A Formal Treatment of Context-Awareness

Gruia-Catalin Roman, Christine Julien, and Jamie Payton

Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
{roman,julien,payton}@wustl.edu

Abstract. Context-aware computing refers to a computing paradigm in which the behavior of individual components is determined by the circumstances in which they find themselves to an extent that greatly exceeds the typical system/environment interaction pattern common to most modern computing. The environment has an exceedingly powerful impact on a particular application component either because the latter needs to adapt in response to changing external conditions or because it relies on resources whose availability is subject to continuous change. In this paper we seek to develop a systematic understanding of the quintessential nature of context-aware computing by constructing a formal model and notation for expressing context-aware computations. We start with the basic premise that, in its most extreme form, context should be made manifest in a manner that is highly local in appearance and decoupled in fact. Furthermore, we assume a notion of context that is relative to the needs of each individual component, and we expect context-awareness to be maintained in a totally transparent manner with minimal programming effort. We construct the model from first principles, seek to root our decisions in these formative assumptions, and make every effort to preserve minimality of concepts and elegance of notation.

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, often packaged as service activities, to blend into the application domain in a manner that makes computers gradually less visible and more agile. These are not new concerns for software engineers, but the attention being paid to context-awareness enhances a system's ability to become ever more responsive to the needs of the end-user or application domain. With the growing interest in adaptive systems and with 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

M. Wermelinger and T. Margaria-Steffen (Eds.): FASE 2004, LNCS 2984, pp. 12–36, 2004.

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 being addressed in this paper. We want to understand what context-aware computing is, and we do so by proposing a simple abstract conceptual model of context-awareness and by attempting to formalize it. Along the way, we examine the rationale behind our decisions, thus providing both a systematic justification for the model and the means for possible future refinements.

The term context-awareness immediately suggests a relation between some 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 its *operational environment*. We differentiate the notion of operational environment from that of 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 are actually relevant to the reference agent's behavior. In formulating a model of context-awareness we need to focus our attention on how this relevant subset is determined. To date, most of the research on context-awareness considers a restricted context, i.e., context is what can be sensed locally, e.g., location, temperature, connectivity, etc. However, distant entities can affect the agent's behavior, and the size of the zone of influence depends upon the needs of the specific application. The scope and quality of the information gathered about the operational environment affect the cost associated with maintaining and accessing context information. These suggest that a general model of context-awareness must allow an agent to work with a context that may extend beyond its immediate locality (i.e., support node or host) while also enabling it to control costs by precisely specifying what aspects of the operational environment are relevant to its individual needs as they change over time.

A model of context-awareness must be *expansive*, i.e., it must recognize the fact that distant entities in the operational environment can affect an agent's behavior [4]. This requirement states that one should not place a priori limits on the scope of the context being associated with a particular agent. While specific instantiations of the model may impose restrictions due to pragmatic considerations having to do with the cost of context maintenance or the nature of the physical devices, application needs are likely to evolve with time. As a consequence, fundamental assumptions about the model could be invalidated. To balance out the expansive nature of the model and to accommodate the need for agents to exercise control over the cost of context maintenance, we also require the model to support a notion of context that exhibits a high degree of *specificity*. In other words, it must be possible for context definitions to be tailored to the needs of each individual agent. Furthermore, as agents adapt, evolve, and alter their needs, context definitions also should be amenable to modification in direct response to such developments.

Expansiveness and specificity are central to achieving generality. They are necessary but not sufficient features of the context-aware computing paradigm that consider the way in which the operational environment relates to an agent's notion of context, i.e., the distinction between potentiality and relevance. They fail to consider the manner in which the agent forms and manipulates its own notion of context. The only way an agent can exercise control over its context is to have an *explicit* notion of context. This gives the agent the power to define its own context and to change the definition as it sees fit. It also formalizes the range of possible interactions between the agent and its operational environment. Consequently, context definition has to be an identifiable element of the proposed model and must capture the essential features of the agent/context interaction pattern. Separation of concerns suggests that an agent's context specification be *separable* from its behavior specification. The agent behavior may result in changes to the definition of context, but the latter should be readily understood without one needing to examine the details of the agent behavior. This requirement rules out the option of having to derive context from the actions of the agent. This distinction is important because many systems interact with and learn about their operational environment without actually employing the context-aware paradigm. Finally, context maintenance must be *transparent*. This implies that the definition of context must be sufficiently abstract to free the agent of 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.

To illustrate our perspective on context-aware computing, let us examine the case of an application in which contextual information plays an important role, but the criteria for being classified as employing the context-aware paradigm are not met. Consider an agent that receives and sends messages, learns about what other agents are present in its environment through the messages it receives, and uses this information to send other messages. Indeed the context plays an important role in what the agent does, and the agent makes decisions based upon the knowledge it gains about its operational environment. More precisely, the agent implicitly builds an acquaintance list of all other agents in the region through a gossiping protocol that distributes this information, and it updates its knowledge by using message delivery failures that indicate agent termination or departure. However, we do not view this as an instance of the context-aware paradigm; the way the agent deals with the environment may be considered extensible but it is not specific, explicit, separable, or transparent. It is particularly instructive to note what is required to transform this message-passing application into one that qualifies as an instance of the context-aware paradigm.

