SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:1307–1330
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.760

SP&E

Using context and preferences to implement self-adapting pervasive computing applications

Karen Henricksen[1,*,†], Jadwiga Indulska[1] and Andry Rakotonirainy[2]

[1]*School of Information Technology and Electrical Engineering, The University of Queensland, QLD 4072, Australia*
[2]*Centre for Accident Research and Road Safety - Queensland (CARRS-Q), Queensland University of Technology, Beams Road, Carseldine, QLD 4034, Australia*

SUMMARY

Applications that exploit contextual information in order to adapt their behaviour to dynamically changing operating environments and user requirements are increasingly being explored as part of the vision of pervasive or ubiquitous computing. Despite recent advances in infrastructure to support these applications through the acquisition, interpretation and dissemination of context data from sensors, they remain prohibitively difficult to develop and have made little penetration beyond the laboratory. This situation persists largely due to a lack of appropriately high-level abstractions for describing, reasoning about and exploiting context information as a basis for adaptation. In this paper, we present our efforts to address this challenge, focusing on our novel approach involving the use of preference information as a basis for making flexible adaptation decisions. We also discuss our experiences in applying our conceptual and software frameworks for context and preference modelling to a case study involving the development of an adaptive communication application. Copyright © 2006 John Wiley & Sons, Ltd.

Received 19 April 2004; Revised 5 April 2006; Accepted 7 April 2006

KEY WORDS: context-awareness; self-adapting applications; pervasive computing; preference modelling

1. INTRODUCTION

Context-awareness has recently emerged in research circles as a popular approach for building self-adapting pervasive computing applications. Context-aware applications exploit information about the

*Correspondence to: Karen Henricksen, School of Information Technology and Electrical Engineering, The University of Queensland, QLD 4072, Australia.
†E-mail: karen@itee.uq.edu.au

context of use, such as the available computing, network and I/O resources and the current location and activity of the user, in order to provide seamless operation in the face of mobility and intelligent support for users' evolving requirements. Context information is typically obtained from sources such as sensors (e.g. GPS receivers, tilt sensors and accelerometers), network monitors, user profiles, and context-aware applications that report their current state.

A variety of prototypical context-aware applications have been developed and evaluated, mainly in research settings, including context-aware guides that present tourists with information tailored to their current location [1,2], and communication tools that route calls and messages so as to minimize intrusion and increase relevance to the user's current activity [3]. Despite the recent flurry of interest, however, most context-aware applications have not been pursued beyond prototype form. This is largely the result of significant technical and software engineering overheads associated with acquiring, interpreting and exploiting reliable context information from sensors and other sources. The focus of recent research has been mostly on the first two problems. This research has resulted in software infrastructures such as the Context Toolkit [4] and Solar [5], which consist of software components that transform raw sensor outputs into the types of high-level context information that are meaningful to applications. In contrast, our focus has been on the third problem. In particular, we have been investigating abstractions that simplify the specification and implementation of adaptive behaviour in context-aware applications. Our goal has been to promote the development of flexible applications, so that:

- applications are not tightly coupled to the infrastructure used to gather context information or the context model; this allows the infrastructure and model to evolve as new sources of information become available or existing sources disappear (e.g. as a result of a transition from a sensor-rich environment to a sensor-poor environment);
- users can customize their applications to meet their own particular requirements; and
- applications can accommodate user requirements that evolve over time.

An added advantage of building applications that offer these types of flexibility is that the software engineering process can be simplified by taking advantage of increased opportunities for exploring alternative application behaviours and performing fine-tuning. This removes the need to perfectly understand and capture requirements by the design phase, and also reduces maintenance overheads.

These improvements in the software engineering process will make it easier to apply context-awareness in new application domains, thereby indirectly helping to overcome the following two obstacles that are currently limiting the success of context-aware applications.

- *Imperfect understanding of the truly compelling uses of context-awareness.* While context-awareness is widely recognized by researchers as crucial ingredient in pervasive computing environments, the 'killer applications' that will first win user acceptance and break into the marketplace have not yet been identified.
- *Novel design challenges.* A variety of new usability problems have been observed in context-aware applications. As a result, new design approaches are needed in order to achieve an appropriate balance of application autonomy to user control [6], provide consistency, transparency and traceability so that users can understand the actions of their applications [7], and address privacy concerns in relation to sensitive types of context information [8].

DOI: 10.1002/spe

2. BACKGROUND AND APPROACH

Although the need for suitable high-level abstractions for programming flexible context-aware applications has been acknowledged for several years [9], very little progress has been made in this area. The best known programming model is the stick-e note model proposed by Brown [10], which aims to transform the development of simple information retrieval applications into a document authoring problem. While the model has been successfully used for building note-taking tools and a tour guide [11,12], it is not applicable to most application domains. Similarly, the context-sensitive object model developed by Yau *et al*. [13] was designed expressly with the problem of dynamic reconfiguration in mind, and is not suitable for many types of application, including information retrieval applications and the communication tool we present later in this paper.

The current lack of suitable generic abstractions for context-awareness also extends to the area of context modelling. Many applications are developed using primitive, flat models of context, such as attribute–value pairs. In addition, context-aware applications must usually explicitly formulate queries using these primitive abstractions to obtain context information from one or more distributed context services. This approach is cumbersome, particularly when there are many different types of context information and many context providers.

Our work of the past few years represents one of the early efforts to shift away from primitive programming models to increasingly high-level abstractions that simplify the task of the application developer and offer greater flexibility. As a first step, we developed an integrated set of context modelling techniques for graphically constructing context models for individual applications, as well as describing and querying context in terms of high-level situation abstractions. Unlike previously proposed context modelling approaches, our approach supports diverse types of context information from a range of sources and addresses complex problems such as dependencies and incomplete, ambiguous and otherwise imperfect information. Crucially, it also provides alternative views of context depending on the level of detail and granularity required, and supports incremental refinement of context models over the software engineering lifecycle as advocated by Coutaz and Rey [14].

Our context modelling abstractions have been described in previous papers [15,16] and are not our focus here. In this paper, we discuss our recent efforts to build further abstractions on top of our context modelling approach in an attempt to:

- decouple the application from the context model so that the model can easily evolve;
- remove the need for the application to explicitly discover and query the context providers; and
- reduce the need to encode the application's response to different contexts as complex decision logic in the source code.

In order to meet these requirements, we have explored a novel approach in which preferences/policies are used to drive the context-dependent behaviour of applications. Although user preferences have previously played an important role in some context-aware applications, such as Active Messenger [17], these have always been expressed and used in an informal and *ad hoc* manner. In contrast, our work focuses on generic preference and programming models that can be used for arbitrary context-aware applications, and can facilitate consistent behaviour and preference sharing among applications. The flexibility of our approach comes from the fact that preferences are specified externally to the application, implying that they can be easily manipulated and evolved to change the behaviour without modifying the source code. By serving as the link between the context and

DOI: 10.1002/spe

appropriate application behaviours, preferences also place a layer of separation between the application and its context model, allowing the application and the model to evolve independently of one another.

In this paper, we introduce our preference model, the programming model we use in conjunction with this to support a common form of context-dependent choice problem, and our first attempt to implement a software infrastructure and programming toolkit for pervasive computing environments. We also discuss our early experiences with using the models and infrastructure to implement a context-aware communication application.

The structure of the paper is as follows. Section 3 presents a survey of current approaches to modelling user requirements and preferences in context-aware systems, together with a brief introduction to preference modelling techniques used in other fields. It concludes that these modelling techniques are inadequate for capturing the types of context-dependent preferences that are required for controlling and customizing the behaviour of context-aware software. Section 4 introduces our novel preference modelling approach, which was developed in response to the findings of the survey. Section 5 presents the programming model we use in conjunction with the preference model to support choice based on a combination of context and user preference information, and outlines our first attempt to develop a programming toolkit that implements the programming model. Section 6 positions the toolkit within our broader infrastructure for context-aware computing, Section 7 presents a case study in application development using our models and infrastructure and Section 8 provides a discussion of our current and future work.

