An Introduction to Mobile UNITY

Gruia-Catalin Roman[1] and Peter J. McCann[2]

[1] Washington University, Department of Computer Science
One Brookings Drive, St. Louis, MO 63130 U.S.A.
roman@cs.wustl.edu http://www.cs.wustl.edu/~roman
[2] Bell Laboratories, Lucent Technologies
1000 E. Warrenville Road, Naperville, IL 60566 U.S.A.
mccap@research.bell-labs.com

Abstract. We define a mobile system as a collection of independently executing components which may migrate through some (logical or physical) space during the course of the computation, with the pattern of connectivity among components changing as they move in and out of proximity. This paper presents Mobile UNITY, a modular notation for expressing mobile computations and a logic for reasoning about their temporal properties.

1 Introduction

The emergence of mobile communications technology is bringing a new perspective to the study of distributed systems. Systems designed to work in this environment must be decoupled and opportunistic. By "decoupled," we mean that applications must be able to run while disconnected from or weakly connected to servers. "Opportunistic" means that interactions can be accomplished only when connectivity is available. These aspects are already apparent in working systems such as Coda [1], a filesystem supporting disconnected operation, and Bayou [2], a replicated database where updates are propagated by pairwise interaction among servers, without involving any global synchronization. In addition to being weakly connected, mobile computers change location frequently, which leads to demand for *context dependent services*, e.g., the location dependent World Wide Web browser of Voelker et al [3]. Even if the goal is transparent mobility, e.g., Mobile IP [4], explicit reasoning about location and location changes is required to argue that a given protocol properly implements location transparency.

This paper proposes a new notation and underlying formal model supporting specification of and reasoning about decoupled, location-aware systems. Our approach is based on the UNITY [5] model of concurrency. This work extends the UNITY notation with constructs for expressing both component location and transient interactions among components. Section 2 is a succinct introduction to our new notation, called Mobile UNITY. A formal axiomatic definition of each construct is included. This section treats Mobile UNITY as a mere technical modification to UNITY independent of any notions of mobility. In Section 3, we discuss the mobility and modularity aspects of the Mobile UNITY notation and show how the composition of mobile units reduces to a form of program union. The new notation is illustrated via a simple example, a serial communication protocol which assumes unidirectional transmission from stationary senders to mobile receivers. Conclusions appear in Section 4.

2 Mobile UNITY without Mobility

In this section we introduce Mobile UNITY in the context of the UNITY notation and proof logic. [1] Because the notation described at this point in the presentation concerns single programs, its applicability to mobile computing will not be immediately obvious. However, in the next section we will discuss how constructs introduced here facilitate the composition of mobile programs in the style of a declarative coordination language.

In standard UNITY, the basic unit of system construction is the program. The structure of a UNITY program consists of a **declare** section, an **initially** section, and an **assign** section. In our notation we preserve the UNITY syntax for the **declare** and **initially** sections and augment that of the **assign** section. Our investigation into programming abstractions suitable for mobile computing led us to the addition of four constructs to the standard UNITY notation:

- *Transactions* provide a form of sequential execution. They consist of sequences of assignment statements which must be scheduled in the specified order with no other statements interleaved in between. The assignment statements of standard UNITY may be viewed as singleton transactions. We will use the term *normal statement* or simply *statement* to denote both transactions and standard statements in a given program. As before, normal statements are selected for execution in a weakly fair manner and executed either as a single atomic action or as part of a series of successive atomic actions.
- *Labels* provide a mechanism by which statements can be referenced in other constructs. This provides us with the ability to modify the definitions of existing statements without actually requiring any textual changes to the original formulation.
- *Inhibitors* provide a mechanism for strengthening the guard of an existing statement without modifying the original. This construct permits us to simulate the effect of redefining the scheduling mechanism so as to avoid executing certain statements when their execution may be deemed undesirable.
- *Reactive statements* provide a mechanism for extending the effect of individual assignment statements with an arbitrary terminating computation. All assignment statements of a given program are extended in an identical manner. The reactive statements form a program that is scheduled to execute to *fixed-point* after each individual assignment statement including those that appear inside a transaction. This construct allows us to simulate the effects of the interrupt processing mechanisms.

In the remainder of this section we examine each of these new constructs in turn and develop a proof logic that accommodates these notational extensions.

The notation for *transactions* assumes the form

$$\langle s_1 ; s_2 ; \ldots ; s_n \rangle$$

where s_i must be an assignment statement. Once the scheduler selects this statement for execution, it must first execute s_1, and then execute s_2, etc. In the absence of any reactive statements, the effect is that of an atomic transformation of the program state.

