A RISC Virtual Machine for Internet Programming

Boumediene Belkhouche[a], Monisha Pulimood[b], Mark McKenney[c]

[a] *Faculty of Information Technology, UAE University, Al Ain, UA*
b.belkhouche@uaeu.ac.ae
[b] *Department of Computer Science,The College of New Jersey*
pulimood@tcnj.edu
[c] *Department of Computer Science, Texas State University-San Marcos*
mckenney@txstate.edu

Abstract

We discuss the design, implementation, and evaluation of a RISC virtual machine (RVM) supporting mobile computations. Our Mobile Computational Model defines the necessary structures to support the Internet programming paradigm. Mobility in a heterogeneous environment and efficient execution of mobile computations are assured through the RVM. A prototype implementation of the model, particularly the RVM, provides a proof-of-concept. Preliminary testing shows encouraging results leading us to conclude that our RVM is a viable approach for handling the heterogeneity of the Internet, while providing an efficient, fast and secure environment for mobile computations.

Keywords: mobile computation, closure, RISC virtual machine, Internet programming

1. Introduction

The focus of this research is the formal design, efficient implementation, and analysis of a computational model for *mobile compuations to support Internet Programming.*This new programming and architecture paradigm impacts the semantics, implementation, and run-time structures of programming languages for the Internet.

The general trend is for the resources available on the Internet to become seamlessly accessible to users as if they resided on an individual desktop, thus making the Internet look like a personal *computing platform.* To realize such a vision, a framework to support mobile computations is required. A mobile computation may be a simple command, a query, or a sophisticated

program that roams through the Internet to efficiently accomplish its task. Traditional languages and environments being adapted to this new task do not seem to effectively the requirements for Internet programming. That is, there is a need for an *Internet Computational Model*.

Heterogeneity of computer systems and architectures has been challenging the research community since the late 1950's. Attempts at providing a homogeneous computational model were initiated by the UNCOL (UNiversal COmputer Language) proposal [1]. Unfortunately, a universal language unifying various languages and architectures proved overly complex. Subsequent developments (e.g., Pascal P-code) restricted the scope to one single language [2]. The topic went dormant for a while until the advent of "internet programming" which engendered a fundamental issue, i.e., the issue of *mobility of computations*.

Mobility is probably the most important and critical concept introduced by Internet programming. It impacts the semantics of the naming structures, the typing system, and the run-time structures. Mobility can be defined as the ability of an entity to move from one execution site to another. Whether it is data or computation, a mobile entity must always be consistently interpreted at each site it visits. There are several forms of mobility: data mobility, code mobility, agent mobility, and computation mobility, with this last form being the most general. Mobility of a computation involves a context-switch and transfer from the current execution site to another site for resumption. This implies a change in the execution environment, the representation, and the bindings (e.g., variables, parameters, I/O files, windows). To support this transfer, the entire "closure" of the computation is moved. A closure consists of the code and the run-time structures. The run-time structures typically include the data area (symbol table and values), the general registers, the special registers, and the heap. For example, in a stack-based implementation, the run-time structures consist of the stack of activation records, the display vector, the heap, and the registers. The ability to interpret closures uniformly (i.e., interoperability) between different implementations can be supported by at least two approaches (there are variations that fall in-between). The first approach provides a universal "virtual machine" that resides at every site over the Internet. The Internet is thus conceptually reduced to a single uniform machine. That is, the representation and interpretation of closures are made uniform throughout the Internet. The second approach requires the translation of the closure of a mobile computation from one machine representation to another while maintaining the intended semantics. This translation process is highly complex, for it involves dynamically mapping from one encoding to another several

diverse structures, such as instruction sets, run-time structures, heap layout, and registers. In both approaches, the "name space"' associated with a computation cannot be determined statically. Instead, it has to be reconstructed dynamically upon every move to recover the environment and the bindings. The name space of a computation consists of all the names that the computation can reference. These include variables, types, functions, labels, files, and library resources.

In Figure 1, we present a classification of some existing languages and systems based on the kind of support they provide for mobility. Languages that provide support for on-the-fly compilation and mobility produce code that is 'on-line portable' or 'mobile'. Such languages can further be characterized by the extent of support they provide for mobility. They may support varying degrees of mobility through either code mobility, agent mobility, or closure mobility. Code mobility requires the transfer of the source code for execution from one site to another. Agent mobility allows autonomous agents and their closure to migrate from site to site. Closure mobility supports the migration of an executing code and its closure, thus allowing partial execution at a given site [3]. The complexity of the computational model increases as we move from code, to agent, to closure mobility. Several languages / systems to support mobility over the Internet have been proposed and are being used. Among the many variations are Java [4, 5], Telescript, Safe-Tcl, Emerald [6, 7], Distributed Oz [8, 9], Inferno [10, 11], Obliq [12, 13], Ambients[14, 15], Omniware [16], Mobile UNITY [17], NOMADS [18, 19], ARA, TACOMA [20, 21], μCode [22], D'Agents [23, 24], MobileML [25], FACILE, and many extensions of the Java Virtual Machine (e.g., Aglets [26], Odyssey, Voyager [27, 28]). Attempts at modeling mobility fall into three categories: (1) code mobility (Java, Tcl) where program code, either source text or bytecode, is moved; (2) agent mobility (Telescript, D'Agents) where a self-contained object is moved; and (3) closure mobility (Obliq) where an active computation and its context are moved. Cases 1 and 2 are classified as *weak mobility*, and case 3 is classified as *strong mobility*.

Our main goal then is to develop a computational model to support mobile computations by addressing design and implementation issues associated with closure mobility. The key contributions of this research are:

- Design of a mobile computational model specifically for Mobile Computations, that provides efficient support for mobility and manages the heterogeneity of the Internet.

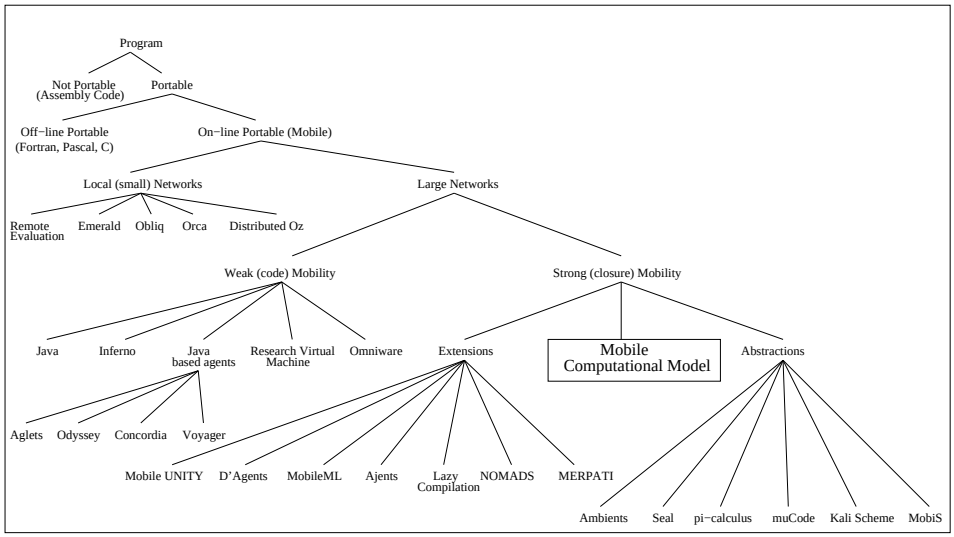

Figure 1: Classification of Languages and Systems Based on Mobility Support

- Design and prototype implementation of a mobile computation system comprising of a mobile computation language for writing mobile computations, a mobile computation manager that manages the administrative aspects of mobile computations, including security, communication and state of execution, and a RISC-based virtual machine that executes mobile computations, starting from the point where they left off execution at the previous host.
- Optimization to improve efficiency.
- Development of an abstract specification of the RVM to enable us to support formal analysis of the behavior of the RVM.

We overview the design of our mobile computational model in section 2. In section 3, we describe the major components of the RVM and its runtime structures. A comparison of the RVM and the Java Virtual Machine is elaborated in section 4. In section 5, we use transition rules to capture the behavior of the computational model. Our results are summarized in section 7.

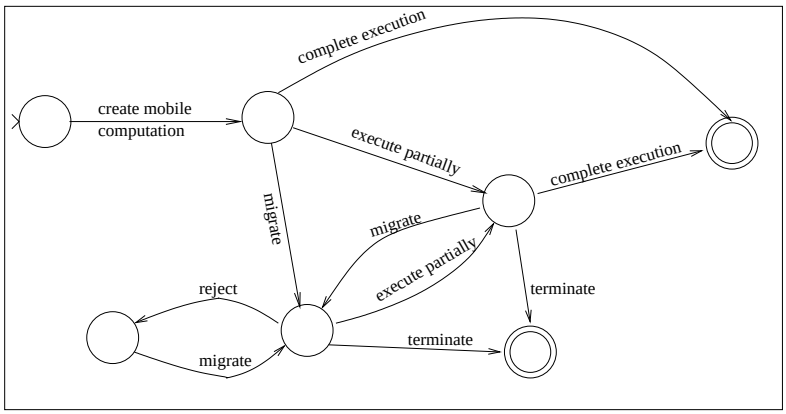

Figure 2: State Transitions of a Mobile Computation

2. A Mobile Computation Model

Our model for Internet Programming, the Mobile Computational Model, is based on closure mobility and has been specially designed to support the Internet Programming paradigm. Our proposed RISC virtual machine (RVM) is at the core of our model and provides the conceptual homogeneity required for Internet Programming. We consider a mobile computation to be a computational unit that starts execution at one site and may subsequently move to another site to continue execution. Such a transfer may be initiated at any point of execution. The current state of execution is transferred along with the mobile computation so that execution can be safely resumed at the point where it left off. Closure mobility supports the migration of an executing code and its closure, thus allowing partial execution at a given site. A host may deny resources to a computation for security or other reasons, in which case the computation is rejected and sent to another host. Computations that merely hop from site to site tying up network resources are terminated. We make no assumptions about the architecture of the accepting host, or whether it is stationary or mobile. Figure 2 gives an abstract view of the possible states and transitions of the mobile computation.

In our mobile computational model, a programmer can write an application in the Mobile Computation Language - a high level object-oriented language. The application is compiled to a mobile computation after rigorous type-checking. The Mobile Computation Manager (MCM) sends the computation to the destination host. The MCM at the destination receives the computation and verifies its credentials. If the computation has the re-

quired permissions, it is executed by the RVM. On encountering a 'move' command the mobile computation suspends execution. The MCM captures the closure and sends the mobile computation and its closure to the specified host. When the computation completes execution, the MCM sends it back to the originating host. Figure 3 shows a schematic of the process flow.

