Formal Specification and Design of Mobile Systems

Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang
Department of Computer Science
Washington University
Saint Louis, MO 63130
{roman, julien, qingfeng}@cs.wustl.edu

Abstract

Termination detection, a classical problem in distributed computing, is revisited in the new setting provided by the emerging mobile computing technology. A simple solution tailored for use in ad hoc networks is employed as a vehicle for demonstrating the applicability of formal requirements and design strategies to the new field of mobile computing. The approach is based on well understood techniques in specification refinement, but the methodology is tailored to mobile applications and helps designers address novel concerns such as the mobility of hosts, transient interactions, and specific coordination constructs. The proof logic and programming notation of Mobile UNITY provide the intellectual tools required to carry out this task.

1. Introduction

Increasing demands for dependability provide a strong impetus for the software engineering community to evaluate and adopt formal methods. Formal notations led to the development of specification languages; formal verification contributed to the application of mechanical theorem provers to program checking; and formal derivation—a class of techniques that ensure correctness by construction—has the potential to reshape the way software will be developed in the future. Program derivation is less costly than *post-factum* verification, is incremental in nature, and can be applied with varying degrees of rigor in conjunction with or completely apart from program verification. More significantly, while verification is tied to analysis and support tools, program derivation deals with the very essence of the design process, the way one thinks about problems and constructs solutions.

In sequential programming, formal derivation enjoys a long standing and prestigious tradition [4, 5, 7, 8, 14, 15]. By contrast, derivation is a relatively new concern in concurrent programming. Although a clean and comprehensive characterization of the field is difficult to make and is beyond the scope of this paper, three general directions seem to have emerged in the concurrency area. Constructivist approaches start with simple components having known properties and combine them into larger ones whose properties may be computed. CSP-related efforts [6, 9, 11] appear to favor this approach in part due to the algebraic mindset that characterizes the work on abstract CSP. Specification refinement has been advocated strongly in the work on UNITY [2, 10, 23]. An initial highly-abstract specification is gradually refined up to the point when it contains so much detail that writing a correct program becomes trivial. Program refinement uses a correct program as a starting point and alters it until a new program satisfying some additional desired properties is produced. In some of the work on action systems [1], for instance, sequential programs are transformed into concurrent or distributed ones. Mixed specification and program refinement [22, 20] has been used in conjunction with the Swarm model [18] and its proof logic [3, 19].

In this paper we pose a simple question. Is it feasible to apply formal derivation techniques to the development of mobile applications? Mobile systems, in general, consist of components that may move in a physical or logical space. If the components that move are hosts, the system exhibits physical mobility. If the components are code fragments, the system displays logical mobility, also referred to as code mobility. Code on demand, remote evaluation, and mobile agents are typical forms of code mobility. Of course, many systems entail a combination of both logical and physical mobility. LIME [16], for instance, provides logical mobility of agents on top of both fixed and ad hoc networks. 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 [21]. While formal methods may not be ready yet to deliver complete practical systems, the complexity of the undertaking clearly can benefit enormously from the rigor associated with a precise design process, even if employed only in the design of the most critical aspects of the system.

Our first attempt to answer the question raised earlier consists of a formal specification and derivation for a termination detection protocol for ad hoc mobile systems; we carry out this exercise by employing the Mobile UNITY [12] proof logic and programming notation. Mobile UNITY provides a notation for mobile system components, a coordination language for expressing interactions among the components, and an associated proof logic. This highly modular extension of the UNITY model extends both the notation and logic to accommodate specification of and reasoning about mobile programs that exhibit dynamic reconfiguration. Once expressed in the Mobile UNITY notation, a system can be subjected to rigorous formal verification against a set of requirements expressed as temporal properties of its execution.

The remainder of this paper is organized as follows. Section 2 briefly reviews the Mobile UNITY syntax and its proof logic. Section 3 provides a formal specification of the termination detection problem. Section 4 provides a series of UNITY-style refinements of the specification. Section 5 presents the Mobile UNITY program generated from the specified requirements. Finally, Section 6 summarizes the lessons we have learned from this exercise.

