Personalising Context-Aware Applications*

Karen Henricksen[1] and Jadwiga Indulska[2]

[1] CRC for Enterprise Distributed Systems Technology (DSTC)
karen@itee.uq.edu.au
[2] School of Information Technology and Electrical Engineering,
The University of Queensland
jaga@itee.uq.edu.au

Abstract. The immaturity of the field of context-aware computing means that little is known about how to incorporate appropriate personalisation mechanisms into context-aware applications. One of the main challenges is how to elicit and represent complex, context-dependent requirements, and then use the resulting representations within context-aware applications to support decision-making processes. In this paper, we characterise several approaches to personalisation of context-aware applications and introduce our research on personalisation using a novel preference model.

1 Introduction

Context-awareness has emerged as a popular design approach for building adaptive applications for mobile and pervasive computing environments. Context-aware applications rely on information about the context of use - such as the user's current location and activity - to provide seamless operation in the face of mobility and intelligent support for users' evolving requirements.

As users of context-aware applications can differ greatly in terms of their requirements and expectations about how their applications should behave, personalisation mechanisms are required. Unfortunately, personalisation of context-aware applications is substantially more challenging than personalisation of traditional desktop applications. Because the actions of context-aware applications are partially determined by the context, user preferences must likewise be predicated on context. The set of distinct contexts recognised by a context-aware application may be large, implying that the set of user preferences might also be large and complex. A further problem related to personalisation is the need to provide users with a clear mental model and appropriate feedback mechanisms that allow them to understand the links between application behaviours and their specified preferences. These are essential in order to prevent user frustration at apparently erratic behaviour, and to facilitate trouble-shooting.

* The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science, and Training).

Owing to the immaturity of the field of context-aware computing, very little research has addressed personalisation. Rather, the main focus has been on techniques for acquiring, interpreting and managing context information from sensors. In this paper, we survey the limited work that has been done in the area and then report on our own research, which has investigated personalisation based on a novel preference model.

The structure of the paper is as follows. Section 2 provides an analysis of three approaches that have been pursued for personalisation of context-aware applications, while Sections 3 and 4 introduce the preference model used in our work. Section 5 briefly touches on implementation issues, describing a software infrastructure and programming toolkit we developed to facilitate the use of our preference model by context-aware applications. Finally, Section 6 highlights some user interface design issues and Section 7 discusses topics for future work.

2 Personalisation Approaches

The existing approaches to personalising context-aware applications can be classified as follows:

- *End-user programming approach.* This approach offers the most radical form of personalisation: rather than applications being developed by software engineers with hooks for customisation by users, end-user programming techniques place the task of constructing applications into the hands of users. Several styles of end-user programming exist, including programming by demonstration [1], in which users train a system to carry out desired actions by manually demonstrating the actions, and programming by specification [2], in which users provide high-level descriptions of desired actions.
- *User modelling/machine learning approach.* This approach removes responsibility for personalisation from the user, instead using machine learning techniques to automatically derive user requirements from historical data. These requirements can be represented in the form of user models suitable for use by applications as a basis for adaptive and pro-active behaviours [3].
- *Preference-based approach.* This approach is the closest to traditional personalisation approaches for desktop applications. It typically relies on user interfaces or configuration files through which users can manipulate settings or rules that control the way applications react to context. Other than our own work, we are not aware of any research that addresses the general problem of personalising context-aware applications with context-dependent preferences; however, several applications that provide custom-designed preference mechanisms have been developed (e.g., [4]).

Each approach has shortcomings which limit its applicability to certain application domains. The main problem of end-user programming techniques is that they are generally suitable only when the application behaviours that users need to specify are reasonably simple. For complex tasks, users experience difficulties in demonstrating or specifying their requirements. As a result, the most common

scenarios presented in the literature on end-user programming concentrate on simple tasks, such as loading presentation files in advance of a meeting [1]. A further problem is that most of the end-user programming solutions are primarily concerned with supporting the initial programming task, and it is unclear how well they can support evolution as user requirements or the environment change. Finally, a large initial investment is expected of users to either train the system or specify the required behaviours.

The second approach, based on user modelling and machine learning, is more appropriate than end-user programming for complex applications and does not require a period of explicit training or set-up by the user. However, mistakes are nearly always made during the learning process, causing frustration to the user. The user can provide feedback to help prevent similar mistakes in the future; however, many rounds of feedback may be required before the desired behaviour emerges. Users may prefer to avoid the frustration of repeatedly providing feedback by explicitly specifying some or all of their requirements; however, manual customisation is unfortunately not supported in this approach.

