Reasoning About Code Mobility with Mobile UNITY

GIAN PIETRO PICCO
Politecnico di Milano
and
GRUIA-CATALIN ROMAN
Washington University in Saint Louis
and
PETER J. MCCANN
Lucent Technologies

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 and the ability to construct assertional-style proofs for programs that employ code mobility.

Categories and Subject Descriptors: C.2.1 [**Distributed Systems**]: Distributed Applications; D.2.4 [**Software/Program Verification**]: Correctness Proofs; D.2.10 [**Specifying and Verifying and Reasoning about Programs**]; D.2.11 [**Software Architectures**]: Patterns; F.1.2 [**Modes of Computation**]: Parallelism and Concurrency

General Terms: Design, Theory, Verification, Languages

Additional Key Words and Phrases: Code mobility, mobile agent, UNITY

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

Authors' addresses: G.P. Picco, Dip. di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci 32, 20133 Milano, Italy, e-mail: `picco@elet.polimi.it`; G.-C. Roman, Dept. of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA. E-mail: `roman@cs.wustl.edu`; P.J. McCann, Lucent Technologies, 263 Shuman Blvd, Naperville, IL 60566-7050. E-mail: `mccap@research.bell-labs.com`
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

they are executed [Carzaniga et al. 1997]. 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* [Vitek and Tschudin 1997]. These languages provide specialized abstractions and run-time support capabilities designed to support various forms of code mobility. The impetus for the development of mobile code languages 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 mobile code languages, the reader is directed to [Cugola et al. 1997] which reviews a number of existing languages and attempts to extract their essential features. The unit of mobility, called *executing unit* in [Cugola et al. 1997], 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.

Current mobile code languages provide different mixtures of the above notions of code mobility. For instance, Telescript [Magic 1995; White 1996] 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 [Gray et al. 1997; Gray 1995], an extension of Tcl [Ousterhout 1995], 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 [Sun Microsystems 1995] 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. Java derivatives like Aglets [Lange and Oshima 1998], Voyager [Kiniry and Zimmerman 1997], or Mole [Straßer et al. 1996], as well as languages like TACOMA [Johansen et al. 1995], Facile [Knabe 1995], and M0 [Tschudin 1994] 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, mobile code languages 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 focus is on the modeling of programs that exhibit code mobility and on the construction of assertional proofs for them. The approach we pursue is that of expressing various forms of code mobility in a model that was originally designed for the specification and verification of mobile computing systems, i.e., systems which involve software components bound to hosts that move in space and interact opportunistically when communication becomes possible. We believe this to be an important first step towards constructing formal models specific to code mobility as it highlights the differences and similarities between the two kinds of mobility. Furthermore, these kinds of studies may eventually lead to the development of a unified model of mobility.

The model we use in our study is called *Mobile* UNITY [Roman et al. 1997; McCann and Roman 1998], an extension of work by Chandy and Misra [Chandy and Misra 1988] on UNITY. 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.

In Mobile UNITY, the unit of mobility is a program. Migration is captured by augmenting the program state with a location attribute whose change in value is used to represent motion. In this manner, UNITY is augmented with an explicit representation of space and its properties—we contend that modeling the space explicitly is desirable when one hopes to take into account the physical reality of moving hosts and its implications on the behavior of the software they carry. While Mobile UNITY is usually concerned with movement in physical space, there is nothing in the model that precludes a more abstract view of space. Nodes of a network or even distinct memory partitions on a single node can be viewed as locations in a domain whose properties are distinct from those of the physical space, e.g., one in which all points are equidistant—a reasonable assumption for a model which ignores variations in communication delays. In the remainder of the paper we will show how one can treat mobile code fragments as Mobile UNITY programs, how one can specify code movement from one location to another by manipulating the location variable, and how one can capture dynamic binding of variables among the mobile and stationary code by means of transient interactions, a concept specific to Mobile UNITY, following the lines of [Picco et al. 1997]. Having accomplished this, we will also show how the Mobile UNITY proof logic enables us to verify the correctness of programs using mobile code.

The remainder of this paper is structured as follows. Section 2 describes a simplified version of a distributed simulation problem. Several solutions to this problem are discussed in this paper to illustrate modeling and verification techniques enabled by Mobile UNITY. The problem is used first in Section 3 to introduce standard UNITY via a centralized solution. A distributed, client-server solution is then presented in Section 4; it illustrates also how Mobile UNITY extends UNITY to model naturally the interactions taking place among distributed components. This theme is refined further in Section 6, where the Mobile UNITY constructs enabling transient interactions are used to model the coordination taking place among mobile components in three different solutions to the distributed simulation problem. The verification of the corresponding programs is discussed in Section 7, that presents an overview of the Mobile UNITY proof logic and sketches the proofs for the mobile code programs. Further details about the proofs are provided in the Appendix. Section 8 explores some of the issues raised by our investigation and discusses the kinds of features that are needed in a richer model of mobility. Brief concluding remarks appear in Section 9.

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. The example is inspired by the work of Chandy and Misra who provided a formal characterization and solution for a distributed simulation problem [Chandy and Misra 1979]. 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 in the next section, the value of the GVT can be stored in a variable and can be updated

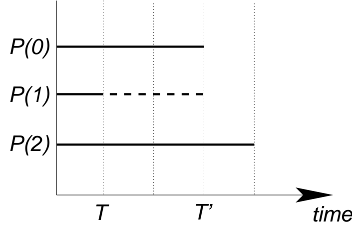

Fig. 1. A snapshot of the distributed simulation.

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 $P(1)$. After the execution of its action, $P(1)$ is 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 $P(0)$ and $P(1)$ are now allowed to execute their respective actions, independently of each other.

The distributed simulation problem can be characterized formally as follows. 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:

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

$$g_i(t(i), z) > t(i)$$

where $t(i)$ denotes the value for a local timer, T denotes the GVT value, and z identifies the simulation mode. This additional parameter accounts for some entity other than time that takes part in determining the behavior of the simulated component. These definitions capture the following requirements:

(1) The local timer cannot change if it is ahead of the GVT.

[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 computing the minimum across the N values.

(2) The local timer, if permitted, can only increase, i.e., actions are always scheduled in the future (see the constraint on g).

(3) The value of a local timer can never be behind the GVT. For such cases the function f is undefined.

The distributed simulation problem will be used in the sections to come as a reference example to present solutions that exploit different degrees of distribution and mobility, and discuss their modeling using Mobile UNITY. A centralized solution will be described in the next section, together with an introduction to the original UNITY notation. The drawbacks of this solution are analyzed and then removed in Section 4, that shows how the constructs introduced by Mobile UNITY can be used to model distributed components interacting through the common client-server paradigm. Mobile code variants of this scenario are then discussed in Section 6, where the ability of Mobile UNITY to express concisely transient interaction and autonomous migration are employed to model the dynamic reconfiguration of code and state. Finally, in Section 7 the properties of the distributed simulation problem are expressed formally using the UNITY logic, and the correctness of the mobile code solutions is demonstrated using the Mobile UNITY proof logic.

3. A CENTRALIZED SOLUTION IN UNITY

Figure 2 shows a program that provides a solution for the distributed simulation problem. The program uses the UNITY notation described in [Chandy and Misra 1988]. All variable declarations appear in the **declare** section. The array t contains the values of the local timers for each process i. 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 assume arbitrary values 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, and has 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 each $t(i)$ 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. The details about the behavior of processes are irrelevant. Finally, the

```
Program DistributedSimulation
    declare
        t : array of integer [] T, z : integer
    initially
        ⟨[] i :: t(i) = 0⟩ [] T = 0
    assign
        T := ⟨min i :: t(i)⟩
      [] ⟨[] i :: t(i) := f_i(t(i), T, z)⟩
      [] z := d(T)
end
```

Fig. 2. A simple UNITY solution for the distributed simulation problem.

value of the simulation mode is updated dynamically according to the definition of a function $d(T)$ whose details are left out.

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 makes explicit the location and encapsulation embodied in each process.

4. 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 different mobile code design paradigm. Each will be expressed in Mobile UNITY [McCann and Roman 1998], a modification of the standard UNITY notation presented in Section 3. In this section we focus our attention on the opportunities for a highly decoupled style of computing promoted by Mobile UNITY. We introduce the Mobile UNITY notation by means of an example that involves a client-server solution to the distributed simulation problem. By distributing the computation among multiple clients and a single server we take a first step towards mobility, i.e., by introducing the notions of location and coordination among independently written components. At this point in our presentation, the emphasis is on understanding the basic differences between UNITY and Mobile UNITY with regard to modular specification of composite systems. The former views a system as a static collection of programs that communicate among themselves by referencing identically named global variables. The latter seeks to maximize decoupling among programs by assuming all variables to be local and by providing a coordination mechanism which is external to the individual programs. A given program exhibits different behaviors as a result of changes in the coordination rules. This section describes only the simplest form of coordination available in Mobile UNITY, i.e., asynchronous data transfer.

4.1 Solution Overview

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

```
System DSClientServer
  Program P(i) at λ
    declare
      t, z : integer [] T : integer ∪ {⊥} [] RQ : request ∪ {⊥}
    initially
      t = 0 [] T = ⊥ [] λ = Location(i) [] RQ = ⊥
    assign
      t, T := f_i(t, T, z), ⊥                  if def(T)
      [] RQ := ⟨SERVER, CS, MINSERV, t⟩        if ¬def(RQ) ∧ ¬def(T)
      [] T, RQ := RQ ↑ 4, ⊥                    if RQ ↑ 1 = i
  end
  Program Server at λ
    declare
      T : integer ∪ {⊥} [] τ : array of integer [] q : array of (request ∪ {⊥})
    initially
      T = ⊥ [] ⟨|| j :: τ(j) = 0⟩ [] ⟨|| j :: q(j) = ⊥⟩ [] λ = Location(SERVER)
    assign
      ⟨[] j :: τ(j), q(j) ↑ 2 := q(j) ↑ 4, WAIT      if q(j) ↑ 1 = SERVER ∧ q(j) ↑ 2 ≠ WAIT⟩
      [] T := ⟨min k :: τ(k)⟩                        if ⟨∃ j :: q(j) ↑ 2 = WAIT⟩ ∧ ¬def(T)
      [] ⟨|| j :: q(j), T := ⟨j, ⊥, ⊥, T⟩, ⊥          if def(T) ∧ q(j) ↑ 2 = WAIT⟩
  end
  Components
      ⟨[] i :: P(i)⟩ [] Server
  Interactions
      Server.q(i) := P(i).RQ                       when ¬def(Server.q(i)) ∧
                                                        serviceRequest(CS, MINSERV, i)
    [] P(i).RQ, Server.q(i) := Server.q(i), ⊥      when serviceReady(i)
end
```

Fig. 3. Client-Server solution for the distributed simulation problem.

located on the server host and they are accessed by the client by interacting with the server through some form of 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.

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 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 $T := \langle \mathsf{min}\ i :: t(i) \rangle$ which imply centralized access to the local timers of each component. The client-server solution of Figure 3, graphically represented in Figure 4, 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.

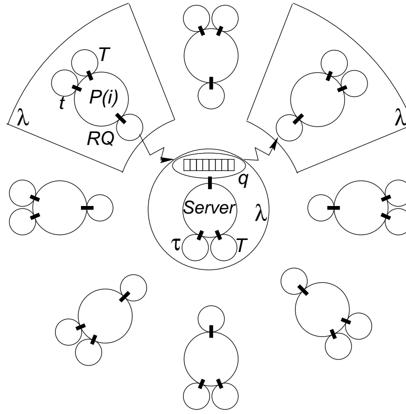

Fig. 4. In the Client-Server solution, communication between the server and the clients takes place via asynchronous message passing.

4.2 System Structure

Figure 3 is illustrative of the new structuring conventions provided by Mobile UNITY. The first line provides the system name, *DSClientServer*. It is followed by a set of program definitions, the first of which, *P(i)*, serves 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.

4.3 Location as a Distinguished Variable

Note that each program definition begins with a line like **Program** *name* **at** λ. This denotes a program that exists at a specific physical or logical location stored in 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 in its **assign** section code 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 contain mobile components, with the connectivity among components depending 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 t and z of the program $P(1)$ have the fully qualified names $P(1).t$, and $P(1).z$. These fully qualified names should be used when carrying out formal reasoning about the behavior of the system.

4.4 Component Coordination

Because variables are local, no communication can take place among components without the presence of interaction clauses spanning the scope of the components. They 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 instantiated 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:

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

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

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

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. As an aside, in the second interaction the symbol $\bot$ is used to denote that the buffer is empty, and consequently the value of $P(i).RQ$ undefined.

4.5 Auxiliary Macro Definitions

The meaning of the macros used in the guards of the interactions is shown in Figure 5. These macros will be used throughout the remaining examples and serve only to improve readability. A request is a tuple consisting of four elements. We denote the tuple by enclosing it in angle brackets and separating its four elements by commas. We use the projection operator $\uparrow$ to access individual fields of a tuple,

clientAddress = $\langle \text{set } n : 0 \leq n \leq N-1 :: n \rangle$
address = $\{\text{SERVER}\} \cup$ clientAddress
opName = $\{\text{CS},\text{REV},\text{COD}\}$
opStatus = $\{\text{WAIT}\}$
request = $\langle$address, opName $\cup$ opStatus $\cup$ $\{\bot\}$, serviceName $\cup$ $\{\bot\}$, integer$\rangle$
serviceRequest(x : opName, y : serviceName, i : clientAddress) $\equiv$
$P(i).RQ \uparrow 1 = \text{SERVER} \wedge P(i).RQ \uparrow 2 = x \wedge P(i).RQ \uparrow 3 = y$
serviceReady(i : clientAddress) $\equiv Server.q(i) \uparrow 1 = i$

Fig. 5. Macro definitions used in the example systems. Allowed values for each data element are represented as sets.

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 represent 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 takes 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 if 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.

4.6 Component Behavior

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 3, except that now the variables t occurs once in each component instead of appearing as global arrays of values: we now use $P(i).t$ in place of $t(i)$. Also, T appears once in each program instead of being globally declared. Note that the simulation mode $P(i).z$, 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. In the code on demand solution presented later, we will compute this value dynamically. The statement to update the local simulation time $P(i).t$ is the same as before, except that the estimated GVT variable $P(i).T$ is simultaneously set to $\bot$ when an update to the local time is made. Throughout the example, the notation def(v) denotes the predicate $v \neq \bot$, 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 second 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 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 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 third assignment statement of the program *P(i)* processes the reply by updating the local GVT estimate $P(i).T$ and clearing the request buffer.

5. MOBILE CODE DESIGN PARADIGMS

The idea behind code mobility is not new, as witnessed by the work by Stamos et al. [Stamos and Gifford 1990] and by Black et al. [Jul et al. 1988]. Nevertheless, these technologies were conceived mostly to provide operating system support on a LAN, while mobile code languages 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. Higher-level mechanisms that by and large hide the location of components from the application programmer, e.g., remote procedure call (RPC) or CORBA [Object Management Group 1995], are also often employed.

By contrast, in mobile code languages 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 thus achieve local interaction with the server. This may improve performance by reducing latency and may increase dependability by avoiding problems inherent in partial failures.

Actually, many researchers see the main advantage of code mobility as an optimization of communication resources. This does not hold in general [Carzaniga et al. 1997; Fuggetta et al. 1998], and applications can benefit from this kind of optimization in other ways as well [Gray et al. 1997; Harrison et al. 1997; Baldi et al. 1997]. We believe that an important benefit of code mobility is the ability to *customize dynamically* the server according to the user's needs. In client-server applications, the server offers a *fixed* set of services, defined *a priori* by the application

designer and accessible through a *statically defined* interface. The services provided or the particular interface may not be suitable for unforeseen user needs. Code mobility can be exploited in order to send on the server active code, as opposed to passive data, that specifies how a given service has to be delivered.