3. A SURVEY OF PREFERENCE MODELLING TECHNIQUES

3.1. Modelling preferences in context-aware systems

The importance and difficulty of accurately identifying user requirements that are influenced by context, and then mapping these to appropriate behaviour in context-aware applications, are well recognized [18–20]. These tasks are made especially challenging by the fact that: (i) users may have very diverse requirements; (ii) many types of context information may need to be considered; and (iii) the inherent uncertainty present in most types of context information tends to cloud decisions about what actions are appropriate. Unfortunately, very little research has focused on these problems, and techniques for eliciting and capturing user requirements in relation to context and customizing context-aware applications remain primitive.

An interesting exception is the research of Byun and Cheverst, which seeks to integrate user modelling techniques into context-aware software [18,19]. This work focuses on the use of probabilistic learning techniques in order to enable applications to automatically elicit user requirements. The learned user models are exploited to support a variety of proactive behaviours, such as the generation of reminder notices when the due date for a library book draws near, or when an office door is left open and the user is not likely to return soon.

While this work appears promising, explicit means of representing user requirements in context-aware systems are also needed. The use of an explicit representation allows users to supply their own preferences if desired, and also provides a tool for exposing preference information and thereby offering transparency, so that users are able to understand the basis for their applications' actions, and to make corrections as necessary. This approach can be compatible with automated learning techniques, as we discuss briefly in Section 8.

DOI: 10.1002/spe

Currently, the only available standard for capturing preferences for use in context-aware systems is CC/PP [21]. This standard for Web context adaptation views preferences as a type of context information, and adopts a simple attribute-based information model. Unfortunately, this approach is only suitable for expressing very simple requirements in relation to Web applications (such as the set of languages that are acceptable for presenting information to the user), but not for more sophisticated, context-dependent preferences (such as those used by the communication application described later in this paper to support appropriate choice of communication channels). CC/PP also suffers from a variety of other limitations, which are described in [22].

3.2. Other preference modelling approaches

While no sophisticated preference modelling techniques have been developed specifically to support context-awareness, the problem of modelling preferences has been widely researched in other fields. In this section, we survey a of variety of preference modelling techniques and analyze their suitability for use in context-aware applications. As the body of previous research in preference modelling is extensive, this survey is intended to be illustrative rather than exhaustive.

Two common preference modelling techniques are the quantitative and qualitative approaches [23]. The quantitative approach imposes an ordering on a set of candidate choices by assigning to each choice a score that reflects its desirability. This approach has been widely explored in the field of decision theory, with preferences being commonly captured as utility functions that map attributes such as cost, reliability or performance to a corresponding numerical measure of utility [24]. Decision making is consequently transformed into an optimization problem that seeks to maximize the utility of the choice.

One of the shortcomings of this approach is the difficulty associated with constructing an appropriate utility function, particularly when there are many parameters involved. This makes the approach unsuitable for use in context-aware applications, in which preferences must be specified over complex and dynamically changing contexts.

An alternative quantitative preference model proposed by Agrawal and Wimmers [25] does not suffer from this problem. It allows users to supply many separate preferences which can later be combined to support decision making. Unlike utility functions, the preference functions proposed by Agrawal and Wimmers are applied to tuples of values that represent the candidate choices. The functions produce scores belonging to the set $[0, 1] \cup \{\natural, \perp\}$, where the numerical scores capture relative desirability (such that higher scores indicate greater desirability), the special score $\natural$ represents a veto, and $\perp$ represents indifference (that is, an absence of preference).

This model makes it possible to combine a set of potentially conflicting preferences for a single user, as well as the preferences of groups of users, by combining preference functions; this makes the task of modelling complex requirements much more manageable than with the approach based on utility functions. However, the model is not appropriate for use in context-aware systems as it can only capture preferences that are static, not those that vary according to the context.

The qualitative approach differs from the quantitative approach in that it does not rely on scores to determine relative preference, but instead describes the ordering of candidate choices directly. This ordering is often captured by binary preference relations that describe relative desirability between alternatives, in pairwise fashion. Chomicki [23] demonstrates the application of this approach to rank the results of database queries according to user preferences. Preference relations, specified using first-order logic, are used to filter results by removing any tuple that is *dominated* (i.e. any tuple for which there is at least one other tuple in the results that is preferable according to the relation).

DOI: 10.1002/spe

The qualitative approach offers greater expressive power than the quantitative approach, as it can be used to describe non-transitive preferences [23,24][‡]. However, a major problem with the qualitative approach is the difficulty associated with combining preferences. Although various solutions have been proposed, these are either overly simplistic, yielding unacceptably poor results, or else are computationally expensive (e.g. NP-complete). Chomicki's approach, which combines preference relations using set union, intersection and difference, falls into the first category as it often leads to empty result sets when cycles occur in the combined preference relation. Cohen *et al.* [26] present several algorithms that fall into the second category. As the need to combine diverse and often conflicting preferences is common to all but the most trivial context-aware systems, while non-transitive preferences are infrequently (if ever) needed, we favour the quantitative approach.

Other preference models that do not fall into the qualitative or quantitative categories have also been explored. The Vector Space Model evaluates choices according to similarity between vectors, and is primarily of interest in problems involving the ranking of documents based on their text (e.g. in information retrieval). The model is typically applied to these problems as follows. Documents and user preferences are represented by n-dimensional vectors, where each dimension corresponds to a term or concept. In order to determine the relevance of a document, the similarity between the document vector and the preference vector is computed by measuring the cosine of the angle between the two vectors.

Çetintemel *et al.* [27] demonstrate the use of the model for pushing Web pages to interested users. They represent user preferences as a collection of interest vectors, where each vector corresponds to a cluster of similar documents that are relevant to the user. User feedback is exploited to evolve the interest vectors over time. The main obstacle associated with using this model to support context-aware choice is that all of the dimensions over which preferences range must be expressed numerically. This is problematic because there are many types of context information that do not have a natural quantitative representation.

3.3. Conclusions

This brief survey indicates that the unique challenges involved in using preferences as a basis for customizing context-aware software demand a new approach to preference modelling. A quantitative approach that allows preferences to be easily combined, in the style of the model of Agrawal and Wimmers, appears to be the most promising, and this the solution that we pursue in the following section.

4. PREFERENCE MODEL

In this section, we begin by defining the problem space addressed by our preference model and its key requirements. Subsequently, we present the model, focusing on how we link preferences to contexts and combine preferences into preference sets and composite preferences.

[‡]This allows somewhat unintuitive preferences such as the following to be expressed: a is preferred to b, and b is preferred to c, but c is preferred to a.

DOI: 10.1002/spe

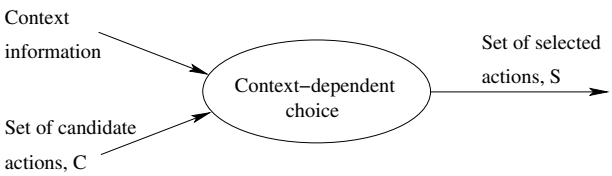

Figure 1. The choice problem, without preferences ($S \subseteq C$).

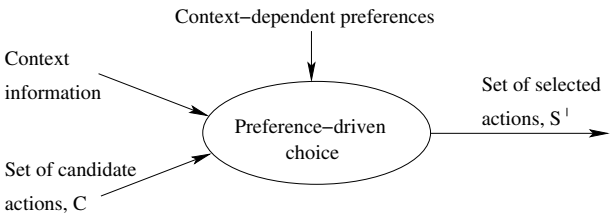

Figure 2. Adding preferences to the choice problem ($S' \subseteq C$).

