Developing Context-Aware Pervasive Computing Applications: Models and Approach

Karen Henricksen

CRC for Enterprise Distributed Systems Technology (DSTC), Level 7, General Purpose South, The University of Queensland, QLD 4072 Australia

Jadwiga Indulska

School of Information Technology and Electrical Engineering, The University of Queensland, QLD 4072 Australia

Abstract

There is growing interest in the use of context-awareness as a technique for developing pervasive computing applications that are flexible, adaptable, and capable of acting autonomously on behalf of users. However, context-awareness introduces a variety of software engineering challenges. In this paper, we address these challenges by proposing a set of conceptual models designed to support the software engineering process, including context modelling techniques, a preference model for representing context-dependent requirements, and two programming models. We also present a software infrastructure and software engineering process that can be used in conjunction with our models. Finally, we discuss a case study that demonstrates the strengths of our models and software engineering approach with respect to a set of software quality metrics.

Key words: context-aware applications, software engineering, pervasive computing infrastructure, context modelling

⋆ 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).

Email addresses: `karen@itee.uq.edu.au` (Karen Henricksen), `jaga@itee.uq.edu.au` (Jadwiga Indulska).

1 Motivation

It is well known that pervasive computing introduces a set of design challenges that are not present in traditional desktop computing. In particular, it requires applications that are capable of operating in highly dynamic environments and placing minimal demands on user attention. Context-aware applications aim to meet these requirements by adapting to selected aspects of the context of use, such as the current location, time and user activities.

In recent years, a variety of prototypical context-aware applications have appeared, such as context-aware guides that present tourists with information tailored to their location [1], and capture tools that augment various types of media with contextual metadata describing the context in which it was recorded [2]. Further, efforts are ongoing to construct context-aware environments that are instrumented with sensors that enable tracking of the occupants and their activities. These environments vary in scale from single rooms, such as classrooms and meeting spaces [3,4], to smart homes that support independent living of the elderly or disabled [5].

Despite the recent flurry of interest, context-aware applications have not yet made the transition out of the laboratory and into everyday use. This is largely a result of high application development overheads, social barriers associated with privacy and usability, and an imperfect understanding of the truly compelling uses of context-awareness. This paper presents a software engineering approach that we have developed to address these three challenges: the first by simplifying design and implementation tasks associated with context-aware software, and the latter two by facilitating the types of rapid prototyping and experimentation that are required in order to overcome these obstacles. This approach is based around a set of novel conceptual foundations, including context modelling techniques, a preference abstraction, and a pair of complementary programming models. These are introduced in Sections 2 to 4, while Section 5 describes the integration of the models into a software infrastructure for pervasive systems. Section 6 outlines the process involved in developing a context-aware application using the models and infrastructure, and Section 7 presents a case study that demonstrates the value of our approach in terms of software quality metrics. Finally, Section 8 provides a summary and discussion of future work.

2 Context modelling techniques

Recent research in the field of context-awareness has predominantly adopted an infrastructure-centred approach; that is, it has assumed that the complexity of engineering context-aware applications can be substantially reduced solely through the use of infrastructure capable of gathering, managing and disseminating context information to applications that require it. In line with this approach, a variety of solutions that acquire and interpret context information from sensors, and manage repositories of information that support queries and notifications, have been proposed. These include the Context Toolkit [6], the Solar platform [7], and various context services [8,9]. While these solutions help to simplify application development and promote reuse of functionality, we argue that an infrastructure-centred view leads to abstractions for describing and programming with context that are not the most natural ones. In an earlier paper [10], we observed that most of the proposed infrastructures are based on context models that are informal and lacking in expressive power.

It has been our goal, therefore, to develop a framework that integrates a set of well-defined context modelling and programming abstractions with the types of infrastructural support described above. To this end, we designed the conceptual foundations of our framework first, starting with context modelling as our primary interest. As we set out with the objective of creating tools that could support the software engineer in a variety of tasks, we developed modelling techniques that enable incremental refinement over the software lifecycle. In Sections 2.2 to 2.4 we present our three separate yet closely integrated modelling approaches that support (i) the exploration and specification of an application's context requirements, (ii) the management of context information stored in a context repository, and (iii) the specification of abstract classes of context that are close to the way the programmer and end user view context. First, however, we briefly discuss the features of context-aware systems that differentiate the modelling and management of context information from other types of data or knowledge representation and management.

2.1 Characteristics of context information

Context information can originate from a wide variety of sources, leading to heterogeneity in terms of quality and persistence. While much of the previous research in context-awareness focuses only on sensed information, we have found rich context models that integrate sensed, static, user-supplied (profiled) and derived information to be the most useful. These four classes of information each display distinctive characteristics [10]; for example, sensed context is usually highly dynamic and prone to noise and sensing errors, while user-supplied information is initially reliable, but easily becomes out of date.

The problem of imperfect context information is well recognised and some of its causes have already been described. Some context modelling solutions address part of this problem by allowing context information to be associated with

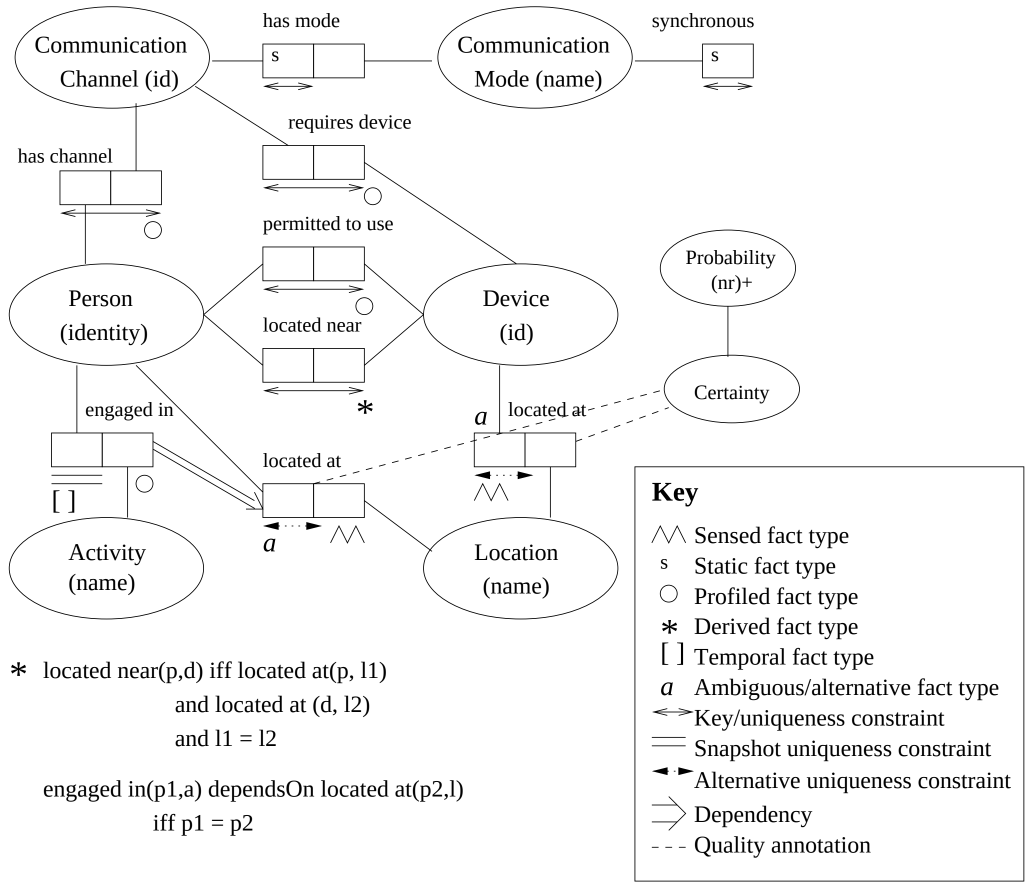

Fig. 1. An example CML model.

quality metadata, such as certainty and freshness estimates [9,11]. However, this approach does not address the entire problem. For instance, it does not provide a solution for representing or reasoning about ambiguity or unknowns. These types of imperfection are common when sensors or other providers of context information report conflicting values, or fail to report values at all. Our modelling abstractions are unique in that they address all of these issues.