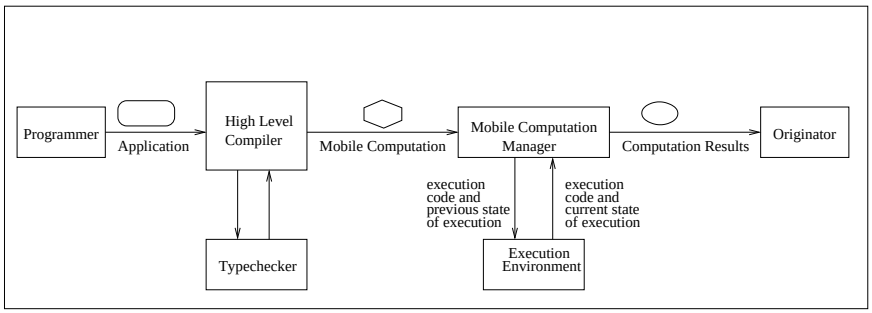

Figure 3: Mobile Computation Process Flow

We conceptualize the Internet as a collection of hosts each set of which belongs to a computational subnet based on certain properties, such as access rights and resources. A mobile computation is given permission to execute on specific hosts. The hosts within the computational subnet of a mobile computation are said to be its 'domain'. Hosts in a computational subnet may have varying architectures. The Mobile Computation Manager present on every host in the computational subnet creates a homogeneous substratum and facilitates execution of a mobile computation. Figure 4 shows the structure of the computational subnets. The Internet (big blob) consists of several subnets (nested blobs). Within each subnet, mobile computations (MC_x) roam. A meta-database to keep track of mobile computations is maintained within each subnet.

2.1. Mobile Computation

A mobile computation consists of the code to be executed, its closure including the point of resumption of execution, and the headers containing information about the locations to be visited, the type of computation, and user identification. These headers provide the necessary information to the Mobile Computation Manager to enable it to initiate the appropriate actions. Figure 5 shows the anatomy of a mobile computation.

A 'live' mobile computation . is one that has been created, and has not yet been terminated. It may be in the process of being transmitted from one

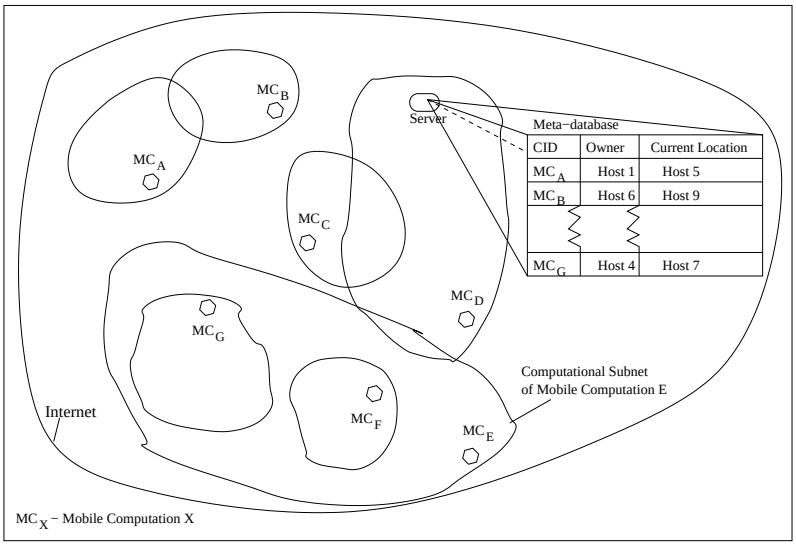

Figure 4: Computational Subnets

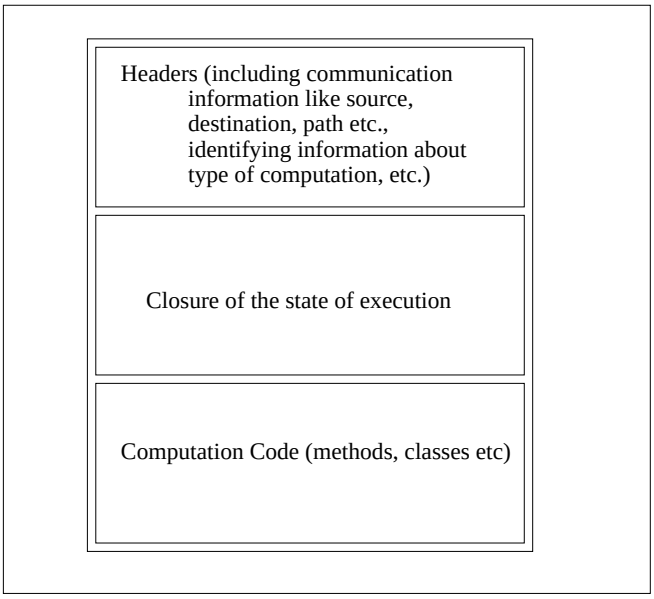

Figure 5: Anatomy of a Mobile Computation

host to another, currently being executed at some host, or waiting at a host for the some 'action' by a Mobile Computation Manager (MCM).

An 'action' . It may be to transmit a computation to another host; to assign some resource at a host to a computation; to obtain information from a mobile computation; to send information to a mobile computation; or to terminate a computation. Resources may include memory, processors, software applications, and hardware devices.

A 'terminated' mobile computation . It is one that has been been halted by the MCM due to some error condition. If the Mobile Computation Manager is unable to resolve the exception, the terminated computation is sent back to the originator for appropriate handling of the situation.

A 'completed' mobile computation . It is one that has successfully completed execution. Such a computation can be sent back to the originator with the results.

2.2. The Mobile Computation Manager

Figure 6 shows the architecture of a host component of the mobile computational model. It consists of a high-level language (HLL) translator, a Mobile Computation Manager (MCM), and a RISC virtual machine (RVM).

The Mobile Computation Manager (MCM) resident on a host is responsible for managing the mobile computations that wish to execute at that host. It is also responsible for safeguarding the host from malicious or insecure mobile computations. It ensures that only valid mobile computations with appropriate access rights are allowed to execute. Some of the components of the Mobile Computation Manager and the interactions between them are shown in Figure 7.

The MCM may be viewed as a daemon process that lies dormant within the operating system until it is informed by the Event Monitor that there is a change in the current state of the mobile computation environment. The Activity Manager initiates and oversees the various tasks performed by the MCM (refer to Figure 8). It determines what the mobile computation is required to do next and places it in the appropriate queue.

A meta-database, stored on a server in the system, maintains overall information about each host in the system to facilitate the process of locating a mobile computation. Depending on the size of the system, the meta-database may be replicated on several servers at strategic points in the system.

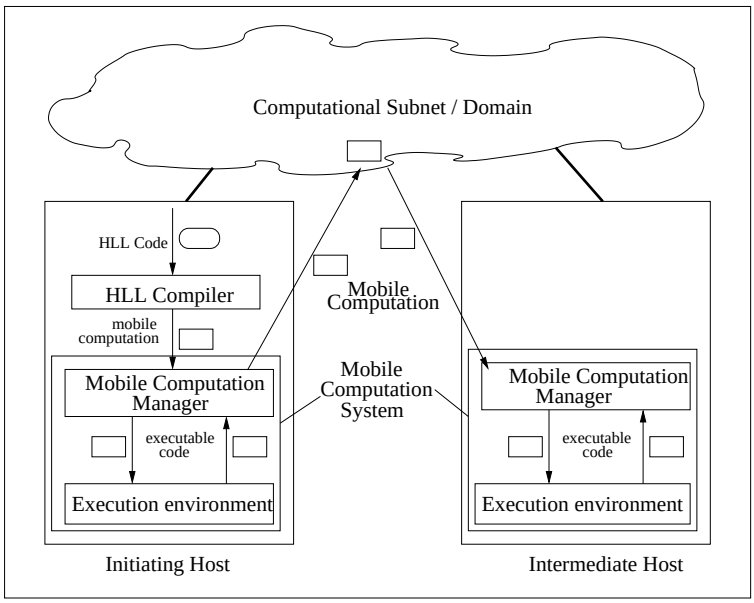

Figure 6: Architecture of Hosts

2.2.1. Local Management

Each host in the mobile computational model maintains information about the mobile computations initiated by it, i.e. the computations it 'owns'. When a mobile computation is created, the MCM gives it a unique computation identifier (CID). This CID is based on the id of the user who created the computation, the id of the location where it was created, and the time it was created at. An entry is then made in the host's database, for that computation. A closure 'map' is also created for the mobile computation and stored in the database. At this point it will be empty. The programmer may specify a 'maximum number of hops' for the computation. This will limit the maximum number of hosts it is allowed to visit without execution, that is without a change in closure, and will prevent aimless wandering of the mobile computation. When the computation is sent to another host, the computation's entry is updated with all this information and that about the destination. The Queue Administrator manages the various queues like the Execution Queue, Load Queue, Unload Queue, and Move Queue. The structures required for execution of the mobile computation, including the current state of execution (activation records and program counters), are generated and loaded by the Loader component, in preparation for execution. When a 'move' operation is encountered, the current state and closure

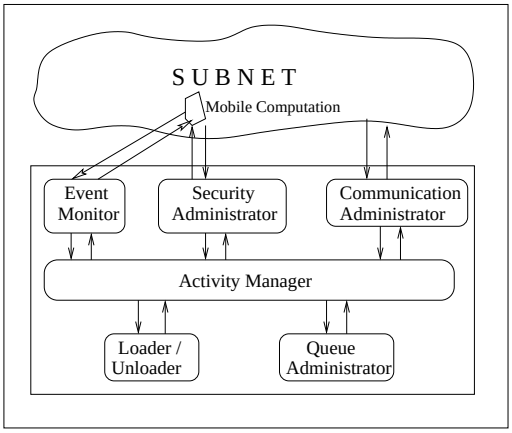

Figure 7: Anatomy of the Mobile Computation Manager

of the execution is captured by the Unloader in preparation for the transfer of the computation to another host.

