Transient data sharing among mobile programs

Authors: Jerome Plun and Gruia-Catalin Roman

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent computing units, a decoupled computing style, frequent disconnections, continuous position changes, and the location-dependent nature of the behavior and communication patterns present designers with unprecedented challenges in the areas of modularity and dependability. This paper describes a modular approach to specifying and reasoning about mobile computing. Its novelty rests with the notion of allowing transient (location-dependent) data sharing among programs which move in space. The notation is a direct extension of that used in UNITY and reasoning about mobile computations relies on the UNITY proof logic. Several examples explain the notation, demonstrate the modularity of the approach, and illustrate the verification methodology. An electronic-mail distribution problem is presented in some detail.

... Read complete abstract on page 2.
Transient data sharing among mobile programs

**Complete Abstract:**

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent computing units, a decoupled computing style, frequent disconnections, continuous position changes, and the location-dependent nature of the behavior and communication patterns present designers with unprecedented challenges in the areas of modularity and dependability. This paper describes a modular approach to specifying and reasoning about mobile computing. Its novelty rests with the notion of allowing transient (location-dependent) data sharing among programs which move in space. The notation is a direct extension of that used in UNITY and reasoning about mobile computations relies on the UNITY proof logic. Several examples explain the notation, demonstrate the modularity of the approach, and illustrate the verification methodology. An electronic-mail distribution problem is presented in some detail.
Transient data sharing among mobile programs

Jerome Plun
Gruia-Catalin Roman

WUCS-95-01

January 1995

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

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent computing units, a decoupled computing style, frequent disconnections, continuous position changes, and the location-dependent nature of the behavior and communication patterns of the individual components render obsolete exiting models and present designers with unprecedented challenges regarding modularity and dependability. This paper describes a modular approach to specifying and reasoning about of mobile computing. Its novelty rests with the notion of allowing transient (location-dependent) data sharing among programs which move in space. The notation is a direct extension of that used in UNITY and reasoning about mobile computations relies on the UNITY proof logic. Several examples explain the notation, demonstrate the modularity of the approach, and illustrate the verification methodology. An electronic-mail distribution problem is presented in some detail.

Correspondence: All communications regarding this paper should be addressed to

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

office: (314) 935-6190
secretary: (314) 935-6160
fax: (314) 935-7302

roman@cs.wustl.edu
1. Introduction

The trend toward mobility started with the user being allowed to gain access to computer resources from distant places via remote connections. As the miniaturization of electronics progressed, the shrinking size of computing devices made it possible for users to carry computers and even use them as they are being moved. The term "mobile computing" came to characterize computations and environments in which computers are free to change location while still able to communicate with one another when they so desire. Mobile computing devices can range from simple emitters ("active badges") and receivers ("beepers", GPS positioning) to full-fledged personal computers (laptop, notebook, subnotebook, palmtop), and include cellular phones and other similar communication devices. While some of these devices can perform self-contained computations, the vast majority of them require communication with a support environment that is designed to accommodate their mobile nature.

Due to the need to allow for unrestricted movement, wireless communication is used to implement both point-to-point connections and broadcasts. In the former case, as the two endpoints of a communication link move around, the components which implement the link might vary in a transparent manner—in the cellular phone system, for instance, a portable phone is passed from one cell handler to another with no discernible impact on the conversation being carried out. In the latter case, the broadcast source often has no knowledge of who the recipients might be and listeners may tune in while a broadcast is already in progress. Other design considerations specific to mobile computing are weight limitations, resilience to shocks, energy efficiency (to prolong the use of battery-powered mobile devices between charges), disconnection handling (the mobile component may be switched off or may move outside of communication range), bandwidth variations (depending on the mode of communication used and the distance between communicating nodes), etc. For an extensive coverage of hardware and software implications of mobility the reader is referred to several recent publications [3, 4, 6, 8, 9, 10, 12].

Regardless of the application, mobility presents software developers with significant new challenges. Fueled by the desire to leave untouched as many existing applications as possible, some designs try to hide the mobility of nodes in the communication protocols so as to allow "host migration" while retaining existing services such as TCP/IP or OSI [1, 7, 11]. Even if this becomes the solution of choice and application programs are written in a location-independent manner, the software subsystems that are charged with building this illusion of ubiquitous access to distant resources will have to take into account both mobility and the changing communication capabilities of individual devices. At the same time, system developers, under pressure to deliver increasingly more dependable software, will require appropriate formal models to specify and reason about location-dependent behaviors and interactions. The main thesis of this paper is that existing models can be adapted to meet these new demands. To argue this point, we describe an extension to the UNITY model [2] in which 1) mobility is expressed by augmenting the program state by a location attribute whose change in value is used to represent motion, and 2) transient location-dependent communication between mobile components is captured by a new form of data sharing.

We focus our attention on communication via shared-variables because it is an important and widely studied distributed computing paradigm whose significance spans both theory and practice. A formal understanding of the fundamental implications of mobility on data sharing is likely to lead to useful new abstractions which, in turn, may affect both programming languages and software engineering methodologies. From a practical perspective, sharing storage among mobile components is not uncommon. A laptop computer, for instance, may be attached to a docking workstation. This physical connection allows the sharing of resources such as video memory, hard disk, and even the random access memory when a co-processor is present in the docking station. Although current docking stations can not be used alone—the controlling CPU is provided by the laptop—there are no significant technical difficulties in ultimately allowing the docking of slave or peer computers. This would provide the means to combine the computing power of multiple components while maintaining individual usability when disconnected. Of course, as the number of mobile components in a single office increases, the notion of physical connections becomes less feasible but logical data sharing will continue to be provided via appropriate protocols that rely on wireless transmission using the radio and infrared bands.

In its most basic form, the shared data model allows multiple processes to access common areas of memory. In UNITY, for instance, any variable name which appears in more than one program is by definition a reference to a shared variable. Other notations employ various scope rules and allow memory addresses to be communicated among programs via pointers. While the methods for specifying which variables are shared and which are not vary, the net effect is the same. A shared variable is associated with a fixed memory address throughout its existence and all processes that share it access one and the same storage object. In the context of this paper, we will employ the term persistent to designate this class of shared data models, we will reserve the term common variable to refer to shared variables as used in these models, and we will use the private qualifier to talk about the non-shared variables. This terminology is needed in order to facilitate precise comparisons between prior work and the model introduced in this paper.
In the persistent shared data model (Fig. 1), a common variable is always engaged in the (potential) transfer of information among processes which share it while a private variable is always disengaged from any such activity. From the vantage point of a given process, a common variable can be read/written both locally and by others while a private one may be read/written only locally. These restrictions are not compatible with the assumptions we make about mobile systems in which access to shared data may be a function of the current location of a process and of what else is present at that precise location or in the general vicinity. A variable may be engaged (in a sharing relation) only when certain conditions hold and may be shared among various groups of processes at different times. Moreover, a variable belonging to one mobile process having its own local memory may be required to cooperate with variables belonging to other processes in the neighborhood in creating temporarily the illusion of a single atomically-accessed shared variable. We call variables that exhibit this kind of cooperative behavior coop variables. They act as private variables when disengaged and as shared variables when engaged—for practical as well as technical reasons explained later, they may be temporarily inaccessible while transitioning between the engaged and disengaged states or in order to accommodate changes in the set of processes involved in data sharing.

A model that employs this kind of highly-dynamic and location-dependent data sharing is the subject of this paper. We use the term transient shared data model to refer to it. The new model is motivated directly by the needs of mobile computing and generalizes the shared data paradigm by introducing the notions of coop and reflective variables; the latter allows one process to see data made available by other processes in the region (e.g., by listening to radio broadcasts)—the variables made visible to other processes are considered exposed. (Fig. 1 provides a high-level comparison between the persistent and transient shared data models.) Shared variables continuously change the sharing relations in which they engage depending upon the current state of the computation. Formally, the circumstances under which processes either merge their variables into one or make available their variables to each other are specified by sharing conditions present in interaction statements. For reasons of modularity, the interaction statements are completely separate from the code describing the individual mobile processes.

<table>
<thead>
<tr>
<th>Shared Data Model</th>
<th>Variable Type</th>
<th>Interaction States and Access Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Disengaged</td>
</tr>
<tr>
<td>Persistent</td>
<td>Private</td>
<td>r/w</td>
</tr>
<tr>
<td></td>
<td>Common</td>
<td>r/w</td>
</tr>
<tr>
<td>Transient</td>
<td>Private</td>
<td>r/w</td>
</tr>
<tr>
<td></td>
<td>Common</td>
<td>r/w</td>
</tr>
<tr>
<td></td>
<td>Exposed</td>
<td>r/w</td>
</tr>
<tr>
<td></td>
<td>Reflective</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td>Coop</td>
<td>r/w</td>
</tr>
</tbody>
</table>

Fig. 1. A comparison between persistent and transient shared data models. The capability r/w indicates local read/write access, R/W indicates read/write access by another process, and X indicates temporary lack of access.

Blank entries signify impossible states.

In the remainder of this paper we show how transient data sharing can be expressed in a highly modular fashion using an augmented UNITY notation and can be reasoned about using the UNITY logic. Section 2 reviews briefly the UNITY syntax and proof logic. A notation for expressing the concepts of location and movement is introduced in Section 3. Section 4 focuses on reflective variables while Section 5 on coop variables. The two sections discuss the chosen notation, consider two related mobile computing schemas (observation and cooperation, respectively), provide illustrative examples, and offer sample proof outlines for simple mobile computations. Section 6 considers a more interesting problem, the delivery of electronic mail in a highly-mobile environment. Several solutions are presented and modular ways of expressing them in the proposed notation are considered. Brief concluding remarks appear in Section 7.

2. A model of computing

Before discussing the mobility of programs, we first review the UNITY [2] notation which is used to express the computation taking place within the mobile elements of a system and the UNITY proof logic which will
be applied to reasoning about mobile computations. We favor UNITY due to its minimalist perspective and because of our broad experience with the model. The general structure of a UNITY program is shown in Fig. 2 which contains a program designed to simulate a random movement within the confines of a two-dimensional grid with \(X\) columns and \(Y\) rows. \(X\) and \(Y\) are assumed to be positive integers, comparisons among grid coordinates are based on lexicographical order, and coordinate summation is defined as component-wise addition. The program has a name, RandomWalk, and four optional sections—throughout the paper we assume that all program names are unique whenever program composition is discussed. The declare section contains Pascal-like declarations for all the variables used in the program, i.e., a characterization of the state space of the program. The always section names expressions which can be used like macros in the body of the program. The initially section constrains the initial values of the program variables. In this paper we take some liberties regarding the always and initially sections by permitting the use of arbitrary predicates in addition to non circular sets of equations—as in the proof logic, all free variables are assumed to be universally quantified. The assign section consists of a set of conditional multiple-assignment statements. The execution of a program starts in a state satisfying the constraints imposed by the initially section. At each step one of the statements is executed. The selection of the statements is arbitrary but fair, i.e., each statement is selected infinitely often in an infinite computation. All computations are infinite.

```
Program RandomWalk
declare
  Position : pair of integer
always
  North, South, East, West = (0,1), (0,-1), (1,0), (-1,0)
  Valid(p) = (1,1) ≤ p ≤ (X,Y)
initially
  Position = (1,1)
assign
  Position := Position+North  if Valid(Position+North)
  Position := Position+East   if Valid(Position+East)
  Position := Position+South  if Valid(Position+South)
  Position := Position+West   if Valid(Position+West)
end
```

Fig. 2. UNITY program structure. A program that simulates random movement on a grid.