2.2 A graphical modelling approach

We developed a graphical context modelling approach, the Context Modelling Language (CML), as a tool to assist designers with the task of exploring and specifying the context requirements of a context-aware application. It provides modelling constructs for describing types of information (in terms of *fact types*), their classifications (sensed, static, profiled or derived), relevant quality metadata, and dependencies amongst different types of information. CML also allows fact types to be annotated to indicate whether ambiguous information is permitted (e.g., alternative location readings), and whether historical information is retained. Finally, it supports a variety of constraints, both general (such as cardinality of relationships) and special-purpose (such as snapshot and lifetime constraints on historical fact types).

Initially, we formulated CML independently of any established information modelling technique. This afforded the flexibility to express the desired concepts in the most flexible way. The results of this initial exploration are presented in a previous paper [10]. Subsequently, we chose to reformulate the modelling concepts as extensions to Object-Role Modeling (ORM) [12]. ORM was adopted because of its closeness to our original modelling approach, its superior formality and expressiveness in comparison to solutions such as ER, and the presence of a mapping to the relational model (allowing a straightforward representation of a context model in terms of a relational database). To illustrate the notation used by CML, we show an example model in Figure 1. This model is suitable for a variety of context-aware communication applications, and represents a subset of the model we developed as part of the case study presented in Section 7. The model captures:

- user activities in the form of a temporal fact type that covers past, present and future activities;
- associations between users and communication channels and devices; and
- locations of users and devices (both absolute and relative, where the latter is represented as a derived fact type).

Each ellipsis in the figure depicts an object type (with the value in parentheses describing the representation scheme used for the object type), while each box denotes a role played by an object type within a fact type. To give an example, the "`has channel`" fact type contains two roles, one played by the Person object type and the other by the Communication Channel object type. An example instance of this fact type is `has_channel`[Michelle Williams, +61 7 3365 4310], where the second value in the fact is a telephone number identifying a channel by which Michelle can be reached. All of our example fact types in Figure 1 contain either one or two roles; however, larger numbers of roles are also permitted.

The annotations on the fact types show that user and device locations are both sensed and can be populated by alternative facts (i.e., each user or device can have multiple recorded locations). Additionally, each recorded location fact has an associated certainty measure in the form of a probability estimate. All other types of context information - with the exception of proximity information, which is derived from the two sensed fact types - are user-supplied (i.e., static or profiled, depending on the persistence of the information). Finally, user activity is modelled as being partially dependent on user location. This dependency is of interest to the infrastructural components responsible for managing context information on behalf of context-aware applications. For example, these components can use their knowledge of the dependency to pro-actively issue queries to refresh activity facts in response to location changes. Further information about CML can be found in some of our previous publications [10,13–15].

2.3 Relational representation

Halpin [12] describes a procedure for mapping from ORM to the relational model. We leverage this mapping to create a relational representation of CML fact types that is well suited to context management tasks, such as enforcement of constraints, storage within a database, and querying by applications. In most cases, the mapping translates each fact type to a corresponding relation, such that roles in the fact type are represented by columns/attributes in the relation, and the ORM and CML constraints are expressed as database constraints. Special treatment is provided for alternative, temporal and quality-annotated fact types and their respective constraints, as discussed in [14].

The relational mapping leads to a representation of context information that consists of a set of facts expressed in the form of database tuples. In order to support reasoning about these facts, we adopt a form of closed world assumption as follows. Assume that R is the set of relations belonging to a context model, I is an instantiation of the model (henceforth termed a context instance), $I(r)$ represents the set of tuples in I belonging to a relation $r \in R$, and **dom** is the set of constant values permitted within any context instance. Then an assertion of the form $r[c_1, ..., c_n]$ (where $r \in R$ and each $c_i \in \mathbf{dom}$ for $1 \leq i \leq n$) is true for I if there is a tuple $< c_1, ..., c_n >$ in $I(r)$, and false otherwise.

As it stands, this simple interpretation does not accommodate uncertain context information. Therefore, we extend it to deal with unknowns (represented by null values in tuples) and ambiguity (represented by alternative facts) using a three-valued logic. An assertion $r[c_1, ..., c_n]$ (where r and $c_1, ..., c_n$ are constrained as before) evaluates to the third logical value (*possibly true*) when the tuple $< c_1, ..., c_n >$ is not present within $I(r)$, but a matching tuple is present when one or more of the constants c_i is replaced with the special *null* value, *or* when the tuple is present, but is ambiguous (that is, there are alternative facts [15]). An assertion is false when it is neither true nor possibly true.

2.4 The situation abstraction

Our graphical modelling notation is well suited for use when specifying the context information used by a context-aware application, and its relational mapping is a natural choice for context storage and management, but neither serves as a natural programming abstraction. Both represent context information at a finer level of granularity than is typically required when describing the conditions that determine application behaviour. Therefore, we developed the situation abstraction as a way to define conditions on the context in terms of the fact abstraction of CML. Situations can be combined, promoting reuse and enabling complex situations to be easily formed incrementally by the programmer. Our situation abstraction is conceptually similar to that proposed by Dey and Abowd [16] for use with their Context Toolkit, but is considerably more expressive.

Situations are expressed using a novel form of predicate logic that balances efficient evaluation against expressive power. They are defined as named logical expressions of the form $S(v_1, ..., v_n) : \varphi$, where S is the name of the situation, v_1 to v_n are variables, and φ is a logical expression in which the free variables correspond to the set $\{v_1, ..., v_n\}$. The logical expression combines any number of basic expressions using the logical connectives, and ($\wedge$), or ($\vee$) and not ($\neg$), and special forms of the universal and existential quantifiers. The permitted basic expressions are either equalities (e.g., $t_1 = t_2$), inequalities (e.g., $t_1 \leq t_2$) or assertions of the form $r[t_1, ..., t_n]$ as described in the previous section.

As there are problems associated with evaluating unconstrained quantified expressions (in terms of efficiency and *unsafe* expressions [15]), we employ the following restricted forms of quantification:

- $\forall x_1, ..., x_i \bullet r[t_1, ..., t_n] \bullet \varphi$
- $\exists x_1, ..., x_i \bullet r[t_1, ..., t_n] \bullet \varphi$

Here, $\{x_1, ..., x_i\} \subseteq \{t_1, ..., t_n\}$, $r[t_1, ..., t_n]$ is an assertion and φ is a logical expression[1]. The assertion in the middle of these expressions serves to restrict the possible values for the variables, $x_1, ..., x_i$, so that φ is evaluated only over these values. Each of the terms in the assertion, $t_1, ..., t_n$, is either equal to one of these variables or a constant value (including the special wildcard value).

