Coordination and Mobility

Gruia-Catalin Roman[*], Amy L. Murphy[*], Gian Pietro Picco[‡]

April 28, 2000

Abstract

Mobility entails the study of systems in which components change location, in a voluntary or involuntary manner, and move across a space that may be defined to be either logical or physical. Coordination is concerned with what happens when two or more components come in contact with each other. In this paper we put forth a working definition of coordination, we construct arguments that demonstrate that coordination is central to understanding mobility, we explore the intellectual richness of the notion of coordination, and we consider the practical implications of coordination-centered system design strategies. We develop these ideas in two steps. First, we analyze the different dimensions that govern the definition of a coordination strategy for mobility: the choice of an appropriate unit of mobility, the treatment of space and its implications on the way we think about mobility, and the manner in which contextual changes induced by component movement are perceived and managed. Then, we explore mechanisms that enable us to model and reason about coordination of mobile components, and to make it available to software developers in the form of middleware. Three very different models of mobility (Mobile UNITY, CODEWEAVE, and LIME) are used as principal sources for illustration purposes.

1 Introduction

The term mobility has grown to the point that it encompasses a very broad range of meanings and technological subcultures spanning from formal theoretical models to wireless transmission protocols and standards. In this paper we seek to explore the richness of this concept in a manner that transcends the distinctions between physical and logical mobility but to the exclusion of concerns having to do with the communication technology and low level protocols. As such, the kinds of issues we are about to consider are likely to be of interest primarily to researchers concerned with models, languages, and middleware.

[*]Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA. E-mail: {roman, alm}@cs.wustl.edu.

[‡]Dipartimento di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci 32, 20133 Milano, Italy. E-mail: picco@elet.polimi.it.

While logical mobility involves the movement of code (in all its forms) among hosts, physical mobility entails the movement of hosts (of all sorts and sizes) in the real world. Future generations of mobile systems are likely to include both forms of mobility even though so far they have been treated as distinct and have been studied by different research communities. For now, logical mobility is viewed as offering designers a new set of conceptual and programming tools that seek to exploit the opportunities made available by the distributed computing infrastructure deployed over the last decade. Physical mobility, on the other hand, is assumed to be closely tied into the next evolutionary step in the development of the worldwide communication infrastructure, the extension of wireline networks to fluid clouds of wireless devices. This new environment presents designers with enormous challenges not the least among them being the seamless integration of programming models with both the wired and wireless platforms. It is for this reason that a unified treatment of physical and logical mobility is important at this time. Clearly, at the most basic level they share a view of the world in which components move through space (be it logical or physical) and interact with each other in accordance with the rules governing some particular model of mobility. Yet there is a wide range of variations among models and systems involving mobility. It is our contention that variability is essentially the result of differing decisions with respect to a narrow set of issues: unit of mobility, properties of space, context definition, and the coordination constructs that facilitate component interactions.

Our coordination perspective on the subject is fostered by the recognition that building open systems requires designers to adopt a new viewpoint. In defining a component, the designer must minimize any dependencies on the mechanics of data communication. Coordination [9] accomplishes this by separating the functionality of individual components from the manner in which interactions take place. Components may be logical or physical. Laptops and PDAs equipped with wireless connectivity may travel around a building or across the country alongside their owners. Code fragments may move from host to host across both wireline and wireless networks and even among various points in the structure of a single program. Mobile agents move of their own volition while class definitions are downloaded on demand. Components may be simple or complex structures. The types of components involved relate closely with the kind of spatial domain one must consider.

Logical level interactions among components are made possible by specialized constructs which specify the coordination services to be provided and define the interface to such services, when necessary. In the most extreme case, components are totally oblivious to the coordination process. As movement occurs and coordination takes place, components find themselves having access to a changing array of resources, making new acquaintances, losing contact with each other, etc. In other words, the dynamics of mobility place each component in a continuously changing contextual setting. The very definition of the context and the manner in which components handle change vary with the coordination constructs available to them.

Coordination is the thread that ties together the dominant themes in mobility today and the principal subject of this paper. Three specific models will be used as primary sources for illustrative examples. Mobile Unity is a formal model of mobility that provides a notation and proof logic for describing a broad range of mobile systems, paradigms, and coordination constructs. CodeWeave is a model that assumes a very fine grained perspective on code mobility, at the level of single variables and statements. Finally, Lime is a middleware designed to provide transient and transitive sharing of tuplespaces among agents and hosts.

The remainder of the paper is organized as follows. Section 2 discusses issues that are central to understanding the relation between coordination concepts and mobility: Section 2.1 considers choices regarding the unit of mobility; Section 2.2 explores variations in the definition of space and discusses issues having to do with movement; Section 2.3 analyzes the definition of context and the manner in which it is perceived by the mobile components. Section 3 is concerned with the kinds of coordination constructs one may encounter. The emphasis is not so much on offering a survey of coordination languages and models as is on examining the range of options that may be considered and the mechanics of how context is constructed in three different models: Mobile Unity, CodeWeave, and Lime. Conclusions appear in Section 4.

2 Mobility Issues

Basic to the notion of mobility is the requirement that there be some entities that perform the moves, some space within which movement takes place, and rules that govern motion. Equally important is the way in which mobile entities perceive the environment that surrounds them, the changes in such environment, or the way it is actively being explored by them. This section considers each of these issues in turn. We start by discussing possible choices regarding the unit of mobility. We follow with a discussion of physical and logical space and its relation to the units of mobility. Finally, we conclude the section by examining the notion of context, i.e., the worldview of the individual units.

