Report Number: WUCS-97-02

1997-01-01

Expressing Code Mobility in Mobile UNITY

Gian Pietro Picco, Gruia-Catalin Roman, and Peter J. McCann

Advancements in network technology have led to the emergence of new computing paradigms that challenge established programming practices by employing weak forms of consistency and dynamic forms of binding. Code mobility, for instance, allows for invocation-time binding between a code fragment and the location where it executes. Similarly, mobile computing allows hosts (and the software they execute) to alter their physical location. Despite apparent similarities, the two paradigms are distinct in their treatment of location and movement. This paper seeks to uncover a common foundation for the two paradigms by exploring the manner in which stereotypical forms of code... **Read complete abstract on page 2.**

Expressing Code Mobility in Mobile UNITY

Gian Pietro Picco, Gruia-Catalin Roman, and Peter J. McCann

Complete Abstract:

Advancements in network technology have led to the emergence of new computing paradigms that challenge established programming practices by employing weak forms of consistency and dynamic forms of binding. Code mobility, for instance, allows for invocation-time binding between a code fragment and the location where it executes. Similarly, mobile computing allows hosts (and the software they execute) to alter their physical location. Despite apparent similarities, the two paradigms are distinct in their treatment of location and movement. This paper seeks to uncover a common foundation for the two paradigms by exploring the manner in which stereotypical forms of code mobility can be expressed in a programming notation developed for mobile computing. Several solutions to a distributed simulation problem are used to illustrate the modeling strategy for programs that employ code mobility.

Expressing Code Mobility in Mobile UNITY

Gian Pietro Picco
Gruia-Catalin Roman
Peter J. McCann

WUCS-97-02

January 21, 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Expressing Code Mobility in Mobile UNITY

Gian Pietro Picco[*], Gruia-Catalin Roman[†] and Peter J. McCann

January 21, 1997

Abstract

Advancements in network technology have led to the emergence of new computing paradigms that challenge established programming practices by employing weak forms of consistency and dynamic forms of binding. Code mobility, for instance, allows for invocation-time binding between a code fragment and the location where it executes. Similarly, mobile computing allows hosts (and the software they execute) to alter their physical location. Despite apparent similarities, the two paradigms are distinct in their treatment of location and movement. This paper seeks to uncover a common foundation for the two paradigms by exploring the manner in which stereotypical forms of code mobility can be expressed in a programming notation developed for mobile computing. Several solutions to a distributed simulation problem are used to illustrate the modeling strategy for programs that employ code mobility.

Keywords: Code mobility, UNITY, Mobile UNITY, coordination, mobile computing, mobile code languages.

1 Introduction

Code mobility is defined informally as the capability, in a distributed application, to dynamically reconfigure the binding between code fragments and the location where they are executed [22]. This simple definition, however, raises a lot of questions. What is the unit of mobility? What are the effects of the move on the computations executing at the point of origin and at destination? What aspects of the run-time state move along with the code? What are the boundaries of a location? These questions have received partial answers in a new generation of programming languages called *mobile code languages (MCLs)* [22]. These languages provide specialized abstractions and run-time support capabilities designed to support various forms of code mobility. The impetus for the development of MCLs stems from the need to overcome some of the technological problems facing the Internet today. The conventional paradigms used for the development of Internet applications, like client-server, do not seem to scale up. New technologies and design paradigms are being considered. Some of them assume that hosts and communication links are part of a highly dynamic global computing platform, where the application code can move freely among the computing nodes. This *network centric* style of computing is at the center of the emerging mobile code languages.

For a comprehensive survey of MCLs, the reader is directed to [5] which reviews a number of existing languages and attempts to extract their essential features. The unit of mobility, that in [5] is called *executing unit*, is implemented differently in different languages, but can be thought of as a process in an operating system or a thread in a multithreaded environment. Mobile code may assume two basic forms, strong and weak. *Strong mobility* allows executing units to move their code and their *execution state* to a different site. The execution state contains control information related to the state of the executing unit, e.g. the instruction pointer. Upon migration, executing units are suspended, transmitted to the destination site, and resumed there. *Weak mobility* allows an executing unit at a site to be bound dynamically to code coming from a different site. This encompasses two cases. Either the executing unit dynamically links code downloaded from the network, or the executing unit receives its code from another executing unit. In the latter case, two distinct cases may be considered. Either the executing unit at the destination site is created from scratch to run the incoming code, or a pre-existing executing unit links the incoming code dynamically and executes it.

Current MCLs provide different mixtures of the above notions of code mobility. For instance, Telescript [12, 23] provides full-fledged support for strong mobility. In Telescript, a special thread called *agent* can migrate to a different site by executing a special go operation, whose effect is to suspend execution of the thread, to pack it in a format suitable for transmission, and to send it to the destination site, where it is unpacked and can resume execution starting from the instruction immediately following the go in the source code of the agent. Agent Tcl [7, 8],

[*]Gian Pietro Picco is with Politecnico di Torino, Dipartimento di Automatica e Informatica, C.so Duca degli Abruzzi 24, 10129 Torino, Italy.

[†]Gruia-Catalin Roman and Peter J. McCann are with Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA.

an extension of Tcl [16], provides support for strong mobility as well, but upon execution of a migration instruction jump the whole UNIX process containing the interpreter is migrated, instead of a single thread within it. On the other hand, in Java [19] the class loader can be programmed to enable a Java program to link dynamically code downloaded from the network, hence providing support for weak mobility. TACOMA [10], Facile [20, 11], and M0 [21] are examples of languages that support weak mobility by allowing a procedure or function to be sent to another node for remote execution, with the portion of the global environment that is needed to proceed with execution—but with no execution state.

By and large, these developments fall outside the traditional concerns of distributed computing since much of the existing work on models, algorithms, proof systems, methodologies, and impossibility has been carried out assuming networks with a fixed topology and static binding between the application code and the hosts where it is being executed. In contrast, MCLs enable more dynamic solutions to distributed computing problems, such as design paradigms that encompass new forms of interaction among the components of a distributed application. Relating these new paradigms to previous research on distributed computing is the main theme of this paper.

The model we use in our study is called *Mobile UNITY* [18, 13], an extension of work by Chandy and Misra on UNITY [4]. Mobile UNITY provides a programming notation that captures the notion of mobility and transient interactions among mobile nodes and includes an assertional-style proof logic. The model adheres to the minimalist philosophy of the original UNITY, supports text-based reasoning about programs, and focuses only on essential abstractions needed to cope with the presence of mobility. As we use this model to examine mobile code paradigms, the fundamental goal is to determine whether Mobile UNITY by itself is adequate to this modeling task.

The remainder of this paper is structured as follows. Section 2 describes a simplified version of a distributed simulation problem and introduces standard UNITY via a centralized solution. Section 3 presents a distributed solution to the problem using the client-server paradigm. In doing so, it also provides the reader with a gentle introduction to Mobile UNITY. Section 4 introduces basic concepts relevant to code mobility, provides some background information, and presents mobile code solutions to the distributed simulation problem that illustrate our strategies for modeling mobile code in Mobile UNITY. Mobile code solutions are modeled after the taxonomy of mobile code design paradigms found in [2]. Finally, Section 5 explores some of the issues raised by this investigation and discusses the kinds of features that may be needed in a richer model of mobility.

