
Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 97 (2004) 259–276
www.elsevier.com/locate/entcs

Reasoning About Context-Awareness in the Presence of Mobility

Christine Julien, Jamie Payton, and Gruia-Catalin Roman [1]

Mobile Computing Laboratory
Department of Computer Science and Engineering
Washington University
Saint Louis, MO, USA

Abstract

Context-awareness is emerging as an important computing paradigm designed to address the special needs of applications that must accommodate or exploit the highly dynamic environments that occur in the presence of physical or logical mobility. A number of formal models are available for reasoning about concurrency. Models designed to capture the specifics of mobility are fewer but still well represented (e.g., Mobile Ambients, π-Calculus, and Mobile UNITY). These models do not, however, provide constructs necessary for explicit modeling of context-aware interactions. This paper builds upon earlier efforts on state-based formal reasoning about mobility and explores the process by which a model such as Mobile UNITY can be transformed to explicitly capture context-awareness. Starting with an examination of the essential features of context-aware systems, this paper explores a range of constructs designed to facilitate a highly decoupled style of programming among context-aware components. The result of this exploration is a model called *Context UNITY*.

Keywords: Formal methods, context-awareness, mobile computing, UNITY, Context UNITY, shared variables

1 Introduction

Formal models aid in understanding the essence of the programming task, e.g., by allowing one to specify and verify a program's behavior. The first formal programming models focused on sequential programming. Floyd [10], for example, characterized program invariants for use in verification. The Hoare triple [12] allows reasoning about program correctness through the use of pre-

[1] Email: {julien, payton, roman}@wustl.edu

1571-0661
doi:10.1016/j.entcs.2004.04.040

and post-conditions. As focus shifted towards concurrent programming, new models for this unique environment emerged. For example, CSP [13] and CCS [17] approach concurrent systems from an algebraic perspective. They use processes and synchronous communication to model the interactions between a system and its environment. The UNITY model [5] emphasizes both specification and verification of concurrent programs through reasoning about transformations in the program's state. Most recently, concurrent programming models have been adapted to account for the added complexities of environments entailing physical and/or logical mobility. The π-calculus [18] builds on CCS as a process algebra for communicating systems that allows expression of reconfigurable mobile processes. Mobile Ambients [4] models the movement of processes between administrative domains. Mobile UNITY [23] extends UNITY with the ability to capture location and movement across logical spaces and facilitates assertional-style reasoning about mobile programs. (Because Mobile UNITY is the foundation of this work, we discuss it in more detail in Section 3.)

As new programming paradigms emerge, it is not always necessary to invent new notations and models. Prior work on models for similar environments can be reused in one of two ways: by reducing the new paradigm to an already existing model or by specializing an existing model to the new paradigm. For example, mobile computing shares many characteristics of concurrent programming. In fact, the mobile computing paradigm itself is an extension of the concurrent programming paradigm. For this reason, adaptation of concurrent programming models to the mobile computing environment proved successful, e.g., π-calculus extended CCS and Mobile UNITY specialized UNITY. Such specializations were useful in providing needed constructs, while continuity with established models allowed reuse of prior intellectual advances in the analysis and evaluation of systems.

Context-aware computing, in which mobile programs adapt their behavior to changes in their environment, is an important emerging computing paradigm. Directly reusing formal models for mobility prevents easy specification of context-based interactions because the primitives are tailored to mobile interactions. However, because context-aware applications often operate in a mobile environment, adapting such a mobility model to account for context-aware interactions allows us to reuse the mobility constructs and portions of the proof logic. Our approach specializes Mobile UNITY to provide constructs that allow reasoning about the manipulation of and interaction with the context. The resulting model, Context UNITY, inherits many of the features of Mobile UNITY, including its notation and proof logic.

Section 2 reviews a mobile program's notion of context, common to many

current systems and applications. Section 3 briefly reviews the Mobile UNITY model. Section 4 overviews the concepts fundamental to Context UNITY, and presents its formalization via a simple context-aware system. Finally, discussions appear in Section 5 and conclusions in Section 6.

2 Context-Aware Computing

Mobile computing applications require the ability for connected hosts to exchange data via wireless connections. The physically mobile hosts form a network whose topology constantly changes. This highly dynamic physical structure supports an even more fluid logical structure composed of application level components that possess the ability to move among mobile hosts. Due to the constantly changing environment, applications often exhibit a behavior that is opportunistic, highly adaptive, and very much dependent on the availability of resources which may also be transient in nature.

A number of researchers have advocated *context-aware computing*, or the ability of applications to detect changes in their environment and to adapt their behavior in response to these changes [26]. Initial work in this area focused on location-awareness and allowed programs to adapt their behavior in response to physical movement [11,27]. More recently, however, context-aware applications and systems have begun including more varied facets of the environment as part of the context. Most notably, the Context Toolkit [25] allows the definition of context-widgets that can sense arbitrary factors in the environment. In fact, as [20] proposes, what we initially viewed as data contents may also be part of the user's context, a notion evident in the EgoSpaces system [14]. Many systems exist for collecting and disseminating context information. Some applications have these facilities built in [1,3,6,9,22,24], while other work focuses on building middleware support systems that distribute context information through publish-subscribe constructs [7] or by focusing on coordination through state-based interactions [14,19].