2.1 Unit of Mobility

The entities that move through space can be physical or logical, simple or composite. Physical components are generally referred to as mobile hosts and they can vary in size from a laptop to a PDA or other wearable device. The trend towards miniaturization is likely to lead to the emergence of minuscule devices and smart sensors that can be attached to people's clothing, movable structures and everyday objects. Robots of all sizes also serve as hosts. They can move through space in a purposeful way under their own volition, can communicate with each other, and are able to orchestrate activities none of them could accomplish in isolation. It is generally expected that the number of mobile hosts connected to the Internet will rapidly reach into the millions. At the same time the size of ad hoc networks [3] (i.e., wireless networks having no wireline support) is also expected to reach into the hundreds. Some networks may be relatively stable, e.g., an assembly line or an office involving tens of devices that use wireless communication to interact with each other. Other situations are much more dynamic and unpredictable, e.g., cars on a highway exchanging information among themselves and with stationary data kiosks. Finally, there are situations in which the underlying application is a complex enterprise mobile in nature, e.g., an emergency response team consisting of many vehicles and individuals. What makes this kind of situation different is the heterogeneity of the computing platforms involved and the level of logistics interwoven into the behavior of the overall system. We rarely think of physically mobile units as coming together and breaking apart, yet such applications clearly point to a future in which the physical units of mobility will vary greatly in size and capabilities and will engage in interactions that go beyond just relative movement and wireless communication to include docking and complex physical reconfiguration.

In the realm of logical mobility, the unit of mobility naturally shifts from being a host to being a code fragment. It should be noted from the onset that the term “code fragment” implies some sort of syntactic element recognized by the programming language. This is because of the necessity to reference it in the code itself. Programs, agents, procedures, functions, classes, and objects are reasonable choices frequently encountered in mobile code languages and agent systems. Of course, language syntax recognizes also much finer grained elements such as statements, expressions, and variables. They can also be moved and such level of mobility it is referred to in this paper and others as fine-grained mobility. Later in the paper we will show examples of units of mobility associated with different syntactic elements and of various granularity. Another point of differentiation among mobile units is the state of execution at the time the movement takes place. If the code fragment is relocated by creating a fresh copy at the destination point or prior to the start of its execution, the movement involves pure code and it is often referred to as weak mobility. By contrast, strong mobility entails the movement of code being executed, i.e., the execution state is relocated along with the code thus allowing it to continue running even after the move. Of course, since the relocation may result in changes to some of the program bindings the actual state may have components that are no longer the same as before the move. Associated with strong mobility is usually the ability to exercise control over the movement both with respect to timing and destination although one can easily envision systems in which the fixed infrastructure has total control over who moves and when.

Is it possible to envision units of mobility that have no syntactic correspondence? While we can argue that the basic unit of mobility must have a syntactic correspondent, we can also see the possibility of creating, through a composition process, computational environments which no longer have any relation to program syntax. A simple example might be a swarm of agents that, once bound to each other, may become a new entity endowed with the ability to move and to restructure itself. Our own work on fine-grained mobility comes very close to this view of mobility, as it will become evident in a later section.

2.2 Space

Current work on mobility has made very little effort to investigate and formalize the concept of space and its properties. To date, physical components can vary greatly in size and capabilities but, most of them perceive location as a pair of coordinates on the earth surface provided by some GPS service. This is useful in terms of being able to exhibit location-dependent behaviors in PDAs and laptops. Robots can go one step further and actually control their position in space. Such capability, however, brings with it the added complexity of a space that is no longer continuous and homogeneous due to the presence of walls and other limitations to movement. Space acquires structure and maybe even semantics. Space can actually change over time as components move relative to each other. A robot moving out of a doorway may all of a sudden open new possibilities of movement for the others. Moreover, locations can be relative rather than absolute and knowledge of the space may be limited to those areas explored to date. This may become important when knowledge is shared among components since in the absence of a global reference system it may be difficult to reconcile the different views acquired by different components.

Can we think of physical space in a new light? We believe that we can and we must rethink our treatment of physical space. Space can have structure and the structure may be essential to making it possible to address important application needs. Consider, for instance, the problem of delivering messages among a group of mobile components that move through space and communicate only when in immediate proximity to each other. Can we guarantee message delivery? In general, this is impossible because one cannot be sure that two specific components ever meet. Yet, if we assume that the components move back and forth along a single line (a train track), the problem is easily solved. Many situations are amenable to similar treatment. Of course these kinds of issues can be handled in the application by superposing a suitable interpretation on top of what amounts to a primitive space. But, a more formal and higher level treatment may give us the tools to reason about such situations and to provide programmers with high level interfaces that reduce the complexity of the development effort.

Since much of the communication among mobile components entails the use of wireless transmission, space and distance metrics are playing growing roles in the way one thinks about the relation among components. Cellular networks provide a prime example of structure being imposed onto the physical space. The space is divided into regular patches with a base station supporting all the communication needs of the components in that specific area. Transitions among cells entail special handoff protocols, i.e., communication behavior is tied to space structure. In the case of ad hoc networks, the space lacks structure but the distance metric is important because communication can take place only when components are within range. If ad hoc routing is available, i.e., mobile components serve as mobile routers as well, the space acquires structure dynamically in response to the very presence of the mobile components. This structure is being used in new kinds of protocols that factor relative location information into their decision process, e.g., discovery protocols interested in determining what neighbors are within a certain distance of a given component. More sophisticated approaches are likely to take advantage of both distance and velocity by predicting future component locations. The density of components in a given region of space may also be exploited in the design of certain systems. In natural situations, distance may be affected by terrain topology. Even in the air, winds may be viewed as altering the properties of space, thus affecting the predictive equations. In artificial environments apparent distances may be altered by the presence of communication resources such as wireless bridges that offer extended connectivity through wireline networks.