2 A Distributed Simulation Problem

In this section we present an example that will be used for illustration purposes throughout the remainder of the paper and use it as a vehicle to introduce the reader to the UNITY notation. The example is inspired by the work of Chandy and Misra who provided a formal characterization and solution for a distributed simulation problem [3]. The basic idea is to simulate the behavior of a physical system such as an electronic circuit on a network of computing nodes which communicate asynchronously and in the absence of global shared memory. Physical entities are allocated to nodes across the network and simulated according to their expected behavior. The nodes must communicate among themselves in order to simulate the interactions normally occuring among the physical components (e.g., passing a signal from one gate to the next) and also in order to preserve the correct temporal relationships among the actions associated with the various simulated entities. It is the latter aspect of the problem which is central to its solution. For this reason, in our simplified version we focus strictly on the temporal aspects of the problem and ignore any other interactions among the components. In other words, we assume that each simulated entity executes at most one action at a time in a deterministic manner and does not interact with any other entities being simulated at other nodes. However, because the simulation may be monitored by some external agent while in progress, the ordering of actions in time must be consistent with those occuring in the simulated system. These simplifying assumptions would be realistic, for instance, when particle movement is simulated in the absence of collisions.

The behavior of each node is very simple. A *local timer* holds the time value at which the next local action is to be executed. The action can be executed only when all nodes participating in the simulation reach that particular time, i.e., all actions scheduled for earlier times have been executed. The notion of *global virtual time (GVT)*, whose value is defined as the minimum among the values of all local timers, formalizes the intuitive idea that the simulation reached a particular point in time. In a centralized solution to the problem, such as the UNITY solution appearing later in this section, the value of the GVT can be stored in a variable and can be updated by examining the value of each local timer. In a distributed solution, each node has to discover the GVT value by communicating with other participating nodes. Once GVT catches up with the local timer, the corresponding action is executed and the local timer is incremented to reflect the time when the subsequent action is scheduled to take place.

In Figure 1 we show three processes P indexed by i which participate in a distributed simulation. Solid lines depict values of the corresponding local timers. The current GVT is shown below the horizontal axis and marked by the symbol T. The only process allowed to execute an action is $P(2)$. After the execution of its action, $P(2)$ is

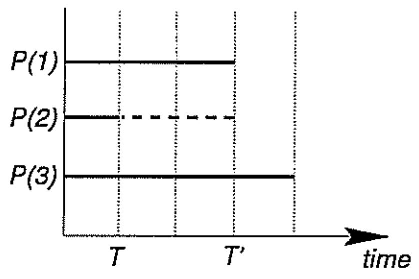

Figure 1: A snapshot of the distributed simulation.

allowed to update the value of its local timer to a new value referring to some time in the future. As a result, in this example, a new GVT value T' is established and $P(1)$ and $P(2)$ are now allowed to execute their respective actions, independently of each other. In the remainder of this section, we will define the problem formally, and show a centralized solution for it.

2.1 Problem Formalization

A system is modeled by a set of N processes, indexed from 0 to $(N-1)$. Each process $P(i)$ has an associated local timer, $t(i)$. The GVT T is defined as the minimum of all the local timers, i.e. $T = \langle min\ i : 0 \leq i \leq N-1 :: t(i)\rangle$[1]. The scheduling criterion used by a process to update its timer is embodied in the definitions for functions f and g, both indexed by a specific process identifier:

$$f_i(t(i), T, z) = \begin{cases} t(i) & \text{if } t(i) > T, \\ g_i(t(i), z) & \text{if } t(i) = T, \\ \perp & \text{if } t(i) < T. \end{cases} \tag{1}$$

$$g_i(t(i), z) > t(i) \tag{2}$$

where $t(i)$ denotes the value for a local timer, T denotes the GVT value, and z identifies the simulation mode, e.g., a parameter of the physical components. These definitions capture the following requirements:

1. The local timer cannot change if it is ahead the GVT.
2. The local timer, if permitted, can only increase, i.e., actions are always scheduled in the future (see the constraint on g in Equation 2).
3. The value of a local timer can never be behind the GVT. For such cases the function f is undefined.

In the next section, we introduce the standard UNITY notation by describing a solution for this problem.

2.2 A Centralized Solution in UNITY

In this section, we discuss a solution for the distributed simulation problem that uses the UNITY notation described in [4]. In the UNITY program shown in Figure 2, the **declare** section contains variable declarations. The arrays t and x contain, for each process i, the value of the local timer and a process variable relevant for the simulation. T stores the current value for the GVT and z represents the simulation mode. The **initially** section contains a set of predicates, separated by the symbol $[]$, which define the allowed set of initial states. Uninitialized variables will assume an arbitrary value belonging to their declared type. All local timers $t(i)$ are initialized to zero and T is initialized consistently. The simulation mode z is initialized to some default initial value. The **assign** section is the core of the program. It consists of a set of assignment statements that specify the program behavior. Program execution starts in the state described in the **initially** section and evolves as a non-deterministic, fair interleaving of statements—in an infinite execution of the program each statement is executed infinitely often. The first statement computes the current GVT as the minimum among the values of the local timers, and stores it in T. The second statement is a set of asynchronous assignments each updating the local timer for a corresponding process $P(i)$. Due to the definition of f, the update is performed only when the timer value is equal to the value of T, otherwise it has

[1]The three-part notation $\langle$op *quantified_variables* : *range* :: *expression*$\rangle$ is borrowed from UNITY and will be used throughout the paper. It is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values in which op is applied. In the case above, it is equivalent to $\{P(i) | 0 \leq i \leq N-1\}$.

```
Program DistributedSimulation
    declare
        $x, t$ : array of integer [] $T, z$ : integer
    initially
        $\langle [] \; i :: x(i), t(i) = 0, 0 \rangle \; [] \; T, z = 0, \bar{z}$
    assign
        $T := \langle \min \; i :: t(i) \rangle$
      [] $\langle [] \; i :: t(i) := f_i(t(i), T, z) \rangle$
      [] $\langle [] \; i :: x(i) := t(i) \rangle$
      [] $z := d(T)$
end
```

Figure 2: Standard UNITY solution for the distributed simulation problem.

no effect on T. The statement is defined using the three-part notation with [] as a quantifier, hence it is equivalent to N assignments separated by [] and executed non-deterministically and independently. If we wanted to specify synchronous execution instead, we could have used || in place of []. Note also that, because of fair execution, each of the assignments separated by [] is non-deterministically interleaved with the one computing the GVT. In some fair execution, it could happen for all the $t(i)$ to be updated before the new GVT is computed. In this case, T actually represents a lower bound for the GVT value, and the T parameter used as an argument for function f is actually an approximation of the GVT. Because in our solution the details about the behavior of processes are irrelevant we abstract them into a set of assignments to the array $x(i)$. Finally, the value of the simulation mode is updated dynamically according to the definition of a function $d(T)$ whose details are left out.

Program properties may be stated formally and may be verified by using the UNITY proof logic. A specialization of temporal logic, the UNITY proof logic allows one to verify the correctness of a program directly from its text in the tradition of sequential programming, but this is outside the scope of this paper.

Although the solution presented in Figure 2 is formally correct, it is not acceptable from a design point of view because every variable appears to be *shared* and, in particular, the GVT is shared among all processes—which has been explicitly forbidden by the statement of the problem. In addition, it is not apparent that the local timers are associated with processes, and the program text does not even capture explicitly the notion of independent processes. In the following section we show a client-server solution that does not employ shared variables and make explicit the location and encapsulation embodied in each process. In doing this, we will introduce Mobile UNITY, which will be used in Section 4 to describe the distributed solutions exploiting code mobility.

3 A Client-Server Solution in Mobile UNITY

In a later section of the paper we will give three alternative solutions to the distributed simulation problem, each modeled after a single mobile code design paradigm. Each will be expressed in Mobile UNITY [13], a modification of the standard UNITY notation presented in Section 2.2. Mobile UNITY offers novel abstractions designed to deal specifically with distributed systems that contain mobile components, as well as an associated proof logic for reasoning about the behavior of such systems. In this section we make use of Mobile UNITY to present our first non-centralized solution to the distributed simulation problem—one that embodies the client-server paradigm.