The UNITY logic is a specialization of temporal logic. Safety properties specify that certain actions (i.e., state transitions) are not possible, while progress properties specify that certain actions will eventually take place. In UNITY, the basic safety property is the unless relation. The formula \(p \text{ unless } q\) states that if the program enters a state in which the predicate \(p\) is true and \(q\) is false, every program action will either preserve \(p\) or establish the predicate \(q\). All other safety properties, such as invariant (written as \(\text{inv.}\)), constant, and stable are defined in terms of unless. The basic progress properties are ensures and leads-to (written \(\rightarrow\)). The formula \(p \text{ ensures } q\) states that \(p\) unless \(q\) holds and, in addition, there is some statement which establishes \(q\), a statement the program is guaranteed to execute in a bounded number of steps. Similarly, the formula \(p \rightarrow q\) states that if the program enters a state in which \(p\) is true, the program will eventually enter a state in which \(q\) holds, although \(p\) need not remain true until \(q\) becomes true.

For illustration purposes, here are several properties of the RandomWalk program (all free variables are universally quantified by convention):

\[
\text{Position}=(x,y) \text{ unless } (\exists x',y': (0,0)<(x',y')-(x,y)<(1,1) :: \text{Position}=(x',y')) \dagger
\]

\[
\text{inv. Valid(Position)}
\]

\[
\text{Position}=(x,y) \text{ ensures Position}=(x,y)
\]

\dagger The three-part notation \(\{\text{op quantified:variables range_constraint :: expression}\}\) used throughout the text is defined as follows: The variables from \text{quantified:variables} take on all possible values permitted by \text{range_constraint}. If \text{range_constraint} 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 \text{expression} producing a multiset of values to which \text{op} is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies \text{range_constraint}, the value of the three-part expression is the identity element for \text{op}, e.g., \text{true} if \text{op} is \text{V}.
The first property states that the new position can be precisely one space north, south, east, or west; the second property states that position is always within the grid; and the last property states that there is at least one statement which, once executed, is guaranteed to cause a position change.

3. A Notation for Mobile Components

The Mobility Factor. The RandomWalk program introduced earlier simulates the movement of a point across a grid but it is not a mobile program even if the program itself resides on a computer which is located on some moving platform (van, ship, or satellite) or migrates from processor to processor across a local or national network in search of free cycles. Only when a reasonable characterization of the program state must include its location as a component can a program be considered mobile. Cellular communication, wireless LAN, and satellite services enable portable computers of varying sizes and capabilities to interact with other computers and with services they provide while changing location. Although the goal is to present the end user with the illusion of a ubiquitous computing environment which hides the effects of movement, the programs charged with accomplishing this task must be cognizant of their own and other’s relative locations and must be sensitive to the consequences of movement. Delivering electronic mail to a mobile unit requires finding out its location first. Movement from one cell to the next involves passing on communication responsibility from one machine to the next. Entering a new location associated with a different administrative domain may alter drastically resource availability. In all these cases, reasoning about the program’s behavior must take into account its location—even if it is only in order to prove that movement is actually transparent to the end user.

By analogy with real-time modeling where time is most often introduced as a distinguished variable not accessible to the programmer, one can capture mobility by introducing a distinguished location variable for each individual process. Restrictions on how such a location variable is accessed and updated would have to reflect the characteristics of the computation. In a cellular network, for instance, the location of the mobile units is determined by the car or person carrying the computer but constrained to movements from one cell to a neighboring one (as long as the connection is on). The verification of any hand-off algorithm must rely on this assumption. Protocols involved in reestablishing connectivity at the time a mobile computer is powered up may have to assume that initial locations are arbitrary. In some applications a program may have to know its own location while not in others. In the former case the location is directly accessible by the program while in the latter the location plays a role only in reasoning about the computation. As automation technology advances it is also conceivable that certain programs may have the ability to actually control the movement of their carrier which may be, for example, a robot doing deliveries in an office building. In this case, movement is no longer under the control of the environment but planned by the program which could request future data delivery at specific locations to be reached along the movement path. In this section we propose a notation for specifying individual mobile programs and location accessing schemas corresponding to some of the situations described in this paragraph.

Our goal is to construct a model of mobile computing on top of UNITY while minimizing changes to the basic notation and proof logic. Whenever possible, the new notation is merely cosmetic in nature and can be mechanically reduced to basic UNITY. In this section we introduce only the notation for specifying computations consisting of a single mobile program—later sections will deal with interactions among mobile programs. Because of this (temporarily) limited perspective, the notation is illustrated on a program atypical for current mobile computing: a robot control program whose code appears in Fig. 3. The robot starts at an arbitrary valid location in some room configuration and moves in the current direction until it encounters a wall. At that point, it changes

Program Robot at A

```
Program Robot at A
    declare Direction : pair of integer
    always
        North, South, East, West = (0,1), (0,-1), (1,0), (-1,0)
        ![ Valid(p) = p is within the confines of the room and is not a wall ]
    initially
        (1) Valid(A)
    assign
        (2) A := A + Direction  if Valid(A + Direction)
        (3) Direction := North if  ¬Valid(A + Direction) ∧ Direction = East
        (4) Direction := East   if  ¬Valid(A + Direction) ∧ Direction = South
        (5) Direction := South if  ¬Valid(A + Direction) ∧ Direction = West
        (6) Direction := West   if  ¬Valid(A + Direction) ∧ Direction = North
    end
```

Fig. 3. A program which exercises control over its own location.

The robot control program has complete knowledge of the room in which it moves.
Fig. 4. Successive attempts to move result in counter-clockwise rotations until the robot faces a valid location.

direction in a counter-clock manner until movement becomes possible again, as shown in Fig. 4.

Notation wise, the distinguished variable holding the location is declared implicitly alongside the name of
the program, as in program Robot at A. Because the program name is assumed to be unique, its variables acquire
unique names if prepended by the program name as in Robot.Direction. This convention will become important in
later sections where communication among mobile programs is discussed. The notation Robot.A is never used since
each program has its own location variable. The predicate Robot@p (read Robot at location p) is provided instead.
Its meaning being given by

\[ \text{Robot}@p \equiv \text{Robot}A=p \]

In addition we provide shorthand notion for predicates that involve both variable values and the program location

\[ \text{Robot}x@p=\alpha \equiv \text{Robot}@p \land \text{Robot}x=\alpha \]

\[ (\text{Robot}x=\alpha)@p \equiv \text{Robot}@p \land \text{Robot}x=\alpha \]

\[ x@p=\alpha \equiv \{ \exists R : R \text{ is a program} :: R@p \land R.x=\alpha \} \]

and allow the notation for location to distribute across logical expressions that do not involve the @ symbol, e.g.,

\[ (x>0 \land y=2)@p = (x>0)@p \land (y=2)@p \]

Implicit in our notational conventions is the notion that a program and its variables are co-located and move as a
single unit.

In the Robot program the type of A was left unspecified. Throughout the paper we assume the existence of
a global declaration for the spatial context in which the programs move. In most cases, we assume a two-
dimensional grid of points. Allowing programs to have diverse perceptions of the basic structure of space adds
unnecessary complexity and would obscure the issues we want to consider in this paper. A related issue is the range
of moves one is willing to accommodate. Should programs be permitted to jump to any point in the grid or should
they move one position at a time? As suggested earlier, jumps may be needed to simulate the movement of
computers which are temporarily turned off but most algorithms will require movement to follow the grid structure.
Even so, this latter condition is too weak to be useful when one considers interactions between a mobile program
and other programs meant to model stationary cell managers covering some section of the space. In the absence of
any timing considerations (typical of UNITY and most other models of concurrency) a program could move along
the grid an arbitrary distance without being "observed" by anybody with the result being the appearance of an
arbitrary jump. For these reasons, we prefer to permit arbitrary movements in the model and assume that individual
applications will structure the access to the location variable in a manner that is meaningful for each specific mobile
computation. More specifically, we advocate the use of program schemas to restrict the access and manipulation of
the location variable.

Consider, for instance, the case of a mobile program which is neither aware of its own location nor able to
influence it. Under such circumstances, the location variable cannot be referenced anywhere in the body of the
program but it is changed somehow by the environment. What makes such a program mobile is the fact that its
interactions with other programs are affected by its location in space. We expect this schema to be very common
among many applications where mobility is provided by a person or vehicle that carries a computer which, in turn,
interacts via wireless communication with any fixed service network which happens to be present in its vicinity.
A second useful schema involves programs that react to the location at which they find themselves and change behavior accordingly. An intelligent badge, for instance, may provide less information when passing through a low security zone or finding itself outdoors. For programs in this category the location variable becomes a read-only variable. By contrast, the *Robot* program falls in yet another class of programs which have full and exclusive control over their position. Their schema allows the location variable to be both tested and modified like any other local variable, subject to some formally stated constraints. Examples of relevant constraints are the need to follow the grid structure, prohibitions against sharing the space location with other programs, or restrictions against traveling through areas that represent walls or other obstacles.

The last schema we consider here is one in which the program has only implicit control over its location, i.e., the value of the location variable is computed in the always section and may or may not be examined by the program. This represents the situation in which the state of the program determines its location. A robot placed in the middle of a room, for instance, may be able to keep accurate information of its movement relative to the initial position but has no knowledge of its actual position—its task may actually be to learn where it is and how to get out.