2.2.2. Security

A computation is allowed to execute only on the hosts in its domain. On arrival of a mobile computation, the Security Administrator decrypts it and verifies its credentials by checking header information associated with the mobile computation. The MCM first checks if the host is in the domain of that computation. If it is not, the computation is rejected and queued to be sent to the next host in its domain. If it is, the computation is verified for authencity. Insecure or illegal computations are queued to be sent to the originating host. If for some reason, the origin is unknown, the computation is assumed to be malicious and is 'terminated'. Once the identity of the computation has been verified, the MCM checks to see if this is the computation's first visit to the site. If it is, the MCM creates a new entry for it. If not, the closure map for the previous entry is checked against that of the computation. The MCM also checks if the computation has been assigned a 'maximum number of hops'. If the maps are identical there is a high chance that the computation is merely hopping from one host to another. This could be because it was genuinely unable to find the resources it needs, or it could be a virus that is attempting to tie up network resources. In either of these cases, or if the 'maximum number of hops' has been exceeded, the mobile computation is rejected and queued to be sent to the originating host for remedial action. If all the necessary conditions have been satisfactorily fulfilled, the mobile computation is queued for execution

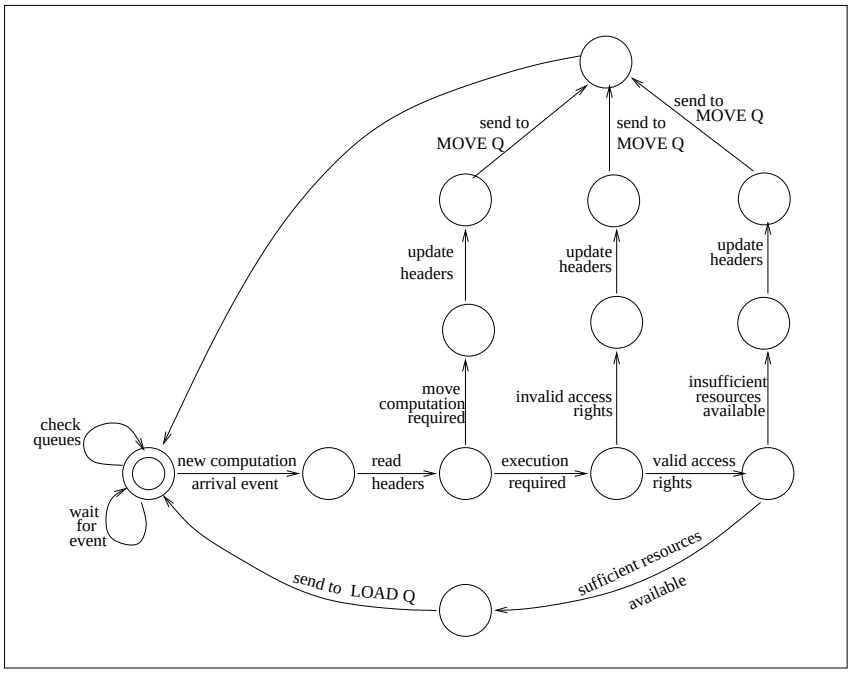

Figure 8: The Activity Manager handling the arrival of a new mobile computation

by the Activity Manager. Otherwise, it is rejected and added to the MOVE Queue. The Activity Manager updates the database with information about the new computation. During execution the computation is also monitored to ensure that it does not exceed the resource usage limits. If there is a resource overrun, the execution is halted. Before a mobile computation is transferred to another host, it is re-encrypted by the Security Administrator. The Activity Manager makes the necessary updates in the database and sends the computation to the MOVE Queue.

2.2.3. Communication

A computation may communicate with other live mobile computations by exchanging 'parcels' under closely controlled circumstances. This is to avoid unsafe data being passed to a verified computation in an attempt to subvert it or its host. During execution, a mobile computation may encounter an instruction to communicate the values of some variables (give information) to another mobile computation. This information is packaged into a parcel and addressed to the recipient computation. Since the computation has been verified to be trustworthy, it is assumed that the parcel is

safe. The Communication Administrator then sends out a message to the recipient informing it that the parcel is waiting for it. A computation, MC_A, may encounter a requirement to obtain information from another computation, MC_B. The MCM checks if there is a parcel from MC_B addressed to MC_A on the same host. If the parcel is not at the current host, the MCM checks for any messages indicating that MC_A has a parcel waiting for it. If such a message is found, the parcel is tracked and transferred to the current site. Once the parcel is found and received at the site, its contents are verified for security purposes and then handed over to MC_A for its use. If the parcel is not found, the computation is informed so that it can either wait for the parcel or continue execution without it. If the computation decides to wait for the parcel, the MCM frees up all resources being used by it and moves it into a wait state, so that it does not tie up any resources. A parcel within this model can only be accessed by the mobile computation it is explicitly addressed to. This prevents unauthorized accesses to data. The Communication Administrator also monitors the network to ensure that the route specified in the mobile computation is still valid. Any changes to the network, e.g. the next host on the route is currently unavailable, may require modifications to the route.

At periodic intervals the MCM sends an update about the mobile computations in its database to the other hosts in the model. Terminated and completed computations that are no longer physically present at the host are removed from its database. Any unclaimed parcels addressed to such computations are also destroyed.

3. The RISC Virtual Machine (RVM)

The virtual machine in the Mobile Computation System is modeled after the RISC computer architectures. We postulated that RISC-based virtual machines proffer several advantages over CISC-based machines. The Java Virtual Machine (JVM) is a fairly ubiquitous CISC-based virtual machine. Therefore, we analyzed the JVM instruction set ([4], [29]) in order to eliminate specialized instructions and to replace them with simpler more general RVM instructions. The obtained instruction set is small resulting in a lightweight virtual machine. The size of the RVM instruction set is about one-fifth the size of the JVM instruction set.

As shown in Table 1, each RVM instruction is a fixed simple pattern of the form:

<index> <opcode> <operand>

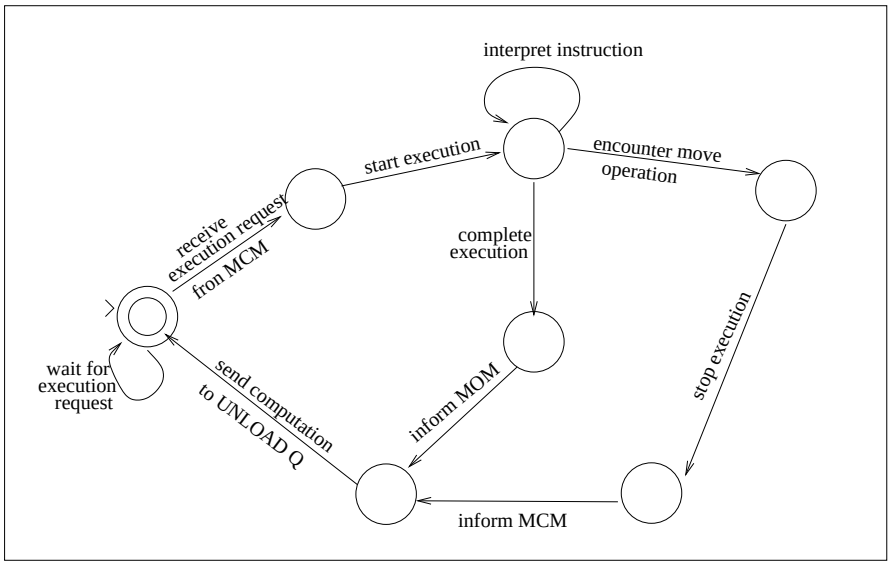

Figure 9: Behavior of the RVM

where *index* is the index of the instruction in the code for this method and *opcode* is the instruction's opcode.

3.1. The Run-time Structures

At an abstract level we view capture the behavior of the RVM by the state transtion diagram shown in Figure 9. The necessary run-time structures of the RVM (see Figure 10) to support this behavior are described below.

Program Counter Stack. The program counter stack (PC Stack) keeps track of the current instruction being executed. It also maintains information about the return point for each calling function.

Object List. The object list maintains information about each object instantiated during execution.

Permanent Activation Record (PermAR). The permanent activation record (PermAR) maintains information about the attributes of each object instantiated. References to the start and end points of an object's attributes are maintained by the object in the object list.

Table 1: RVM Instruction Set

Opcode	**Operand**	**Description**
add / sub / mul / div	*type*	arithmetic operations
and / or	*type*	logical operations
cmpeq / cmpne / cmple / cmplt / cmpge / cmpgt	*type*	comparison operators
endmove	*location*	indicates end of code segment to be executed at the current *location*
get / put	*type*	user input / output
goto	*index*	jumps to instruction at *index*
ifcmp	*index*	jumps to instruction at *index* depending on the value at the top of the execution stack
invoke	*object name*	locates the object referred to by *object name*
jsr	*method name*	starts execution of function referred to by *method name* of the object *object name*
load	*variable name/ constant*	pushes reference to *variable name* or value of *constant* onto stack
move	*location*	initiates migration of mobile computation to *location*
neg	*type*	negates value
ret	*type*	returns value to calling function
stop	*method name*	indicates end of code for the method
store	*type*	stores value of top element on stack into the variable

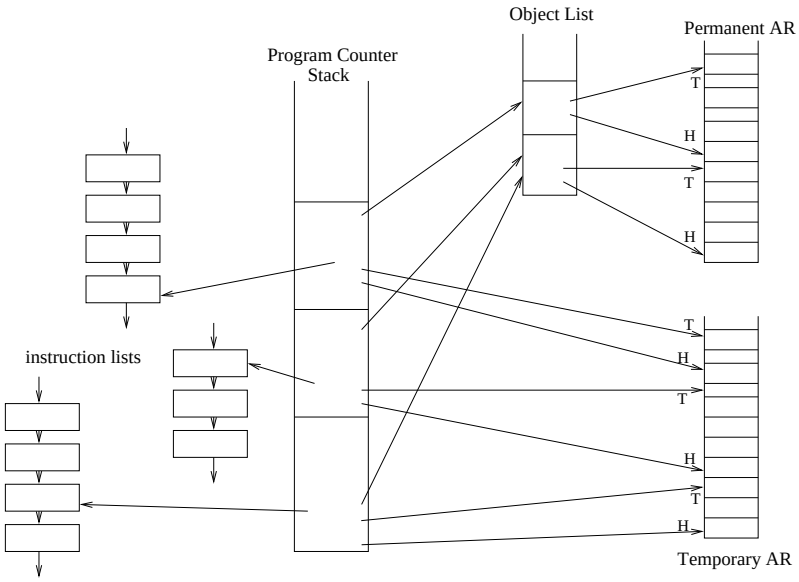

Figure 10: Run-time structures for the RVM

Temporary Activation Record (TempAR). The temporary activation record (TempAR) maintains information about the local variables and parameters for a method. References to the start and end points of the variables of a method are maintained by the program counter item for that method in the program counter stack.