The preference-based approach does not suffer from this problem, as it allows users to explicitly specify requirements at any time. It can also be used in conjunction with automated preference learning mechanisms, so as to reduce the burden on users to specify preference information that is complete and up to date. Finally, unlike end-user programming, the preference-based approach is appropriate even for complex applications. This means that it is arguably the most promising and widely applicable of the three approaches.

3 A Preference Model for Context-Aware Applications

The remainder of the paper focuses on our preference-based personalisation approach, which is based on a novel preference model. This section introduces the model, while preference examples are deferred until following section.

When starting our work on personalisation, we surveyed preference modelling approaches from diverse fields such as decision theory and document retrieval, with the aim of identifying a preference model that could be used as a basis for personalisation of context-aware applications. This survey can be found in [5]. However, none of the approaches that we examined was able to represent context-dependent preferences. Accordingly, we developed our own preference model designed to address this limitation. This model supports user-customisable decision-making by context-aware applications, as shown in Fig. 1.

In this decision-making process, user preferences are evaluated against a set of context information, candidate choices (which may be associated with one or more corresponding actions) and application state variables, to yield an assignment of ratings to the candidate choices. The user preferences may reflect the requirements of one or multiple users. Arbitrary kinds of choices can be supported: for example, the choices may be documents or search terms in the case of an information retrieval application, or email folders in the case of an email filtering tool.

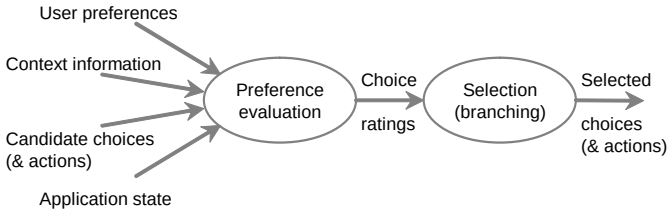

Fig. 1. Context- and preference-based decision process for context-aware applications.

After using the preferences to rate the candidate choices, the context-aware application selects zero or more of the choices, and carries out a set of corresponding actions (for example, displaying a set of chosen documents to the user or filtering an email to a selected folder). We refer to this step as *branching*.

Each user preference takes the form of a scope and a scoring expression. The scope specifies the context and choices to which the preference applies using a special form of logical predicate, which we term a *situation*. Our notation for defining situations is described in an earlier paper [6], in which we outlined our approach to context modelling, and therefore will not be described in detail here. The scoring expression produces a rating that indicates the suitability of the choices that match the scope within the given context. This rating is one of the following:

- a numerical value in the range [0,1], where increasing scores represent increasing desirability;
- *prohibit*, indicating that a choice must not be selected in a given context;
- *oblige*, indicating that a choice must be selected in a given context;
- *indifferent*, indicating an absence of preference; or
- *undefined*, signalling an error condition (e.g., an attempt to combine *prohibit* and *oblige* scores).

Preferences may be either *simple* preferences, which express atomic user requirements, or *composite* preferences, which specify how other preferences are combined to produce appropriate aggregate ratings. Related preferences can also be dynamically grouped into preference sets. In the following section, we present some example preferences and preference sets.

4 Preference Examples

Although user preferences may be generic enough to be applied to many context-aware applications, most preferences are specific to a particular application (and type of choice within that application). Here, for illustrative purposes, we focus on the specification of preferences for a context-aware email client. This application uses context information to enhance a set of standard email management

l7_unread =	**when** *outOfOffice*(*me*) and *contains*(*to*, "level7@dstc.edu.au") and *equals*(*status*, "new") and *equals*(*folder*, "inbox-secondary") **rate** *high*
l7_read =	**when** *equals*(*status*, "read") and *contains*(*to*, "level7@dstc.edu.au") and *equals*(*folder*, "level7") **rate** *high*
personal =	**when** (*familyMembers*(*me*, *sender*) or *friends*(*me*, *sender*)) and *equals*(*folder*, "personal") **rate** *medium_high*
important =	**when** *equals*(*priority*, "highest") and *equals*(*status*, "new") and *equals*(*folder*, "inbox") **rate** *oblige*
filtering_set =	{*l7-unread*, *l7-read*, *personal*, *important*}
filtering =	**when** true **rate** *average*(*filtering_set*)

Table 1. Example preferences for email forwarding.

features, such as automatic forwarding and filtering of messages, alerting for important messages, and auto-replying to messages. The use of context-awareness allows the email client to behave more pro-actively than would otherwise be possible. For instance, the client can automatically produce auto-reply messages when the user is on vacation (without any prior set-up) and perform message filtering on arbitrary types of context, not only on message headers and content.