Specificity could be achieved by having each agent exercise some judgment regarding which other agents should or should not be included in its acquaintance list. Explicitness could be made manifest by having the agent include a distinct representation of the acquaintance list in its code – a concrete representation of its current context. Separability could be put in place by having the code that updates the acquaintance list automatically extract agent infor-

mation from arriving messages for placement in the acquaintance list, e.g., by employing the interceptor pattern. Transparency requires the design to go one step further by having the agent delegate the updating of the acquaintance list to an underlying support infrastructure; this, in turn, demands that the definition of the context be made explicit to the support infrastructure either at compile or at run time – an evaluation procedure to establish which other agents qualify as acquaintances and which do not suffices. The result is an application that exhibits the same behavior but a different design style; the agent's context is made manifest through an interface offering access to a data structure that appears to be local, is automatically updated, and is defined by the agent which provides the admission policy controlling which agents in the region are included or excluded from the list. This is not the only way to employ the context-aware paradigm but clearly demonstrates the need for the designer to adopt a different mind-set.

The principal objective of this paper is to explore the development of an abstract formal model for context-aware computing; no such model is available in the literature to date, other than a preliminary version of this work [5]. Because our ultimate goal is to achieve a better understanding of the essence of the context-aware computing paradigm, we seek to achieve minimality of concepts and elegance of notation while remaining faithful to the formative assumptions that define our perspective on context-awareness. The resulting model is called Context UNITY and has its roots in our earlier formal work on Mobile UNITY [6,7] and in our experience with developing context-aware middleware for mobility. 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. In this way, the agent code is local in appearance and totally decoupled from that of all the other agents in the system. The context definition is an explicit part of the program type description, is specific to the needs of each agent as it changes over time, and is separate from the behavior exhibited by the agent. The design of the Context UNITY notation is augmented with an assertional style proof logic that facilitates formal reasoning about context-aware programs.

The remainder of this paper is organized as follows. The next section presents our formalization of context-awareness in detail. In Section 3, we outline the proof logic associated with the model. Section 4 shows how the model can express key features of several existing context-aware systems. Conclusions appear in Section 5.

2 Formalizing Context-Awareness

Context UNITY represents an application as a community of interacting agents. Each agent's behavior is described by a program that serves as the agent's proto-

type. To distinguish agents from each other, each has a unique identifier. Because we aim to model context-aware systems, an agent must access its environment, which, in Context UNITY, is defined by the values of the variables other agents in the system are willing to expose. As described in the previous section agents require context definitions tailored to their individualized needs. In Context UNITY, agents interact with a portion of the operational environment defined through a unique set of variables designed to handle the agent's context needs.

A central aspect of Context UNITY is its representation of program state. Three categories of variables appear in programs; they are distinct in the manner in which they relate to context maintenance and access. First, a program's *internal variables* hold private data that the agent uses but does not share with the other agents in the system. They do not affect the operational environment of any other agent. *Exposed variables* store the agent's public data; the values of these exposed variables can contribute to the context of other agents. The third category of variables, *context variables*, represent the context in which the particular agent operates. These variables can both gather information from the exposed variables of other agents and push data out to the exposed variables of other agents. These actions are governed by context rules specified by each agent and subject to access control restrictions associated with the exposed variables.

In the remainder of this section, we first detail the structure of a Context UNITY system. We then show how programs use context variables to define a context tailored to the needs of each particular agent and the mechanics that allow an agent to explicitly affect its operational environment. Throughout we provide examples using the model to reinforce each concept.

2.1 Foundational Concepts

Context UNITY represents an application as a *system specification* that includes a set of *programs* representing the application's component types. Fig. 1 shows the Context UNITY representation of a **System**. The first portion of this definition lists programs that specify the behavior of the application's individual agents. Separating the programs in this manner encapsulates the behavior of different application components and their differing context needs. The **Components** section of the system declares the instances of programs, or agents, that are present in the application. These declarations are given by referring to program names, program arguments, and a function (*new_id*) that generates a unique id for each agent declared. Multiple instantiations of the same program type are possible; each resulting agent has a different identifier. The final portion of a system definition, the **Governance** section, captures interactions that are uniform across the system. Specifically, the rules present in this section describe statements that can impact exposed variables in all programs throughout the system. The details of an entire system specification will be made clearer through examples later in this section. First we describe in detail the contents of an individual Context UNITY program.

Each Context UNITY program lists the variables defining its individual state. The declaration of each variable makes its category evident (internal, exposed,

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. 1. A Context UNITY Specification

or context). A program's **initially** section defines what values the variables are allowed to have at the start of the program.