The evaluation of a situation S against a binding of values for its n variables, $v_1, ..., v_n$, and a context instance, I, occurs according to the usual semantics of the logical operators under a three-valued logic (with the modifications described above for the universal and existential quantifiers), and according to the closed-world interpretation of assertions that was outlined in the previous section. Typically, the variable bindings are supplied by the context-aware application, and describe selected aspects of the current application state, whereas the context instance is the set of information available through a context management infrastructure residing outside the application (this is the information captured by the CML model).

[1] Note that the symbol "•" acts as a separator and has no special semantics here.

$Occupied(person)$:

$$\exists t_1, t_2, activity \bullet \texttt{engaged\_in}[person, activity, t_1, t_2] \bullet$$
$$(t_1 \leq timenow() \wedge (timenow() \leq t_2 \vee isnull(t_2)) \vee$$
$$(t_1 \leq timenow() \vee isnull(t_1)) \wedge timenow() \leq t_2) \wedge$$
$$(activity = \text{“in meeting”} \vee activity = \text{“taking call”})$$

$CanUseChannel(person, channel)$:

$$\forall device \bullet \texttt{requires\_device}[channel, device]$$
$$\bullet \texttt{located\_near}[person, device] \wedge \texttt{permitted\_to\_use}[person, device]$$

$SynchronousMode(channel)$:

$$\forall mode \bullet \texttt{has\_mode}[channel, mode] \bullet \texttt{synchronous}[mode]$$

$Urgent(priority)$:

$$priority = \text{“high”}$$

Fig. 2. Situations for a communication application, based on the context model in Figure 1.

Some example situations, used in the communication application described in Section 7, are shown in Figure 2. These are specified in terms of the fact types that were defined by the CML model in Figure 1. The *Occupied* situation describes the condition in which a person is engaged in an activity that generally should not be interrupted (here defined to be “in meeting” or “taking call”), on the basis of the temporal “`engaged in`” fact type/relation. It examines exactly those activity facts for which the current time (returned by the *timenow*() function) overlaps with the recorded time interval (providing special treatment for facts that have no recorded start/end time). Similarly, *CanUseChannel* is satisfied for a 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 has permission to use the devices. The *SynchronousMode* situation holds for a given communication channel provided that the mode of this channel (as recorded by the “`has mode`” relation) is synchronous (indicated by its appearance in the “`synchronous`” relation). Finally, the simple *Urgent* situation is satisfied whenever the *priority* variable has the value “high”.

3 Preference model

Appropriate context modelling techniques are a necessary, but not sufficient, prerequisite to managing the complexity involved in engineering context-aware applications. In all but the most trivial applications, additional tools are also desirable to support the decision-making process involved in mapping the context to appropriate application behaviours. This process is complicated by well known usability challenges associated with context-awareness, such as those related to predictability and the trade-off between autonomy and user control [17]. In order to address these problems, the decision-making process must accommodate requirements that vary from person to person and over time.

However, very little research has addressed this problem. One exception is the work of Byun and Cheverst [18], which explores the integration of user modelling techniques into context-aware applications. This work involves the use of machine learning techniques to derive user models that can be leveraged in order to support various proactive behaviours. This work appears promising; however, we argue that explicit means of representing user preferences are also required. The use of an explicit representation allows users to formulate their own preferences if desired, and also provides a tool for context-aware systems to explain choices to users by exposing the associated preference traces. Traceability significantly improves user acceptance, as users are more tolerant of incorrect actions taken by context-aware applications if they are able to understand that they have a rational basis [19]. In addition, preference traces can help users to prevent future erroneous actions by identifying and correcting those preferences that do not have the desired effect. An explicit representation of preferences can also be used in conjunction with automated learning techniques to enable evolution of preferences over time in response to user feedback.

We surveyed a variety of preference modelling approaches, both in the area of context-awareness and in fields such as decision theory and document retrieval, with the aim of identifying a preference model that could be used to support customisable context-aware behaviour. Within context-aware systems, preferences are sometimes regarded as a type of context and modelled accordingly; this is the approach taken by CC/PP [20]. This solution is appropriate for describing very simple requirements (such as languages that are acceptable for presenting information to a user), but not for more sophisticated, context-dependent preferences. We encountered a similar problem with preference models used in other fields: none offered a way to incorporate context as a determinant in preferences. Accordingly, we developed a novel preference model based on our situation abstraction. This was designed to be compatible with automated preference elicitation techniques, and to support composition of preferences (such that users can express simple, possibly conflicting requirements and later combine these to form comprehensive preference descriptions).

Our preference model is loosely based on prior work of Agrawal and Wimmers [21], and supports the ranking of choices according to the context. In the case of our communication application, the choices are communication channels available to people who would like to interact with one another, while, in the information retrieval domain, the choices may be documents or search terms.

`p1=` `when` $SynchronousMode(channel) \wedge \neg CanUseChannel(callee, channel)$

`rate` $\natural$

`p2=` `when` $Urgent(priority) \wedge SynchronousMode(channel)$

`rate` 1

`p3=` `when` $Urgent(priority) \wedge \neg SynchronousMode(channel)$

`rate` 0.5

Fig. 3. Example preferences for channel selection in a context-aware communication application.

Each preference takes the form of a named pair consisting of a scope and a scoring expression. The scope describes the context in which the preference applies, in terms of situations. Recall that situations may evaluate to true, false or possibly true. A preference is considered applicable within a given context only if the scope expression is true.

The scoring expression assigns a score to a choice, which is either a numerical value in the range [0,1] (where increasing scores represent increasing desirability) or one of four special values, as follows:

- $\natural$ represents a veto, indicating that the choice to which the score is assigned should *not* be selected in the context specified in the preference scope;
- $\overline{\wedge}$ represents obligation, which is essentially the opposite of veto;
- $\perp$ represents indifference or an absence of preference; and
- ? represents an undefined score, signalling an error condition.

Figure 3 presents some example preferences that a user might supply to a context-aware communication application to indicate how (s)he would like to be contacted by other people (or, more precisely, which types of communication channels (s)he prefers in particular circumstances). The preference name is shown at the left, while the scope and scoring expression are preceded by the keywords `when` and `rate`, respectively. The first example forbids the use of synchronous channels, such as telephone and video-conferencing channels, when the user does not have access to all of the requisite devices. Preferences `p2` and `p3` together imply that synchronous channels are the preferred choice for urgent calls: `p2` assigns these the highest possible score (1), while `p3` assigns all asynchronous channels (such as email and SMS) a score of 0.5.

Note that the preference format shown in Figure 3 is *not* exposed directly to users. Instead, users typically select from standard preference sets based on natural language descriptions, or construct and combine preferences using graphical editing tools that supply libraries of predefined situations and scoring policies.

Preferences can be grouped into sets and combined according to policies, such that a single score is produced for each choice that reflects all preferences in the set. The policies dictate the weights attached to individual preferences and determine how conflicting preferences are handled. One common policy involves averaging the numerical scores, but allowing vetoes, obligations and undefined scores to override. To see how this policy works, consider the preferences in Figure 3 and a context and set of variable bindings for which *SynchronousMode(channel)* and *Urgent(priority)* are true, and *CanUseChannel(callee, channel)* is false. Here, `p1` evaluates to $\natural$ (veto), `p2` evaluates to 1 and `p3` evaluates to $\perp$ (indifferent), as its scope expression is false in this context. In this small example, the average of the numerical scores is simply 1, however the veto produced by `p1` overrides. This implies that the channel represented by the *channel* variable is unsuitable in this context.

Policies need not be static. For example, user feedback can be used to dynamically adapt a policy to a user's requirements (e.g., by redistributing the weights assigned to preferences).

The following section shows how preferences are used by context-aware applications to support decision making about which behaviours or actions are appropriate in a given context.