Our view of context focuses on providing applications with flexible mechanisms for defining individualized contexts that are transparently maintained as the environment changes. This view of the context encompasses the definitions used by current applications and systems. To review, the context is defined by any information available on connected devices (both traditional context information and arbitrary data). One of the most important aspects of our context definition stems from the observation that individual applications demand different things from their environment. For this reason, we define context from the perspective of a single component, taking an egocentric view of the world. The key ramification of this decision is that not every

component "sees" the same context. Because the target environment contains many connected mobile hosts, we extend the boundaries of this context to contain not only information sensed by the local host but also information on any connected host.

While the availability of systems and applications for context-aware computing has rapidly increased, no formal model for their behavior has emerged. The time has come to explore context-awareness from a formal perspective. The notation and proof logic of the resulting model should facilitate reasoning about context-aware programs formally, yet the model should be similar enough to actual programming languages to transition easily to an implementation. In the next section we review the Mobile UNITY model on which we base Context UNITY.

3 A Review of Mobile UNITY

This section provides a gentle introduction to Mobile UNITY through the examination of a concrete example. A significant body of published work is available to the reader interested in a more detailed explanation of the model [23] and its applications to the specification and verification of Mobile IP [16] and to the modeling and verification of mobile code [21].

Mobile UNITY is based on the UNITY model of Chandy and Misra [5], with extensions to both the notation and proof logic. Figure 1 shows a Mobile UNITY system called *BaggageTransfer*. A system of this type is shown in Figure 2. In this system, three programs are defined that combine and interact to form a system in which carts carry loads between a loader and an unloader on a one-dimensional track, where the loader is at the left of the track, and the unloader is at the right.

Cart(k) defines a program in which baggage carts move along a track. As in UNITY, the key elements of program specification are variables and assignments. Programs are sets of conditional assignment statements, separated by the symbol $[]$. Each statement is executed atomically and is selected for execution in a weakly fair manner—in an infinite computation, each statement is scheduled for execution infinitely often. The guards of any normal statements (as opposed to reactive and inhibits statements) can be strengthened without modifying the statement through the use of *inhibit statements*. A construct unique to Mobile UNITY is the *reactive statement* which can be triggered

[1] The *non-deterministic assignment statement* [2] $x := x'.Q$ assigns to x a value x' nondeterministically selected from among the values satisfying the predicate Q.

[2] Though its semantics are identical to those of the **if** keyword, the **when** keyword is used for emphasis in the **Interactions** section of Mobile UNITY systems.

```
System BaggageTransfer
  program Cart(k) at λ
    declare
      x : integer
    initially
      x = 0
    assign
      go_right :: λ := λ + 1
    [] go_left  :: λ := λ − 1
    [] inhibit go_right when x = 0
    [] inhibit go_left when x ≠ 0
    [] λ := 0 reacts-to λ < 0
    [] λ := N reacts-to λ > N
  end

  program Loader(i) at λ
    declare
      y : integer
    initially
      y = 0
    assign
      load :: y := y'.(y' > 0) [1] if y = 0
  end

  program Unloader(j) at λ
    declare
      z : integer
    initially
      z = 0
    assign
      unload :: z := 0 if z ≠ 0
  end

  Components
    Cart(1) [] Cart(2)
  [] Loader(1) at 0 [] Unloader(1) at N

Interactions [2]
  Cart(k).x, Loader(i).y := Loader(i).y, 0
      when Cart(k).x = 0
           ∧ Loader(i).y ≠ 0
           ∧Cart(k).λ = 0
[] Cart(k).x, Unloader(j).z := 0, Cart(k).x
      when Cart(k).x ≠ 0
           ∧ Unloader(j).z = 0
           ∧Cart(k).λ = N
end BaggageTransfer
```

Fig. 1. An example Mobile UNITY system

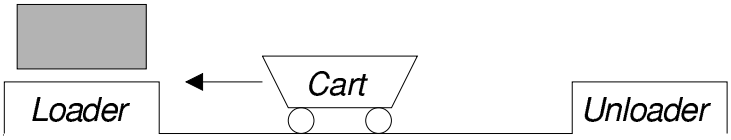

Fig. 2. A System composed of a cart, a loader, and an unloader

by a precondition. Operationally, one can think of each assignment as being extended with the execution of all defined reactions up to such a point that no further state changes are possible by executing reactions alone. More formally, the set of all reactive statements forms a program $\Re$ that is executed to fixed point after each atomic state change. Clearly, $\Re$ must be a terminating program. Both inhibit and reactive statements will be explained in more detail via the example system.