2. Methodology and Notation

This section provides a gentle introduction to Mobile UNITY. A significant body of published work is available for the reader interested in a more detailed understanding of the model and its applications to the specification and verification of Mobile IP [13], and to the modeling and verification of mobile code [17], among others. Mobile UNITY is based on the UNITY model of Chandy and Misra [2], with extensions to both the notation and logic. Each UNITY program comprises a **declare**, **always**, **initially**, and **assign** section. The **declare** section contains a set of variables that will be used by the program. Each is given a name and a type. The **always** section contains definitions that may be used for convenience in the remainder of the program or in proofs. The **initially** section contains a set of state predicates which must be true of the program before execution begins. Finally, the **assign** section contains a set of assignment statements. In each section, the symbol '[]' is used to separate the individual elements (declarations, definitions, predicates, or statements).

Each assignment statement is of the form $\vec{x} := \vec{e}$ **if** p, where $\vec{x}$ is a list of program variables, $\vec{e}$ is a list of expressions, and p is a state predicate called the *guard*. When a statement is selected, if the guard is satisfied, the right-hand side expressions are evaluated in the current state, and the resulting values are stored in the variables on the left-hand side. The standard UNITY execution model involves a nondeterministic, weakly-fair execution of the statements in the **assign** section. The execution of a program starts in a state satisfying the constraints imposed by the **initially** section. At each step, one of the assignment statements is selected and executed. The selection of the statements is arbitrary but weakly fair, i.e., each statement is selected infinitely often in an infinite execution. All executions are infinite. The Mobile UNITY execution model is slightly different, due to the presence of several new kinds of statements, e.g., the *reactive* statement and the *inhibit* statement described later.

A toy example of a Mobile UNITY program is shown below.

```
Program $host(i)$ at $\lambda$
    declare
        $token : integer$
    initially
        $token = 0$
    assign
        $count$      $token := token + 1$
    []  $move ::$    $\lambda := Move(i, \lambda)$
end host
```

The name of the program is $host$, and instances are indexed by i. The first assignment statement in $host$ increases the token count by one. The second statement models movement of the $host$ from one location to another. In Mobile UNITY, movement is reduced to value assignment of a special variable λ that denotes the location of the host. We use $Move(i, \lambda)$ to denote some expression that captures the motion patterns of $host(i)$.

The overall behavior of this toy example host is to count tokens while moving. The program $host(i)$ actually defines a class of programs parameterized by the identifier i. To create a complete system, we must create instances of this program. As shown below, the **Components** section of the Mobile UNITY program accomplishes this. In our example we create two hosts placed at initial locations λ_0 and λ_1.

```
System Token-Collection
    Program $host(i)$ at $\lambda$
        ...
    end host
    Components
        $host(0)$ at $\lambda_0$
    []  $host(1)$ at $\lambda_1$
    Interactions
        $host(0).token, host(1).token$
            $:= host(0).token + host(1).token, 0$
                when $(host(0).\lambda = host(1).\lambda)$
                    $\wedge(host(1).token \neq 0)$
    []  inhibit $host(1).move$ and $host(0).move$
```

when $(host(0).\lambda = host(1).\lambda)$
$\wedge(host(1).token > 10)$

End Token-Collection

Unlike UNITY, in Mobile UNITY all variables are local to each component. A separate section specifies coordination among components by defining when and how they share data. In mobile systems, coordination is typically location dependent. Furthermore, to define the coordination rules, statements in the **Interactions** section can refer to variables belonging to the components themselves using a dot notation. The section may be viewed as a model of physical reality (e.g., communication takes place only when hosts are within a certain range) or as a specification for desired system services. In the **Interactions** section of our Token-Collection example, the first statement allows $host(0)$ to collect the tokens from $host(1)$ when the two are co-located. The second statement inhibits the execution of the "$move$" statement in both hosts when the two hosts are co-located and $host(1)$ has more than ten tokens. The operational semantics of the **inhibit** construct is to strengthen the guard of the affected statements whenever the **when** clause is true. The statements in the **Interactions** section are selected for execution in the same way as those in the component programs. Thus, without the **inhibit** statement, $host(0)$ and $host(1)$ may move away from each other before the token collection takes place, i.e., before the first interaction statement is selected for execution. With the addition of the **inhibit** statement, when two hosts are co-located, and $host(1)$ holds more than ten tokens, both hosts are prohibited from moving, until $host(1)$ has fewer than eleven tokens. The **inhibit** construct adds both flexibility and control over the program execution.