In the *Client-Server (CS)* paradigm, a server component exports a set of services. The client component, on the other hand, at some point in its execution lacks some of the resources needed to proceed with its computation. The resources are located on the server host and they are accessed by the client by interacting with the server through message passing. The interaction specifies what kind of service needs to be invoked on the server in order to access the resources. Consequently, in the CS paradigm the resources are co-located with the know-how needed to access them, and no relocation of components takes place.

Although the solution we present does not contain mobile components, it will serve to introduce the reader to the new constructs available in Mobile UNITY, such as the system structuring conventions and simple interaction clauses. After the presentation of the example we will discuss other features of Mobile UNITY not exhibited by the text of the client-server solution, such as transiently shared variables.

3.1 System Definition

As in our earlier centralized solution, we will ignore the internal simulation steps except for their effect on the advancement of the local timers. Each process must calculate an estimate T of the GVT to determine whether the next step in its local simulation is allowed. For correctness, this estimate should never exceed the real GVT, otherwise a process might take a step even when its timer value is in the future with respect to some other component. The distributed solutions presented in this and the following sections must compute a lower bound on the GVT without using statements like $T := \langle \min\ i :: t(i) \rangle$ which imply centralized access to the local timers of each component. The client-server solution of Figure 3 provides for such distribution by breaking up the system into a single server and a set of clients. The server contains an array τ which attempts to maintain the global state of all the local timer values from each of the clients. This array is updated via asynchronous message passing, and therefore may sometimes contain old values of the local timers. A new estimate for the GVT is calculated at the server and returned to waiting clients again via asynchronous message passing.

```
System DSClientServer
    Program P(i) at $\lambda$
        declare
            $x, t, z$ : integer [] $T$ : integer $\cup \{\bot\}$ [] $RQ$ : request $\cup \{\bot\}$
        initially
            $x, t, z = 0, 0, \bar{z}$ [] $T = \bot$ [] $\lambda = \text{Location}(i)$ [] $RQ = \bot$
        assign
            $x := t$
          [] $t, T := f_i(t, T, z), \bot$                    if $\text{def}(T)$
          [] $RQ := \langle \text{SERVER}, \text{CS}, \text{MINSERV}, t \rangle$      if $\neg\text{def}(RQ) \wedge \neg\text{def}(T)$
          [] $T, RQ := RQ \uparrow 4, \bot$                   if $RQ \uparrow 1 = i$
    end
    Program Server at $\lambda$
        declare
            $T$ : integer $\cup \{\bot\}$ [] $\tau$ : array of integer [] $q$ : array of (request $\cup \{\bot\}$)
        initially
            $T = \bot$ [] $\langle \| \ j :: \tau(j) = 0 \rangle$ [] $\langle \| \ j :: q(j) = \bot \rangle$ [] $\lambda = \text{Location}(\text{SERVER})$
        assign
            $\langle [] \ j :: \tau(j), q(j) \uparrow 2 := q(j) \uparrow 4, \text{WAIT}$       if $q(j) \uparrow 1 = \text{SERVER} \wedge q(j) \uparrow 2 \neq \text{WAIT}$
          [] $T := \langle \min\ k :: \tau(k) \rangle$                 if $\langle \exists\ j :: q(j) \uparrow 2 = \text{WAIT} \rangle \wedge \neg\text{def}(T)$
          [] $\langle \| \ j :: q(j), T := \langle j, \bot, \bot, T \rangle, \bot$          if $\text{def}(T) \wedge q(j) \uparrow 2 = \text{WAIT} \rangle$
    end
    Components
            $\langle [] \ i :: P(i) \rangle$ [] Server
    Interactions
            $Server.q(i) := P(i).RQ$                 when $\neg\text{def}(Server.q(i)) \wedge \text{serviceRequest}(\text{CS}, \text{MINSERV}, i)$
          [] $P(i).RQ, Server.q(i) := Server.q(i), \bot$   when $\text{serviceReady}(i)$
end
```

Figure 3: Client-Server solution for the distributed simulation problem.

Figure 3 is illustrative of the new structuring conventions provided by Mobile UNITY. The first line provides the system name, *DSClientServer*. Next are a set of program definitions, the first of which, *P(i)*, will serve as the code executed by each client and is parameterized by a single index representing the client number. The second, *Server*, has no such parameter. The program definitions are treated as types that are instantiated in the **Components** section. The program instances listed there are considered to be the running components of a distributed computation. In this example, the **Components** section instantiates one client for every value of i in the appropriate range, and a singleton *Server* instance. Each component has a distinct name: the clients, because they are indexed, and the server, because it is instantiated only once.

Note that each program definition begins with a line like **Program** *name* **at** λ. This denotes that each program exists at a specific physical or logical location denoted by the distinguished program variable λ. Each program contains a predicate in its **initially** section that gives the initial position of the component. We assume the existence of a function Location() that returns the initial position of each component based on an address which is either a client number or the constant SERVER. Each program may also contain code in its **assign** section that reads or modifies the position of the component. An assignment to λ models actual migration of the component through some physical or logical address space. Throughout this paper, we will leave the type of λ unspecified. For many problems, a simple discrete domain that reflects the connectivity among components will suffice, for example, a single bit that denotes whether a mobile host is within broadcast range of a fixed router. For other problems, the location type may be more complex and may exhibit much more structure. The type may be determined by the characteristics of a particular problem domain or may be implied by the way λ is used, e.g., how and whether it is incremented or checked for equality, but it is not necessary to completely specify this type in order to carry out useful reasoning about a system of mobile components. We will make much more use of λ in later examples that, in contrast to *DSClientServer*, actually do contain mobile components, and where the connectivity among components depends on their dynamically changing locations.

In standard UNITY, there is no notion of changing connectivity among the programs making up a composed system, and two variables with the same name in different programs are considered shared throughout execution of the system. All variables of a Mobile UNITY component are considered local to that component. When dealing with a collection of instantiated components, a specific instance variable is referenced by prepending the name of the variable with the name of the program in which it appears. For example, the variables x, t, and z of the program *P(1)* have the fully qualified names $P(1).x$, $P(1).t$, and $P(1).z$. These fully qualified names should be used when carrying out formal reasoning about the behavior of the system, although we will sometimes leave off the program name when the context is clear.

Because variables are local, no communication can take place among components without the presence of interaction clauses spanning the scope of components. These appear in the **Interactions** section, and serve to provide implicit communication and coordination among the components. The two interaction clauses given in Figure 3 allow for communication between each client and the server. We assume that the index i is instanced over the appropriate range. Note that these clauses look like ordinary UNITY assignment statements, except that they use the keyword **when** in place of the keyword **if**. Also, because they are not internal to some component, they may reference variables of any component, using the naming conventions given above. For example, the first interaction clause provides asynchronous message transfer from the client message buffer $P(i).RQ$ to the server message buffer $Server.q(i)$, when the server buffer is empty and there is a valid request message waiting in the client buffer:

$$Server.q(i) := P(i).RQ \qquad \textbf{when } \neg\mathsf{def}(Server.q(i)) \wedge \mathsf{serviceRequest}(\textsc{cs}, \textsc{minServ}, i)$$

The second interaction transfers a reply back to the client and empties the server buffer, when the reply is ready:

$$P(i).RQ, Server.q(i) := Server.q(i), \bot \qquad \textbf{when } \mathsf{serviceReady}(i)$$

Semantically, these two statements are treated like ordinary assignment statements, and we assume that execution consists of a fair interleaving of all assignment statements of each component as well as these "extra statements" in the **Interactions** section.