4 Programming models

Suitable programming models are crucial in helping to limit the complexity and effort involved in implementing context-aware applications; however, progress in developing new models has been slow. Although context servers are now frequently used for acquiring and managing context information, most applications do not make use of any form of support (for instance, programming toolkits or infrastructure) for interpreting and making decisions about context. In general, context-aware software is developed using traditional programming methods and models, and the use of context information is embedded directly into the source code. In some cases, the logic used to process context information and react to context changes is isolated within special components, as in the enactor model proposed by Newberger and Dey [22]. This approach leads to cleaner code than an unstructured approach, but still results in applications that are difficult to maintain, as source code must be modified in order to support additional classes of behaviour and context. Some models have been proposed that do not suffer from this problem [23], but these are applicable only to very narrow domains.

In the following sections we describe two general programming models that build on our situation and preference abstractions. The branching model offers

```
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);
```

Fig. 4. Selected methods of the programming toolkit's branching API.

a novel and flexible means to insert context- and preference-dependent decision points into the flow of application logic. In contrast, the triggering model supports an event-driven programming style. It has been widely used previously in the implementation of adaptive applications, but is reformulated here to exploit the situation abstraction as a basis for describing context changes.

The two programming models are complementary, in that they can be used together in a single application to address different problems. To see this, consider a context-aware tourist guide. The triggering model can be used by this application to generate requests for information in a proactive fashion (for example, when the user moves), while the branching model can be used to select the best information to display in the new context. However, some applications may require only one of the models, as in the case of the communication application we discuss later in the paper.

4.1 Branching

The branching model is designed to assist in decision problems involving a context-dependent choice amongst a set of alternatives. Arbitrary choice types can be supported; for instance, in information retrieval, branching can be used to select relevant information to present to the user and suitable modes of presentation, while, in a communication domain, it can be used to identify appropriate communication channels for interactions between users. In each of these domains, context-dependent choices are typically implemented using if- or case-statements. However, this approach results in a tight binding of the context model to the application logic, making it difficult to later evolve the context model as the sensing infrastructure and user requirements change.

To overcome this problem, we exploit the preference model described in Section 3 in our model of branching. User preferences form the link between the context and the chosen action(s); that is, preferences assign ratings to the alternatives according to the context and other application parameters, and, based on these ratings, the application selects and invokes one or more actions.

This solution is extremely flexible, as preference information is expressed in an application-neutral format that (i) enables modification and fine-tuning when required and (ii) facilitates sharing of preferences between applications.

We have implemented support for branching in the form of a Java programming toolkit. A small subset of the API is shown in Figure 4. This provides a variety of methods for evaluating and selecting one or more candidate choices according to the context. For instance, the `rate` method has as its parameters:

- a set of choices;
- a `Preference`, which is generally a composite preference (or policy) combining a large set of requirements for one or more users;
- a `Valuation`, binding variables contained in the preference to constant values according to the current application state; and
- a `Context`, which is a wrapper for a repository of context information.

The method uses these values to compute and return a mapping of choices to scores, which the application can interrogate and act upon as necessary. The next two methods perform similar evaluations, but, instead of returning mappings, they select and return the single best and the best n choices, respectively, on the basis of scores assigned by the preference. The remaining methods perform selections based on other criteria: `selectAbove` returns the set of choices assigned numerical scores above a specified threshold, while `selectMandatory` returns the set of choices assigned the obligation ($\overline{\wedge}$) score.

An example use of the toolkit to select communication channels appropriate to users' contexts and preferences is illustrated later in Section 7.2.

In future versions of the branching toolkit, we intend to support not only selection of choices, but also trace and user feedback mechanisms. The trace mechanisms will allow users to visualise links between their preferences and their applications' actions, while the feedback mechanisms will be used to support automated preference learning and evolution.

4.2 Triggering

To support an asynchronous style of programming in which actions are invoked in response to context changes, we also provide a trigger mechanism which builds on the situation abstraction. Context changes are described as changes in situation states. As there are three states (true, false and possibly true), there are six distinct state transitions. Triggers can be associated with any of these transitions, which we write as *TrueToFalse(S)*, *TrueToPossiblyTrue(S)*, and so on, where S is a situation. We also allow transitions such as *EnterFalse(S)*, which matches both *TrueToFalse(S)* and *PossiblyTrueTo-*

```
upon    EnterFalse(Occupied("Amy Carr"))
when    true
do      Notify of recent missed calls
always
```

Fig. 5. An example trigger.

False(S), and *Changed(S)*, which matches any of the six state transitions on S. Finally, we allow triggers to be attached to sequences of transitions (written $t_1 \rightarrow ... \rightarrow t_n$, where t_1 to t_n are transitions), or sets of alternative triggers (written $t_1|...|t_n$).

Our triggering mechanism follows the event-condition-action model, in which each trigger includes a precondition on the invocation of the specified action that is evaluated upon detection of the event. The precondition, like the event, is specified in terms of situations. Our model also associates each trigger with a lifetime, which is one of the following:

- once;
- from <start> until <end>;
- until <end>;
- n times; or
- always.

An example trigger appears in Figure 5. The event, condition and action are prefixed by the keywords `upon`, `when` and `do`, respectively. The action is described in natural language for simplicity, but usually takes the form of an invocation of relevant source code. This trigger has the effect of notifying the user, Amy Carr, about recent missed calls at the conclusion of any engagement, where engagements are defined according to the *Occupied* situation in Figure 2. The trigger has no additional preconditions beyond the detection of the specified event, so the condition is simply the value *true*.

Support for the triggering model is provided in our programming toolkit alongside the branching functionality described in the previous section.

5 Software infrastructure

We have implemented a software infrastructure incorporating our programming toolkit and support for related tasks, such as management of context information. In this section, we present an overview of the architecture and implementation.

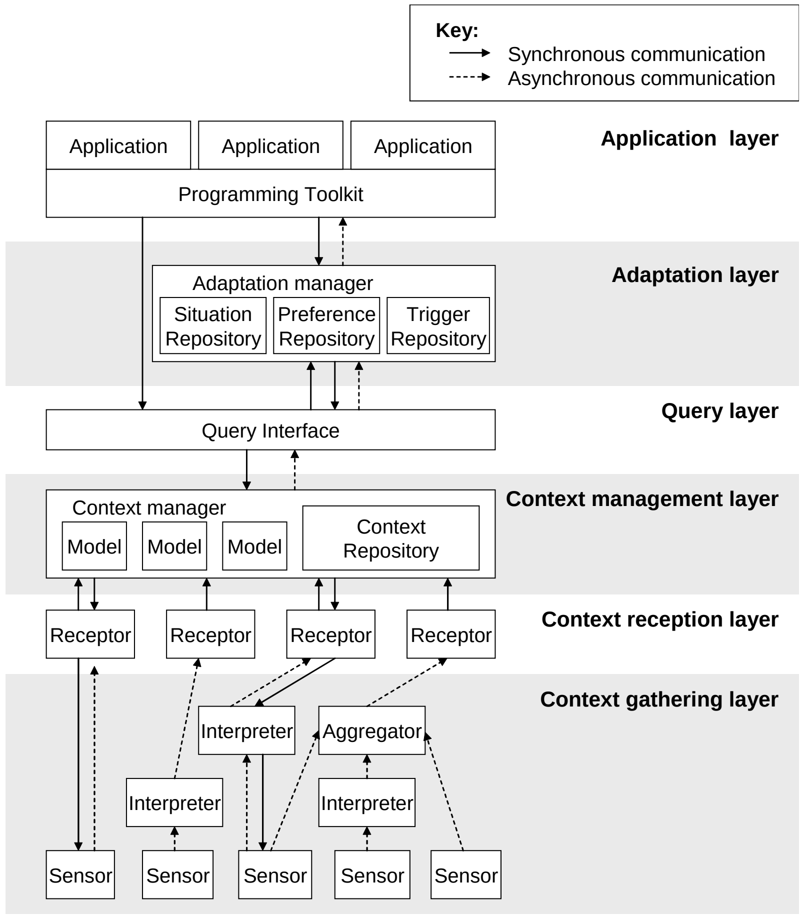

Fig. 6. The layered architecture of our software infrastructure.

