrent Programming in an Extended Java. Kluwer Academic Publishers, to appear in 2004.

[4] PVM: Parallel Virtual Machine. Oak Ridge National Laboratory. November 24, 2002 <http://www.csm.ornl.gov/pvm/>