The **assign** section defines how variables are updated. These assignment statements can include references to any of the three types of variables. Like UNITY and its descendants, Context UNITY's execution model selects statements for execution in a weakly-fair manner – in an infinite execution, each assignment statement is selected for execution infinitely often. In the assignment section, a program can use simple assignment statements, transactions, or reactions. A *transaction* is a sequence of simple assignment statements which must be scheduled in the specified order with no other (non-reactive) statements interleaved. They capture a form of sequential execution whose net effect is a large-grained atomic state change. In the **assign** section of a program, a transaction uses the notation: $\langle s_1; s_2; \ldots; s_n \rangle$. A *reaction* allows a program to respond to changes in the state of the system. A reaction is triggered by an enabling condition Q and has the form s **reacts-to** Q. As in Mobile UNITY, Context UNITY modifies the execution model of traditional UNITY to accommodate reactions. 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, forming what we call a *reactive program*, executes until it reaches *fixed-point*. 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.

In Context UNITY, an agent's behavior is defined exclusively through its interaction with variables. To handle context interactions, Context UNITY introduces context variables and a special **context** section that provides the rules that manage an agent's interaction with its desired context. Specifically, the

context section contains definitions that sense information from the operational environment and store it in the agent's context variables. The rules can also allow the agent to affect the behavior of other agents in the system by impacting their exposed variables. The use of this special **context** section explicitly separates the management of an agent's context from its internal behavior.

Two prototypical uses of the **context** section lie at the extremes of sensing and affecting context. First, a program's context definition may only read the exposed variables of other programs but not affect the variables' values. When used in such a way, we refer to the context variables as *sentient variables* because they only gather information from the environment to build the agent's context. In the other extreme case, a program can use its context variables to disperse information to components of the environment. From the perspective of the reference agent, this affects the context for other agents, and we refer to context variables used in this manner as *impact variables*. While these two extremes capture the behavior of context-aware systems in the most common cases, the generality of Context UNITY's context specification mechanism allows it to model a variety of systems that fall between these two extremes. The examples discussed in Section 4 demonstrate this in more detail.

The acquaintance list application introduced in the previous section provides a list of nearby coordination participants. Several context-aware systems in the literature, e.g., Limone [8], use this data structure as a basis for more sophisti-

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

Fig. 2. A Context-Aware System for Acquaintance Maintenance

cated coordination mechanisms. The acquaintance list is defined by dynamically changing needs of a reference agent. Fig. 2 shows a Context UNITY specification for an application that relies on the usage of an acquaintance list. This system consists of three agents of two differing types. Each agent stores its unique agent id in an exposed variable named agent_id that is available to other programs. Because we are modeling systems that entail agent mobility, each agent also has a variable named location that stores its location. The movement of the agent is outside this example; it could occur through local assignment statements to the location variable (in the **assign** section of the individual program) or even by a global controller (via the **Governance** section of the system). Both *id* and λ are local handles for built-in variables whose names are agent_id and location, respectively. We discuss these built-in variables in more detail later in this section. Each program type has individualized behavior defined via the **assign** section that may use additional context variables or definitions. In this example, we are most concerned with the maintenance of the acquaintance list. Each agent declares a context variable Q of type set that will store the contents of the acquaintance list. Different program types (in this case, *Agent1* and *Agent2*) employ different eligibility qualification criteria for the members of the acquaintance list, exemplified in the **context** section of each program type. This example shows a high-level definition of a context variable. In the acquaintance management specification, each program's **context** section contains a rule that describes how the context variable Q is updated. Later in this section we will show exactly what this rule entails. First however, we expound on the structure of Context UNITY exposed variables.

Exposed Variables Revisited. In UNITY and many of its descendants, variables are simply references to values. In Context UNITY, both internal and context variables adhere to this standard. However, references to exposed variable appearing in the program text are references to more complex structures needed to support context-sensitive access within an unknown operational environment. These handle names have no meaning outside the scope of the program. A complete semantic representation of exposed variables is depicted in Fig. 3. Each exposed variable has a unique id ι – uniqueness could be ensured by making each variable unique within an agent and combining this with the unique agent id. This unique id is used in the context interaction rules to provide a handle to the specific variable. The agent owning the exposed variable, π or type agent_id, also appears in the semantic structure and allows an exposed variable to be selected based on its owner. An exposed variable's name, η, provides information about the kind of data the variable contains; the name of an exposed variable can be changed by the program's assignment statements. The type τ reflects

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. 3. Variable Components

the exposed variable's data type and is fixed. An exposed variable's value, ν, refers to the data value held by the variable. Programs refer to the value when assigning to the variable or when accessing the value the variable stores. The value of an exposed variable can be assigned in the **assign** section or can be determined by a different program's impact on its environment. The program can control the extent to which its exposed variables can be modified by others using the access control policy described below.

Modeling Access Control. The final component of an exposed variable in Context UNITY, α, stores the variable's access control policy. Because many context-aware systems and applications use some form of access restriction, Context UNITY provides a generalized mechanism for modeling access control. An access policy determines access based on properties of the particular agent accessing the variable. The access control policy determines both the readability and writability of the particular variable on a per-agent basis. The function α takes as arguments credentials provided by the reference agent and returns the set of allowable operations on this variable, e.g., {r, w} signifies permission to both read and write the particular exposed variable. Because Context UNITY associates an access control policy with each variable, it models the finest-grained access restrictions possible in a context-aware application. This model can be tailored to the needs of current context-aware systems, including those that utilize a trusted third party for authentication.