The meaning of the macros used in the guards of the interactions is shown in Figure 4. These macros will be used throughout the remaining examples. A request is a tuple consisting of four elements. We denote the tuple by

$\mathsf{clientAddress} = \langle Set\ n : 0 \le n \le N-1 :: n \rangle$
$\mathsf{address} = \{\{\textsc{server}\} \cup \mathsf{clientAddress}\}$
$\mathsf{opName} = \{\textsc{cs}, \textsc{rev}, \textsc{cod}\}$
$\mathsf{opStatus} = \{\textsc{wait}\}$
$\mathsf{request} = \langle \mathsf{address},\ \mathsf{opName} \cup \mathsf{opStatus} \cup \{\bot\},\ \mathsf{serviceName} \cup \{\bot\},\ \mathsf{integer} \rangle$
$\mathsf{serviceRequest}(x : \mathsf{opName}, y : \mathsf{serviceName}, i : \mathsf{clientAddress}) \equiv P(i).RQ \uparrow 1 = \textsc{server} \wedge P(i).RQ \uparrow 2 = x \wedge P(i).RQ \uparrow 3 = y$
$\mathsf{serviceReady}(i : \mathsf{clientAddress}) \equiv Server.q(i) \uparrow 1 = i$

Figure 4: Macro definitions used in the example systems. Allowed values for each data element are represented as sets.

enclosing it in angle brackets and separating its four elements by commas. We use the field index operator $\uparrow$ to access individual fields of a record, e.g., $P(i).RQ \uparrow 1$ represents the address field of the request variable $P(i).RQ$. This field is used to denote the destination of the message, and must be either a client index or the constant SERVER. The second field is used to denote the paradigm used to deliver the service, and must be an opName, which is one of CS for client-server, REV for remote evaluation, or COD for code on demand. This field may also represents the status of the request with an opStatus, if the server is in the process of constructing a reply. The third field denotes the specific service requested, which in the case of the client-server solution is specified by the client to be MINSERV, which indicates that the client wants an estimate of the minimum local time held by any client. This field is always MINSERV in the client-server example, but we provide it because a server, in general, may provide multiple services. This field will take on a different value for the code on demand example presented later. The fourth and final field of a request must be an integer data item, which the client uses to transmit its current value of t to the server. With this in mind, the reader can see that the predicate serviceRequest checks to see that the message buffer of client i contains a message destined for the SERVER, with the opName and serviceName given. The predicate serviceReady checks the $Server.q(i)$ buffer for a message destined for client i.

Now we examine the inner workings of each component, as given by the **assign** section of each program definition. The assignment statements in the client program *P(i)* look very much like the ones in the centralized solution *DistributedSimulation* presented in Section 2.2, except that now the variables x and t occur once in each component instead of appearing as global arrays of values: we now use $P(i).x$ in place of $x(i)$. Also, T appears once in each program instead of being globally declared. Note that the simulation mode $P(i).x$, in contrast to the centralized solution, is initialized statically for each component and does not change during simulation execution. This is done for the sake of simplicity and clarity. In the code on demand solution presented later, we will compute this value dynamically. Otherwise, the statements to update the local simulation variable $P(i).x$ and the local simulation time $P(i).t$ are the same as before, except that the estimated GVT variable $P(i).T$ is simultaneously set to $\perp$ when an update to the local time is made. Throughout the example, the notation def(v) denotes the predicate $v \neq \perp$, i.e., the variable v is defined. This is used in the guard of the update to the local timer, and signals the fact that the client needs a new value of T before it can proceed with another simulation step. The new estimate of GVT is computed in the server at the request of the client, and the request is made by the third statement in the program *P(i)*, which writes a request record to the message buffer $P(i).RQ$.

Once the assignment to $P(i).RQ$ has taken place, the first interaction clause is enabled. Its guard makes use of the macro serviceRequest, which is a predicate that detects the presence of a valid request. After the interaction is enabled it is eventually executed, which asynchronously transmits the request to the server.

The *Server* program consists of three groups of assignments. The first processes input requests by updating the array $Server.\tau$ with the local time sent by the client. The second computes a new estimate of the GVT based on the current values in $Server.\tau$, if some client request is waiting in a buffer. The third takes the estimate and constructs a reply message to all waiting clients, clearing the estimate. Once the reply has been written to $Server.q(i)$, the second interaction clause is enabled. Its guard makes use of the macro serviceReady, which is a predicate that detects the presence of a valid reply. After the interaction is enabled it is eventually executed, which asynchronously transfers the reply back to the client. The fourth assignment statement of the program *P(i)* processes the reply by updating the local GVT estimate $P(i).T$ and clearing the request buffer.

3.2 Additional Interaction Constructs

The client-server example does not exercise all of the features of Mobile UNITY, and we now discuss some of those unused features in preparation for their use in later examples. For instance, none of the components in the *DSClientServer* system are mobile. Also, the only interactions present are two extra assignment statements. In general, a Mobile UNITY system can contain more powerful interaction clauses, such as the specification of ***transient shared variables.*** These are variables of one component that are dynamically bound to variables of another component in a location-dependent manner. Any assignments to one of the variables are considered to propagate atomically to the other, while the two variables are bound. This allows the designer to express location-dependent consistency, i.e., when components are "near" each other, a high degree of consistency is maintained due to the availability of a high bandwidth, low latency communication channel. When components move apart, only a low degree of consistency would be possible for good performance in the face of decreased or non-existent bandwidth. For example, assume for a moment that the clients *P(i)* are allowed to migrate around a network of workstations for the purpose of load balancing. The server resides on one of the workstations, and some clients may also be located there. Due to the availability of operating system support for inter-process shared memory, the designers decide that the clients which are co-located with the server may share the message buffers directly with the server rather than using asynchronous message passing. The Mobile UNITY specification of this sharing could be written as:

$$Server.q(i) \approx P(i).RQ \qquad \textbf{when } Server.\lambda = P(i).\lambda$$

which states that the request buffer of client i should be shared with $Server.q(i)$ when the two components are at the same place. In the case of a network of workstations, location would be the IP address of the machine, for instance.

While a pair of transiently shared variables are disconnected, they may take on different values, because assignments to one are not immediately propagated to the other. This may present a problem when the variables are later reconnected. If they are to be treated intuitively as one variable, they should immediately take on the same value when they become shared. Mobile UNITY allows for the specification of an **engage** value, which is assigned atomically to both variables immediately upon a transition of the **when** predicate from false to true. We may want to specify that the message buffer will take on the value present at the server when a new process arrives—e.g., because implementation details of inter-process shared memory make it more efficient for the buffers to be allocated in a single block, and thus placed on the server. We would write this as

$$\begin{array}{ll} Server.q(i) \approx P(i).RQ & \textbf{when } Server.\lambda = P(i).\lambda \\ & \textbf{engage } Server.q(i) \end{array}$$

Similarly, a pair of **disengage** values may be specified that are assigned atomically to the variables when they become disconnected. If, for instance, it is too expensive to copy the contents of the message buffer into the client when it moves to a new workstation, but the contents should be retained at the server, we would specify

$$\begin{array}{ll} Server.q(i) \approx P(i).RQ & \textbf{when } Server.\lambda = P(i).\lambda \\ & \textbf{engage } Server.q(i) \\ & \textbf{disengage } Server.q(i),\ \perp \end{array}$$

This particular disengage pattern, where one variable retains its old value and the other is cleared, is quite common in the examples that follow. Note that if no **engage** value is specified, the variables will remain in an inconsistent state after sharing takes effect until the first assignment is propagated. If no **disengage** value is specified, the variables will retain the values they had before the variables are disconnected. Later examples will make extensive use of transient sharing, including one-way sharing, where updates are propagated in one direction but not the reverse. This is expressed as above except with an $\leftarrow$ in place of $\approx$, pointing in the same direction as updates are to be propagated.