To the best of our knowledge no systems to date provide what we would call a non-trivial view of physical space. The situation is quite different when it comes to considering logical space in programming languages and models. One can encounter a great deal of variability with respect to the choice of unit of mobility, the perception of logical space, and even the kind of movement that is permitted.

In Mobile Unity [12], for instance, a program consists of a set of statements and the set of local variables being read and modified. A program becomes mobile through the addition of a distinguished location variable. Changes to the location variable represent movement and programs change location always as a whole. Since movement is reduced to value assignment, reasoning about mobility can be handled by employing the standard Unity [2] logic. By contrast, CodeWeave [11] shares a common formal foundation with Mobile Unity but adopts a fine-grained paradigm. Single variables and individual statements can be moved from one location to another and can be injected in, aggregated into and extracted from programs created on the fly. Large-grained aggregates can be manipulated in exactly the same manner. The mobility and restructuring of programs is not unique to CodeWeave. Algebraic models accomplish the same thing by allowing for the movement of processes, be they elementary or composite. In Mobile Ambients [1], for instance, rules are provided for moving processes from one context to another regardless of the complexity of the process involved in the move (the reader should think of context as being a location).

While our mental image of mobility is usually associated with some form of autonomy this cannot be assumed in all cases. In mobile agent systems the decision to move may indeed rest with the agent itself. Classes, however, are loaded on demand. One can also envision systems in which agents are pushed along (forced to move) when their services are no longer needed or when a need arises at some other site. When modeling physical systems, devices are carried around from one location to another without having any say. The motion is actually induced by outside forces, which may need to be modeled when reasoning about the resulting system.

In our discussion so far, we casually mentioned location without actually considering what it might be. In the context of logical mobility, location is often equated with a host in the network. Things are a little more complicated than this because the ability to execute code presupposes the existence of an appropriate computing environment. Nevertheless, treating space as a graph with vertices representing locations and edges constraining movement is a reasonable model. However, this is only one among many models one encounters in the literature and there are many more that are likely to be studied in the future. Consider, for instance, a program residing on a host and its structure which, for discussion purposes, we assume to be hierarchical in nature. Each node in its tree structure can be viewed as a location. The very structure of the programs provides a notion of space. Code fragments may move to locations in the program and may even extend the program. Thus, the notion of space induced by the program may be used to support mobility and as the basis for redefining the space itself. This is precisely the view adopted by CodeWeave, which recognizes both hosts and structured programs residing on hosts. In Mobile Ambients, the model is structured in terms of nested processes that define administrative domains. Mobility is constrained by the structure of the model while motion alters that very structure, i.e., the definition of space. MobiS [10] too is a model that offers a hierarchical structure with mobility restricted among parent child locations consisting of tuple spaces. Finally, in Mobile Unity the space is left completely outside the model. Spatial properties may be used, however, in reasoning about the behavior of the system.

This latter strategy allows one to explore a broad range of spaces having different formal properties, purely abstract constructions or models of physical reality. In general, the ability to unify logical and physical views of mobility is useful in the analysis of mobile systems and also as the basis for developing new models. Moreover, if such efforts result in a practical integration of logical and physical mobility, a wide range of novel applications can be contemplated. Lime [18] is one such attempt to integration. Mobile agents reside on mobile hosts that can form ad hoc networks when in proximity of each other. When this happens, the agents appear to be sharing a common data environment (tuple space) and have the opportunity to jump from one host to another.

With the advent of mobility, space is fast becoming the new research frontier. Our treatment of space impacts our ability to analyze systems and shapes the models and languages we develop. The assumptions we make about the structure of space and the mobility profile of the components that inhabit it have profound effects on the kinds of protocols and algorithms we develop. Problems specific to mobility are often impossible to solve unless proper restrictions are imposed. A more precise and formal evaluation of space holds the promise for significant intellectual and practical advances in our treatment of mobility.

2.3 Context Management

As components move through space, their relation to other components and to fixed resources changes over time. Of course, even if a component is stationary, other components may move relative to it. The notion of context relates to the way in which a component perceives the presence of other components and available stationary resources at each point in time. While location plays an important role in determining the current context, it is not the sole controlling factor. Similar components at the same location are likely to see very distinct contexts. Two components, for instance, may not have the same access rights at that particular location because one is local while the other is a visitor. Their respective needs may also be distinct thus forcing the components to look for different things among the locally available resources. Even more interesting is the fact that components may obey different binding rules, i.e., ways to associate names to resources. Depending on the nature of the mobile system, when a component leaves a site existing bindings may be severed permanently, may be disabled temporarily and restored upon return, or may continue to be preserved in spite of the location change. The first option is most common in settings involving physical mobility while the third is readily implemented when logical mobility takes place across connected sites, as in the case of the Internet.

A purely local context (i.e., involving a single location) is often favored because it appears to be easier to maintain and implement. This is only partially true because even a local context may involve transparent coordination among multiple hosts. In the ad hoc mobility setting, for instance, the maintenance of a local context is very complex. Components need to discover each other and to negotiate the extent to which they are willing to collaborate among themselves. Keeping track of who else is in immediate proximity and which resources they are willing to provide or seek to use is not an easy task. The level of complexity is affected by the assumptions one can make about disconnection patterns, the relative speeds of components, and the reliability of both components and wireless links. The degree of consistency demanded by the application is another major factor. The trend is towards weak versions of consistency but they may not be acceptable in all situations. Even in the case of logical mobility, a component arriving at a site is required to establish new bindings. From the point of view of the component, this entails a resource discovery process and possible negotiations.