**Beginning to reason about mobility.** Because location is simply just another program variable, reasoning about the kinds of programs discussed so far follows the typical UNITY paradigm with the proof being extricated directly from the program text. Returning to the *Robot* program, for instance, we can prove that (in a finite-sized closed room) the robot travels only through open space, maintains the same direction up to the point when a wall is reached, and is guaranteed to eventually reach one of the walls:

\[ \text{inv. Robot@p \land Valid(p)} \]
\[ \text{Direction=δ unless } (\exists p : \text{Robot@p \land \neg Valid(p+δ)} \}
\[ \text{Robot@(x,y) \land Robot.Direction=δ} \]
\[ \implies (\exists x',y' : x'=x \lor y'=y :: \text{Robot@(x',y') \land Valid((x',y')+δ)} \}

which can be rewritten as

(a) \[ \text{inv. Valid(Δ)} \]

(b) \[ \text{Direction=δ unless \neg Valid(Δ+δ)} \]

(c) \[ \Delta=(x,y) \land \text{Direction=δ} \implies (\exists x',y' : x'=x \lor y'=y :: \Delta=(x',y') \land \neg \text{Valid(Δ+δ)} \}

To prove an invariant property such as (a), one needs to show that the property holds initially, which is true for (a) from line (1) in Fig. 3, and that the property is maintained after the execution of any statement of the program. From the text of the program, only statement (2) modifies the location of the robot and thus might invalidate the invariant. But since the statement modifies the location only if the new location is valid, the property holds.

As mentioned in Section 2, a \( p \) unless \( q \) property like (b) is proven by showing that any statement executed in a state satisfying \( p \land \neg q \) either maintains \( p \) or establishes \( q \). Given that in the case of (b) \( p \land \neg q \) is equivalent to \( \text{Direction=δ} \land \neg \text{Valid(Δ+δ)} \), only statement (2) can have an effect on the state. Executing (2) does not change the direction, so \( p \) is maintained as long as \( q \rightarrow \neg \text{Valid(Δ+δ)} \)—is false. Moreover, \( q \) is established as a result of executing (2) when the robot is one step away from the wall it faces.

Finally in order to prove (c), one needs to define some positive metric that decreases while the robot moves until a wall is reached. We define the metric as the current distance between the robot and the wall it faces. Thus one can immediately state that when the distance is 0, the robot can not move any further in the current direction, i.e.:

\[ \Delta=(x,y) \land \text{Direction=δ} \land \text{Distance=0} \implies \neg \text{Valid(Δ+δ)} \]

Moreover, as long as the distance is greater than 0, only statement (2) can be executed, with the result being a decrease in the distance to the facing wall, i.e.:

\[ \Delta=(x,y) \land \text{Direction=δ} \land \text{Distance=Δ>0} \textbf{ ensures } \Delta=(x,y+δ) \land \text{Direction=δ} \land \text{Distance=Δ-1} \]

By induction one obtains property (c).

**Communication.** Having defined the concept of location for a component, we now turn to the communication among these components. The communication architecture [6] represent a mix of the current cellular technology differentiates among mobile units and static hosts; communication from and to a mobile unit transits through whichever static host is currently handling the mobile unit. In contrast, our view of the communication architecture is that any two nodes can, in principle, communicate with each other. Restrictions on the
communication pattern are assumed to originate with the computation itself. This view is more general and brings into sharper focus those issues which are fundamental to mobile computing. In the following sections, we describe two communication mechanisms which provide nodes with the ability to exchange data.

4. Observation Schema and Reflective Variables

Motivation. Passive observation takes place when a program may read but not write variables belonging to other programs. In mobile computing there are many instances of applications that fit this pattern. Wireless communication, for instance, allows one program to send a message to any number of recipients within an appropriate receiving space without the sender being aware of who the recipients are. If the broadcasts are repeated often enough, the overall logical effect is that of posting information for anyone to read. The Global Positioning System (GPS [5]) is one such system. It consists of a network of geostationary satellites which regularly emit a reference signal that can be used by a receiver to determine its location on Earth by triangulation. In this case, the receiver is aware of the location of the sender and the use of an observation schema provides a good representation of the interaction taking place. Other examples include broadcasting facilities for stock prices, weather information, movie showing times, etc.

Interactions. One simple way to capture observation of non-local data is to permit one program to refer to others' variables in the right hand-side of assignment statements and definitions. The dynamic data sharing requirement is met by allowing the effect of the assignment statements to be determined by the location of the non-local variables they reference. While this approach provides a simple and direct observation mechanism, the need to know which other mobile programs participate in the computation is unacceptable. We desire to write mobile programs which may be reused in multiple settings by merely changing the definition of their interactions with the other components of the system. This goal does not affect the nature of the interaction pattern but it does influence the notation we choose to specify it.

Notation. A first step towards decoupling a program's code from its interactions is to localize all access to non-local variables to the always section whose definition-like style enforces the observation schema. The second step is to move all such definitions outside the code of the mobile program. For this purpose we associate with each system, i.e., community of mobile programs, an Interactions section which relates the values of certain always-variables of one program to variables in other programs. Syntactically, we extend the always section with the introduction of a shared keyword to indicate that the corresponding variable provides read-only access to non-local data. An active badge, for instance, may have to know if the person that carries it is needed some place but not how this information is actually obtained in a particular environment. A declaration such as

always Needed : shared boolean

suffices to actually write the badge's code. Since the definition is deferred to the Interactions section, only the type of the variable must be given to ensure compatibility with the assignment statements in which it is used. The definitions appearing in the Interactions section represent the data sharing specification for the respective system. They are referred to as interactions statements or simply as interactions. For instance, in some system, Needed may be bound to a ceiling sensor being in the Paging mode as in the interaction below where Badge and Sensor(s) are mobile unit names

\[
\text{Interactions}
\]

\[
\text{Badge.Needed = true if } (\exists s, \lambda: \text{Badge@}\lambda \land \text{Sensor(s)@}\lambda :: \text{Sensor(s).Paging})
\]

\[
\sim \text{false otherwise}
\]

We refer to read-only shared variables appearing in the always section as reflective variables—they reflect some aspect of the state of the environment. In this paper any local variable can be potentially read by other programs. Local variables which do not appear in the Interactions section are, by implication, private. All other local variables are considered exposed. Our notation allows a variable local to some component to be private in one system context and exposed in another. The distinction is revealed only by the Interactions section. A reflective variable is considered engaged when its value is computed from that of non-local variables, and disengaged otherwise. The latter case corresponds to the situation when a default value is assigned to the reflective variable because no observation is possible. The terms engaged and disengaged also apply to the exposed variables to indicate if, in a given system state, the value of the respective variable contributes to that of some reflective variable or not.

Illustration. To illustrate the reflective variable concept we return to the badge paging application we alluded to earlier and elaborate a complete program for a simplified version of the problem. We limit our concern to the process by which a single paging request is delivered to a single active badge—by adding unique identifiers, the problem could be easily extended to multiple requests and badges. In our example, we assume a building in which the rooms have sensors (devices that sense the presence of the badge) able to communicate with each other via point-
to-point connections in an attempt to broadcast a single paging request until it reaches the room in which the badge is located. At that point the badge simply logs the request and accepts no further requests. During the entire time, the badge may move from one sensor location to another. The program is shown in Fig. 5.

Each sensor has a unique identifier $s$ and a location $\Lambda$. The location is unique among the sensors and constant throughout the computation. The set $\Psi$ of sensor’s locations

\[
\Psi = \{ s, \lambda : \text{Sensor}(s)@\lambda \} 
\]

represents the space in which the badge is expected to roam. Moreover, each sensor has a set of “neighbors”, namely other sensors with which it can communicate directly. Given a location $\lambda$, the function $\text{neigh}(\lambda)$ represents the set of adjacent locations. We assume that the graph of possible communications between sensors is connected, i.e.:

\[
\forall s, t : \text{Sensor}(s) \land \text{Sensor}(t) \land s \neq t :: \text{linked}(s,t) 
\]

where

\[
\text{linked}(s,t) = \langle \exists \lambda, \mu : \text{Sensor}(s)@\lambda \land \text{Sensor}(t)@\mu :: \mu \in \text{neigh}(\lambda) \rangle \lor \langle \exists u : u \neq s \land u \neq t :: \text{linked}(s,u) \land \text{linked}(u,t) \rangle 
\]

The state of a sensor is captured by two exposed boolean variables $\text{Paging}$ and $\text{Done}$. A sensor starts with $\text{Paging}$ set to false, except for the one at location $q$, the origin of the paging request, and $\text{Done}$ set to false. A paging sensor broadcasts the pending request to all of its non-paging neighbors, as shown in Fig. 6, which in turn eventually start paging as well, thus further propagating the request.

The badge starts at the location of one of the sensors and moves from sensor to sensor, following the restriction imposed by the $\text{neigh}$ function. The badge is initially unaware of the request (exposed variable $\text{Aware}$ is false) and remains so until it reaches a sensor which is paging (the reflective variable $\text{Needed}$ becomes true). At that point, it may set its variable $\text{aware}$ to true, indicating that it has received the request. The badge ignores any

**System Paging**

**Program Sensor(s) at $\Lambda$**

- **declare** $\text{Paging, Done, NeighPaging, NeighDone, Delivered} : \text{shared boolean}$
- **initially** $\text{Paging} = (\Lambda \neq q)$ — $q$ is the origin of the paging request
- **assign** $\text{Done} = \text{false}$

(a) $\text{Paging} := \text{true} \quad \text{if NeighPaging}$ — paging request propagation
(b) $\text{Done} := \text{true} \quad \text{if Delivered } \lor \text{NeighDone}$ — detection and propagation of acknowledgment

**end**

**Program Badge at $\Lambda$**

- **declare** $\text{Aware : boolean}$
- **initially** $\text{Aware} = \text{false}$

- **assign** $\langle \forall \mu : \mu \in \Sigma :: \lambda = \mu \quad \text{if } \mu \in \text{neigh}(\Lambda) \rangle$ — legal move
- $\text{Aware} := \text{true} \quad \text{if Needed}$ — paging reception

**end**

**Components**

\[
\langle \forall s, \lambda : s \text{ is a sensor identifier and } \lambda \text{ is its unique location : Sensor}(s)@\lambda \rangle
\]

- $\text{Badge}@p$ — $p$ is the initial location of the badge

**Interactions**

- $\text{Sensor}(s).\text{NeighPaging} = \langle \exists \lambda : \text{Sensor}(s)@\lambda :: (\exists t, \mu : \mu \in \text{neigh}(\lambda) \land \text{Sensor}(t)@\mu :: \text{Sensor}(t).\text{Paging} \rangle$
- $\text{Sensor}(s).\text{NeighDone} = \langle \exists \lambda : \text{Sensor}(s)@\lambda :: (\exists t, \mu : \mu \in \text{neigh}(\lambda) \land \text{Sensor}(t)@\mu :: \text{Sensor}(t).\text{Done} \rangle$
- $\text{Sensor}(s).\text{Delivered} = \langle \exists \lambda : \text{Sensor}(s)@\lambda :: \text{Badge}@\lambda \land \text{Badge}.\text{Aware} \rangle$
- $\text{Badge}.\text{Needed} = \text{true} \quad \text{if } \exists \lambda, s : \text{Badge}@\lambda \land \text{Sensor}(s)@\lambda :: \text{Sensor}(s).\text{Paging}$
- $\sim \text{false} \quad \text{otherwise}$

**end**

**Fig. 5.** An application involving sensors implementing a paging protocol designed to reach a roaming badge.
duplicate deliveries of the request by other nodes. Once the request has been delivered to the badge, one sensor may eventually detect the successful delivery (the reflective variable Delivered is true) thus setting its Done variable to true and triggering another wave-like spreading of that knowledge to the rest of the network.