The infrastructure is organised into loosely coupled layers as shown in Figure 6. The context gathering layer acquires context information from sensors and then processes this information, through interpretation and data fusion (aggregation), to bridge the gap between the raw sensor output and the level of abstraction required by the context management system. This involves mapping sensor outputs onto appropriate context facts and ensuring an appropriate frequency of updates, in order to balance resource consumption against the quality of the context data. A content-based routing scheme [24] is used to achieve a loose coupling between the sensing and processing components and the reception layer. This introduces tolerance for component failures, disconnections and evolution of the sensing infrastructure.

The context reception layer provides a bi-directional 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.

The context management layer is responsible for maintaining a set of context models and their instantiations, expressed in terms of the relational representation described in Section 2.3. Applications may define their own context models, or share models with other similar applications. In addition to handling sensed information that propagates up through the reception layer, the context management layer supports static, derived and profiled information. Derived information is handled internally using standard database mechanisms (typically, using either virtual or materialised views, depending on the performance requirements and frequency of queries). To support static and profiled information, which are inserted into the context management system by human users, the layer provides customisable user interfaces for browsing and manipulating selected types of context information.

The query layer provides applications and other components of our software infrastructure with a convenient interface with which to query the context management system. It supports queries in terms of both facts and situations, and masks distribution within the context management layer by providing query routing services. Both simple, synchronous queries and persistent, asynchronous queries are permitted. The former are used in preference evaluation and the latter in trigger evaluation.

The adaptation layer manages common repositories of situation, preference and trigger definitions, and evaluates these on behalf of applications using services of the lower layers. Repositories are generally shared by groups of applications (e.g., applications running on a single device or network, or belonging to a single user).

Finally, the application layer provides toolkit support for our programming models. The branching API was described in detail in Section 4.1. The triggering API supports dynamic creation, activation and deactivation of triggers.

Our current version of the infrastructure is implemented in Java, using various pieces of open-source software. The context and adaptation managers use the standard Java Database Connectivity (JDBC) API[2] and the PostgreSQL RDBMS[3] for storage of fact types, situations and preferences. However, the use of JDBC leaves open the possibility of substituting other RDBMS software in the future (for instance, lightweight implementations suitable for operation on resource-constrained devices). Parsing of situation, preference and trigger definitions is performed by parsers constructed using the JavaCC parser generator[4]. This approach enables us to trivially regenerate the parsers whenever we extend the grammars, which is invaluable for rapid prototyping purposes.

[2] http://java.sun.com/products/jdbc
[3] http://www.postgresql.org/
[4] http://javacc.dev.java.net/

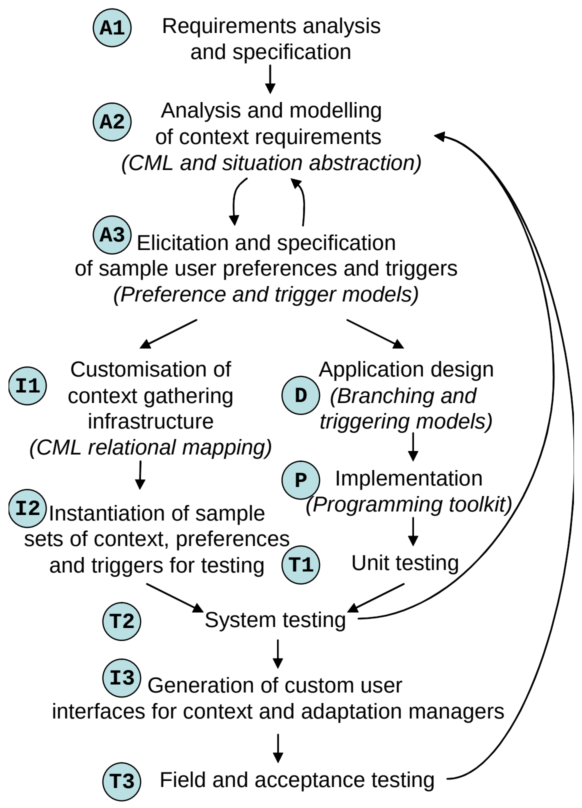
A1
Requirements analysis and specification
A2
Analysis and modelling of context requirements
(CML and situation abstraction)
A3
Elicitation and specification of sample user preferences and triggers
(Preference and trigger models)
I1
Customisation of context gathering infrastructure
(CML relational mapping)
D
Application design
(Branching and triggering models)
P
Implementation
(Programming toolkit)
I2
Instantiation of sample sets of context, preferences and triggers for testing
T1
Unit testing
T2
System testing
I3
Generation of custom user interfaces for context and adaptation managers
T3
Field and acceptance testing

Fig. 7. The software engineering process. The tools used at each step are shown in parentheses. A3, T2 and T3 can be followed by re-iteration through earlier steps.

6 Software engineering methodology

The models and infrastructure that we have presented are designed to support a wide variety of software engineering tasks. In this section, we outline the process that is generally followed when building a context-aware application using these tools. This process was abstracted from our experiences with building several context-aware applications, some of which are described in Section 7.

Figure 7 illustrates our generic software engineering process graphically. The steps can be partitioned into the following tasks: **analysis** (A); **design** (D); **implementation/programming** (P); **infrastructure customisation** (I); and **testing** (T).

The design and implementation steps rely on the branching, triggering and context query APIs to incorporate context-aware functionality, but otherwise adopt traditional methodologies and languages. Therefore, we focus our discussion on the remaining parts of the software engineering process.

6.1 Analysis

The analysis task begins with the general goal of documenting the functionality and requirements of the application (step A1), as in most other software lifecycle models. Following this initial step, our software engineering process includes two additional steps specific to context-aware applications. The first (A2) focuses on the types of context information that are required in order to implement the functionality identified at A1. Gray and Salber [25] provide a good discussion of the issues that need to be considered at this stage. They include identifying, for each type of context information:

(1) the required information quality in terms of resolution, timeliness, and so on; and
(2) sources for the information that are suitable in terms of intrusiveness, cost and other constraints.

In our approach, this step produces a context model in two parts. The first is a CML model, as in Figure 1, documenting basic fact types, their origins (sensed, derived or profiled), appropriate types of quality metadata, domain constraints, and so on. The second is a set of situations to which the application can adapt, as in Figure 2, defined in terms of fact types and other situations. These outputs are rarely produced from scratch, as there are considerable overlaps in the context requirements of many context-aware applications.

The final analysis step (A3) refines the functionality identified in A1, focusing on the specification of those aspects that are context-dependent. In this step, the analyst identifies those choices and events that are influenced or triggered by the context, and, for each of these, documents the role of context by producing sample preferences and triggers. This is generally performed iteratively with step A2, until the outputs of the two steps are consistent.

Following analysis, the software engineering process diverges into two sets of tasks that can be performed in parallel, one concerned with design and implementation, and the other with customisation of the software infrastructure.