Cart(k) defines the variable x type **integer** in the **declare** section; x represents the size of the cart's load. Every Mobile UNITY program also has a location, represented by the distinguished variable λ; this variable is outside the Mobile UNITY model. The **initially** section states that the cart is empty at the start of execution. Note that λ is not explicitly initialized; it can take on any integer value at the beginning of program execution. Further, although this example uses a simple, one-dimensional notion of space, the model does not restrict a system's definition of space. The **assign** section of *Cart(k)* illustrates the use of several Mobile UNITY constructs. The statements *go_right*

and *go_left* simply update the cart's location on the track. The first inihibit statment prevents the execution of the *go_right* statement when the cart is empty. Similarly, the other inhibit statement prevents the cart from moving left when it is not empty. The remaining two statements are reactive statements. The first is enabled when the cart is at a position less than 0. If after the execution of a normal statement in the program, this statement becomes enabled, the cart's position is updated to a legal position (position 0) on the track. Similarly, the second reactive statement, when enabled, will force the cart to a legal position on the track, position N.

The remaining two programs in the system, *Loader(i)* and *Unloader(j)* are simple programs that each contain one assignment statement. In *Loader(i)*, a value is prepared to be loaded onto a cart; in *Unloader(j)*, a value is removed from the system. The operation of these two programs is more interesting when we consider how these three programs interact. The **Components** section of a Mobile UNITY system defines the programs that make up the system and their initial locations. In this example, the system contains two carts (indexed as *Cart(1)* and *Cart(2)*), one loader, and one unloader. The loader and unloader have specified initial locations, while the carts initial positions are unrestricted.

The **Interactions** section allows the carts, loader, and unloader program instantiations to work together to transport baggage. The first statement is an asynchronous value transfer conditional on the location of the cart and the status of the loader. Since all free variables are assumed to be universally quantified, the statement describes the relationship between a typical loader and a typical cart, so the statement applies to both carts. The load stored in *Loader(i).y* is transferred to the cart and stored in *Cart(k).x*. This enables the cart to start its movement toward the unloader. Similarly, the arrival of a cart at the right side of the track enables the load to be transferred from *Cart(k).x* to *Unloader(j).z*, later to be discarded as apparent in the code of the unloader. The transfers between the cart and the loader or unloader are guaranteed to happen due to the inhibit statements that prevent movement of the cart.

As shown in [15], many different coordination constructs can be built out of the basic constructs presented so far. Among them, one of particular interest is transient and transitive variable sharing, denoted by $\approx$. For instance, the code below describes an interaction between a cart and an inspector where the cart and the inspector share variables y and w as if they denoted the same variable, when co-located.

$$\begin{array}{l} \mathit{Cart(k).y} \approx \mathit{Inspector(q).w} \;\; \textbf{when}\; \mathit{Cart(k).}\lambda = \mathit{Inspector(q).}\lambda \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{engage}\; \mathit{Cart(k).y} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{disengage}\; \mathit{Cart(k).y}, 0 \end{array}$$

At the point when the cart and inspector become co-located, the shared vari-

able is given the value of the cart's y variable as specified by the **engage** clause. When the cart and inspector are no longer co-located, the cart's y variable retains the value of the shared variable and the inspector's w variable is set to 0, as stated in the **disengage** clause.

In addition to the constructs described above, Mobile UNITY also provides a *transaction* (not present in UNITY) for use in the assign section. Transactions capture a form of sequential execution whose net effect is a large-grained atomic state change (in the absence of reactive statements). A transaction consists of a sequence of assignment statements which must be scheduled in the specified order with no other statements interleaved. The notation for transactions is:

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

The reactive statements described above *can* interleave with the statements in a transaction. The **inhibit** construct can also be used to guard the execution of transactions.

Mobile UNITY's proof logic differs from UNITY's only in the proof of Hoare triples. This difference results from the introduction of inhibit and reactive statements. For instance, in UNITY a property such as:

$$\{p\}s\{q\} \textbf{ where } s \textbf{ in } P$$

refers to a standard conditional multiple assignment statement s exactly as it appears in the text of the program P. By contrast, in a Mobile UNITY program one needs to use:

$$\{p\}s^*\{q\} \textbf{ where } s \in \aleph,$$

where $\aleph$ denotes the normal statements of P while s^* denotes a normal statement s modified to reflect the guard strengthening caused by inhibit statements and the extended behavior resulting from the execution of the reactive statements in the reactive program $\Re$ consisting of all reactive statements in P. The following inference rule captures the proof obligations associated with verifying a Hoare triple in Mobile UNITY under the assumption that s is not a transaction:

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

For each non-reactive statement s, $\iota(s)$ is defined 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. $\{p \wedge \neg\iota(s)\}s\{H\}$ (from the hypothesis) is taken to be a standard Hoare triple for the non-augmented statement s. H is a predicate that holds after execution of s in a state where s is not inhibited. It is required that H leads to fixed-point and q in the reactive program $\Re$. Verifying the Hoare triple for a transaction requires another inference rule [23] which is omitted here for brevity.