Built-in Variables. To ease representation of context-aware interactions, Context UNITY programs contain four built-in exposed variables. In Context UNITY, these 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 variables has the name "location" and facilitates modeling mobile context-aware applications by storing the location of the program owning the variable. This variable is exposed and available to other programs to use. An example use of this variable was shown in the system in Fig. 2. 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 of Context UNITY's built-in variables is also exposed and has the name "type", and its value is the program's name (e.g., "*Agent1*" or "*Agent2*" in the example system). As we will see, the use of this variable can help context variables select programs based on their general function. The third of the built-in variables has the name "agent_id" and holds the unique identifier assigned to the agent when the agent is instantiated in the **Components** section. The final built-in variable is internal and has the local handle "*credentials*". It is used in Context UNITY interactions to support access control restrictions. Specifically, the variable stores a profile of attributes of the program that are provided to the access control policies of the exposed variables of other programs. These credentials are available to access control policies when determining whether or not this program has access to a particular exposed variable.

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.

As indicated in the beginning of this section, due to the unpredictable nature of the dynamic environments in which context-aware agents operate, their context definitions require a mechanism to handle their lack of a priori knowledge about the operational environment. In Context UNITY, we introduce *non-deterministic assignment statements* to the definition of context. Specifically, the non-deterministic assignment statement $x := x'.Q$ assigns to x a value x' non-deterministically selected from all values satisfying the condition Q [9]. A program's context rules define how an agent can access and interact with the exposed variables of other agents. It can select which other agents' variables affect its behavior by employing non-deterministic assignments and existential quantification. The flexibility of this selection mechanism allows agents that contribute to the context to be selected based on attributes defined in their exposed variables. For example, in a mobile context-aware application, an agent can use the built-in Context UNITY location variable to store its current physical location. Whenever the component moves, the agent updates the location variable using an assignment statement in the local **assign** section. Another agent can use relative distance to identify which other agents are to contribute to its context. We refer to this selection of agents based on their properties as *context-sensitive program selection*.

Context UNITY wraps the use of non-deterministic assignment in a specialized notation for handing context-aware interactions. 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, which we refer to as 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 the exposed variables that relate to the context variable c. The scope of these dummy variables is limited to the particular context rule that declares them. The expression can refer to any exposed variables in the system, but referring to other

programs' exposed variables explicitly requires the program to have advance knowledge about the other components it will encounter over time, which programs rarely have. Typically, context-aware applications rely on opportunistic interactions that cannot be predetermined. To capture this style of interaction in Context UNITY, the exposed variables that contribute to the context rule are selected in a context-sensitive manner using the restrictions provided in the rule's definition. As one example, because a wireless context-aware application contains many agents that may or may not be connected, the restrictions used in a context rule for a particular application must account for the connectivity restrictions imposed by the operational environment.

Given the set of exposed variables selected in accordance with the restrictions, the context rule can define an expression, *expr*, over the 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.

The execution of each context rule can optionally be declared **reactive**, which dictates the degree of consistency with which the context rule reflects the environment. 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.

Within a context rule, if no explicit restrictions are placed on the referenced exposed variables, two restrictions are automatically assumed. The first requires that the variable referenced be an exposed variable in its owner program since only exposed variables are accessible from other programs. The second implicit restriction requires that the program whose context uses a particular exposed variable must satisfy the variable's access control policy. Consider the following simple context rule that pulls the value out of some exposed variable, places the value in the context variable c, and deletes the value from the exposed variable used. The statement is a reactive statement that is triggered when a is larger than the value of some local variable x:

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

This reactive construct makes the rule part of the system's set of reactive statements. This context rule corresponds to the following formal definition, which includes the two implicit restrictions on the exposed variable a as discussed above:

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

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 selecting the variable a from the table, the statement above really selects its variable id, which serves as a reference to a specific entry in the table var. In this statement, for instance, the exposed variable a is non-deterministically selected from all exposed variables whose access control policies allow this agent access to read and write the exposed variable that the dummy variable a refers to. The latter is determined by applying the variable's access control policy to this agent's credentials. The set returned by this application can contain any combination of r and w, where the presence of the former indicates permission to read the variable, and the presence of the latter indicates permission to write the variable. 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.

The power of the context-sensitive selection of exposed variables becomes apparent only when the restrictions within the context rules are used. Within the restrictions, the context rule can select exposed variables to be used based on the exposed variables' names, types, values, owning agent, or even based on properties of other variables belonging to the same or different agents. To simplify the specification of these restrictions, we introduce a few new pieces of notation. Referring to the system-wide table of variables (i.e., var) is cumbersome and confusing because the table is both virtual and distributed. For this reason, context rules refer directly to indexes in the table instead. Specifically, in this notation, 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 $a.\eta$ to denote $\mathsf{var}[a].\eta$. Because a common operation in context-sensitive selection relies on selecting variables from the same program, we also introduce a shorthand for accessing a variable by the combination of name and program. To do this, when declaring dummy variables, a context rule can restrict both the names and relative owners of the variables. For example, the notation: $x\,!\,\mathsf{name}_1, y\,!\,\mathsf{name}_2$ *in* $p; z\,!\,\mathsf{name}_3$ *in* q refers to three variables, one named name_1 and a second named name_2 that both belong to the

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

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 as p, depending on further restrictions that might be specified. Additional variables can be listed in this declaration; they are grouped by program and separated by semicolons. If no combination of variables in the system satisfies the constraints, then the dummy variables are undefined, and the rule reduces to a skip.