4.1. The problem space

Context-aware applications exhibit flexible and autonomous behaviour by adapting to the context. Adaptation typically involves a choice problem in which the inputs are the relevant context information and the set of possible actions that can be performed by the application, and the output is a selected subset of the input actions considered appropriate to the context. This problem is illustrated in Figure 1.

Context-dependent choices are common to virtually all context-aware applications. For example, in context-aware communication applications, choices of suitable communication channels (e.g. suitable phone numbers or e-mail addresses) are driven by the current context, while context-aware information retrieval relies on a similar choice in order to select or filter documents to match the user's activity and interests [28]. Other applications often rely on context information to choose suitable I/O devices for interaction with the user, or to select services that best meet context-based criteria, such as proximity to the user.

We are interested in using preferences as means to support this common choice problem as shown in Figure 2. Here, the role of preferences is to allow choices to be easily customized to meet user requirements, and also to be evolved in response to changes in the types of available context information and the set of actions supported by the application.

4.2. Requirements

The principal requirements of a preference model that supports this choice problem in a pervasive computing setting are related to usability. The success of the model hinges, in particular, on the ease

DOI: 10.1002/spe

with which users can formulate their own preferences. The model must therefore be conceptually simple, and also allow application developers, administrators and sophisticated users to independently define a variety of simple requirements, which can be incrementally combined to build increasingly sophisticated preference sets[§]. The latter requirement is particularly challenging in light of the likelihood of conflicting preferences. The suitability of the preference model for use with automated preference elicitation and evolution techniques is likewise important, as these techniques will be essential in complex systems to help lighten or remove the burden placed on users to maintain accurate preference information.

In addition to meeting high standards of usability, the preference model must provide adequate expressive power. In particular, it should support common policy concepts, including prohibition and obligation, allowing users to both forbid and require choices in certain contexts.

Finally, the preference model must be generic enough to be applicable to arbitrary context-aware applications, and to support sharing of preference information over a set of applications.

4.3. Overview of the preference model

The preference model that we developed in response to these requirements employs a scoring mechanism that is loosely based on the scheme proposed by Agrawal and Wimmers. Each preference assigns to a candidate choice a score which is one of the following:

- a numerical value in the range [0,1], where larger scores imply increasing preference; or
- one of the special scores in the set $\{\natural, \bot, \overline{\wedge}, ?\}$, where $\natural$ represents veto or prohibition and $\bot$ represents indifference (as in the model of Agrawal and Wimmers), while the new score $\overline{\wedge}$ represents obligation (indicating that a candidate choice must be selected), and ? represents an error condition.

Preferences are specified as pairs consisting of context and scoring components. The context takes the form of a condition specified in our own variant of predicate logic, which is described briefly in the following section, while the scoring component is an expression that is evaluated to yield one of the above scores. Both are specified over a set of variables that characterize a candidate choice and selected aspects of the current application state such as the identity of the user(s) or historical information about previous choices. Preferences are evaluated against a context and a set of variable bindings, termed a *valuation*, using the following simple rule. If p is a preference, $p.c$ is the context associated with the preference, and $p.s$ is the scoring expression, the score assigned by the preference p in a context C to a candidate choice represented by the valuation v is

$$rate(p, C, v) = \begin{cases} score(p.s, C, v) & \text{if the condition } p.c \text{ is true for } C \text{ and } v \\ \bot & \text{otherwise} \end{cases}$$

where $score(p.s, C, v)$ denotes the value of the scoring expression $p.s$ when evaluated with respect to C and v. That is, when the context condition, $p.c$, holds, the score assigned by the preference is simply

[§] We expect that naive users would never be directly exposed to the preference model, and discuss this further in Section 4.3.

DOI: 10.1002/spe

```
p1 = when SynchronousMode(channel) ∧ ¬CanUseChannel(callee, channel)
     rate ⊥
p2 = when SynchronousMode(channel) ∧ Occupied(callee) ∧ ¬Urgent(priority)
     rate ⊥
p3 = when Urgent(priority) ∧ SynchronousMode(channel)
     rate 1
p4 = when Urgent(priority) ∧ ¬SynchronousMode(channel)
     rate 0.5
```

Figure 3. Example preferences used by the context-aware communication application.

that defined by $p.s$. Otherwise, when the condition does not hold, the preference yields the indifferent score.

Some example preferences are shown in Figure 3. These preferences are used by a communication application in the task of selecting suitable channels (e.g. e-mail, telephone or videoconference channels) for communication between users. (Aside: this application is discussed in detail as a case study in Section 7.) The preference name is shown at the left, while the context and scoring expression are preceded by the keywords `when` and `rate`, respectively. A variety of predefined situations (*SynchronousMode*, *CanUseChannel*, *Occupied* and *Urgent*) are used to describe the contexts in which the preferences apply. These situations are discussed in the following section.

The first preference forbids the use of synchronous channels, such as telephone and videoconference channels, when the user does not have access to all of the requisite devices. Similarly, `p2` forbids the use of synchronous channels for non-urgent purposes when the user is engaged in an important activity such as a meeting. Preferences `p3` and `p4` together imply that synchronous channels are preferred for high-priority interactions: `p3` assigns these the highest possible numerical score (1), while `p4` assigns all asynchronous channels (such as e-mail and SMS) a score of 0.5[¶]. Each of these preferences employs a simple (constant-valued) scoring expression, but considerably more complex expressions are also supported. We show further examples later in Section 4.5.

We note at this point that the preference format shown in Figure 3 need never be exposed directly to users. Instead, users might select from standard preference profiles prepackaged with their applications, or manipulate a set of application-specific options (which would then be mapped automatically to appropriate preferences) through customization interfaces. In some applications, users might never specify preferences at all, as these would instead be learned based on the actions or explicit feedback of the user. In addition to the obvious benefits to users of these approaches, in terms of complexity and effort required, they also help to avoid the problem of unexpected behaviours when users change preferences in uncontrolled ways. We discuss this problem in Section 7.3.2.

[¶]These scores have been chosen somewhat arbitrarily and can be adjusted to give more or less weight to either of the channel types. We envisage schemes in which the scores are dynamically adjusted by the application using learning based on user feedback (this is discussed further in Section 8), or manipulated manually by users using graphical user interface (GUI) components such as sliders.

DOI: 10.1002/spe

As seen from the examples, individual preferences generally capture very narrow slices of users' overall requirements. The power of our preference model comes not from using these simple preferences in isolation, but from aggregating sets of preferences, often from multiple sources, including application default preferences, user-defined preferences for individuals or groups of users, and organizational preferences/policies (e.g. provided by an employer or service provider). We describe how this is done in Section 4.5. Prior to this, however, we present a brief discussion of our context modelling approach.

4.4. Describing contexts (situations) in which preferences apply

In our general approach for building context-aware software [16], we use novel logic-based formalisms and support two levels of context modelling depending on the level of detail and abstraction required. At the lower level, we represent context information as a set of atomic facts. These facts conform to a context model, which defines the allowed fact types, meta-data on these (such as relevant quality metrics and dependencies), and a variety of integrity constraints capturing the domain semantics. Above the fact-based model, we provide a situation abstraction for defining abstract classes of context in terms of logical expressions. These can be easily combined using conjunction, disjunction and negation to describe increasingly complex contexts. Some example situations (*SynchronousMode*, *CanUseChannel*, *Occupied* and *Urgent*) were seen in the previous section.

As our preference model directly relies only on the situation abstraction, we focus on this level of modelling here and refer the reader to our earlier papers [15,16] for information on the fact-based modelling approach.

We define situations in terms of expressions formulated using a novel variant of predicate logic. These expressions consist of a combination of:

- logical operators and quantifiers;
- equalities and inequalities (e.g. $t_1 = t_2$ and $t_1 < t_2$);
- predefined and user-defined functions; and
- assertions over fact types, written $f[t_1, \ldots, t_n]$, where f is a fact type of arity n and $t_1, \ldots, t_n$ are arbitrary terms (constants, variables or simple mathematical expressions).