Execution Stack. Each instruction in the code is implemented as a sequence of 'push' and 'pop' operations on the execution stack. An item in the execution stack stores either the value of a constant or a reference to a variable / attribute in one of the ARs.

represents the run-time structures of the RVM.

3.2. Code Execution

Before execution can commence, the code for the mobile computation is loaded into the data structures shown in Figure 11. At the very start of execution, the program counter stack, the object list, and the activation records are empty. The first object instantiated is 'Me' for the main object being executed. The attributes of this object are loaded into the PermAR. The first method executed is 'main( )' which belongs to the 'Me' object. Any variables required for this method are loaded into the TempAR. Entries are

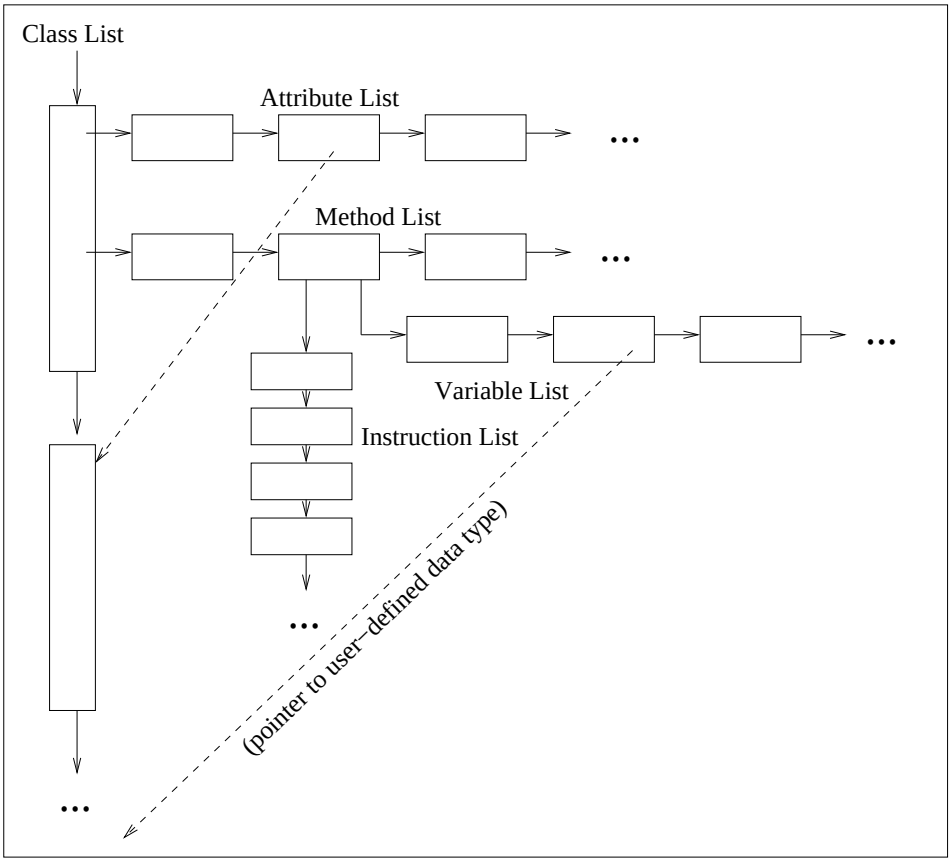

Figure 11: Mobile Computation Data Structures

created in the relevant ARs for any attributes or variables that may be objects with a reference to the parent class. However an object is actually instantiated only when it is used for the first time. This design decision was made with a view to reducing closure overheads.

A program counter item is created with information about the current object being operated on (in the first instance this is 'Me'), the current method being executed (in the first instance this is 'main'), the current instruction being executed, and references to the start (head) and end (tail) of the variables and parameters for the method in the TempAR.

Scope verification is done at the time of compilation of the high-level source code. During the execution of the instructions in a method, there may be a call to another method. If the method belongs to the same object, the method can be referenced directly. If the method belongs to a different

object, the object list is first checked to see if it has already been instantiated. If this is the first use of the object, it is instantiated and added to the object list. The method is then referenced for that object. The top item on the program counter stack stores information about the next instruction to be executed when the control returns to the calling method. A new program counter item is created with information about the called method and pushed onto the program counter stack. The TempAR is updated and values of any parameters are updated as required. The top item of the execution stack contains a reference to the variable or attribute where the return value is to be stored.

A method completes execution when it encounters the 'stop method' instruction. The value to be returned is pushed onto the execution stack, the references to the TempAR are deleted and all entries in the TempAR for this method are removed. If an object in the object list was local to the method, it is also removed from the object list along with references to its attributes, and the corresponding entries in the PermAR. The top item in the program counter stack is popped and the control returns to the new 'top of stack', i.e. the calling function.

If a 'move' operation is encountered before execution is completed, the Unloader captures the state of each of these structures and includes them as part of the mobile computation. On receipt by the new host, the closure is loaded back into the data and run-time structures in the execution area by the Loader component of that host's MCM.

A computation completes execution when the program counter stack is empty. At this time all the other run-time structures are checked to ensure that they are empty.

Memory binding for simple variable / attributes occurs at load time, while that for complex (user-defined) data types occurs at run-time. This implies that if a declared object is never used, it is never instantiated.

During the execution of the instructions in a method, there may be a call to another method. If the method belongs to the same object, the method can be referenced directly. If the method belongs to a different object, the object list is first checked to see if it has already been instantiated. If this is the first reference to the object, then the object is instantiated and added to the object list. The top item on the program counter stack will store information about the next instruction to be executed when the control returns to the calling method. A new program counter item is created with information about the called method and pushed onto the program counter stack. The Temp AR is updated and values of any parameters are updated as required. The top item of the execution stack contains a reference to the

variable or attribute where the return value is to be stored.

To maintain portability of the mobile computations, hardware-specific memory usage, like use of registers, is avoided.

4. Analysis of the RVM

As a proof-of-concept, we implemented a prototype of this Mobile Computational Model as the Mobile Computation System. In order to be able to test generation and execution of mobile computations in the prototype, the system includes an object-oriented programming language, the Mobile Computation Language. The system also includes an implementation of the MCM and a RISC-based virtual machine. The rest of the implementation was written in the C programming language. The implementation is not specialized or optimized for any architecture, so it can be compiled onto any architecture that supports a standard C compiler.

4.1. Performance Evaluation Environment

Mobility of the computations among the Sun SPARC workstations (running the Solaris operating system) and PCs (running the 'cygwin' environment) in our department was tested. A number of small programs were written in MCL and compiled to mobile computations. Identical programs were written in Java and compiled to Java bytecodes. The programs each showcased a specific aspect of computation, like simple arithmetic, if-then-else statements (branch), while loops (iteration) and function calls. The Unix program 'Ptime' was used to measure the user and system execution times for programs with precision to a millisecond. To minimize the effect of fluctuations in the system load, each program was run 60 times and the average execution time was taken. Scripts written in Perl were used to perform the actual runs and compute the averages. We utilized the 'javap' disassembler available with the JVM to obtain human-readable code from the Java bytecode for analysis purposes. Since the JVM does not support migration, we could not compare this feature.

4.2. Contrasting the Instruction Sets

The JVM instruction size is variable due to the varying number of operands. An instruction encodes in one step what can be encoded with simpler instructions in several steps. Some of the information required for the execution of the instruction is implicitly specified in the opcode itself. Other information is gained by popping the current stack. Each Java instruction is of the form:

<*index*> <*opcode*> [<*operand1*> [<*operand2*> ...]] [<*comment*>]

where <*index*> is the index of the opcode of the instruction in the array that contains the bytes of JVM code for this method and <*opcode*> is the instruction's opcode. There may be 0 or more operands for the instruction. The optional <*comment*> is in Java-style end-of-line comment syntax. Since the number of operands can vary and the comment is optional, the size of the instruction can vary.

By carefully evaluating each instruction in the JVM instruction set, we eliminated several specialized instructions and replaced them with simpler, more general RVM instructions. For example, opcodes 21 - 53 in the JVM instruction set are all variations of the load instruction. Replacing all JVM instructions of the form:

<*t*>load_<*n*> and <*t*>load *index*

(where *t* indicates the data type, e.g i for integer, *n* indicates the constant in the constant pool, and *index* specifies the offset of the location of the variable)

uniformly by an RVM instruction of the form:

load *operand*

where *operand* is a constant value or a reference to a variable) results in a drastic reduction of instructions. As mentioned earlier, the size of our RVM instruction set is thus about one-fifth the size of the JVM instruction set.

4.3. Computation Size

We measured the physical size of the code generated in terms of the number of bytes. This aspect is important since network transfer traffic and speeds will be affected by the physical size of data being transmitted. Figure 12 shows the graph comparing the sizes of the code in bytes. MC refers to Mobile Computation and BC refers to JVM bytecode.

From the results it is apparent that the size of the JVM bytecode is a little smaller than that of the corresponding mobile computation, for most of the programs evaluated. For programs with function calls however the reverse is true. The reasons for these differences are possibly the following:

- The mobile computation always includes the header information (route and closure), while the JVM bytecode does not contain such information. This would add some overhead to the size of the mobile computation. However as the size of the program increases, the ratio of header to code will decrease in the mobile computation.

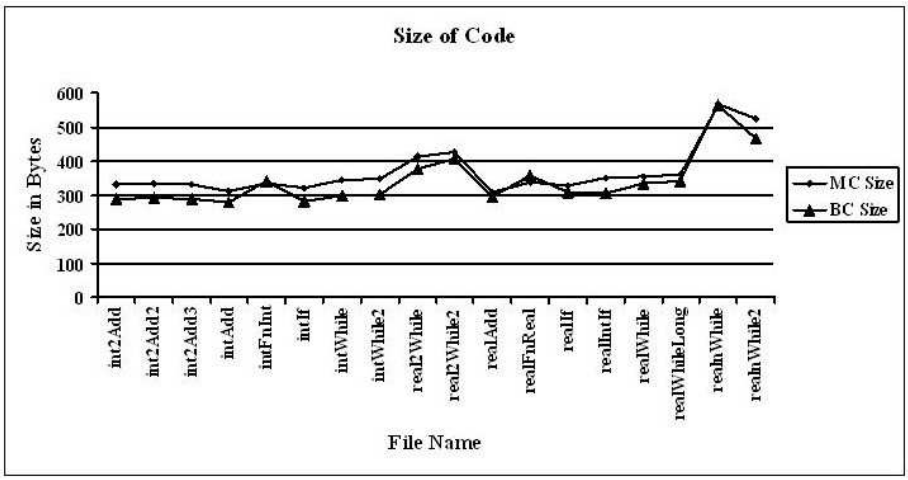

Figure 12: Size Comparison

- A task that requires one complex JVM instruction may require more than one simple RVM instruction. However, for applications that have several operations not directly supported by the JVM, the number of instructions required will be more than that required by the RVM.
- Function calls in Java require considerable more information regarding the stacks and parameters to be maintained, thus increasing the overheads for the bytecode.
- The compression technique used for the bytecode is more efficient than that used for the mobile computation, resulting in a smaller size of code.