In Table 1, we show some sample user preferences related to filtering. The goal of filtering is to assist with organising email into folders. These preferences assume that the user's email folders include the following: inbox, inbox-secondary, personal and level7. Four simple preferences (*l7_unread*, *l7_read*, *personal* and *important*), one composite preference (*filtering*), and one preference set (*filtering_set*) are shown.

The scope of each preference follows the "when" keyword, while the scoring expression is preceded by the "rate" keyword. The first preference states that filtering to the secondary inbox is highly preferred when:

- the message is addressed to the level7 mailing list ("level7@dstc.edu.au");
- the user is currently out of the office (which possibly implies that the message is irrelevant, as much of the list traffic is solely of interest to the current occupants of floor level 7); and
- the message is new (i.e., unread).

The second preference states that already read messages addressed to the same mailing list should be filed in the level7 folder, with high preference. The

personal preference states that messages from family members or friends should be filed to the personal folder, with medium-high preference, while the *important* preference states that unread messages that have the highest priority level must always remain in the user's inbox. The preference ratings *medium_high* and *high* are mapped to numerical values as defined by the application developer, in order to allow aggregation of scores. Note that, although the preferences are somewhat similar in appearance to the user-defined message filters that are already supported by email clients such as Mozilla Thunderbird and Microsoft Outlook, two of the preferences (*l7_unread* and *personal*) refer to external context definitions (i.e., the *outOfOffice*, *familyMembers* and *friends* situations) that cannot be included in standard email filters.

To combine the requirements expressed by these four simple preferences to support decision making about how to filter messages, the preferences are first grouped into a preference set (*filtering_set*). The *filtering* composite preference then defines the overall rating for a given folder as the average of the ratings produced by the preferences in this set. Here, averaging is performed according to the following simple algorithm:

1. if any preference produces the *undefined* score, the result of averaging is the *undefined* score; else
2. if one or more preferences produces the *oblige* score *and* one or more preference produces the *prohibit* score, then the result is the *undefined* score; else
3. if one or more preferences produces the *oblige* score, then the result is the *oblige* score; else
4. if one or more preferences produces the *prohibit* score, then the result is the *prohibit* score; else
5. if one or more preferences produces a numerical score, then the result is the average of all numerical scores; else
6. if all preferences produce *indifferent* scores (which occurs by default when the preference scopes do not hold), then the result is the *indifferent* score.

To illustrate, we consider the scenario in which the email client filters an already read message that was sent by a friend of the user to the level7 mailing list. The ratings produced by the preferences defined in Table 1 are shown in Table 2. Only the *l7_read* and *personal* preferences are relevant to this example. The remaining preferences produce indifferent ratings for all four email folders. As the *l7_read* preference produces a higher preference rating, this preference takes precedence. Therefore, the message in this example would be filtered to the level7 folder, rather than the personal folder.

5 Infrastructural Support for Personalisation

To assist with implementing context-aware applications that support personalisation using our preference model, we have developed a layered software infrastructure that supports:

Preference	inbox	inbox-secondary	personal	level7
l7_unread	*indifferent*	*indifferent*	*indifferent*	*indifferent*
l7_read	*indifferent*	*indifferent*	*indifferent*	*high*
personal	*indifferent*	*indifferent*	*medium-high*	*indifferent*
important	*indifferent*	*indifferent*	*indifferent*	*indifferent*
filtering	*indifferent*	*indifferent*	*medium-high*	*high*

Table 2. Preference ratings for an already read message sent by a friend to the level7 mailing list.

- integration, management and querying of context information from various sources, including sensors, context-aware applications and human users (*context management layer*);
- management and evaluation of user preference information (*preference management layer*); and
- decision making and branching at the application layer, using the services of the context and preference management layers (*programming toolkit*).

The context and preference management layers are implemented in Java, using relational databases for information storage and management. However, they accept requests via several different communication protocols (XML/HTTP, Java RMI and Elvin [7]), and therefore can be used in conjunction with a variety of platforms and programming languages. The programming toolkit, which provides various helper classes for formulating decision problems and selecting appropriate actions based on the ratings produced by the preference management layer, can currently only be used by Java applications, but could be ported to other languages in the future. Further information about the software infrastructure can be found in some of our earlier papers [6, 8].