Assertions are evaluated against the set of available context facts and a set of variable bindings to yield a truth value. As the set of facts is often incomplete and/or ambiguous, we adopt a three-valued logic, such that an assertion adopts the third logical value (*unknown*) when its truth cannot be absolutely determined from the available information.

The situations used to express the preferences presented in the previous section are defined by way of example in Figure 4. As seen in the examples, we adopt restricted forms of universal and existential quantification for reasons of efficiency and safety. These ensure that all variables are immediately bound according to the context. Quantifications contain three parts joined by the separator symbol (•). The first lists the variables as usual, and the third the logical expression over which the quantification occurs. The middle (binding) part contains an assertion that restricts the values of the variables appearing in the quantification. Evaluation proceeds as follows. Values of the variables that satisfy the binding assertion are computed by consulting the available context information and matching the assertion to a set of facts. The result of the quantification is then computed by evaluating the expression supplied in the third part of the quantification over the each of the variable bindings.

DOI: 10.1002/spe

SynchronousMode(*channel*) :
    $\exists$*mode* • *HasMode*[*channel*, *mode*] • *Synchronous*[*mode*]
CanUseChannel(*person*, *channel*) :
    $\forall$*device* • *RequiresDevice*[*channel*, *device*] •
    (*LocatedNear*[*person*, *device*] $\wedge$ *PermittedToUse*[*person*, *device*])
Occupied(*person*) :
    $\exists t_1, t_2$, *activity* • *EngagedIn*[*person*, *activity*, t_1, t_2] •
    ($t_1 \leq$ *timenow*() $\wedge$ *timenow*() $\leq t_2 \wedge$
    (*activity* = "in meeting" $\vee$ *activity* = "taking call"))
Urgent(*priority*) :
    priority = "high"

Figure 4. Example situation predicates.

Table I. Fact types.

Fact type	Description
HasMode	Associates communication channels with a mode such as e-mail, telephone or SMS.
Synchronous	Lists all modes that are synchronous.
RequiresDevice	Indicates the devices that are required in order to use a given communication channel.
LocatedNear	Records pairs of users and devices that are in close proximity to one another.
PermittedToUse	Indicates which users are allowed to use which devices.
EngagedIn	Captures a history of user activities, using a pair of timestamps to describe the duration of the activity.

The situations defined in Figure 4 assume the existence of the fact types listed in Table I. The *SynchronousMode* situation holds for a given communication channel provided that the mode associated with this channel is synchronous (i.e. appears in the *Synchronous* fact type). *CanUseChannel* is satisfied for a given person, p, and communication channel, c, when all of the devices required in order to use c are located in close proximity to p, and p additionally has permission to use these devices. The *Occupied* situation reflects whether a given person is currently engaged in an activity that should not be interrupted ('in meeting' or 'taking call'), on the basis of the *EngagedIn* fact type. This is determined by examining exactly those activity facts for which the current time (returned by the function *timenow*()) overlaps with the recorded time interval. Finally, the *Urgent* situation holds whenever the *priority* variable has the value 'high'.

A more formal presentation of the situation abstraction, and a discussion of the properties of the situation logic in terms of expressiveness and efficiency of evaluation, can be found in [29].

DOI: 10.1002/spe

```
user&default = when true
               rate average(UserPrefs ∪ DefaultPrefs)
employer     = when true
               rate average(EmployerPrefs)
workinghours = when WorkingHours()
               rate override(user&default, employer)
afterhours   = when ¬WorkingHours()
               rate as(user&default)
allhours     = when true
               rate average({workinghours, afterhours})
```

Figure 5. Combining preferences. Names of preference sets have been capitalized here to distinguish them from names of composite preferences.

4.5. Combining preferences

In pervasive computing environments, users can have many applications, very large sets of preferences and requirements that change over time. Consequently, effective techniques are needed to organize and dynamically combine preferences. In this section, we describe the use of preference sets and composite preferences to satisfy these requirements.

Preference sets serve as a structuring mechanism, allowing preferences to be grouped according to the types of choice they support, the applications to which they are applicable, their owners and other factors. Preference sets are dynamic structures whose members can evolve over time. In addition, they can be combined arbitrarily using set union, intersection and difference.

Scores produced by the members of a preference set are combined to produce an aggregate score in accordance with a user-specified policy. In order to allow recursive composition of preferences, the policy is itself specified in terms of one or more (composite) preferences. Figure 5 illustrates some example composite preferences that incrementally combine the following preference sets: `DefaultPrefs` (the set of default preferences for a given application), `UserPrefs` (the set of user-defined preferences) and `EmployerPrefs` (the preferences of the user's employer). The predefined functions *average*, *override* and *as* are exploited here to combine scores; we describe these further shortly. Additional predefined functions corresponding to a variety of common statistical measures, such as median and weighted average, are also supported, as are arbitrary user-defined functions.

The first two composite preferences illustrate the aggregation of scores produced by entire preference sets using a form of averaging. These preferences are evaluated by first evaluating all of the members of the preference sets, and then combining the resulting scores using the *average* function. In the absence of the special scores $\natural$ (veto), $\overline{\wedge}$ (obligation) and ? (error), the function computes the mean of the numerical scores (or, if there are no numerical scores present, it produces the indifferent score, $\perp$). When both veto and obligation scores are present, or when there is an error score, the result is error. Otherwise, if there are only obligations without vetoes, or *vice versa*, these prevail.

The `workinghours` preference is defined so that during working hours, the composite employer preference overrides the combined user and default preferences whenever it produces either of the special scores $\natural$ or $\overline{\wedge}$ (as per the definition of the *override* function). In contrast, the `afterhours`

DOI: 10.1002/spe

preference is defined so that only the user and default scores are considered after hours, while the employer preferences are ignored. Here, the *as* function is used to indicate that the score of `afterhours` is identical to that of the `user&default` preference. Finally, the `workinghours` and `afterhours` preferences are combined using averaging to produce a preference that can be used at all times. As the contexts of the `workinghours` and `afterhours` preferences are disjoint, at most one of these preferences will produce a non-indifferent score at any time, and, as per the definition of the *average* function, this will be the score adopted by the `allhours` preference.

5. PROGRAMMING MODEL AND TOOLKIT

Having presented the fundamentals of our preference model, we now turn our focus for the remainder of the paper to the practical issues associated with using the preference model to implement self-adapting applications. We begin by describing a Java programming toolkit that we developed to support a programming model in which the application selects its actions based on the available context and preference information. We refer to this programming model as the *branching* model.

The toolkit supports various flavours of the basic choice problem that was outlined in Section 4.1. In some instances, the goal is to select exactly one choice (as, for example, in the case of a communication tool that has the task of selecting the single most appropriate communication channel for an interaction on behalf of the user). In others, the goal is to select all highly rated choices, or a fixed number of the best-rated choices (as in the case of a document retrieval application that returns the best matches to the user).

A representative fragment of the toolkit's branching API is shown in Figure 6. The API provides a variety of methods in order to accommodate many variants of the basic choice problem. In each case, the programmer supplies as parameters a set of candidate choices, the preference that is used to generate scores for the candidates, a valuation that encapsulates relevant aspects of the current application state, and a context (which is usually simply a reference to a repository of context information residing outside the application). The preference is almost always a complex composite preference in the style of the preferences shown in Figure 5. Each candidate choice, represented by a `Choice` object, is characterized by a valuation which captures the key parameters of the choice as variable bindings. This valuation is merged with the separate valuation that describes the application state when evaluating the preference; the merged valuation *must* supply values for all of the variables that appear in the preference (and its component preferences in the case of a composite preference), otherwise an error occurs.