Improvements to the compression technique used in our system will help to reduce the size of the mobile computation. Hence we do not consider the physical size of the code to be a negative factor for the mobile computation.

We also compared the number of instructions in the JVM bytecode against that for the mobile computation. We display our findings in Figure 13.

The data types we used for these programs were either 'integer' or 'real'. Since the JVM has specialized support for these data type for most of the basic operations, we expected the bytecode to have fewer instructions than the mobile computation for most of the programs. In actuality this difference

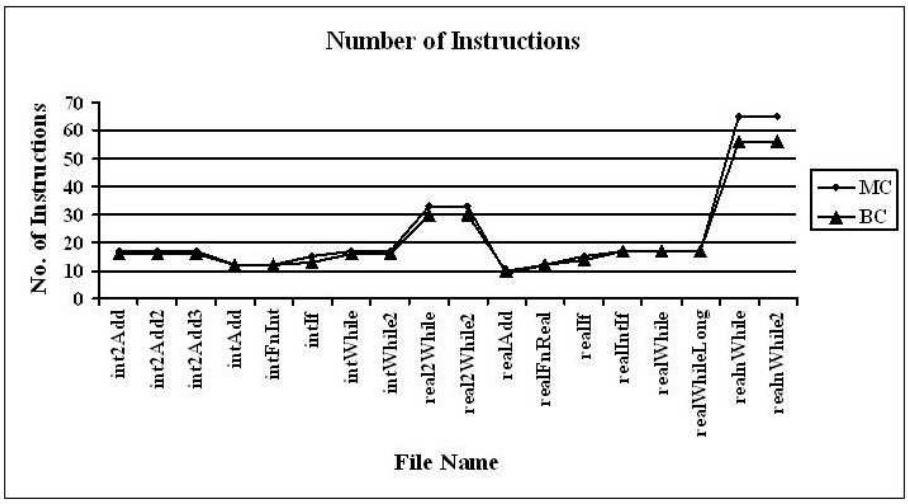

Figure 13: Comparing the number of instructions in JVM Bytecode and Mobile Computation

was marginal. For programs with operations on float values we found that both the bytecodes and the corresponding mobile computations had the same number of instructions.

4.4. Execution Time

An instruction with a fixed length can be loaded into the processor in one cycle. For an instruction with a variable length, on the other hand, the processor will first have to decode the instruction to find out whether there are any operands, and if so how many. The loading can therefore take several cycles depending on the length of the instruction and the number of operands. A JVM instruction has variable length, so the time taken to load it into the processor can vary. We denote this time by 'x' where $x > 1$. The time taken to decode and locate a JVM instruction in the instruction set is denoted by 'y'. The time taken for constant pool resolution is denoted by 'w' where $w \gg 1$. The time taken to decode and locate a RVM instruction in the instruction set is denoted by 'z', where $z \cong y \cong 1$, $x > y$, and $x > z$. Table 2 provides a theoretical comparison of the execution times for some JVM operations and the corresponding RVM operations. From the above examples we find that an operation on an integer - which is a 'privileged' data type - may be more efficient in the JVM than in the RVM. Operations on less 'privileged' data types degrade performance of the JVM. Memory

Table 2: Comparing Execution Times for the JVM and RVM

Operation	JVM OC	RVM OC	# JVM cycles	# RVM cycles	Compare JVM and RVM
load integer onto stack	iload	load data	3 + x + y $\cong$ 4 + x	4 + z $\cong$ 5	JVM $\cong$ RVM
load character data onto stack	iload	load data	3 + w + x + y $\cong$ 4 + w + x	4 + z $\cong$ 5	JVM > RVM
add integers	iadd	add int	4 + x + y $\cong$ 5 + x	5 + z $\cong$ 6	JVM $\cong$ RVM
add floats	fadd	add float	4 + x + y $\cong$ 5 + x	5 + z $\cong$ 6	JVM $\cong$RVM
integer comparison	if_icmplt	cmplt if_cmp	5 + x + y $\cong$ 6 + x	8 + 2z $\cong$ 10	JVM < RVM
double comparison	dcmpl iflt	cmplt if_cmp	7 + 2x + 2y $\cong$ 9 + 2x	8 + 2z $\cong$ 10	JVM $\cong$ RVM
char comparison	internal_op if_icmplt	cmplt if_cmp	9 + 2w + x + y $\cong$ 10 + 2w + x	8 + 2z $\cong$ 10	JVM > RVM

usage and performance in the RVM are consistent regardless of the data type being operated on.

We measured the actual time taken in seconds to execute the programs we evaluated in the previous sections. Figure 14 shows the comparison chart for our observations. We find a marked difference in performance of mobile computations and the corresponding bytecode. The mobile computations consistently executed faster than the bytecodes. This was in spite of the fact that neither the mobile computation nor the RVM have been optimized. In contrast, the JVM implementation has gone through several iterations and improvements. The bytecode does make use of the constant pool to speed up execution. Therefore, we would expect the JVM's performance to have been superior to that of the RVM in most cases. These results appear to validate our assertion that a RISC-based machine will perform better than a CISC-based machine.

4.5. Support for closure mobility

The JVM was not originally intended for mobile applications and does not support capture of the closure of a computation. A Java applet must be executed in its entirety at any host, although it is considered to be 'online

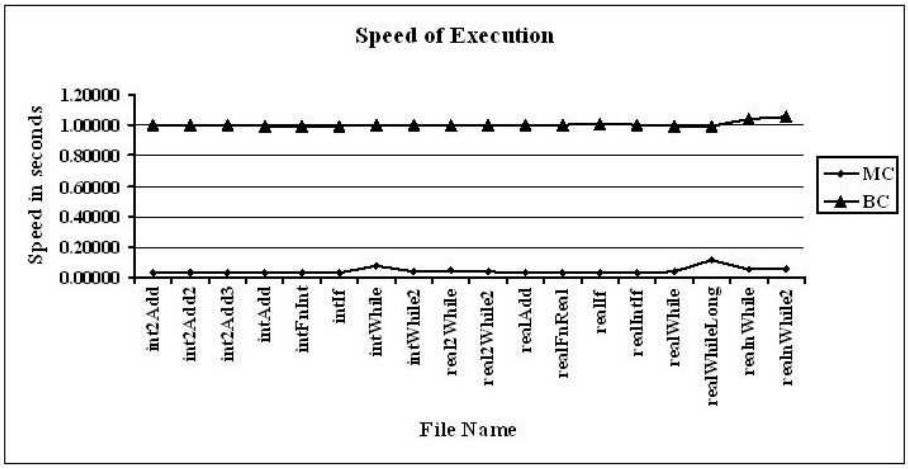

Figure 14: Comparison of execution speeds

portable' or 'mobile'. Therefore while the JVM supports 'weak' mobility, it does not support 'strong' mobility. On the other hand, the RVM has been designed specifically for mobile computations. Direct support for mobility and communication are provided in the RVM instruction set. Capturing the closure of a computation at any point is inherent in the system. Movement of the mobile computation is initiated by the 'move' construct.

5. Specification of the Abstract Mobile Computation Model

By providing a formal specification we can abstract away from the details of the implementation of the Mobile Computational Model and focus on its behavior. In this section we develop a formal specification. The notation, definitions, and a partial list of rules are summarized in the following tables. Table 3 describes the abstract syntax of the machine. Table 4 describes the various run-time structures, the major ones being the closure (configuration) Γ and associated components, such as stacks. Sample rules in tables 5 through 7 describe the various transition rules that affect the closure of a mobile computation.

5.1. Operations on the Execution Stack Σ

The executing environment Γ is composed of the program counter stack Π, the execution stack Σ, the list of objects that have been instantiated

Table 3: Syntactic Definition of the Abstract Machine

		Syntactic Definition	Description
ins	::=	i opc opr	*A RVM instruction consists of the index, the opcode and the operand*
opc	::=	aop | cop | bop | neg | get | put | invoke | jsr | ret | stop | move | endmove | goto | ifcmp | getparcel | makeparcel | load | store	*opcode encodes an operation that can be performed by the RVM*
opr	::=	τ | ρ | θ | ϕ | δ | ω | *const* | ι | μ | χ	*an operand is argument to operation.*
aop	::=	plus | minus | division | multiplication	*arithmetic operator*
cop	::=	less than | less than or equal to | greater than | greater than or equal to | equal to | not equal to	*comparison operator*
bop	::=	and | or	*boolean operator*

and accessed so far Ω, the permanent activation record Φ, the temporary activation record Δ and the location of the mobile computation ρ.

Rule 1: *remove top element σ from the Execution Stack*

$$\frac{\Gamma=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>,\ pop(\sigma.\Sigma)}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$$

σ represents the top item on the execution stack. On applying the pop operation on the execution stack, σ is discarded. The resulting environment Γ' differs from Γ in the content of the execution stack, i.e. $\sigma.\Sigma$ is changed to Σ.

Rule 2: *retrieve top element σ from the execution stack into x*

$$\frac{\Gamma=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>,\ top(\sigma.\Sigma)}{\Gamma=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>,\ x=\sigma}$$

σ represents the top item on the execution stack. On applying the top operation on the execution stack, the value stored at the top of the execution stack, σ, is accessed and copied into a temporary variable 'x'. The resulting environment is no different from the original environment since no changes have been made to any of the components.

Rule 3: *add element, σ, onto the top of the execution stack*

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ push(\sigma,\Sigma)}{\Gamma'=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>}$$

On applying the push operation on the execution stack, the new item σ is added to the top of the stack. The resulting environment Γ' differs from Γ in the content of the execution stack, i.e. Σ is changed to $\sigma.\Sigma$.