The essential features of the interaction patterns found in mobile code languages 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 [Carzaniga et al. 1997], we can accomplish this without having to appeal to the details of any particular language. In the remainder of this section, we will summarize the taxonomy presented in [Carzaniga et al. 1997] (subject to minor changes in terminology) in order to introduce the reader to the mobile code design paradigms we will express using Mobile UNITY. The interested reader can find an in-depth discussion of mobile code technologies, paradigms, applications, and their inter-relationship in [Fuggetta et al. 1998].

We consider a *design component* as the atomic design element, i.e., the smallest software entity that can be given an identity and can be distinguished from the other entities in an architectural schema. Each design component consists of code and its runtime state. The *code* of a component can be regarded as the specification of the *know-how* placed in it by the system designer, and determines the run-time behavior of the component. The *state* of a component is further decomposed into:

—*Data State.* Local, private data representing features intrinsic to the component are encapsulated within the data state. Retrieval by a component of data contained in its own data state does not involve interactions with other components.

—*Control State.* If a component is being executed, the control state holds the information concerning the status of the execution.

—*Bindings.* A component may contain names that reference other components, that possibly reside on different hosts. If a component C needs access to other bound components in order to be executed successfully, we say that the bound components are *resources* for C. Hence, to retrieve data belonging to its resources, C must have access to and interact with them.

Mobile code design paradigms are characterized by the possibility to *relocate* either a whole component or some of its constituents, in order to bypass the communication infrastructure. In this scenario, the designer of a distributed application is no longer constrained to the limitations of the client-server paradigm, where the know-how and the resources involved in a computation are statically placed on the network hosts. With a mobile code design, the know-how about a component and the resources needed for the associated computation can be located in different places on the network and can be relocated dynamically according to the user's needs. Hence, the space of design choices involved in a mobile code design is extended to include the following:

—Determine which components of the distributed application are bound statically to a network host and which ones are allowed to be relocated at some point in the execution of the application.

—Determine, for each component of the distributed application, where the know-how and resources needed for its associated computation must be placed initially.

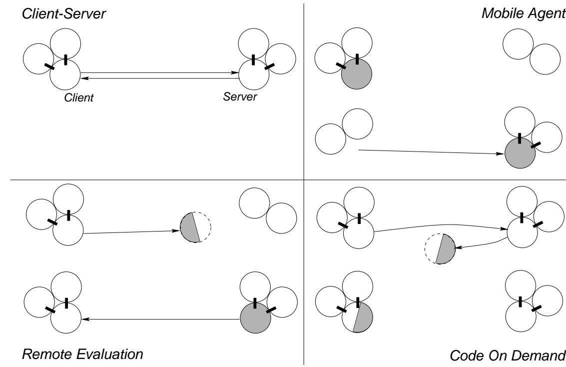

Fig. 6. Mobile code design paradigms. Components are identified by circles, and bindings between components are identified by black rectangles connecting circles. Grayed components participate in migration. Half-grayed dashed circles represent migration of a portion of a component, i.e. the code, or 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. Arrows indicate message traffic, i.e., calls, return values, and code movement.

—Determine which interaction patterns may be exploited and what relocation of components is necessary.

Of primary interest for this paper are the interaction patterns for relocation of components. Nevertheless, even if our definition of a design component in principle allows sophisticated forms of relocation where different combinations of the constituents are moved, an exhaustive analysis about this issue is outside the scope of this work. We limit our discussion to the design paradigms proposed in [Carzaniga et al. 1997] because they reflect the characteristics of currently available mobile code languages. In order to provide a basis for comparison, we include the client-server paradigm in our discussion. A schematic view of all the paradigms discussed here appears in Figure 6. In the next section, we present mobile code solutions developed using the aforementioned design paradigms and specified using Mobile Unity.

6. MOBILE CODE SOLUTIONS IN Mobile Unity

In this section we present three mobile code solutions to the distributed simulation problem introduced in Section 2. Each solution is written in the distinct style associated with the currently dominant design paradigms for mobile code: mobile agent, remote evaluation, and code on demand [Carzaniga et al. 1997]. All three cases entail the movement of a component. Its structure is determined by the granularity of the mobile code language, typically a procedure, process, or object that has associated code, data state, control state, and references to various (typically local) resources. The latter will be referred to as bindings. Given these assumptions about the unit of mobility[2]—henceforth said to be a program—one can easily see that movement may entail not only a change of location but also a transfer

[2]The reader interested in a more detailed discussion of the issues concerned with code mobility is directed to [Fuggetta et al. 1998].

of state information and changes in the bindings. Furthermore, since movement may be initiated at any time, relations which in a traditional scenario are seen as permanent assume a transient form. This requires us to employ a model capable of handling transient interactions among components. Fortunately, the design of Mobile Unity allows us to express a broad range of transient interactions associated with the movement of components. Before turning our attention back to the distributed simulation problem we will take a brief detour to explain how Mobile Unity expresses location-dependent transient interactions.

6.1 Transient Interactions

The coordination mechanisms of Mobile Unity were only partially explained in the previous section—they were limited to the asynchronous transfer of data among components. In the case of mobile components, however, such transfers of data may be conditional upon the relative locations of components, e.g., constrained by co-location on a single host for mobile agents, or by transmission range in the case of mobile hosts with wireless communication. Mobile Unity provides several other high level coordination constructs such as transient variable sharing, which will be shown to be helpful in capturing the resource binding policies of mobile code. Transiently shared variables and other powerful constructs, such as synchronized clocks with and without drift, are all built on top of a small set of coordination primitives described in [McCann and Roman 1998]. We prefer to focus our Mobile Unity review on the high level constructs we actually needed to build the mobile code solutions discussed in this section.

Transiently shared variables 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:

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

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 takes 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

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

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

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

Note that if no **engage** value is specified, the variables remain in an inconsistent state after sharing takes effect until the first assignment is propagated. If no **disengage** value is specified, the variables 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 as well as to the proof logic. 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**. Logically, all reactive statements are executed to fixed point right after each non-reactive statement and one reactive statement may possibly trigger execution of other reactive statements. The reader should keep in mind that transient sharing is really a shorthand notation for a set of reactive statements which define a communication protocol. Particular care must be exercised in the case when transitive transient sharing takes place, due to the possibility of circular reactions which may lead the system to non-termination.

In general, the Mobile UNITY proof logic copes with these problems by considering the set of reactive statements as a separate program, whose termination has to be proven in order to guarantee correct sharing. However, for the purposes of this work this issue does not arise. Due to the characteristics of mobile code, transient sharing is always pairwise and never leads to a situation when two or more

variables which are shared being assigned values explicitly in the same non reactive statement. As indicated earlier, in this paper we will always use only the high level abstraction provided by transient shared variables without appealing directly to reactive statements. The reactive statements that code this kind of sharing are well-formed and guarantee termination. Mobile UNITY also allows for global constraints on the scheduling of statements, called **inhibit** clauses. Logically, these kinds of statements serve to strengthen guards and express scheduling constraints that may not be stated using local state information alone. 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.

We are ready now to return to the distributed simulation problem. In the following solutions the GVT estimate 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 at initialization time. We will relax this assumption in Section 6.4, where we describe a solution based on the code on demand paradigm that involves the computation of a dynamically changing simulation mode.

6.2 Mobile Agent

In the *Mobile Agent (MA)* paradigm, the client needs to access some of the resources that are located on the server and it owns the necessary know-how to use such resources. 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 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 *Server* 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, as shown in Figure 8.

The *Server* 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. The *Server* is also responsible for computing a new GVT estimate. However, the update of the global state and the computation of the GVT can happen in any order, because the *Server* cannot depart until both the values are up-to-date. The actions performed by the components above are coordinated by the transient variable shar-

```
System DSMobileAgent
    Program P(i) at λ
        declare
            t, z : integer [] T : integer ∪ {⊥}
        initially
            t = 0 [] T = ⊥ [] λ = Location(i)
        assign
            t, T := f_i(t, T, z), ⊥      if def(T)
    end
    Program Server at λ
        declare
            t, T : integer ∪ {⊥} [] τ : array of integer [] pos : clientAddress
        initially
            t, T = 0, 0 [] ⟨|| j :: τ(j) = 0⟩ [] λ = Location(pos)
        assign
            τ(pos) := t
          [] T := ⟨min k :: τ(k)⟩
          [] λ, pos := Location(pos + 1 mod N), pos + 1 mod N       if t = τ(pos) ∧
                                                                       T = ⟨min k :: τ(k)⟩
    end
    Components
            ⟨[] i :: P(i)⟩ [] Server
    Interactions
            P(i).T ← Server.T        when P(i).λ = Server.λ
                                     engage Server.T
          [] Server.t ← P(i).t       when P(i).λ = Server.λ
                                     engage P(i).t
end
```

Fig. 7. Mobile Agent solution for the distributed simulation problem.

ing 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 *Server*, as long as the two components are co-located:

$$
\begin{array}{ll}
P(i).T \leftarrow Server.T & \textbf{when } P(i).\lambda = Server.\lambda \\
 & \textbf{engage } Server.T \\
Server.t \leftarrow P(i).t & \textbf{when } P(i).\lambda = Server.\lambda \\
 & \textbf{engage } P(i).t
\end{array}
$$

The last statement in *Server*, which modifies explicitly the location λ of the component, exercises the **engage** clause of the second interaction. Upon departure, absence of a **disengage** clause guarantees that both the client and the *Server* retain their value for the timer and the GVT estimate. Upon arrival at the next location, the **engage** clause specified in the second read-only shared variable definition guarantees that *as soon as* the two components become co-located, they share the same value for the local timer. The **engage** clause in the first interaction guarantees that the GVT value computed by *Server* on the previous client is communicated to *P(i)*. In general, this value is to be changed by the recomputation of the minimum of all timers, which takes into account the current value of the local timer in *P(i)*. Nevertheless, if this timer has not changed since the last visit, the corresponding value in τ remains unchanged and the GVT computed on the previous client—and already communicated to *P(i)*—is still valid and does not need

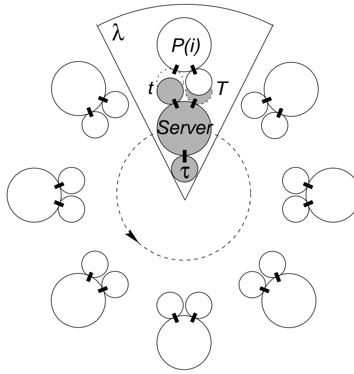

Fig. 8. In the Mobile Agent solution, the *Server* is able to change its location in order to visit each client in a round-robin fashion.

to be recomputed. Hence, in this case the third statement in *Server* is enabled, which allows *Server* to depart immediately without waiting for execution of the two other statements. As an aside, note that disengagement of values, movement, and subsequent re-engagement form a single, atomic action. This captures the fact that the agent cannot perform any action while travelling across a communication link.

6.3 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 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 [Stamos and Gifford 1990] system, which extends the remote procedure call with one additional parameter containing the code to execute on the server. Among recent mobile code languages, the paradigm is supported directly by most of the weakly mobile languages, with the notable exception of Java.

In the solution shown in Figure 9, the client components *P(i)* behave similarly to the ones in the *DSClientServer* system. Besides computing a new local timer, clients can also request the execution of a service that computes the new GVT estimate, 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

```
System DSRemoteEvaluation
    Program P(i) at λ
        declare
            t, z : integer [] T : integer ∪ {⊥} [] RQ : request ∪ {⊥}
        initially
            t = 0 [] T = ⊥ [] λ = Location(i) [] RQ = ⊥
        assign
          [] t, T := f_i(t, T, z), ⊥                   if def(T)
          [] RQ := ⟨SERVER,REV, MINSERV, ⊥⟩             if ¬def(RQ) ∧ ¬def(T)
          [] T, RQ := RQ ↑ 4, ⊥                        if RQ ↑ 1 = i
    end
    Program Min(i) at λ
        declare
            t : integer [] T : integer [] τ : (array of integer) ∪ {⊥} [] q : request ∪ {⊥}
        initially
            t, T = 0, 0 [] τ = ⊥ [] q = ⊥ [] λ = Location(i)
        assign
            τ(i) := t                  if def(τ)
          [] T := ⟨min k :: τ(k)⟩       if def(τ)
          [] q := ⟨i, ⊥, ⊥, T⟩          if τ(i) = t ∧ T = ⟨min k :: τ(k)⟩
    end
    Program Server at λ
        declare
            τ : array of integer [] q : array of (request ∪ {⊥}) []
        initially
            ⟨|| j :: τ(j) = 0⟩ [] ⟨|| j :: q(j) = ⊥⟩ [] λ = Location(SERVER)
    end
    Components
            ⟨[] i :: P(i)⟩ [] Server [] ⟨[] i :: Min(i)⟩
    Interactions
            Min(i).t ← P(i).t              when Min(i).λ = P(i).λ
                                           engage P(i).t
          [] Min(i).λ := Server.λ          when Min(i).λ = P(i).λ ∧
                                                serviceRequest(REV, MINSERV, i)
          [] Min(i).τ ≈ Server.τ           when Min(i).λ = Server.λ
                                           engage Server.τ
                                           disengage ⊥, Server.τ
          [] Min(i).q ≈ Server.q(i)        when Min(i).λ = Server.λ
                                           engage ⊥
          [] P(i).RQ, Server.q(i)[, Min(i).λ] := Server.q(i), ⊥[, P(i).λ]     when serviceReady(i)
end
```

Fig. 9. Remote Evaluation solution for the distributed simulation problem.

corresponding 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

$$RQ := \langle \textsc{server}, \textsc{rev}, \textsc{minServ}, \bot \rangle \quad \textbf{if } \neg\mathsf{def}(RQ) \wedge \neg\mathsf{def}(T),$$

specifying that the code for the service named MINSERV must be sent to the *Server*'s location, where its execution exploits bindings with *Server*'s resources. In contrast

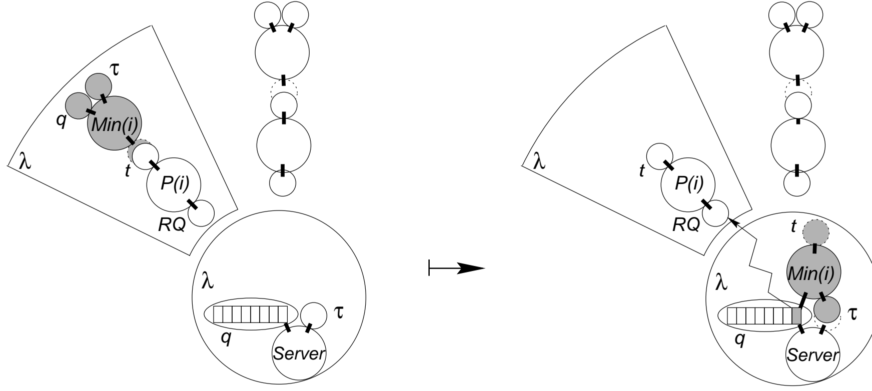

Fig. 10. A graphical representation of the REV solution.

with the CS solution, the request message is not delivered to the message buffer of the *Server*. Its presence within the client buffer enables subsequent code migration. The code for service minServ is described by the program *Min(i)* and is structured in three steps: update of the global state maintaned in $Server.\tau$ with the timer value of the corresponding client *P(i)*, computation of the new GVT, and communication of the results to the client. All these steps can take place only when *Min(i)* and *Server* are co-located. In this situation, the statement

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

in the **Interactions** section specifies a transient sharing between the representation of the global state τ owned by *Min(i)* and *Server*. Upon departure, the **disengage** clause voids the value of τ in *Min(i)*, thus preventing execution of any statement within this component while it is not at the *Server*'s location. Upon a subsequent arrival, the τ values owned by the two components are reconciled by the **engage** clause, which assigns to $Min(i).\tau$ the up-to-date τ value kept on *Server*. Updating a client's timer value stored in τ is made possible by the interaction

$$\begin{array}{ll} Min(i).t \leftarrow P(i).t & \textbf{when } Min(i).\lambda = P(i).\lambda \\ & \textbf{engage } P(i).t \end{array}$$

which 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)* from *P(i)*'s location, *Min(i)* retains the current value[3] of t, which is used at the *Server* location to update of the global state. The actual code migration happens

[3]Mobile code languages implement parameter passing either explicitly by referring to input parameters and code using RPC-like primitives or implicitly by attaching to the procedure to be executed remotely the portion of data space needed for remote computation [Cugola et al. 1997]. We chose the second alternative, in order to illustrate how to dynamically establish and remove bindings among variables in Mobile Unity.

when

$$\begin{aligned} Min(i).\lambda \;:=\; & Server.\lambda \\ & \textbf{when } \mathsf{serviceRequest}(\textsc{rev}, \textsc{minServ}, i) \wedge Min(i).\lambda = P(i).\lambda \end{aligned}$$

is eventually executed after a client has sent a message request.