A distributed context (e.g., one that refers to distant hosts) entails all the complexities associated with a local one plus a lot more. At a minimum, the component navigating across the network must remember the identity of the resources it needs from different locations (e.g., IP addresses) and the network must provide the ability to support communication with the resources. However, this is adequate only if the resources are essentially passive, i.e., respond to requests but they do not initiate any. If the relation is such that resources can actually initiate communication, the complexity rises dramatically. The network must support delivery to components that move in space, be they agents or nomadic devices that rely on base station support. Mobile IP [16] is one protocol that provides this kind of service. Of course, the ultimate challenge is to allow mobile components to send messages to each other while offering delivery guarantees. Recently, we proposed several algorithms that support this kind of communication in a nomadic setting [14, 15]. Systems that provide notification services are another example in which resources need to contact mobile components. It should be noted, however, that providing messaging support is only the first step towards supporting the context management needs of a mobile computing system.

While context management can be left in the hands of the individual components, this is not a strategy likely to lead to rapid software development. It is more reasonable to provide middleware that enforces a certain clean conceptual view while offering the right tools for managing contextual changes. We draw a distinction here between maintaining the context and responding to contextual changes, a topic we will return to at the end of this section. Regarding context maintenance, two issues seem to be particularly important in differentiating among various software support strategies, whether they are implemented as part of middleware, agent systems, or mobile code language: the level of support provided by the underlying runtime system and the conceptual model enforced. Regarding the former, the two extremes seem to be making context maintenance explicitly the responsibility of the component or achieving full transparency. When an agent changes location on its own and arrives at a new site, it may be required to decide: how visible it is necessary to be to others by engaging in some registration process; which old acquaintances should be preserved; what services to register with the local site for availability to other agents; what resources it needs to discover and access at the new location; etc. All these activities, even if supported by some sort of mobility middleware, provide flexibility but also place significant demands on the component designer.

Much of our own work has centered on making context maintenance fully transparent. In such cases, the conceptual model that underlies the basis for the context maintenance is of paramount importance. The designer relies on its understanding to generate correct code. One of the models supported by Mobile UNITY is the notion of transparent transient sharing of program variables. Component code is simply written under the assumption that variables may undergo spontaneous value changes in response to the arrival and departure of other components in the vicinity or due to modifications made by them. What variables are shared and under what conditions is specified using a declarative notation—its operational semantics are ultimately reduced to coordination actions associated with statements in the basic Mobile UNITY notation. The condition for sharing variables can be arbitrary but it is usually related to relative positions among components, e.g., at the same location or within radio contact.

A more complex example of transparent context management involves the use of global virtual data structures. In a virtual memory system the application program perceives the memory space to be larger than the physical reality and the support system takes upon itself the responsibility of maintaining this illusion in a seamless manner. Similarly, a global virtual data structure creates the appearance that the individual mobile unit has access to shared data despite the presence of mobility. Consider, for instance, a graph and two very distinct settings, one involving agents and the other ad hoc networks. In one case, the graph is stored in a distributed fashion across the nodes of a fixed network. Agents move from node to node like crawling ants carrying data from vertex to vertex. Each agent is aware of the graph and of the presence of other agents colocated at the same vertex. In the second case, the graph is distributed among the mobile hosts but only that portion of the graph that is connected and stored among hosts within communication range is accessible to the application. Hosts can trade sections of the graph as long as such changes cannot have any effect on hosts that are out of contact. In both cases, behavior analysis is carried out by reasoning about the global structure but all actions are local. Furthermore, the actions involving the structure are specific to it. LIME is one system that follows this strategy by employing a tuple space partitioned among both hosts and agents.

Context changes can be induced not only by movement (through associated changes in data and resource availability) but also due to quality of service considerations, e.g., variations in bandwidth and delay. Regardless of their source, context changes are always important to the application and mechanisms for responding to such changes are needed. The most commonly used strategy is to provide an event notification mechanism. A predefined set of events is made available to the application, which can register in turn appropriate responses for specific events. A more general approach is to furnish the application with a general event notification mechanism and allow it to define both the set of events of interest and the choice of responses. A very different alternative involves the notion of reactive statements. As used in Mobile UNITY and LIME, the execution of reactions is not triggered by events, rather by specific state properties. Once activated, they continue to execute at high priority for as long as the condition persists. We will return to this topic in the next section that covers coordination constructs and the manner in which they contribute to context definition and maintenance.

3 Coordination Constructs

Coordination is a programming paradigm that seeks to separate the definition of components from the mechanics of interaction. In traditional models of concurrency, processes communicate with each other via messages or shared variables. The code of each component is explicit about the use of communication primitives and the components are very much aware of each other's presence. Actually, communication fails when one of the parties is missing. By contrast, coordination approaches promote a certain level of decoupling among processes and the code is usually less explicit about mechanics of the interactions. Ideally, the interactions are defined totally outside the component's code. Linda [6] is generally credited with bringing coordination to the attention of the programming community. By using a globally shared tuple space, Linda made temporal and spatial decoupling a reality in parallel programming and simplified the programming task by providing just three basic operations for accessing the tuple space. Interactions among processes were brought up to a new level of abstraction and the programming task was made simpler. Reasonable implementations of the tuple space made the approach effective.

Our concern, however, is not with coordination in general but with the role it can play in simplifying the task of developing mobile applications. Mobility can benefit from a coordination perspective because decoupling enhances one's ability to cope with open systems. At the same time, mobility adds a new and challenging element to coordination, the dynamic changes taking place as components move through space. Some properties that may be desirable in general become even more important in the mobile setting. Promoting a coordination style that is totally transparent to the participating components, for instance, may enhance decoupling and increase the ability to interact with components previously unknown. Other interesting properties are specific to mobility. The notion of transient interactions is a direct result of the fact that components move relative to each other going in and out of communication range. The concept of transitive interactions surfaces when one needs to consider establishing group-level interactions out of pairwise communications. At the other extreme, logical mobility in the presence of full server access across global networks leads to almost unreal modes of distant interaction where components move from location to location while preserving the ability to access resources as if they were local. In general, space becomes a major factor in formulating coordination issues. In physical mobility, the distance between components can become a barrier to wireless communication thus making interactions conditional on relative positions of components. Even in logical mobility, components may be limited to interacting only when present at a common location. Finally, spatial properties are commonly combined with quality of service and security considerations to define the nature of the coordination process.