Table 4: Definition of Notation Used

Notation	Definition
$\Gamma =$ $< \Pi, \Sigma, \Omega, \Phi, \Delta, \rho >$	*current state of execution*
$\Pi =$ $\pi_1.\pi_2.\pi_3...\pi_i$	*current program counter stack*
π $< \omega, \chi, \mu, \delta, \iota >$	*item on program counter stack*
$\Sigma =$ $\sigma_1.\sigma_2.\sigma_3...\sigma_j$	*current execution stack*
σ'	*item on the execution stack*
$\Omega =$ $\omega_1, \omega_2, \omega_3, ..., \omega_k$	*list of instantiated objects*
ω'	*an instantiated object*
$\Phi =$ $\phi_1, \phi_2, \phi_3, ..., \phi_k$	*permanent activation record*
ϕ'	*item in the permanent activation record*
$\Delta =$ $\delta_1, \delta_2, \delta_3, ..., \delta_i$	*temporary activation record*
δ''	*item in the temporary activation record*
$I=$ $\iota_1.\iota_2.\iota_3.\iota_4....\iota_l$	*code for the method being executed*
ι	*one instruction in the code*
$P =$ $\rho_1, \rho_2, ..., \rho_m$	*computational subnet of mobile computation*
ρ	*current location of the mobile computation*
$\Theta =$ $\theta_1, \theta_2, \theta_3, ...\theta_n$	*list of currently active mobile computations*
θ	*one mobile computation*
$M_\chi =$ $\mu_1, \mu_2, ...\mu_p$	*list of methods for the class* χ
μ	*a method of the class.*
$B =$ $\beta_{\rho_1}, \beta_{\rho_2}, ...\beta_{\rho_m}$	*resources required by mobile computation*
β_{ρ_m}	*resources required at location* ρ_m
$X =$ $\chi_1, \chi_2, ...\chi_q$	*list of classes defined*
χ	*a class / abstract data type*
$A =$ $\{A_{\theta_1}, A_{\theta_2}, ..., A_{\theta_{n-1}}, A_{\theta_n}\}$	*parcels available in the system*
$A_{\theta_x} =$ $\{\alpha_{\theta_x\theta_1}, \alpha_{\theta_x\theta_2}, ..., \alpha_{\theta_x\theta_y}\}$	*parcels waiting for* θ_x
$\alpha_{\theta_x\theta_y}$	*parcel sent by* θ_y *waiting for* θ_x
$\tau \in$ $\{Int, R, C, B, \chi\}$	*data type*
η_{max}	*maximum number of hops allowed without a change in closure*
η_{curr}	*number of hops so far without a change in closure*

Table 5: Operations on Execution Stack Σ

Rule 1:	$\frac{\Gamma=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>,\ pop(\sigma.\Sigma)}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$	*remove σ from top of Execution Stack*
Rule 2:	$\frac{\Gamma=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>,\ top(\sigma.\Sigma)}{\Gamma=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>,\ x=\sigma}$	*retrieve top element from execution stack into x*
Rule 3:	$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ push(\sigma,\Sigma)}{\Gamma'=<\Pi,\sigma.\Sigma,\Omega,\Phi,\Delta,\rho>}$	*add element onto top of execution stack*

Table 6: Operations on Program Counter Stack Π

Rule 4:	$\frac{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ pop(\pi.\Pi)}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$	*remove top π from the program counter stack*
Rule 5:	$\frac{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ top(\pi.\Pi)}{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ x=\pi}$	*retrieve top π from the program counter stack into x*
Rule 6:	$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \pi=<\omega,\chi,\mu,\delta,\iota>,\ push(\pi,\Pi)}{\Gamma'=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$	*add π onto the top of the program counter stack*
Rule 7:	$\frac{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \pi=<\omega,\chi,\mu,\delta,\iota_i>,\ next(\iota)}{\Gamma'=<\pi'.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \pi'=<\omega,\chi,\mu,\delta,\iota_{i+1}>}$	*move to the next instruction to be executed*

Table 7: Mobility Operations (*move* ρ')

Rule 27:	$\frac{\Gamma_{\rho}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, \exists\, \beta_{\rho}, status(\beta_{\rho})=waiting, move\,(\rho'), \rho'\in P}{\Gamma\Rightarrow Halt}$ $\frac{\Gamma_{\rho_i}=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \pi=<\omega,\chi,\mu,\delta,\iota_j>, \forall\, \beta_{\rho_i}, status(\beta_{\rho_i})\neq waiting,\ next\,(\iota)}{\Gamma_{\rho_i}=<\pi'.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \pi'=<\omega,\chi,\mu,\delta,\iota_{j+1}>}$ $\frac{\Gamma_{\rho_{i-1}}=<\Pi_{i-1},\Sigma_{i-1},\Omega_{i-1},\Phi_{i-1},\Delta_{i-1},\rho_{i-1}>, \Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \Gamma_{\rho_{i-1}}\neq\Gamma\rho_i, move\,(\rho'), \rho'\in P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho'>}$ $\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1i-1}>, \Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \Gamma_{\rho_{i-1}}=\Gamma\rho_i, move\,(\rho'), \eta_{curr}<\eta_{max}, \rho'\in P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho'>}$ $\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1}>, \Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \Gamma_{\rho_{i-1}}=\Gamma\rho_i, move\,(\rho'), \eta_{curr}\geq\eta_{max}, \rho'\in P}{\Gamma'\Rightarrow Halt}$ $\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1}>, \Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \Gamma_{\rho_{i-1}}\neq\Gamma\rho_i, move\,(\rho'), \rho'\notin P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i+1}>}$ $\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1}>, \Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \Gamma_{\rho_{i-1}}=\Gamma\rho_i, move\,(\rho'), \eta_{curr}<\eta_{max}, \rho'\notin P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i+1}>}$

5.2. Operations on the Program Counter Stack Π

The executing environment Γ is composed of the program counter stack Π, the execution stack Σ, the list of objects that have been instantiated and accessed so far Ω, the permanent activation record Φ, the temporary activation record Δ and the location of the mobile computation ρ. π represents an item on the program counter stack. It keeps track of the instruction to be executed, the current method being executed, the object being operated on, the class the object is an instance of, the list of local variables, for the method, in the temporary activation record, and the list of attributes, of the object, in the permanent activation record.

Rule 4: *remove top element* π *from the program counter stack*

$$\frac{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ pop(\pi.\Pi)}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$$

On applying the pop operation on the program counter stack, π is discarded. The resulting environment Γ' differs from Γ in the content of the program counter stack, i.e. $\pi.\Pi$ is changed to Π.

Rule 5: *retrieve top element* π *from the program counter stack into* x

$$\frac{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ top(\pi.\Pi)}{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ x=\pi}$$

π represents the top item on the program counter stack. On applying the top operation on the program counter stack, π is accessed and copied into a temporary variable 'x'. The resulting environment is no different from the original environment since no changes have been made to any of the components.

Rule 6: *add element, π, onto the top of the program counter stack*

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \pi=<\omega,\chi,\mu,\delta,\iota>,\ push(\pi,\Pi)}{\Gamma'=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$$

π represents a new program counter item. This is usually created when a call to a method is encountered. On applying the push operation on the program counter stack, π is added to the top of the program counter stack. The resulting environment Γ' differs from Γ in the content of the program counter stack, i.e. Π is changed to $\pi.\Pi$.

Rule 7: *move to the next instruction to be executed*

This operation is performed at the end of execution of most operations.

$$\frac{\Gamma=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \pi=<\omega,\chi,\mu,\delta,\iota_i>,\ next(\iota)}{\Gamma'=<\pi'.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \pi'=<\omega,\chi,\mu,\delta,\iota_{i+1}>}$$

π represents the top item on the program counter stack, where the current instruction is denoted by ι_i. On applying the next() operation, the program counter is advanced to the next instruction, i.e. to ι_{i+1}. π' is merely π modified so that it now refers to the next instruction to be executed. This results in a change in the executing environment Γ.

5.3. Mobility Operations

Rule 27: *move ρ'*

This instruction initiates the process for the mobile computation to migrate to a different host.

$$\frac{\Gamma_\rho=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\exists\,\beta_\rho,\ status(\beta_\rho)=waiting,\ move\,(\rho'),\ \rho'\in P}{\Gamma\Rightarrow Halt}$$

If the mobile computation is still waiting to access any resource at the current location, then the move operation cannot be completed at this time. The computation is halted and the Mobile Computation Manager will queue the computation for the required resource. When the resource has been accessed and is no longer required, the move instruction is processed again.

$$\frac{\Gamma_{\rho_i}=<\pi.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>,\pi=<\omega,\chi,\mu,\delta,\iota_j>,\forall\,\beta_{\rho_i}, status(\beta_{\rho_i})\neq waiting,\ next\,(\iota)}{\Gamma_{\rho_i}=<\pi'.\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>,\pi'=<\omega,\chi,\mu,\delta,\iota_{j+1}>}$$

If all the resources required at the current location have been accessed, the program counter is incremented so that it now refers to the next instruction to be executed. At the new location execution will continue from instruction ι_{j+1}.

$$\frac{\Gamma_{\rho_{i-1}}=<\Pi_{i-1},\Sigma_{i-1},\Omega_{i-1},\Phi_{i-1},\Delta_{i-1},\rho_{i-1}>,\Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>,\Gamma_{\rho_{i-1}}\neq\Gamma\rho_i, move\,(\rho'),\rho'\in P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho'>}$$

If the location specified by the operand is on the list of locations the mobile computation is allowed to visit then the value of ρ is set to the new location. The Mobile Computation Manager then performs the tasks of closure capture and garbage collection. The current closure (at location ρ_i) is compared against the closure of the mobile computation when it arrived at the current site (this is the closure that was captured at the previous location ρ_{i-1}). If the two are different then the mobile computation is sent to the specified host.

$$\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1i-1}>,\Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>,\Gamma_{\rho_{i-1}}=\Gamma\rho_i, move\,(\rho'),\eta_{curr}<\eta_{max},\rho'\in P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho'>}$$

If the two closures are the same, then the MCM checks if the maximum number of hops the mobile computation is allowed without a change in closure is still greater than the number of hops the mobile computation has made so far without a change in closure. If η_{max} is greater than η_{curr} then the mobile computation is sent to the specified host.

$$\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1}>,\Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>,\Gamma_{\rho_{i-1}}=\Gamma\rho_i, move\,(\rho'),\eta_{curr}\geq\eta_{max},\rho'\in P}{\Gamma'\Rightarrow Halt}$$

If η_{max} is less than or equal to η_{curr} then the operation fails. An exception is generated and the virtual machine halts execution of the mobile computation. The Mobile Computation Manager is required to take the appropriate action.

$$\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1}>,\Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>,\Gamma_{\rho_{i-1}}\neq\Gamma\rho_i, move\,(\rho'),\rho'\notin P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i+1}>}$$

If the location specified by the operand is not on the list of locations the mobile computation is allowed to visit, then the value of ρ is set to the next location on the list where the required resources are available. The Mobile Computation Manager then performs the tasks of closure capture and garbage collection. The closure (at location ρ_i) is compared against the closure of the mobile computation when it first started executing at the current site (this is the closure that was captured at location ρ_{i-1}). If the two are different then the mobile computation is sent to the next host.

$$\frac{\Gamma_{\rho_{i-1}}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i-1}>, \Gamma_{\rho_i}=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_i>, \Gamma_{\rho_{i-1}}=\Gamma\rho_i, move\,(\rho'), \eta_{curr}<\eta_{max}, \rho'\notin P}{\Gamma'=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho_{i+1}>}$$

If the two closures are the same, then the MCM checks if the maximum number of hops the mobile computation is allowed without a change in closure is still greater than the number of hops the mobile computation has made so far without a change in closure. If η_{max} is greater than η_{curr} then the mobile computation is sent to the next host.