As a simple example of a context rule, consider a program with a context variable called c that holds the value of an exposed variable with the name data and located on an agent at the same location as the reference. This context variable simply represents the context, and it does not change the data stored on the agent owning the exposed variable. To achieve this kind of behavior, the specification relies on the existence of the built-in exposed variable with the name location, locally referred to as λ. The context rule for the context variable 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.

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

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 selection of the exposed variables. The formal expression non-deterministically selects a variable to pull data from that satisfies a set of conditions. These conditions rely on the selection of a second exposed variable 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 ensures that the two variables (d and l) are located in the same program. 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 facilitate modeling real-world context-aware interactions, we revisit the acquaintance list example from earlier in the section. More extensive examples will be discussed in Section 4.

In Fig. 2, we gave only a high level description of the context rules required to define an agent's acquaintance list. To define the membership qualifications

exactly, the agent uses a context rule that adds qualifying agents to the context variable Q that stores the acquaintance list. In this particular case, assume that the program wants to restrict the acquaintance list members to other agents within some predefined range. This range is stored in a local variable whose local handle is referred to as *range*. The acquaintance list context variable can be defined using the following rule:

$$\begin{array}{ll} Q \textbf{ uses} & l \,!\, \mathsf{location} \textit{ in } a \\ \textbf{given} & |l - \lambda| \leq \textit{range} \\ \textbf{where} & Q \textbf{ becomes } Q \cup \{a\} \\ \textbf{reactive} & \end{array}$$

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 a reactive statement that is enabled when an agent is within range, 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 of 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:

$$\begin{array}{ll} Q \textbf{ uses} & l \,!\, \mathsf{location} \textit{ in } a \\ \textbf{given} & |l - \lambda| > \textit{range} \\ \textbf{where} & Q \textbf{ becomes } Q - \{a\} \\ \textbf{reactive} & \end{array}$$

Governing Universal Behaviors. Fig. 1 showed that the final portion of a Context UNITY system specification is a **Governance** section. It 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:

$$\begin{array}{ll} \textbf{use} & \textit{quantified variables} \\ \textbf{where} & \textit{restrictions on quantified variables} \\ & \textit{expr}_1 \textbf{ impacts } \textit{exposed variable}_1 \\ & \textit{expr}_2 \textbf{ impacts } \textit{exposed variable}_2 \\ & \ldots \end{array}$$

As a simple example of governance, imagine a central controller that, each time its governance rule is selected, non-deterministically chooses an agent in the system and moves it, i.e., it models a random walk. 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 along the line; 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:

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

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:

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

Using the unique combination of independent programs, their context rules, and universal governance rules, Context UNITY possesses the ability to model a wide-variety of applications in the area of context-aware computing. We demonstrate this in Section 4 by providing snippets of Context UNITY systems required to model applications taken from the context-aware literature. First, in the next section, we briefly overview the proof logic associated with the Context UNITY model.

3 Proof Logic

Context UNITY has an associated proof logic largely inherited from Mobile UNITY [6], 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 all of these systems, the fundamental aspect of proving programs correct deals with the semantics of individual program statements. UNITY contains only standard conditional multiple assignment statements, while both Mobile UNITY and Context UNITY extend

this model with reactive statements and transactions. Context UNITY also adds non-deterministic assignment statements. In all of these models, proving individual statements correct starts with the use of the *Hoare triple* [11].

For the normal statements 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 or Context 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 Context 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. It is required that the predicate H leads to a fixed-point and q in the reactive program $\mathcal{R}$. This proof 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 can use the following inference rule before application of the one above:

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

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

Finally, Context UNITY introduces the notion of non-deterministic assignment to the Mobile UNITY proof logic. The proof obligation of these non-deterministic assignments differs 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:

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

Special care must be taken to translate Context UNITY context rules from both the local program **context** sections and the **Governance** section to standard notation (i.e., to the appropriate normal or reactive statements) before applying the proof logic outlined here. Once translated as described in the previous section, proof of the system can be accomplished directly by applying the rules outlined above.

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 is 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 and Context UNITY use 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.

4 Patterns of Context-Awareness

Much published research acknowledges the need for applications that rapidly adapt to changes in resource availability and the operational environment. As a result, a number of researchers sought to provide context-aware software systems designed to function in a variety of operating scenarios. These systems vary in their approaches to managing context; models that underlie context-aware systems range from a simple client-server model in which servers provide context information directly to clients, to sophisticated tuple space coordination models in which the details of communicating context information is transparent to the application. In this section, we examine a representative set of context-aware systems found in the literature, abstract their key features, and suggest ways to model them in Context UNITY.