At this point, it is worth noting how, in contrast with the CS solution, the REV request does not contain information that needs to be passed to the *Server*. However, in both solutions the action of putting a message in the request buffer *RQ* can be regarded as modeling the invocation of a communication primitive, which enables actions in the underlying run-time support. In the CS solution, these actions are represented by the transfer of the message to the *Server*; in the REV solution, the actions encompass migration of the *Min(i)* component to the *Server*'s location. Hence, in both solutions the client component is given a uniform interface (the request buffer) to the rest of the system, whose details are handled within the **Interactions** sections—thus modeling the run-time support for communication provided by, say, implementations of RPC and of the REV [Stamos and Gifford 1990] system. Analogous considerations hold for the output buffer of *Server*. In the REV solution,

$$\begin{aligned} Min(i).q \approx Server.q(i) \qquad & \textbf{when } Min(i).\lambda = Server.\lambda \\ & \textbf{engage } \bot \end{aligned}$$

models the fact that *Min(i)* is given access to the communication facilities co-located on the *Server*; the last interaction, in turn, models the actual transfer of information provided by the underlying run-time support. Note that in our REV solution the *Server* does not provide any service, except for offering a name space in which each *Min(i)* can have access to the global state by sharing τ and to the communication facilities by sharing the message buffer q.

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 as in the CS solution, except for the assignment

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

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 available in Mobile UNITY. This issue will be revisited in Section 8.

6.4 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 above, 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. In both cases, the programmer can determine the actions to be performed by the runtime support whenever a name cannot be resolved locally. In particular, these actions may encompass retrieval of the corresponding code from a remote site.

We present the solution exploiting the COD paradigm by enhancing the CS solution shown earlier. In the system shown in Figure 11, 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, e.g., the simulated electronic component has reached the point of breakdown, and 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 *static* becomes *false*, the client is enabled to issue a request

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

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. Although the latter situation can be modeled, we choose this assumption for the sake of simplicity. 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 the message exchanges needed to compute the GVT estimate as in the CS paradigm. In turn, the statement

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

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 disables the

```
System DSCodeOnDemand
    Program P(i) at λ
        declare
            t, z : integer [] T : integer ∪ {⊥} [] static : boolean ∪ {⊥} [] RQ : request ∪ {⊥}
        initially
            t = 0 [] T = ⊥ [] λ = Location(i) [] static = true [] RQ = ⊥
        assign
            [] static := false                               if def(static)
            [] t, T := f_i(t, T, z), ⊥                       if def(T)
            [] RQ := ⟨SERVER,CS, MINSERV, t⟩                 if ¬def(RQ) ∧ ¬def(T)
            [] T, RQ := RQ ↑ 4, ⊥                            if RQ ↑ 1 = i
            [] RQ := ⟨SERVER,COD, DYNMODE, ⊥⟩                if ¬def(RQ) ∧ static = false
    end
    Program Dyn(i) at λ
        declare
            z : integer [] T : integer ∪ {⊥}
        initially
            λ = Location(SERVER)
        assign
            z := d(T)      if def(T)
    end
    Program Server at λ
        declare
            T : integer ∪ {⊥} [] τ : array of integer [] q : array of (request ∪ {⊥})
        initially
            T = ⊥ [] ⟨|| j :: τ(j) = 0⟩ [] ⟨|| j :: q(j) = ⊥⟩ [] λ = Location(SERVER)
        assign
            ⟨[] j :: τ(j), q(j) ↑ 2 := q(j) ↑ 4, WAIT⟩       if q(j) ↑ 1 = SERVER ∧ q(j) ↑ 2 ≠ WAIT
            [] T := ⟨min k :: τ(k)⟩                          if ⟨∃ j :: q(j) ↑ 2 = WAIT⟩ ∧ ¬def(T)
            [] ⟨|| j :: q(j), T := ⟨j, ⊥, ⊥, T⟩, ⊥           if def(T) ∧ q(j) ↑ 2 = WAIT⟩
    end
    Components
            ⟨[] i :: P(i)⟩ [] Server [] ⟨[] i :: Dyn(i)⟩
    Interactions
            Dyn.λ, P(i).static := P(i).λ, ⊥                  when Dyn(i).λ = Server.λ ∧
                                                                  serviceRequest(COD, DYNMODE, i)
            [] P(i).z ← Dyn(i).z                             when P(i).λ = Dyn(i).λ
                                                             engage P(i).z
            [] Dyn(i).T ← P(i).T                             when P(i).λ = Dyn(i).λ
                                                             engage P(i).T
            [] Server.q(i) := P(i).RQ                        when ¬def(Server.q(i)) ∧
                                                                  serviceRequest(CS, MINSERV, i)
            [] P(i).RQ, Server.q(i) := Server.q(i), ⊥        when serviceReady(i)
end
```

Fig. 11. Code On Demand solution for the distributed simulation problem.

statement issuing the request. Finally,

$$
\begin{array}{ll}
P(i).z \leftarrow Dyn(i).z & \textbf{when } P(i).\lambda = Dyn(i).\lambda \\
 & \textbf{engage } P(i).z \\
Dyn(i).T \leftarrow P(i).T & \textbf{when } P(i).\lambda = Dyn(i).\lambda \\
 & \textbf{engage } P(i).T
\end{array}
$$

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

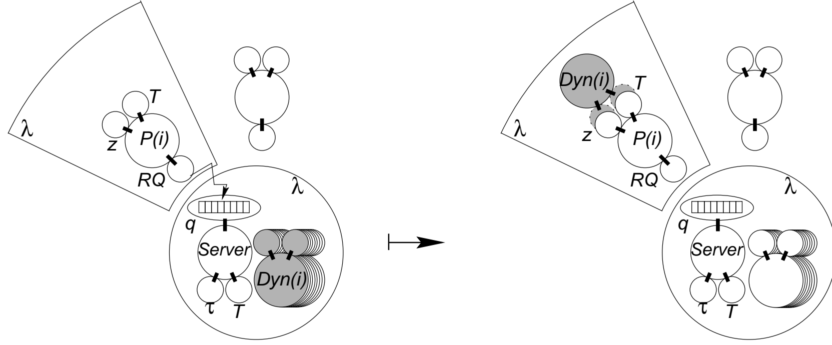

Fig. 12. A graphical representation of the Code On Demand solution.

values in *P(i)*. A graphical representation of this system is shown in Figure 12. 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 discussed further in Section 8.

7. VERIFICATION

One of the advantages of expressing mobile code solutions in Mobile UNITY is the availability of its proof logic. This section provides an introduction to the verification of mobile code programs. The goal is not so much to carry out precise formal verification of the solutions put forth in this paper, but to expose the basic strategy one would have to follow whether the verification is carried out informally (i.e., at the proof outline level as in this section) or in more detail (i.e., as shown in the Appendix).

Our presentation will attempt to explore the verification process in a manner that will be accessible even to the casual reader having only a limited background in verification. We will start by first exploring the UNITY proof system in the context of the earlier centralized solution. After that we move on to Mobile UNITY. The client-server solution will be used as an illustration and the point will be made that, ultimately, the Mobile UNITY notation allows one to reduce proofs to the basic UNITY verification style. The manner in which the verification of the mobile code solutions is performed is expanded upon in the remainder.

7.1 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. Progress properties are captured by a predicate relation called **leads-to** (written $\mapsto$). At this point we rely on the earlier centralized solution for illustration purposes (Figure 2). In this context, for instance,

$$t(i) = \langle \min\ j :: t(j) \rangle = \alpha \ \mapsto\ t(i) > \alpha$$

states that from a state in which the local timer of a process is the minimum of all the local timers, the program eventually enters a state where that local timer is

advanced. (By convention, all free variables are assumed to be universally quantified.)

A **leads-to** property is most often derived by relying on the transitivity of this relation or on a primitive form of **leads-to** called **ensures** ($\mathcal{T}_1$). In the UNITY program, for instance, one can show that

$$t(i) = \langle \min\ j :: t(j) \rangle = \alpha \ \textbf{ensures}\ T = \alpha \wedge t(i) = \alpha$$

This can be verified directly from the program text by observing that, for as long as the right hand side (RHS) of the **ensures** relation is false:

(1) All program statements either leave the left hand side (LHS) of the **ensures** unchanged because they do not update T or they actually update T with the correct value ($\mathcal{T}_3$). (Technically this expresses a safety property called **unless** in UNITY.)

(2) There is a statement which, if selected, updates T to the correct value in a single atomic step ($\mathcal{T}_2$). Fairness guarantees that this statement is eventually selected for execution.

To summarize, the three UNITY inference rules we introduced so far are:

$$\frac{p \ \textbf{ensures}\ q}{p \mapsto q}, \qquad (\mathcal{T}_1)$$

$$\frac{p \ \textbf{unless}\ q,\ \langle \exists\ s :: \{p\}\ s\ \{q\} \rangle}{p \ \textbf{ensures}\ q}, \qquad (\mathcal{T}_2)$$

and

$$\frac{\langle \forall\ s :: \{p \ \wedge \neg q\}\ s\ \{p \vee q\} \rangle}{p \ \textbf{unless}\ q}, \qquad (\mathcal{T}_3)$$

where $\{p\}\ s\ \{q\}$ is the Hoare triple which can be proven using the standard assignment axiom and pure logic. The **unless** and **leads-to** relations can be combined in an **until** relation,

$$p \ \textbf{until}\ q \equiv (p \ \textbf{unless}\ q) \wedge (p \mapsto q), \qquad (\mathcal{T}_4)$$

meaning that p holds as long as q does not, and that q holds eventually[4]. The correctness criteria for the distributed simulation problem are conveniently expressed as two **until** properties:

PROPERTY 1 $t(i) = \langle \min\ j :: t(j) \rangle = \alpha$ **until** $t(i) > \alpha$.
A local timer increases only at the point when its value equals the GVT.

PROPERTY 2 $t(i) = \alpha$ **until** $t(i) = \langle \min\ j :: t(j) \rangle = \alpha$.
The GVT eventually catches up with with the local timer, i.e., the simulation makes continuous progress.

---

[4]It can be noted how **ensures** and **until** look alike, but while **ensures** requires a specific statement to establish the RHS in one step, **until** allows for multiple steps, each proven by **leads-to**.

The UNITY logic includes many other useful predicate relations and inference rules. We prefer to introduce them only when they are really needed. Some of them are needed immediately in verifying Property 2. First, is the fact that implication is a special case of a **leads-to**:

$$\frac{p \Rightarrow q}{p \;\mapsto\; q} \qquad (\mathcal{T}_5)$$

Second is the definition of an invariant property. A predicate p is called an invariant (written **inv.**) if it holds in the initial state and forever throughout the execution of the program:

$$\frac{\mathit{INIT} \Rightarrow p,\; p \;\textbf{unless}\; \mathit{false}}{\textbf{inv.}\; p} \qquad (\mathcal{T}_6)$$

where INIT is a predicate which characterizes the set of acceptable initial states. As an example, we can show that

$$\textbf{inv.}\; t(i) \geq T$$

for every i. This is true initially, since both $t(i)$ and T are initialized to zero. Moreover, the execution of any statement in a state in which this property holds preserves the property—local timers can only advance past T and any updates to T guarantee that it acquires a value that is the minimum across all local timers. In addition, we apply the following *induction principle* for **leads-to**. If W is a set well-founded under some relation $\prec$, and M is a *metric* function mapping program states to W, then

$$\frac{\langle \forall\; m : m \in W :: p \wedge M = m \;\mapsto\; (p \wedge M \prec m) \vee q \rangle}{p \;\mapsto\; q} \qquad (\mathcal{T}_7)$$

holds. The hypothesis states that from any state where p holds, program execution eventually reaches a state where either q holds or the metric M is decreased and p holds. From the well-foundedness of W, M cannot decrease indefinitely, which allows us to conclude that eventually a state is reached where q holds. Finally, we will use the *disjunction rule* of **leads-to**

$$\text{for any set } W,\; \frac{\langle \forall\; m : m \in W :: p(m) \;\mapsto\; q \rangle}{\langle \exists\; m : m \in W :: p(m) \rangle \;\mapsto\; q} \qquad (\mathcal{T}_8)$$

which has the simpler consequence

$$\frac{p \;\mapsto\; q,\; p' \;\mapsto\; q}{p \vee p' \;\mapsto\; q}$$

and allows to prove **leads-to** properties by case analysis.

Armed with this knowledge, we can prove the properties stated above. Proof of each **until** relation involves proving its **unless** and **leads-to** parts.

PROOF OF PROPERTY 1. We prove separately:

(1) $t(i) = \langle \min\; j :: t(j) \rangle = \alpha$ **unless** $t(i) > \alpha$ can be proven by observing that, due to the definition of f and g, the asynchronous statements updating $t(i)$ always increase the timer value, while the other statements always leave $t(i)$ and the minimum unchanged. Hence, it is proven by application of $(\mathcal{T}_3)$.

(2) $t(i) = \langle \min\ j :: t(j) \rangle = \alpha \ \mapsto\ t(i) > \alpha$ can be split by $(\mathcal{T}_1)$ and transitivity of **leads-to** into

$$t(i) = \langle \min\ j :: t(j) \rangle = \alpha \ \textbf{ensures}\ T = \alpha \wedge t(i) = \alpha, \text{ and}$$
$$T = \alpha \wedge t(i) = \alpha \ \textbf{ensures}\ t(i) > \alpha.$$

The first obligation was proven earlier. The second one is proven by arguments similar to the ones used to prove the **unless** above and observing that there exists one statement that increases $t(i)$, while fairness guarantees that eventually it is selected.

Proof of Property 2. Again, we prove separately:

(1) $t(i) = \alpha$ **unless** $t(i) = \langle \min\ j :: t(j) \rangle = \alpha$ is proven observing that, after the execution of each statement in the program, $t(i)$ can either be the minimum or not, in which case its value is not modified.

(2) $t(i) = \alpha \ \mapsto\ t(i) = \langle \min\ j :: t(j) \rangle = \alpha$ is proven by considering two cases, according to the value of $t(i)$:

(a) $t(i) = \alpha \wedge t(i) = \langle \min\ j :: t(j) \rangle \ \mapsto\ t(i) = \langle \min\ j :: t(j) \rangle = \alpha$, which holds by virtue of the fact that

$$t(i) = \alpha \wedge t(i) = \langle \min\ j :: t(j) \rangle \Rightarrow t(i) = \langle \min\ j :: t(j) \rangle = \alpha,$$

followed by application of $(\mathcal{T}_5)$.