5.4. Communication Operations

Rule 28: *makeparcel* $\alpha_{\theta_i\theta_j}$ *for* θ_i

This instruction initiates the process for the mobile computation θ_j to package information into a parcel $\alpha_{\theta_i\theta_j}$ for θ_i to access.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi.\phi,\Delta.\delta,\rho>, parcel\,(\phi,\delta)}{\Gamma=<\Pi,\Sigma,\Omega,\Phi.\phi,\Delta.\delta,\rho>, \alpha_{\theta_i\theta_j}}$$

The data to be made available is packaged and addressed to mobile computation θ_j.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, \alpha_{\theta_i\theta_j}, A_{\theta_i}\in A}{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, A_{\theta_i}=A_{\theta_i}\bigcup\alpha_{\theta_i\theta_j}}$$

If there is already a set of parcels waiting for θ_i then $\alpha_{\theta_i\theta_j}$ is added to that set.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, \alpha_{\theta_i\theta_j}, A_{\theta_i}\notin A}{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, A_{\theta_i}=\alpha_{\theta_i\theta_j}}$$

If this is the first parcel for θ_i then a new set A_{θ_i} is created with $\alpha_{\theta_i\theta_j}$.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, A_{\theta_i}=\alpha_{\theta_i\theta_j}, A_{\theta_i}\notin A}{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>, A=A\bigcup A_{\theta_i}}$$

A_{θ_i} is then added to the set of all parcels in the system.

Rule 29: *getparcel* $\alpha_{\theta_i\theta_j}$ *from* θ_j

This instruction initiates the process for the mobile computation θ_i to obtain information (parcel) $\alpha_{\theta_i\theta_j}$ from θ_j.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j}\in A_{\theta_i},\ \theta=\theta_i}{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j},\ A_{\theta_i}=A_{\theta_i}-\alpha_{\theta_i\theta_j}}$$

If the requested parcel, $\alpha_{\theta_i\theta_j}$, is available, the identity the computation θ is verified. If the request is for a parcel addressed to it, the parcel, $\alpha_{\theta_i\theta_j}$, is given to θ and removed from the set of parcels, A_{θ_i}, waiting for the computation.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j}\in A_{\theta_i},\ \theta\neq\theta_i}{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j}\in A_{\theta_i}}$$

If the request is not for a parcel addressed to it, the parcel, $\alpha_{\theta_i\theta_j}$, is not given to θ. There is no change in the entities involved.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j},\ unparcel\,(\alpha_{\theta_i\theta_j})}{\Gamma=<\Pi,\Sigma,\Omega,\Phi.\phi,\Delta.\delta,\rho>}$$

If the parcel has been obtained successfully, it is opened and the data contained therein is updated in the temporary and permanent activation records as required.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j}\notin A_{\theta_i}}{\Gamma\Rightarrow Wait}$$

If the requested parcel, $\alpha_{\theta_i\theta_j}$, is not yet available in the set of parcels, A_{θ_i}, waiting for it, then θ_i may decide to wait until it is made available.

$$\frac{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>,\ \alpha_{\theta_i\theta_j}\notin A_{\theta_i}}{\Gamma=<\Pi,\Sigma,\Omega,\Phi,\Delta,\rho>}$$

Alternatively, the computation may decide to continue execution without the updated information.

```
header
route
juno.eecs.tulane.edu all (burn cd cd writer 50;
      read display 50;) 50;
endroute
id pegasus.eecs.tulane.edu; Monisha; endid
endheader

class intComp {

      public void main () {
            new int a;
            new int b;
            a=4+7;
      }

}
```

Figure 15: Program in MCL used to illustrate the Abstract Machine

6. Demonstration of the Abstract Machine

We use a simple example to demonstrate the working of the abstract machine just described. Figure 15 displays a program in MCL that adds two integers. Figure 16 shows the mobile computation generated from this program. When the computation is received by the MCM and all the security verification has been successfully completed, the closure is loaded. The program counter has one item. The only instantiated object is 'Me' which belongs to class intComp. We assume that the current location is 'juno.eecs.tulane.edu' (for brevity we refer to it as 'juno'). All the other stacks and lists are empty.

$\Gamma =< \Pi, \Sigma, \Omega, \Phi, \Delta, \rho >$where

$\Pi = \pi$

$\Sigma = \emptyset$, i.e., the execution stack is empty

$\Omega = Me$

$\Phi = \emptyset$, i.e., the permanent activation record is empty

$\Delta = \emptyset$, i.e., the temporary activation record is empty

$\rho = juno.eecs.tulane.edu$

```
header
route
juno.eecs.tulane.edu all (burn cd cd writer 50;
        read display 50;) 50;
endroute
id pegasus.eecs.tulane.edu Monisha endid
endheader

closure
        pc
            intComp Me    0    main   0    0   10
        endpc
        tempar
        endar
        permar
        endar
        object
        endobject
        stack
        endstack
endclosure

class   intComp 0
method  0       public  void    main    0
var     int     a
var     int     b
code
10      load    a
20      load    4
30      load    7
40      add     int
50      store   int
60      stop    main
endcode
endclass
```

Figure 16: Mobile Computation used to illustrate the Abstract Machine

$\pi =< \omega, \chi, \mu, \delta, \iota >$, where

$\omega = Me$

$\chi = intComp$

$\mu = main$

$\delta = empty$

$\iota = 10$

The program counter stack is checked using Rule 5, top ($\pi.\Pi$). Since there is an item on the stack, it is used to start execution. The method to be executed is the 'main' method of the class 'intComp'. Execution is to start with instruction at index 10. The local variables for the 'main' method are added to the temporary activation record Δ, using Rule 8 $add(\delta, \Delta)$, where $\delta = a, b$.

$$\frac{\Gamma=<\pi,\emptyset,Me,\emptyset,\emptyset,juno>,\ \pi=<Me,intComp,main,\emptyset,10\},\ add\,(\delta,\Delta)}{\Gamma'=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,10\}}$$

Now execution commences with the RVM executing the instruction 'load a', using Rule 23.

$$\frac{\Gamma=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ \sigma=a,\ \sigma\in\Delta,\ push\,(\sigma,\Sigma)}{\Gamma'=<\pi,a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,10\}}$$

The program counter is then incremented using Rule 7.

$$\frac{\Gamma=<\pi,a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,10\},\ next\,(\iota)}{\Gamma'=<\pi,a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,20\}}$$

The instruction at index 20 is now executed, i.e., 'load 4', using Rule 23.

$$\frac{\Gamma=<\pi,a,Me,\emptyset,a.b,juno>,\ \sigma=4,\ \sigma\in Const,\ push\,(\sigma,\Sigma)}{\Gamma'=<\pi,4.a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,20\}}$$

The program counter is then incremented using Rule 7.

$$\frac{\Gamma=<\pi,4.a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,20\},\ next\,(\iota)}{\Gamma'=<\pi,4.a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,30\}}$$

The instruction at index 30 is now executed, i.e., 'load 7', using Rule 23.

$$\frac{\Gamma=<\pi,4.a,Me,\emptyset,a.b,juno>,\ \sigma=7,\ \sigma\in Const,\ push\,(\sigma,\Sigma)}{\Gamma'=<\pi,7.4.a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,30\}}$$

The program counter is then incremented using Rule 7.

$$\frac{\Gamma=<\pi,7.4.a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,30\},\ next\,(\iota)}{\Gamma'=<\pi,7.4.a,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,40\}}$$

The instruction at index 40 is now executed, i.e., 'add int' using Rule 14. The top element in the execution stack, 7, is obtained using Rule 1 and stored in a temporary location x.

$$\frac{\Gamma{=}<\pi,7.4.a,Me,\emptyset,a.b,juno>,\ top(\Sigma)}{\Gamma{=}<\pi,7.4.a,Me,\emptyset,a.b,juno>,\ x{=}7}$$

The top element is then popped using Rule 2.

$$\frac{\Gamma{=}<\pi,7.4.a,Me,\emptyset,a.b,juno>,\ x{=}7,\ pop(\Sigma)}{\Gamma'{=}<\pi,4.a,Me,\emptyset,a.b,juno>,\ x{=}7}.$$

The new top element in the execution stack, 4, is obtained using Rule 1 and stored in a temporary location y.

$$\frac{\Gamma{=}<\pi,4.a,Me,\emptyset,a.b,juno>,\ top(\Sigma)}{\Gamma{=}<\pi,4.a,Me,\emptyset,a.b,juno>,\ y{=}4}$$

This top element is then popped using Rule 2.

$$\frac{\Gamma{=}<\pi,4.a,Me,\emptyset,a.b,juno>,\ y{=}4,\ pop(\Sigma)}{\Gamma'{=}<\pi,a,Me,\emptyset,a.b,juno>,\ y{=}4}$$

The arithmetic 'add' operation is now applied on x and y to obtain a value z. Since both x and y are of data type 'integer', z is also of data type 'integer'.

$$\frac{\Gamma{=}<\pi,a,Me,\emptyset,a.b,juno>,\ x,y{:}Int,\ x{=}7,\ y{=}4,\ z{=}x\ aop\ y}{\Gamma{=}<\pi,a,Me,\emptyset,a.b,juno>,\ z{=}11,\ z{:}Int}$$