In the remainder of this section we discuss three models corresponding to three distinct modes of coordination. While they involve both physical and logical mobility, separately and in a fully integrated fashion, the distinctions among the three models are most striking when we examine the way coordination is used to address similar problems in very different contexts. Mobile UNITY is illustrative of what one might call an *active* coordination strategy. Coordination is specified operationally and bridges (in a mostly transparent manner) the states of components when they are found in specific relations to each other. LIME is representative for a *passive* coordination style very similar to Linda but adjusted to the realities of ad hoc mobility. Transiently and transitively shared tuple spaces provide the coordination medium but programs need to make explicit use of tuple space operations to gain access the tuples located on hosts within proximity. Finally, CODEWEAVE illustrates one of the more exotic applications of coordination. Components (codes fragments and aggregates) are provided with primitives that facilitate code mobility. An operational specification defines the meaning of these operations in terms of simpler coordination primitives. One might call this style of coordination *constructivist*. The three examples exhibit a great degree of similitude at the conceptual level but also variability in the range of coordination constructs being employed. We view this to be indicative of two complementary facts. On one hand, one can build a common foundation and use it to examine the way coordination is used in mobility. Mobile UNITY seems to have many of the features required to accomplish this. On the other hand, mobility covers such a vast expanse of possibilities that many distinct models are likely to emerge. They will provide the conceptual foundation for software systems (mostly middleware) designed to support the development of mobile applications.

3.1 Mobile Unity

Mobile Unity [12] proposes a new notation and underlying formal model supporting specification of and reasoning about mobile systems. The approach is based on the Unity [2] model of concurrent computation. Its notation is extended with constructs for expressing transient interactions among components in the presence of movement and reconfiguration.

Unity was conceived as a vehicle for the study of distributing computing, and defined a minimalist model for specifying and reasoning about such systems. The key elements of the Unity model are the concepts of variable and conditional multiple assignment statement. Programs are simply sets of assignment statements that execute atomically and are selected for execution in a weakly fair manner. Multiple programs can be composed through the *union* operator. The result is a new system that consists of the union of all the program variables and the union of all the assignment statements. Variables with the same name are assumed to be identical, i.e., they reference the same memory location. Our interest in mobility forced us to reexamine the Unity model with the following goals in mind: to provide for a strong degree of program decoupling, to model movement and disconnection, and to offer high-level programming abstractions for expressing the transient nature of interactions in a mobile setting.

One key design decision in the development of Mobile Unity was the choice of the program as the unit of mobility. This is a natural choice for Unity because it allows for simple functional decomposition and composition (e.g., through program union and the use of similarly named variables). Therefore, the first major aspect of a Mobile Unity system description is the specification of the individual mobile components, a standard Unity program for each. However, unlike standard Unity, Mobile Unity seeks to foster a highly decoupled style of programming by requiring the namespaces of the programs to be disjoint. This allows each program to operate without interference from the other programs it is composed with. As shown later, coordination among components is allowed, but it is separated from normal processing. Additionally, each program is augmented with a distinguished location variable. This variable may correspond to latitude and longitude for a physically mobile component, or it may be a network or memory address for a mobile agent. The specific definition is left intentionally out of the model. However, by making location a part of the program specification, it can be manipulated from within the program to allow a program to control its own location. Furthermore, the location of the individual components can be used in reasoning about the system behavior. Actually, by reducing movement to value assignment, the standard Unity proof logic may be used to verify program properties despite the presence of mobility.

The second major aspect of a Mobile Unity specification is the **Interactions** section which defines all interactions among components. Because the namespaces of the programs are disjoint, no statement within a program can reference a variable in another program. The **Interactions** section is the only place where variables from multiple programs can be addressed. In this manner, the definition of coordination is separated from the definition of standard processing.

In our work we have shown that only a very small set of primitive constructs is needed to build a wide range of high-level coordination constructs and models. These primitives include asynchronous value transfer, statement inhibition, and reactive statements. Statement inhibition restricts the execution of a statement in one program based on the state of another program. Reactive statements are enabled by programmer specified global conditions and continue to execute at high priority until they no longer cause any state changes. The standard Unity proof logic has been extended to incorporate the new Mobile Unity primitives, but the underlying proof logic remains unchanged. Using these three primitives and a minor technical change to the basic Unity notation, Mobile Unity allowed us to define a number of interesting coordination constructs including: variables which are shared in a transient and transitive manner based on the relative positions of the mobile programs; statements that are synchronized in a transient and transitive manner according to a variety of synchronization rules; clock synchronization with and without drift; etc.

Reactive statements provide a mechanism for extending the effect of individual assignment statements with an arbitrary terminating computation. This construct allows us to simulate the effects of the interrupt processing mechanisms that are designed to react immediately to certain state changes. The construct is particularly useful when its guard involves the relative locations of components. The result is the execution of an additional state transition in response to a new connection or disconnection.

The standard Unity model for shared variables is static. In Mobile Unity transient variable sharing is implemented using the reactive statements. High level constructs are provided for symmetric and asymmetric update of variables throughout the period during which they are shared, for establishing a single common value when a new sharing relation is established (engagement), and for defining the values resulting from the disengagement of variables as components move away. Transient statement synchronization is defined using variable sharing as a building block.