To accommodate shared variables, as well as other forms of component interaction, Mobile UNITY makes certain adjustments to the standard UNITY operational model. Because the updates to shared variables must happen in the same atomic step as an assignment, but sharing is specified separately from the (possibly many) assignments that may change the value of a variable, Mobile UNITY has a two-phased operational model where the first phase is an ordinary assignment statement and the second is responsible for propagating changes to shared variables. We call the statements that execute in the second phase *reactive statements*, and they are denoted in the text of a Mobile UNITY program by the use of **reacts-to** in place of **if**. Mobile UNITY also allows for global constraints on the execution of statements, called **inhibit** clauses. These serve to strengthen the guard on a statement and can express scheduling constraints that may not be expressible using only local state information. A third construct, the *transaction*, is used for specifying a sequence of statements that are executed as a unit with respect to other, non-reactive statements. Together, these primitives provide a powerful notation for expressing inter-component interaction in a highly decoupled and dynamic way, although for the purposes of this paper, we will express our solutions using only shared variables. The reader should keep in mind that transient sharing is really a shorthand notation for a set of reactive statements. Further information, including a proof logic that has been developed to match the new operational model, can be found in [13].

4 Mobile Code Design Paradigms

The idea behind code mobility is not new, as witnessed by the work by Stamos et al. [9] and by Black et al. [1]. Nevertheless, these technologies were conceived mostly to provide operating system support on a LAN, while MCLs explicitly target large scale distributed systems—like the Internet. On the Internet, client-server is the most used paradigm for the development of applications. In this paradigm, the application code is statically bound to the client and server hosts and the binding cannot be changed during the execution of the distributed application. Notably, each interaction between the client and the server must exploit the communication infrastructure through message passing or some higher level mechanism like remote procedure call (RPC). Mechanisms that actually hide the location of components from the application programmer are also being considered, e.g., CORBA [15].

By contrast, in MCLs component locations are not hidden. Location is explicitly handled by the programmer who is able to specify *where* the computation of a given code fragment must take place. This capability leads to new paradigms for the design of distributed applications where the interaction between client and server is no longer constrained to exchanging simple, non-executable data via the network. For example, a portion of the client may move in order to bypass the communication infrastructure and to achieve local interaction with the server. This may improve performance by reducing latency and may increase dependability by avoiding problems inherent in partial failures.

The essential features of the interaction patterns found in MCLs can be characterized by considering the kinds of pairwise interactions that are possible between two software design components located on different hosts. As shown in [2], we can accomplish this without having to appeal to the details of any particular language. In this section, we introduce the reader to three mobile code design paradigms and show how to use them and the conventional client-server paradigm to construct distributed solutions for the problem stated in Section 2. The mobile code design paradigms, defined in [2], are called Remote Evaluation, Mobile Agent, and Code on Demand. A schematic view of all the paradigms appears in Figure 5.

In the following solutions the GVT is always computed by a single component, like in the CS solution, and its value is communicated back to the processes *P(i)*. Thus, we continue to refer to components *P(i)* as *clients* of the process computing the GVT, whether they are mobile or not. Furthermore, as in the CS solution we assume that the value for the simulation mode z is computed statically. We will relax this assumption in Section 4.3, where we describe a solution based on the COD paradigm that involves the computation of a dynamically changing simulation mode.

4.1 Remote Evaluation

The *Remote EValuation (REV)* paradigm can be regarded as a variation of the CS paradigm where the server component offers its computational power and resources, like the server in a CS paradigm, but does not provide any application specific service. Again, the client component lacks some of the resources needed to proceed with

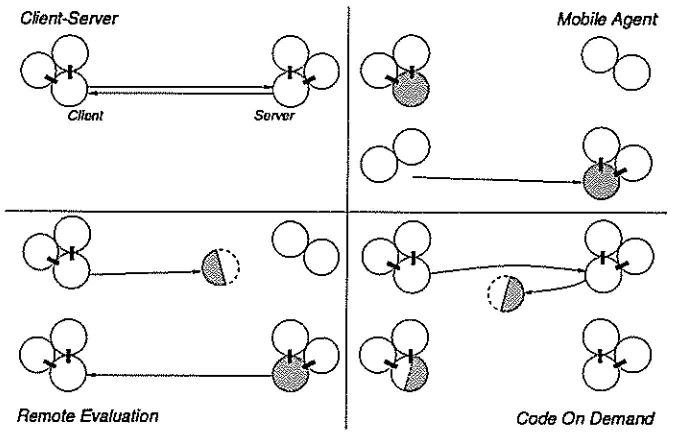

Figure 5: Mobile code design paradigms. Components are identified by circles, and bindings between components are identified by black rectangles connecting circles. A component can be thought of as being composed of code, data state (holding data local to the component), and control state (holding information about its run-time execution). Grayed components participate in migration. Half-grayed dashed circles represent migration of a portion of a component, i.e. the code, or code plus the data state. Such portions can be used to create a new component, like in REV, or to augment an already existing component, like in COD.

computation and these resources are located on the server host. Nevertheless, in the REV paradigm the know-how needed to access the resources is not predetermined and co-located statically with the resources, rather it must be provided by the client that needs access to the resources. No control state is provided for the component, while some initial data state can be provided in order to set an initial environment for component execution. In other words, in the REV paradigm the client provides the code constituent for a component that will be instantiated on the server and bound there to the resources it needs to access. Eventually, a result will be sent back to the client component via a message, like in a CS paradigm. Hence, the REV paradigm leverages off the flexibility provided by the server, instead of relying on a fixed functionality. The REV paradigm is inspired by work on the REV [9] system, which extends remote procedure call with one additional parameter containing the code to execute on the server. Among recent MCLs, the paradigm is supported natively in TACOMA, Agent Tcl, Facile, and M0.

In the solution shown in Figure 6, the client components *P(i)* behave similarly to the ones in the *DSClientServer* system. Besides computing a new local timer and using the timer value to update their simulation variable x, clients can also request the *Server* component to compute the new GVT estimate for them, provided that the old GVT estimate has been already consumed during an earlier timer update and that a message request has not yet been sent. The reply message is eventually collected and its fields are checked to verify that the recipient's and receiver's addresses match. In this case, the new estimate becomes defined and available for the assignment that updates the timer, while the message buffer is reset to enable further requests—just like in the CS solution. The key difference between the two solutions is the second parameter in the message request

$$RQ := \langle \text{SERVER}, \text{REV}, \text{MINSERV}, \perp\rangle \quad \text{if } \neg\text{def}(RQ) \wedge \neg\text{def}(T)$$

specifying that the service named MINSERV, which will be executed on the *Server* location, must be sent together with the message request. The code for service MINSERV is described by the program *Min(i)* and is made of two steps: the global state maintaned in τ on the *Server* is first updated with the timer value for *P(i)* and then the new GVT is computed. These steps can take place only if *Min(i)* has access to the global state. This happens as soon as the message request is issued, that is, when *Min(i)*, initially co-located with the corresponding client *P(i)*, becomes co-located with *Server*. Correct ordering between statements is guaranteed by means of the variable *updated*, which is set upon modification of the global state and reset upon computation of the new GVT. The global state τ is hosted by the *Server* component, which provides also the code to construct and append to an output queue the message replies for the clients. In

$$\langle [\![\; j :: q(j), T, v := \langle j, \perp, \perp, T\rangle, \perp, \perp \quad \text{if } v = j \wedge \text{def}(T) \wedge \neg\text{def}(q(j)) \rangle$$

