Modeling Adaptive Behaviors in Context UNITY

Gruia-Catalin Roman

Department of Computer Science and Engineering
Washington University in Saint Louis
roman@wustl.edu

Christine Julien

Department of Electrical and Computer Engineering
The University of Texas at Austin
c.julien@mail.utexas.edu

Jamie Payton

Department of Computer Science
University of North Carolina
payton@uncc.edu

Abstract

Context-aware computing refers to a paradigm in which applications sense aspects of the environment and use this information to adjust their behavior in response to changing circumstances. In this paper, we present a formal model and notation (Context UNITY) for expressing quintessential aspects of context-aware computations; existential quantification, for instance, proves to be highly effective in capturing the notion of discovery in open systems. Furthermore, Context UNITY treats context in a manner that is relative to the specific needs of an individual application and promotes an approach to context maintenance that is transparent to the application. In this paper, we construct the model from first principles, introduce its proof logic, and demonstrate how the model can be used as an effective abstraction tool for context-aware applications and middleware.

Key words: context-aware computing, UNITY, adaptive systems, formal methods

1 Introduction

Context-aware computing is a natural next step in a process that started with the merging of computing and communication during the last decade and continues with the absorption of computing and communication into the very fabric of our society and its infrastructure. The prevailing trend is to deploy systems that are increasingly sensitive to the context in which they operate. Flexible and adaptive designs allow computing and communication to blend into the application domain, making computers gradually less visible and more agile. Context-awareness enhances a system's ability to become ever more responsive to the needs of the end-user or application domain. In an aware home, context can be used to change applications' behavior based on a room's occupants or the time of day. In an automotive application, context such as the density of traffic or the speed of the car may change the application's desired functionality. With the growing interest in adaptive systems and the development of tool kits [1,2] and middleware [3] supporting context-awareness, one no longer needs to ponder whether context-aware computing is emerging as a new paradigm, i.e., a new design style with its own specialized models and support infrastructure. However, it would be instructive to develop a better understanding of how this transition took place, i.e., what distinguishes a design that allows a system to adapt to its environment from a design that could be classified as employing the context-aware paradigm. This is indeed the central question addressed in this paper. We seek to develop an understanding of context-aware computing by proposing a simple abstract conceptual model of context-awareness and formalizing it. This model focuses on the specification and verification of context-aware systems, an imperative step in a careful design process. The provision of such a model not only supports the ability to formally express and reason about existing perspectives on context-awareness, but also promotes methodical design of new applications that embody a context-aware design.

The term context-awareness immediately suggests a relation between an entity and the setting in which it functions. Let us call such an entity the *reference agent*—it may be a software or hardware component—and let us refer to the sum total of all other entities that could (in principle) affect its behavior as the reference agent's *operational environment*. We differentiate the operational environment from the *context* by drawing a distinction between potentiality and relevance. While all aspects of the operational environment have the potential to influence the behavior of the reference agent, only a subset is actually relevant to the reference agent's behavior. In formulating a model of context-awareness, we focus our attention on how this relevant subset is determined.

This paper presents Context UNITY, which, to the best of our knowledge, represents the first general formal model of context-aware computing. This

model has its roots in our earlier work on Mobile UNITY [4,5] and in our experience with developing context-aware middleware for mobility [3,6,7]. Context UNITY assumes that the universe (called a system) is populated by a bounded set of agents whose behaviors can be described by a finite set of program types. At the abstract level, each agent is a state transition system, and context changes are perceived as spontaneous state transitions outside of the agent's control. However, the manner in which the operational environment can affect the agent state is an explicit part of the program definition. A context definition is therefore explicitly included in a program type description; it is specific to the dynamic needs of each agent and is separate from the behavior exhibited by the agent. In this way, the agent's formalization is self-contained, i.e., local in appearance and totally decoupled from that of all the other agents in the system. Key to the separation of behavioral and contextual concerns and among agent specifications is the reliance on existential quantification as an abstraction of the context discovery process. The design of the Context UNITY notation is augmented with an assertional style approach to verification facilitating formal reasoning about context-aware programs.

The remainder of this paper is organized as follows. Section 2 discusses the basic requirements for a model of context-awareness and explains why not every model that exploits contextual information is appropriate for exploring the fundamentals of context-aware computing. Section 3 presents our formalization of context-awareness, explaining both the model's organization and the principles that governed our specific choices. Because our ultimate goal is a better understanding of context-aware computing, we seek minimality of concepts and elegance of notation while remaining faithful to our perspective on context-awareness made explicit in Section 2. Section 4 shows how the model can express key features of several existing context-aware applications. In Section 5, we outline the verification techniques associated with the model and explore both their strengths and limitations. Conclusions appear in Section 6.

2 Problem Definition and Model Requirements

We next examine the key requirements of context-awareness that must pervade a formal model. A formalization that meets these requirements will achieve not only the expressiveness required of adaptive and interactive applications but also relevance to widely varied application domains.

- *Expansiveness*: A model of context-awareness must recognize the fact that distant entities in the operational environment can affect an agent's behavior [8]. The model should not place *a priori* limits on the scope of the context associated with a particular agent, though specific instantiations of the model may impose restrictions due to pragmatic considerations relating

to the cost of context maintenance or the nature of physical devices.

- *Specificity*: To balance expansiveness and allow agents to exercise control over the cost of context maintenance, the model must allow context definitions to be tailored to the needs of each agent. Furthermore, as agents' needs evolve, context definitions should be amenable to modification. Together with expansiveness, specificity ensures the model's generality.
- *Explicitness*: The previous requirements fail to consider *how* an agent forms and manipulates its context. The only way an agent can exercise such control is to have an explicit notion of context. This allows the agent to define and change its context definition as best suits its processing requirements.
- *Separability*: An agent's context definition must be an identifiable element of a formal model of context-awareness, and the context definition must capture the essential features of the agent/context interaction pattern. The agent's changes to its context definition(s) should be readily understood without examining the details of the agent's behavior.
- *Transparency*: Finally, the definition of context must be sufficiently abstract to free the agent from the operational details of discovering its own context and sufficiently precise for some underlying support system to be able to determine what the context is at each point in time.

These requirements frame our perspective on context-awareness. To illustrate this perspective, we examine an application in which context plays an important role, but the criteria for employing the context-aware paradigm are not met. Consider an agent that receives and sends messages and learns about the presence of other agents through these messages. The agent adapts its behavior based upon the knowledge it gains about its context. The agent implicitly builds an acquaintance list of other agents in the region and updates its knowledge using message delivery failures that indicate agent termination or departure. We do not view this as an instance of the context-aware paradigm; an agent's interaction with environment is expansive but it is not specific, explicit, separable, or transparent. We next detail what is required to transform this application into one that exemplifies the context-aware paradigm.

Specificity could be achieved by allowing each agent to individually filter which agents should be included in its *acquaintance list*. Explicitness could be realized by including a distinct acquaintance list—a concrete representation of the agent's current context—as an explicit data structure within the agent's code. Separability could come from designing the code that updates the acquaintance list to automatically extract agent information from arriving messages, e.g., through the interceptor pattern [9]. Transparency requires the agent to delegate the updating of the acquaintance list to an underlying infrastructure; this, in turn, demands that the definition of the context be made explicit to the support infrastructure. The result is an application that exhibits the same behavior but a different design style; the agent views and interacts with its context through a data structure that appears to be local, is automatically up-

dated, and is defined by the agent's personalized admission policy controlling which agents are included in the list.

Context UNITY formalizes applications that follow this style of context-aware design. The model helps a developer frame his design in terms of the requirements described above, ensuring that it conforms to principled context-aware design and allowing rigorous validation of the resulting application.

3 Formalizing Context-Awareness

Applications modeled in Context UNITY adhere to the perspective of context-awareness outlined above. This section begins with an overview of Context UNITY, highlighting its key concepts. We present the model's notation in detail and demonstrate its application through an example. Finally, we discuss the special properties of Context UNITY variables, which aid in providing and discovering context, and explain how an agent specifies its context.

3.1 Model Overview

In the dynamic mobile environments where context-aware applications are prevalent, mobile hosts opportunistically form networks with changing topologies. Applications reside on these mobile hosts, and, in our computational model, the applications are encapsulated as logically mobile agents that may migrate among connected hosts. Each agent provides context information to other reachable agents that may impact the agents' actions, and may utilize the context information provided by other agents. When discussing a particular agent's context, we refer to the agent as the *reference agent*. We apply a general and expansive approach to defining an agent's context, yet the reference agent can tailor its context based on properties of the environment and the information itself. Fig. 1 depicts our computational model from a single reference agent's perspective.