In addition to its programming notation, Mobile UNITY also provides a proof logic, a specialization of temporal logic. As in UNITY, safety properties specify that certain state transitions are not possible, while progress properties specify that certain actions will eventually take place. The safety properties include **unless**, **invariant**, and **stable**:

- p **unless** q asserts that if the program reaches a state in which the predicate $(p \wedge \neg q)$ holds, p will continue to hold at least as long as q doesn't, which may be forever.
- **stable** p is defined as p **unless** $false$, which states that once p holds, it will continue to hold forever.
- **Inv** p means ((**INIT** $\Rightarrow p) \wedge$ **stable** p), i.e., p holds initially and throughout the execution of the program. **INIT** characterizes the program's initial state.

The basic progress properties include **ensures**, **leads-to**, **until**, and **detects**:

- p **ensures** q simply states that if the program reaches a state where p is $true$, p remains $true$ as long as q is $false$, and there is one statement that, if selected, is guaranteed to make the predicate q $true$. This is used to define the most basic progress property of programs.
- p **leads-to** q states that if program reaches a state where p is true, it will eventually reach a state in which q is true. Notice that in the **leads-to**, p is not required to hold until q is established.
- p **until** q defined as ((p **leads-to** q) $\wedge$ (p **unless** q)), is used to describe a progress condition which requires p to hold up to the point when q is established.
- p **detects** q is defined as $(p \Rightarrow q) \wedge (q$ **leads-to** $p)$

All of the predicate relations defined above represent a short-hand notation for expressions involving Hoare triples quantified over the set of statements in the system. Mobile UNITY and UNITY logic share the same predicate relations. Differences become apparent only when one examines the definitions of **unless** and **ensures** and the manner in which they handle the new programming constructs of Mobile UNITY [12].

Here are some properties the toy-example satisfies:

(1) $(host(0).token + host(1).token = k)$
unless $(host(0).token + host(1).token > k)$[1]
— the total count will not decrease

(2) $host(0).token = k$ **leads-to** $host(0).token > k$
— the number of tokens on $host(0)$ will eventually increase

In the next section we employ the Mobile UNITY proof logic to give a formal requirements definition to the termination detection problem.

3. Problem Specification

We illustrate our methodology for formal specification and design of mobile systems by examining the case of a termination detection protocol. We consider a set of mobile hosts with identifiers 0 through $(N-1)$ moving through space. Initially some of the hosts are idle while others are active. Hosts communicate with each other while in range. A host can becomes idle at any time but can be reactivated if it encounters an active host. The basic requirement is that of determining that all hosts are idle and storing that information in a boolean flag ($claim$) located on some specific host of our choice, say $host(0)$. Formally, the problem reduces to:

stable W $(\mathcal{S}_1)$

$claim$ **detects** W $(\mathcal{P}_1)$

[1] In this notation we assume all free variables are universally quantified

where W is the termination condition

$$W \equiv \langle \wedge i : 0 \leq i < N :: idle[i] \rangle^{2} \qquad (\mathcal{D}_1)$$

$(\mathcal{S}_1)$ is a safety property stating that once all hosts are idle, no host ever becomes active again. $(\mathcal{P}_1)$ is a progress property requiring the flag claim to eventually record the system's quiescence.

We use $idle[i]$ to express the quiescence of a host and define $active[i]$ to be its negation. It is important to note that the problem definition in this case does not depend on the nature of the underlying computation.

In the next section we demonstrate how to refine this specification step by step.

4. Formal Derivation

In this section, we employ specification refinement techniques towards the goal of generating a programming solution that accounts for the architectural features of ad hoc networks that form opportunistically as hosts move in and out of range. The refinement process starts by capturing high level behavioral features of the underlying application. It moves on to introduce the key elements of the termination detection protocol. The final refinement steps bring mobility and transient communication to the surface before generating Mobile UNITY code adhering to the specification.

The remainder of this section proceeds one refinement at a time. In each case, we provide the informal motivation behind that particular step and show the resulting changes to the specification. As an aid to the reader, each refinement concludes with a list of specification statement labels that captures the current state of the refinement process, as in:

Refinement 0: $\mathcal{P}_1, \mathcal{S}_1$

4.1. Refinement 1: Activation Principle.

While a host may become idle at any time, it can only return to active status in the presence of an active host. This property of the underlying computation can be expressed as a safety property:

$$\begin{aligned} & idle[i] \qquad (\mathcal{S}_2) \\ & \quad \textbf{unless} \\ & \quad \langle \exists j : j \neq i :: active'[j] \wedge active[j] \wedge active[i] \rangle \end{aligned}$$

[2]The three-part notation $\langle \textbf{op}\ quantified\_variable : range :: expression \rangle$ used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for **op**, e.g., *true* when **op** is $\forall$ or zero if **op** is "$+$" .