4.1 Simple Context Interactions

Initial work in context-aware computing resulted in the development of applications that use relatively simple context definitions. Such systems often separate concerns related to providing and using context. Many systems consist of *kiosks*, entities which provide context information to *visitors*, which use context and state information to adapt their behavior.

Applications exhibiting the characteristics of the simple kiosk-visitor interaction pattern include context-aware office applications such as Active Badge [12] and PARCTab [13]. In these systems, personnel carry devices that periodically communicate a unique identifier via a signal to fixed sensors, allowing the location of the carrier to be known. An application uses the location information to adapt the office environment accordingly in response to the changing location of the carrier, e.g., by forwarding phone calls to the appropriate office or changing the applications available on a workstation. Another type of context-aware applications that use simple context interactions relate to the development of tour guides, e.g., Cyberguide [14] and GUIDE [15]. In these applications, tourists carry mobile devices equipped with context-aware tour guide software. As a tourist moves about in a guide-friendly area, his display is updated according to locally stored preferences combined with context information provided by stationary access points located at points of interest.

In all of the context-aware applications described above, a particular type of entity provides context information and another type reads and uses the provided information. Generally, one of the parties is stationary, while the other is mobile. We can readily capture this style of interaction in Context UNITY. Agents providing context information to other agents in the Context UNITY system do so through the use of *exposed* variables. Agents obtain the provided context information through the use of *context* variables, the values of which are defined by values of selected exposed variables of context-providing agents.

Fig. 4 illustrates the interaction between a visitor and kiosks in a simple museum guide system. In this system, each stationary museum kiosk provides information about an exhibit at its location using an exposed variable. 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 each provide information about a certain sculpture by naming its exposed variable e "sculpture," and assigning to the variable a short textual description of the work of art at that location. As a particular visitor moves around the room, his context variable, c, defined to contain a co-located sculpture exhibit, changes in response to the available context. If new context information about a sculpture is available, the visitor's display is updated to show the information. The figure depicts what happens when a visitor walks around the museum. The initial position of the visitor agent is depicted by the dashed box labeled "Agent." As the visitor moves around the museum in the path indicated by the dotted arrow, the context variable c is updated. Specifically, when the visitor reaches the northeast corner of the museum, the context variable c is updated to contain information about the sculpture at that location. Such an application can be specified in the Context UNITY notation, as shown below.

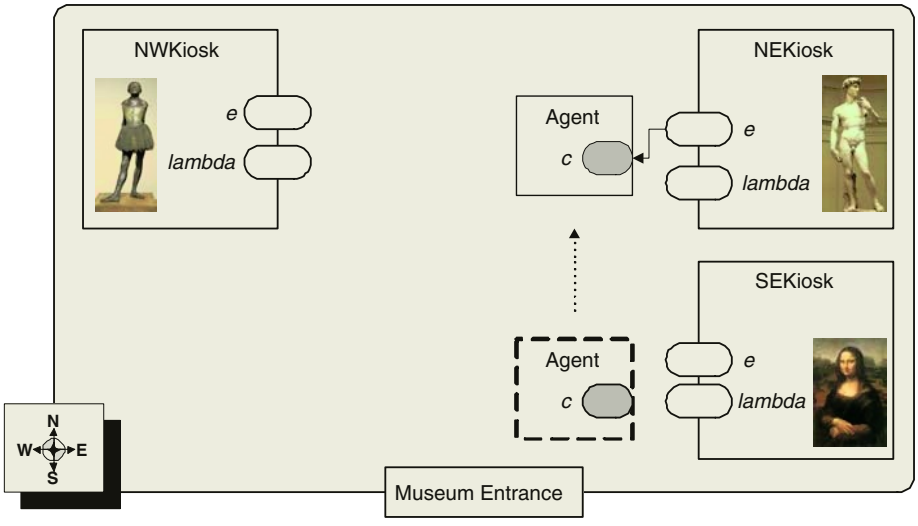

Fig. 4. A simple guide system in Context UNITY

For brevity, we show only the most interesting aspect of the system specification, which is a visitor's context rule:

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

More complex patterns of interaction are frequently utilized in the development of context-aware systems. In some systems, for instance, kiosks provide context information to a stationary context manager, and the context manager communicates directly with visitors to adapt their behavior accordingly given the current conditions of the environment. An instance of this pattern of interaction is found in the Gaia operating system [16], which manages *active spaces*. An active space is a physical location in which the physical and logical resources present can be adapted in response to changes in the environment. A typical interaction in an active space is as follows: a user enters the active space and registers with the context manager, which uses information about the user and the environment to perform appropriate actions, e.g., turn on a projector and load the user's presentation. Such a system can be modeled in Context UNITY similarly to those systems described above that exhibit simple context interactions: users are providing context information to the context manager through the use of exposed variables, and the context manager uses context variables to obtain context information and react accordingly.