the reply containing the estimate T for a process with address j is constructed only if this reply is not already pending on the output queue q and if the the current value of T in *Server* has been computed by the corresponding component *Min(j)*. This latter condition is expressed using a semaphore v, which must be held exclusively by a component *Min(j)* before it updates the GVT estimated and is released after the reply message has been constructed

```
System DSRemoteEvaluation
    Program P(i) at $\lambda$
        declare
            $x, t, z$ : integer $[]$ $T$ : integer $\cup \{\perp\}$ $[]$ $RQ$ : request $\cup \{\perp\}$
        initially
            $x, t, z = 0, 0, \bar{z}$ $[]$ $T = \perp$ $[]$ $\lambda = \text{Location}(i)$ $[]$ $RQ = \perp$
        assign
            $x := t$
          $[]$ $t, T := f_i(t, T, z), \perp$                          if $\text{def}(T)$
          $[]$ $RQ := \langle \text{SERVER}, \text{REV}, \text{MINSERV}, \perp \rangle$     if $\neg\text{def}(RQ) \wedge \neg\text{def}(T)$
          $[]$ $T, RQ := RQ \uparrow 4, \perp$                        if $RQ \uparrow 1 = i$
    end
    Program Min(i) at $\lambda$
        declare
            $t$ : integer $[]$ $T$ : integer $\cup \{\perp\}$ $[]$ $\tau$ : (array of integer) $\cup \{\perp\}$ $[]$ $updated$ : boolean
        initially
            $t, T = 0, \perp$ $[]$ $\tau = \perp$ $[]$ $updated = false$ $[]$ $\lambda = \text{Location}(i)$
        assign
            $\tau(i), updated := t, true$                        if $\text{def}(\tau) \wedge \neg updated$
          $[]$ $T, updated = \langle \min k :: \tau(k) \rangle, false$          if $\neg\text{def}(T) \wedge \text{def}(\tau) \wedge updated$
    end
    Program Server at $\lambda$
        declare
            $T$ : integer $\cup \{\perp\}$ $[]$ $\tau$ : array of integer $[]$ $q$ : array of (request $\cup \{\perp\}$) $[]$ $v$ : clientAddress $\cup \{\perp\}$
        initially
            $T = \perp$ $[]$ $\langle \| j :: \tau(j) = 0 \rangle$ $[]$ $\langle \| j :: q(j) = \perp \rangle$ $[]$ $\lambda = \text{Location}(\text{SERVER})$ $[]$ $v = \perp$
        assign
            $\langle [] j :: q(j), T, v := \langle j, \perp, \perp, T \rangle, \perp, \perp$     if $v = j \wedge \text{def}(T) \wedge \neg\text{def}(q(j)) \rangle$
    end
    Components
            $\langle [] i :: P(i) \rangle$ $[]$ $Server$ $[]$ $\langle [] i :: Min(i) \rangle$
    Interactions
            $Min(i).t \leftarrow P(i).t$             when $Min(i).\lambda = P(i).\lambda$
          $[]$ $Min(i).\lambda := Server.\lambda$         when serviceRequest(REV, MINSERV, $i$) $\wedge$ $Min(i).\lambda = P(i).\lambda$
          $[]$ $Min(i).\tau \approx Server.\tau$          when $Min(i).\lambda = Server.\lambda$
                                               engage $Server.\tau$
                                               disengage $\perp$, $Server.\tau$
          $[]$ $v := i$                            when $Min(i).\lambda := Server.\lambda \wedge v = \perp$
          $[]$ $Min(i).T \approx Server.T$          when $Min(i).\lambda = Server.\lambda \wedge v = i$
                                               engage $\perp$
          $[]$ $P(i).RQ, Server.q(i)[, Min(i).\lambda] := Server.q(i), \perp[, P(i).\lambda]$     when serviceReady($i$)
end
```

Figure 6: Remote Evaluation solution for the distributed simulation problem.

in *Server*. The semaphore v and the variable bindings across different components are specified in the **Interactions** section. The sharing interaction

$$Min(i).t \leftarrow P(i).t \qquad \textbf{when } Min(i).\lambda = P(i).\lambda$$

specifies that the timer value in a client *P(i)* is shared with the one in the corresponding *Min(i)*—as long as they are co-located. Upon departure of *Min(i)*, it will retain the current value[2] of t which will be used after

$$\begin{array}{ll} Min(i).\tau \approx Server.\tau & \textbf{when } Min(i).\lambda = Server.\lambda \\ & \textbf{engage } Server.\tau \\ & \textbf{disengage } \perp, \ Server.\tau \end{array}$$

takes effect. This statement specifies that the global state τ maintained in *Server* can become shared with the corresponding variable in *Min(i)* if the two components are co-located. The **engage** clause guarantees that the value for τ is up-to-date (i.e., it is the one maintained in *Server*), while the **disengage** clause resets τ in *Min(i)*, thus preventing further computation. If this variable sharing is established, the *Server* receives the new value from

[2]Mobile code languages implement parameter passing either explicitly by referring to input parameters and code using RPC-like primitives, as in [9], or implicitly by attaching to the procedure to be executed remotely the portion of data space needed for remote computation, as in [20]. We chose the second alternative, in order to illustrate how to dynamically establish and remove bindings among variables in Mobile UNITY.

P(i) that was stored in *Min(i)*. Migration of *Min(i)* to the *Server* is provided when the macro serviceRequest detects the presence of a message request—as in the CS solution. The semaphore v is acquired by a component *Min(i)* upon execution of assignment

$$v := i \quad \textbf{when } Min(i).\lambda := Server.\lambda \wedge v = \perp$$

whose guard guarantees mutual exclusion. The sharing of the GVT estimate T between *Min(i)* and *Server*

$$Min(i).T \approx Server.T \qquad \begin{array}{l}\textbf{when } Min(i).\lambda = Server.\lambda \wedge v = i \\ \textbf{engage } \perp\end{array}$$

is constrained by v—sharing is established only if *Min(i)* has acquired the semaphore. Furthermore, the **engage** clause enables the statement computing GVT in *Min(i)* and prevents communication of old values in *Server*. Finally, in the last statement of the system the message replies pending in the output queue are sent to the corresponding client. This is accomplished like in the CS solution, except for the assignment

$$[Min(i).\lambda := P(i).\lambda] \quad \textbf{when } \mathsf{serviceReady}(i).$$

This statement is enclosed in square brackets to highlight the fact that it is not a direct consequence of the REV paradigm, yet is needed because of the way Mobile UNITY is currently defined. The REV paradigm involves migration of a copy of a component's code. After that, a message containing the result of the computation on the server is sent back to the client, and what happens to the code remaining on the server is left to the implementation. On the other hand, in the **Component** section of the solution presented we create statically N components which are initialized with their own data and control state. These components are unique in the system, consequently they must return to the client's location in order to become available for another message request—dynamic instantiation of components is not yet available in Mobile UNITY. This and other issues will be discussed further in Section 5.

4.2 Mobile Agent

In the *Mobile Agent (MA)* paradigm, the client needs to access some of the resources that are located on the server and, like in the REV paradigm, it owns the necessary know-how to use such resources. In contrast with the REV paradigm, the client sends a whole component that is already being executed on the client host, together with its data and control state. Bindings to resources on the client host are voided and replaced by the new bindings to resources on the server host. The component, once arriving on the server host, resumes execution as if no migration took place. Typically, this step is repeated many times by the same component, which consequently is able to visit a number of hosts on behalf of the client without requiring interaction with it. The MA paradigm is supported natively by languages exploiting strong mobility, like Telescript and Agent Tcl.