6 User Interface Design

The preference notation that we described in Sections 3 and 4 is used internally by our programming toolkit and preference management layer, but would rarely be exposed directly to users. In this section, we discuss some issues related to the design of user interfaces to support personalisation of context-aware applications. Appropriate user interface designs must necessarily be considered on a case-by-case basis; because of this, our discussion focuses on the design of a user interface for the email application we discussed in Section 4, as a case study. However, we also offer a set of general design guidelines in Section 6.2.

6.1 Personalisation Interfaces for Context-Aware Email

Email applications provide a useful starting point for thinking about user interface design issues, as most already support personalisation. Therefore, instead of thinking about how personalisation can be incorporated from scratch, it is

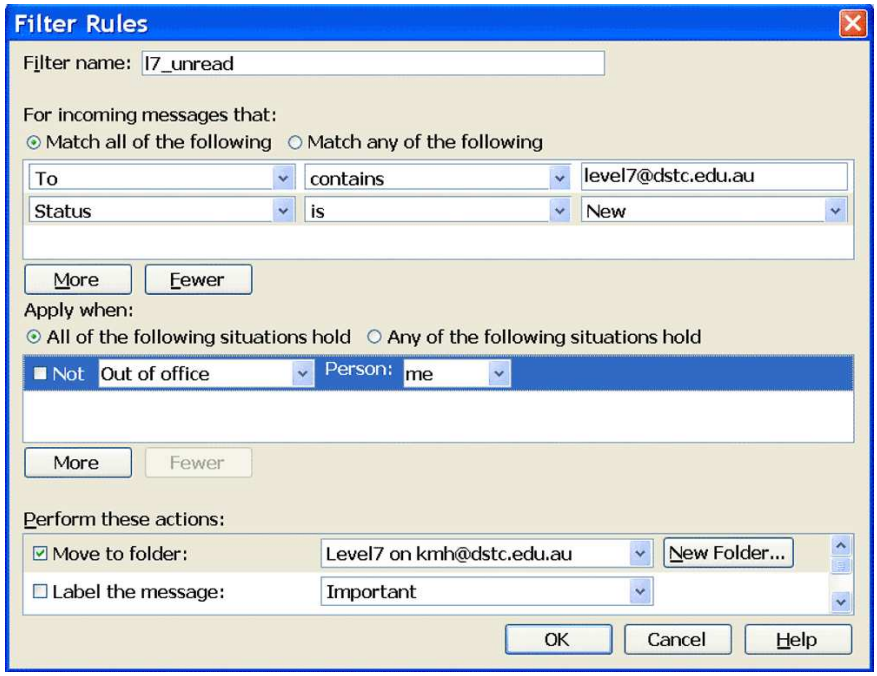

Fig. 2. An extension of the current Thunderbird filter rule interface to support context-dependent filters.

only necessary to think about how to extend the existing personalisation to support context-dependent user preferences. We have been working on a set of context-aware extensions for the Thunderbird email client[3].

Thunderbird supports many types of personalisation, but here we focus on filtering. Users can define new message filters, and enable/disable already created filters. Filters are specified as matching conditions, defined in terms of message headers, and corresponding actions (e.g., "move a message to a specified folder"). To support context-based filtering, only minor changes are needed to the existing user interface. In Fig. 2, we show a trivial extension of the current interface which allows the matching conditions to be augmented with relevant situations. Each filter can be mapped to a preference that conforms to the preference model described in Sections 3 and 4. Observe, for example, that the filter defined in Fig. 2 matches the scope of the *l7_unread* preference in Table 1. The rating assigned to this preference/filter is set on another screen (not shown), which lists all defined filters and provides controls for enabling/disabling filters.

The fact that our preference model provides such a close fit with the existing personalisation model used by Thunderbird - despite the fact that we did not have email in mind as a target application when designing the preference model -

[3] http://www.mozilla.org/products/thunderbird/

helps to validate the design of the model. As we show in this example, personalisation interfaces should be consistent with the general appearance and function of a context-aware application, rather than being closely tied to the preference model. In particular, a personalisation interface should always be more than a simple preference editor that expects the user to directly formulate preferences of the kind shown in Table 1.