The reader may notice that in our explanation of the sensors and the badge we relied heavily on information available only in the Interactions section of the system specification. We presented the components "in context" for the simple reason of keeping the presentation simple and short. This is generally not a good idea since one may want to reuse the components in different contexts. Providing context-independent characterizations of the components is not always easy. For the badge, for instance, this is a simple task. The badge is not required to know why it is needed. The situation with the sensors is more complex. While they do not really need to know anything about the badge or about the origin of the request or the point of notification, explaining the behavior of a single sensor makes little sense outside the broader context of an entire set of sensors. One simply has to consider the entire set (and their interactions) in order to prove anything meaningful about the application. The sensors form a tightly designed subsystem and their behavior is better understood within this sub-context. These issues become especially important as one analyzes the program verification.

Verification. Despite movement and dependency on program locations, reflective variables do not represent a significant departure from the basic UNITY model. Its proof system may be used without change. A system is simply the union of several standard UNITY programs, subject to minor rewriting of the always sections. Thus one can prove global system properties using local properties of the components and the UNITY union theorems[2]. We illustrate this approach by proving that the badge eventually becomes aware that it is paged:

\[ \text{INIT} \rightarrow \text{Badge.Aware} \]

\[ (P) \]

We present the proof in a top down fashion. First, we decompose the global property \( (P) \) in terms of a set of simpler system-wide properties. Because we wish to prove the system properties from component properties, we need to eventually bring the proof to the component level. Unfortunately, only certain instances of unless and ensures properties in a component can be extended to the system as a whole. Moreover, the leads-to properties must be proved at the system level. This is because the result of the union of two programs is a program composed of the statements of the components whose executions are freely interleaved. As a result, a leads-to property pertaining to one of the component might be "canceled" by the execution of statements of the other component. So, in our example, we aim to eventually prove \( (P) \) from unless and ensures properties which can be easily inherited from component-level proofs.

To establish \( (P) \) we first show that in the worst case eventually all sensors receive the paging request. Then we show that once the paging request arrived at every location, the badge is bound to become aware of the request:

\[ \text{INIT} \rightarrow \langle \forall s :: \text{Sensor(s).Paging} \rangle \]  

\[ (P1) \]

\[ \langle \forall s :: \text{Sensor(s).Paging} \rangle \text{ensures Badge.Aware} \]  

\[ (P2) \]
(Note: Quantifiers whose range is not specified are assumed to range over \( \Sigma \) or the set of sensor identifiers as required by the context.) By transitivity of the \( \rightarrow \) operator, we have:

\[
\text{INIT} \iff ( \forall s :: \text{Sensor}(s).\text{ Paging } ) \rightarrow \text{Badge.Aware}
\]

Property (P1) is still fairly complex and needs further decomposition. For this purpose, we introduce a well-founded metric \( N \) representing the number of sensors for which \textit{Paging} is false

\[
N = \# s :: \neg \text{Sensor}(s).\text{ Paging }
\]

and derive property (P1) from the following three properties (where \( |\Sigma| \) is the number of sensor locations):

\[
\text{INIT} \Rightarrow N = |\Sigma|-1 \land ( \exists s :: \text{Sensor}(s).\text{ Paging } ) \quad (\text{P11})
\]

\[
N = k \land ( \exists s :: \text{Sensor}(s).\text{ Paging } ) \Rightarrow N = 0 \quad (\text{P12})
\]

\[
N = 0 \Rightarrow ( \forall s :: \text{Sensor}(s).\text{ Paging } ) \quad (\text{P13})
\]

(P12) is still a \( \text{leads-to} \) property but is proven by induction on an \textit{ensures} property, namely the fact that if the request is present somewhere then the number of non-paging sensors eventually decreases but cannot be less than zero (because the metric \( N \) is well-founded):

\[
N = k \land ( \exists s :: \text{Sensor}(s).\text{ Paging } ) \quad \text{ensures} \quad (N < k \land ( \exists s :: \text{Sensor}(s).\text{ Paging } )) \lor N = 0 \quad (\text{P121})
\]

At this point, we are left to prove a set of system properties which can be derived from local properties of the different components. To accomplish this we rely on an inference rule which states that a property \( p \text{ op } q \) (where \( \text{op} \) is \textit{unless} or \textit{ensures}) of one component holds for the system if in all other components stable \( p \) holds.

Property (P121) holds for the group of sensor programs and its left-hand side is stable in the context of the Badge. The left-hand side of property (P2) is stable for the sensors while (P2) itself holds for the badge when rewritten as

\[
( \forall \lambda : \lambda \in \Psi : \text{Badge.Needed@} \lambda ) \quad \text{ensures} \text{Badge.Aware} \quad (\text{P3})
\]

Another desirable property of the system is that if the badge becomes aware of the request, eventually, all sensors learn about this, i.e., the \textit{Done} variable becomes true everywhere:

\[
\text{Badge.Aware} \iff ( \forall s :: \text{Sensor}(s).\text{Done} ) \quad (\text{Q})
\]

Establishing this property requires first for one sensor to notice that the badge is aware of the request after which the information propagates to all other sensors:

\[
( \exists s :: \text{Sensor}(s).\text{Delivered } ) \quad \text{ensures} \quad ( \exists s :: \text{Sensor}(s).\text{Done } ) \quad (\text{Q1})
\]

\[
( \exists s :: \text{Sensor}(s).\text{Done } ) \iff ( \forall s :: \text{Sensor}(s).\text{Done } ) \quad (\text{Q2})
\]

Unfortunately, the program listed in Fig. 5 does not satisfy property (Q1) because it is possible for the badge to always move to a new location before the sensor at one location can set its variable \textit{Done} to true. Establishing property (Q) is possible because the system reaches a stable state (all sensors paging) from which the badge is bound to make the desired progress. In contrast, property (Q) can not be established because there is not a stable state from which a sensor can realize that the badge became aware of the request. One way to create stability would be to "prevent" the aware badge from changing its location as long as the corresponding sensor hasn't changed its \textit{Done} variable. Another solution is to provide a sensor with some sort of trace of having encountered an aware badge even after it has moved to another location. This way, once the aware badge passes by the location of a sensor, the sensor is bound to eventually take the trace into account. Yet another solution is to have the badge become aware of the paging request simultaneously with the corresponding sensor learning that the request was delivered, i.e., \textit{Badge.Aware} and \textit{Sensor(s).Done} are set to true together. The last two solutions are not feasible when employing reflective variables but can be implemented using the coop variables we describe in the next section.

5. Cooperation Schema and Coop Variables

\textbf{Motivation.} We showed in Section 4 that when employing reflective variables data may disappear before a program gets a chance to access it. Additional coordination is required to avoid this situation. It involves multiple variables and an appropriate protocol to create the appearance of atomic assignments to a single shared variable. The protocol becomes even more complex if multiple parties are involved. For these reasons we turn next to a scheme which enables multiple programs to read and modify common data. The sharing is still assumed to be of a transient nature, i.e., dependent upon the location and state of the participants, but the value associated with each variable persists beyond the duration of the sharing. We call this approach the \textit{cooperation} schema and we refer to the variables participating in the sharing as \textit{coop} variables.

\textbf{Two-party Interactions.} The cooperation schema allows variables belonging to multiple mobile programs to appear, while certain conditions hold, as if they were a single global variable, i.e., any atomic change to
a variable in one of the programs is immediately available to all other programs that share it at the time. Once the conditions under which sharing takes place become false, the shared variables turn into local variables. When and how two variables are shared is formally defined by an interaction statement; the variables are referred to as parties to the interaction. The condition controlling whether or not sharing occurs between the parties is called the sharing condition of the interaction. We describe the interaction as engaged when sharing takes place, and disengaged otherwise.

Because at the time the sharing condition transits from false to true—the interaction is engaging—the two parties may hold distinct values, we assume the existence of a protocol that computes an initial common value before the sharing is activated. During the engaging state the parties to the interaction are considered unreconciled, i.e., they might hold distinct values. Reconciliation refers to the process of assigning a common value—the engagement value of the interaction—to each party. The engagement value may be the current value of one of the shared variables but, in general, one may specify an arbitrary expression for computing the engagement value, e.g., the minimum or maximum value. We allow arbitrary expressions for the engagement value and assume that each application restricts the range of values as it sees fit. The only constraint we impose upon the assignment of the engagement value is that it does not modify the sharing condition of any interaction in the system. This requirement is motivated by technical considerations explained later in this section.

Inversely, we also assume the existence of a protocol taking place when sharing become disabled—the interaction is disengaging—to specify the resulting value of each party once the interaction is fully disengaged. It is typical for both parties to keep their last shared value but one can also specify that one of the variables "loses" access to the shared data (being assigned some distinguished value, e.g., zero or the empty set) as would be the case with a portable computer losing contact with the disk of a docking station. Once again, we require this protocol not to modify the sharing condition for any interaction. Fig. 7 shows the possible state transitions that an interaction can go through.

![Diagram](Fig. 7. States transitions for an interaction.)

To illustrate the concept of a two-party interaction, and to introduce the notation used to represent it, we consider two simple programs that can move within a one-dimensional space, as shown in Fig. 8. Each program has a coop variable, indicated by the shared keyword—the location variable of a program can also be declared as shared, using the notation Program P at shared A, hence allowing the environment to modify it if so required—and, at each step, it may modify its variable or change location, with the programs having opposite movements. Initially the programs are located such that they will eventually meet.

In our example, the system contains one interaction of the form:

\[ x = y \text{ when } \sigma \text{ engage } f \text{ disengage } x = \delta_x \parallel y = \delta_y \]

where

- \( x \) and \( y \) are the parties to the interaction, i.e., the coop variables sharing a common data when the interaction is engaged,
- \( \sigma \) is the sharing condition—\( \exists \lambda :: P@\lambda \land Q@\lambda \) in the example, indicating that the interaction is engaged when the programs containing the parties are at the same location,
- \( f \) is the expression defining the engagement value shared by the parties, and
- \( x = \delta_x \parallel y = \delta_y \) are the expressions assigned to the parties when the interaction is disengaging, i.e., the disengagement values. Either disengagement expression can be omitted, resulting in the corresponding variable maintaining the shared value during the disengaging. Moreover, for
clarity, each expression can refer to the common value before the disengagement by using the symbol _current_, as well as the variable names.