In Context UNITY, a complete application is represented as a community of interacting agents that capture an application's behavior in a *system specification*. A system structures an application into component types which describe agent behavior, the instantiation of these types as application components, and application-wide context interactions. Each agent's behavior is described by a *program* prototype. A program explicitly separates an agent's behavior from management of its context interactions. Programs are instantiated separately within a Context UNITY system, with each instance defining an application agent. Multiple instantiations of the same program are differenti-

ated by a unique program identifier. Because some context interactions may apply to all programs, it is also possible to specify uniform context interactions within a Context UNITY system.

A Context UNITY program represents all of an agent's state using variables, which allows complex context-aware actions to be modeled via simple variable assignments. Like UNITY [10], Context UNITY's execution model selects program statements for execution in a weakly-fair manner—in an infinite execution, each assignment statement in a system is selected for execution infinitely often. To ensure fairness, a Context UNITY system must be comprised of a finite number of program statements, requiring that all instantiations of a program participating in a system must be specified in advance. While this approach may seem limiting at first, it is possible to simulate a more dynamic view by instantiating a large but finite number of program instances in a system beyond the expected need of the application and activating the instances as needed using the effects of assignment statements.

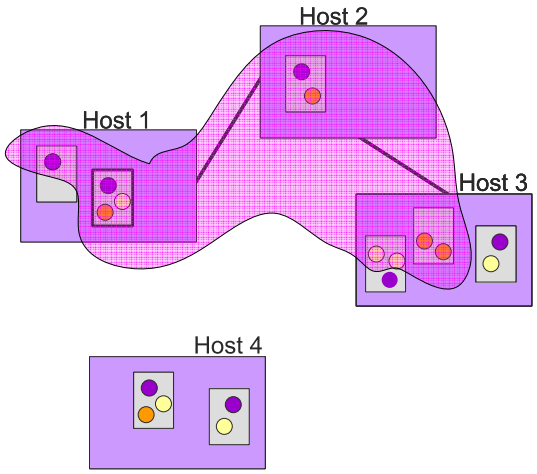

Fig. 1. Computational model. Hosts (large rectangles) serve as containers of agents (smaller rectangles), which provide data (circles). Bold lines illustrate physical connectivity. The reference agent is denoted by the heavily outlined rectangle on Host 1, and its context includes information within the shaded cloud. Because Host 4 is not connected to Host 1, its information is not eligible for inclusion in the context.

Each Context UNITY program can use three types of statements: simple assignment statements, which assign a value to a variable; transactions, which execute multiple simple assignment statements in an atomic step; and reactions, which execute in response to a specified change in the state of the system. In Context UNITY, variables are used both to represent program state and to facilitate an agent's interaction with its context. Three categories of variables appear in programs: *internal, exposed,* and *context* variables. *Internal variables* hold private data that the agent does not share; they do not affect the operational environment of other agents. *Exposed variables* store public data; the values of these variables can contribute to other agents' contexts. Finally, *context variables* reflect the agent's context and can be used both to gather information from the exposed variables of other agents and to push data out to the exposed variables of other agents. The actions of context variables are

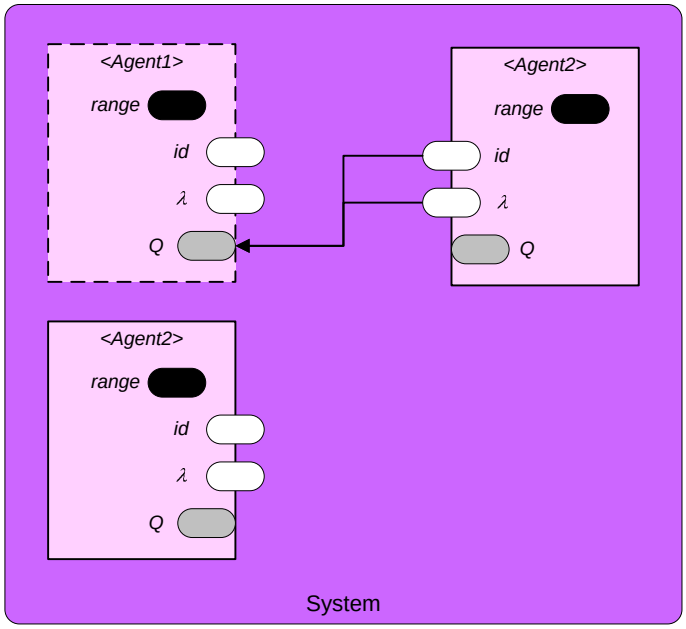

Fig. 2. A Context-Aware Acquaintance List Modeled in Context UNITY

governed by context rules specified by each agent. Assignment statements in an agent's internal behavior specification can include references to any of the three types of variables, allowing the state of internal and exposed variables to be influenced by both internal state and state from the environment.

Due to the unpredictable nature of dynamic context-aware application environments, the Context UNITY model must handle the lack of *a priori* knowledge about an agent's operational environment when utilizing exposed variables in the agent's context specification. To meet this need, Context UNITY employs *non-deterministic assignment statements* and *existential quantification* in context definitions. These mechanisms allow agents that contribute to a context to be discovered based on attributes defined within their exposed variables. Context UNITY provides additional flexibility by allowing an agent to specify the consistency with which its context variables reflect the environment. Rules can be defined to operate in one of two modalities: normal or reactive. Normal context rules are selected for execution in a weakly-fair manner, while reactive context rules reflect a stronger level of consistency, which is demonstrated in more detail later.

To illustrate the use of programs, their instantiations, and their uniformly applied context interactions, we return to the acquaintance list example, which maintains a set of ids of agents operating on hosts within communication range. This example captures a support task utilized by many context-aware applications; several context-aware systems, e.g., Limone [7], use such a data structure as a basis for coordination. Fig. 2 illustrates the application of Con-

text UNITY to describe an application that uses an acquaintance list. This system consists of three agents of two differing types. Each agent stores its unique agent id in an exposed variable. Because we are modeling systems that entail agent mobility, each agent also has an exposed variable (λ) that stores its location. Movement of the agent is outside the scope of this example; it may occur through local assignment to the location variable or by a global controller via a system's uniform context definitions. Each agent declares a context variable Q of type set to store the contents of the acquaintance list. Each program type (in this case, *Agent1* and *Agent2*) employs different eligibility criteria for the members of its acquaintance list, exemplified in the context rules provided in each program type that describe how the context variable Q is updated. As shown in the figure, the context rule defined by *Agent1* uses the exposed location variable of agents of type *Agent2* to determine if the agent is within a prescribed *range* (stored in an internal variable). If the agent is within range, its id is added to the reference agent's acquaintance list by updating the context variable Q. In the case shown in the figure, the agent at the bottom is not within range, so it is not reflected in *Agent1*'s acquaintance list.

3.2 Context UNITY Notation

The notation used to represent a **System** is shown in Fig. 3. The first portion of this definition lists textual descriptions of programs that specify the behavior of the agent types. The **Components** section declares the *instances* of programs, or agents, present in the application. These declarations refer to program names, arguments, and a function (*new_id*) that generates a unique id for each agent declared.

The **Governance** section captures uniform system interactions that can impact exposed variables in all programs in the system. The details of an entire system specification will be made clearer through examples later in this section.

System *SystemName*
 Program *ProgramName* (*parameters*)
  declare
   internal — *internal variable declarations*
   exposed — *exposed variable declarations*
   context — *context variable declarations*
  initially — *initial conditions of variables*
  assign — *assignments to declared variables*
  context
   definitions affecting context variables—
    they can pull information from and
    push information to the environment
 end
 . . . additional program definitions . . .
 Components
  the agents that make up the system
 Governance
  global impact statements
end *SystemName*

Fig. 3. A Context UNITY Specification

Each Context UNITY program's **declare** section lists the variables defining its individual state. The declaration of each variable makes its category evident (internal, exposed, or context). The **initially** section defines what values the variables are allowed to have at the start of the program. The **assign** section defines how variables capturing the program's internal state are updated. Assignment statements can include references to any of the three types of variables but can assign to only internal or exposed variables (since context variables simply reflect some state of the environment). To provide a measure of control over the execution of assignment statements, two additional assignment constructs introduced in Mobile UNITY are also available in addition to simple assignment statements. A *transaction* has the notation $\langle s_1; s_2; \ldots; s_n \rangle$ and specifies a sequence of simple assignment statements which must be scheduled in the specified order with no other (non-reactive) statements interleaved. It captures a sequential execution whose net effect is a large-grained atomic state change. Transactions are selected for execution in the same weakly-fair manner as normal statements. A *reaction*, denoted s **reacts-to** Q, allows a program to respond to changes in the system's state as given by an enabling condition Q (where s is an assignment statement).