(b) $t(i) = \alpha \wedge t(i) \neq \langle \min\ j :: t(j) \rangle \ \mapsto\ t(i) = \langle \min\ j :: t(j) \rangle = \alpha$. We use the induction principle of **leads-to** stated in $(\mathcal{T}_7)$, where W is the set of natural numbers $\mathbb{N}$, $\prec$ is the $<$ relation, and M is the number of local timers that are behind $t(i)$, i.e., $M = \langle \sharp\ j :: t(j) < \alpha \rangle$ with $M \geq 0$ by construction[5]. Thus, we prove that

$$\begin{aligned} & t(i) = \alpha \wedge t(i) \neq \langle \min\ j :: t(j) \rangle \wedge M = m \\ \mapsto\ & (t(i) = \alpha \wedge t(i) \neq \langle \min\ j :: t(j) \rangle \wedge M < m) \vee \\ & t(i) = \langle \min\ j :: t(j) \rangle = \alpha. \end{aligned}$$

We observe that, since $t(i)$ is not the minimum, some other timer must be the minimum and, because of Property 1 and of the fact that local timers range over natural numbers, it eventually increases, thus eventually decreasing the value of M in one or more steps. Furthermore, $t(i)$ cannot increase unless it is the minimum, as proven earlier. Eventually no timer is behind $t(i)$, i.e., $t(i)$ holds the minimum value of all timers.

Application of the disjunction rule $(\mathcal{T}_8)$ completes the proof.

7.2 Mobile Unity Proof Logic

In this section we introduce the Mobile Unity proof logic. In doing this, we also shift the context to the Mobile Unity solutions presented in Section 6. The distributed solutions must continue to maintain Property 1 and 2 stated for the centralized one. The proof can make use of the standard Unity proof logic as long as

[5]The operator $\sharp$ used in the three part notation is used to count the number of elements in the domain of j for which the provided expression holds.

we use the fully qualified names at every step; we simply have a larger number of statements that must be considered. We first rewrite Property 1 and 2 stated in Section 2 with the appropriate naming conventions observed:

PROPERTY 1 $P(i).t = \langle \min\ j :: P(j).t \rangle = \alpha$ **until** $P(i).t > \alpha$
A client P(i) increases its timer only when its value equals the GVT.

PROPERTY 2 $P(i).t = \alpha$ **until** $P(i).t = \langle \min\ j :: P(j).t \rangle = \alpha$
The GVT will catch up eventually with the timer in P(i).

The properties and the structure of their proof outline are the same across all the distributed solutions presented in this paper. Nevertheless, each proof makes use of lemmas and invariants which are proven differently in each solution. Next, we present the general proof structure. In following subsections, we discuss only the proof specifics for each distributed solution, starting with the client-server and continuing with the mobile code solutions. We will not provide any verification of the COD solution, since the computation of z does not affect the correctness of the properties stated in Section 7.2.

7.2.1 *General Proof Structure.* The properties stated earlier can be proven for all the distributed solutions presented in this paper by using the following proof structure.

PROOF OF PROPERTY 1. We prove separately the two parts of the **until** relation:

(1) $P(i).t = \langle \min\ j :: P(j).t \rangle = \alpha$ **unless** $P(i).t > \alpha$ can be proven from the program text by observing that, for any of the solutions, every statement in the system leaves unchanged the LHS except possibly for the first statement of *P(i)*. This statement, according to the definition of f and strict monotonicity of g, may either leave unchanged the LHS or establish the RHS.

(2) $P(i).t = \langle \min\ j :: P(j).t \rangle = \alpha \mapsto P(i).t > \alpha$ can be split by transitivity of **leads-to** into two separate proof obligations. First, we prove that whenever a client's[6] timer corresponds to the GVT, the GVT estimate eventually catches up with the timer within the client, i.e.,

$$P(i).t = \langle \min\ j :: P(i).t \rangle = \alpha \mapsto P(i).t = P(i).T = \alpha$$

This is proven by applying transitivity to the following two lemmas:

$$P(k).t = \langle \min\ l :: P(l).t \rangle = \beta \mapsto P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \qquad (\mathcal{L}_1)$$

which asserts that if a given GVT value is reached, the timers whose values match the GVT are eventually mirrored on the *Server*, and lemma

$$P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \mapsto P(k).t = P(k).T = \beta \qquad (\mathcal{L}_2)$$

[6]In the following, we use the term "client" to encompass both the client in the client-server or remote evaluation paradigm and the agent in the mobile agent paradigm.

which asserts that the minimum of all the values mirrored on the *Server* eventually corresponds to the GVT estimate communicated back to the client.

Then, we prove that, in the situation when the GVT estimate is equal to the timer within a client, the client eventually makes progress by increasing its timer, which is formally captured by the following lemma:

$$P(k).t = P(k).T = \beta \textbf{ ensures } P(k).t > \beta \qquad (\mathcal{L}_3)$$

Lemma ($\mathcal{L}_1$) is proven the same way across all the solutions, so it is proven in the remainder of this section. Lemmas ($\mathcal{L}_2$) and ($\mathcal{L}_3$) will be proven for each solution in Section 7.2.2-7.2.4, and Appendix A-C, respectively.

Proof of Property 2. We prove separately the two parts of the **until** relation:

(1) $P(i).t = \alpha$ **unless** $P(i).t = \langle \min\ j :: P(j).t \rangle = \alpha$. We observe that

$$P(i).t = \alpha \textbf{ unless } P(i).t = P(i).T = \alpha$$

holds because, in all the solutions, the only statement affecting $P(i).t$ is the first statement in *P(i)*. After execution of such a statement it might be the case that the RHS holds, according to the invariant

$$\textbf{inv. } \mathsf{def}(P(k).T) \Rightarrow P(k).T \leq P(k).t, \qquad (\mathcal{I}_1)$$

which asserts that whenever the GVT estimate is defined within a client, its value is always at most equal to the corresponding local timer value. Furthermore, the invariant

$$\textbf{inv. } P(k).t = P(k).T = \beta \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta, \qquad (\mathcal{I}_2)$$

states that whenever the values of the local timer and the GVT estimate are equal in a client, their values equal the GVT value as well. Hence, according to the *consequence weakening theorem* [Chandy and Misra 1988]

$$\frac{p \textbf{ unless } q, q \Rightarrow r}{p \textbf{ unless } r} \qquad (\mathcal{T}_9)$$

the conclusion holds. Invariant ($\mathcal{I}_1$) is proven in the remainder of this section. Invariant ($\mathcal{I}_2$), as well as others introduced in this section, are proven differently in each solution. The corresponding proof outlines can be found in the Appendix.

(2) $P(i).t = \alpha \ \mapsto\ P(i).t = \langle \min\ j :: P(j).t \rangle = \alpha$. To demonstrate that the system eventually reaches a state where the GVT is equal to the timer value α we need to demonstrate that the difference between α and the GVT decreases continuously. To this end, we need to show that the number of clients whose timer is greater than a given GVT but less than α eventually decreases to zero. This is a consequence of the continuous increase of the GVT and of the fact that the local timer holds its value up to the point where the GVT catches up with it. Formally, this proof requires the application of the induction principle of **leads-to**. The decrease in the distance δ between α and the current GVT is captured by

$$\begin{array}{ll} & P(i).t = \alpha \wedge 0 < (\alpha - \langle \min\ j :: P(j).t \rangle) = \delta \\ \mapsto & (P(i).t = \alpha \wedge 0 < (\alpha - \langle \min\ j :: P(j).t \rangle) < \delta) \vee \alpha = \langle \min\ j :: P(j).t \rangle. \end{array}$$

The equation above is proven by demonstrating that the number ν of clients at a given current GVT $\bar{\alpha}$ is decreasing, and that the GVT eventually increases, that is

$$\begin{aligned} & \langle \min\ j :: P(j).t \rangle = \bar{\alpha} \wedge 0 < \langle \sharp\ j :: P(j).t = \bar{\alpha} \rangle = \nu \\ \mapsto\ & (\langle \min\ j :: P(j).t \rangle = \bar{\alpha} \wedge 0 < \langle \sharp\ j :: P(j).t = \bar{\alpha} \rangle < \nu) \vee \\ & \langle \min\ j :: P(j).t \rangle > \bar{\alpha}, \end{aligned}$$

which follows from Property 1 proven earlier.

PROOF OF ($\mathcal{L}_1$). We do not consider further the case when $P(k).t = \langle \min\ l :: \mathit{Server}.\tau(l) \rangle = \beta$ because it can be demonstrated trivially by application of the implication theorem ($\mathcal{T}_5$). Then, we introduce the invariant

$$\textbf{inv.}\ \mathit{Server}.\tau(k) \leq P(k).t, \qquad (\mathcal{I}_3)$$

which asserts that the timer value mirrored on the server is always at most equal to the original value within the client. This invariant enables us to consider only the case $\langle \min\ l :: \mathit{Server}.\tau(l) \rangle < \beta$. Hence, we prove

$$\mathsf{oldServerState}(k, \beta)\ \mapsto\ P(k).t = \langle \min\ l :: \mathit{Server}.\tau(l) \rangle = \beta,$$

where $\mathsf{oldServerState}(k, \beta) \equiv P(k).t = \langle \min\ l :: P(l).t \rangle = \beta \wedge \langle \min\ l :: \mathit{Server}.\tau(l) \rangle < \beta$. We observe that this obligation can be proven by showing that the number of clients at GVT for which the timer value mirrored on the *Server* is not up-to-date is decreasing. Formally, this corresponds to application of the induction principle of **leads-to**:

$$\begin{aligned} & \mathsf{oldServerState}(k, \beta) \wedge 0 < \langle \sharp\ l :: \mathit{Server}.\tau(l) < \beta \rangle = \nu \\ \mapsto\ & (\mathsf{oldServerState}(k, \beta) \wedge 0 < \langle \sharp\ l :: \mathit{Server}.\tau(l) < \beta \rangle < \nu) \vee \\ & P(k).t = \langle \min\ l :: \mathit{Server}.\tau(l) \rangle = \beta \end{aligned}$$

which follows from the lemma

$$P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta\ \textbf{until}\ P(k).t = \mathit{Server}.\tau(k) = \beta. \qquad (\mathcal{L}_4)$$

This lemma, whose proof will be given case by case in the following sections, states that the value of a local timer is eventually copied on the *Server*, which guarantees continuous decrease of the metric used for induction.

PROOF OF ($\mathcal{I}_1$). The invariant holds from ($\mathcal{I}_3$) and invariant

$$\textbf{inv.}\ \mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min\ l :: \mathit{Server}.\tau(l) \rangle, \qquad (\mathcal{I}_4)$$

which asserts that whenever the GVT estimate is defined within a client, it is at most equal to the minimum among the timer values mirrored on the server.

7.2.2 *Proof of Client-Server Solution.* The proof obligations associated with the client-server solution appear in Table 1. All the properties, and other supportive assertions, are proven in the Appendix. The only exception is Lemma ($\mathcal{L}_2$) and one of the two properties on which it relays. They are considered below.

PROOF OF ($\mathcal{L}_2$). We do not consider further the case $P(k).T = \beta$ after application of ($\mathcal{T}_5$). We introduce the following definitions in order to improve readability:

$P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \ \mapsto\ P(k).t = P(k).T = \beta$	$(\mathcal{L}_2)$
$P(k).t = P(k).T = \beta$ **ensures** $P(k).t > \beta$	$(\mathcal{L}_3)$
$P(k).t = \beta \wedge Server.\tau(k) < \beta$ **until** $P(k).t = Server.\tau(k) = \beta$	$(\mathcal{L}_4)$
inv. $\mathsf{def}(Server.T) \Rightarrow Server.T \leq \langle \min\ l :: Server.\tau(l) \rangle$	$(\textsc{cs}_1)$
inv. $\mathsf{def}(Server.q(k)) \Rightarrow\ Server.q(k) \uparrow 2 = \textsc{cs} \vee$ $Server.q(k) \uparrow 2 = \textsc{wait} \vee\ \neg\mathsf{def}(Server.q(k) \uparrow 2)$	$(\textsc{cs}_2)$
inv. $P(k).t = P(k).T = \beta \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta$	$(\mathcal{I}_2)$
inv. $Server.\tau(k) \leq P(k).t$	$(\mathcal{I}_3)$
inv. $\mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min\ l :: Server.\tau(l) \rangle$	$(\mathcal{I}_4)$
inv. $\mathsf{def}(P(k).T) \Rightarrow \neg\mathsf{def}(P(k).RQ)$	$(\textsc{cs}_3)$
inv. $Server.q(k) \uparrow 1 = \textsc{server} \Rightarrow Server.q(k) \uparrow 4 = P(k).t$	$(\textsc{cs}_4)$
$\langle \min\ l :: Server.\tau(l) \rangle = \beta$ **unless** $\langle \min\ l :: Server.\tau(l) \rangle > \beta$	$(\textsc{cs}_5)$

Table 1. Lemmas and invariants needed for verification of the Client-Server solution.

$$\begin{aligned}\mathsf{minTimer}(k,\beta) &\equiv P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \\ \mathsf{oldClientT}(k,\beta) &\equiv P(k).T < \beta \vee \neg\mathsf{def}(P(k).T) \\ \neg\mathsf{oldServerT}(\beta) &\equiv Server.T = \beta \vee \neg\mathsf{def}(Server.T)\end{aligned}$$

that is, respectively, *P(k)* owns the minimum among the timers in *Server*, the GVT is out-of-date in client *P(k)*, and the GVT is not out-of-date in *Server*. Then, invariant $(\mathcal{I}_1)$ allows us to reformulate the goal of our proof as

$$\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \ \mapsto\ P(k).t = P(k).T = \beta. \qquad (\textsc{cs}_6)$$

i.e., if a client *P(k)* owns the timer corresponding to the minimum timer value stored on *Server* but the corresponding GVT estimate $P(k).T$ is behind the timer, the GVT estimate eventually catches up with the timer in *P(k)*. In order for $\mathsf{minTimer}(k,\beta)$ to hold, the minimum β over $Server.\tau$ must have been established by a request message from *P(k)*, or from some other client with the same timer value, which updates the value mirrored on *Server*. Due to the interleaving semantics, however, it could happen that, at the moment when this minimum is established in $Server.\tau$, a value referring to a previous value $\bar{\beta} = \langle \min\ l :: Server.\tau(l) \rangle \neq \beta$ has been already assigned to the GVT estimate $Server.T$. However, invariant $(\textsc{cs}_1)$ guarantees that $Server.T$ can never be assigned a value greater than β. Thus, we can split $(\textsc{cs}_6)$ into the following two cases

$$\begin{aligned}&\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \qquad (\textsc{cs}_7)\\ \mapsto\ &P(k).t = P(k).T = \beta, \text{ and}\\ &\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \qquad (\textsc{cs}_8)\\ \mapsto\ &P(k).t = P(k).T = \beta,\end{aligned}$$

which, by application of disjunction, prove $(\textsc{cs}_6)$ and then $(\mathcal{L}_2)$. Next, we turn our attention to the proof of $(\textsc{cs}_7)$.