4 Formalization

As indicated previously, the goal of this work is to adapt a formal model of mobility to the context-aware environment to allow reasoning about programs that adapt their behavior to their changing context. Given that we are adapting Mobile UNITY to the context-aware environment, the tools at our disposal include Mobile UNITY programs (units of modularity), variable and variable assignment, and the ability to specify interactions between programs. In this section, we introduce the details of the notation for expressing context-aware systems in Context UNITY through the use of a detailed example. After the discussion of the basic program and notation, we discuss the power of the nondeterministic assignment statement in defining contexts through an extension of the example program. We then present options for defining relationships between a program and its context. Finally, we discuss how a Context UNITY program can be reduced to Mobile UNITY.

4.1 A Simple Context UNITY Example

A sample Context UNITY system is given in Figure 3. The *Cart* program text specifies a modification of the original cart program such that the cart adapts its movement according to its context. The cart senses whether it should be loaded or unloaded and in response travels in the direction of a loader or unloader. Unlike the previous example, the cart does not need to know in advance the locations of loaders and unloaders, or even the directions in which they are located. To keep the example simple and focused on the incorporation of context, this program allows only a single cart, loader, and unloader. We discuss a more complex program later that allows multiple instantiations of each program. The most novel portion of the system is the *Cart* program's definition of its context. In this discussion, we talk mostly from the cart's perspective, i.e., the cart is our *reference program*. The other two programs in our system, the *Loader* and *Unloader*, are passive entities and use only local information.

The *Cart* program defines variables x, d, l, and u in the **declare** section. As in the Mobile UNITY system presented in Section 3, x is a variable representing the cart's current load. We refer to x as a local variable because there is no usage of the variable outside of the cart program. The remaining variables d, l, and u are called *context variables* in Context UNITY because they directly model, access, and modify context in a program. In the cart program, d is used to represent the cart's destination. l is a context variable representing the load available at the *Loader*, and u represents the availability of space for the cart's cargo at the *Unloader*.

System *BaggageTransfer*

program *Cart at* λ

declare
x, d, l, u : integer,
$[\bar{l}, \bar{u}$: integer$]$

always $changed(l) = [(l \neq \bar{l})] \ [] \ changed(u) = [(u \neq \bar{u})]$

initially $x = 0 \ [] \ [\bar{x} = 0]$

assign
inc :: $\lambda := \lambda + 1$
[] dec :: $\lambda := \lambda - 1$
[] **inhibit** inc **when** $d \leq \lambda$
[] **inhibit** dec **when** $d \geq \lambda$
[] $load$ :: $x, l := l, 0$ **if** $l \neq \perp$
$\|[\bar{l} := l$ **if** $l \neq \perp\,]$
[] $unload$ :: $x, u := 0, x,$ **if** $u \neq \perp$
$\|[\bar{u} := u$ **if** $u \neq \perp\,]$

context
define
$d := Loader.\lambda$ **when** $x = 0 \wedge Loader.y \neq 0$
$\sim Unloader.\lambda$ **when** $x \neq 0 \wedge Unloader.z = 0$
[] $l := Loader.y$ **when** $Loader.\lambda = d \wedge Loader.y \neq 0 \wedge d = \lambda \wedge x = 0$
[] $u := Unloader.z$ **when** $Unloader.\lambda = d \wedge Unloader.z = 0 \wedge d = \lambda \wedge x \neq 0$
resolve
$Loader.y := l$ **reacts-to** $changed(l) \wedge l = 0$
$\|[\bar{l} := l$ **reacts-to** $changed(l) \wedge l = 0]$
[] $Unloader.z := u$ **reacts-to** $changed(u) \wedge u \neq 0$
$\|[\bar{u} := u$ **reacts-to** $changed(u) \wedge u \neq 0]$

end

program *Loader at* λ

declare y : integer

initially $y = 0$

assign $load :: y := y'.(y' > 0)$ **if** $y = 0$

end

program *Unloader at* λ

declare z : integer

initially $z = 0$

assign $unload :: z := 0$ **if** $z \neq 0$

end

Components
$Cart \ [] Loader \ [] \ Unloader$

end *BaggageTransfer*

Fig. 3. An example Context UNITY system

The *Cart* program defines two additional variables, $\bar{l}$ and $\bar{u}$, which we refer to as *shadow variables*. In the system shown in Figure 3, the program explicitly ensures that $\bar{l}$ always contains the last known value of l in order to trigger statements that wish to respond to a change in l. The same is true of the shadow variable $\bar{u}$. These shadow variables are included in this example program for clarity, but Context UNITY assumes these variables to be standard, and the programmer need not deal with them unless an explicit change is desired. Such an explicit use is seen in the **resolve** section of the program shown in Figure 3 when the program uses the *changed* macro that relies on the shadow variable. We highlight the use of shadow variables in the program with square brackets, e.g., $[(l \neq \bar{l})]$. Generally, in a Context UNITY program, the statement assigning to these shadow variables would not be visible.

The **initially** section of the cart program can be used to give initial values to variables at the start of execution. The cart's **initially** section is used to state that the cart is empty at the start of execution. The **assign** and **context** sections highlight the use of context in Context UNITY.