6.2 General Design Guidelines

To date, we have built a number of personalisable context-aware applications using our preference model. These have included several communications applications [6, 9], a vertical handover application for managing the streaming of multimedia to mobile users [8], and virtual community applications to support independent living of the elderly [10]. As a result of our experiences with these applications, we offer the following general design guidelines:

- *Constrain the types of personalisation that can be performed by users.* That is, even when using a generic preference model such as the one we have presented, the full power of the model should not be exposed to users. There are two reasons for this: (i) users are likely to be overwhelmed and confused, and (ii) complex preference sets should be thoroughly tested before they are deployed to ensure that no unexpected behaviours emerge.
- *Integrate personalisation mechanisms into the everyday use of the application.* This helps to ensure that personalisation is a natural and visible part of the application, and increases the chance that users will use and understand the personalisation mechanisms. This design principle is somewhat similar to the one advocated by Lederer et al. [11] in relation to designing systems to support privacy.
- *Provide logging and feedback mechanisms.* These should let users (i) see how their preferences are linked to actions and (ii) override actions if necessary. Logging and feedback can help to prevent user frustration and assist users with correcting preference and/or context information when required.
- *Provide useful default behaviours.* That is, ensure that most people will be able to use the application reasonably well from first use, even without any personalisation. Some people will resist using personalisation mechanisms at all, no matter how visible and straightforward they are.

7 Future Work

This paper outlined our efforts to develop a preference model for personalisation of context-aware applications. As discussed in Section 6.2, we have used the model in conjunction with a variety of context-aware applications. Although our experiences with using the model have been positive, we have already identified some important refinements and extensions for future work. In particular, we have started designing some modifications to the preference model that should

improve both the usability of the model for application developers and the efficiency of preference evaluation. We have also begun working on techniques for automated preference learning based on user feedback. In the near future, we hope to extend our programming toolkit and preference management system to support these mechanisms. In the longer term, we plan to investigate extensions of the preference model to a broader set of decision problems relevant to context-aware applications. At present, our model is best suited to choices over a fixed (and reasonably small) set of alternatives; in the future, we plan to study decision problems that are both larger and more open-ended. Finally, appropriate user evaluation is crucial, not only for our preference model, but also for the other personalisation approaches discussed in Section 2.

References

1. Dey, A.K., Hamid, R., Beckmann, C., Li, I., Hsu, D.: a CAPpella: Programming by demonstration of context-aware applications. In: ACM Conference on Human Factors in Computing Systems (CHI), Vienna (2004)
2. Truong, K.N., Huang, E.M., Abowd, G.D.: CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In: 6th International Conference on Ubiquitous Computing (UbiComp). Volume 3205 of Lecture Notes in Computer Science., Springer (2004) 143–160
3. Byun, H.E., Cheverst, K.: Harnessing context to support proactive behaviours. In: ECAI2002 Workshop on AI in Mobile Systems, Lyon (2002)
4. Lei, H., Ranganathan, A.: Context-aware unified communication. In: 5th International Conference on Mobile Data Management (MDM), Berkeley (2004)
5. Henricksen, K.: A Framework for Context-Aware Pervasive Computing Applications. PhD thesis, School of Information Technology and Electrical Engineering, The University of Queensland (2003)
6. Henricksen, K., Indulska, J.: A software engineering framework for context-aware pervasive computing. In: 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE Computer Society (2004) 77–86
7. Segall, B., Arnold, D., Boot, J., Henderson, M., Phelps, T.: Content based routing with Elvin4. In: AUUG2K Conference, Canberra (2000)
8. Henricksen, K., Indulska, J., McFadden, T., Balasubramaniam, S.: Middleware for distributed context-aware systems. International Symposium on Distributed Objects and Applications (DOA) (to appear) (2005)
9. McFadden, T., Henricksen, K., Indulska, J., Mascaro, P.: Applying a disciplined approach to the development of a context-aware communication application. In: 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE Computer Society (2005) 300–306
10. Indulska, J., Henricksen, K., McFadden, T., Mascaro, P.: Towards a common context model for virtual community applications. In: 2nd International Conference on Smart Homes and Health Telematics (ICOST). Volume 14 of Assistive Technology Research Series., IOS Press (2004) 154–161
11. Lederer, S., Hong, J.I., Dey, A.K., Landay, J.A.: Personal privacy through understanding and action: five pitfalls for designers. Personal and Ubiquitous Computing **8** (2004) 440–454