The constructs provided by Mobile Unity have been put to test in the specification and verification of Mobile IP [13] and of a variety of mobile code paradigms including code-on-demand, remote evaluation, and mobile agents [19]. In the next two subsections we will present two other uses of Mobile Unity. One involves its application to fine-grained code mobility and the other relates to defining the formal foundation for mobility middleware involving transient sharing of tuple spaces.

3.2 CodeWeave

The core concepts of Mobile Unity are geared towards the mobility of components. Programs may represent mobile hosts moving across space, or mobile agents roaming network hosts. Nevertheless, this coarse-grained view of mobility is indeed limited, as evidenced by state of the art technology. In the realm of logical mobility, mobile agent technology still awaits for massive exploitation in the design of distributed applications, while finer-grained forms of mobility, often collectively referred to as mobile code, already found their way into recent proposals for distributed middleware. For instance, Java Remote Method Invocation (RMI) exploits the dynamic class loading capabilities of the Java language to allow, on both client and server, on-the-fly retrieval of stub and application classes that are needed to support a remote method invocation. The capability of relocating portion of the code and of the state of a mobile unit, rather than always moving all the constituents together, brings additional flexibility in the design of distributed applications.

CodeWeave [11] is a specialization of Mobile Unity conceived for modeling fine-grained mobility. CodeWeave retains the operational model underlying Mobile Unity, but allows the designer to specify migration of the constituents of a Mobile Unity program. The unit of mobility in CodeWeave can be as small as a single Unity statement or variable. The former is referred to as a *code unit*, while the latter is called a *data unit*. Units take part into the general system behavior only when they are part of a *process*. Processes can be organized in hierarchies, and the containment relation constrains the ambit of visibility of a unit. Thus, processes are the unit of scoping and execution. Processes and units exist at a given location, which may be a process or a site. Units that reside on a site do not belong to any process, they represent available resources that may be shared among the co-located processes.

CodeWeave provides primitives for moving (or cloning) units. Migration of units across hosts may represent the relocation of a class or an object in a distributed middleware. Migration of a unit from a host into a process may represent dynamic linking or deserialization mechanisms. Relocation of state and behavior is not the only dimension relevant to logical mobility. As discussed in [5], logical mobility is often exploited to gain access to a site's shared resources, hence the management of bindings to shared resources upon migration is a key issue. CodeWeave provides a notion of *reference* that enables processes to access a unit without explicitly (and exclusively) containing it.

The same primitives discussed thus far actually apply to processes. For instance, a **move** operation applied to a process causes its migration together with all its constituents. Hence, the fine-grained model put forth by CodeWeave subsumes, rather than replace, the coarse-grained perspective where the units of execution and of mobility coincide. For instance, mobile agents are still modeled naturally by using the process abstraction. Furthermore, the ability to represent nested processes enables the modeling of complex structures built by sites, places, and agents like those introduced in Telescript [20], similarly to what Mobile Ambients or MobiS provide.

CodeWeave represents fine-grained mobility by relying completely on the semantics of Mobile Unity. The units of mobility of CodeWeave, i.e., statements and variables, are reinterpreted as Mobile Unity programs, whose movement and sharing is ruled by statements in the **Interactions** section, according to the semantics specified in CodeWeave. For instance, the movement of a process along with all its constituents is actually reduced to the movement of a Mobile Unity program representing the CodeWeave process, followed by a set of reactions that migrate the Mobile UNITY programs representing its constituents in the same atomic step.

Several reflections can be made about the model fostered by CODEWEAVE. The fact that the model's semantics is reduced completely in terms of Mobile UNITY is indicative of the fact that the constructs of Mobile UNITY effectively capture the essence of mobile interactions. However, the ability to model and reason about mobile systems at a level of abstraction that is closer to the domain opens up new possibilities.

On one hand, the availability of fine-grained constructs fosters a design style where mobile agents are represented at increasing levels of refinement. Existing systems typically do not allow a mobile agent to move along with all its code, due to performance reasons. Different systems employ different strategies, ranging from a completely static code relocation strategy that separates at configuration time the code that must be carried by the agent from the one that remains on the source host (like in Aglets [8]), to completely dynamic forms under the control of the programmer (like in μCODE [17]). Our approach allows modeling of a mobile agent application at different levels of detail. One can start with a high level description in terms of processes representing the mobile agents and continue to refine this view using the fine-grained constructs that specify precisely which constituents of an agent are allowed to move with it and which are not.

On the other hand, the perspective put forth by CODEWEAVE sheds a new light on coordination. Coordination no longer involves just communication about parties, which at most may be mobile and migrate to achieve local coordination. When fine-grained logical mobility is part of the picture, components are malleable and open to changes that may occur as part of the coordination protocol. The cooperative behavior of components is no longer determined only by the information exchanged during coordination, but also by new behaviors that can be exchanged as program fragments, and dynamically become part of a component. Examples include the ability to exploit new coordination primitives and coordination protocols downloaded on the fly while the computation is being executed.

This view may even lead to new models of computation. Insofar we always assumed that moving a statement or a variable in a CODEWEAVE system does not necessarily imply achieving this also in the implementation language. We thought of it allowing us to model the movement of a unit of mobility (e.g., a Java object or class) that is finer grained than the unit of execution (e.g., a Java thread). Nevertheless, the ability to move a single statement or variable in a real programming language is an intriguing possibility, one that may be realized by exploiting the new generation of scripting languages. Some researchers [4] proposed schemes where XML tags are migrated and dynamically "plugged in" an XML script already executing at the destination. This scheme may amplify the improvement in flexibility and customizability brought in by mobile code. An even wilder scenario is the one where it is possible not only to add or substitute programming language statements that conform to the semantics of the language, but also to extend such semantics dynamically by migrating statements along with a representation of the semantics of the constructs they use. As proposed at a recent conference [7], this could open up an economic model where the concept of software component includes not only application code or libraries, but even the very constituents of a programming language.