Context UNITY introduces context variables and a **context** section to contain the rules that manage an agent's interaction with its context. The **context** section explicitly separates the management of an agent's context from its internal behavior. Specifically, the **context** section contains definitions that sense information from the operational environment and store it in the agent's context variables. The rules also allow the agent to affect the behavior of other agents in the system by impacting their exposed variables (i.e., the context section allows changes in the state of the environment to be reflected in the values of an agent's exposed variables which can then affect other agents whose context variables rely on those exposed variables). The acquisition and provision of context in an environment full of unknown participants is achieved in context rules by selecting exposed variables according to constraints on their attributes.

A context rule is selected for execution like any other statement. The Context UNITY execution model, like its UNITY ancestor, exhibits weakly-fair selection of statements for execution. As in Mobile UNITY, Context UNITY adopts a modified form of UNITY's execution model to accommodate reactive behavior. Normal statements, i.e., all statements other than reactions, continue to be selected for execution in a weakly-fair manner. After execution of a normal statement, the set of all reactions in the system forms a *reactive program* that executes until it reaches *fixed-point*. The reactive program itself is a terminating UNITY program for which a fixed point predicate can be computed. During the reactive program's execution, the reactive statements are selected for execution in a weakly-fair manner while all normal statements are ignored. When the reactive program reaches a fixed-point, the weakly-fair

selection of normal statements continues.

We return to the acquaintance list application to illustrate the structure of a system specification. Fig. 4 provides the Context UNITY specification for a context-aware application that relies on the usage of an acquaintance list. The system specification first describes the agent types that utilize context to build acquaintance lists. Both program type definitions begin with identical **declare** sections (the specifics for *Agent2* are omitted for brevity). This section defines two exposed variables (the agent's id and location). Both *id* and λ are local handles for these exposed variables whose names are agent_id and location, respectively. In general, the **declare** section of both program types uses the notation l ! n : t to define an exposed variable with local handle l and publicly accessible name n of the given type t. Both **declare** sections also define the context variable, Q, used to store the context-sensitive acquaintance list. Q is defined using the notation: *local_handle* : type, where type is the type of the variable. In this case, the local handle is Q and the type is a set of agent_ids. Because context variables and internal variables use the same simple structure for representation, the same notation is applied in the definition of the program's internal variables. While each agent type has individualized behavior defined via the **assign** section that may use context variables once they are defined, these details are omitted. The interesting aspect of this example is the use of context and exposed variables to define an acquaintance list. In each program type, the **context** section defines rules that dictate how properties of exposed variables of other agents are selected and used to update the context variable Q. Context rules for the variable Q are presented in section 3.4.4, which details context specifications.

System *AcquaintanceManagement*
Program *Agent1*
declare
exposed *id* ! agent_id : agent_id
λ ! location : location
context Q : set of agent_id
assign
... *definition of local behavior* ...
define
define Q based on desired properties of acquaintance list members
end
Program *Agent2*
... similar to *Agent1* ...
end
Components
Agent1[*new_id*], *Agent1*[*new_id*],
Agent2[*new_id*]
end *AcquaintanceManagement*

Fig. 4. A Context-Aware System for Acquaintance Maintenance

3.3 Variables Revisited

Context UNITY programs represent state and context using variables and assignment. The unique needs of context-aware applications necessitate a re-examination of variable representation and what state is required across all

ι	the variable's unique id
π	the id of the owner agent
η	the name
τ	the type
ν	the value
α	the access control policy

Fig. 5. Components of an Exposed Variable

programs to support specification in the Context UNITY model. We address these issues in the remainder of this section.

3.3.1 Exposed Variable Structure

In UNITY and many of its descendants, a variable is simply a reference to an object which holds a value. In Context UNITY, both internal and context variables adhere to this standard. However, because the handle names of variables have no meaning outside the scope of the program, references to exposed variables appearing in the program text are actually references to more complex structures needed to support context-sensitive access within an unknown operational environment. A complete semantic representation of exposed variables is depicted in Fig. 5. Each attribute of an exposed variable is examined in detail below:

- Each exposed variable has a unique id ι used to provide a handle to the specific variable. Uniqueness can be ensured by making each variable unique within an agent and combining ι with the unique agent id. This variable id is assigned at component instantiation and cannot be changed.
- The element π of type agent_id designates the agent owning the variable and allows an exposed variable to be selected based on its owner.
- An exposed variable's name, η, acts as a short descriptor that identifies the variable's role in the application; this name can be changed by the program's assignment statements.
- The variable's type τ allows the variable to be selected according to its type, e.g., integer, set, and so on. The variable's type is immutable.
- An exposed variable's value, ν, refers to the variable's data value. The value of an exposed variable can be assigned in the **assign** section of a program or can be determined by another program's impact on its context.
- The program can control the extent to which other agents access its exposed variables using the variable's access control policy, α, which determines access based on properties of the particular agent attempting access. α accepts the reference agent's credentials as parameters and returns the set of allowable operations on the variable, e.g., {r, w} signifies permission to both read and write. Credentials are described in more detail later; briefly, this

approach models the finest-grained access restrictions possible and supports policies which meet the needs of current context-aware systems.

3.3.2 Built-in Variables

Context UNITY programs contain three built-in exposed variables, each of which is essential to context-aware program behavior in our model. These exposed variables are automatically declared and have default initial values. An individual program can override the initial values in the program's **initially** section and can assign and use the variables throughout the **assign** and **context** sections. The first of these exposed variables has the name "location" and facilitates modeling mobile context-aware applications by storing the location of the program owning the variable. The definition of location can be based on either a physical or logical space and can take on many forms. This style of modeling location is identical to that used in Mobile UNITY. The second built-in exposed variable has the name "type", and its value is the program's name (e.g., "*Agent1*" or "*Agent2*" in the example system). The use of this variable can help context variables select programs based on their general function. The third of the built-in exposed variables has the name "agent_id" and holds the unique identifier assigned to the agent when the agent is instantiated in the **Components** section. This variable cannot be modified.

In addition to exposed variables that represent a program's location, type, and id, Context UNITY programs contain an internal built-in variable fundamental to the model's approach to controlling access to exposed variables. This built-in internal variable has the local handle "credentials", and is used to store a profile of program's attributes (e.g., passwords, certificates, etc.). Like the other built-in variables, a Context UNITY program's credentials variable is automatically declared with default initial values. Its value can be changed in the program's **assign** section and can be used in the program's **context** section. The value of the credentials variable is provided as a parameter to the access control policies of the exposed variables of other programs. Essentially, a reference agent communicates its credentials to the remote agent, which uses the credentials to determine whether or not the reference agent has access to a particular exposed variable. For example, an agent may require access to a password protected file owned by another agent. The remote agent evaluates its access control function over the reference agent's credentials variable, which must have a field containing the correct password to gain access to the file.

For reasons of mathematical convenience, the Context UNITY model supplies the remote agent with all attribute values of the reference agent's credentials variable. This approach offers an abstract, general purpose construct that relies on the use of a single concept to evaluate the satisfaction of access control policies. A more sophisticated model of access control could apply a projec-

tion to the credentials variable to extract only the parameters needed in the evaluation of the access control policy. In any case, a practical implementation of Context UNITY's approach to access control would only deliver appropriate fields of the variable to the requester, and would likely use encryption and authentication techniques to ensure secure transmission of such sensitive information.

3.4 Context Specification

Context-aware applications rely on conditions in the environment for adaptation. Context UNITY facilitates specification of context interactions through the use of context variables that use the exposed variables of other agents to provide exactly the context that a reference agent requires. In a Context UNITY program, the **context** section of a program contains the rules that dictate restrictions over the operational environment to define the context over which an agent operates. Additionally, the rules in the **context** section allow the agent to feed back information into its context. Structuring the **context** section as a portion of each program allows agents to have explicit and individualized interactions with their contexts.