The toolkit offers three broad usage styles. In the first style, scores are generated for the set of candidate choices based on the context and preferences, and it is left to the application to inspect and act upon the results. This usage style is supported by the `rate` method, which returns a `Scores` object containing a mapping of the candidate choices to scores.

The second usage style is supported by the `select*` methods. These return one or more choices that match specified requirements. `selectBest` returns the single choice which is assigned the highest numerical score. If there are several choices that receive this score, one is selected arbitrarily. Alternatively, if no choice is assigned a numerical score, the return value is null. `selectBestN` works identically, except that it returns the n highest rated choices (or all of the choices that receive numerical scores if there are fewer than n), where n is determined by the first parameter. Similarly, `selectAbove` returns all of the choices that receive numerical scores at or above a specified threshold and `selectMandatory` returns all choices that receive obligation scores ($\widehat{\wedge}$).

DOI: 10.1002/spe

```
Scores rate(Choice[] c, Preference p, Valuation v, Context cx);

Choice selectBest(Choice[] c, Preference p, Valuation v, Context cx);
Choice[] selectBestN(int n, Choice[] c, Preference p, Valuation v, Context cx);
Choice[] selectAbove(Score threshold, Choice[] c, Preference p, Valuation v,
                     Context cx);
Choice[] selectMandatory(Choice[] c, Preference p, Valuation v, Context cx);

void invokeBest(Choice[] c, Preference p, Valuation v, Action default,
                Context cx);
void invokeBestN(int n, Choice[] c, Preference p, Valuation v, Action default,
                 Context cx);
void invokeAbove(Score threshold, Choice[] c, Preference p, Valuation v,
                 Action default, Context cx);
void invokeMandatory(Choice[] c, Preference p, Valuation v, Action default,
                     Context cx);
```

Figure 6. Selected methods of the programming toolkit.

The remaining methods are almost identical to the `select*` methods, except that, instead of returning the selected choices, they automatically invoke actions associated with these. The action for each `Choice` object is embedded within its `invoke` method. A default action is supplied as an additional parameter, and this is executed in the case that there are no suitable candidate choices.

The use of the toolkit replaces the usual approach in which the application explicitly discovers and queries the context providers to obtain the required context information, and uses a complex sequence of if or case statements to express the decision logic. As well as considerably simplifying the source code, the toolkit-based solution is substantially more flexible as the decision process can be modified by editing preferences rather than code, and a looser coupling is achieved between the application and the context model and providers.

6. SOFTWARE INFRASTRUCTURE

The programming toolkit resides between context-aware applications and a set of supporting infrastructural components, which we briefly describe in this section. These components are responsible for acquiring, managing, storing and disseminating context information from a variety of sources, in addition to maintaining repositories of preference information.

The infrastructure is organized into loosely coupled layers as shown in Figure 7. The context gathering layer acquires context information from sensors and processes this information, using interpreters and aggregators, to bridge the gap between the raw sensor output and the level of abstraction and frequency of updates required by applications. An event notification scheme is used to achieve loose coupling between the sensing and processing components and the components of the layer above. This minimizes the problems associated with component failures, disconnections and evolution of the sensing infrastructure.

DOI: 10.1002/spe

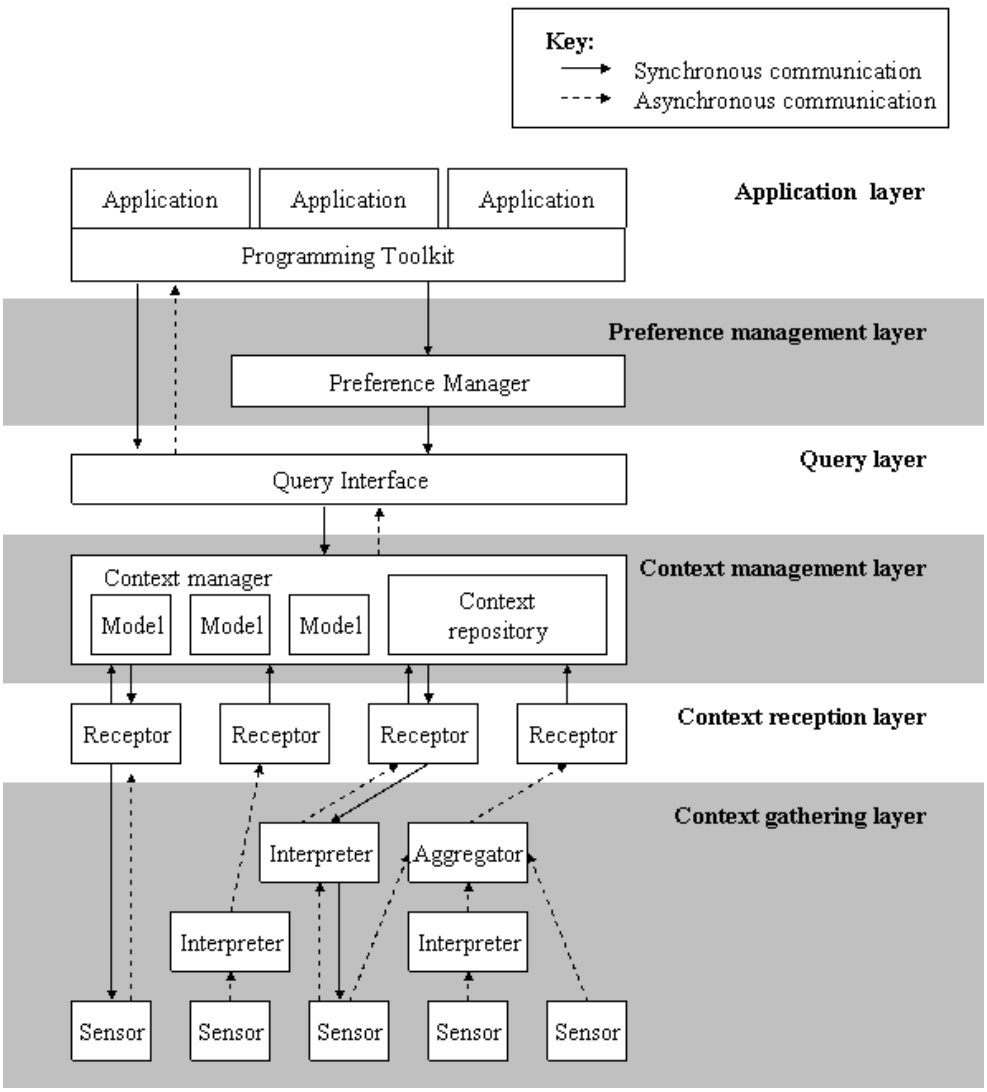

Figure 7. Software infrastructure for context-aware computing.

DOI: 10.1002/spe

The context reception layer provides a bidirectional mapping between the context gathering and management layers. That is, it translates inputs from the former into the fact-based representation of the latter, and routes queries from the latter to the appropriate components of the former. Each type of sensed context information is represented by its own receptor, which performs management functions specific to the type. For example, a location information receptor would be responsible for merging inputs from heterogeneous location sensors (such as Global Positioning System (GPS) receivers and radio frequency identification (RFID) readers), and resolving conflicts using appropriate algorithms [30].

The context management layer is responsible for maintaining a set of context models (expressed in terms of fact types, metadata and constraints, as described in Section 4.4) and instantiations of these. Context information is pooled into a shared repository of facts, and then mapped into the application-specific models. The context management system accepts information from a heterogeneous set of context providers, including human users, applications and context receptors, through both graphical and programmatic interfaces.