4.2 Security-Constrained Context Interactions

Security is a major concern in the development of all modern software systems, including those supporting context-awareness. In several systems, multi-level security mechanisms are provided through the use of *domains*. A domain provides a level of security and isolates the available resources according to the level of security offered. Agents authorized to operate within that domain have the ability to act upon all resources within a domain, and a domain may have an authorizing authority that grants and revokes entering and exiting agents' access rights. Examples of systems exhibiting such characteristics include the Gaia file system [16] and the multi-level access control proposed by Wickramasuriya and Venkatasubramanian [17].

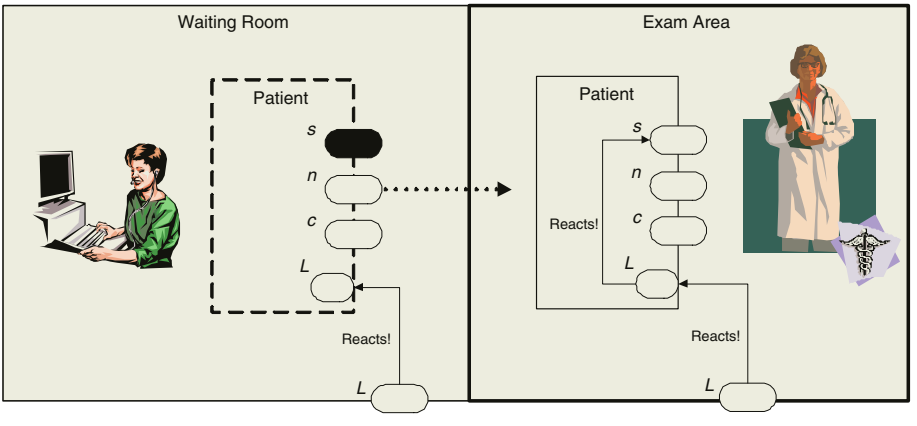

Fig. 5. An example security-constrained context-aware application in Context UNITY. In the waiting room domain, which offers a low level of security in its exposed variable L, the patient's sensitive information about symptoms is protected from inclusion in the domain by the symptom variable's access control function. The shading on the oval labeled s indicates that the symptom variable is not accessible to anyone in the environment. As the patient moves to the exam area domain offering high level security, the patient's domain security level is updated immediately, as indicated by the arrow labeled "reacts." As a result of the changed security level, a second reaction is triggered whose effect is to alter the access control function of the symptom variable s to allow the value to be available to those in the exam area domain.

Fig. 5 illustrates an example use of such an interaction style. In the example, a patient at a doctor's office must provide information about himself in order to receive treatment. Some of the information provided is fairly public knowledge and can be viewed by the receptionist and other patients, e.g., name and contact information. Other information is highly sensitive and personal, e.g., health history, and should only be shared with a doctor. To facilitate this kind of interaction, the doctor's office is divided into two areas that provide different levels of privacy: the waiting room and the exam area. The waiting room is a public space (low-security), since the receptionist and other patients in the waiting room can

view the information provided therein. The exam area is private (high-security), since only the patient and doctor can view the information.

To describe such applications in Context UNITY, domains could reveal their security level using an exposed variable L named "security level." Each patient agent uses a context rule for its context variable L to discover the level of security offered by the domain in which it is located. Because the definition is built to be strongly consistent using a reactive statement, the agent's perception of the security level offered by its current domain is guaranteed to be accurate and up to date. Each patient provides his name, contact information, and symptoms through the use of exposed variables n, c, and s. A patient controls how his information is made available through the use of each variable's access control function. This access control function can be changed during the execution of the program to reflect the agent's changing data protection needs. Using a reaction, it is possible to ensure that the access control function is immediately changed to reflect a change in the security level as soon as a new domain (and hence, a new level of security) is entered.

4.3 Tailored Context Definitions

Often, the amount of context information available to a context-aware agent grows large and unmanageable. To avoid presenting an agent with an overwhelming amount of context in such a scenario, it is desirable to limit the amount of context information that the agent "sees" based on properties of its environment. An example of a context-aware system that does just this is EgoSpaces [3], a middleware for use in ad hoc networks. At the heart of EgoSpaces is the *view* concept, which restricts an agent's context according to the agent's individualized specification. A view consists of constraints on network properties, the agents from which context is obtained, and the hosts on which such agents reside. These constraints are used to filter out unwanted items in the operational environment and results in presenting the agent with a context (view of the world) tailored to its particular needs.

In a general sense, systems such as EgoSpaces consist of possibly mobile agents that are both providers and users of context, and a context management strategy that is performed on a per-agent basis. An individualized context is managed on behalf of each agent by matching items from the entire operational environment against the restrictions provided in the view definition, and presenting the result to the agent as its context. Such a system can be readily expressed in Context UNITY. To act as a context provider, an agent generates pieces of context information and places them in an exposed variable, a tuple space, in the case of EgoSpaces, i.e., a data repository consisting of tuples 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. They allow other agents to filter the operational environment according to the host and agent constraints in their view definitions. To act as a context user, we model 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 to "see" can be described as follows:

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

The function *reachable* encapsulates the network constraints that establish whether an agent should or should not be considered based on network topology data. The notation $v \uparrow i$ indicates a projection over the set v that contains tuples owned by the agent i. 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. In order for an agent to perform changes to the view v and have them propagate to the correct tuple space *lts* additional context rules are needed.