In the system of Fig. 8, each program increments its variable independently while the programs approach one another. When the programs reach a common location, their coop variables are set to the sum of their respective values before the interaction is engaged. From then on, as long as the nodes remain at the same location, any increment of a by node _P_ or of _b_ by _Q_ results in incrementing the shared value. Finally, when the nodes move apart, the sharing condition becomes false and the coop variables _P_.a and _Q_.b are set, respectively, to 0 and the common value before the disengagement. (The expression _Q_.b = _current_ could have been omitted, resulting in the same effect by default.) After disengagement, modifications to _a_ or _b_ are local to the programs _P_ and _Q_, respectively.

<table>
<thead>
<tr>
<th>System Meet</th>
<th>Program <em>P</em> at <em>Λ</em></th>
<th>Program <em>Q</em> at <em>Λ</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>declare</em></td>
<td><em>a</em> : shared integer</td>
<td><em>declare</em></td>
</tr>
<tr>
<td><em>initially</em></td>
<td><em>a</em> = 0 ∧ <em>Λ</em> ≤ 0</td>
<td><em>initially</em></td>
</tr>
<tr>
<td><em>assign</em></td>
<td><em>a</em> := <em>a</em> + 1 ∧ <em>Λ</em> := <em>Λ</em> + 1</td>
<td><em>assign</em></td>
</tr>
<tr>
<td><em>end</em></td>
<td></td>
<td><em>end</em></td>
</tr>
</tbody>
</table>

Components: _P@0 || Q@10_

Interactions:

- _P_.a = _Q_.b
- _when_ (∃ _λ_ :: _P@λ_ ∧ _Q@λ_)
- _engage_ _P_.a + _Q_.b
- _disengage_ _P_.a = 0 || _Q_.b = _current

---

Fig. 8. Example of an interaction between 2 coop variables.

Simultaneous Engagements and Disengagements. Up to this point we assumed that engagement and disengagement are atomic operations which occur simultaneously (or immediately after) with the respective change in the sharing condition. However, in a system containing the following pair of interactions

(I1) _x_ = _y_ when _β_ _engage_ _f_1(_x_, _y_) _disengage_ _x_ = _δ_ _x_

(II) _x_ = _z_ when _¬_ _β_ _engage_ _f_2(_x_, _z_)

when the predicate _β_ transits from true to false, interaction I1 disengages while interaction I2 engages. Because of I1, _x_ needs to be assigned the disengagement value _δ_ _x_ while I2 requires _x_ to receive the engagement value _f_ 2. Moreover, if the computation of the engagement value _f_ 2 refers to the value of _x_, one needs to decide which value of _x_ should be used, the one before or after the disengagement? To eliminate these conflicts, all disengagements are performed before any engagement. In our example, _x_ would first be assigned the disengagement value _δ_ _x_ followed by the assignment of _f_ 2, which would use the value _δ_ _x_ for _x_.

Multi-party Interactions. Consider a system in which a coop variable _x_ is related to two other variables _y_ 1 and _y_ 2 via two engaged interactions. This means that the pair of variables _x_ and _y_ 1 represents some common data, and so does the pair _x_ and _y_ 2. But since _x_ can only have one value, then all three variables must be representations of the same data, as long as both interactions are enabled. In other words, sharing is transitive and requires us to allow multi-party interactions built from independently specified two-party interaction and transitivity.

In any given state, the interactions of a system partition its set of coop variables into _clusters_ with all the variables in one cluster sharing a common value. A coop variable that is not engaged in any interaction is assumed to belong to a singleton cluster. In fact, by extension, we consider each private variable as belonging to a persistent singleton cluster. For a given variable _x_, the cluster it belongs to is represented by _cluster_. _x_. Each element _v_ of _cluster_. _x_ is a reference to a variable from which one can access:

- the name of variable: _name_. _v_
- the program containing the variable: _node_. _v_
- the value of the variable: _value_. _v_

Within the expression defining the engagement value of an interaction, one can simply use ‘_cluster_’ to refer implicitly to the cluster containing the two parties to the interaction.

Formally, given a variable _x_i_, the cluster it belongs to is defined as

_cluster_. _x_i_ = \{ _set_ _j_ : \Σ(_i,j) :: _x_j_ \}
where $\Sigma_{(i,j)}$ is the transitive closure of the set of enabled sharing conditions, i.e., given that the sharing condition of an interaction between two variables $x_i$ and $x_j$ is represented by $\alpha(i,j)$

$$\Sigma(i,j) = (i = j) \lor \alpha(i,j) \lor (\exists k : k \neq i \land k \neq j :: \Sigma(i,k) \land \Sigma(k,j))$$

Note that because an interaction involves two distinct variables, $\alpha(i,j)$ is implicitly false but $\Sigma(i,j)$ is always true, by definition.

A problem introduced by the transitivity of sharing is that two coop variables of a program $P$ might end up in the same cluster. If this were the case, then one must be careful that no statement of $P$ assigns different values to the two variables while they belong to a common cluster. For this reason, we require two variables of the same program never to be allowed to be members of the same cluster.

A similar problem arises when a variable is being assigned multiple engagement (disengagement) values as a result of several interactions engaging (disengaging) simultaneously. Consider, for instance, a system containing the following pair of interactions

\begin{align*}
(11) & \quad x = y \text{ when } \beta \text{ engage } f_1 \\
(12) & \quad x = z \text{ when } \gamma \text{ engage } f_2
\end{align*}

and let's assume that the system is in a state in which both interactions are disengaged. If both interactions engage at the same time then, because all engagements are performed simultaneously, the variable $x$ is assigned $f_1$ and $f_2$ as the engagement values due to 11 and 12, respectively. This is acceptable only if both engagement values are identical at the time. Of course, the constraint doesn't apply to any state in which only one of the interactions engages. Since all variables in a cluster refer to the same data, this assignment constraint applies to any two engagement (disengagement) values assigned that might affect variables of the same cluster. Since disengagements are performed before engagements, the clusters considered in a disengaging state reflect only the consequences of disengagement (e.g., \{a, d\} and \{b, c\} in Fig. 9) and not yet the results of the subsequent engagements (e.g., \{a, d, e\} and \{b, c, f\} in Fig. 9).

![Fig. 9. Cluster changes during simultaneous disengagement (cross hatched lines) and engagements (dashed lines).](image)

In practice, one is likely to favor sharing conditions which may be verified independently of the system behavior. An increase in modularity because the approach avoids the need to reverify the lack of conflict among engagement and disengagement values any time programs or interactions are altered. In the remainder of this paper, we use disjoint sharing relations in all the examples we discuss.

**Proof Logic.** We describe in Appendix A how a system composed of a set of programs and interactions can be converted (mechanically) into an operationally equivalent UNITY program which uses only common variables. As a result, the UNITY proof logic can still be used in the context of coop variables.

**Illustration.** To illustrate the use of coop variables, we turn to a variant of the paging program described in Section 4 (Fig. 10), one in which the sensor variable delivered (which tells a sensor that the badge is aware of the paging) has been changed from a reflective to a coop variable. A sensor's delivered variable and the badge's aware variable (which records the fact that the paging request was observed) are temporarily shared when the two programs are co-located. Because the engagement value is provided by aware, the overall effect is for the badge to leave a copy of its state behind at all visited locations.

**Verification.** By switching to the use of coop variables, we can now show that all the sensors eventually learn that the paging request was delivered to the badge

\[
\text{Badge2.Aware} \iff (\forall s :: \text{Sensor2}(s).Done)
\]

(Eq)

Establishing this property requires first for one sensor to notice that the badge is aware of the request after which the information propagates to all other sensors:

\[
\text{Badge2.Aware} \text{ ensures } (\exists s :: \text{Sensor2}(s).Done)
\]

(Q1)

\[
(\exists s :: \text{Sensor2}(s).Done) \implies (\forall s :: \text{Sensor2}(s).Done)
\]

(Q2)

Property (Q2) is proven in similar manner as property (P1) in Section 4, using the induction rule for the leads-to property with the metric being the number of sensors for which Done is false.
System Paging2

Program Sensor2(s) at A

declare Paging, Done : boolean
Delivered : shared boolean

always NeighPaging, NeighDone : shared boolean

initially Paging = (A ≠ q) — q is the origin of the paging request
Done = false

assign
(a) Paging := true if NeighPaging — paging request propagation
(b) Done := true if Delivered ∨ NeighDone — detection and propagation of acknowledgment
end

Program Badge2 at A

declare Aware : shared boolean

always Needed : shared boolean

initially Aware = false

assign
(c) (\(\mu : \mu \in \Sigma :: A := \mu \text{ if } \mu \in \text{neigh}(A)\)) — legal move
(d) Aware := true if Needed — paging reception
end

Components

\(\text{Badge2}\@p\) — p is the initial location of the badge

\[\langle \text{f s, } \lambda :: \text{s is a sensor identifier and } \lambda \text{ is its unique location :: Sensor2(s)@}\lambda \rangle\]

Interactions

Sensor2(s).NeighPaging =
\[\langle \exists \lambda : \text{Sensor2(s)@}\lambda :: (\exists t, \mu : \mu \in \text{neigh}(\lambda) \land \text{Sensor2(t)@}\mu :: \text{Sensor2(t).Paging}) \rangle\]

Sensor2(s).NeighDone =
\[\langle \exists \lambda : \text{Sensor2(s)@}\lambda :: (\exists t, \mu : \mu \in \text{neigh}(\lambda) \land \text{Sensor2(t)@}\mu :: \text{Sensor2(t).Done}) \rangle\]

Badge2.Needed =
\[\text{true if } (\exists \lambda, s : \text{Badge2}@\lambda \land \text{Sensor2(s)@}\lambda :: \text{Sensor2(s).Paging}) \land \text{Sensor2(s).Delivered = Badge2.Aware when } (\exists \lambda : \lambda \in \Sigma :: \text{Sensor2(s)@}\lambda \land \text{Badge2}@\lambda) \text{ engage Badge2.Aware}\]

end

Fig. 10. Alternate implementation of the paging protocol using coop variables.