In the remainder of this section, we examine the techniques utilized in Context UNITY to capture context rules. We begin with an overview of how context rules define a particular agent's context, and introduce the mechanisms used to support context-sensitive selection of the exposed variables. We then discuss how an agent's data is protected through the inclusion of context-sensitive access control restrictions. Next, we illustrate the use of the Context UNITY notation in capturing an agent's context rules based on more complex restrictions applied to properties of exposed variables. Finally, we address specification of uniform context rules that apply to all programs within a Context UNITY system. Throughout the section, we provide precise definitions of Context UNITY's context specification constructs.

3.4.1 Context-sensitive Selection of Exposed Variables

Requiring a reference agent to explicitly refer to another programs' exposed variables to define its context requires the agent to have advance knowledge about any other components it might encounter over time. Programs rarely have such *a priori* knowledge; in fact, typical context-aware applications rely on opportunistic interactions that cannot be predetermined. To capture this in Context UNITY, exposed variables that contribute to a context definition are selected in a context-sensitive manner using *existential quantification* and *non-deterministic assignment statements*. Existential quantification

allows agents to refer to an agent without advance knowledge of its participation or context values. Non-deterministic assignment allows the reference agent to descriptively select which variables belonging to other agents affect its behavior based on the attributes defined in the exposed variables of those agents. The non-deterministic assignment statement $x := x'.Q$ assigns to x a value x' non-deterministically selected from all values satisfying the condition Q [11]. As a simple example of non-deterministic assignment, consider a statement $x := x'.Q$ whose condition Q dictates the selection of an exposed variable (owned by some agent in the system) within the numerical range 1 to 9. Several values may be available which fit the description, but only one is chosen and assigned to x. In a more practical example used in a mobile context-aware application, an agent uses the built-in Context UNITY location variable to store its current physical location; an agent captures its movement by updating this variable using an assignment statement in the local **assign** section. Another agent can use an existentially quantified non-deterministic assignment statement in which the relative distance between the reference agent's location and the exposed location variables of other agents is used as a condition to identify which other agents are to contribute to the reference agent's context.

Context UNITY wraps the use of non-deterministic assignments in a specialized notation. To manage its interaction with context information, a program uses statements of the following form in its **context** section:

c **uses**	*quantified variables*
given	*restrictions on variables*
where	*c* **becomes** *expr*
	$expr_1$ **impacts** $exposed\ variable_1$
	$expr_2$ **impacts** $exposed\ variable_2$
	...
[**reactive**]	

This expression, a *context rule*, governs the interactions associated with the context variable *c*. A context rule first declares existentially quantified dummy variables to be used in defining the interactions with exposed variables. The scope of these dummy variables is limited to the context rule that declares them. The expression can refer to any exposed variables in the system by applying a context-sensitive selection mechanism to the restrictions (constraints on the attributes of the selected exposed variables, as specified using non-deterministic assignment statements) provided in the rule's definition. The context rule can define an expression, *expr*, over the selected set of exposed variables and any locally declared variables (internal, exposed, or context). The result of evaluating this expression is assigned to the context variable. The context rule can also define how this context variable impacts the operational environment. These impacts statements are much like assignment statements written in reverse, where a rule outside of a program can change the value of a variable within that program. If no combination of variables in

the system satisfies the restrictions specified in the context rule, the dummy variables in the expression are undefined, and the rule reduces to a skip.

The execution of each context rule can optionally be declared to be **reactive**, which dictates that the context rule reflects the environment with a strong degree of consistency. In fact, when a change occurs anywhere in the Context UNITY system, execution in the system is logically halted, all reactive context rules are evaluated, and normal execution of statements in the system resumes.

Reactive context rules are actually defined as reactive assignment statements that are utilized in Context UNITY programs. If a context rule is declared **reactive**, it becomes part of the system's reactive program that is executed to fixed-point after the execution of each normal statement. Using a reaction guarantees that the context information expressed by the rule remains consistently up to date because no normal statements can execute until the reactive program reaches fixed-point. If not declared **reactive**, the context rule is a normal, unguarded statement and part of Context UNITY's normal execution model. Representation of context rules in Context UNITY notation along with formal definitions which precisely explain the semantics of context rules, reactive and otherwise, are presented in the following sections.

3.4.2 Access Control Restrictions in Context-Sensitive Selection

Within a context rule, even if no explicit restrictions are placed on the referenced exposed variables, two restrictions are automatically assumed. The first requires that any variable referenced be an exposed variable. The second implicit restriction requires that the program whose context uses an exposed variable must satisfy the variable's access control policy.

Consider the following simple context rule that acquires the value of some exposed variable, places the value in the context variable c, and deletes the value from the exposed variable used. The context rule is a reactive statement triggered when a is larger than the value of some local variable x:

$$
\begin{array}{ll}
c \textbf{ uses} & a \\
\quad \textbf{given} & a > x \\
\quad \textbf{where} & c \textbf{ becomes } a \\
 & 0 \textbf{ impacts } a \\
\quad \textbf{reactive} &
\end{array}
$$

This context rule corresponds to the following formal definition, which includes the two implicit restrictions on the exposed variable a as discussed above:

$$
\begin{array}{l}
\langle || a : \ a = a'.(\mathsf{var}[a'] > x \wedge \{r, w\} \subseteq \mathsf{var}[a'].\alpha(\mathit{credentials})) \\
\quad :: (c := \mathsf{var}[a].\nu \ || \ \mathsf{var}[a].\nu := 0) \ \textbf{reacts} - \textbf{to } \mathit{true} \\
\rangle^1
\end{array}
$$

In this definition, we introduce var, a logical table that allows us to refer to all variables in the system, referenced by the unique variable id. When the variable a is selected from var in the statement above, what is actually selected is a's variable id, which references a specific entry in the table. In this statement, a single exposed variable is non-deterministically selected from all exposed variables whose access control policies allow the reference agent to read and write the exposed variable referred to by the dummy variable a. This requires applying the exposed variable's access control policy to this agent's credentials; the set of permissions returned by the evaluation of the access control function α can contain any combination of r (indicating read permission) and w (indicating write permission). After selecting the particular exposed variable to which a refers, the rule contains two assignments. The first assigns the value stored in a (i.e., $\mathsf{var}[a].\nu$) to the context variable c. The second assignment captures the fact that the context rule can also impact the environment, in this case by zeroing out the exposed variable used.

3.4.3 Utilizing Exposed Variable Attributes in Context-sensitive Selection

The power of the context-sensitive selection of exposed variables becomes apparent when the restrictions within the context rules are utilized to describe properties of desired context information. The context rule can specify restrictions to select exposed variables based on the exposed variables' names, types, values, owning agents, or even based on properties of other variables belonging to the same or different agents. To simplify the specification of these restrictions, we introduce several new pieces of notation.

Referring to the system-wide table var is cumbersome and confusing because the table is both virtual and distributed. For this reason, context rules refer instead to indices in the table. We allow the variable id a to denote the value of the variable in var for entry a, i.e., $\mathsf{var}[a].\nu$. To access the other components of the variable (e.g., name), we abuse the notation slightly and allow, for instance, $a.\eta$ to denote $\mathsf{var}[a].\eta$. Context rules frequently utilize a variable's descriptive name to select exposed variables. As such, we use the shorthand $x \,!\, y$ to indicate that the exposed variable referenced by the dummy variable x must have the name y, i.e., $\mathsf{var}[x].\eta = y$. Since a common operation in context-sensitive selection is to select variables that exist within the same

[1] The three-part notation $\langle$**op** *quantified_variable* : *range* :: *expression*$\rangle$ 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 $\forall$, zero if **op** is "+," or *skip* if **op** is ||.

agent, we also introduce a shorthand for accessing a variable by the combination of name and program. When declaring dummy variables, a context rule can restrict both the names and relative owners of the variables using the notation: $x \,!\, \mathsf{name}_1, y \,!\, \mathsf{name}_2$ *in* $p; z \,!\, \mathsf{name}_3$ *in* q. This notation refers to three variables, one named name_1 and a second named name_2 that both belong to the same agent whose agent_id can be referenced as p. The third variable, z, must be named name_3 and located in program q. q may or may not be the same agent as p, depending on further restrictions that might be specified.