[1] For the sake of brevity we assume the reader to be familiar with UNITY [5], its notation and related concepts.

A *label* may precede any statement and must be followed by the symbol '::' as in

$$n :: \langle s_1 ; s_2 ; \ldots ; s_n \rangle$$

All labels must be unique in the context of the entire program and there is no need to label every statement. The primary motivation for the introduction of labels is their use in constructing inhibitors.

The *inhibitor* syntax follows the pattern

inhibit n **when** p

where n is the label of some statement in the program and p is a predicate. The net effect is a strengthening of the guard on statement n by conjoining it with $\neg p$ and thus inhibiting execution of the statement when p is true.

A *reactive statement* is an assignment statement (not a sequence of statements) extended by a reaction clause that strengthens its guard as in

s **reacts-to** p

The set of all reactive statements, call it $\Re$, must be a terminating program. We can think of this program as executing immediately after each assignment statement. To account for the propagation of complex effects, we allow the set of all reactive statements to execute in an interleaved fashion until fixed-point. As $\Re$ is merely a standard terminating UNITY program, a predicate $FP(\Re)$ can be computed which is the largest set of states for which no reactive statement will modify the state when executed. This is the fixed-point of $\Re$.

This two-phased mode of computation where every assignment statement is punctuated by a flurry of reactions may seem unreasonable at first, and indeed, it is possible to write completely unrealistic system specifications with many complicated actions relegated to the reactive statements. However, it is also possible to write unrealistic UNITY programs. Assignment statements can be arbitrarily complex and may have no efficient implementation. We favor, however, expressive power over predefined constraints and pursue strategies in which it is the responsibility of the designer to exercise control over the notation in order to achieve an efficient realization on a particular architecture. As shown later, proper use of these constructs will help us to write modular and efficiently implementable specifications of mobile computations.

A program making use of the above constructs is shown below. It consists of two non-reactive statements (one of which is a transaction), one inhibiting clause, and one reactive statement.

program *toy-example*
declare
x,*debug* : integer
initially
x =0
[] *debug* = 0
assign
s :: $x := x + 1$
[] t :: $\langle x := x + 1; x := x - 1 \rangle$
[] **inhibit** s **when** $x \geq 15$
[] *debug* $:= x$ **reacts-to** $x > 15$

end

The statement s increments x by one. The statement t is a transaction consisting of two substatements. The first increments x by one. The second decrements x by one. The programmer might add the inhibiting clause to prevent x from being incremented past 15. This prevents statement s from performing this action, but the statement t may still execute and temporarily increase x to 16. This intermediate state would not be visible to the programmer and indeed the proof logic given below would allow one to prove **inv.** $x \leq 15$ from the text of *toy-example*. Such states can be detected, however, by adding reactive statements such as the last one, which assigns the value of x to *debug* whenever $x > 15$, including during intermediate states of transactions. This is a modular way to add side-effects to a large set of statements without re-writing each statement. We will see later how these aspects of our notation help to model mobile systems.

Now we give a logic for proving properties of programs that use the above constructs. Our execution model has assumed that each non-reactive statement is fairly selected for execution, is executed if not inhibited, and then the reactive program $\Re$ is allowed to execute until it reaches a fixed point state, after which the next non-reactive statement is scheduled. In addition, $\Re$ is allowed to execute to fixed point between the sub-statements of a transaction. These reactively augmented statements thus make up the basic atomic state transitions of our model and we denote them by s^*, for each non-reactive statement s. We denote the set of non-reactive statements by $\aleph$. Thus, the definitions for basic **co** and **ensures** properties become[2]

$$p \textbf{ co } q \equiv \langle \forall s \in \aleph :: \{p\} s^* \{q\} \rangle$$

and

$$p \textbf{ ensures } q \equiv (p \wedge \neg q \textbf{ co } p \vee q) \wedge \langle \exists s \in \aleph :: \{p \wedge \neg q\} s^* \{q\} \rangle$$

Even though s^* is really a statement augmented by reactions, we can still use the Hoare triple notation $\{p\} s^* \{q\}$ to denote that if s^* is executed in a state satisfying p, it will terminate in a state satisfying q. The Hoare triple notation is appropriate for *any* terminating computation.

[2] For two state predicates p and q the expression p **co** q means that for any state satisfying p, the next state in the execution sequence must satisfy q. The relation p **ensures** q means that for any state satisfying p and not q, the next state must satisfy p or q. In addition, there is some statement s that guarantees the establishment of q if executed in a state satisfying p and not q.