The query layer provides applications and the preference management layer with a convenient interface with which to query the context management infrastructure. Queries can use either of our context modelling abstractions; that is, they can be fact-based or situation-based. Two styles of query are supported: standard, synchronous queries, as well as an asynchronous, subscription-based scheme in which the client receives notifications of context changes that match its subscription. The query layer also implements a simple transaction model that allows a series of synchronous queries to be performed against a consistent (i.e. static) set of context information, regardless of the frequent updates coming from sensors and other context providers. This feature is useful for applications, but is especially important for meaningful preference evaluation.

The preference management layer is responsible for storing repositories of preferences and evaluating preferences on behalf of the context toolkit using the services of the query layer. Like the query layer, this layer provides transaction support so that several preference evaluations can be executed atomically against an unchanging set of context information.

Finally, the programming toolkit sits between applications and the preference management and query layers. In addition to providing a variety of methods for preference-based branching, as described in the previous section, it supplies hooks into the query layer that applications can easily use to obtain context information.

7. CASE STUDY: CONTEXT-AWARE COMMUNICATION

We developed a context-aware communication tool as a testbed for experimenting with our preference and programming models. In the following sections, we introduce the application and provide a discussion of our experiences and lessons learned from the case study.

7.1. Overview

Context-aware communication serves as an interesting application domain for our research as it offers many opportunities for customization using diverse and frequently conflicting preferences. Our case study considers a communication tool which assists users in selecting appropriate channels for their

DOI: 10.1002/spe

interactions with other people based on the combined context and preferences of the participants. The tool can support arbitrary types of communication channel, including telephone, e-mail, text messaging and videoconferencing channels. It also provides a history of interactions for the user, arranged logically into threads, which we term dialogues.

To initiate a new interaction, the user invokes a personal communication agent (CA)[‖]. The CA, in cooperation with the CAs of the other participants, examines the current context and preferences of all parties in order to recommend an appropriate channel. The main types of context information that are considered are the current activities of the participants, the priority of the interaction, the nature of the relationships between the participants (e.g. professional or private), and the availability of computing and communication devices.

The advantages of our context-aware communication tool over traditional forms of communication are threefold. First, disruption is minimized not only by taking user activity into account when selecting channels, but also by allowing users to specify through their preferences which forms of interruption are acceptable under which circumstances. For instance, users can choose to only admit urgent calls during meetings. Second, the communication tool helps to prevent missed calls by relying on context information to avoid recommending channels for which the user lacks the required communication devices. Finally, the problem of irrelevant communications is addressed by using an expiry scheme to discard messages that have exceeded their useful lifetimes (such as outdated seminar announcements) from the history of interactions, and by enabling users to direct unsolicited communications so that they cause minimal annoyance (e.g. to an infrequently checked e-mail address)[**].

7.2. Developing the communication prototype

We developed a prototype of the CA in Java using our programming toolkit. This involved the following steps, performed in roughly sequential order:

1. identification/specification of core functionality;
2. identification of choice points in the application in which context and preferences can be exploited;
3. analysis and modelling of context information required at the choice points;
4. analysis and modelling of default preferences and sample user preferences;
5. application design;
6. implementation using the programming toolkit;
7. configuration of the software infrastructure; and
8. testing of the application and refinement of the preference sets.

[‖]We use the term agent loosely here, and do not imply that our application is (or should be) aligned with any intelligent agent platform or multiagent decision-making approach. The class of decision problem we are concerned with in this paper is quite different to the one that is usually addressed by multiagent decision-making and negotiation protocols. Multiagent protocols typically focus on reaching consensus in the face of disagreement between agents, and do not consider contextual information. Although there are superficial similarities between the negotiation protocol we describe in Section 7.2 and multiagent negotiation, these are coincidental, and are not present in other context-aware applications that we have considered, such as those discussed in Section 8.

[**]This solution is only partial, however, as we cannot (and, indeed, probably should not) force the initiator of an interaction to use the recommended channel.

DOI: 10.1002/spe

In the first step, we focused on the user interface design, experimenting with mock-ups of the key user interface components. In the second phase, we explored the protocol by which the CAs negotiate communication channels, which currently represents the sole use of context and preference information by the application. We adopted a distributed, privacy-preserving protocol in which the context and preference information belonging to each user is visible exclusively to the user's own CA. The protocol operates as follows.

1. When creating a new dialogue, the user specifies a dialogue type (query, discussion or announcement), a subject and a list of participants (Figure 8). The dialogue type and number of participants later become important parameters in the choice of communication channel, and appear as variables in user preferences (this allows users to express very specific requirements: for example, that they prefer using e-mail for multiparty announcements and the telephone for two-person discussions).
2. A participant in the dialogue can initiate a new interaction at any time by editing the original participant list, if desired, and specifying a priority and an optional expiry time (Figure 9). Again, these attributes can appear as variables in user preferences. We refer to the user who performs this step as the initiator of the interaction.
3. The initiator's CA (ICA) carries out a negotiation procedure with the CAs of each of the other participants (PCAs) as follows. The ICA sends a channel request to each PCA, indicating the identity of the initiator, the type of dialogue, the priority of the interaction, and n, the desired number of channel proposals. Each PCA responds with a list of up to n channels that are suitable for the interaction, given the participant's current context and preferences.
4. The ICA merges the proposals to derive a list of channels suitable for all parties, then filters and sorts these according to the context and preferences of the initiator.
5. The ICA presents the highest scoring proposals from step 4 to the initiator, who selects one of the channels or cancels the interaction (Figures 10 and 11).
6. Finally, the ICA sends a request to the PCAs to establish the interaction using the channel selected at step 5.

In some cases no channels remain by step 5. When this occurs, the ICA has two options: the parameter n can be increased and a second negotiation attempted, or the protocol can be terminated and the choice of a further course of action delegated to the user.

The tasks of modelling the context and preference information exploited by this negotiation protocol (steps 3 and 4 in the software engineering process shown earlier in this section) proved to be fairly interdependent, and therefore were performed in an iterative fashion. First, a context model covering common factors in channel selection (user location and proximity to computing and communication devices, histories of user activity, etc.) was constructed. This step involved identifying a set of basic fact types and potential sources for the corresponding facts (sensors, user profiles, and so on), and then describing abstract situations, suitable for use in defining preferences in terms of these fact types. Next, we constructed two sets of sample preferences for the application: one suitable for use by the PCA at step 3 in the generation of a set of proposed channels, and another for use by the ICA at step 4 to filter and rank the proposed channels before presenting them to the initiator. Further types of relevant context information identified at this stage were integrated into the context model as required. Selected subsets of the resulting context model and preference sets were already presented in Section 4, and further details can be found in one of our earlier papers [16].

DOI: 10.1002/spe

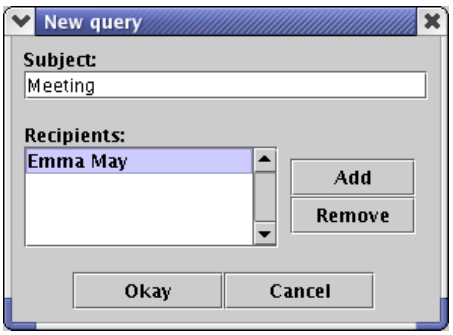

Figure 8. New query dialogue.

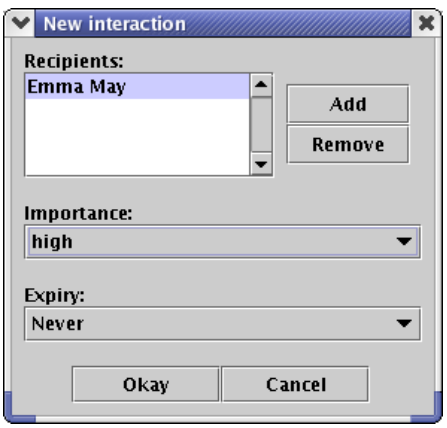

Figure 9. New interaction dialogue.