As a simple example of a context rule, consider a program with a context variable c that holds the value of an exposed variable with the name data located on an agent at the same location as the reference. This context variable does not change the data stored on the agent owning the exposed variable. To achieve this behavior, the specification relies on the built-in exposed variable λ. The context rule for c uses a single exposed variable that refers to the data that will be stored in c. In this example, we leave the rule unguarded, and it falls into the set of normal statements that are executed in a weakly-fair manner.

$$\begin{array}{ll} c \textbf{ uses} & d \,!\, \mathsf{data}, l \,!\, \mathsf{location} \textit{ in } p \\ \textbf{given} & l = \lambda \\ \textbf{where} & c \textbf{ becomes } d \end{array}$$

Formally, using the above notation is equivalent to the following expression:

$$\begin{array}{l} \langle || d, l : (d, l) = (d', l').(\{r\} \subseteq \mathsf{var}[d'].\alpha(\mathsf{credentials}) \wedge \{r\} \subseteq \mathsf{var}[l'].\alpha(\mathit{credentials}) \wedge \\ \qquad\qquad\qquad\qquad \mathsf{var}[d'].\eta = \mathsf{data} \wedge \mathsf{var}[l'].\eta = \mathsf{location} \wedge \\ \qquad\qquad\qquad\qquad \mathsf{var}[d'].\pi = \mathsf{var}[l'].\pi \wedge \mathsf{var}[l'].\nu = \lambda.\nu) \\ \quad :: c := \mathsf{var}[d].\nu \\ \rangle \end{array}$$

Because the expression assigned to the context variable c is simply the value of the selected exposed variable, the most interesting portion of this expression is the non-deterministic assignment statement that selects the exposed variables. The formal expression non-deterministically selects a variable (referred to by the dummy variable d) that satisfies a set of conditions, which rely on the selection of a second exposed variable (referred to by the dummy variable l) that stores the program's location. The first line of the non-deterministic selection checks the access control function for each of the variables to ensure that this agent is allowed read access given its credentials. The second line restricts the names of the two variables. The variable d being selected must be named data, according to the restrictions provided in the rule. The location variable is selected based on its name being location. The final line in the non-deterministic selection deals with the locations of the two variables. The first clause ($\mathsf{var}[d'].\pi = \mathsf{var}[l'].\pi$) ensures that the two variables (d and l) are located in the same program instance (agent). The second clause ensures that the agent that owns these two variables is at the same location as the agent defining the rule.

To show how these expressions can be used to model real-world interactions, we revisit the acquaintance list example from earlier in the section. Previously, we gave a high level description of the context rules required to define an agent's acquaintance list. To define the membership qualifications, the agent uses a context rule that adds qualifying agents to the context variable Q. In this case, assume that the program restricts acquaintance list members to other agents within some predefined range. This range is stored in an internal variable whose local handle is *range*. Q is defined using the following rule:

Q **uses**	l ! location *in* a
given	$\lvert l - \lambda \rvert \leq$ *range*
where	Q **becomes** $Q \cup \{a\}$
reactive	

This expression uses the two handles *range* and λ to refer to local variables that store the maximum allowable range and the agent's current location, respectively. This statement adds agents that satisfy the membership requirements to the acquaintance list Q one at a time. Because it is reactive, the rule ensures that the acquaintance list remains consistent with the state of the environment. As a portion of the reactive program that executes after each normal statement, this context rule reaches fixed-point when the acquaintance list contains all the agents that satisfy the requirements for membership. An additional rule is required to eliminate agents that might still be in Q but are no longer in range:

Q **uses**	l ! location *in* a
given	$\lvert l - \lambda \rvert >$ *range*
where	Q **becomes** $Q - \{a\}$
reactive	

The result is a readable, explicit, and separable definition of a context-sensitive acquaintance list. More extensive examples illustrating context-sensitive selection using constraints on exposed variable attributes will be discussed in Section 4.

3.4.4 Specifying a Uniform Context

The final portion of a Context UNITY system specification is a **Governance** section, which contains rules that capture behaviors that have universal impact across the system. These rules use the exposed variables available in programs throughout the system to affect other exposed variables in the system. The rules have a format similar to the definition of a program's local context rules except that they do not affect individual context variables:

use	*quantified variables*
where	*restrictions on quantified variables*
	$expr_1$ **impacts** *exposed variable*$_1$
	$expr_2$ **impacts** *exposed variable*$_2$
	...

As a simple example of governance, imagine a central controller that non-deterministically chooses an agent in the system and moves it. This example assumes a one-dimensional space in which agents are located; essentially the agents can move along a line. Each agent's built-in location variable stores the agent's position on the line, and another variable named direction indicates which direction along the line the agent is moving. If the value of the direction variable is $+1$, the agent is moving in the positive direction; if the value of the direction variable is -1, the agent is moving in the negative direction. We arbitrarily assume the physical space for movement is bounded by 0 on the low end and 25 on the upper end. The governance rule has the following form:

$$\begin{array}{ll} \textbf{use} & d \,!\, \mathsf{direction}, l \,!\, \mathsf{location}\ \mathit{in}\ p \\ \textbf{where} & l + d\ \textbf{impacts}\ l \\ & (\mathit{if}\ l + d = 25 \vee l + d = 0\ \mathit{then}\ -d\ \mathit{else}\ d)\ \textbf{impacts}\ d \end{array}$$

The non-deterministic selection clause chooses a d and l from the same program with the appropriate variable names. The first of the impact statements moves the agent in its current direction. The second impact statement switches the agent's direction if it has reached either boundary. The rules placed in the **Governance** section can be declared reactive, just as a local program's context rules are. The formal semantic definition of context rules in the **Governance** section differs slightly from the definition outlined above in that the governance rules need not account for the access control policies of the referenced exposed variables. This is due to the fact that the specified rules define system-wide interactions that are assumed, since they are provided by a controller, to be safe and allowed actions. As an example, the formal definition for the rule described above would be:

$$\begin{array}{l} \langle d, l : (d, l) = (d', l').(\mathsf{var}[l'].\eta = \mathsf{location} \wedge \mathsf{var}[d'].\eta = \mathsf{direction} \wedge \\ \qquad\qquad\qquad\qquad \mathsf{var}[l'].\pi = \mathsf{var}[d'].\pi) \\ \quad :: \ \mathsf{var}[l].\nu := \mathsf{var}[l].\nu + \mathsf{var}[d].\nu \\ \quad\ \ ||\mathsf{var}[d].\nu := -\mathsf{var}[d].\nu\ \mathit{if}\ l + d = 25 \vee l + d = 0 \\ \rangle \end{array}$$

Using the unique combination of independent programs, their context rules, and universal governance rules, Context UNITY can model a wide variety of context-aware applications. In fact, we acknowledge that Context UNITY may be used to express systems that currently have no practical solutions, e.g., those that provide real-time or robustness guarantees in highly dynamic and mobile settings. At the same time, however, the model provides the power to capture application semantics at a level of detail that is needed to develop a precise engineering approach in order to address such issues. The expressiveness of the model is demonstrated in Section 4 by providing snippets of Context UNITY systems required to model applications taken from the literature on context-awareness.

4 Modeling Real-World Applications

In this section, we investigate several classes of context-aware applications in light of the Context UNITY model introduced in the previous section and show how these applications (or generalizations of them) can be simply modeled using the constructs from Section 3. Our examples follow the evolution of context-aware programming from simple environmental interactions between only two parties, through interactions requiring consideration of security properties to more advanced systems that require context-aware coordination among groups of computational entities.

4.1 Simple Context Interactions

Some of the earliest work in context-aware computing focused on applications using relatively simple context definitions. Such applications often separated concerns related to providing context from those related to using context by introducing *kiosks*, or entities that provide context information to *visitors*, who use the context information to adapt their behavior. For example, in workplace applications like Active Badge [12] and PARCTab [13], users' devices collect location context from sensors fixed in the building to provide location-sensitive services. Guide applications like Cyberguide [14] and GUIDE [15] equip tourists with mobile computing devices and context-aware tour guide software. The software presents location-relevant information to the user by connecting to nearby kiosks and downloading local maps, exhibit information, etc. Such a scenario is depicted in Fig. 6, where a visitor in a museum interacts with kiosks that provide information about the museum's artifacts.