Local and context variables are used in the **assign** section to achieve the program's objective. The statements *inc* and *dec* update the location of the cart through simple assignment to the distinguished variable λ. Two *inhibit* statements are used to prevent unwanted cart movement. As explained previously in Section 3, these inhibit statements strengthen the guards on the *inc* and *dec* statements, preventing their execution under certain conditions. The first inhibit statement prevents the cart from moving to the right when the cart is already at its destination (d), or when the destination is to the left. The second prevents the cart from moving left when either the cart is at its destination or should move right to reach its destination. The value of the destination d is determined by the context. We will discuss its value in more detail shortly. When the *load* statement is executed, the load l is taken from the loader and placed into the cart through assignment to x. At the same time, the cart requires a mechanism to indicate to the loader that it has taken the load. The *Cart* program accomplishes this through a combination of a local assignment to l, followed by an assignment of the value l to the loader's exposed variable *Loader.y*. The former is part of the program code while the latter is expressed as part of the context manipulation. We refer to this update of context in response to change in the program state as a *projects* relationship capturing the notion that a local manipulation of the context results in the new value being projected onto the system configuration. This projection is defined by a rule in the **context** section, discussed below. Finally, in conjuction with the movement of the load, the *Cart* program explicitly updates the

shadow variable $\bar{l}$, which will ultimately trigger the aforemention projection back to the loader. The *unload* statement works in a similar fashion. Both the *load* and *unload* statements utilize the program's abstraction of context in two ways. First, the statements use the reflection of context in the variables l and u to determine if the cart should be loaded or unloaded, and to transfer the load into the cart or onto the unloader. The reflection of context in the current value of the context variables is captured by a *reflects* relationship, which is specified here as an assignment to context variables in the **context** section of the program. Second, the *load* and *unload* statements project changes onto the global context through assignment to the context variables l and u. As mentioned before, the projects relationship is also specified in the **context** section of the program.

The context of a program is defined in the **context** section. In general, a program's context could contain any information available in the entire network. In the Mobile UNITY terms of programs and variables, this *global state* of the system includes all variables defined by all programs in the system. In truth, however, programs want to keep some of their data private. Therefore, we introduce the term *exposed variables* to refer to the variables in a program that are used by others' contexts. The maximal context of a program in the system is the union of all of the exposed variables across all programs; we refer to this as the *system configuration*. In the actual program definition, exposed variables appear the same as any other variable; the term simply provides the necessary language for discussing context definitions.

The manner in which the system configuration is reflected by the context variables is specified in the **define** section of the **context** description. To put it simply, the variables used in the program to capture its context are defined based on the current system configuration. As the configuration changes, the values of the context variables also change to reflect the current environment. In the cart program, the reflection of context is defined by context variables d, l, and u. The destination d of the cart is defined as the location of the loader when the cart should be loaded, i.e., when the cart is empty ($x = 0$) and the loader has a load available ($Loader.y \neq 0$). When the cart needs to be unloaded and the unloader has space available, the destination d is defined to be the location of the unloader.

The more complex reflection relationship for the context variable l is given as follows:

$$l := Loader.y \textbf{ when } Loader.\lambda = d \wedge Loader.y \neq 0 \wedge d = \lambda \wedge x = 0$$

The use of the **when** clause ensures that when this statement is selected for execution, the load l will be assigned only if the four listed conditions hold. The first condition states that the loader's location, $Loader.\lambda$, is the same as the destination d. The second states that the loader has a load to give

the cart, i.e., $Loader.y \neq 0$. The third condition says that the cart is empty ($x = 0$). The last condition, $d = \lambda$, states that the cart is at the destination specified by the context variable d. When all of these conditions hold, the empty cart is located at the chosen non-empty loader, and the cart will be given the load ($l := Loader.y$). The reflection relationship for the context variable u is specified in a similar fashion: u indicates a space to place a load, and it is defined only when the non-empty cart is present at the chosen empty unloader.

A program's impact on its environment through assignment to context variables is specified by the **resolve** section of **context**. Each time the cart's load, x, is updated to contain a new value, a change is also made to a context variable (either l or u). This condition is captured in the cart's **resolve** section through the use of a function, *changed*. The first statement in the resolve section projects the effect of loading the cart on the context:

$$Loader.y := l \textbf{ when } changed(l) \wedge l = 0$$
$$\| [\overline{l} := l \textbf{ reacts-to } changed(l) \wedge l = 0]$$

This definition of the program's impact on the context is a reactive statement and is evaluated immediately after the execution of any statement. Such a reactive definition for projecting context ensures that the loader knows that its load has been picked up as soon as it happens. With this definition, every time a statement in the cart program is executed, resulting in a state in which the cart is not empty and the cart's load l has changed, the value of the cart's context variable l is projected into the loader's variable *Loader.y* (representing the available load at the loader) through simple variable assignment. Since the cart sets l to be zero in the **assign** section when it changes its local load x to be a non-zero value, the value projected to *Loader.y* will be zero. A similar feedback of context occurs for the unloader: when the cart drops off a load in the **assign** section, the context variable u is set to be the value of the load x.