Another interesting opportunity is grounded in the binding mechanism that rules execution of units within a process. In our model, a statement can actually execute only if it is within a process, and if all of its variables are bound to corresponding data units. This represents the intuitive notion that a program executes within a process only if the code is there and memory has been allocated for its variables. In languages that provide remote dynamic linking, it is always a code fragment that gets dynamically downloaded into a running program. However, the symmetry between data and code units in our model suggests a complementary approach where not only the code gets dynamically downloaded, but also the data. Thus, for instance, much like a class loader is invoked to resolve the name of a class during execution of a program, similarly an "object loader" could be exploited to bring an object to be co-located with the program and thus enable resumption of the computation. Another, even wilder, follow-up on this idea is an alternative computation model where code and data are not necessarily brought together to enable a program to proceed execution, rather it is the program itself (or a whole swarm of them, to enhance probability of success) that migrates and proceeds to execute based on the set of components currently bound to it. This way, a program is like an empty "pallet" wandering on the net and occasionally performing some computation based on the pieces that fill its holes at a given point.

Verification may be considered under a new light as well. CodeWeave inherits the temporal logic of Mobile Unity and thus, besides enabling reasoning about the location of mobile programs, it also allows reasoning about the location of their constituents. Hence, verification can be exploited not only to prove the overall correctness of the system, but also to optimize the placement of the constituents of its mobile components, placing them only on the nodes where they will be needed. For instance, a mobile agent could be written in such a way that it does not need to carry with it a given class, because a formal proof has been developed to guarantee that, for the given system in a given state, the class will be already present at destination. Clearly, this approach potentially enables bandwidth and storage savings, and is particularly amenable for use in environments where resources are scarce, e.g., wireless computing with PDAs.

3.3 Lime

Lime [18] takes a pragmatic step toward the development of applications for mobility by describing a model for coordination among mobile components which frees the application programmer from direct concern with many complexities of the environment while still providing a powerful programming paradigm. The model itself is formed by adapting the well-known Linda coordination model to accommodate the essential features of mobility, and the paradigm is presented to the application programmer in the form of middleware.

The fundamental properties of the Linda model are a globally accessible, static tuple space and a set of processes which interact by writing to and reading from the shared tuple space. None of the processes need to co-exist in either time or space to coordinate, and because access to the data is based on pattern matching, a process need not know the identity of the other processes in order to successfully interact. This decoupled style of interaction is of great utility in a mobile setting where the parties involved in communication change dynamically due to migration or shifts in connectivity patterns. On the other hand, the decoupled nature of Linda is made possible by the global accessibility and persistence of the tuple space.

When mobility is considered, especially the ad hoc model of physical mobility, no predefined, static, global context can exist for the computation. The current context is defined by transient communities of mobile components. The idea underlying LIME is to maintain a global *virtual* tuple space, physically distributing the contents of this structure among the mobile units. As connectivity among components changes, different projections of the global virtual tuple space are accessible to each component. From a more local perspective, each mobile unit is responsible for a portion of the global state (i.e., one partition of the global virtual tuple space). When two or more components are within communication range, the contents of their tuple spaces are shared transiently and transparently.

To the application programmer, all interactions with the shared data space occur via local accesses to a component known as the *interface tuple space*, or ITS. The ITS effectively provides a window into the global virtual tuple space which expands as mobile components come within range and contracts as connectivity among components is lost. Transient sharing of the contents of the tuple spaces within the ITS constitutes a very powerful coordination abstraction, as it provides a mobile unit with the illusion of a local tuple space that contains all the tuples belonging to the members of the currently connected community, without the need to explicitly know their identities. Additionally, because access to the ITS is entirely local, the application programmer can issue local operations, while reasoning about their effect in the global context.

The operations provided on the ITS are identical to the basic Linda operations, and by default operate over the currently accessible data. This permits a context dependent style of interaction by fulfilling queries based on the current connectivity state. Additionally, LIME extends the basic Linda operations to allow data to be placed with a specific mobile component as connectivity allows, and also to query a specific projection of the tuple space based on the mobile component responsible for storing that data. Such interactions enable a programming style in which the application programmer can access data based on a specific, known location. Thus, LIME provides both location-transparent and location-aware styles of data access, in order to support applications requiring different styles of programming.

Thus far, the description of the mobile components has been left intentionally abstract, however the LIME model provides a unique integration of physical and logical mobility. Basically, the logically mobile components, or mobile agents, form the active computational units of the Lime system, and it is their responsibility to hold individual partitions of the tuple space. As an agent migrates, it carries its portion of the tuple space with it as part of its state. The integration with physical mobility comes because these logically mobile components must reside on physical hosts that move through space and connect with one another based on the distance between them. We assume that when two agents reside on the same host, they are able to communicate, thus forming a *host level tuple space* containing the tuples of the agents located on that host. Similarly, when hosts are within communication range, the host level tuple spaces are shared to form a *federated tuple space*. To emphasize the power of the abstraction provided by the model, we note again that the basic Linda operations function over the federated tuple space. This makes coordination visible only as a changing set of tuples over which operations are evaluated, protecting the programmer from the changes inherent in the mobile environment.