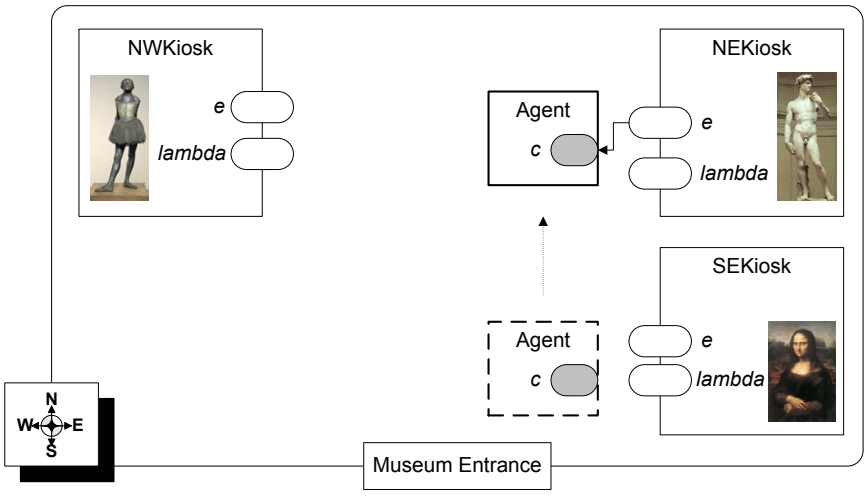

Fig. 6. A simple guide system in Context UNITY

In a Context UNITY model of such an application, agents on the kiosks offer context information to other agents through their *exposed variables*. Visitors' *context variables* determine how relevant exposed variables impact the user's view of the world. For example, a kiosk in the southeast corner of the museum gives information about a painting through its exposed variable e named painting with a textual description of the painting as the variable's value. The kiosks in the northeast and northwest corners of the museum provide information about two different sculptures by naming their exposed variables sculpture and assigning the variable a short textual description. As a visitor moves around the museum with his handheld device, his context variable c, defined to contain only co-located sculptural exhibits, changes in response to the available context. In the figure, the initial position of the visitor agent is depicted by the dashed box labeled "Agent." In the visitor's initial position, there is no sculpture, so the agent's context variable c is not updated. As the visitor moves along the path shown with the dotted arrow, c is updated. When the visitor reaches the northeast corner of the museum, c reflects information about the sculpture at that location. For brevity, we show only the Context UNITY definition of the c context variable. Given this definition, the application can interact locally with the context variable to retrieve and display information about specific artifacts. The visitor's context rule is:

$$
\begin{array}{ll}
c \textbf{ uses} & e \,!\, \mathsf{sculpture}, l \,!\, \mathsf{location} \textit{ in } p \\
\textbf{given} & l = \lambda \\
\textbf{where} & c \textbf{ becomes } e
\end{array}
$$

Informally, this context rule selects two variables from the same agent, one named sculpture and one named location. The further restriction requires that the value of the location variable be equivalent to the visitor agent's location (i.e., λ). When the restrictions can be met, the visitor's context variable c reflects the exposed variable of a co-located statue; otherwise c is empty, reflecting no available statue.

It is important to note that this particular context definition may result in the use of stale context information. For instance, as the visitor moves away from the sculpture and is no longer co-located, his handheld device may still display information about the sculpture, even as he becomes co-located with a different sculpture in the museum. If it is required that the user be provided information about the co-located sculpture immediately upon arrival, a reactive context rule would be more appropriate. However, even the use of reactive context rules cannot eliminate the risk of stale context. For example, if the above context rule were made reactive, as the user moves away from the sculpture and is no longer co-located the context variable c will continue to contain"stale" context information about the sculpture until a new, co-located sculpture is encountered. In some application scenarios, this kind of interaction may be acceptable. In cases where it is not, the context rule can easily be adjusted to reactively reset the contents context variable c upon some condition, e.g., c should be reset when the user is no longer co-located with the sculpture.

In addition to the style of interactions described, context-aware applications frequently employ more complex interactions. In some instances, kiosks provide context information to a stationary context manager, who communicates directly to visitors to direct and adapt their behavior. For example, Gaia [16] manages *active spaces*. An active space is a physical location (e.g., a conference room) in which the available physical and logical resources can be adapted in response to changes in the environment. A typical scenario may entail a user entering the active space and registering with the context manager. The context manager uses information about the user and the environment to perform context-sensitive interactions, e.g., to turn on a projector and load the user's presentation. Such a system can be represented in Context UNITY much as the above application. In this case, however, the visitor provides context information to the manager (kiosk), which subsequently uses its context variables to perform automatic actions.

4.2 Security-Constrained Context Interactions

More recent context-aware applications have directly incorporated security provisions that handle authentication, authorization, encryption and other operations on behalf of users. In several systems, multi-level security mechanisms are provided through *domains* [16,17]. A domain provides layered security and isolates the available resources according to the level of security offered. Agents authorized to operate within a particular domain have the ability to act upon all of the domain's resources, and a domain may have an authorizing authority that grants and revokes entering and exiting agents' access rights.

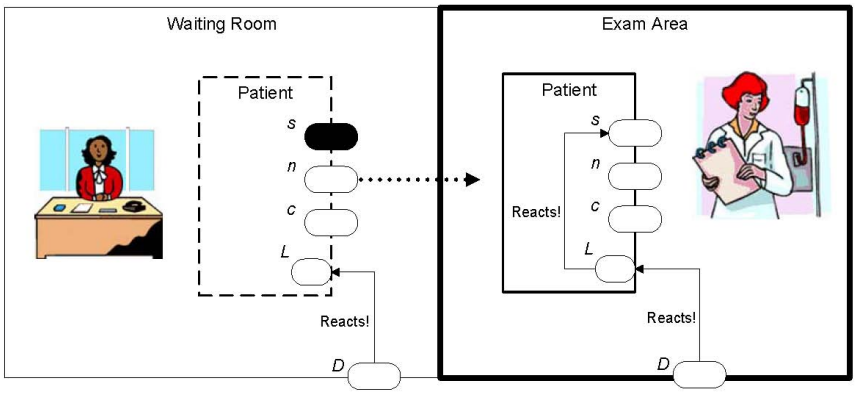

Fig. 7. An example security-constrained application in Context UNITY.

Fig. 7 depicts a doctor's office where two domains coexist: the waiting area and the exam area. In this example, a patient in the office must provide information about herself to receive treatment. Some of the information is

public knowledge to be viewed by the receptionist and perhaps even other patients (e.g., name and contact information). Other information is sensitive and personal and should be displayed only to the doctor (e.g., medical history or symptoms). To facilitate interactions, the doctor's office is divided into the two domains shown that provide differing levels of privacy. The patient's information includes his name (n), contact information (c), and symptoms (s), each stored in an exposed variable. The exposed variable D in each of the domains represents the level of security offered in the domain, while the exposed variable L in the patient's record reflects the security quality of the user's current location. In the context definition and usage described below, the value of the patient's L value determines the access control function used for the patient's symptom information stored in s. (The shaded nature of the s variable in the waiting room in the figure indicates that it is not accessible.)

In our Context UNITY expression of this application, we abstract away the authentication of the security domains and assume that the patient can authenticate a domain that claims to be "high-security" (i.e., a domain with a string value of "high-security" for its exposed variable D). In an implementation, this string would be a password or secret key that would guarantee the patient's secrecy. When the patient's L variable stores the value "high-security," the patient can be confident that he is in a secure area and can therefore share his symptoms. The patient's context variable L is defined as:

$$
\begin{array}{ll}
L \textbf{ uses} & x \,!\, \mathsf{security}, l \,!\, \mathsf{location}\ in\ p \\
\textbf{given} & l = \lambda \\
\textbf{where} & L \textbf{ becomes } x \\
\textbf{reactive} &
\end{array}
$$

The reactive nature of the context definition ensures that the patient's agent is notified immediately following a security domain change. This is especially important as the patient moves from a high-security area to a lower security area to guarantee that the access privileges for the symptoms variable are immediately revoked. The following two statements appear in the patient's **assign** section and use the value of the patient's L context variable to adaptively change the access policy for the exposed variable s:

assign
...
$s.\alpha := F(L)$ **reacts-to** $L =$ "high-security"
$s.\alpha := F(L)$ **reacts-to** $L \neq$ "high-security"
...

where $F(L)$ returns {r} if L has the value "high-security" and {} otherwise.