4.2 The Mechanics of Context Definition

The example program used above to present Context UNITY's basic notation uses a relatively simple definition of context in a program where the entire state of the system is always known. The power of Context UNITY can only be fully realized when one examines the complex contexts that can be expressed using the model. Next we discuss the use of non-deterministic assignment statements for use in providing more specialized context reflection and projection. We then add the notion of quantification to extend the power of Context UNITY's context definitions.

Non-Deterministic Assignment. In Context UNITY, the use of non-deterministic assignment statements proves essential to providing flexible context definitions. This was not apparent in the simple example presented pre-

context
define
$d := d'.(\exists j, y :! Loader(j) \wedge Loader(j).y \neq 0 :: d' = Loader(j).\lambda)$ **when** $x = 0$
$\sim d'.(\exists j, z :! Unloader(j) \wedge Unloader(j).z = 0 :: d' = Unloader(j).\lambda)$ **when** $x \neq 0$
$[]\ l, i := (l', i').(\exists j, y :! Loader(j) \wedge Loader(j).\lambda = d \wedge Loader(j).y \neq 0 :: l', i' = Loader(j).y, j)$ **when** $d = \lambda \wedge x = 0$
$[]\ u, i := (u', i').(\exists j, z :! Unloader(j) \wedge Unloader(j).\lambda = d \wedge Unloader(j).z \neq 0 :: u', i' = Unloader(j).z, j)$ **when** $d = \lambda \wedge x \neq 0$
resolve
$Loader(i).y := l$ **reacts-to** $changed(l) \wedge l = 0$
$\|[\overline{l} := l$ **reacts-to** $changed(l) \wedge l = 0]$
$[]\ Unloader(i).z := u$ **reacts-to** $changed(u) \wedge u \neq 0$
$\|[\overline{u} := u$ **reacts-to** $changed(u) \wedge u \neq 0]$

Fig. 4. A Non-Deterministic Context Definition

viously because the system itself was deterministic. If the cart were placed in an environment about which it had no information, however, it should be able to make use of knowledge of its task to discover needed information about its context. By using a non-deterministic assignment statement to define its context, a program can select values for its context variables from a set of values satisfying some provided conditions. For example, in our program, if the cart is empty, it needs to find any loader who has a load $y \neq 0$. If such a program is found, the cart should start moving toward the loader's destination. This allows the system to contain multiple loaders, indexed by integers as in the original Mobile UNITY program (e.g., $Loader(i)$ refers to the i^{th} loader). The **context** section for this modified cart program is shown in Figure 4. The notation $!Loader(i)$ is used to indicate that the i^{th} program named *Loader* exists. In the context definition, the cart now defines its destination d using a non-deterministic assignment statement that chooses a loader or unloader based on properties that the respective component must meet. For example, in the first case, the cart is empty ($x = 0$), and it is looking for a loader with a load to offer ($Loader(i).y \neq 0$). Any loader meeting such a specification will suffice. If any loaders satisfy the list of conditions, one's location is chosen non-deterministically as the destination, which is stored in d. An unloader is chosen in a similar fashion.

Once at the destination, the context variable l or u is also updated appropriately. In the case of the loading action, a key addition is the use of a new context variable, i, which is used to remember exactly which loader the cart took a load from. This information is used in a deterministic fashion by the resolve section to update the appropriate loader. In choosing the load to load into the cart, the context definition non-deterministically chooses a loader at the destination location (it is possible that multiple loaders are at the same location) that has a load. The load at the chosen loader and the identity of the loader are stored in the context variables l and i. Again, the context

definition for u is similar to that for l. The only change in the **resolve** section ensures that the loader the cart took a load from is the same one to which the change projects back.

Non-deterministic assignment statements can also be used in the **resolve** section of the context to project changes onto non-deterministically selected variables. While not immediately useful in the context of our example program, we can imagine the case when multiple unloaders are at the same location. The context variable u might actually reflect a particular unloader, but in the projection phase, the cart only has to be sure that it unloads onto an unloader at the location d with no current load. An example resolution rule for an unloader might be:

$$\langle \| j : j = j'.(!\mathit{Unloader}(j) \wedge \mathit{Unloader}(j).\lambda = d \wedge \mathit{Unloader}(j).z = 0) \\ :: \mathit{Unloader}(j).z := u \rangle$$

This non-deterministically selects a single unloader that satisfies the condition and sets its variable z to be the load unloaded.