PROOF OF (CS_7). We must show that if the *Server* has already established the correct GVT estimate or it is about to (i.e., its GVT estimate is currently void), while the client still holds an out-of-date GVT estimate, the new one is eventually communicated to the client. We accomplish this by reasoning about the communication taking place between *P(k)* and the *Server*. The only way the client can learn about the new GVT estimate is by receiving a reply to a formerly sent message request. In the worst case, such a request has not been issued yet, i.e., the *Server* does not have a message from *P(k)* in its queue. We show that all the other intermediate states are encompassed in the analysis of communication taking place in this worst case. Thus, we prove that, in the worst case above, a message request containing the timer value is eventually generated by the client, that is,

$$\begin{array}{ll} & \mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \\ \mapsto & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle. \end{array} \qquad (\text{CS}_9)$$

It can be proven directly from the text that

$$\begin{array}{ll} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \\ \textbf{ensures} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \\ & Server.q(k) = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \end{array} \qquad (\text{CS}_{10})$$

that is, such a request is eventually delivered to the *Server*. As a consequence, *Server* eventually accepts the request by forcing it into the WAIT state,

$$\begin{array}{ll} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \\ & Server.q(k) = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \\ \textbf{ensures} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge Server.q(k) \uparrow 2 = \text{WAIT}. \end{array} \qquad (\text{CS}_{11})$$

which, again, can be proven directly from the program text. Finally, we prove that the pending request is eventually satisfied and a reply containing the updated value for the GVT is delivered back to *P(k)*, that is,

$$\begin{array}{ll} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge Server.q(k) \uparrow 2 = \text{WAIT} \\ \mapsto & P(k).t = P(k).T = \beta. \end{array} \qquad (\text{CS}_{12})$$

Application of disjunction and transitivity of **leads-to** to (CS_9), (CS_{10}), (CS_{11}), and (CS_{12}) prove (CS_7). Equation (CS_9) is proven by expanding $\mathsf{oldClientT}(k,\beta)$ into the possible values of $P(k).T$ according to ($\mathcal{I}_1$), and applying disjunction and transitivity to the following chain of **ensures**:

$$\begin{array}{ll} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge P(k).T < \beta \\ \textbf{ensures} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \neg\mathsf{def}(P(k).T) \\ \textbf{ensures} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle, \end{array} \qquad (\text{CS}_{13})$$

where the initial state implies that *P(k)* has not issued a request yet, in accordance with invariant ($\textsc{cs}_3$). Similarly, ($\textsc{cs}_{11}$) can be proven by considering the two values of the GVT estimate *Server.T* as defined by $\neg\mathsf{oldServerT}(\beta)$. Finally, disjunction and transitivity applied to the following chain of **ensures**

$$\begin{aligned} & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{def}(Server.T) \wedge Server.q(k) \uparrow 2 = \textsc{wait} \\ \textbf{ensures } & \mathsf{minTimer}(k,\beta) \wedge Server.T = \beta \wedge Server.q(k) \uparrow 2 = \textsc{wait} \\ \textbf{ensures } & \mathsf{minTimer}(k,\beta) \wedge Server.q(k) = \langle k, \bot, \bot, \beta \rangle \\ \textbf{ensures } & \mathsf{minTimer}(k,\beta) \wedge P(k).RQ = \langle k, \bot, \bot, \beta \rangle \\ \textbf{ensures } & P(k).t = P(k).T = \beta, \end{aligned}$$

which can be derived from the program text, proves ($\textsc{cs}_{12}$) and completes the proof, together with invariant ($\textsc{cs}_2$) which constrains the value of $Server.q(k)$ and thus guarantees coverage of all the possible cases in the above obligations.

The reader may notice at this point that the proof of property ($\textsc{cs}_7$) has the look and feel of a standard UNITY proof. This is not in the least accidental. First, throughout this paper (and in other related work) we seek to align the verification process with the UNITY style to the greatest possible extent. Second, even though the **Interactions** section plays a critical role in the verification of the property ($\textsc{cs}_7$), the manner in which it affects the proofs is in no way distinguishable from that of the programs themselves. The **Interactions** section contains statements that transfer data from one location to another asynchronously. Aside from the location of the data, the **Interactions** statements are treated as if they were standard UNITY statements: they are always included in the verification of safety properties and, on occasion, are used to prove some of the **ensures** properties. The latter deal with the data transfer guarantees between the clients and the server.

7.2.3 *Proof of Mobile Agent Solution.* The general proof structure provided in Section 7.2.1 is still valid for the MA solution. However, verification must encompass reasoning about the read-only sharing specified in the **Interactions** section. Table 2 summarizes the lemmas and invariants needed to prove the MA solution. In this section, we provide the proof outlines for the lemmas used in Section 7.2.1, together with some obligations specific of the MA solution. The remaining obligations are proven in Appendix B. Besides the properties described in Section 7.2.1, two additional properties capture the round-robin migration of *Server*. Invariant ($\textsc{ma}_1$) states that *Server* is always co-located with some client, while lemma ($\textsc{ma}_2$) guarantees that *Server* is forced to visit the next node.

PROOF OF ($\mathcal{L}_2$). Our proof will exploit the *progress-safety-progress theorem* (PSP) [Chandy and Misra 1988]

$$\frac{p \;\mapsto\; q,\; r \textbf{ unless } b}{p \wedge r \;\mapsto\; (q \wedge r) \vee b}, \qquad (\mathcal{T}_{10})$$

which allows for combination of a progress and a safety property in proving a progress property. In our case, the progress property is

$$true \;\mapsto\; Server.T = P(k).T = \langle \min\ l :: Server.\tau(l) \rangle, \qquad (\textsc{ma}_8)$$

	$P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta\ \mapsto\ P(k).t = P(k).T = \beta$	($\mathcal{L}_2$)
	$P(k).t = P(k).T = \beta$ **ensures** $P(k).t > \beta$	($\mathcal{L}_3$)
	$P(k).t = \beta \wedge Server.\tau(k) < \beta$ **until** $P(k).t = Server.\tau(k) = \beta$	($\mathcal{L}_4$)
	inv. $\langle \exists\ k : 0 \leq k \leq N-1 :: Server.\lambda = P(k).\lambda \rangle$	(MA$_1$)
	$Server.\lambda = P(k).\lambda$ **until** $Server.\lambda = P(k+1 \bmod N).\lambda$	(MA$_2$)
	inv. $Server.T \leq \langle \min\ l :: Server.\tau(l) \rangle$	(MA$_3$)
	$\langle \min\ l :: Server.\tau(l) \rangle = \beta$ **unless** $\langle \min\ l :: Server.\tau(l) \rangle > \beta$	(MA$_4$)
inv.	$P(k).t = P(k).T = \beta \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta$	($\mathcal{I}_2$)
inv.	$Server.\tau(k) \leq P(k).t$	($\mathcal{I}_3$)
inv.	$\mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min\ l :: Server.\tau(l) \rangle$	($\mathcal{I}_4$)
inv.	$Server.\lambda = P(k).\lambda \Rightarrow Server.t = P(k).t$	(MA$_5$)
inv.	$Server.\lambda = P(k).\lambda \Rightarrow Server.T = P(k).T \vee \neg\mathsf{def}(P(k).T)$	(MA$_6$)
inv.	$Server.\lambda = P(k).\lambda \Rightarrow Server.pos = k$	(MA$_7$)

Table 2. Lemmas and invariants needed for verification of the Mobile Agent solution.

which states that a client always knows eventually the value of a GVT estimate, and the safety property is

$$P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \textbf{ unless } P(k).t = P(k).T = \beta, \qquad (\text{MA}_9)$$

which actually would allow us to reformulate ($\mathcal{L}_2$) in terms of **until** rather than **leads-to**. Application of the PSP theorem to (MA$_8$) and (MA$_9$) yields

$$\begin{aligned} & P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \\ \mapsto\ & (Server.T = P(k).t = P(k).T = \langle \min\ l :: Server.\tau(l) \rangle = \beta) \vee \\ & P(k).t = P(k).T = \beta \\ \Rightarrow\ & P(k).t = P(k).T = \beta \end{aligned}$$

which proves ($\mathcal{L}_2$) by application of the implication theorem.

PROOF OF (MA$_8$). This obligation can be regarded as a consequence of transitivity and the induction principle of **leads-to** applied to (MA$_2$) and

$$Server.\lambda = P(k).\lambda\ \mapsto\ Server.T = P(k).T = \langle \min\ l :: Server.\tau(l) \rangle,$$

which states that co-location of *Server* with a client eventually leads to communication of the GVT estimate. We do not consider further the case when the RHS already holds, as it follows trivially from implication. It is then worth considering two cases, according to the value of the GVT at *Server*.
If such value is out-of-date, then the obligation

$$\begin{aligned} & Server.\lambda = P(k).\lambda \wedge Server.T < \langle \min\ l :: Server.\tau(l) \rangle \\ \textbf{ensures}\ & Server.T = P(k).T = \langle \min\ l :: Server.\tau(l) \rangle \end{aligned}$$

guarantees that it becomes up-to-date, and can be proven by reasoning about *Server* alone. When the LHS holds, the only statements enabled are the first two in *Server*, the guard of the third statement being false. Thus, *Server* movement is prevented and no other statement can modify the value of $Server.T$ except

for the second one in *Server*. This is eventually executed because of the fairness assumption and establishes the RHS.

If, on the other hand, the GVT at *Server* is up-to-date, it may still be the case that the GVT at the client is undefined, because the client advanced its local timer. In this case, we prove that

$$\begin{aligned} & Server.\lambda = P(k).\lambda \wedge Server.T < \langle \min\ l :: Server.\tau(l)\rangle \wedge \neg \mathsf{def}(P(k).T) \\ \mapsto\ & Server.T = P(k).T = \langle \min\ l :: Server.\tau(l)\rangle \end{aligned}$$

Again, two cases are possible depending on whether the new value of the local timer has been already recorded by the *Server*:

(1) if $Server.\tau(k) = t$, the local timer has been recorded. If this timer is actually the new minimum among the values contained in τ, then movement of *Server* is prevented and the proof follows from the previous case where the GVT at *Server* is out-of-date. Otherwise, it means that both the local timer has been recorded and the new GVT has been computed, thus the third statement can cause migration of *Server*. The mobile component, however, will eventually complete a full round, thanks to (MA_2), and return to *P(k)*. There, the value of the GVT will be communicated to the client because of the transient variable sharing taking place at engagement, as specified by the first statement in the **Interactions** section.

(2) if $Server.\tau(k) < t$, the local timer has not yet been recorded by *Server*, and thus movement of *Server* is prevented. However, it is straightforward to prove that

$$\begin{aligned} & Server.\lambda = P(k).\lambda \wedge Server.T < \langle \min\ l :: Server.\tau(l)\rangle \wedge \\ & \neg \mathsf{def}(P(k).T) \wedge Server.\tau(k) < t \\ \textbf{ensures}\ & Server.\lambda = P(k).\lambda \wedge Server.T < \langle \min\ l :: Server.\tau(l)\rangle \wedge \\ & \neg \mathsf{def}(P(k).T) \wedge Server.\tau(k) = t \end{aligned}$$

because the first statement in *Server* will establish the RHS due to the fairness assumption. At this point, the proof follows from the previous point.

7.2.4 *Proof of Remote Evaluation Solution.* Figure 3 shows the lemmas and invariants needed to prove the REV solution. In this section, we provide the proof outlines for one of them, building upon what we already demonstrated in the previous sections. The proof outlines for the remaining obligations are provided in Appendix C.

PROOF OF ($\mathcal{L}_2$). The proof of this lemma is similar to the one derived for the CS solution, hence we provide only an informal description for it. We do not consider further the case $P(k).T = \beta$, after applying the implication theorem. Then, we are left with two cases, according to (REV_1). With the definitions provided in Section 7.2.2, it can be proven that

$$\begin{aligned} & \mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Min(k).\lambda = P(k).\lambda \\ \mapsto\ & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge Min(k).\lambda = P(k).\lambda \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{REV}, \text{MINSERV}, \bot \rangle \end{aligned}$$

$P(k).t = \langle \min\ l :: Server.\tau(l)\rangle = \beta \mapsto P(k).t = P(k).T = \beta$	$(\mathcal{L}_2)$
$P(k).t = P(k).T = \beta$ **ensures** $P(k).t > \beta$	$(\mathcal{L}_3)$
$P(k).t = \beta \wedge Server.\tau(k) < \beta$ **until** $P(k).t = Server.\tau(k) = \beta$	$(\mathcal{L}_4)$
inv. $Min(k).\lambda = P(k).\lambda \vee Min(k).\lambda = Server.\lambda$	(REV_1)
inv. $Min(k).\lambda = P(k).\lambda \Rightarrow \neg\mathsf{def}(Min(k).q)$	(REV_2)
inv. $\mathsf{def}(Min(k).\tau) \Leftrightarrow Min(k).\tau = Server.\tau$	(REV_3)
inv. $Min(k).t = P(k).t$	(REV_4)
inv. $Min(k).\lambda = Server.\lambda \Leftrightarrow Min(k).\tau = Server.\tau$	(REV_5)
inv. $true \Rightarrow \mathsf{def}(Server.\tau)$	(REV_6)
inv. $P(k).t = P(k).T = \beta \Rightarrow \langle \min\ l :: P(l).t\rangle = \beta$	$(\mathcal{I}_2)$
inv. $Server.\tau(k) \leq P(k).t$	$(\mathcal{I}_3)$
inv. $\mathsf{def}(P(k).T) \Rightarrow P(k).T \leq \langle \min\ l :: Server.\tau(l)\rangle$	$(\mathcal{I}_4)$
inv. $\neg\mathsf{def}(Min(k).\tau) \Rightarrow \neg\mathsf{def}(Min(k).q)$	(REV_7)
inv. $Min(k).\lambda = Server.\lambda \Rightarrow \neg\mathsf{def}(P(k).T)$	(REV_8)
inv. $Min(k).\lambda = Server.\lambda \Leftrightarrow \mathsf{def}(Min(k).\tau)$	(REV_9)
inv. $true \Rightarrow \mathsf{def}(Min(k).t)$	(REV_{10})
inv. $\neg\mathsf{def}(Server.q(k)) \Leftrightarrow \neg\mathsf{def}(Min(k).q)$	(REV_{11})
inv. $P(k).RQ \uparrow 1 = k \Rightarrow Min(k).\lambda = P(k).\lambda$	(REV_{12})

Table 3. Lemmas and invariants needed for verification of the Remote Evaluation solution.

with arguments similar to those used in proving (CS_9), together with invariant (REV_2). The presence of a client request triggers execution of the second statement in the **Interactions** section, which causes migration of the corresponding *Min(k)* process to get co-located with the *Server*, because of the fairness assumption. Once co-located, invariant (REV_3) guarantees that the first two statements are enabled, thus enabling update of the GVT estimate. If the LHS of the lemma holds, actually no one else can execute actions and thus changing the GVT estimate, because $P(k).t$ is the minimum timer. *Min(k)* remains co-located with *Server*, thus sharing the values in τ and in the message buffer $q(i)$, until the last statement in *Min(k)* is executed. It can be proven, analogously to (CS_{11}), that such statement stores a value in the *Server* message queue, which is eventually communicated to *Min(k)* and there assigned to client's GVT estimate, thus satisfying the goal.

8. DISCUSSION

Mobile UNITY is a new model of distributed computing specialized for mobile computations, 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 suffices 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 [McCann and Roman 1997].

Against this background of promising technical developments, this paper raised a simple question: Can Mobile UNITY model in straightforward manner the kinds of interactions 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. 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 not being available (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 trigger for generating a request in the first place—this led to an elegant separation between the actions 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 may be 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.

In addition to precise modeling of the characteristics of mobile code, we were also able to reason about such models and prove properties of their expected behavior. This has been accomplished by using the UNITY proof logic, whose corpus of theorems and techniques is still available for use with Mobile UNITY.

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 assumptions which are at the foundation of the proof logic. 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.