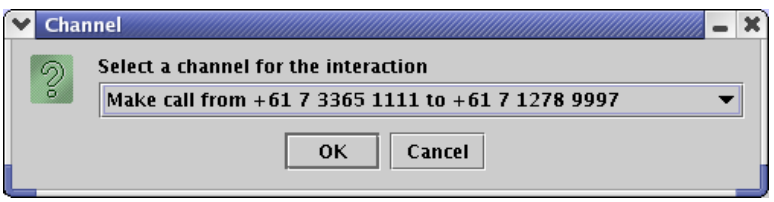

Figure 10. Channel selection dialogue.

DOI: 10.1002/spe

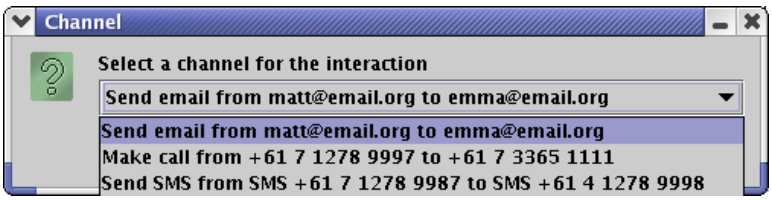

Figure 11. Channel selection dialogue, showing a drop-down list of options.

Following these steps, we designed the class structure of the CA, then implemented the agent using our programming toolkit. As the programming model is very successful in separating the details of the context model and preferences from the application itself, these tasks were carried out in a very similar manner as for traditional applications and we will not discuss them further.

Prior to testing the application, we inserted the details of our context model (including fact types, metadata and constraints) into the context management system, and placed the default and sample user preferences into the preference management system. These tasks were performed using fairly primitive mechanisms (i.e. through text files and direct manipulation of the context databases), but in the future we envisage the use of graphical tools that support editing of context models and preferences.

Finally, we populated the context model with the sets of sample context information required for our test cases and performed an iterative process of testing and preference refinement. As we discuss in the following section, we discovered that the preference sets do not always yield the expected results and that, in these cases, a process of experimentation and fine-tuning of the preference sets is required.

7.3. Results

The case study had several useful outcomes. In particular, it provided early validation for the success of our programming model in achieving the types of flexibility that we described in Section 1. As further validation, we are now building other applications, which we describe briefly in Section 8. In addition, and perhaps more importantly, it helped us to discover potential problems and pitfalls associated with the preference model, and to make improvements to the programming toolkit to address these. The following sections provide a discussion of these results.

7.3.1. Evolving the context

We motivated early in the paper the importance of loose coupling between the application and the underlying context model and context gathering infrastructure. This is important because the available sensing components (and, by implication, the available types of context information) in pervasive computing environments are prone to change. The case study served as confirmation of the value of this approach.

By using the programming model and toolkit, we succeeded in implementing the agent so that it exploits rich types of contextual information in the channel selection process, yet has minimal

DOI: 10.1002/spe

dependencies on the context model. An explicit query on the context model is made in only one place in the application; this occurs at the beginning of the channel negotiation algorithm, when the agent retrieves a list of all communication channels belonging to the user. This query consults only a single fact type; references to all other fact types appear only in the preferences, and not in the application itself. This implies that we can modify almost all of the context model (the exception of course being the single fact type that is explicitly queried), without changes to the source code. In addition, we can add new fact types without any impact at all on either the code or the previously defined preferences.

We successfully exploited this feature in recent extensions to the prototype in which we incorporated additional context information derived from infrared location beacons and device status monitors. We also found it useful in dealing with unexpected problems that we encountered late in the software engineering process in relation to the accuracy of some types of context information. For example, we discovered that the agent performed better using a combination of historical and current location information, rather than just the current location alone, and were able to easily incorporate the historical information with minimal effort and no changes to the agent itself.

7.3.2. Problems with incompleteness and unexpected behaviours

The case study also uncovered some problems with the way in which we initially used the preference and programming models, and highlighted some challenges in the construction of preference sets. The first issue was related to incomplete preference information. The preference model is designed so that each preference has a narrow domain of applicability, indicated by its context description. This means that, when rating a choice in a given context, the majority of preferences are usually effectively ignored (i.e. they produce indifferent scores). With many preference sets, and particularly those that are reasonably small, the contexts associated with the preferences incompletely cover the set of all possible contexts, implying that there are contexts in which *all* of the preferences yield indifferent scores. In the initial implementation of the CA, we did not adequately anticipate this possibility. When selecting channels, we looked exclusively for choices that received obligation scores or high numerical scores. This implied that channels that received only indifferent scores were ignored, and the agent failed to recommend any channel in some contexts even though several channels were actually available.

This bias was present not only in the application but also in the branching toolkit. We have since made improvements to the toolkit that enable better handling of indifferent scores and other special scores (most notably, the error score). A subsequent version of communication tool that we implemented using the improved toolkit no longer suffers from problems as a result of incomplete preferences.

The second challenge was related to unexpected results that we observed in the recommendations produced by CAs as a result of imperfect context information and the difficulties involved in writing complex preference sets. Although we analyzed potential sources of error, ambiguity and incompleteness in the context information used by the agent during the design of the context model, additional problems emerged only when we began testing the application. These required us to redefine some of the situations (and occasionally also modify the preferences), but fortunately did not require us to modify the application itself, thanks to the loose coupling we described in the previous section.

DOI: 10.1002/spe

Orthogonal to the problem of imperfect context information, we identified challenges in writing, and especially combining, preferences. Specifically, we observed that the behaviour resulting from preference sets is often not exactly as anticipated, especially when the sets are reasonably large or complex; this is a phenomenon we refer to as 'preference surprise'. The preference surprise phenomenon highlights the importance of experimentation and fine-tuning; for example, relative weights of preferences often require adjustment until the expected behaviour emerges. We note that this type of problem is not unique to our preference and programming models, but is usual whenever complex choices over many variables are required. Typically, however, it would require the application developer to modify decision logic embedded in the source code until the behaviour becomes satisfactory, and this represents a larger and more error-prone task than that of simply editing preferences.

The challenges of constructing appropriate preference sets supports our belief that the preference model should never be directly exposed to the user. Instead we advocate other approaches, such as offering predefined preference profiles or providing a controlled set of configuration options, as discussed in Section 4.3.

8. CURRENT AND FUTURE WORK

The communication tool we have presented in this paper represents our first attempt to build an application using our preference and programming models. Since completing this initial case study, we have begun implementing two additional applications. The first is a substantially modified version of the communication application which improves upon the design presented in Section 7.1 by incorporating support for call setup and adaptation using the Session Initiation Protocol (SIP) [31][††]. It also adds additional location and device status sensors to improve the channel selection process.

The second application is a vertical handover prototype that dynamically switches between network interfaces used for streaming a video to the user based on network characteristics, location changes and other context information. The currently supported network types are Ethernet, Wi-Fi, GPRS and Bluetooth. This work builds on an earlier prototype [32] that was developed without any form of infrastructural support for the management of context and user preference information. By using our programming model and software infrastructure, we have been able to substantially simplify the implementation and increase its flexibility. Importantly, we have also been able to show that we can reuse substantial portions of the context model and context gathering infrastructure used for the communication application.

As part of our future work on these and other applications, we intend to explore the use of automated preference elicitation techniques, in conjunction with the current approach in which users must explicitly set their own preferences or else adopt the supplied default preferences. We are currently considering two approaches that both rely on feedback mechanisms.

The first approach involves the use of a weighting scheme to promote or reduce the importance of individual preferences participating in a composite preference according to user feedback. In its simplest form, this scheme involves using negative user feedback (provided after an inappropriate choice) to decrease the weighting of preferences that assigned a high score to the choice, or to increase the weighting of preferences that assigned a low score.

[††]The earlier version of the application provided recommendations about which channels to use, but deferred the task of actually establishing a call to the user.