Quantification. The key to the use of non-deterministic assignment statements in the definition of a program's context is their ability to widen the definition capabilities. For example, not only can a cart program choose any component from a set of loaders or unloaders, it can choose the closest component or the loader with the largest load to offer. In both these cases, only the definition of the context variable d changes, while the remainder of the **context** section remains the same. This also demonstrates the modularity of the context variable definitions. The redefinition of d for moving to the closest loader or unloader is shown in Figure 5. We use the symbol $\mathcal{D}$ to refer to a distance formula, i.e., $\mathcal{D}(\lambda_1) = |\lambda_1 - \mathit{Cart}.\lambda|$. From these examples, we

$$\begin{aligned} d := d'.(\exists j, y :!\mathit{Loader}(j) \wedge j = \langle \min k : \mathit{Loader}(k).y \neq 0 :: \mathcal{D}(\mathit{Loader}(k).\lambda) \rangle \\ :: d' = \mathit{Loader}(j).\lambda) \textbf{ when } x = 0 \\ \sim d'.(\exists j, z :!\mathit{Unloader}(j) \wedge j = \langle \min k : \mathit{Unloader}(k).z = 0 :: \mathcal{D}(\mathit{Unloader}(k).\lambda) \rangle \\ :: d' = \mathit{Unloader}(j).\lambda) \textbf{ when } x \neq 0 \end{aligned}$$

Fig. 5. Finding the Closest Loader and Unloader

can see that using non-deterministic assignment statements to define context variables allows the program to define more flexible and expressive contexts in which to operate.

4.3 Context Consistency

Context UNITY not only provides the ability to give flexible and expressive specifications of context, but also provides the ability to specify the level of consistency between the system configuration and the program's abstraction of

context. A relationship between a program's context variable and the available global context can be defined as being *eager* or *lazy*. As the names suggest, an eager relationship enforces a high level of consistency between programs and the context while a lazy relationship has weaker semantics.

To fully explain the semantics of eager and lazy relationships, it is worth noting how the type of relationship is specified and realized. Eager relationships are specified using reactive statements. Reactive statements are evaluated each time any statement in the program is selected and executed. When reactive statements are used in a context definition or resolution statement, the result is a strong level of consistency between the program context variable and the system configuration. Conditional assignment statements with a **when** clause are used to specify lazy relationships. These statements are selected non-deterministically. If the statement is selected and the conditions exerted by the **when** clause are met, then the statement is executed. This results in a weak level of consistency, since the conditional assignment is selected at arbitrary points in the execution but infinitely often. Thus, by using a lazy statement in the context **resolve** section, it is possible that not every change in the program state will be projected onto the system configuration. Likewise, not every change to the system configuration will be reflected in the context variable values with the use of a lazy assignment statement in the context **define** section.

4.4 Context UNITY and Mobile UNITY

Notably missing from our Context UNITY program was Mobile UNITY's **Interactions** section. In Context UNITY, most of the interactions among programs are defined via the newly introduced **context** section. This is due to one of the key aspects of context-aware computing: the desire for the definition of asymmetric contexts that present context based on the program's unique perspective of the environment. The **Interactions** section may still be present in a Context UNITY program, but its use in such cases is outside the scope of this paper.

All of the constructs in Context UNITY presented in this paper can be reproduced in Mobile UNITY through the use of the **Interactions** section. Essentially, the context definitions and resolution statements of all programs in the system can be slightly modified and included in the Mobile UNITY **Interactions** section, thereby accomplishing the same task as the Context UNITY system. Because Context UNITY reduces directly to Mobile UNITY, we can reuse Mobile UNITY's proof logic entirely. Context UNITY, however, brings a notation tailored to context-aware computing, and the representation it offers lends itself more naturally to the representation of a system whose

purpose is to present context information to a single program operating in an environment composed of many heterogeneous, dynamic programs.

5 Discussion

The model discussed in the previous section highlights the basic concepts necessary to formalize context-aware computing. In the area of context-aware systems, this model can be used to describe many components already realized in applications and systems. For example, the context variables can be structured in such a way that they represent a fluid data structure defined by the data available in some portion of the system. Several mobile computing systems [8,14,19] view the data available in the system using such data structures. The distinction between private and exposed variables can even allow reasoning about context-aware programs that require secure communication.

Context UNITY as presented in this paper represents the introduction of a new model for context-aware computing in a mobile environment. The notation used to express context definitions and resolutions in our example programs, while quite expressive, may at times appear complex and difficult to read. Further refinements to the Context UNITY model will include constructs to simplify this notation; for example, the need to shadow variables to be able to react to their changing state will be included in the model. The full notation is shown in this paper to completely and clearly describe the steps necessary to model context-awareness. Section 4 also defined a few consistency semantics for reflecting context and projecting onto the context, specifically eager and lazy transfer semantics. Exploring additional possible semantics for these two relationships may provide programmers more flexibility and control over the definition and use of their defined context.

As we incorporate these and other refinements into the model, we will explore the inherited Mobile UNITY proof logic for correctness. We will continue to evaluate the reducibility of our additions to Mobile UNITY; some changes may result in required modifications to the proof logic. An evaluation of the proof logic, the expression of of further application scenarios in Context UNITY programs, and the refinements discussed above will all feed back in evaluating the completeness and expressiveness of our model.