Mobile Unity constructs are also likely to find applicability in describing how to deal with security issues in mobile code. For instance, the policies that rule acceptance or rejection of incoming code and the corresponding countermeasures could be specified in the **Interactions** section as a set of reactive statements, thus effectively separating the description of the system behavior from the policies that keep it in a safe state. Nevertheless, despite its relevance in code mobility, security is outside of the scope of this work. Here, we focused our investigation on the capability to model precisely the notion of migrating code. In our view, this is a precondition for defining a corresponding security model.

Other models of mobility have been proposed, and it is interesting to note that until very recently the only formal model of concurrency to refer to mobility explicitly was π-calculus [Milner et al. 1992], a process algebra proposed by Milner and his colleagues. In π-calculus there is no formal concept of space. Mobility is equated to the ability to "express processes which have changing structure". Under this definition, any model able to pass processes as values, e.g., the Actor model, or link names as values (π-calculus) qualifies. The approach pursued by π-calculus has inspired many researchers [Fournet et al. 1996; De Nicola et al. 1998; Amadio 1997], who today are building variants and extensions of process algebras in order to provide a better notion of location. The work by Cardelli and Gordon [Cardelli and Gordon 2000], for instance, appears to be directly inspired by the domain of mobile code. The concept of nested localities whose access is regulated by capabilities associated to mobile processes closely recalls the Telescript [Magic 1995; White 1996] model. Although a precise evaluation of π-calculus and its derivatives against Mobile Unity is outside the scope of this paper, we recognize that the approach pursued by researchers in process algebras is clearly distinct from the modeling strategy we adopted. Mobile Unity is essentially state-based, and we believe that this aspect, combined with a notation which is closer to the one of a conventional programming language, arguably makes Mobile Unity easier for people that are more confident with programming than with formal modeling. Furthermore, we believe that approaching mobility from a different angle is likely to unveil insights that are complementary to those elicited with process algebras, thus contributing to building a better understanding of mobility.

9. CONCLUSIONS

Mobile Unity has been the product of two distinct mindsets. First, it was the notion of developing transient and transitive forms of the two Unity modes of composition, union and superposition. Second, it was the fascination with ad hoc networks of components moving freely through space, computing in a highly decoupled style, and interacting opportunistically. Neither of the two motivations had anything to do with code mobility. Yet, our evaluation of the model against several established paradigms in mobile code languages shows that the essential features of code mobility have immediate counterparts in Mobile Unity thus allowing direct applicability of its proof logic to programs involving mobile code. While more

work is needed to refine the relation between code and (physical) component movement, this exercise in modeling and formal reasoning provides a strong incentive to seek out solutions and models which are insensitive to the nature of the movement process, be it logical or physical.

ACKNOWLEDGMENTS

This paper is based upon work supported in part by the National Science Foundation (NSF) under grants No. CCR-9217751 and CCR-9624815. Gian Pietro Picco was partially supported by Centro Studi e Laboratori Telecomunicazioni (CSELT) S.p.A., Italy. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or CSELT.

APPENDIX

We provide here the proof outlines for the obligations used but not proven in the previous sections.

A. CLIENT-SERVER

Invariant (CS_2) can be proven directly from the program text.

PROOF OF (CS_8). If *Server* holds an out-of-date value for the GVT, this is eventually cleared and subsequently replaced by a new, up-to-date value corresponding to the minimum timer in $Server.\tau$. At this point, we fall back in the state on the LHS of (CS_7), for which we have already shown that such a value is eventually communicated to the client. The GVT estimate on the *Server* is cleared whenever a reply is sent back to the client, hence we prove that a message request is eventually sent to the *Server*, with the macro definitions of Section 7.2.2:

$$\begin{aligned} & \mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \neg\mathsf{def}(Server.q(k)) \\ \mapsto\ & \mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta \rangle \end{aligned}$$

The above can be proven with arguments similar to those used for (CS_9). Invariant (CS_1) lets us rewrite the obligation as

$$\begin{aligned} & \mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \neg\mathsf{def}(Server.q(k)) \\ \mapsto\ & (\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta \rangle) \vee \\ & (\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta \rangle), \end{aligned} \qquad (\text{CS}_{14})$$

thus splitting the obligation in two cases. Due to the concurrency in the system, the statements of the clients *P(k)* are interleaved with those of the *Server*. Hence, while client *P(k)* is preparing a request, $Server.T$:

(1) may still contain an out-of-date value;

(2) may be either undefined or up-to-date, because in the meanwhile the *Server* has sent a reply to some of the other clients or it has established the new GVT estimate as the minimum in $Server.\tau$.

The second disjunct implies the LHS of (CS_{10}), then we can apply transitivity thanks to the implication theorem and thus show that it eventually leads to the goal, that is

$$\begin{aligned}&\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ &P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \\ \mapsto\ &P(k).t = P(k).T = \beta\end{aligned} \qquad (\text{CS}_{15})$$

We can exploit this fact through application of the following *cancellation theorem* of **leads-to** [Chandy and Misra 1988]

$$\frac{p \mapsto q \vee b,\ \ b \mapsto r}{p \mapsto q \vee r}. \qquad (\mathcal{T}_{11})$$

In our case, application of the cancellation theorem to (CS_{14}) and (CS_{15}) yields

$$\begin{aligned}&\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \neg\mathsf{def}(Server.q(k)) \\ \mapsto\ &(\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ &P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle) \ \vee P(k).t = P(k).T = \beta\end{aligned}$$

Our proof strategy uses the cancellation theorem to perform the verification of the communication steps by ruling out the case where the GVT estimate becomes up-to-date on the *Server*, thus examining always the worst case. In doing this, we proceed along the lines of the proof developed for (CS_7), thus reusing the results already achieved. Our next goal is to prove that for the first disjunct the following holds:

$$\begin{aligned}&\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \neg\mathsf{def}(Server.q(k)) \wedge \\ &P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \\ \mapsto\ &P(k).t = P(k).T = \beta.\end{aligned}$$

Similarly to (CS_{10}), it can be proven from the program text that

$$\begin{aligned}&\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \\ &\neg\mathsf{def}(Server.q(k)) \ \wedge P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \\ \textbf{ensures}\ \ &\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \\ &Server.q(k) = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle\end{aligned}$$

which, thanks to invariant (CS_1), can be rewritten as

$$\begin{aligned}&\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta \wedge \\ &\neg\mathsf{def}(Server.q(k)) \wedge \\ &P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle \\ \textbf{ensures}\ \ &(\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge Server.T < \beta) \vee \\ &(\mathsf{minTimer}(k,\beta) \wedge \mathsf{oldClientT}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \\ &Server.q(k) = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta\rangle\end{aligned}$$

Repeated application of the cancellation theorem and of the proof obligations developed for (CS$_7$), whose details are here omitted, yields to the final goal of proving that

$$\begin{aligned} &\mathsf{minTimer}(k,\beta) \wedge \mathit{Server}.T < \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \mapsto\; & P(k).t = P(k).T = \beta, \end{aligned}$$

that is, even in the case when the *Server* accepts a request while still owning an out-of-date value for the GVT estimate, this eventually leads to communication of an up-to-date value to *P(k)*. We demonstrate this by proving that

$$\begin{aligned} &\mathsf{minTimer}(k,\beta) \wedge \mathit{Server}.T < \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \mapsto\; & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).T < \beta \end{aligned} \qquad (\text{CS}_{16})$$

holds, by observing that the RHS corresponds to the LHS of (CS$_{13}$), for which we have already proven that it leads to the goal. This completes the proof together with invariant (CS$_2$), and disjunction and transitivity of **leads-to**. Property (CS$_{16}$) is proven by the following chain of **ensures**:

$$\begin{aligned} & \mathsf{minTimer}(k,\beta) \wedge \mathit{Server}.T < \beta \wedge \mathit{Server}.q(k) \uparrow 2 = \text{WAIT} \\ \textbf{ensures}\; & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{def}(\mathit{Server}.T) \wedge \mathit{Server}.q(k) = \langle k,\bot,\bot,\bar\beta\rangle \wedge \bar\beta < \beta \\ \textbf{ensures}\; & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge P(k).RQ = \langle k,\bot,\bot,\bar\beta\rangle \wedge \bar\beta < \beta \\ \textbf{ensures}\; & \mathsf{minTimer}(k,\beta) \wedge \neg\mathsf{oldServerT}(\beta) \wedge \neg\mathsf{def}(\mathit{Server}.q(k)) \wedge P(k).T < \beta. \end{aligned}$$

If the *Server* owns an out-of-date value for the GVT estimate and there is a request waiting, the only step that can be taken by the *Server* is to clear the GVT value and transmit the out-of-date GVT estimate. This in turn triggers message delivery to the client and the subsequent voiding of the corresponding element of the message queue of *Server*. However, the important point is that, because the minimum value on $\mathit{Server}.\tau$ is β, whenever the GVT estimate is eventually established, it is given this value, which is eventually communicated to the client, as proven by (CS$_7$).

PROOF OF ($\mathcal{L}_3$). Invariant (CS$_3$) guarantees that if the GVT estimate is defined within a client, the latter has no pending request. Thus, if the LHS of ($\mathcal{L}_3$) holds there is no incoming new value for the GVT estimate which could possibly change it, and ($\mathcal{I}_1$) guarantees that $P(k).T$ can never increase past $P(k).t$. Consequently, the LHS of ($\mathcal{L}_3$) is preserved. Finally, the first statement in *P(k)* together with the fairness assumption guarantee that the RHS is established, according to the definitions of f and g.

PROOF OF ($\mathcal{L}_4$). We prove separately the two parts of the **until**. As for the **unless** part, invariant ($\mathcal{I}_4$) guarantees that the following holds

$$\mathit{Server}.\tau(k) < \beta \Rightarrow (\mathsf{def}(P(k).T) \Rightarrow P(k).T < \beta),$$

then $P(k).t$ cannot increase while $\mathit{Server}.\tau(k) < \beta$ holds. Nevertheless, after execution of the first statement in *Server*, the RHS of the **unless** might be established. The **leads-to** is proven with arguments similar to those used to reason about communication in the previous proofs. Informally, starting

from the state in the LHS, the client eventually generates a request that carries the new value for the local timer. The message containing this value is subsequently transferred to the message queue of the *Server* by the statements in the **Interactions** section, and then the corresponding value in $Server.\tau$ is updated.

PROOF OF ($\mathcal{I}_2$). The invariant holds initially, because the value of each timer and GVT estimate is initially zero. To prove the invariant we need to prove that

$$\textbf{inv.}\ P(k).t = P(k).RQ \uparrow 4 = \beta \wedge P(k).RQ \uparrow 1 = k \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta \qquad (\text{CS}_{17})$$

In order to prove this invariant, we observe that $P(k).t$ is not allowed to change because of invariant (CS_3), which can be rewritten as

$$\textbf{inv.}\ \mathsf{def}(P(k).RQ) \Rightarrow \neg\mathsf{def}(P(k).T),$$

and thus proves that when the LHS of (CS_{17}) holds the first statement is always disabled. Furthermore, (CS_{17}) is proven by the following invariants

$$\textbf{inv.}\ P(k).t = Server.q(k) \uparrow 4 = \beta \wedge Server.q(k) \uparrow 1 = k \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta$$

$$\textbf{inv.}\ P(k).t = Server.T = \beta \wedge Server.q(k) \uparrow 2 = \text{WAIT} \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta$$

$$\textbf{inv.}\ P(k).t = \langle \min\ l :: Server.\tau(l) \rangle = \beta \wedge Server.q(k) \uparrow 2 = \text{WAIT} \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta$$

which can be proven directly from the program text, and observing that

$$\textbf{inv.}\ P(k).t = \beta \wedge \langle \min\ l :: Server.\tau(l) \rangle = \beta \Rightarrow \langle \min\ l :: P(l).t \rangle = \beta$$

holds because of ($\mathcal{I}_3$).

PROOF OF ($\mathcal{I}_3$). The invariant holds initially, because the value of each local timer and each element of $Server.\tau$ is initially zero. To prove that ($\mathcal{I}_3$) holds, we observe that the first statement in *Server* is the only one that modifies $Server.\tau$, and that this is done according to invariant (CS_4).

PROOF OF ($\mathcal{I}_4$). The invariant holds initially, because each element of $Server.\tau$ and each GVT estimate is initially zero. The first statement in *P(k)* does not affect the invariant, as if the LHS is false before execution of the statement this is disabled, otherwise the LHS becomes false after execution of the statement. To prove that the invariant holds before and after the execution of the other statements which might affect the invariant we can use the invariants

$$\textbf{inv.}\ P(k).RQ \uparrow 1 = k \Rightarrow P(k).RQ \uparrow 4 \le \langle \min\ l :: Server.\tau(l) \rangle$$

$$\textbf{inv.}\ Server.q(k) \uparrow 1 = k \Rightarrow Server.q(k) \uparrow 4 \le \langle \min\ l :: Server.\tau(l) \rangle$$

$$\textbf{inv.}\ \mathsf{def}(Server.T) \Rightarrow Server.T \le \langle \min\ l :: Server.\tau(l) \rangle,$$

together with the lemma (CS_5), which states that the minimum value in $Server.\tau$, if changed, can only increase monotonically.

PROOF OF (CS_1). The invariant holds initially, and is proven from the program text, taking advantage of lemma (CS_5).

PROOF OF (CS_3). The invariant can be rewritten as

$$\textbf{inv.}\ \mathsf{def}(P(k).RQ) \Rightarrow \neg\mathsf{def}(P(k).T)$$

In order to prove the invariant above, the last interaction requires us to prove that

$$\textbf{inv.}\ \ \mathit{Server}.q(k) \uparrow 1 = k \Rightarrow \neg\mathsf{def}(P(k).T)$$

which can be proven directly from the program text by observing that

$$\textbf{inv.}\ \ \mathit{Server}.q(k) \uparrow 1 = k \Rightarrow \neg\mathsf{def}(\mathit{Server}.q(k) \uparrow 2),\ \text{and}$$
$$\textbf{inv.}\ \ P(k).RQ \uparrow 1 = k \Rightarrow \neg\mathsf{def}(P(k).T)$$

hold.

PROOF OF ($\textsc{cs}_4$). The invariant holds initially. Then, we observe that the only statement that could possibly invalidate the invariant is the first interaction, which copies the value of $P(i).RQ$ into $\mathit{Server}.q(i)$. However, we demonstrate that

$$\textbf{inv.}\ \ P(k).RQ \uparrow 1 = \text{SERVER} \Rightarrow P(k).RQ \uparrow 4 = P(k).t$$

holds, so that the value communicated to *Server* is still equal to $P(i).t$. This is proven observing the statements modifying $P(i).RQ$, namely, the first statement in *P(i)* and the second interaction, preserve the invariant, and that no statement can modify the value of the local timer except for the first one in *P(i)*. Moreover, it can be demonstrated from the program text that

$$\textbf{inv.}\ \ P(k).RQ \uparrow 1 = \text{SERVER} \Rightarrow \neg\mathsf{def}(P(k).T),$$

holds, which guarantees that such a statement is disabled when the LHS holds.

PROOF OF ($\textsc{cs}_5$). We observe that no statement modifies the LHS of the **unless**, except for the first one in *Server*. Under the guard of this statement, invariant ($\textsc{cs}_4$) guarantees that $\mathit{Server}.\tau(l)$ is updated to the current value of $P(l).t$. Then, the new value for the minimum is

$$\min(P(l).t, \langle \min\ m : m \neq l :: \mathit{Server}.\tau(m)\rangle) \geq \beta$$

because of invariant ($\mathcal{I}_3$).