4.4 Uniform Context Definition

Coordination models offer a high degree of decoupling, an important design characteristic of context-aware systems. In many distributed computing environments, tuple spaces are permanently attached to agents or hosts. In some models, these pieces merge together to logically form a single shared tuple space in a manner that takes into consideration the connectivity among agents or hosts. An agent interacts with other agents by employing content-based retrieval (`rd(pattern)` and `in(pattern)`), and by generating tuples (`out(tuple)`). Often, the traditional operations are augmented with reactions that extend their effects to include arbitrary atomic state transitions. Systems borne out of such a tuple space coordination paradigm can be considered context-aware; an agent's context is managed by the tuple space system in the form of tuples in a logically shared tuple space.

Examples of such context-aware systems are TSpaces [18], JavaSpaces [19], MARS [20], and LIME [21]. A common characteristic of all these systems is the fact that agents that enter in a sharing relation have the same definition of context, i.e., the context rules are uniform and universally applied. Among the systems we cite here, LIME is the most general, as it incorporates both physical mobility of hosts and logical mobility of agents, and provides tuple space sharing in the most extreme of network environments – the ad hoc network. In LIME, agents are units of execution, mobility, and data storage, while hosts are simply containers of agents. Hosts may be mobile, and agents can migrate from host to host. Agents may be associated with several local tuple spaces, distinguished by name. Since it is a passive entity, a host has no tuple space. A LIME agent's relevant context is determined by the logically merged contents of identically named tuple spaces held by mutually reachable agents.

To capture the essential features of context-aware systems having the characteristics described above in Context UNITY, it suffices to endow an agent with one exposed variable named localTS that offers its local tuple space for sharing and a second exposed variable named sharedTS that should provide access to all the tuples making up the current context. The value of the latter is the union of tuples contained in exposed local tuple space variables belonging to connected agents. Connectivity can be defined based on various properties of the network, e.g., network hops, physical distance, etc. In MARS, only agents residing on the same host are *connected*. In LIME, agents are *connected* when residing on the same host or on physically connected hosts.

A final and important point to note about the modeling of such systems is that since the shared tuple space definition is uniform across all agents, we can capture it in the **Governance** section of a Context UNITY system. While it is possible to define an agent's context locally in its program description, using the **Governance** section 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.

This brings to an end our discussion on how Context UNITY relates to some of the existing models of context-awareness. The most striking observation about this informal evaluation of the model is the simplicity exhibited by each of the context rules that were generated in this section.

5 Conclusions

The formulation of the Context UNITY model is a case study 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 program will encounter throughout its life time. The model enables

this decoupling 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 then 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. While we acknowledge that further refinements and evaluation of the model are needed, all indications to date suggest that the essential features of context-aware computing are indeed present in the model.

Acknowledgements

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. Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the development of context-enabled applications. In: Proceedings of CHI'99. (1999) 434–441
2. Hong, J., Landay, J.: An infrastructure approach to context-aware computing. Human Computer Interaction **16** (2001)
3. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering. (2002) 21–30
4. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: IEEE Workshop on Mobile Computing Systems and Applications. (1994)
5. Julien, C., Payton, J., Roman, G.C.: Reasoning about context-awareness in the presence of mobility. In: Proceedings of the 2^{nd} International Workshop on Foundations of Coolrdinations Languages and Software Architectures. (2003)
6. Roman, G.C., McCann, P.J.: A notation and logic for mobile computing. Formal Methods in System Design **20** (2002) 47–68
7. McCann, P.J., Roman, G.C.: Compositional programming abstractions for mobile computing. IEEE Transactions on Software Engineering **24** (1998) 97–110
8. Fok, C.L., Roman, G.C., Hackmann, G.: A lightweight coordination middleware for mobile computing. In: Proceedings of the 6^{th} International Conference on Coordination Models and Languages. (2004) (to appear).
9. Back, R.J.R., Sere, K.: Stepwise refinement of parallel algorithms. Science of Computer Programming **13** (1990) 133–180
10. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, NY, USA (1988)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580, 583
12. Harter, A., Hopper, A.: A distributed location system for the active office. IEEE Networks **8** (1994) 62–70

13. Want, R., et al.: An overview of the PARCTab ubiquitous computing environment. IEEE Personal Communications **2** (1995) 28–33
14. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997) 421–433
15. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proceedings of MobiCom, ACM Press (2000) 20–31
16. Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.: Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing (2002) 74–83
17. Wickramasuriya, J., Venkatasubramanian, N.: A middleware approach to access control for mobile concurrent objects. In: Proceedings of the International Symposium on Distributed Objects and Applications. (2002)
18. IBM: T Spaces. http://www.almaden.ibm.com/cs/TSpaces/ (2001)
19. Sun: Javaspaces. http://www.sun.com/jini/specs/jini1.1html/js-title.html (2001)
20. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
21. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21^{st} International Conference on Distributed Computing Systems. (2001) 524–533