6 Conclusion

This paper presents the basics of Context UNITY, a first step in building a formal model for explicit reasoning about context-aware computing. To present the model's notation, we relied on a simple context-aware application exam-

ple that allowed exposition of the key notational differences between Context UNITY and Mobile UNITY. The Context UNITY model introduces constructs necessary to specifically formalize key aspects of context-awareness, including the ability to interact with contexts defined by connected components and the ability to define a personalized context, specialized to a particular program's unique perspective. An interesting observation is the fact that most of the complexity is in the definition and manipulation of a program's interaction with the context. This is indeed in concert with our research goal, to simplify the development of context-aware applications. This initial exploration of the model highlights the need for context-aware specific constructs, and the resulting expressive Context UNITY shows great promise in providing a complete formal model for context-aware computing.

Acknowledgments

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and by the Office of Naval Research MURI Research Contract No. N00014-02-1-0715. 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 or the Office of Naval Research.

References

[1] Abowd, G., C. Atkeson, J. Hong, S. Long, R. Kooper and M. Pinkerton, *Cyberguide: A mobile context-aware tour guide*, ACM Wireless Networks **3** (1997), pp. 421–433.

[2] Back, R. J. R. and K. Sere, *Stepwise refinement of parallel algorithms*, Science of Computer Programming **13** (1990), pp. 133–180.

[3] Brown, P. J., *The stick-e document: A framework for creating context-aware applications*, in: *Proceedings of EP'96*, 1996, pp. 259–272.

[4] Cardelli, L. and A. Gordon, *Mobile ambients*, Theoretical Computer Science, Special Issue on Coordination **240** (2000), pp. 177–213.

[5] Chandy, K. M. and J. Misra, "Parallel Program Design: A Foundation," Addison-Wesley, NY, USA, 1988.

[6] Cheverst, K., N. Davies, K. Mitchell, A. Friday and C. Efstratiou, *Experiences of developing and deploying a context-aware tourist guide: The GUIDE project*, in: *Proceedings of MobiCom* (2000), pp. 20–31.

[7] Cugola, G., E. D. Nitto and A. Fuggetta, *The JEDI event-based infrastructure and its application to the development of the OPSS WFMS*, IEEE Transactions on Software Engineering **27** (2001), pp. 827–850.

[8] Cugola, G. and G. P. Picco, PeerWare*: Core middleware support for peer to peer and mobile systems*, Technical report, Politecnico di Milano (2001).

[9] Dey, A. K. and G. D. Abowd, *Cybreminder: A context-aware system for supporting reminders*, in: *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, 2000, pp. 172–186.

[10] Floyd, R. W., *Assigning meaning to programs*, in: *Proceedings of the Symposium on Applied Mathematics*, 1967, pp. 19–37.

[11] Harter, A. and A. Hopper, *A distributed location system for the active office*, IEEE Networks **8** (1994), pp. 62–70.

[12] Hoare, C. A. R., *Axiomatic basis for computer programming*, Communications of the ACM **12** (1969), pp. 576–580.

[13] Hoare, C. A. R., "Communicating Sequential Processes," Prentice-Hall International, 1985.

[14] Julien, C. and G.-C. Roman, *Egocentric context-aware programming in ad hoc mobile environments*, in: *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, 2002, pp. 21–30.

[15] McCann, P. J. and G.-C. Roman, *Compositional programming abstractions for mobile computing*, IEEE Transactions on Software Engineering **24** (1998), pp. 97–110.

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

[17] Milner, R., "A Calculus of Communicating Systems," LNCS **92**, Springer-Verlag, 1980 .

[18] Milner, R., "Communicating and Mobile Systems: The Pi Calculus," Cambidge University Press, 1999.

[19] Murphy, A. L., G. P. Picco and G.-C. Roman, LIME*: A middleware for physical and logical mobility*, in: *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001, pp. 524–533.

[20] Pascoe, J., N. Ryan and D. Morse, *Issues in developing context-aware computing*, in: *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, LNCS **1707**, 1999, pp. 208–221.

[21] Picco, G. P., G.-C. Roman and P. J. McCann, *Reasoning about code mobility in Mobile UNITY*, ACM Transactions on Software Engineering and Methodology **10** (2001), pp. 338–395.

[22] Rhodes, B., *Margin notes: Building a contextually aware associative memory*, in: *Proceedings of the 5th International Conference on Intelligent User Interfaces*, 2001, pp. 219–224.

[23] Roman, G.-C. and P. J. McCann, *A notation and logic for mobile computing*, Formal Methods in System Design **20** (2002), pp. 47–68.

[24] Ryan, N., J. Pascoe and D. Morse, *Fieldnote: A handheld information system for the field*, in: *Proceedings of the 1st International Workshop on TeloGeoProcessing*, 1999, pp. 155–163.

[25] Salber, D., A. Dey and G. Abowd, *The Context Toolkit: Aiding the development of context-enabled applications*, in: *Proceedings of CHI'99*, 1999, pp. 434–441.

[26] Schilit, B., N. Adams and R. Want, *Context-aware computing applications*, in: *IEEE Workshop on Mobile Computing Systems and Applications*, 1994, pp. 85–90.

[27] Want, R. et al., *An overview of the PARCTab ubiquitous computing environment*, IEEE Personal Communications **2** (1995), pp. 28–33.