It states that a host may remain idle forever, or it may become active at a time some other active host is present in the system. The history variable, $active'[j]$ records j's status before i became active. Clearly, if all hosts are idle, quiescence is established and it will be retained forever. As we add $(\mathcal{S}_2)$ to our specification, property $(\mathcal{S}_1)$ is no longer needed because it can be derived from $(\mathcal{S}_2)$.

Refinement 1: $\mathcal{P}_1, \mathcal{S}_2$

4.2. Refinement 2: Token Based Accounting.

One way to determine termination is to simply count how many hosts are idle. Frequent host movement makes direct counting inconvenient, but we can accomplish the same thing by associating a token with each idle host. Independent of whether it is currently idle or active, each host in the system holds zero or more tokens. The advantage of this approach is that tokens can be collected and then counted at the collection point. Of course this strategy works only if the number of tokens always equals the number of idle hosts. If we define T to be the number of tokens in the system and I to be the number of idle hosts, i.e.,

$$T \equiv \langle +i :: token[i] \rangle \qquad (\mathcal{D}_2)$$

$$I \equiv \langle +i : idle[i] :: 1 \rangle \qquad (\mathcal{D}_3)$$

the relationship between the two is established by the invariant:

$$\textbf{Inv. } T = I. \qquad (\mathcal{S}_3)$$

By adding this constraint to the specification, the quiescence property (W) may be replaced by the predicate $(T = N)$, where N is the number of hosts in the system. Property $(\mathcal{P}_1)$ is then replaced by:

$$claim \textbf{ detects } T = N \qquad (\mathcal{P}_2)$$

with the collection mechanism left undefined for the time being.

Refinement 2: $\mathcal{P}_2, \mathcal{S}_2, \mathcal{S}_3$

4.3. Refinement 3: Token Consumption.

To maintain the invariant that the number of tokens in the system reflects the number of idle hosts, activation of an idle host requires that the number of tokens decrease by one. Therefore, when an active host awakens an idle host, they must consume a token between them. To express this, we add a history variable, $token'[i]$, which maintains the value of $token[i]$ held before execution of the most recent statement. The safety property:

$$\begin{array}{l} idle[i] \qquad\qquad (\mathcal{S}_4) \\ \quad \textbf{unless} \\ \qquad \langle \exists j : j \neq i :: active'[j] \wedge active[j] \wedge active[i] \\ \qquad\ \wedge (token[i] + token[j] = \\ \qquad\qquad token'[i] + token'[j] - 1) \geq 0\rangle \end{array}$$

captures the requirement that, when host j activates i, one of their tokens must be destroyed in the same step. Clearly, this new property strengthens property ($\mathcal{S}_2$) by accounting for token consumption. The fact that an active node cannot always activate an idle node that is within reach may be atypical for diffusing computations, but it is not surprising for the mobile setting in which nodes may not be available because they are simply out of reach. The algorithm draws no distinction between disconnection and the lack of disposable tokens.

Refinement 3: $\mathcal{P}_2, \mathcal{S}_3, \mathcal{S}_4$

4.4. Refinement 4: Token collection.

As mentioned in refinement 2, we would like to collect the tokens and count them at the collection point. In this step, we choose $host(0)$ to be our collection point, and we introduce a token passing mechanism. To simplify our narration, we introduce the concept of rank. A host with a higher id is said to rank higher than a host with a lower id. Once all hosts are idle, $host(0)$ should eventually collect all N tokens. We will accomplish this by forcing hosts to pass their tokens to lower ranked hosts. For this, we introduce two definitions. First,

$$\iota = \langle +i : idle[i] :: token[i]\rangle \qquad (\mathcal{D}_4)$$

counts the number of tokens idle agents hold. Obviously, $\iota = N$, when all hosts are idle. We also add

$$\omega = \langle max\ i : \iota = N \wedge token[i] > 0 :: i\rangle \qquad (\mathcal{D}_5)$$

to keep track of the highest ranked host holding tokens. Notice that ω is undefined unless all hosts are idle. After all hosts are idle, we will force ω to decrease until it reaches 0. When $\omega = 0$, $host(0)$ will have collected all the tokens.