B. MOBILE AGENT

Invariants ($\mathcal{I}_2$), ($\textsc{ma}_1$), ($\textsc{ma}_3$), and ($\textsc{ma}_7$) can be proven directly from the program text.

PROOF OF ($\textsc{ma}_9$). We do not consider further the case $P(k).T = \beta$ because, analogously to ($\mathcal{T}_5$), implication is a special case of **unless**. Thus, we prove

$$\begin{aligned} & P(k).t = \langle \min\ l :: \mathit{Server}.\tau(l)\rangle = \beta \wedge (P(k).T < \beta \vee \neg\mathsf{def}(P(k).T)) \\ \textbf{unless}\ & P(k).t = P(k).T = \beta, \end{aligned} \qquad (\textsc{ma}_{10})$$

by observing that, for the case considered, execution of the first statement in *P(k)* can never affect the value of the local timer in *P(k)*, due to the definition of f and g. Furthermore, if *P(k)* is not co-located with *Server*, $P(i).T$ cannot be changed to a value different from those in the LHS of ($\textsc{ma}_{10}$), because sharing is not in place and the first statement in *P(k)* can only void the

GVT estimate. Finally, according to invariant (MA$_1$), the *Server* must be co-located with some other client which might possibly affect the minimum value in $Server.\tau$. However, invariant ($\mathcal{I}_3$) and lemma (MA$_4$), together with the LHS, ensure that no other client may lower the minimum in $Server.\tau$. Hence, we can now restrict our analysis to the case where *P(k)* is co-located with *Server*. In this case, the first statement in *Server* does not affect the LHS of (MA$_{10}$) because of invariant (MA$_5$). Execution of the second statement in *Server* may actually establish the RHS, through the read-only sharing expressed by the first interaction and captured by invariant (MA$_6$). Finally, the migration taking place upon execution of the last statement triggers both engagement and disengagement. Engagement may either establish the RHS of (MA$_{10}$), in the case where *P(k)* and *Server* engage and the latter already has an up-to-date value of the GVT, or leave the LHS unaffected, by assigning an out-of-date value to $P(i).T$. Disengagement always leaves the LHS unaffected, because no **disengage** clause is specified, and thus both components retain their current values.

PROOF OF ($\mathcal{L}_3$). The only statements which can affect the LHS of ($\mathcal{L}_3$) are the first statement in *P(k)* and the second statement *Server*. The former actually establishes the RHS, due to the definition of f and monotonicity of g. The latter might in principle affect the value of $P(k).T$ through the sharing expressed by the first interaction. However, the LHS of ($\mathcal{L}_3$), together with invariants ($\mathcal{I}_4$) and ($\mathcal{I}_3$), yields

$$P(k).t = P(k).T = \langle \min\ l :: Server.\tau(l) \rangle = \beta$$

and hence the LHS of ($\mathcal{L}_3$) is preserved by execution of this statement. Finally, fairness guarantees that the first statement in *P(k)* is executed eventually, thus proving the **ensures**.

PROOF OF ($\mathcal{L}_4$). The **unless** part of the property is proven as in the CS solution. In order to prove the **leads-to** part, we proceed analogously to the proof of ($\mathcal{L}_2$) by applying the PSP theorem. The progress property is

$$true \ \mapsto\ P(k).t = Server.\tau(k), \qquad (\text{MA}_{11})$$

which states that a timer value is always eventually communicated to the *Server*, and the safety property is the **unless** part of ($\mathcal{L}_4$). Application of ($\mathcal{T}_{10}$) yields

$$\begin{aligned} & P(k).t = \beta \wedge Server.\tau(k) < \beta \\ \mapsto\ & (P(k).t = Server.\tau(k) \wedge P(k).t = \beta \wedge Server.\tau(k) < \beta) \vee \\ & P(k).t = Server.\tau(k) = \beta, \end{aligned}$$

from which the goal follows from implication, since the first disjunct in the RHS of the above never holds. Property (MA$_{11}$) is proven by transitivity and the induction principle of **leads-to** applied to (MA$_2$) and

$$Server.\lambda = P(k).\lambda \ \textbf{ensures}\ Server.\tau(k) = P(k).t.$$

This property is proven by observing that, as long as the RHS does not hold, *Server* cannot move and hence the LHS remains unchanged. Finally, invariants (MA$_7$) and (MA$_5$) and the first statement in *Server*, together with the fairness assumption, guarantee that such a statement is executed, thus establishing the RHS of the **ensures**.

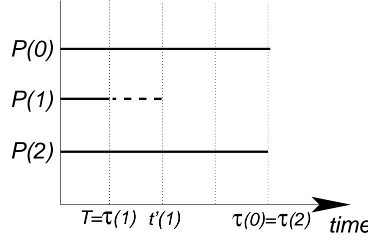

Fig. 13. A situation where *Server*'s departure is delayed.

Proof of (MA_2). The **unless** part is proven straightforwardly by observing that no statement affects the LHS except for the last one in *Server*, which actually establishes the RHS. As for the **leads-to** part, inspection of text of the *Server* program evidences that *Server* is allowed to leave only after the local timer of *P(k)* has been registered in τ and the GVT estimate has been updated accordingly. However, *Server* is not guaranteed to depart right after these two conditions are established together—i.e., the guard of the third statement in *Server* is true. Figure 13 describes a situation where *Server* departure is delayed. Here, client *P(1)* owns the minimum timer, and all the other processes have scheduled their actions far in the future with respect to it. After τ has been updated with the value of the local timer and the new GVT estimate has been computed, $T = \tau(1) = t(1)$ holds. Then, *Server* is allowed to depart. However, at the same time *P(1)* is allowed to execute its statement and update its timer to $t'(1)$, as shown. Since the other processes have scheduled actions in the distant future, *P(1)* and *Server* are the only processes enabled to make progress. Depending on the fair, non-deterministic scheduling, either statement can be selected for execution. If the statement of *P(1)* is selected, its execution changes the value of the local timer, thus disabling the guard of the last statement in *Server* and thus preventing its departure. In the worst case, if *P(1)* is "faster" than *Server*, the last statement in *Server* might be enabled several times, each time being subsequently disabled by the timer update in *P(1)*. This situation is guaranteed to cease eventually when the timer in *P(1)* is greater than the values in τ for the other timers. In this state, when *Server* establishes the GVT estimate, invariant (MA_3) guarantees that the GVT estimate is behind the local timer of *P(1)* and thus, due to the definition of f and g, prevents further update of this timer. Our proof demonstrates that even in the worst case, as the one depicted in Figure 13, *Server* eventually departs. The other cases are encompassed in the formal proof for the worst case. Hence, our goal is to show that

$$\begin{aligned} & \mathit{Server}.\lambda = P(k).\lambda \ \wedge P(k).t \leq \beta \wedge \langle \min\ l : k \neq l :: \mathit{Server}.\tau(l) \rangle = \beta \\ \mapsto\ & (\mathit{Server}.\lambda = P(k).\lambda \ \wedge \mathit{Server}.T = \langle \min\ l :: \mathit{Server}.\tau(l) \rangle = \beta \ \wedge \\ & \mathit{Server}.\tau(k) = P(k).t \wedge P(k).t > \beta) \ \vee \mathit{Server}.\lambda = P(k+1 \bmod N).\lambda, \end{aligned} \tag{MA_{12}}$$

holds, where the LHS captures the formal definition for the worst case as described earlier. The property above states that a state is reached eventually

where either *Server* has departed or the guard of the last statement in *Server* is true and cannot be falsified by an update to $P(k).t$—since it can no longer affect the minimum in $Server.\tau$. From this state, in the RHS of (MA$_{12}$), application of disjunction, transitivity, and the cancellation theoreom of **leads-to** to the above and to

$$\begin{array}{ll} & Server.\lambda = P(k).\lambda \wedge P(k).t > \beta \wedge \\ & Server.T = \langle \min\ l :: Server.\tau(l) \rangle = \beta \wedge Server.\tau(k) = P(k).t \\ \textbf{ensures} & Server.\lambda = P(k+1 \bmod N).\lambda \end{array} \quad (\text{MA}_{13})$$

proves (MA$_2$). This last property is proven directly from the program text by observing that its LHS cannot be changed by any statement, and it enables the guard of the last statement in *Server*. Then, the fairness assumption guarantees execution of such a statement, which establishes the goal.

PROOF OF (MA$_{12}$). First, we observe that the following holds from invariant ($\mathcal{I}_3$):

$$\begin{array}{l} \langle \min\ l : k \neq l :: Server.\tau(l) \rangle = \beta \wedge P(k).t \leq \beta \\ \Rightarrow P(k).t \geq \langle \min\ l :: Server.\tau(l) \rangle. \end{array} \quad (\text{MA}_{14})$$

If $P(k).t > \langle \min\ l :: Server.\tau(l) \rangle$ holds, it can be shown directly from the program text that, thanks to invariant (MA$_5$), the guard of the last statement is false, and then the *Server* is stuck at the client location. The fairness assumption guarantees that $P(k).t = Server.\tau(k) = \langle \min\ l :: Server.\tau(l) \rangle$ is established eventually, thanks to the first statement in *Server*. Then, for the sake of readability, we introduce the predicate

$$\begin{array}{rl} \mathsf{farMin} \equiv & \langle \min\ l : k \neq l :: Server.\tau(l) \rangle = \beta \wedge \\ & Server.\tau(k) = P(k).t = \langle \min\ l :: Server.\tau(l) \rangle \leq \beta, \end{array}$$

that holds when $P(k).t$ is less than or equal to the next minimum value in $Server.\tau$ and its value is already known to *Server*. We demonstrate (MA$_{12}$) by proving that

$$\begin{array}{ll} & Server.\lambda = P(k).\lambda \wedge \mathsf{farMin} \wedge \delta = \max(\beta - P(k).t, 0) \\ \mapsto & (Server.\lambda = P(k).\lambda \wedge (\mathsf{farMin} \wedge \delta < \max(\beta - P(k).t, 0)) \vee \\ & (Server.T = \langle \min\ l :: Server.\tau(l) \rangle = \beta \wedge P(k).t > \beta)) \vee \\ & Server.\lambda = P(k+1 \bmod N).\lambda. \end{array} \quad (\text{MA}_{15})$$

holds using the induction principle of **leads-to**, applied over the distance between the timer $P(k).t$ and the value β of the next smallest element in $Server.\tau$. The above states that either the distance among the minimum and the next minimum value in $Server.\tau$ is reduced, or the value of the local timer becomes greater than the current minimum value, or *Server* departs. The progress prop-

erty proving induction is provided by the following:

$$\begin{array}{ll} & Server.\lambda = P(k).\lambda \wedge \mathsf{farMin} \\ \textbf{ensures} & Server.\lambda = P(k).\lambda \wedge \mathsf{farMin} \wedge Server.T = \langle \min\ l :: Server.\tau(l)\rangle = \beta \\ \textbf{ensures} & (Server.\lambda = P(k).\lambda \wedge \\ & (Server.T = \langle \min\ l :: Server.\tau(l)\rangle = \beta \wedge P(k).t > \beta) \vee \\ & Server.\lambda = P(k+1 \bmod N).\lambda. \end{array}$$

The first **ensures** is proven by ruling out the case where the RHS already holds, and considering the case when the GVT on *Server* is out-of-date. In this case, the last statement in *Server* is false and hence is keeping the *Server* co-located with the client. Fairness and the second statement in *Server* prove the property. As for the second **ensures**, we observe that when its LHS holds the guard in the last statement of *Server* is enabled. Due to the fairness assumption, the statement executes eventually, thus determining *Server* departure. However, due to the sharing expressed by invariant ($\textsc{ma}_6$), the statement in *P(k)* is enabled and executes eventually even if *Server* is already departed, since the GVT estimate is retained according to the **disengage** clause. Since no assumption can be made on the scheduling of these statements, according to the order in which they are executed either the first or the second disjunct in the RHS of the **ensures** is established. Repeated application of this progress property when the first disjunct of ($\textsc{ma}_{15}$) holds leads either to third disjunct, which actually proves our goal, or to the second one. In this latter case, fairness, the transient sharing captured by invariant ($\textsc{ma}_5$), and the first statement in *Server* guarantee that the *Server* stays at the client location until the timer value mirrored in $Server.\tau$ equals the one owned by the client, thus leading to the RHS of ($\textsc{ma}_{12}$).

PROOF OF ($\textsc{ma}_6$). The invariant holds initially. If the *Server* is initially co-located with *P(k)*, the GVT estimate owned by the latter is undefined, thus preserving the invariant. If the two components are not co-located, then the LHS of the implication does not hold and thus preserves the invariant. As for the statements in the system, the first statement in *P(k)* preserves the invariant as well, in that it possibly undefines the GVT estimate in *P(k)* or leaves it unaffected. Execution of the second statement when *P(k)* and *Server* are co-located actually guarantees that the GVT estimates in *P(k)* and *Server* assume the same value. Finally, the migration expressed by the last statement in *Server* does not affect the invariant, since upon engagement on *P(k)* the **engage** clause of the first interaction guarantees that equality of the two GVT estimates is established, while disengagement falsifies the LHS of the implication and thus preserves the invariant.

PROOF OF ($\mathcal{I}_2$). The invariant holds initially. Then, the statements that may affect the invariant, namely, the one in *P(k)* and the second and third in *Server*, either leave the variables involved unaffected or falsify the LHS, thus preserving the invariant.

PROOF OF ($\mathcal{I}_3$). The invariant holds initially. Then, the first statement of *P(i)* satisfies the invariant, since $P(i).t$ can only increase, due to the definition of f and g. Furthermore, execution of the first statement in *Server* actually establishes the invariant. No other statement can affect the invariant.

PROOF OF ($\mathcal{I}_4$). The invariant holds initially. The first statement in *P(k)* falsifies the LHS of the invariant, thus preserving it. Then, we observe that the behavior of *Server* may affect the invariant only when the *Server* and *P(k)* are co-located. Then, invariants (MA$_3$) and (MA$_6$) guarantee that the invariant is preserved. The migration statement in *Server* does not affect the invariant neither on engagement, for which the aforementioned argument holds, or disengagement, where values are retained thus preserving the invariant.

PROOF OF (MA$_4$). We observe that the only statement that could affect the lemma is the first one in *Server*. However, invariant ($\mathcal{I}_3$) guarantees that either the LHS of the **unless** is unaffected or its RHS is established.

PROOF OF (MA$_5$). This invariant is proven with the same argument used for the verification of (MA$_6$) in Section 7.2.3.

C. REMOTE EVALUATION

Invariants (REV$_1$), (REV$_6$), (REV$_{10}$) and (REV$_{12}$) can be proven directly from the program text.

PROOF OF ($\mathcal{L}_3$). This lemma is proven with the same arguments used in Section 7.2 for the CS solution. In fact, it needs only reasoning about a single client, whose behavior is unchanged.

PROOF OF ($\mathcal{L}_4$). The **unless** part of the property is proven as in the CS solution. As for the **leads-to** part, we consider two cases according to invariant (REV$_1$). Our proof strategy is to show that:

(1) if *Min(k)* and *P(k)* are co-located, *Min(k)* eventually migrates to *Server*'s location,

$$\begin{aligned} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = P(k).\lambda \\ \mapsto\ & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda; \end{aligned} \qquad (\text{REV}_{13})$$

(2) if *Min(k)* and *Server* are co-located, then

$$\begin{aligned} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \\ \mapsto\ & P(k).t = Server.\tau(k) = \beta \end{aligned} \qquad (\text{REV}_{14})$$

holds, expressing the fact that the value of the local timer of *P(k)* is eventually communicated to *Server*.

Disjunction and transitivity of (REV$_{13}$) and (REV$_{14}$) complete the proof.