4.3 Uniform Context Definition

As context-aware applications have evolved, the applications' interactions have moved from the simple two-way sharing as described above to include more complex group interactions that foster complex coordination. Coordination models [6,18–20] have emerged that provide a high degree of decoupling, an important design concerned touched upon in Section 2. A common characteristic of these systems is that agents that enter a sharing relationship must all have the same definition of context, i.e., the context rules are uniform and universally applied. This is representative of, for example, applications that support collaborative work environments where a team of distributed agents collaborate to perform a task, e.g., write a research paper.

Of the coordination models cited above, LIME [6,21] is the most general as it incorporates both physical mobility of hosts and logical mobility of agents. LIME uses tuple spaces permanently attached to mobile agents which logically merge together to form a single shared tuple space among connected agents. Agents may be associated with several local tuple spaces, distinguished by name. An agent interacts with other agents by employing content-based retrieval (`rd(pattern)` and `in(pattern)`), and by generating tuples (`out(tuple)`). These traditional operations are augmented with reactions that extend their effects to include arbitrary atomic state transitions. In LIME, an agent's relevant context is determined by the logically merged contents of identically named tuple spaces held by mutually reachable agents.

To use Context UNITY to capture the essential features of context-aware systems having the characteristics described above, we endow each agent with an exposed variable named localTS that offers its local tuple space for sharing and a second exposed variable named sharedTS that provides the agent access to all the tuples in the current context. These variables are of type tupleSpace, which is a simple set of tuples. The value of the sharedTS variable should be the union of tuples contained in exposed local tuple space variables belonging to connected agents. Because the shared tuple space definition is uniform across all agents, we can capture it in the **Governance** section, which highlights the fact that connected agents share a symmetric context. In addition, it is more economical for a programmer to write a single context definition since it applies to the entire system. The resulting context rule included in the **Governance** section is as follows:

use ts_c ! sharedTS *in* a; ts_l ! localTS *in* b
given $connected(a, b)$
where $ts_c - (ts_c \uparrow b) \cup ts_l$ **impacts** ts_c
reactive

The result of this context rule is a tuple space shared among connected agents. The notation $ts_c \uparrow b$ indicates a projection over the set ts_c, i.e., the tuples in

ts_c owned by the agent b. It is possible to obtain such a projection since we assume that each generated tuple has a field which identifies the owner of the tuple using the generating agent's unique id. The update expression therefore has the effect of removing from ts_c all of the tuples in it that belong to b and adding to the set all of the tuples from b's local tuple space (ts_l). This is required to ensure that, when changes occur to the data stored in the tuples, the stale copies of the data are removed from a's local copies and replaced with the updated values. The context rule's reactive nature ensures that this update happens as soon as any changes occur.

4.4 Tailored Context Definitions

The applications addressed by the above coordination paradigm all view and interact with the same context. Other applications, however, require more individualized interactions, where they gather context information from a distributed network and then use this context information for their own personalized behavior [1,3]. However, as the scale of computing environments grows, the amount of context information available to influence an agent's behavior becomes large and unmanageable. To avoid presenting an agent with an overwhelming amount of context, many of these applications limit the amount of context information that an agent "sees" based on properties of its environment and desired interactions. For example, EgoSpaces [3] is founded on the *view concept*, which restricts an agent's context according to a personalized specification. A view consists of constraints on network properties, the other agents from which context is obtained, and the hosts on which such agents reside. These constraints filter out unwanted items in the operational context, and the system ultimately presents the application with a context tailored to its particular needs. As a specific example, an agent on an automobile may monitor traffic information for a region in front of it that defines the driver's potential route home. This "context" information should be pulled to the agent which can use it to adapt its behavior (e.g., reroute the driver).

Such applications consist of agents that serve as both providers and users of context. The agents employ a context management strategy tailored to their individual needs. When behaving as a context provider, a Context UNITY agent generates pieces of context information and places them in an exposed variable that serves as a data repository (e.g., a tuple space, as above) consisting of data that the agent wishes to contribute as context. An agent provides information about itself and properties about the host on which it resides in exposed variables named "agent profile" and "host profile," respectively. These variables allow other agents to filter the context according to the host and agent constraints in their view definitions. From the perspective of a context user, Context UNITY models an agent's view using a rule for a context variable v named "view." The value of v is defined to be the set of all tuples

present in exposed tuple space variables of other reachable agents for which the exposed agent profile properties, exposed host profile properties, and exposed network properties of hosts match the reference agent's constraints. An example context rule that establishes a view v for an agent with id i can be defined as follows:

$$
\begin{array}{ll}
v \textbf{ uses} & lts \,!\, \mathsf{tuple\_space}, a \,!\, \mathsf{agent\_profile}, h \,!\, \mathsf{host\_profile} \; in \; i \\
\textbf{given} & reachable(i) \wedge eligibleAgent(a) \wedge eligibleHost(h) \\
\textbf{where} & v \textbf{ becomes } v \; - \; (v \uparrow i) \; \cup \; lts \\
\textbf{reactive} &
\end{array}
$$

The function *reachable* encapsulates the network constraints that establish whether an agent should or should not be considered based on network topology data. The reactive nature of this definition rule ensures that the view definition is updated simultaneously for *all* agents i that completely satisfy the constraints in the context rule and that, as soon as any properties affecting the definition of the view change, the view's contents are updated. In these applications, the reference agent may also make changes to data items in the view; additional context resolution rules handle the propagation of these changes back to the other context agents.

This discussion has demonstrated that increasingly complex context-aware applications can be simply and elegantly modeled using Context UNITY. The power of the Context UNITY model is two-fold: not only can it be used to represent existing context-aware systems but it can aid in the careful design of future applications by enforcing the design principles embodied in the requirements outlined in Section 2.

5 Formal Verification

Context UNITY has an associated proof logic largely inherited from Mobile UNITY [4], which in turn builds on the original UNITY proof logic [10]. Program properties are expressed using a small set of predicate relations whose validity can be derived directly from the program text, indirectly through translation of program text fragments into Mobile UNITY constructs, or from other properties through the application of inference rules. In this section we provide a review of the Mobile UNITY proof logic and examine strategies for the verification of Context UNITY programs.

In Mobile UNITY (as in UNITY), program verification starts with the semantic properties of the individual program statements. While UNITY contains only standard conditional multiple assignment statements, Mobile UNITY includes reactive statements and transactions; as discussed later, Context UNITY also adds non-deterministic assignment statements. The basic execution model of Mobile UNITY is one in which normal statements are selected non-deterministically and in a weakly fair manner, and, after the execution of each normal statement, all reactive statements are executed as a single separate program until its fixed-point is reached. Transactions force sequential selection of the normal statements that they contain, but otherwise the execution model remains unchanged. Since the semantics of Context UNITY have been defined by reduction to Mobile UNITY statements (normal and reactive) we provide next a brief review of the Mobile UNITY proof logic, which will need to be employed in the verification of Context UNITY programs.

Regardless of the model under consideration, proving individual statements correct in state transition systems starts with the use of the *Hoare triple* [22]. 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, the presence of reactive statements requires us to use:

$$\{p\}s^*\{q\} \textbf{ where } s \in \mathcal{N}$$

where $\mathcal{N}$ denotes the normal statements of P, while s^* denotes a normal statement s modified to reflect the extended behavior resulting from the execution of the reactive statements in the reactive program $\mathcal{R}$ 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:

$$\frac{\{p\}s\{H\}, H \mapsto (FP(\mathcal{R}) \wedge q) \text{ in } \mathcal{R}}{\{p\}s^*\{q\}}$$

The first component of the hypothesis states that, when executed in a state satisfying p, the statement s establishes the intermediate postcondition H. This postcondition serves as a precondition of the reactive program $\mathcal{R}$, that, when executed to fixed-point, establishes the final postcondition q. The "in $\mathcal{R}$" must be added because the proof of termination is to be carried out from the text of the reactive statements, ignoring other statements in the system. This can be accomplished with a variety of standard UNITY techniques. The

predicate H must lead to a fixed-point *and* establish q in the reactive program $\mathcal{R}$. This obligation (i.e., $H \mapsto (FP(\mathcal{R}) \wedge q)$ in $\mathcal{R}$) can be proven with standard techniques because $\mathcal{R}$ is treated as a standard UNITY program.

For transactions of the form $\langle s_1; s_2; \ldots; s_n \rangle$ we first apply 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. We then can apply the first, more complex inference rule (presented earlier in this section) 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.