The result z is now pushed onto the execution stack using Rule 3.

$$\frac{\Gamma{=}<\pi,a,Me,\emptyset,a.b,juno>,\ z{=}11,\ push(\sigma,\Sigma)}{\Gamma'{=}<\pi,11.a,Me,\emptyset,a.b,juno>,\ z{=}11}$$

The program counter is then incremented using Rule 7.

$$\frac{\Gamma{=}<\pi,a,Me,\emptyset,a.b,juno>,\ \pi{=}<Me,intComp,main,a.b,40\},\ next\,(\iota)}{\Gamma'{=}<\pi,a,Me,\emptyset,a.b,juno>,\ \pi{=}<Me,intComp,main,a.b,50\}}$$

The next instruction at index 50 is executed, i.e. 'store int', using Rule 24. The top element in the execution stack, 11, is obtained using Rule 1 and stored in a temporary location x.

$$\frac{\Gamma{=}<\pi,11.a,Me,\emptyset,a.b,juno>,\ top(\Sigma)}{\Gamma{=}<\pi,11.a,Me,\emptyset,a.b,juno>,\ x{=}11}$$

The top element is then popped using Rule 2.

$$\frac{\Gamma=<\pi,11.a,Me,\emptyset,a.b,juno>,\ x=11,\ pop(\Sigma)}{\Gamma'=<\pi,a,Me,\emptyset,a.b,juno>,\ x=11}.$$

The new top element in the execution stack, a, is obtained using Rule 1.

$$\frac{\Gamma=<\pi,a,Me,\emptyset,a.b,juno>,\ top(\Sigma)}{\Gamma=<\pi,a,Me,\emptyset,a.b,juno>,\ a}$$

This top element is then popped using Rule 2.

$$\frac{\Gamma=<\pi,a,Me,\emptyset,a.b,juno>,\ a,\ pop(\Sigma)}{\Gamma'=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ a}$$

The 'store' operation is now applied and 11 is stored in a. Since a is of data type 'integer' and the operand specifies 'integer', the operation succeeds.

$$\frac{\Gamma=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ a{:}Int,\ x=11,\ a=sto(x)}{\Gamma=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ a=11}$$

The program counter is then incremented using Rule 7.

$$\frac{\Gamma=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,50\},\ next\,(\iota)}{\Gamma'=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,60\}}$$

The instruction at index 60 is executed using Rule 22. The local variables for the method are removed from the temporary activation record, and then the top item is popped from the program activation stack.

$$\frac{\Gamma=<\pi,\emptyset,Me,\emptyset,a.b,juno>,\ \pi=<Me,intComp,main,a.b,60\},\ stop(\mu)}{\Gamma'=<\emptyset,\emptyset,Me,\emptyset,\emptyset,juno>}$$

The program counter stack is checked for the next method to be executed. Since there are no more items in the stack, the program has completed execution. Garbage collection is performed, removing all the instantiated objects and other references that remain.

$$\frac{\Gamma=<\emptyset,\emptyset,Me,\emptyset,\emptyset,juno>}{\Gamma'=<\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,juno>}$$

The computation is now terminated and can now be sent back to the originator if required or else just removed from the mobile computation database.

7. Conclusions

We described the development of a Mobile Computational Model to to support the efficient execution of mobile computations. Our goal was to be able to suspend execution of a mobile computation at any arbitrary point, migrate to another host and seamlessly continue execution from where it left off. This entails capture of the current state of execution at the origin and restoration of the closure at the destination. Our Mobile Computational Model provides the framework for managing such mobile computations. Through the RVM, the model provides an efficient homogeneous environment for computations to execute within, while supporting capture of state of execution. Our comparison of the JVM and RVM demonstrates the suitability of our approach. The abstract decription of the RVM provides a basis for formally analyzing the properties of the RVM.

References

[1] Nori, K., Ammann, U., Jensen, K., Nageli, H., and Jacobi, C., *Pascal: The language and its implementation*, ch. Pascal-P Implementation Notes, pp. 113 – 214. Wiley, 1981.

[2] Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T., "The problem of programming communication with changing machines," *Communications of ACM*, vol. 1, pp. 12–18, Aug. 1958.

[3] Cardelli, L., "Mobile computational ambients," tech. rep., Digital Equipment Corporation, 1997.

[4] Gosling, J. and McGilton, H., "The Java language environment," a white paper, Sun Microsystems, May 1996.

[5] Yourdon, E., *Rise and Resurrection of the American Programmer*, ch. Java and the new Internet programming paradigm. http://www.javaworld.com/javaworld/jw-08-1996/jw-08-yourdonbook.html: Prentice Hall, 1996. excerpt.

[6] Black, A., Hutchinson, N., Jul, E., and Levy, H., "Object structure in the Emerald system," in *OOPSLA*, pp. 78 – 86, ACM, Sept. 1986.

[7] Jul, E., Levy, H., Hutchinson, N., and Black, A., "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, pp. 109–133, Feb. 1988.

[8] Roy, P. v., Brand, P., Haridi, S., and Collet, R., "A lightweight reliable object migration protocol," in *Workshop on Internet Programming Languages, ICCL '98* (Bal, H. E., Belkhouche, B., and Cardelli, L., eds.), vol. 1686 of *Lecture Notes in Computer Science*, (Chicago, IL, USA), pp. 32 – 46, Springer, May 1998.

[9] Roy, P. v., Haridi, S., Brand, P., Smolka, G., Mehl, M., and Scheidhauer, R., "Mobile objects in Distributed Oz," *ACM Transactions on Programming Languages and Systems,*, vol. 19, pp. 804 – 851, Sept. 1997.

[10] Lucent Technologies, Inc., *Inferno*.

[11] Lucent Technologies, Inc, *Inferno: la Commedia Interattiva*, 1996.

[12] Cardelli, L., "Obliq: A language with distributed scope," Tech. Rep. 122, Digital Equipment Corporation Systems Research Center, June 1994. SRC Research Report.

[13] Cardelli, L., "A language with distributed scope," in *Computing Systems*, vol. 8 of *USENIX*, pp. 27 – 59, Jan. 1995. Prelim. ver. Conference Record of POPL '95, San Francisco, CA, January 1995.

[14] Cardelli, L. and Gordon, A. D., "Mobile ambients," in *FOSSACS'98, Intl. Conference on Foundations of Software Science and Computational Structures* (Nivat, M., ed.), vol. 1378 of *Lecture Notes in Computer Science*, pp. 140–155, Springer-Verlag, 1998.

[15] Cardelli, L., "Abstractions for mobile computation," in *Secure Internet Programming: Security Issues for Distributed and Mobile Objects* (Vitek, J. and Jensen, C., eds.), vol. 1603 of *Lecture Notes in Computer Science*, pp. 51 – 94, (also available as Microsoft Research Tech. Report MSR-TR-98-34): Springer, 1999.

[16] Adl-Tabatabai, A., Langdale, G., Lucco, S., and Wahbe, R., "Efficient and language-independent mobile programs," in *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, Philadelphia, PA, USA*, pp. 127 – 136, May 1996.

[17] Mascolo, C., Picco, G. P., and Roman, G.-C., "A fine-grained model for code mobility," in *European Software Engineering Conference*, (Toulouse, France), pp. 39 – 56, Sept. 1999. Held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC / FSE).

[18] Suri, N., Bradshaw, J., Breedy, M. R., Groth, P. T., Hill, G. A., and Jeffers, R., "Strong mobility and fine-grained resource control in NOMADS," in *2nd Intl. Symp. on Agent Systems and Applications and 4th Intl. Symp. on Mobile Agents* (Kotz, D. and Mettern, F., eds.), vol. 1882 of *Lecture Notes in Computer Science*, pp. 2 – 15, Springer-Verlag, Sept. 2000.

[19] Suri, N., Bradshaw, J., Breedy, M. R., Ford, K. M., Groth, P. T., Hill, G. A., and Saavedra, R., "State capture and resource control for Java: The design and implementation of the Aroma virtual machine." (personal communication with Dr. Suri).

[20] Johansen, D., Schneider, F. B., and van Renesse, R., "What TACOMA has taught us," in *Mobility, Mobile Agents and Process Migration - an edited collection* (Milojicic, D., Douglis, F., and Wheeler, R., eds.), Addison Wesley Publishing Company, 1998.

[21] Jacobsen, K. and Johansen, D., "Ubiquitous devices united: Enabling distributed computing through mobile code," in *Symposium on Applied Computing*, Addison Wesley Publishing Company, Feb. 1999.

[22] Picco, G. P., "muCode: A lightweight and flexible mobile code toolkit," in *2nd International Workshop on Mobile Agents* (Rothermal, K. and Hohl, F., eds.), vol. 1477 of *Lecture Notes in Computer Science*, (Stuttgart, Germany), pp. 160 – 171, Springer, Sept. 1998.

[23] Gray, R. S., "Agent Tcl: A transportable agent system," in *Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM '95)*, (Baltimore, MD), Dec. 1995.

[24] Gray, R., Cybenko, G., Kotz, D., and Rus, D., "Agent Tcl," in *Mobile Agents: Explanations and Examples* (Cockayne, W. and Zyda, M., eds.), ch. 4, ftp://ftp.cs.dartmouth.edu/pub/kotz/papers/gray:bookchap.ps.Z: Manning Publishing, 1996.

[25] Hashimoto, M. and Yonezawa, A., "MobileML: A programming language for mobile computation," in *Coordination Languages and Models* (Porto, A. and Roman, G.-C., eds.), vol. 1906 of *Lecture Notes in Computer Science*, pp. 198 – 215, Springer, 2000.

[26] Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M., and Peet, B., "Concordia: An infrastructure for collaborating mobile agents," in *First International Workshop on Mobile Agents* (Rothermal, K. and Popescu-Zeletin, R., eds.), vol. 1219 of *Lecture Notes in Computer Science*, (Berlin, Germany), pp. 86 – 97, Apr. 1997. http://www.meitca.com/HSL/Projects/Concordia/documents.htm.

[27] Conde, J., "Mobile agents in Java." http://wwwinfo.cern.ch/asd/rd45/white-papers/9812/agents2.html, Dec. 1998. CERN/IT/ASD/RD45/98/12.

[28] "Mobile agent computing." White Paper, Horizon Systems Laboratory, Mitsubishi Electric ITA, Jan. 1998. http://www.meitca.com/HSL/Projects/ Concordia/documents.htm.

[29] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*. The Java Series, Addison Wesley, 2 ed., 1999.