At this stage we add a new progress property,

$$\omega = k > 0 \textbf{ until } \omega < k \qquad (\mathcal{P}_3)$$

that begins to shape the progress of token passing. As mentioned in Section 2, an **until** property is a combination of a safety property (the **unless** portion) and a progress property (the **leads-to** portion). Therefore, the above property requires that the metric ω never increases and guarantees that it will eventually decrease. Note that this property only restricts the behavior of the highest ranked host holding tokens but says nothing about the behavior of the other hosts. As long as the highest ranked token holder passes its tokens to a lower ranked host, we can show that all the tokens will reach $host(0)$ without having to restrict the behavior of any host except $host(\omega)$. Therefore termination can now be detected when $(\omega = 0)$, so we can replace ($\mathcal{P}_2$) with

$$claim \textbf{ detects } (\omega = 0). \qquad (\mathcal{P}_4)$$

Refinement 4: $\mathcal{S}_3, \mathcal{P}_3, \mathcal{P}_4, \mathcal{S}_4$

4.5. Refinement 5: Rank independent behavior.

Because determining the identity of ω requires stopping the computation and asking each host whether it has any tokens, we would like to unify the behavior of any host be it ω or not. We do so by rewriting property ($\mathcal{P}_3$) in terms of variables of the underlying program. We begin by adding the progress property:

$$\begin{array}{l} token[j] > 0 \wedge \iota = N \wedge j \neq 0 \qquad (\mathcal{P}_5) \\ \quad \textbf{until} \\ \qquad token[j] = 0 \wedge \iota = N \end{array}$$

which states that all hosts except the token collection point should eventually surrender their tokens. Note that ($\mathcal{P}_5$) alone cannot replace ($\mathcal{P}_3$) because it does not prevent $host(\omega)$ from passing its tokens to a higher ranked host. We fix this by adding the stability property

$$\textbf{stable } \omega \leq k \qquad (\mathcal{S}_5)$$

which guarantees that $host(\omega)$ never passes tokens to a higher ranked host. The combination of ($\mathcal{P}_5$) and ($\mathcal{S}_5$) guarantees that once all hosts are idle, all tokens must be collected by $host(0)$, i.e., all other hosts pass their tokens to $host(0)$ either directly or indirectly. Property ($\mathcal{S}_5$) only constrains $host(\omega)$'s behavior, and the variable ω, still appears in property ($\mathcal{S}_5$); we will replace this property in a later refinement.

Refinement 5: $\mathcal{S}_3, \mathcal{P}_4, \mathcal{S}_4, \mathcal{S}_5, \mathcal{P}_5$

4.6. Refinement 6: Pairwise communication.

So far, hosts can awaken other hosts and hosts can pass tokens to other hosts. Clearly, a host can only activate another host or pass tokens to another host if the two parties can communicate. However, the previous refinements do not embody any notion of communication among hosts required for these activities to actually take place. To accomplish this, we introduce the predicate, $com(i, j)$ that holds if and only if hosts i and j can communicate.

We begin this refinement with a new safety property:

$$\begin{array}{l} \mathit{idle}[i] \\ \quad \textbf{unless} \\ \qquad \langle \exists j : j \neq i :: \mathit{active}'[j] \wedge \mathit{active}[j] \wedge \mathit{active}[i] \\ \qquad\quad \wedge\ (\mathit{token}[i] + \mathit{token}[j] = \\ \qquad\qquad \mathit{token}'[i] + \mathit{token}'[j] - 1) \geq 0 \\ \qquad\quad \wedge\ \mathit{com}(i,j) \rangle \end{array} \qquad (\mathcal{S}_6)$$

which requires that hosts i and j be in communication when j activates i. This property replaces property ($\mathcal{S}_4$) without losing any of the constraints it imposes.

We also add the property:

$$\begin{array}{l} \mathit{token}[j] > 0 \wedge \iota = N \wedge j \neq 0 \\ \quad \textbf{until} \\ \qquad \mathit{token}[j] = 0 \wedge \iota = N \wedge \langle \exists i < j :: \mathit{com}(i,j) \rangle \end{array} \qquad (\mathcal{P}_6)$$

which requires a host to pass its tokens and, when it does, to have been able to communicate with a lower ranked host. As we add this property, we can easily remove property ($\mathcal{P}_5$). Property ($\mathcal{P}_6$), in conjunction with property ($\mathcal{S}_5$), requires that the highest ranked token holder, ω, pass its tokens to a lower ranked host. Again, it still says nothing about the behavior of other hosts.