Property (Q1) can now be proven because Sensor(s).Delivered remains true once set. We derive (Q1) from the following facts:

- If the badge has seen the paging request, a sensor co-located with the badge eventually records this fact in its variable Delivered

\[\text{Badge2.Aware ensures } (\exists \lambda : \lambda \in \Sigma :: \text{Badge2.Aware@}\lambda \land (\exists s :: \text{Sensor2(s).Delivered@}\lambda)) \] (Q11)

- Awareness by the badge and delivery knowledge by a sensor persist

\[\text{stable Badge2.Aware } \land \text{Sensor2(s).Delivered} \] (Q12)

- A sensor which learned of the delivery of the paging request eventually acknowledges this fact

\[\text{Badge2.Aware } \land \text{Sensor2(s).Delivered ensures Sensor2(s).Done} \] (Q13)

(Q11) holds for the system as a whole because its left-hand side is stable in the badge and sensors, and the right-hand side is established by the UNITY representation of the Interactions section (see Appendix A for details.) Property (Q12) holds for all components of the system. Finally, property (Q13) holds for the sensor and, combined with (Q12), holds for the system.
6. Electronic mail among mobile nodes

In this section, we illustrate further the use of coop and reflective variables by means of three examples that deal with the exchange of electronic messages among a set of mobile nodes. The general concept of electronic mail involves one entity—the sender—generating a message that is destined for another entity—the recipient. The underlying delivery mechanism must ensure that each message is eventually received by its recipient, albeit without any ordering constraints on the deliveries as typical of today's electronic mail behavior. We assume that there are N user nodes capable of sending and receiving messages, each node having a unique identifier between 1 and N. Because we are concerned only with the delivery of messages, we abstract away their content and represent each message simply by some unique identifier (integer value). The nodes move within some discrete space. Two or more nodes can interact with each other only when they are present at the same location.

Direct Exchange. In the first example, we consider only direct message exchanges among nodes. A message generated by one node for another is delivered to its recipient only when the two nodes meet at some common location. Such a situation might occur in practice when subway trains go back and forth on the same linear route. Any two trains eventually are bound to pass each other, at which time they may exchange mail. A solution for this case is shown in Fig. 11.

Each node has an input mailbox In in which it receives new messages when it encounters other nodes, a private mailbox MyBox which holds messages received but not yet processed, and an array of output mailboxes, Out[1] through Out[N], each entry corresponding to a possible destination—by convention, the mailbox Out[p] of node p remains empty throughout the computation. Each node roams freely from one location to the next (statements (a)). Message exchanges take place whenever nodes meet at the same location. While each location could in fact be viewed as an area within which two components are able to communicate, a more realistic approach is to specify when interactions can take place in terms of the relative distance between components.

Messages present in the mailbox In are eventually moved into the private mailbox MyBox (statement (b)). As long as MyBox is non-empty, a message is selected and removed for processing (statement (c))—the variable Msg refers to the message currently being processed. Outgoing messages destined for other nodes are eventually

System DirectExchange

Program User(p) at λ

declare
   In: shared set of integer — incoming mailbox
   MyBox: set of integer — local set of unprocessed messages
   Cut: shared array [1^N] of set of integer — outgoing mailboxes
   Msg : integer ∪ {⊥} — current message to process

initially In = "" ∪ ( ∪ i : 1 ≤ i ≤ N :: Cut[i] = "") ∩ Msg = ⊥
assign

(a)  The node moves to a neighboring location
    ( ∪ µ : µ ∈ neigh(λ) :: λ := µ )
(b)  All received messages are transferred into the private mailbox
    MyBox, In := MyBox ∪ In, ""
(c)  A message to be processed next is selected and removed from the private mailbox
    Msg, MyBox := Select(MyBox), MyBox - Select(MyBox) l f Msg = ⊥ ∧ MyBox ≠ ""
(d)  New messages are generated for some of the other nodes and placed in the output mailboxes
    Msg := ⊥
    ∪ ( ∪ i : 1 ≤ i ≤ N ∧ i ≠ p ::
        Cut[i] := Cut[i] ∪ ProcessMsg(Msg,i,p) l f Msg ≠ ⊥ ∧ ProcessMsg(Msg,i,p) ≠ ⊥ )

end

Components:
   ( ∪ i, λ : 1 ≤ i ≤ N and λ is the initial location of User(i) :: User(i)@λ )

Interactions:
   ( ∀ p,q : p ≠ q :: User(p).Cut[q] = User(q).In
     when ( ∃ λ :: User(p)@λ ∧ User(q)@λ )
     engage ( ∪ x : x ∈ cluster :: value.x )
     disengage User(p).Cut[q] = ""

end

Fig. 11. Direct exchange of messages between the sender and receiver nodes.
generated and placed in the appropriate output mailboxes (statement (d)). When several nodes are present at the same location, the corresponding input and output mailboxes are shared and, because the engagement value is the union of the participating variables, each node sees in its In mailbox all the messages the others have for it. Actually, because of the way things are shared, an input mailbox is accessible to anyone present at that location—to see what is in User[q].In, the node p (present at that location) merely needs to examine its own variable User[p].Out[q]. Any messages not removed by the recipient continue to be present in its input mailbox even after subsequent engagements and disengagements. The output mailboxes, however, are cleared upon disengagement in order to avoid inadvertent multiple-deliveries and the possibility of accumulating obsolete messages. From the point of view of the sender, the fact that a message is no longer present in its output mailbox is an indication that it has been delivered—it may reappear, however, during a later engagement if the receiver delayed transferring the message to its private mailbox.

Figures 12, 13, and 14 illustrate the interactions between variables of three nodes, with no other nodes being at the locations of these three. Fig. 12 presents the nodes as they reside at three distinct locations. Each node is represented with its In and MyBox variables and the first four elements of its Out array. The îth element of User(i).Out is grayed out since it is not meaningful. At that point in the computation, all the represented coop variables are non-engaged, which is indicated by a filled black half-circle next to each one.

Fig. 12. Three mobile nodes at distinct locations μ, λ, and ξ.

In Fig. 13, the three nodes have met at a common location and all the appropriate interactions have been engaged. The shared coop variables are now indicated by unfilled circles. As an example, the variables User(1).Out[4], User(2).Out[4], and User(4).In now form one single shared variable (shown underneath the nodes) containing the combined value of their cluster. The other two clusters are not shown for clarity reasons.

Fig. 13. The nodes are at some common location λ, thus sharing the appropriate variables.

The cluster containing User(1).Out(4), User(2).Out(4), and User(4).In is represented.

Finally, Fig. 14 presents the nodes just after the first one has moved to a new location. Its coop variables are not engaged anymore nor are the Out[1] variables of the other two nodes. As a result of the disengagements, all just-disengaged Out[i] variables now contain an empty set.
Fig. 14. Node 1 has moved away from the location $\lambda$, resulting in the disengagement of several interactions.

Drop off. In the previous example, the successful delivery of messages to their respective recipients relies on the assumption that each sender eventually meets all the other nodes with which it wishes to communicate. This requirement is too restrictive for most applications since certain pairs of nodes are likely to meet very rarely if ever. This is the case in most workplaces where members of different departments interact only occasionally. One way to work around this problem is to provide one or more locations where someone can deposit messages without the recipient being present at that moment but having the guarantee that everyone will stop by, sooner or later, to pick up the pending messages. Wall-mounted mailboxes and voice mail work this way.

The new solution requires the introduction of repository nodes in which messages can be deposited for later retrieval by their respective recipients. We assume that each mobile program eventually passes by each of the mail repositories which have fixed locations. Fig. 15 shows the system using repositories. An interesting aspect of the solution is that the User programs are not aware of the change in the communication protocol which is encapsulated in the interactions—for this reason the code for the User programs is omitted.

Each repository is a “passive” component in the sense that it contains no assignment statements. All changes to its data—in this case a group of mailboxes, one for each user node, represented by $\text{Forward}[I]$ through $\text{Forward}[N]$—result from a computation performed by other components while sharing is engaged, or as a result of engaging or disengaging interactions.

A node $p$ co-located with a repository shares its $\text{In}$ mailbox with the $\text{Forward}[p]$ mailbox of the repository, thus retrieving its pending incoming messages, while at the same time sharing each of its outgoing mailboxes (except $\text{Out}[p]$) with the corresponding Forward mailbox in the repository as a means of transferring outbound

System DropCff

Program User($p$) at $\lambda$  
Same code as before

Program Rep($r$) at $\lambda$

declare $\text{Forward}$: shared array [1..N] of set — forwarding mailboxes for the user nodes

initially $\forall i: 1 \leq i \leq N :: \text{Forward}[i] = "$

assign Nothing being done, all the processing takes place via the interactions with the Users

end

Components:

$\forall i, j: 1 \leq i \leq N \text{ and } \lambda \text{ is the initial location of User(i) :: User(i)@}\lambda,$

$\forall j, j: 1 \leq j \leq R \text{ and } \lambda \text{ is the initial location of Rep(j) :: Rep(j)@}\lambda,$

Interactions:

A user deposits messages in the corresponding mailboxes of some repository

$\forall q: q \neq p :: \text{User(p).Cut}[q] \sim \text{Rep(r).Forward}[q]$ when $\exists \lambda :: \text{User(p)@}\lambda \land \text{Rep(r)@}\lambda$

engage $\cup x: x \in \text{cluster} :: \text{value.x}$

disengage $\text{User(p).Cut}[q] = "$$

A user retrieves incoming messages from its mailbox in some repository

User(p).In $\sim \text{Rep(r).Forward}[p]$ when $\exists \lambda :: \text{User(p)@}\lambda \land \text{Rep(r)@}\lambda$

engage $\cup x: x \in \text{cluster} :: \text{value.x}$

disengage $\text{Rep(r).Forward}[p] = "$$

end

Fig. 15. Indirect mail exchanges between senders and receivers via repositories placed at fixed locations.
messages to the repository for later retrieval by their recipients which are not present and for immediate retrieval by recipients which happen to be present at that location. When a node p moves away from a repository, the messages for p are removed from Forward(p)—they have been delivered to the node—while all messages originating with p are left in the corresponding Forward mailboxes and erased from those belonging to p—they have been sent out by p.

Several other variations of this protocol could be easily specified with only minor changes to the Interactions section of the system description. We could consider, for instance, having all repositories share (continuously or periodically) their mailboxes. In this manner, we can further relax our constraints on the movement pattern of the mobile nodes: every node would be required to eventually visit some (not all) repositories. Furthermore, data sharing could take place among all the repositories or could be implemented in an incremental manner with messages spreading from one repository to another—this latter strategy would require some acknowledgment protocol to avoid the accumulation of garbage.

Hand-Off between cells. In many mobile systems, direct communication between two mobile nodes is not feasible and messages are routed through a static network of intermediary repositories. Each repository is responsible for transferring messages to and from the mobile nodes in some domain of space, named a cell. The repository is said to handle the nodes in its cell. When a mobile node leaves a cell and enters another one, the system performs a hand-off protocol in which responsibility for managing the messages of that node is switched from one cell handler to another. A key aspect of this protocol involves changing communication channels encoded in some allocated frequency range. This is typical of cellular phone systems.