6.2 Infrastructure customisation

Prior to executing a new application on top of our software infrastructure, it is usually necessary to customise some of the components. If new fact types or situations are required, these must be inserted into the management layers shown in Figure 6. This step is largely automated by a tool that inputs textual representations of fact types and situations and, based on the relational mapping described in Section 2.3, uses them to generate scripts that manipulate the relevant databases [26]. The addition of sensed fact types may also require the implementation of receptors, interpreters and aggregators for the context gathering and reception layers. These tasks make up step I1 in Figure 7.

For system testing purposes, we generally require sample sets of context information, preferences and triggers. Step I2 consists of building appropriate sets and using them to populate the context and adaptation managers.

Step I3 occurs prior to deployment, once the application and underlying models are stable, and consists of mapping the context model, preference sets and triggers developed at steps A2, A3, I2 and T2 to customisation interfaces. These enable developers, system administrators and users to browse and manipulate profiled context information and configure preferences and triggers via appropriate user interface abstractions.

6.3 Testing

The task of testing context-aware applications involves unique challenges. Tse et al. [27] point out that it is no longer adequate to perform unit testing solely on the basis of source code, as in traditional white box methods, when part of the application behaviour is determined by triggers or rules that are separate from the code. Bylund and Espinoza [28] describe additional difficulties that arise when testing applications that rely on sensor data, and argue that testing with both live and simulated data is essential. Unfortunately, satisfactory solutions to these problems do not yet exist, so we describe a typical, rather than an ideal, testing process.

At the unit testing stage (T1), we apply traditional white box testing methods, using a test framework such as JUnit [5] . At the system test stage (T2), we carry out black box testing using specially constructed sets of context information, preferences and triggers. Because the behaviour of a context-aware application is dependent on many variables (including numerous kinds of context and each user's particular configuration of preferences and triggers), it can be difficult to anticipate the exact nature of the behaviour in advance solely by studying the context model, preferences and triggers. Consequently, a large part of the goal of system testing is to experiment with and fine-tune the preferences and triggers developed during step A3, and verify that each combination of context, preferences and triggers yields satisfactory behaviour. The final stage of testing (T3) evaluates the application in the field, using a realistic hardware environment, live sensor data, and real users. At this stage, information quality and privacy issues associated with sensors and other sources of context

[5] http://junit.sourceforge.net/

information may be uncovered, leading to possible re-iteration through one or more earlier steps.

7 Case study: context-aware communication

As a means of validating our models and infrastructure, we carried out a case study in which we built a context-aware communication tool. This section presents the objectives, design and outcomes of this study. Since completing the case study, we have further demonstrated the value of our approach by applying it to a variety of applications, some of which are briefly outlined in Section 7.3.3. A full discussion of these applications is beyond the scope of this paper; however, we refer the interested reader to two recent papers [29,30] for more information.

7.1 Objectives

The goals of the case study were to evaluate the ability of our models and infrastructure to support software engineering tasks and to facilitate the development of flexible and evolvable software. The study primarily considered core application development tasks, rather than infrastructure-related tasks (e.g., development of interpreters and receptors). As our primary interest lies in rapid prototyping, our evaluation was *not* concerned with performance issues, such as the efficiency of the context management system. Neither was it concerned with the expressiveness of our modelling approaches; this is already discussed in some of our other papers [10,30] and demonstrated implicitly by the fact that we have used the approaches in the development of a variety of context-aware applications, as discussed briefly in Section 7.3.3.

Figure 8 lists the software quality measures that we considered in the study, and the metrics we used to evaluate the quality of software with respect to each measure. This set of measures and metrics is not exhaustive, but reflects the principal goals of our software engineering approach. Usability metrics, such as the rate of inappropriate context-aware actions and ease of configuration, are also important, but were outside the scope of the initial study.

Our goal was to perform a comparative evaluation, contrasting the prototype that we developed with alternative implementations developed without our programming models and infrastructure. We discuss the prototype in the following section and the results in Section 7.3. Note that, as many alternative implementations are possible (depending on assumptions about how context information is queried, interpreted and so on), it is not feasible to present a

Quality measures	**Metrics for evaluation**
Code complexity	Lines of application code concerned with context querying, manipulation and processing
Maintainability and support for evolution	New or modified lines of code required to support changes in context model or context-based requirements
Reusability	Ease of reuse of context definitions and context processing components (e.g., fact and situation definitions, preferences, triggers, context interpreters and aggregators)

Fig. 8. Selected software quality measures for context-aware applications.

straightforward quantitative comparison. Therefore, our discussion is necessarily qualitative in places rather than quantitative. Note, also, that the formality of our evaluation is somewhat limited by the absence of established methods for evaluating software quality in relation to context-aware applications.

7.2 Application

The application developed in our case study took the form of a tool designed to recommend communication channels for interactions between people based on context and preferences. Given a priority, topic and list of people, the tool suggests appropriate contact addresses, such as phone numbers or instant messaging IDs. It aims to minimise disruption, prevent missed calls/messages and improve the timeliness of interactions.

In the remainder of this section, we briefly outline the process that was used to develop the communication tool, with reference to a subset of the software engineering steps described in Section 6. As the tool was developed as a prototype, rather than a stable application, our testing was mainly carried out informally. Similarly, as the design and infrastructure customisation tasks were both straightforward, we do not discuss them here.

7.2.1 Analysis

We developed a context model and sample sets of user preferences for the tool following an informal user study[6]. In this study, we asked people to record their interactions with other people during the course of a day, the channel used for each interaction (e.g., a particular telephone number or email address), and the reason for the choice of channel. By examining the cited reasons, we were able to compile a list of the most relevant types of context,

[6] Note that triggers were not required for the chosen application design.

such as the current activity of the other person, the importance of the interaction, and the time (e.g., inside or outside working hours). Most of the basic types of context, such as user activity, could be mapped directly to fact types, while some derived context types, such as temporal conditions, were better represented as situations. We also mapped some types of context (e.g., the priority of the user's current interaction) to application state variables, rather than to fact types, to represent information supplied directly to the application by the user. The application state variables also appear in relevant situations and preferences.

We already showed a subset of the CML model produced for the application in Figure 1. The main types of information captured by the model are activity, location and proximity data (represented by the "`engaged in`", "`located at`" and "`located near`" fact types shown in Figure 1), associations between people, communication channels and devices (captured by the "`has channel`" and "`permitted to use`" fact types) and information about interpersonal relationships (not shown). For each fact type, we investigated suitable sources of the information for our prototype, taking into account the required implementation effort and relevant usability issues (e.g., privacy concerns and the burden on the user). We decided to derive location and proximity information from a variety of location sensors, but to rely on users to specify most other types of context information. Some of the user-supplied information requires direct configuration (as in the case of information about users' communication channels, represented by the "`has channel`" fact type), while other information can be obtained indirectly (as in the case of user activity information, which can be pulled from calendar applications).

After creating the context model, we re-examined the list of reasons cited by our study participants for their choices of communication channels, and mapped each reason to one or more preferences. In doing so, we identified some fact types and situations that were previously missing from our context model. We also formed comprehensive preference sets for several representative study participants, which we later used for testing purposes. A small subset of one of the preference sets appeared in Figure 3, while the corresponding situation definitions appeared in Figure 2.

The most challenging aspect of specifying the preferences was deciding suitable preference ratings and choosing a policy with which to combine the ratings. For the communication tool (and most of our other applications), we used the simple averaging policy described in Section 3 and a restricted set of numerical preference ratings representing *high*, *medium-high*, *medium*, *medium-low* and *low* preference. In our experience, it is rarely necessary to use more than these five levels of preference (in addition to the special non-numerical values), although our model allows arbitrary values in the range [0,1].