To prove more sophisticated properties, UNITY-based models use predicate relations. Basic safety is expressed using the **unless** relation. For two state predicates p and q, the expression p **unless** q means that, for any state satisfying p and not q, the next state in the execution must satisfy either p or q. There is no requirement for the program to reach a state that satisfies q, i.e., p may hold forever. Progress is expressed using the **ensures** relation. 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 in the program that guarantees the establishment of q if executed in a state satisfying p and not q. Note that the **ensures** relation is not itself a pure liveness property but a conjunction of a safety and a liveness property; the safety part of the **ensures** relation can be expressed as an **unless** property. In UNITY, these predicate relations are defined by:

$$p \textbf{ unless } q \equiv \langle \forall s : s \textbf{ in } P :: \{p \wedge \neg q\}s\{p \vee q\}\rangle$$

$$p \textbf{ ensures } q \equiv (p \textbf{ unless } q) \wedge \langle \exists s : s \textbf{ in } P :: \{p \wedge \neg q\}s\{q\}\rangle$$

where s is a statement in the program P. Mobile UNITY uses the same definitions since all distinctions are captured in the verification of the Hoare triple. Additional relations may be derived to express other safety (e.g., **invariant** and **stable**) and liveness (e.g., **leads-to**) properties.

5.2 Context UNITY Proof Mechanics

The verification of Context UNITY programs relies by and large on the Mobile UNITY proof logic. However, Context UNITY introduces non-deterministic

assignment which is not handled by the Mobile UNITY proof logic as defined so far. Fortunately, the proof obligation for non-deterministic assignments differs only slightly from that of the standard assignment statements. Given the property $\{p\}s\{r\}$ in UNITY, if the statement s is a non-deterministic assignment statement of the form $x := x'.Q(x')$, then the inference rule describing the associated proof obligation for the statement s has the form:

$$\frac{\{p \wedge \exists x' :: Q(x')\}s\{\forall x' : Q(x') :: r\}}{\{p\}s\{r\}}$$

At this point all the tools needed to verify Context UNITY programs have been presented, even though we did not explicitly describe a Context UNITY proof logic. Due to the manner in which we formalized Context UNITY's semantics, each Context UNITY statement is defined operationally by its translation into Mobile UNITY (with the addition of the above rule for non-deterministic assignment statement). The resulting strategy is to translate Context UNITY context rules from both the local program **context** sections and the **Governance** section to standard Mobile UNITY notation (i.e., to the appropriate normal or reactive statements) before applying the proof logic outlined for Mobile UNITY. Once translated as described in the previous section, verification of the system can be accomplished directly by applying the rules outlined above.

The approach makes sense because Context UNITY is a specialization of Mobile UNITY. It is clear that mechanical verification techniques, if developed, would not be affected negatively because our mapping to Mobile UNITY is very mechanistic. As a matter of fact, the straightforward translation process has only a minimal impact even on pencil and paper proofs. This is because each context specification statement is mapped either to a multiple assignment statement or to a reaction, but never to a complex set of program statements or a program fragment consisting of both normal and reactive statements.

To illustrate the verification process we return to the earlier context specification for the automatic maintenance of the acquaintance list Q. We might want to prove, for instance, that an agent a is in the acquaintance list Q of b if and only if a and b are within communication range. This can be captured by the following invariant:

$$\textbf{inv. } a \in b.Q \Leftrightarrow (a \neq b \wedge |a.\lambda - b.\lambda| \leq range)$$

If we assume that initially no two agents are in range and all acquaintance lists are empty, we need to prove that the invariant is preserved throughout the execution of the program. Assuming that none of the agents have the direct ability to modify the context variable Q, the only way to violate the invariant is by affecting the agent position which we assume is under the sole control of

the individual agents.

The proof obligation reduces to showing that the reactive statements that update Q reach fixed-point and re-establish the invariant after the execution of any statement in the program. Statements that do not affect location have no impact on the invariant and can be ignored. Statements that do change an agent's location take the system into a state in which the invariant no longer holds. This, in turn, leads to the obligation to show that, started in such a state, the reactive program leads to re-establishing the invariant. We can show this to be true by induction on the number of inconsistent acquaintance lists. To show that this variant function decreases, we consider any acquaintance list that is incorrect and show that it is corrected as soon as the right statement executes. More precisely, we consider two cases, when the agent a needs to be added and when it needs to be removed. In each case we can use an **ensures** property to prove that the list is updated in one step by the appropriate context rule. The two separate cases can be formally combined into a leads-to property, which guarantees that the arbitrarily selected acquaintance list eventually is up to date. This in turn, establishes the base case for the induction and completes the proof. It is only at the level of verifying the two **ensures** obligations that the translation into the Mobile UNITY reactive statement is invoked. Even in a simple example such as this one it is evident that the proof of the Hoare triple is a small part of the overall verification effort and the only part which is affected by the translation rules. For this reason we view our reductionist approach as offering a viable and practical strategy for the formal verification of context-aware programs.

6 Conclusions

The formulation of Context UNITY is designed to help us gain a better understanding of the essential features of the context-aware computing paradigm. A key feature of the model is the delicate balance it achieves between placing no intrinsic limits on what the context can be while empowering the individual agent with the ability to precisely control the context definition. Linguistically the distinction is captured by the notions of operational environment and context, expansive with respect to potential and specific with respect to relevance. In the model, the two concepts have direct representations in terms of exposed and context variables. The other fundamental characteristic of the model is rooted in the systematic application of software engineering methodological principles to the specifics of context-aware computing. The functionality of the application code is separated from the definition of context. This decoupling is fundamental in a setting where adaptability is important—a program design cannot anticipate the details of the various operational environments the agent will encounter throughout its lifetime. The model enables this de-

coupling through the introduction of context rules that exploit existential quantification and non-determinism in order to accommodate the unknown and unexpected. Context UNITY explicitly captures the essential characteristics of context-awareness, as we experienced them in our work and observed them in that of others. Moreover, the defining traits of many existing models appear to have simple and straightforward representations in Context UNITY, at least at an abstract level.

ACKNOWLEDGMENTS

This research was supported in part by the Office of Naval Research under ONR MURI research contract 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 sponsoring agencies.

References

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

[2] J. Hong, J. Landay, An infrastructure approach to context-aware computing, Human Computer Interaction 16 (2001) 287–303.

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

[4] G.-C. Roman, P. J. McCann, A notation and logic for mobile computing, Formal Methods in System Design 20 (1) (2002) 47–68.

[5] P. J. McCann, G.-C. Roman, Compositional programming abstractions for mobile computing, IEEE Transactions on Software Engineering 24 (2) (1998) 97–110.

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

[7] C.-L. Fok, G.-C. Roman, G. Hackmann, A lightweight coordination middleware for mobile computing, in: Proceedings of the 6^{th} International Conference on Coordination Models and Languages, 2004, pp. 135–151.

[8] B. Schilit, N. Adams, R. Want, Context-aware computing applications, in: IEEE Workshop on Mobile Computing Systems and Applications, 1994.

[9] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley and Sons, 2000.

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

[11] R. J. R. Back, K. Sere, Stepwise refinement of parallel algorithms, Science of Computer Programming 13 (2–3) (1990) 133–180.

[12] A. Harter, A. Hopper, A distributed location system for the active office, IEEE Networks 8 (1) (1994) 62–70.

[13] R. Want, et al., An overview of the PARCTab ubiquitous computing environment, IEEE Personal Communications 2 (6) (1995) 28–33.

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

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

[16] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, Gaia: A middleware infrastructure to enable active spaces, IEEE Pervasive Computing (2002) 74–83.

[17] J. Wickramasuriya, N. Venkatasubramanian, A middleware approach to access control for mobile concurrent objects, in: Proceedings of the International Symposium on Distributed Objects and Applications, 2002.

[18] G. Cabri, L. Leonardi, F. Zambonelli, MARS: A programmable coordination architecture for mobile agents, Internet Computing 4 (4) (2000) 26–35.

[19] IBM, T Spaces, http://www.almaden.ibm.com/cs/TSpaces/ (2001).

[20] Sun, Javaspaces, http://www.sun.com/jini/specs/jini1.1html/js-title.html (2001).

[21] A. Murphy, G.-P. Picco, Using coordination middleware for location-aware computing: A LIME case study, in: Proceedings of the 6^{th} International Conference on Coordination Models and Languages, 2004, pp. 263–278.

[22] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580, 583.