Figure 7 shows a Mobile UNITY system designed using the MA paradigm. As in the CS solution, the client processes *P(i)* can compute the value of the simulation variable x using their local timer or increment the timer value, consuming the local estimate for GVT—which prevents further timer increments until a new estimate becomes available. In contrast with the CS solution, no handling of message requests is needed. The location of each client is initialized to a given value, which cannot be changed. The *Agent* component, in turn, is initially co-located with an arbitrarily chosen client and changes explicitly its location during execution in order to visit all clients in a round-robin fashion. The *Agent* carries with it the global state of all local timers, which is updated with the timer value of a client *P(i)* while the two components are co-located. This update consumes the value of the timer within the *Agent*: this fact is used to guarantee that the update of the GVT value together with migration to the next client location are accomplished only after the global state has been updated. The actions performed by the components above are coordinated by the transient variable sharing defined in the **Interactions** section. The GVT estimate and the local timer belonging to the client *P(i)* are shared with their analogues within the *Agent*, as long as the two components are co-located. When the *Agent* arrives at the client location, the **engage** clause specified in the read-only shared variable definition

$$Agent.t \leftarrow P(i).t \qquad \begin{array}{l}\textbf{when } P(i).\lambda = Agent.\lambda \\ \textbf{engage } P(i).t\end{array}$$

actually communicates the timer value in the client to the *Agent*. On the other hand, upon departure of the *Agent* the **disengage** clause in

$$P(i).T \leftarrow Agent.T \qquad \begin{array}{l}\textbf{when } P(i).\lambda = Agent.\lambda \\ \textbf{disengage } P(i).T,\ \perp\end{array}$$

guarantees that the client holds the new estimate for GVT while the *Agent* resets its current GVT value. This fact is used upon arrival of the *Agent* at the next client location, in order to guarantee that every update (based on the old estimate value) of the local timer within the new client is forbidden until a new estimate has been computed by the *Agent* and communicated to the client through transient sharing.

```
System $DSMobileAgent$
    Program $P(i)$ at $\lambda$
        declare
            $x, t, z$ : integer $[]$ $T$ : integer $\cup \{\perp\}$
        initially
            $x, t, z = 0, 0, \bar{z}$ $[]$ $T = \perp$ $[]$ $\lambda = \text{Location}(i)$
        assign
            $x := t$
            $[]$ $t, T := f_i(t, T, z), \perp$      if def$(T)$
    end
    Program $Agent$ at $\lambda$
        declare
            $t, T$ : integer $\cup \{\perp\}$ $[]$ $\tau$ : array of integer $[]$ $pos$ : clientAddress
        initially
            $t, T = \perp, \perp$ $[]$ $\langle \| j :: \tau(j) = 0 \rangle$ $[]$ $\lambda = \text{Location}(pos)$
        assign
            $\tau(pos), t := t, \perp$                                                      if def$(t)$
            $[]$ $T, pos, \lambda := \langle \min k :: \tau(k) \rangle, pos + 1 \bmod N, \text{Location}(pos + 1 \bmod N)$   if $\neg$def$(t)$
    end
    Components
            $\langle [] \; i :: P(i) \rangle$ $[]$ $Agent$
    Interactions
            $P(i).T \leftarrow Agent.T$        when $P(i).\lambda = Agent.\lambda$
                                      disengage $P(i).T, \perp$
        $[]$ $Agent.t \leftarrow P(i).t$        when $P(i).\lambda = Agent.\lambda$
                                      engage $P(i).t$
end
```

Figure 7: Mobile Agent solution for the distributed simulation problem.

4.3 Code On Demand

The *Code On Demand (COD)* paradigm is gaining in popularity mainly due to the success of the Java language. In this paradigm, a component on a host performs some kind of computation on its local resources. When it recognizes that a portion of the know-how needed to perform the computation is lacking, the know-how is retrieved from some host on the network. The retrieved code augments the one already present in the client component and new bindings may be established on the client host. After this, the client component can resume execution. Hence, in the COD paradigm, in contrast with the paradigms discussed before, the resources are co-located with the client component that can access them freely, and the know-how needed to perform computation on the resources is sent to the client. The COD paradigm is natively supported in Java through the class loader feature, as discussed before. Tcl derivatives provide a similar feature, through an unknown function that is automatically invoked when a procedure is not found by the Tcl interpreter and whose code is under the control of the programmer.

We present the solution exploiting the COD paradigm by enhancing the CS solution shown earlier. In the system shown in Figure 8, clients *P(i)* are augmented with some statements that enable them to request the code needed to compute dynamically the simulation mode. The requests are issued when a given condition is established, which is modeled by the assignment setting the variable *static* to *false*. This variable is *true* initially, thus the clients initially behave like the ones in *DSClientServer*, which use the initial value $\bar{z}$ for the simulation mode. When $\neg static$ is established, the client is enabled to issue a request

$$RQ := \langle \text{SERVER}, \text{COD}, \text{DYNMODE}, \perp \rangle \quad \text{if } \neg\text{def}(RQ) \wedge \neg static$$

in order to make the code for the service DYNMODE available to *P(i)*. This code is contained in program *Dyn(i)*, which simply contains an assignment to update the simulation mode by evaluating the function $d(T)$—provided that the GVT estimate is currently defined in *P(i)*. We assume that, once the simulation mode is computed dynamically, it can no longer be reverted to a statically determined value. The *Server* component is left unmodified with respect to the CS solution, while N components *Dyn(i)* are instantiated in the **Components** section—for reasons similar to those explained for the REV solution. Within the **Interactions** section, the last two statements are unchanged and manage message exchange needed to compute the GVT estimate with a CS paradigm. In turn, the statement

$$Dyn.\lambda, P(i).static := P(i).\lambda, \perp \quad \textbf{when } \text{serviceRequest}(\text{COD}, \text{DYNMODE}, i) \wedge Dyn(i).\lambda = Server.\lambda$$

satisfies a code request issued by a client *P(i)* by changing the location of the corresponding component *Dyn(i)*. Furthermore, it prevents further changes in the way *z* is computed by setting *static* to undefined—which permanently

```
System DSCodeOnDemand
    Program P(i) at $\lambda$
        declare
            $x, t, z$ : integer $[]$ $T$ : integer $\cup \{\perp\}$ $[]$ $static$ : boolean $\cup \{\perp\}$ $[]$ $RQ$ : request $\cup \{\perp\}$
        initially
            $x, t, z = 0, 0, \bar{z}$ $[]$ $T = \perp$ $[]$ $\lambda = \text{Location}(i)$ $[]$ $static = true$ $[]$ $RQ = \perp$
        assign
            $x := t$
          $[]$ $static := false$
          $[]$ $t, T := f_i(t, T, z), \perp$                      if $\text{def}(T)$
          $[]$ $RQ := \langle \text{SERVER}, \text{CS}, \text{MINSERV}, t \rangle$          if $\neg\text{def}(RQ) \wedge \neg\text{def}(T)$
          $[]$ $T, RQ := RQ \uparrow 4, \perp$                     if $RQ \uparrow 1 = i$
          $[]$ $RQ := \langle \text{SERVER}, \text{COD}, \text{DYNMODE}, \perp \rangle$       if $\neg\text{def}(RQ) \wedge \neg static$
    end
    Program Dyn(i) at $\lambda$
        declare
            $z$ : integer $[]$ $T$ : integer $\cup \{\perp\}$
        initially
            $\lambda = \text{Location}(\text{SERVER})$
        assign
            $z := d(T)$      if $\text{def}(T)$
    end
    Program Server at $\lambda$
        declare
            $T$ : integer $\cup \{\perp\}$ $[]$ $\tau$ : array of integer $[]$ $q$ : array of (request $\cup \{\perp\}$)
        initially
            $T = \perp$ $[]$ $\langle \| \, j :: \tau(j) = 0 \rangle$ $[]$ $\langle \| \, j :: q(j) = \perp \rangle$ $[]$ $\lambda = \text{Location}(\text{SERVER})$
        assign
            $\langle [] \, j :: \tau(j), q(j) \uparrow 2 := q(j) \uparrow 4, \text{WAIT} \rangle$       if $q(j) \uparrow 1 = \text{SERVER} \wedge q(j) \uparrow 2 \neq \text{WAIT}$
          $[]$ $T := \langle \min k :: \tau(k) \rangle$                  if $\langle \exists \, j :: q(j) \uparrow 2 = \text{WAIT} \rangle \wedge \neg\text{def}(T)$
          $[]$ $\langle \| \, j :: q(j), T := \langle j, \perp, \perp, T \rangle, \perp$          if $\text{def}(T) \wedge q(j) \uparrow 2 = \text{WAIT} \rangle$
    end
    Components
            $\langle [] \, i :: P(i) \rangle$ $[]$ $Server$ $[]$ $\langle [] \, i :: Dyn(i) \rangle$
    Interactions
            $Dyn.\lambda, P(i).static := P(i).\lambda, \perp$          when $\text{serviceRequest}(\text{COD}, \text{DYNMODE}, i) \wedge Dyn(i).\lambda = Server.\lambda$
          $[]$ $P(i).z \leftarrow Dyn(i).z$                    when $P(i).\lambda = Dyn(i).\lambda$
                                                   engage $P(i).z$
          $[]$ $Dyn(i).T \leftarrow P(i).T$                    when $P(i).\lambda = Dyn(i).\lambda$
                                                   engage $P(i).T$
          $[]$ $Server.q(i) := P(i).RQ$                  when $\neg\text{def}(Server.q(i)) \wedge \text{serviceRequest}(\text{CS}, \text{MINSERV}, i)$
          $[]$ $P(i).RQ, Server.q(i) := Server.q(i), \perp$       when $\text{serviceReady}(i)$
end
```