In hypothesis-conclusion form, we can write an inference rule for deducing $\{p\}s^*\{q\}$, given some H, a predicate that holds after execution of s in a state where s is not inhibited, and I, an invariant that holds throughout execution of the reactive statements $\Re$. We require that H is sufficient to establish $I (H \Rightarrow I)$, that eventually $\Re$ reaches fixed point ($H \mapsto FP(\Re)$ in $\Re$), and that at fixed point, q is established ($I \wedge FP(\Re) \Rightarrow q$). The following rule holds for non-reactive statements s that are singleton transactions:

$$\frac{p \wedge \iota(s) \Rightarrow q, \{p \wedge \neg\iota(s)\}s\{H\}, H \mapsto FP(\Re) \text{ in } \Re, \textbf{stable } I \text{ in } \Re, H \Rightarrow I, I \wedge FP(\Re) \Rightarrow q}{\{p\}s^*\{q\}}$$

For each non-reactive statement s, we define $\iota(s)$ to be the disjunction of all **when** predicates of inhibit clauses that name statement s. Thus, the first part of the hypothesis states that if s is inhibited in a state satisfying p, then q must be true of that state also. We take $\{p \wedge \neg\iota(s)\}s\{H\}$ from the hypothesis to be a standard Hoare triple for the non-augmented statement s.

For those statements that are of the form $\langle s_1; s_2; \ldots s_n \rangle$ we can use the following inference rule before application of the one above:

$$\frac{\{a\}\langle s_1; s_2; \ldots s_{n-1}\rangle^*\{c\}, \{c\}s_n{}^*\{b\}}{\{a\}\langle s_1; s_2; \ldots s_n\rangle^*\{b\}}$$

where c may be guessed at or derived from b as appropriate. This represents sequential composition of a reactively-augmented prefix of the transaction with its last sub-action. This rule can be used recursively until we have reduced the transaction to a single sub-action. Then we can apply the more complex rule above to each statement. This rule may seem complicated, but it represents standard axiomatic reasoning for ordinary sequential programs, where each sub-statement is a predicate transformer that is functionally composed with others.

The proof obligations $H \mapsto FP(\Re)$ in $\Re$ and **stable** I in $\Re$ can be proven with standard techniques because $\Re$ is treated as a standard UNITY program. We can simplify the rule if we know that the non-reactive statement s will not enable any reactive statements, that is, will leave $\Re$ at fixed point. This can be expressed as:

$$\frac{p \wedge \iota(s) \Rightarrow q, \{p \wedge \neg\iota(s)\}s\{q\}, q \Rightarrow FP(\Re)}{\{p\}s^*\{q\}}$$

which allows us to substitute the obligation $q \Rightarrow FP(\Re)$ for the more complicated invariant and fixed-point argument.

The notation and basic inference mechanism provide tools for reasoning about basic programs. Apart from our redefinition of **co** and **ensures** we keep the rest of the UNITY inference toolkit unchanged. This allows us to derive more complex properties in terms of these primitives. In the following section, we will show how the notation can be used to construct systems of mobile components.

3 Adding Mobility and Structured Composition

Our interest in mobility forced us to reexamine the UNITY model. The initial intent was to provide the means for a strong degree of program decoupling, to model movement

System *Senders-Receivers-Timers*

program *sender(i)* **at** λ

declare

bit : boolean

[] *word* : array[$0..N-1$]of boolean

[] *c,t,sendstamp* : integer

initially

$\lambda = SenderLocation(i)$

assign

transmit	:: $bit, c := word[c], c+1$	**if** $c < N \wedge t \geq sendstamp + \Delta \cdot c$
[] *new*	:: $word, c, sendstamp := NewWord(), 0, t$	**if** $c \geq N$
[] *timer*	:: $t := t+1$	**if** $t < sendstamp + \Delta \cdot c + \Delta/4$

end

program *receiver(j)* **at** λ

declare

bit : boolean

[] *buffer* : array[$0..N-1$] of boolean

[] *c,t,recvstamp* : integer

assign

receive	:: $buffer[c], c := in, c+1$	**if** $c < N \wedge t \geq recvstamp + \Delta \cdot c + \Delta/2$
[] *zero*	:: $c, recvstamp := 0, t$	**reacts-to** $bit = 1 \wedge c \geq N$
[] *timer*	:: $t := t+1$	**if** $t < recvstamp + \Delta \cdot (c+1) - \Delta/4$
[] *move*	:: $\lambda := buffer$	**reacts-to** *ValidLocation(buffer)* $\wedge\ c \geq N$