As we leave this refinement, we separate property ($\mathcal{P}_6$) into its two parts; a progress property,

$$\begin{array}{l} \mathit{token}[j] > 0 \wedge \iota = N \wedge j \neq 0 \\ \quad \textbf{leads-to} \\ \qquad \mathit{token}[j] = 0 \wedge \iota = N \wedge \langle \exists i < j :: \mathit{com}(i,j) \rangle \end{array} \qquad (\mathcal{P}_7)$$

and a safety property,

$$\begin{array}{l} \mathit{token}[j] > 0 \wedge \iota = N \wedge j \neq 0 \\ \quad \textbf{unless} \\ \qquad \mathit{token}[j] = 0 \wedge \iota = N \wedge \langle \exists i < j :: \mathit{com}(i,j) \rangle \end{array} \qquad (\mathcal{S}_7)$$

In later steps, we refine these two pieces separately.

Refinement 6: $\mathcal{S}_3, \mathcal{P}_4, \mathcal{S}_5, \mathcal{S}_6, \mathcal{P}_7, \mathcal{S}_7$

4.7. Refinement 7: Contact Guarantee.

Property ($\mathcal{P}_7$) conveys two different things. First, it ensures that a host with tokens will meet a lower ranked host, if one exists. Second, it requires the tokens to be passed to the lower ranked host. The former requires us to either place restrictions on the movement of the mobile hosts or make assumptions about the movement. For this reason, we refine property ($\mathcal{P}_7$) into two obligations. The first,

$$\begin{array}{l} \mathit{token}[j] > 0 \wedge \iota = N \\ \quad \textbf{leads-to} \\ \qquad \mathit{token}[j] > 0 \wedge \iota = N \wedge \langle \exists i < j :: \mathit{com}(i,j) \rangle \end{array} \qquad (\mathcal{P}_8)$$

guarantees that a host with tokens will meet a lower ranked host. The second,

$$\begin{array}{l} \mathit{token}[j] > 0 \wedge \iota = N \wedge \langle \exists i < j :: \mathit{com}(i,j) \rangle \\ \quad \textbf{leads-to} \\ \qquad \mathit{token}[j] = 0 \wedge \iota = N \wedge \langle \exists i < j :: \mathit{com}(i,j) \rangle \end{array} \qquad (\mathcal{P}_9)$$

forces a host that has met a lower ranked host, to pass its tokens. At the point of passing, communication is still available. These two new properties replace property ($\mathcal{P}_7$); they neither strengthen nor weaken it.

Refinement 7: $\mathcal{S}_3, \mathcal{P}_4, \mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_7\ \mathcal{P}_8, \mathcal{P}_9$

4.8. Refinement 8: Decentralization.

As alluded to previously, because of the existence of ω in our specification, token collection is still not completely decentralized. In this final refinement, we eliminate ω by forcing every host, rather than just $\mathit{host}(\omega)$ to pass its tokens to a lower ranked host, if one exists. Here, then, we add the following safety property:

$$\begin{array}{l} \mathit{token}[j] > 0 \wedge \iota = N \wedge j \neq 0 \\ \quad \textbf{unless} \\ \qquad \mathit{token}[j] = 0 \wedge \iota = N \\ \qquad\ \wedge\ \langle \exists i < j :: \mathit{com}(i,j) \\ \qquad\qquad \wedge\ \mathit{token}[i] = \mathit{token}'[i] + \mathit{token}'[j] \rangle \end{array} \qquad (\mathcal{S}_8)$$

which requires that if any host passes its tokens, there must have been a host with a lower id within communication range that gained that exact number of tokens. We can use this property to replace both properties ($\mathcal{S}_7$) and ($\mathcal{S}_5$). At this point, if we look at properties ($\mathcal{P}_8$), ($\mathcal{P}_9$), and ($\mathcal{S}_8$) we see that the specification requires a host with tokens to meet a lower ranked host (if one exists) and to pass its tokens. The two hosts must be in communication at that time. Because of this, termination is now reduced to detecting when the token count in $\mathit{host}(0)$ reaches N. Therefore we replace ($\mathcal{P}_4$) by:

$$\mathit{claim}\ \textbf{detects}\ \mathit{token}[0] = N \qquad (\mathcal{P}_{10})$$

Refinement 8: $\mathcal{S}_3, \mathcal{S}_6, \mathcal{P}_8, \mathcal{P}_9, \mathcal{S}_8, \mathcal{P}_{10}$