Figure 8: Code On Demand solution for the distributed simulation problem.

disables the statement issuing the request. Finally,

$$P(i).z \leftarrow Dyn(i).z \qquad \textbf{when } P(i).\lambda = Dyn(i).\lambda \quad \textbf{engage } P(i).z$$

and

$$Dyn(i).T \leftarrow P(i).T \qquad \textbf{when } P(i).\lambda = Dyn(i).\lambda \quad \textbf{engage } P(i).T$$

specify the bindings established between *P(i)* and *Dyn(i)* when they are co-located. The **engage** clauses initialize the values of z and T in *Dyn(i)* with the corresponding values in *P(i)*. As in the REV solution, we are forced to instantiate statically multiple components from the same program, instead of migrating the code and instantiate components only when and if needed. This and other issues raised by the solutions presented so far will be discussed in the next section.

5 Conclusions

Mobile UNITY is a new model of distributed computing specialized for mobile computation, i.e., for systems in which components travel through space, compute in a decoupled fashion, and communicate opportunistically when co-located. Mobile UNITY provides a notation system for capturing mobility and an assertional proof logic. Research on Mobile UNITY has shown that a small number of constructs suffice to express transitive forms of transient data sharing and transient synchronization. Restricted forms of these proposed interaction constructs appear to have efficient implementations and more abstract and powerful interaction constructs can be built from the basic forms. In addition, the proof logic has been tentatively evaluated in the verification of the Mobile IP protocol [14].

Against this background of promising technical developments, this paper raised a simple question: Can Mobile UNITY model in straightforward manner the kinds of interaction that take place in applications involving mobile code? The question is reasonable in light of the fact that Mobile UNITY makes no explicit distinction between physical and logical movement of components. A positive answer would allow the immediate application of its proof logic to mobile code and may also define a more abstract and objective basis for a different kind of taxonomy of mobile code paradigms. A negative answer would lead to a clarification of possible fundamental differences between the domains of mobile code and mobile components or could reveal possible shortcomings in the way Mobile UNITY was conceived.

The investigative style of this paper is empirical. We started with established mobile code paradigms and sought out corresponding Mobile UNITY solutions. As expected, the decoupled style of computation promoted by Mobile UNITY appears to be a good match for the realities of mobile code. The **Interactions** section was able to encapsulate appropriately the communication taking place between components. Asynchronous data transfer had a direct counterpart in Mobile UNITY and code movement was easily expressed by the same mechanisms by which components change location. Because the only notion of blocking in Mobile UNITY is busy waiting, blocking for responses to requests was naturally captured by tagging relevant variables as being unavailable (undefined) and strengthening the guards of related statements to check for availability of the data. In several cases we used the fact that the variable was no longer available as the cause for generating a request in the first place—this led to elegant separation between action embedded in the application program and those supplied by the run-time support. Finally, the transient sharing constructs offered a good solution for the data binding process that needs to take place when a mobile code fragment arrives at a new location. Since the mobile code is treated as a program having its own internal state, the movement of code can be accompanied by data movement. The **engage** feature of transient variable sharing encapsulates the binding process while the **disengage** plays a role in implementing policies that define how much state information maybe carried along by a departing code fragment. When a piece of code carries no data state, for instance, the disengagement reinitializes all its shared variables.

The only possible mismatch identified by this case study has to do with dynamic instantiation of code segments. In the REV solution, for instance, there is no need to "return" the code being evaluated as we do in our example. New fresh copies can be sent each time and several copies may co-exist on different servers. In Mobile UNITY, however, the set of components making up a system is fixed. Further research is needed to evaluate this issue. One solution that requires no change to the Mobile UNITY notation is to create an unbounded set of clones (uniquely indexed) that are placed in a stand-by state until needed. This could have some negative implications on verification and, if not considered carefully, could interfere with the fairness assumption which are at the foundation of the proof logic. Another option involves building upon the experience with Swarm [17, 6], a version of UNITY in which both data and statements are dynamically created and destroyed. The prospect of making changes to Mobile UNITY may also force us to re-examine the issue of what is an appropriate unit of mobility. So far we selected the program to play this role but a finer grained solution at the statement level, for instance, may also be appropriate to consider.

Acknowledgements This paper is based upon work supported in part by the National Science Foundation under Grant No. CCR-9217751. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Gian Pietro Picco was partially supported by Centro Studi e Laboratori Telecomunicazioni S.p.A., Italy. Furthermore, he wishes to thank people at Politecnico di Milano who shared with the author most of the research work about code mobility.

References

[1] A. Black, N. Hutchinson, E. Jul, and H. Levy. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[2] A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with a Mobile Code Paradigm. In A. Fuggetta and A. Wolf, editors, *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, 1997. To appear.

[3] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transaction on Software Engineering*, 5(5):440–452, September 1979.

[4] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[5] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems.* Lecture Notes on Computer Science, 1996. To appear.

[6] H.C. Cunningham and G.-C. Roman. A UNITY-style Programming Logic for a Shared Dataspace Language. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.

[7] R. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-825, Department of Computer Science, Darmouth College, Hanover, May 1996. `ftp://ftp.cs.dartmouth.edu/TR/TR96-285.ps.Z`.

[8] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM'95 Workshop on Intelligent Information Agents.*

[9] J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[10] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An Introduction to the TACOMA Distributed System - Version 1.0. Technical Report 95-23, University of Tromsø and Cornell University, June 1995.

[11] F.C. Knabe. Language Support for Mobile Agents. Technical Report ECRC-95-36, European Computer-Industry Research Centre, Munich, Germany, December 1995.

[12] General Magic. *Telescript Language Reference.* General Magic, October 1995.

[13] P.J. McCann and G-.C. Roman. Mobile UNITY: A Language and Logic for Concurrent Mobile Systems. Technical Report WUCS-97-01, Department of Computer Science, Washington University in St.Louis, December 1996. Submitted for publication.

[14] P.J. McCann and G-.C. Roman. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. Technical Report WUCS-96-15, Department of Computer Science, Washington University in St.Louis, May 1996.

[15] Object Management Group. *CORBA: Architecture and Specification*, August 1995.

[16] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1995.

[17] G.-C. Roman and H.C. Cunningham. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, December 1990.

[18] G-.C. Roman, P.J. McCann, and J.Y. Plun. Assertional Reasoning about Pairwise Transient Interactions in Mobile Computing. In *Proceedings of the 18th International Conference on Software Engineering*, pages 155–164. IEEE Computer Society Press, March 1996.

[19] Sun Microsystems. *The Java Language Specification*, October 1995.

[20] B. Thomsen. Facile Antigua Release Programming Guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany, December 1993.

[21] C. F. Tschudin. *An Introduction to the* `MO` *Messenger Language.* University of Geneva, Switzerland, 1994.

[22] J. Vitek and C. Tschudin, editors. *Mobile Object Systems.* Lecture Notes on Computer Science. Springer-Verlag, 1997. Special issue, to appear.

[23] J.E. White. Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents.* MIT Press, 1996.