As previously noted, the mobile environment is highly dynamic, and it is often desirable for applications to react to changes in the environment. In Mobile Unity, this led to the development and integration of the reactive programming model. Linda already supports one model to react to the state of the system, namely a "pull" model in which a process blocks until a tuple matching a query appears in the tuple space. Because Lime provides the same primitive operations as Linda, this functionality is present in the mobile environment, however Lime also makes available Mobile Unity's reactive model of programming, enabling a "push" style of coordination. In other words, instead of a process waiting to pull information from the tuple space when it is written, the tuple space itself is charged with pushing information about the state to a process or executing a piece of code which has been registered. Because the reactions are controlled from within the Lime system, a higher degree of atomicity between the discovery of a matching tuple and the execution of the user's registered reaction can be guaranteed. Pragmatic considerations about how to enforce atomicity guarantees in the distributed setting forced us to incorporate in Lime two kinds of reactions, weak and strong. They provide differing degrees of atomicity and are subject to different applicability constraints motivated by the need to provide an effective implementation. In our experience, the combination of this form of reactive programming with the transiently shared tuple space provides a programming abstraction that is extremely powerful and useful. Thus, for instance, programmers are able to specify once and for all that a reaction must be executed whenever a given condition takes place in any point of the federated tuple space, and possibly even within the context of a component that was unknown at registration time.

The reactive mechanism of Lime has proven useful not only for reacting to changes in the data state of applications, but also for reacting to changes in the system context, specifically the arrival and departure of agents and hosts. This hints at another feature of Lime, namely the exposure of the system context to the application programmer. Specifically the system context consists of the mobile hosts that are within communication range and the mobile agents contained on each of those hosts. It is also possible to augment this basic information with other system characteristics such as the available bandwidth between a pair of hosts or the resources available within a host. We approach the system context as the dual to the data context, making it available as a read only tuple space maintained wholly from within the Lime system and accessible with the same primitives as all other transiently shared Lime tuple spaces. In other words, the application programmer can query and react to this Lime system tuple space in the same manner it operates over the data tuple space, with the exception that the system tuple space cannot be written to.

Lime has been successful primarily on two fronts. First, a formal specification of the Lime model has been written in Mobile Unity, providing both clear semantics for the operations, and serving as an example of the practical use of Mobile Unity as the specification language for a mobile middleware system. With this semantic definition in hand, it is possible to prove properties about the overall Lime system. Examples include the completion of the engagement protocol when a new host joins the Lime system and properties of applications which have been specified in Mobile Unity using the Lime constructs. Second, an implementation of Lime exists as a Java package available for distribution. It has been used to develop a variety of mobile applications ranging from collaborative work scenarios to spatial games. The simplicity of the Linda coordination model and its natural adaptation to transiently shared tuple spaces leads to a shallow learning curve for the Lime programmer and to ease of implementation.

4 Conclusions

Mobility is an area rich in research opportunities, both intellectually and pragmatically. It demands a new way of thinking and requires design strategies that are distinct from the traditional distributed computing. Our own research into mobility spans a broad range of issues, from formal models to middleware development. Regardless of the direction we explored, three issues emerged as central to our investigation in each case: the choice of the unit of mobility, the definition of space, and the manner in which context is maintained and perceived. In this paper we tried to develop these themes by exploring the range of options one can envision and by contrasting what is possible with what we actually employed in three specific models. Another important message of this paper is the notion that coordination played a key role both in our ability to provide a clean formal treatment of mobility and in our attempt to simplify the development of mobile applications. On one hand, a coordination perspective facilitated an abstract and modular treatment of mobility. On the other hand, a coordination-centered design of the middleware made it possible to offer the programmer a simpler conceptual model of interaction among mobile units and to delegate to the runtime support system much of the effort associated with maintaining and updating the context visible to the individual units. Because coordination promotes global thinking and local action, it is ideally suited for addressing the needs of mobility in all its forms.

References

[1] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000. To appear.

[2] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] M. Corson, J. Macker, and G. Cinciarone. Internet-Based Mobile Ad Hoc Networking. *IEEE Internet Computing*, 3(4), July 1999.

[4] W. Emmerich, C. Mascolo, and A. Finkelstein. Incremental Code Mobility with XML. Technical Report 99-95, University College London, October 1999. Submitted for publication.

[5] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.

[6] D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.

[7] G. Glass. Agents and Internet Component Technology. Invited talk at *3rd Int. Conf. on Autonomous Agents (Agents'99)*, May 1999.

[8] D. B. Lange and M. Oshima, editors. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[9] T.M. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

[10] C. Mascolo. MobiS: A Specification Language for Mobile Systems. In P. Ciancarini and A. Wolf, editors, *Proceedings of the 3rd Int. Conf. on Coordination Languages and Models (COORDINATION)*, volume 1594 of *LNCS*, pages 37–52. Springer, April 1999.

[11] C. Mascolo, G.P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility. In *Proc. of the 7th European Software Engineering Conf. held jointly with the 7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '99)*, LNCS, Toulouse (France), September 1999. Springer.

[12] P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2), 1998.

[13] P.J. McCann and G-.C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 8(2), April 1999.

[14] A.L. Murphy and G.P. Picco. Reliable Communication for Highly Mobile Agents. In *Proc. of the 1st Int. Symp. on Agent Systems and Applications and 3rd Int. Symp. on Mobile Agents (ASA/MA '99)*, pages 141–150, Palm Springs, CA, USA, October 1999. IEEE Computer Society.

[15] A.L. Murphy, G.-C. Roman, and G. Varghese. Tracking Mobile Units for Dependable Message Delivery. Technical Report WUCS-99-30, Washington University, Dept. of Computer Science, St. Louis, MO, USA, December 1999.

[16] C. Perkins. IP Mobility Support. RFC 2002, IETF Network Working Group, 1996.

[17] G.P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. of the 2nd Int. Workshop on Mobile Agents*, LNCS 1477. Springer, 1998.

[18] G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.

[19] G.P. Picco, G.-C. Roman, and P.J. McCann. Expressing Code Mobility in Mobile UNITY. In M. Jazayeri and H. Schauer, editors, *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*, pages 500–518, Zurich, Switzerland, September 1997. Springer.

[20] J.E. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents.* AAAI Press/MIT Press, 1996.