5. Mobile Program

In the final step of the design process, we use our refined specification to mechanistically construct the program text. We first define the program components, and then we derive the program statements directly from the final specification. The resulting program (called a system in Mobile UNITY) is shown below and discussed in the text following it. In the process, we shift to using the programming notation of Mobile UNITY, e.g., $token[i]$ becomes $host(i).token$ to express the local nature of the variable and the ownership by that particular component. Communication is modeled as co-location by introducing:

$$com(i,j) \equiv (host(i).\lambda = host(j).\lambda)$$

```
System MobileSystem
    Program host(i) at $\lambda$
        declare
            $token, n : integer$
        [] $idle, claim : boolean$
        always
            $active = \neg idle$
        initially
            $token = 1$ if $idle \sim 0$ if $active$
        [] $claim = false$
        [] $n = N$ if $i = 0 \sim 0$ if $i \neq 0$
        assign
            idle    :: $idle, token := true, token + 1$ if $active$
        [] detect   :: $claim := (token = n)$ if $i = 0$
        [] move     :: $\lambda := Move(i, \lambda)$
    end host(i)

    Components
        $\langle [] i : 0 \leq i < N :: host(i)\ \ at\ \ \lambda_i \rangle$

    Interactions
    $\langle [] i, j ::$
        $host(i).idle, host(i).token, host(j).token$
          $:= false,$
             $\lceil \frac{host(i).token + host(i).token - 1}{2} \rceil,$
             $\lfloor \frac{host(i).token + host(i).token - 1}{2} \rfloor$
          when $host(i).idle$
                    $\wedge host(j).active$
                    $\wedge (host(i).\lambda = host(j).\lambda)$
                    $\wedge (host(i).token + host(j).token > 0)$
    [] $host(i).token, host(j).token$
          $:= host(i).token + host(j).token, 0$
          when $(host(i).\lambda = host(j).\lambda)$
                    $\wedge (j > i)$
                    $\wedge host(i).idle$
                    $\wedge host(j).idle$
                    $\wedge host(j).token \neq 0$
    [] inhibit host(i).move and host(j).move
          when $(host(i).\lambda = host(j).\lambda)$
                    $\wedge (j > i)$
                    $\wedge host(i).idle$
                    $\wedge host(j).idle$
                    $\wedge host(j).token \neq 0$
    $\rangle$
End MobileSystem
```

Our final specification shapes the resulting program in a particular manner reflected by the code associated with each host and by the coordination policies governing the interactions among the hosts. It is the latter that represents our novel contribution to program derivation literature and the key to tying our derivation techniques to mobility.

The local actions of the individual host are simple; two relate to the behavior of the underlying computation, while the third implements the detection decision. The statement "*idle*" captures the transition from active to idle and, to preserve the invariant ($\mathcal{S}_3$), generates a new token in the process. The "*detect*" statement is reduced to a skip in all hosts except $host(0)$ where tokens are checked against the number of known hosts in order to record the termination in the local variable, $host(0).claim$. Finally, the "*move*" statement models mobility by assigning new values to the location variable, λ.

All actions involving multiple components (pairs in our example) are captured in the **Interactions** section, which may be viewed either as a model of the physical reality or as a specification for services to be provided by the operating system or by middleware designed to support mobility. In this particular example, the first interaction captures the requirement that a token is consumed when an active host wakes an idle one—the fact that the remaining tokens are distributed among the two hosts is just an arbitrary implementation detail. We use the **when** statement here to indicate that host activation is nondeterministic and not obligatory at the time of the encounter—**when** statements are selected for execution in the same weakly-fair manner as all other statements in the system. The second interaction is mechanically similar but captures the transfer of tokens among idle hosts from a higher to a lower ranked host. Here, however, the specification demands that such transfer be accomplished before the two hosts can move apart. We accomplish this by employing a third interaction construct, an **inhibit** statement, that explicitly precludes the movement for as long as the transfer of tokens is not yet complete.

It may appear at this point that we indeed have a correct program, but any attempt to verify its correctness reveals that we have overlooked property ($\mathcal{P}_8$) which demands that (when all hosts are idle) a host (other than $host(0)$) carrying tokens must meet a lower ranked host. In an ad hoc network, constructing a program which satisfies this requirement is generally impossible. Fortunately, Mobile UNITY includes the notion of conditional properties, thus allowing us to define correctness as being conditional of ($\mathcal{P}_8$), i.e., if the latter holds, the desired behavior is achieved. In certain circumstances, we can actually guarantee that ($\mathcal{P}_8$) is met by taking advantage of the properties of motion and the structure of space. For instance, if hosts move back and forth in a long hallway, pairwise meetings become a reality.