DOI: 10.1002/spe

The second approach generates entirely new preferences in response to feedback. For example, if a user rejects a choice, a new preference, having a context corresponding to that in which the inappropriate choice was originally made and a low score (or a veto) is incorporated into the user's preference set. This approach is more flexible than the approach based on weighting, as the latter is only able to adjust the importance of the existing preferences, which may not cover all user requirements. This benefit is offset, however, by the fact that learning can be very slow. The change that results from each instance of user feedback is generally only observable in the narrow context for which the feedback is given, meaning that preferences that are applicable in broad contexts may require many rounds of feedback before they are derived in reasonably general forms. However, the two approaches are not mutually exclusive, and can be employed together in a hybrid solution that combines their respective advantages.

9. SUMMARY

In this paper, we motivated the need for new approaches to building context-aware applications that yield the flexibility required for graceful evolution and customization by users. To address this challenge, we introduced novel preference and programming models that support context-based choices in arbitrary types of application. The power of these models comes from the fact that they allow the behaviour of applications at choice points to be manipulated simply by editing preferences. We used a case study, involving a context-aware communication tool, as a practical demonstration of the utility of this feature. We also outlined further self-adapting applications that we are currently developing using our models.

Our experiences with using our novel approach for building context-aware software suggest that it can be applied to a wide variety of applications to substantially reduce the complexity of exploiting diverse types of context information for self-adaptation. We are optimistic that this feature of our approach will help to create possibilities for exploring new application designs and domains, so that headway can be made with other problems associated with context-aware software, including the usability and design challenges discussed in Section 1.

ACKNOWLEDGEMENTS

We gratefully acknowledge significant contributions by Ted McFadden and Peter Mascaro to the design and implementation of the most recent versions of the communication and vertical handover prototypes, described in Section 8. We also wish to thank Sasitharan Balasubramaniam, who provided the original version of the vertical handover prototype and kindly assisted with the development of the subsequent version.

REFERENCES

1. Cheverst K, Davies N, Mitchell K, Friday A. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. *Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiCom)*, Boston, MA, 2000. ACM Press: New York, 2000; 20–31.
2. Abowd GD, Atkeson CG, Hong J, Long S, Kooper R, Pinkerton M. Cyberguide: A mobile context-aware tour guide. *Wireless Networks* 1997; **3**:421–433.
3. Schilit BN, Hilbert DM, Trevor J. Context-aware communication. *IEEE Wireless Communications* 2002; **9**:46–54.
4. Dey AK, Salber D, Abowd GD. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction* 2001; **16**:97–166.
5. Chen G, Kotz D. Context aggregation and dissemination in ubiquitous computing systems. *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, Callicoon, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002.

DOI: 10.1002/spe

6. Barkhuus L, Dey AK. Is context-aware computing taking control away from the user? Three levels of interactivity examined. *Proceedings of the 5th International Conference on Ubiquitous Computing*, Seattle, WA, 2003 (*Lecture Notes in Computer Science*, vol. 2864). Springer: Berlin, 2003.
7. Cheverst K, Davies N, Mitchell K, Friday A. The role of connectivity in supporting context-sensitive applications. *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing* (*Lecture Notes in Computer Science*, vol. 1707). Springer: Berlin, 1999; 193–207.
8. Jiang X, Landay JA. Modeling privacy control in context-aware systems. *IEEE Pervasive Computing* 2002; **1**:59–63.
9. Brown PJ, Bovey JD, Chen X. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications* 1997; **4**:58–64.
10. Brown PJ. Triggering information by context. *Personal Technologies* 1998; **2**:1–9.
11. Ryan N, Pascoe J, Morse D. Fieldnote: A handheld information system for the field. *Proceedings of the 1st International Workshop on TeleGeoProcessing*, Lyon, 1999; 156–163.
12. Brown PJ. The stick-e document: A framework for creating context-aware applications. *Proceedings of the Conference on Electronic Publishing*, Palo Alto, CA 1996; 259–272.
13. Yau SS, Karim F, Wang Y, Wang B, Gupta SKS. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing* (*Special Issue on Context-Aware Computing*) 2002; **1**:33–40.
14. Coutaz J, Rey G. Recovering foundations for a theory of contextors. *Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces (CADUI)*, Valenciennes, 2002. Kluwer Academic: Boston, MA, 2002.
15. Henricksen K, Indulska J, Rakotonirainy A. Modeling context information in pervasive computing systems. *Proceedings of the 1st International Conference on Pervasive Computing (Pervasive)* (*Lecture Notes in Computer Science*, vol. 2414). Springer: Berlin, 2002: 167–180.
16. Henricksen K, Indulska J. A software engineering framework for context-aware pervasive computing. *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE Computer Society Press: Los Alamitos, CA, 2004; 77–86.
17. Schmandt C, Marmasse N, Marti S, Sawhney N, Wheeler S. Everywhere messaging. *IBM Systems Journal* 2000; **39**:660–677.
18. Byun HE, Cheverst K. Exploiting user models and context-awareness to support personal daily activities. *Proceedings of the UM2001 Workshop on User Modeling for Context-Aware Applications (UM2001)*, Sonthofen, 2001.
19. Byun HE, Cheverst K. Harnessing context to support proactive behaviours. *Proceedings of the ECAI2002 Workshop on AI in Mobile Systems*, Lyon, 2002.
20. Jameson A. Modeling both the context and the user. *Personal and Ubiquitous Computing* 2001; **5**:29–33.
21. Klyne G, Reynolds F, Woodrow C, Ohto H, Hjelm J, Butler MH, Tran L. Composite capability/preference profiles (CC/PP): Structure and vocabularies 1.0. *W3C Proposed Recommendation*, 2003.
22. Indulska J, Robinson R, Rakotonirainy A, Henricksen K. Experiences in using CC/PP in context-aware systems. *Proceedings of the 4th International Conference on Mobile Data Management (MDM)* (*Lecture Notes in Computer Science*, vol. 2574). Springer: Berlin, 2003: 247–261.
23. Chomicki J. Querying with intrinsic preferences. *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)* (*Lecture Notes in Computer Science*, vol. 2287). Springer: Berlin, 2002; 34–51.
24. Fishburn P. Preference structures and their numerical representations. *Theoretical Computer Science* 1999; **217**:359–383.
25. Agrawal R, Wimmers EL. A framework for expressing and combining preferences. *Proceedings of the ACM SIGMOD Conference on Management of Data*, Dallas, TX, 2000. ACM Press: New York, 2000; 297–306.
26. Cohen WW, Schapire RE, Singer Y. Learning to order things. *Advances in Neural Information Processing Systems* 1998; **10**:451–457.
27. Çetintemel U, Franklin MJ, Lee Giles C. Self-adaptive user profiles for large-scale data delivery. *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, San Diego, CA, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2000; 622–633.
28. Brown PJ, Jones GJF. Context-aware retrieval: Exploring a new environment for information retrieval and information filtering. *Personal and Ubiquitous Computing* 2001; **5**:253–263.
29. Henricksen K. A framework for context-aware pervasive computing applications. *PhD Thesis*, School of Information Technology and Electrical Engineering, The University of Queensland, 2003.
30. Indulska J, McFadden T, Kind M, Henricksen K. Scalable location management for context-aware systems. *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)* (*Lecture Notes in Computer Science*, vol. 2893). Springer: Berlin, 2003; 224–235.
31. Rosenberg J, Schulzrinne H, Camarillo G, Johnston A, Peterson J, Sparks R, Handley M, Schooler E. SIP: Session Initiation Protocol. *RFC 3261*, IETF, 2002.
32. Balasubramaniam S, Indulska J. Vertical handovers as adaptation methods in pervasive systems. *Proceedings of the IEEE International Conference on Networks*, Sydney, 2003. Bloxham & Chambers: Sydney, 2003; 705–710.

DOI: 10.1002/spe