end

Components

$sender(1) [] sender(2) [] receiver(0)$

Interactions

$receiver(j).bit := sender(i).bit$

reacts-to $sender(i).\lambda = receiver(j).\lambda$

inhibit $sender(i).timer$

when $sender(i).t - sendstamp > receiver(j).t - recvstamp$

$\wedge sender(i).\lambda = receiver(j).\lambda$

inhibit $receiver(j).timer$

when $receiver(j).t - recvstamp > sender(i).t - sendstamp$

$\wedge sender(i).\lambda = receiver(j).\lambda$

end

Fig. 1. Serial communication protocol involving one roving receiver.

and disconnection, and to offer high-level programming abstractions for expressing the transient nature of interactions in a mobile setting. Mobility is accommodated by attaching a distinguished location variable to each program; this provides both location awareness and location control (locomotion) to the individual programs. Decoupling, defined as the program's ability to continue to function independently of the communication context in which it finds itself, is achieved by making the process namespaces disjoint and by separating the description of the component programs from that of the interactions among components.

In this section we will show how each of the new constructs presented in the previous section contributes to a decoupled style of program composition. The reactive statement captures the semantics of interrupt-driven processing and enables us to express synchronous execution of local and non-local actions. The inhibit clause captures the semantics of processing dependencies. In essence, both kinds of statements express scheduling constraints that cut across the local boundaries of individual components. Extra statements are sometimes added to a composition to capture the semantics of conditional asynchronous data transfer among components. Together, these constructs define a basic coordination language for expressing program interactions. Simple forms of these statements have direct physical realization and can be used to construct a rich set of abstract interactions including UNITY-style shared variables, location-dependent forms of interaction, and clock-based synchronization. Figure 1 illustrates the use of the Mobile UNITY notation for a mobile system consisting of two senders and one roving receiver.

In UNITY, a system might consist of several programs which share identically named variables. Each program has a name and a textual description. The operator "[]" is used to specify the assembly of components into a system. In this paper we construct a system in a similar manner but we introduce a syntactic structure that makes clear the distinction between parameterized program types and processes which are the components of the system. To construct a system consisting of multiple *sender* and *receiver* processes we simply add a parameter to the program names and instantiate as many processes as we desire, in this case two senders and one receiver. A more radical departure from standard UNITY is the isolation of the namespaces of the individual processes. We assume that variables associated with distinct processes are distinct even if they bear the same name. Thus, in Figure 1 the variable *bit* in a program like *sender* is not shared with the *bit* in the receiver—they should be thought of as distinct variables. To fully specify a process variable, its name should be prepended with the name of the component in which it appears, for example *sender(i).bit* or *receiver(j).bit*.

Because we consider the individual process to be the natural unit of mobility, each process has a distinguished variable λ that models its current location. This might correspond to latitude and longitude for a physically mobile platform, or it may be a network or memory address for a mobile agent. A process may have explicit control over its own location which we model by assignment of a new value to λ. In a physically moving system, this statement would need to be compiled into a physical effect like actions on motors, for instance. Even if the process does not exert control over its own location we can still model movement by an internal assignment statement that is occasionally selected for execution. Any restrictions on the movement of a component should be

reflected in this statement. In our example, each *sender* process exists at some fixed location in space. The process is neither aware of nor in control of its own location. We express this fact by the absence of any statements that make reference to or modify λ. In contrast to the *sender*, we assume a roving *receiver* that may change location in response to receiving a word containing a valid spatial location.

What processes do in the context of a particular system depends greatly upon the last section of the system specification. The **Interactions** section defines the way in which processes communicate with each other. The statements in the **Interactions** section define these rules using the constructs presented in the previous section, naming variables explicitly by their fully-qualified names. The entire system can be reasoned about using the logic presented in the previous section, because it can easily be re-written into an unstructured program with the name of each variable and statement expanded according to the program in which it appears, and all statements merged into the **assign** section.

The **Interactions** section of Figure 1 contains three statements. The first one makes sure that any change to the bit in a sender is communicated to the corresponding bit variable of any receiver present at the same location. The other two statements are designed to synchronize the local clocks of the collocated components. Under these assumptions about the transient interactions occurring among the components, understanding the individual behaviors in isolation becomes possible.