PROOF OF (REV$_{14}$). Invariant (REV$_4$) guarantees that *Min(k)* holds the proper value to be communicated to the *Server*. In addition, invariant (REV$_5$) guarantees that each update made to the copy of the global state belonging to *Min(k)* is seen by *Server* as well. The proofs for invariants (REV$_4$) and (REV$_5$), which both involve reasoning about transient sharing, are given later in the remainder of this section. The implication theorem ($\mathcal{T}_5$), together with the two invariants above, yields

$$\begin{aligned} & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \\ \mapsto\ & P(k).t = \beta \wedge Server.\tau(k) < \beta \wedge Min(k).\lambda = Server.\lambda \wedge \\ & Min(k).\tau(k) = Server.\tau(k) \wedge Min(k).t = P(k).t \end{aligned}$$

From the program text, it can be argued that

$$\begin{aligned} & P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \wedge \mathit{Min}(k).\lambda = \mathit{Server}.\lambda \wedge \\ & \mathit{Min}(k).\tau(k) = \mathit{Server}.\tau(k) \wedge \mathit{Min}(k).t = P(k).t \\ \textbf{ensures}\ & P(k).t = \beta \wedge \mathit{Min}(k).\lambda = \mathit{Server}.\lambda \wedge \\ & \mathit{Min}(k).\tau(k) = \mathit{Server}.\tau(k) \wedge \mathit{Min}(k).t = P(k).t \wedge \\ & \mathit{Min}(k).t = \mathit{Min}(k).\tau(k) \end{aligned}$$

which clearly leads to the goal by application of the implication theorem.

Proof of (REV_{13}). This is proven similarly to (CS_{13}), by application of disjunction according to invariant ($\mathcal{I}_2$) and of transitivity to the following chain of **ensures**:

$$\begin{aligned} & P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \wedge \mathit{Min}(k).\lambda = P(k).\lambda \\ \textbf{ensures}\ & P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \wedge \mathit{Min}(k).\lambda = P(k).\lambda \wedge \neg\mathsf{def}(P(k).T) \\ \textbf{ensures}\ & P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \wedge \mathit{Min}(k).\lambda = P(k).\lambda \wedge \\ & P(k).RQ = \langle \text{SERVER}, \text{CS}, \text{MINSERV}, \beta \rangle \\ \textbf{ensures}\ & P(k).t = \beta \wedge \mathit{Server}.\tau(k) < \beta \wedge \mathit{Min}(k).\lambda = \mathit{Server}.\lambda. \end{aligned}$$

Proof of (REV_3). The invariant holds initially, from the program text. If the LHS of the invariant holds before execution of the first statement, invariant (REV_9) guarantees that *Min(k)* and *Server* are co-located. Hence, the sharing expressed by the third interaction is in place, and the invariant is preserved after execution. On the other hand, if the LHS does not hold, the statement is disabled and the invariant is preserved. Migration statements preserve the invariant as well, because engagement and disengagement preserve the invariant either by establishing its RHS or by falsifying its LHS. No other statement can affect the invariant.

Proof of (REV_4). The only statement that may affect this invariant is the first one in *P(k)*. However, invariant (REV_8) let us conclude that, if (REV_4) and $\mathit{Min}(k).\lambda = \mathit{Server}.\lambda$ hold before execution of the statement, (REV_4) is preserved because the statement is disabled. On the other hand, if (REV_4) holds before execution of the statement while $\mathit{Min}(k).\lambda = P(k).\lambda$, then the first interaction takes place as soon as $P(k).t$ is updated, changing the local timer in *P(k)* accordingly. Migration statements do not affect the invariant, although in principle they could involve the read only shared variable in the first interaction. However, the **engage** clause in the first interaction actually establishes (REV_4). Furthermore, no **disengage** clause is specified, hence the value of the variables as specified by the invariant before disengagement is retained.

Proof of (REV_5). The invariant holds as the conjunction of (REV_3) and (REV_9).

Proof of (REV_9). The invariant holds initially, since *Min(k)* and *P(k)* are co-located and τ is undefined in *Min(k)*—both sides of (REV_9) are false. Looking at the program text, the only statements which may affect the invariant are those involving migration of *Min(k)* and the ones modifying $\mathit{Min}(i).\tau$. Let us consider the second statement in the **Interactions** section. If the LHS of (REV_9) holds before execution, the statement is disabled. On the other hand, if the LHS does not hold, $\mathit{Min}(k).\tau$ must be undefined in order to preserve the invariant.

Execution of the statement forces *Min(k)* to become co-located with *Server* and causes engagement of the τ variables belonging to *Min(k)* and *Server*. Because of invariant (REV_6), τ becomes defined in *Min(k)*, thus preserving the invariant. Similar reasoning holds for the last statement of the system. If the LHS and the RHS of (REV_9) are both false, invariant (REV_7) prevents execution of the statement and thus preserves the invariant. On the other hand, if both sides of (REV_9) are true, the migration expressed by the bracketed statement take place, thus triggering disengagement of the values of τ. As a consequence, after execution of the statement the LHS of the invariant is false and, according to **disengage** clause of the third interaction, τ becomes undefined in *Min(k)*, thus falsifying the RHS as well and preserving the invariant. Finally, let us consider the first statement in *Min(k)*. If the LHS of (REV_9) does not hold, the RHS is false as well, thus preventing execution of the statement and preserving the invariant. On the other hand, if *Min(k)* and *Server* are co-located, the invariant states that before execution of the statement τ is defined in *Min(k)*. However, execution of the statement must comply with invariant (REV_{10}), and thus preserves (REV_9).

PROOF OF ($\mathcal{I}_2$). The invariant holds initially, because the value of each timer and GVT estimate is initially zero. As in the corresponding proof in Appendix A, proving ($\mathcal{I}_2$) involves proving

$$\textbf{inv.}\ P(k).t = P(k).RQ \uparrow 4 = \beta \wedge P(k).RQ \uparrow 1 = k \Rightarrow \langle \min\ l :: P(l).t\rangle = \beta.$$

and observing that the first statement in *P(k)* is disabled when the LHS of the above holds, as a consequence of invariant (REV_5). Then, the above is proven by observing that

$$\textbf{inv.}\ P(k).t = Min(k).q \uparrow 4 = \beta \wedge Min(k).q \uparrow 1 = k \Rightarrow \langle \min\ l :: P(l).t\rangle = \beta$$

can be proven directly from the program text together with invariant (REV_2) and (REV_{11}), and also

$$\textbf{inv.}\ Min(k).t = Min(k).T = \beta \Rightarrow \langle \min\ l :: P(l).t\rangle = \beta$$

can be proven from the program text. The above, together with (REV_4), proves ($\mathcal{I}_2$).

PROOF OF ($\mathcal{I}_3$). The invariant holds initially, from program text. The are only two statement that may affect the invariant. If *Min(k)* and *P(k)* are co-located, the first statement in *P(k)* may execute. However, the invariant is preserved according to the definition of f and g. If the two components are not co-located, invariant (REV_8) guarantees that such a statement is disabled. In this case, the first statement in *Min(k)* is enabled, and it actually establishes the invariant because of (REV_3) and (REV_4).

PROOF OF ($\mathcal{I}_4$). The invariant is proven with the same arguments used in Appendix A, although the lemma

$$\langle \min\ l :: Server.\tau(l)\rangle = \beta\ \textbf{unless}\ \langle \min\ l :: Server.\tau(l)\rangle > \beta$$

is proven differently. The only statement that may affect the lemma is the first in *Min(k)*, which is enabled only when *Min(k)* and *Server* are co-located, as

stated by (REV$_9$). Execution of this statement, together with invariant ($\mathcal{I}_3$) and (REV$_5$), establishes

$$\min(P(l).t, \langle \min\ m : m \neq l :: Server.\tau(m) \rangle) \geq \beta$$

as in the CS solution, which satisfies the lemma.

PROOF OF (REV$_2$). The invariant holds as the conjunction of (REV$_1$), (REV$_7$), and (REV$_9$).

PROOF OF (REV$_7$). The invariant holds initially, from the program text. The first statement and the third statement in *Min(k)* are enabled only when the LHS of the invariant does not hold. Execution of the first statement, due to (REV$_{10}$), keeps the LHS false, and thus preserves the invariant. Execution of the third statement does not affect the LHS at all, and thus the invariant is preserved straightforwardly. Migration statements may affect the invariant. Execution of the second statement in the **Interactions** section establishes $\mathsf{def}(Min(k).\tau)$ due to the **engage** clause in the third interaction and to (REV$_6$). Finally, execution of the last interaction establishes the invariant. The migration statement is executed synchronously with the statement voiding $Server.q(i)$: disengagement of this variable and $Min(k).q$ causes the latter to remain undefined after migration. At the same time, the **disengage** clause in the third interaction voids the value of $Min(k).\tau$.

PROOF OF (REV$_8$). The invariant holds initially, since *P(k)* and *Min(k)* are initially co-located and hence the LHS of the implication is false. Then, there are only two statements which may affect the invariant. If the LHS holds before execution of the first statement in *P(k)*, then the statement is disabled and the invariant preserved. If the LHS holds before execution of the third statement in *P(k)*, invariant (REV$_{12}$) guarantees that the statement is disabled, preserving the invariant.

PROOF OF (REV$_{11}$). The invariant holds initially, from the program text. The first statement in *Min(k)* may affect the invariant. However, if *Min(k)* is not co-located with *Server*, invariant (REV$_9$) guarantees that this statements is disabled. On the other hand, if the two components are co-located, execution of the statement assigns a defined value to $Min(i).q$ and then to $Server.q(i)$, being sharing in place through the fourth interaction. Migration statements may also affect the invariant. The second statement in the **Interactions** section actually establishes the invariant through the **engage** clause of the fourth interaction. The last statement in the system establishes the invariant as well, since it assigns an undefined value to $Server.q(i)$, which is still shared with $Min(i).q$, and causes migration of *Min(k)*; since no disengagement clause is specified for the fourth interaction, the two variables retain an undefined value.

REFERENCES

AMADIO, R. 1997. An Asynchronous Model of Locality, Failure, and Process Mobility. In D. GARLAN AND D. L. MÉTAYER Eds., *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, Volume 1282 of *LNCS* (Berlin, Germany, Sept. 1997), pp. 374–391. Springer.

BALDI, M., GAI, S., AND PICCO, G. P. 1997. Exploiting Code Mobility in Decentralized and Flexible Network Management. In K. ROTHERMEL AND R. POPESCU-ZELETIN Eds.,

Mobile Agents: 1st International Workshop MA '97, Volume 1219 of *LNCS* (Apr. 1997), pp. 13–26. Springer.

CARDELLI, L. AND GORDON, A. 2000. Mobile Ambients. *Theoretical Computer Science 240*, 1.

CARZANIGA, A., PICCO, G., AND VIGNA, G. 1997. Designing Distributed Applications with Mobile Code Paradigms. In R. TAYLOR Ed., *Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97)* (1997), pp. 22–32. ACM Press.

CHANDY, K. AND MISRA, J. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Software Engineering 5*, 5 (Sept.), 440–452.

CHANDY, K. AND MISRA, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley.

CUGOLA, G., GHEZZI, C., PICCO, G., AND VIGNA, G. 1997. Analyzing Mobile Code Languages. In J. VITEK AND C. TSCHUDIN Eds., *Mobile Object Systems: Towards the Programmable Internet*, Volume 1222 of *LNCS*, pp. 93–111. Springer.

FOURNET, C. ET AL. 1996. A Calculus of Mobile Agents. In *Proc. of the 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, Volume 1119 of *LNCS* (Pisa, Italy, Aug. 1996), pp. 406–421. Springer.

FUGGETTA, A., PICCO, G., AND VIGNA, G. 1998. Understanding Code Mobility. *IEEE Transactions on Software Engineering 24*, 5.

GARLAN, D. AND MÉTAYER, D. L. Eds. 1997. *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, Volume 1282 of *LNCS* (Berlin, Germany, Sept. 1997). Springer.

GRAY, R. 1995. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents* (Baltimore, Md., Dec. 1995).

GRAY, R. ET AL. 1997. Mobile agents for mobile computing. In *Proc. of the 2nd Aizu Int. Symp. on Parallel Algorithms/Architectures Synthesis* (Fukushima, Japan, Mar. 1997).

HARRISON, C., CHESS, D., AND KERSHENBAUM, A. 1997. Mobile Agents: Are they a good idea? In J. VITEK AND C. TSCHUDIN Eds., *Mobile Object Systems: Towards the Programmable Internet*, Volume 1222 of *LNCS*, pp. 25–47. Springer. Also available as IBM Technical Report.

JOHANSEN, D., VAN RENESSE, R., AND SCHNEIDER, F. 1995. An Introduction to the TACOMA Distributed System - Version 1.0. Technical Report 95-23 (June), Dept. of Computer Science, University of Tromsø and Cornell University, Tromsø, Norway.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems 6*, 2 (Feb.), 109–133.

KINIRY, J. AND ZIMMERMAN, D. 1997. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing 1*, 4 (July-Aug.), 21–30.

KNABE, F. 1995. *Language Support for Mobile Agents*. Ph. D. thesis, Carnegie Mellon University, Pittsburgh, Pa. Also available as Carnagie Mellon School of Computer Science Tech. Rep. CMU-CS-95-223 and European Computer Industry Centre Tech. Rep. ECRC-95-36.

LANGE, D. AND OSHIMA, M. 1998. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley.

MAGIC, G. 1995. *Telescript Language Reference*. General Magic.

DE NICOLA, R., FERRARI, G., AND PUGLIESE, R. 1998. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering 24*, 5.

MCCANN, P. AND ROMAN, G.-C. 1997. Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts. In D. GARLAN AND D. L. MÉTAYER Eds., *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, Volume 1282 of *LNCS* (Berlin, Germany, Sept. 1997), pp. 338–354. Springer.

MCCANN, P. AND ROMAN, G.-C. 1998. Compositional Programming Abstractions for Mobile Computing. *IEEE Transactions on Software Engineering 24*, 2.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A Calculus for Mobile Processes I. *Information and Computation 100*, 1 (Sept.), 1–40.

Object Management Group. 1995. *CORBA: Architecture and Specification*. Object Management Group.

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

Picco, G., Roman, G.-C., and McCann, P. 1997. Expressing Code Mobility in Mobile UNITY. In M. Jazayeri and H. Schauer Eds., *Proc. of the 6th European Software Engineering Conf. held jointly with the 5th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '97)*, Volume 1301 of *LNCS* (Zurich, Switzerland, Sept. 1997), pp. 500–518. Springer.

Roman, G.-C., McCann, P., and Plun, J. 1997. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Trans. on Software Engineering and Methodology 6*, 3 (July), 250–282.

Stamos, J. and Gifford, D. 1990. Remote Evaluation. *ACM Trans. on Programming Languages and Systems 12*, 4 (Oct.), 537–565.

Strasser, M., Baumann, J., and Hohl, F. 1996. Mole—A Java Based Mobile Agent System. In M. Mühlaüser Ed., *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conf. on Object-Oriented Programming ECOOP'96* (July 1996), pp. 327–334. dpunkt.

Sun Microsystems. 1995. *The Java Language Specification*. Sun Microsystems.

Tschudin, C. 1994. *An Introduction to the* `MO` *Messenger Language*. Univ. of Geneva, Switzerland.

Vitek, J. and Tschudin, C. Eds. 1997. *Mobile Object Systems: Towards the Programmable Internet*, Volume 1222 of *LNCS*. Springer.

White, J. 1996. Telescript Technology: Mobile Agents. In J. Bradshaw Ed., *Software Agents*. AAAI Press/MIT Press.