After drafting the preferences sets, we worked through the preference evaluations by hand for a small set of scenarios, and tweaked the preferences in certain cases. The task of drafting, testing and tweaking preferences in this way is, of course, beyond the scope of the average user. This reinforces our argument presented in Section 3 that our preference model should be used by application developers, but not exposed to users. Instead, the developer should provide users with configuration options that can be used to manipulate the preferences in a more controlled way than editing preferences directly.

7.2.2 Implementation

We implemented a prototype in Java using the query and branching facilities of our programming toolkit. It is mainly designed for use on desktop PCs or laptops, but could also run on a mobile device such as a PDA. Each user executes a copy of the tool (henceforth referred to as an agent), which is responsible for evaluating his/her context and preferences and maintaining a personal history of interactions. When requested for a recommendation, the agent sends the details of the requested interaction to the agents operating on behalf of the other participants, soliciting suggestions of suitable channels. These agents query the "`has channel`" fact type to find contact addresses suitable for their users, and then invoke the branching toolkit's "`rate`" method to evaluate their suitability against the user's context and preferences. If any contact addresses receive the obligation score ($\overline{\wedge}$), the agent returns these to the querying agent; otherwise, it returns highly rated addresses. The querying agent combines the results, filters and sorts them on preferences (again using the `rate` method), and presents the results to the user.

The implementation of the prototype and its use of the branching toolkit are discussed further in Section 7.3.1.

7.3 Evaluation

An evaluation of the prototype with respect to the software quality measures described in Section 7.1 follows.

7.3.1 Code complexity

Our design of the tool restricts context queries and invocations of the branching toolkit to the single Java class that implements the channel selection algorithm. This functionality represents approximately two dozen lines of source code (or less than 1 percent of the total). A representative sample consisting of a little more than half of this code is shown in Figure 9. This code fragment implements a method that is used to first discover the communication channels available to the user (lines 7 to 9), and then to filter these to obtain the most suitable channels according to the user's requirements and context (lines 12 to 35). Without loss of generality, we concentrate on the implementation of this method throughout most of our evaluation, in order to keep the discussion focused and concrete.

We have numbered the lines in the listing to aid in our discussion; note, however, that when discussing line of code (LOC) counts, we actually refer to statement counts, not numbers of new line characters[7]. Thus, we count the code in Figure 9 as 16 lines of code rather than 36.

Each agent explicitly queries only a single fact type each time the channel selection procedure is invoked (lines 7 to 9). This means that, although our context model initially contained over a dozen fact types (and grew subsequently as we incorporated new types of context), most of these could be ignored by the programmer. The branching model shifts most of the context evaluation to the programming toolkit and the preference management layer. This evaluation is carried out in response to the agent's two calls to the `rate` method of the programming toolkit's branching API (one of which is shown in lines 20 to 24).

Without the use of branching (and the supporting preference and situation abstractions), the number of context queries and the lines of code devoted to context processing would both be substantially larger. To illustrate, we show an alternative implementation of the "`selectChannels`" method in Figure 10 which uses our programming toolkit only for direct queries on context facts. This implements preference `p1` from Figure 3; that is, it retrieves all communication channels associated with the user, and then discards any synchronous channel that requires a device (i) that is not in close proximity to the user, or (ii) for which the user lacks the required permissions. Note that, although only one preference is implemented, the number of context queries has risen from one to six (lines 7-9; 16-18; 23-24; 29-31; 38-40 and 41-43), and the LOC count for the method has grown from 16 to 26. Lines 14 to 48 in the listing are responsible for implementing preference `p1`, and collectively have a LOC count of 18. Extrapolating from this, the implementation of even a moderate number of additional user preferences in a similar fashion - say, a dozen preferences - adds above two hundred lines of code. This represents a more than ten-fold increase over our implementation of `selectChannels` in Figure 9, which is able to support arbitrary numbers and types of user preferences.

Note that with appropriate tool support, the LOC count for code written

[7] To be more precise, we count the number of semicolons, including special cases such as semicolons in the conditions of for-loops, *plus* the number of for-, if-, else- and case-statements.

```
1 : Channel[] selectChannels(
2 :    Identity initiator, Priority priority, String subject)
3 : throws
4 :    NoChannelsException
5 : {
6 :    // Look up suitable communication channels
7 :    Valuation[] channels = context.bind(
8 :       FactTypes.HAS_CHANNEL,
9 :       new Tuple(user, Variables.CHANNEL));
10:
11:    // Bind state variables for preference and situation evaluation
12:    Valuation stateVariables = new Valuation();
13:    stateVariables.bind(Variables.PERSON, config.getProperty(USER));
14:    stateVariables.bind(Variables.CALLEE, config.getProperty(USER));
15:    stateVariables.bind(Variables.CALLER, initiator);
16:    stateVariables.bind(Variables.PRIORITY, priority);
17:    stateVariables.bind(Variables.SUBJECT, subject);
18:
19:    // Evaluate the channels using the branching toolkit
20:    Scores scores = Branching.rate(
21:       Choice.toChoices(channels),
22:       new Preference(config.getProperty(USER_PREFERENCE_NAME)),
23:       stateVariables,
24:       context);
25:
26:    // Process the results of the preference evaluation
27:    if (scores.hasOblige())
28:       return toChannelArray(scores.getOblige());
29:    else if (scores.hasNumerical())
30:       return toChannelArray(scores.getBestN(
31:          config.getProperty(MAX_CHANNELS)));
32:    else if (scores.hasIndifferent())
33:       return toChannelArray(scores.getIndifferent());
34:    else
35:       throw new NoChannelsException();
36: }
```

Fig. 9. A code fragment taken from our prototype. This contains the single context query made by the application, and one of the two preference evaluations mentioned in Section 7.2.

using our programming toolkit can be reduced by one third to one half the current level. McFadden et al. [26] demonstrate that helper classes generated for specific context models can reduce the need for tasks such as packing and unpacking of the `Valuation` objects containing variable bindings used in context queries and preference evaluations (lines 12 to 17 in Figure 9).

```
1-10:  AS PER FIGURE 9.
11:    List availableChannels = new Vector();
12:    for (int i = 0; i < channels.length; i++)
13:    {
14:       // Determine whether the channel type is synchronous
15:       Channel channel = channels[i].lookup(Variables.CHANNEL);
16:       Valuation[] modes = context.bind(
17:          FactTypes.HAS_MODE,
18:          new Tuple(channel, Variables.COMMUNICATION_MODE));
19:       if (modes.length != 0)
20:       {
21:          CommunicationMode mode =
22:             modes[0].lookup(Variables.COMMUNICATION_MODE);
23:          TruthValue synchronous = context.EvaluateAssertion(
24:             FactTypes.SYNCHRONOUS, new Tuple(mode));
25:          if (synchronous.isTrue())
26:          {
27:             // For synchronous channels, check that the user
28:             // currently has the requisite devices on hand
29:             Valuation[] requiredDevices = context.bind(
30:                FactTypes.REQUIRES_DEVICE,
31:                new Tuple(channel, Variables.DEVICE));
32:             for (int j = 0; j < requiredDevices.length; j++)
33:             {
34:                Device device =
35:                   requiredDevices[j].lookup(Variables.DEVICE);
36:                Tuple tuple =
37:                   new Tuple(config.getProperty(USER), device);
38:                boolean permittedToUse =
39:                   context.EvaluateAssertion(
40:                      FactTypes.PERMITTED_TO_USE, tuple).isTrue();
41:                boolean locatedNear =
42:                   context.EvaluateAssertion(
43:                      FactTypes.LOCATED_NEAR, tuple).isTrue();
44:                if (locatedNear && permittedToUse)
45:                   availableChannels.add(channel);
46:          } else
47:             availableChannels.add(channel);
48:       }
49:    }
50:    if (availableChannels.size() == 0)
51:       throw new NoChannelsException();
52:    return toArray(availableChannels);
53: }
```

Fig. 10. A code fragment that evaluates the suitability of communication channels, *without* using the situation and preference definitions and branching API. Lines 1 to 10 are omitted as they are identical to those in Figure 9. The code has the same result as the first preference listed in Figure 3.