The *sender* maintains a variable *word* which holds a sequence of bits to be transmitted. The counter c is a pointer to the next bit that will be copied to the variable *bit*, which represents the state of some lower-level communications medium. Upon transmitting the current bit, the counter c is incremented. When it reaches N, no further bits are transmitted until a new word is written to *word* and c is reset by the statement *new*. The process is capable of transmitting bits without any receiver present. The transmission is conditional on the value of the local clock t and a timestamp that is updated when a new word is generated.

The receiver has a similar structure and buffers the arriving bits. Its counter is reset via a reactive statement whenever a value of 1 is received as an indication that the transmission of a new word has started. Upon receipt of a full word which happens to be a valid location the receiver moves to that location before the start of a new data transmission. Another reactive statement accomplishes this.

The constant Δ is used in each of the programs to represent the nominal time interval (in ticks of the $sender(i).t$ or $receiver(j).t$ clock) between transmissions or receptions of a bit. The statement $sender(i).transmit$ is allowed to execute only if time has advanced to at least the cth interval since $sender(i).new$ executed. This is a lower bound on the time at which the statement may execute. The statement $sender(i).timer$ is not allowed to execute if it will advance time more than one-fourth of the duration of the current interval before the current bit has been transmitted. This is an upper bound on the time at which $sender(i).transmit$ may execute. The receiver has a pair of similar constraints, shifted to allow for reception only after the sender has transmitted a bit, with proper choice of Δ. Reasoning about the correctness of the above protocol will naturally require assumptions about the value of Δ. The expression of the real-time constraints here is similar to the *MinTime* and *MaxTime* of [6], except that we choose here to deal with discrete, local clocks rather than a continuous, global one.

4 Conclusion

The Mobile UNITY notation and logic is the result of a careful reevaluation of the implications of mobility on UNITY, a model originally intended for statically structured distributed systems. The example illustrates the basic flavor of the modeling strategy we are pursuing. Movement was reduced to variable assignment thus allowing us to reason about locomotion in the same way that we reason about computation. Transient communication among mobile components was recast as a coordination problem thus freeing the code of the individual components from the burden of having to consider the multiple contexts to be encountered in their lifetime. The set of constructs that form the basis for the coordination language component of Mobile UNITY is very small but has been shown to be highly expressive. Several pairwise high-level transient interaction constructs (e.g., shared variables and statement synchronization) were presented in [7]. Moreover, Mobile UNITY has been used in [8] in an exercise involving the specification and verification of a network protocol (Mobile IP [4]) and to express various forms of code mobility [9]. The notation promises to be a useful research tool for investigating new abstractions in mobile computing. These problems have only recently received attention in the engineering and research community, and formal reasoning has an important role to play in communicating and understanding proposed solutions as well as the assumptions made by each. We are currently investigating coordination constructs that have effective implementations in the ad-hoc networks setting.

Acknowledgment. This paper is based upon work supported in part by the National Science Foundation under Grants No. CCR-9217751 and CCR-9624815. 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. M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu, "Experience with disconnected operation in a mobile computing environment," in *Proceedings of the USENIX Symposium on Mobile and Location-Indepedent Computing*, Cambridge, MA, 1993, pp. 11–28.
2. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *Operating Systems Review*, vol. 29, no. 5, pp. 172–83, 1995.
3. Geoffrey M. Voelker and Brian N. Bershad, "Mobisaic: An information system for a mobile wireless computing environment," in *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, 1994, pp. 185–90, IEEE.
4. Charles Perkins, "IP mobility support," RFC 2002, IETF Network Working Group, 1996.
5. K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, New York, NY, 1988.
6. M. Abadi and L. Lamport, "An old-fashioned recipe for real-time," in *Lecture Notes in Computer Science*, J. W. de Bakker, C. Huizing, W. P. Roever, and G. Rosenberg, Eds., vol. 600, pp. 1–27. Springer-Verlag, 1991.
7. G.-C. Roman, P. J. McCann, and J. Y. Plun, "Mobile UNITY: Reasoning and specification in mobile computing," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, 1997, 250–282.

8. P.J. McCann and G.-C. Roman, "Mobile UNITY coordination constructs applied to packet forwarding for mobile hosts," in *Second International Conference on Coordination Languages and Models*, D. Garlan, D. Metayer, and D. Le, Eds., Berlin, September 1997, pp. 338–354, Springer-Verlag.
9. G.P. Picco, G.-C. Roman, and P.J. McCann, "Expressing code mobility in Mobile UNITY," in *Sixth European Software Engineering Conference (ESEC'97)*, Zurich, 1997, pp. 500–518.