Handling a node can take diverse forms of data management on behalf of that node, such as the storage and access of data files unable to fit within the node itself or the sending and reception of email messages. In our example, we consider the case where the email messages of a mobile node reside in its current cell handler. Node access to the messages located in the handler is represented by means of coop variables with the sharing being controlled by two factors: distance and communication channel number, itself stored in a variable referred to in the Interactions section.

The general overview of the protocol is as follows. While accessing data from one cell handler, the mobile node continuously scans a list of reference channels in search of the "clearest" signal. Upon realizing that another cell handler would provide a better service than its current handler, the mobile node sends a connection request to the preferred handler, while at the same time severing the connection with its current handler. If the request can be processed, the new handler retrieves the node's data from the old handler and informs the node of the success of the request, providing it also with the necessary information to establish a connection with its mailbox on the new handler. Fig. 16 contains the hand-off protocol code for the (mobile) User and (static) Cell nodes. Since we are only concerned with the protocol itself, we do not include the actual statements pertaining to the access and modification of the mail data, and we represent all mail (incoming, internal, and outgoing) simply as a set.

Each cell handler has a distinct channel UserChan(i) associated with each mailbox it can store—this limits the maximum number of mobile nodes a cell can service. To prevent inadvertent interference among nodes while also providing a maximum utilization of the allocated frequency range, channels are reused in cells which are sufficiently apart. To capture this, we use a predicate $\Xi(\lambda)$ to denote the set of cells which can be accessed by a user node located at $\lambda$ (for simplicity, we assume that all nodes have similar communication capabilities). In a flat obstacle-free world, a node can access anything within the range of its communicator and the predicate $\Xi$ can be defined as

$$\Xi(\lambda) = \{ \text{set } c, \mu : \text{Cell}(c)@\mu \land \text{dist}(\lambda, \mu) \leq \delta : c \}$$

while the restriction on channel reassignment amongst the cells can be expressed as

$$\alpha \in \Xi(\lambda) \land \beta \in \Xi(\lambda) \land \alpha \neq \beta \Rightarrow \forall i, j : \text{Cell}(\alpha).\text{UserChan}(i) \neq \text{Cell}(\beta).\text{UserChan}(j)$$

A user node initiates the hand-off protocol if it realizes that a cell would provide a better service. In a wireless environment, this corresponds to having the better cell being closer to the user. To evaluate its distance to the nearby cells, a user node listens to a set of reference signals emitted by the cells. All nodes listen to the same set of reference channels—RefChans—and each cell has its own reference channel—RefChan—which is used not only to signal its presence but also in two-way communication of requests from and acknowledgments to mobile nodes inside the cell perimeter. To prevent any interference between two cells using the same reference channel, we assume that the number of reference channels and their distribution among the cells are such that, for any location $\lambda$, no two cells in $\Xi(\lambda)$ use the same value for their respective reference channels, i.e.,

$$\alpha \in \Xi(\lambda) \land \beta \in \Xi(\lambda) \land \alpha \neq \beta \Rightarrow \text{Cell}(\alpha).\text{RefChan} \neq \text{Cell}(\beta).\text{RefChan}$$
System Hand-Off

Program User(p) at A

declare
Mail: shared set — Mailbox data (on current cell)
Chan : integer — Mailbox channel number for the current cell
Cell : integer — Identifier of the current cell
RefChan : integer — Reference channel for the current cell
Request : integer — Request (node identifier) for a new handler
NewRChan : integer — Reference channel for the requested cell
Checked : location — Last checked location

always
RefChans = [f_1, f_2, ..., f_M] — Common to all User nodes
Intensity: shared array[1..K] of real — Intensity of the reference signals of cells
— Indices of the clearest reference signals

Clearest = \{ set i: 1 \leq i \leq K \land intensity[i] = \{ Max i: 1 \leq i \leq K :: intensity[i] \} :: i \}
SelectClear = \{ Min i: i \in clearest :: i \}
Silent = \{ \forall i: 1 \leq i \leq K :: intensity[i] = 0 \} — No cell handler in the range

Replies : shared set of struct of — Set of replies for all nodes
user: integer — Node identifier
cell: integer — Cell identifier
chan: integer — Communication channel

AllocChan = \{ f \{ \exists c :: <p,c,f> \in Replies } \downarrow otherwise\}
NewCell = \{ c \{ \exists f :: <p,c,f> \in Replies } \downarrow otherwise\}

initially
Chan = \perp \and Cell = \perp \and RefChan = \perp \and Request = \perp \and NewRChan = \perp \and Checked = \perp

assign
(a) User movements
\{ \mu : \mu \in \text{neigh}(A) :: \Lambda := \mu \land \{ Checked = \perp \}\}
(b) □ Check for any clearer signal or no signals at all
\{ Checked := \perp \land \{ Silent \land (\forall f ::宁) \rightarrow (3 i: i \in Clearest :: RefChans[i] = RefChan ) ::
 Chan := \perp \land "Forget" the mail channel information
 Request := p \land Set the request
 NewRChan := \text{RefChans[SelectClear]} — Store the channel to send the request\}
(c) Once a reply is received from the new cell, the hand-off is completed by updating the local data
\{ Checked := \Lambda \land Request \neq \perp \land AllocChan \neq \perp ::
 Chan := AllocChan — Set the channel number
 Cell := NewCell — Keep the cell identifier
 RefChan := NewRChan — Keep the reference channel number
 Request, NewRChan := \perp, \perp — Clear up the request\}
(d) □ Regular processing of incoming and outgoing mail
\{ Checked := \Lambda :: ○\}

end

Fig. 16. Hand-off protocol between mobile User and static Cell handlers.
Program Cell(c) at \( \Lambda \)

**declare**

User\(i\)no : array \([1^\text{M}]\) of integer
User\(Mail\) : shared array \([1^\text{M}]\) of set
Replies : set of struct of
  user : integer
  cell : integer
  chan : integer

**always**

RefChan = \( \forall(c) \)
User\(Chan\)(i) = \( \Phi(i,c) \)
Requests : shared set of integer
FreeIndex = \( \{ \min i : \text{User\(Info\)[i]} \neq \bot :: i \} \)

New\(Request\)(v) = \( v \in \text{Requests} \land \neg (\exists i : \text{User\(Info\)[i]} = v \land \text{FreeIndex} \neq +\infty) \)

New\(Reply\)(v,t) = \( v \in \text{Requests} \land \text{User\(Info\)[i]} = v \land \neg (\exists f :: <v,c,f> \in \text{Replies}) \)

**initially**

Replies = \( \{ \{ i : 1 \leq i \leq \text{M} :: \text{User\(Info\)[i]}, \text{User\(Mail\)[i]} = \bot, c \} \}

**assign**

\( e \) \n\text{Prepare to handle a requesting user if possible}
\( (\forall v : \text{New\(Request\)(v)} :: \text{User\(Info\)}[\text{FreeIndex}] := v) \)

\( f \) \n\text{Retrieve the mail of a user previously handled by some other cell. The cell that relinquishes the data also removes any reply it might have for the user node.}

\( f \) \n\text{Inform the requesting users of their new communication channels once their mail has been transferred here}
\( (\forall v, t : \text{New\(Reply\)(v,t)} :: \text{Replies} := \text{Replies} \cup <v, c, \text{User\(Chan\)(t)}> ) \)

**end**

**Components:**

\( \{ i, \lambda : 1 \leq i \leq \text{N} \land \lambda \text{ is the initial location of User}(i) :: \text{User}(i)@\lambda \} \)

\( \{ j, \lambda : 1 \leq j \leq \text{R} \land \lambda \text{ is the initial location of Cell}(j) :: \text{Cell}(j)@\lambda \} \)

**Interactions:**

The intensity of a reference signal is based on the positions of the node and the cell

User\(p\).Intensity[k] = \( 1(\lambda, \mu) \) if \( \text{User}(p)@\lambda \land \text{Cell}(c)@\mu \land c \in \Xi(\lambda) \land \text{Ref\(Ch\)ans}[k] = \text{Cell}(c).\text{Ref\(Ch\)an} \)

0 otherwise

A node receives the replies of the cell it tries to connect to or is connected with

User\(p\).Replies = \( \text{Cell}(c).\text{Replies} \) if \( \text{User}(p)@\lambda \land c \in \Xi(\lambda) \land \text{User}(p).\text{New\(R\)Chan} = \text{Cell}(c).\text{Ref\(Ch\)an} \)

otherwise

Each cell receives all the requests from the users trying to contact it

Cell\(c\).Requests = \( \{ \cup p, \lambda : \text{User}(p)@\lambda \land c \in \Xi(\lambda) \land \text{User}(p).\text{Request} \neq \bot \land \text{User}(p).\text{New\(R\)Chan} = \text{Cell}(c).\text{Ref\(Ch\)an} \land \text{User}(p).\text{Request} \} \)

A user accesses its mail in the closest cell with which its has a common channel

User\(p\).Mail = \( \text{Cell}(c).\text{User\(Mail\)[i]} \)

when \( \text{User}(p)@\lambda \land c \in \Xi(\lambda) \land \text{User}(p).\text{Chan} = \text{Cell}(c).\text{User\(Chan\)[i]} \)

engage \( \{ \cup x : x \in \text{cluster} :: \text{value}\(x\) \}

disengage \( \text{User}(p).\text{Mail} = \" \)

**Fig. 16. (Cont.)** Hand-off protocol between mobile User and static Cell nodes.
We model the strength of the reference signals as an array \textit{Intensity} of reflective variables which store values in the range 0 to 1. The signal strength is computed in the \textit{Interactions} section as a function of the relative locations between cell handlers and mobile nodes with zero representing a very weak signal. Using this array, we define two additional values: \textit{Closest} which is the set of indices in \textit{Intensity} of the clearest reference signals received by the user node, and \textit{Silent} which is true if no signal is received by the user.

Every time a user node changes its location (statement (a)), it first checks for a better signal than the one of its current handler, if any (statement (b)). If not, then the regular computation proceeds. (In reality, this requirement simply expresses the facts that 1) the computation is much faster than the speed of movement for most realistic applications, and 2) the mobile nodes performs a check on the intensity of the signal on a regular basis.) If no signals are received then the node has lost contact with its cell and thus erases its channel information to avoid any interference with another node using the same channel number when it reenters the range of a different cell. Finally, if a clearer signal is heard (or any signal is heard after the node erased its channel number), the node stops its current mailbox sharing by erasing its channel number and broadcasts a connection request to the desired cell, i.e., by putting its identifier in the local variable \textit{Request}. A cell receives all of broadcast requests directed to it via a reflective variable \textit{Requests}. A cell can handle as many nodes as it has channels. If some channels are unused, the cell selects a request and allocates a channel to the corresponding node (statement (c)). Then, before informing the user, the cell retrieves the user's mail from its current location (latest handler), allowing that handler to free the corresponding channel for use by another node (since this is an aside for this protocol, we do not include the code implementing the data transfers). Once a user's mailbox is transferred, the cell informs the user by broadcasting a reply (statement (d)), containing the user and cell identifiers and the newly allocated channel number. All replies are received by all concerned nodes interacting on the same reference channel via their reflective variables \textit{Replies} from which each node can extract its own reply, if any. Finally, once a node receives a reply to its request, it sets its local channel number to the one provided by the cell handler, thus allowing the sharing of mail data to be engaged.