6. Conclusion

In this paper we presented the formal derivation of a termination detection protocol for mobile computing. While our discussion focused on physical mobility, i.e., ad hoc networks, the solution works equally well in wired networks supporting communities of mobile agents. Despite the simplicity of the actual protocol, this exercise demonstrates once more the versatility of Mobile UNITY, reveals a number of subtle methodological issues, and raises some interesting questions as well. In this particular study, the problem definition is general enough to be independent of mobility. Is this a desirable goal? When do we need to address mobility from the onset? A precise definition of space was not required since co-location was sufficient to model the idea of two hosts being in range of each other, in part because we ignored ad hoc routing. Can we take advantage of the structure of space and its properties? The order of refinements is clearly important and late introduction of location-dependent communication constraints made the derivation simpler. Is this always the case? While it has been known for a long time that the underlying architecture greatly affects the derivation process, it was interesting to see that the available coordination constructs (e.g., those of Mobile UNITY) also shape the manner in which refinements take place. Can we take advantage of this and simplify the derivation process by focusing on tailored abstract coordination constructs (easily built from Mobile UNITY primitives) rather than the traditional communication primitives considered by much of the literature on distributed computing? Finally, this exercise reconfirmed the usefulness of conditional properties in reasoning about open systems—knowledge of how many hosts we had in the system not withstanding. This paper provides strong evidence that a formal treatment of mobility is not only feasible but, given the complexities of mobile computing, also very desirable.

Acknowledgements: This research was supported in part by the National Science Foundation under Grant No. CCR-9970939. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comput. Prog.*, 13(2):133–180, 1990.

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

[3] H. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. *IEEE Trans. on Parall. Dist. Sys.*, 1(3):365–376, 1990.

[4] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[5] E. Dijkstra and W. Fiejen. *A Method of Programming*. Addison-Wesley, Reading, MA, 1988.

[6] J. Ebergen and R. Hoogerwoord. A derivation of a serial-parallel multiplier. *Sci. Comput. Prog.*, 15:201–215, 1990.

[7] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[8] D. Gries and J. Prims. A new notion of encapsulation. *ACM SIGPLAN Notices*, 20(7):131–139, 1985.

[9] R. Hoogerwoord. A calculational derivation of the CASOP algorithm. *Inf. Process. Letters*, 36:297–299, 1990.

[10] E. Knapp. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Trans. on Prog. Lang. and Sys.*, 12(2):203–223, 1990.

[11] C. Lenguaer. A methodology for programming with concurrency: The formalism. *Sci. Comput. Prog.*, 2(1):19–52, 1982.

[12] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Trans. on Soft. Eng.*, 24(2):97–110, 1998.

[13] P. J. McCann and G.-C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Trans. on Software Engineering and Methodology*, 8(2):115–146, 1999.

[14] C. Morgan. The specification statement. *ACM Trans. on Prog. Lang. and Sys.*, 10(3):403–419, 1988.

[15] J. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[16] G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.

[17] G. P. Picco, G. C. Roman, and P. J. McCann. Reasoning about Code Mobility in Mobile UNITY. WUCS-97-43, Washington University in St. Louis, Department of Computer Science, 1997. (an extended version to appear in ACM TOSEM).

[18] G.-C. Roman and H. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Trans. on Soft. Eng.*, 16(12):1361–1373, 1990.

[19] G.-C. Roman and H. Cunningham. Reasoning about synchronic groups. In J. Banatre and D. Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 21–38, 1992.

[20] G.-C. Roman, R. Gamble, and W. Ball. Formal derivation of rule-based programs. *IEEE Trans. on Soft. Eng.*, 19(3):227–296, 1993.

[21] G.-C. Roman, G. Picco, and A. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *Future of Software Engineering*, pages 241–258. 22nd Intl. Conf. on Soft. Eng., ACM Press, 2000.

[22] G.-C. Roman and C. Wilcox. Architecture-directed refinement. *IEEE Trans. on Soft. Eng.*, 20(4):239–258, 1994.

[23] M. Staskauskas. A formal specification and design of a distributed electronic funds-transfer network. *IEEE Trans. Comput.*, 37(12):1515–1528, 1988.