If a user node, while waiting for a reply, reaches a new location where yet another cell handler is deemed better than the one previously requested, a new request is sent and any reply to a previous request is ignored. A handler having obsolete replies eventually becomes aware of the fact when the user's mailbox is requested by some other handler.

Figures 17 and 18 present the sequence of steps corresponding to handing off the user node User(\(\alpha\)) from Cell(1) to Cell(2). In Fig. 17, the user node, having discovered that Cell(2) would provide a better service, has severed its sharing with Cell(1) (\(\circlearrowright\))—all the mail is being left on Cell(1)—and sent a request to Cell(2) (\(\circledast\)). In Fig. 18, after the user's mail has been transferred from Cell(1) to UserMail[b] on Cell(2) and a reply has been sent to the user's request (\(\ominus\)), sharing can be established between the mail on the new cell and the user (\(\circledast\)).

![Diagram](image)

\textbf{Fig. 17.} User \(\alpha\) has severed its connection with Cell(1) and directed a request for Cell(2).

To verify this protocol, one needs to show 1) that a user node doesn't have access to the mail data of any other nodes, and 2) that connection requests to a cell handler are honored if feasible. The former property is derived from the fact that if a node has access to a channel, the cell handler has the node's mailbox associated with that channel. This is accomplished by having the cell handler acquire the mailbox before notifying its owner and by having a node severe the connection as soon as it gets out of range. (The actual technical details are a bit more complex than our informal arguments might suggest.) The latter depends on a set of factors that are not made explicit in our example: the actual density of nodes in a cell, the velocity of user nodes and its relation to the speed
of communication and computation on the nodes, the precise boundaries and layout of the cells, and the movement patterns. Under reasonable assumptions, however, the property may be shown to hold.

The most striking thing about these examples is the high degree of modularity that can be achieved through the use of transient data sharing. Each node makes no direct references to any other node. Thus, in the hand-off example, one can easily replace the Cell code by other types of handler nodes, as long as there is a way to provide the necessary interactions with the user nodes. Another interesting aspect is the fact that reasoning about mobile computations requires one to make assumptions about the physical reality—things not under the control of the program. Our approach relegates most of these issues to the Interactions section. This partitioning is fortunate since the code is separated into the local computations performed by each node and the relations between the interfaces of the different components of the system. One exception is the mobility restrictions included in the User node. These restrictions are necessary to represent time-dependent behaviors—such as checking for reference signals frequently enough with respect to the motion of the node—in a time-independent model such as UNITY. Sometimes, the motion of a node can very well be under the control of the node, as would be the case with a robot.

7. Conclusion

The basic thesis of this paper is the idea that the notion of location is central to specifying and reasoning about mobile computations. The paper shows that location can be introduced in a traditional programming notation in a straightforward manner and with little or no impact on the ability to reason about the resulting computations. Furthermore, when combined with the concept of transient data sharing (in its two forms, coop and reflective variables), spatial qualifications contribute to achieving a high degree of modularity in the specification of mobile computations. The examples illustrate the kind of modular designs one can specify by using location to control (directly or indirectly) the data-sharing pattern among programs. Research to date also supports the conjecture that shared variables are a reasonable abstraction for constructing mobile computations. Finally, to the best of our knowledge, this is the first time transient data sharing has been viewed as a potentially useful programming construct. However, the implications of employing this construct outside the context of mobile computing remain unexplored at this time.

8. References

A. Appendix: Operational semantics of coop variables

For a more precise characterization of the semantics of coop variables and in order to explain how reasoning about cooperation can be carried out in the context of the UNITY logic, we show how a system involving interactions may be converted (mechanically) into an operationally equivalent UNITY program which uses only common variables.

Let's consider two programs having a single interaction of the form

\[ x_i \sim x_j \text{ when } \sigma(i,j) \]
\[ \text{engage } e(i,j) \]
\[ \text{disengage } x_i = \delta(i,j) \]
\[ || \ x_j = \delta(j,i) \]

and an initial state in which \( \sigma(i,j) \) is false, i.e., the programs start with the interaction disengaged. When the sharing condition \( \beta \) becomes true, the computation must be stopped and the values of the variables must be reconciled to the engagement value. To keep track of the fact that the interaction is engaging, i.e., the sharing condition holds but the variables have not been reconciled, we associate with this particular interaction a new boolean variable \( \rho(i,j) \). This so-called reconciliation variable is required to be true if and only if the parties to the interaction are in a reconciled state. The interaction is actually engaged only when the reconciliation is complete, a situation captured by the fact that both \( \sigma \) and \( \rho \) hold. When the sharing condition becomes false again, the interaction enters first a disengaging state, designed to allow for the assignment of the disengagement values to the parties and the resetting of the reconciliation variable, before moving into the disengaged state again. Fig. 19 presents the state transitions for a simple interaction.

In order to make sure that the reconciliation takes place as soon as the sharing condition becomes true, we need to block the computation until the reconciliation is complete. This can be accomplished by strengthening the guard of each program statement so as to be false while the condition \( (\sigma \land \neg \rho) \) holds and by adding one statement.

---

**Fig. 19.** States transitions for an interaction with sharing condition \( \sigma \) and reconciliation variable \( \rho \).
whose role is to set \( \rho \) to true while initializing the coop variables to the engagement value. Similarly, once an interaction becomes disengaged, the computation needs to be blocked while waiting for \( \rho \) to be reset and the parties set to their disengagement values. Thus, in fact, each program statement needs to have its guard strengthened to hold only if, for each interaction, the sharing condition and the reconciliation variable have the same value. Moreover, any assignment to a variable \( x_i \) needs to be "shadowed" by identical assignments to all other variables of the cluster \( \text{cluster}(x_i) \). So, an assignment of the form

\[
x_i := f \Gamma g
\]

in an original program needs to be replaced by the \textsc{Unity} statement

\[
\langle \| j : \Sigma(i,j) \land \langle \forall u,v :: \sigma(u,v) \leftrightarrow \rho(u,v) \rangle : x_j := f \Gamma g \rangle
\]

Note that if \( x_i \) is a local variable then \( \Sigma(i,j) \) is true only for \( i = j \) and thus any assignment to \( x_i \) is not shadowed to any other variable, nor is \( x_i \) assigned a value as the result of an assignment to another variable.

In addition to transforming the original statements, we also need to add something to handle the engaging and disengaging of interactions. We specified earlier that we consider all pending disengagements to be performed before all the pending engagements, and we mirror this by having a separate \textsc{Unity} statement performing each task.

To disengage an interaction, we need to reset its reconciliation variable while assigning the appropriate disengagement values to the parties. But a party \( x_i \) to a disengaging interaction might still be part of a non-singleton cluster and so the disengagement value assigned to \( x_i \) must be assigned to all members of the cluster \( \text{cluster}(x_i) \). Thus, the additional statement handling all the disengagements at once is

\[
\text{(A)} \quad \langle \| i,j : \text{disengaging}(i,j) ::
\quad \langle \| k : \Pi(i,k) :: x_k := \delta_i(j) \rangle \| \rho(i,j) := \sigma(i,j) \rangle
\]

where

\[
\text{disengaging}(i,j) = \neg \sigma(i,j) \land \rho(i,j)
\]

and \( \Pi(i,j) \) indicates that variables \( x_i \) and \( x_j \) continue to belong to the same cluster after disengagement. \( \Pi(i,j) \) does not reflect, however, the effects of the pending engagements

\[
\Pi(i,j) = (i = j) \lor (\sigma(i,j) \land \rho(i,j)) \lor (\exists k : k \neq i \land k \neq j :: \Pi(i,k) \land \Pi(k,j))
\]

One must be careful to ensure that if two variables of a cluster are involved in disengaging interactions, then both interactions must assign the same disengagement value to the two variables, i.e., the system must satisfy the property

\[
\text{inv.} \quad \langle \forall i,j,u,v : \Pi(i,j) \land \text{disengaging}(i,u) \land \text{disengaging}(j,v) :: \delta_i(u,v) = \delta_j(u,v) \rangle
\]

Finally, to ensure that the disengaging process is finite, we require that the disengaging statement, \( \Delta \), has no effect on any sharing condition \( \sigma(i,j) \):

\[
\{ \sigma(i,j) \} \Delta \{ \sigma(i,j) \}
\]

Once all the disengaging interactions have been taken care of, we can proceed with the engaging process. To engage an interaction, we need to compute its engagement value and assign it to all the variables which are joined into a single cluster by the respective interactions.

\[
\text{(E)} \quad \langle \| i,j : \text{engaging}(i,j) \land \neg (\exists u,v :: \text{disengaging}(u,v)) ::
\quad \langle \| k : \Sigma(i,k) :: x_k := e(i,j) \rangle \| \rho(i,j) := \sigma(i,j) \rangle
\]

where

\[
\text{engaging}(i,j) = \sigma(i,j) \land \neg \rho(i,j)
\]

Once again, because several engaging interactions might simultaneously contribute to the formation of a single cluster, any two engagement expressions assigned to variables of such a cluster must yield the same value:

\[
\text{inv.} \quad \langle \forall i,j,u,v : \Sigma(i,j) \land \neg (\exists a,b :: \text{disengaging}(a,b)) \land \text{engaging}(i,u) \land \text{engaging}(j,v) :: e(i,u) = e(j,v) \rangle
\]

and the engaging statement \( \mathcal{E} \) must also leave all sharing conditions unchanged

\[
\{ \sigma(i,j) \} \mathcal{E} \{ \sigma(i,j) \}
\]

If this constraint and the related one for statement \( \Delta \) were not enforced, a pair of interactions such as
\[ x = y \text{ when } x = 1 \text{} \text{engage } 2 \\
x = z \text{ when } x = 2 \text{} \text{engage } 1 \]

would prevent the computation from resuming because of an infinite sequence of alternating disengaging and engaging statements, each one canceling the effect of the other.

Finally, in the modified system, all the reconciliation variables are initially set to the value of the corresponding sharing conditions, and the system is required to start in a state where all variables in a cluster have the same value, i.e.,

\[
\text{INIT} \Rightarrow \langle \forall i,j :: \sigma(i,j) \Leftrightarrow \rho(i,j) \rangle \land \langle \forall i,j :: \Sigma(i,j) :: x_i = x_j \rangle
\]