7.3.2 Maintainability and support for evolution

The loose coupling between the source code of our prototype and the underlying context model makes it trivial to modify almost all of the latter in response to changes in the sensing infrastructure or user requirements, without changes to the former. Removal or modification of fact types may require situation or preference definitions to be updated; however, only changes to the definition of "`has channel`" can necessitate code changes, and these changes are likely to be confined to just a few lines (7 to 9) of the `selectChannels` method that we showed in Figure 9. The same cannot be said of our alternative implementation, which, despite having severely limited functionality (a single preference), is already dependent on six fact types ("`has channel`", "`has mode`", "`synchronous`", "`requires device`", "`permitted to use`" and "`located near`"), and is therefore much less tolerant of changes in the context model.

New fact types and situations can also be added to our prototype without modification of code or preferences, although these cannot be exploited until the preferences are updated. We have used this feature on several occasions to incorporate new types of sensors. In our alternative implementation, some coding effort is required to take advantage of additions to the context model. In most cases, this effort is likely to be substantial, as the context processing logic is complex and difficult to structure appropriately for large context models or large numbers of user preferences.

Finally, changes in context-based user requirements can be incorporated into our prototype simply by manipulating the preference definitions; this is considerably easier than changing the decision logic embedded in the source code of the alternative implementation.

When using our software engineering approach, the most time consuming task involved in accommodating changes in the sensing infrastructure lies in producing components that process the outputs of new sensors (i.e., interpreters, aggregators and receptors). This task is unavoidable in all context-aware systems. However, by using a common infrastructure and allowing elements of the context models to be shared, the effort can usually be amortised over several applications. Additionally, receptor code for our infrastructure can be automatically generated as demonstrated by McFadden et al. [26].

7.3.3 Reusability

The communication prototype represented the first application developed using our models and infrastructure, so it was necessary to design its context model and preferences from scratch. However, the potential for reuse became apparent when we developed subsequent applications, which included additional communication tools and a vertical handover prototype capable of dynamically switching between network interfaces based on network Quality of Service (QoS), location changes and other context information. For our later communication applications, we modified the original context model only slightly to add information about device capabilities and status. However, we could reuse portions of the preference sets defined for the first prototype. For the vertical handover prototype, we made minimal extensions to the context model to represent network QoS and network interfaces supported by each device. We were also able to reuse situations, interpreters and receptors developed for the communication tools, but had to define new preferences to support the selection of network interfaces.

Reuse is more difficult in the alternative implementation shown in Figure 10. As this implementation retains the context model used by our prototype (as well as the use of our context management infrastructure), reuse of fact types and infrastructural components such as interpreters and aggregators remains possible. However, situation and preference definitions are both embedded in the source code. As this code is reasonably complex (given that it implements only a single preference), and is difficult to structure effectively, reuse is problematic and error-prone.

7.3.4 Summary

Figure 11 summarises the results of our evaluation with respect to the set of quality metrics shown in the leftmost column. The second column presents the results for the `selectChannels` implementation we produced as part of our prototype, which uses our branching model and API, and is shown in Figure 9. The third column evaluates the implementation presented in Figure 10, which uses our programming toolkit only for direct queries on fact types, and implements only a single preference. Finally, the rightmost column considers an extension of the single-preference implementation to a reasonably large number of preferences, according to the assumptions discussed in Section 7.3.1.

The implementations in the last two columns can be regarded as generally representative of context-aware software of low and medium levels of complexity, implemented with infrastructural support for context management and querying (somewhat similar to that provided by the Context Toolkit and Solar), but *without* high-level programming models to support a flexible mapping of contexts to appropriate actions. Most context-aware applications are currently implemented in this fashion. The results present a compelling justification for software engineering approaches and infrastructures of the form presented in this paper, which provide a well integrated set of high-level design and programming abstractions, in addition to support for context acquisition, management and querying.

Software quality metric	*With branching API (Figure 9)*	*Without branching API - 1 preference (Figure 10)*	*Without branching API - many preferences*
Total LOC	16	26	$\geq$ 200
Number of explicit context queries	1	6	$\geq$ number of fact types [a]
LOC affected by removal or modification of fact types	$\geq$1	$\geq$6	$\geq$ number of fact types [a]
New LOC required to exploit new fact types	0	Numerous [b]	Numerous [b]
New LOC required to support new preferences	0	Numerous [c]	Numerous [c]
New/modified LOC required to support modification of preferences	0	Numerous [c]	Numerous [c]
Mechanisms for context reuse	Fact types and situations; interpreters and aggregators	Fact types; interpreters and aggregators	Fact types; interpreters and aggregators
Mechanisms for preference reuse	Shared preferences	Copied/shared source code	Copied/shared source code

[a] Assuming that all types of context information are used in the preferences.
[b] Actual LOC count varies depending on the use of context.
[c] Actual LOC count varies depending on the preferences.

Fig. 11. A comparison of `selectChannels` implementations.

8 Concluding remarks

This paper presented a set of conceptual models designed to facilitate the development of context-aware applications by introducing greater structure and improved opportunities for tool support into the software engineering process. As the evaluation in the previous section showed, our models and approach lead to applications that are maintainable, evolvable and based upon a set of reusable foundations, such as context definitions and context processing components. Further, they support a high degree of customisation by users, which is generally not the case with context-aware applications developed using ad hoc software engineering approaches.

As future work, we plan further case studies to evaluate other aspects of our approach, such as the usability issues mentioned in Section 7.1. We also intend to extend our branching toolkit to support user feedback mechanisms that can be used in conjunction with algorithms for learning preferences. In the longer term, we are interested in investigating several aspects of the software engineering process that remain poorly understood in relation to context-aware software. We believe that challenges related to testing, in particular, will need to be addressed further before context-aware applications become widely deployed.

9 Acknowledgements

We gratefully acknowledge the contributions of Ted McFadden, Sasitharan Balasubramaniam, Peter Mascaro and Jessica Purser to the applications developed following our initial case study, which were described briefly in Section 7.3.3.

References

[1] K. Cheverst et al., *Experiences of developing and deploying a context-aware tourist guide: the GUIDE project.* 6th International Conference on Mobile Computing and Networking, Boston, August, 2000. pp. 20-31.

[2] S. N. Patel and G. D. Abowd, *The ContextCam: Automated Point of Capture Video Annotation.* 6th International Conference on Ubiquitous Computing. In: Lecture Notes in Computer Science, Volume 3205, pp. 301-318. Springer, 2004.

[3] Y. Shi et al., *The Smart Classroom: Merging Technologies for Seamless Tele-Education.* IEEE Pervasive Computing, 2(2): pp. 47-55, April-June, 2003.

[4] H. Chen et al., *Intelligent Agents Meet Semantic Web in a Smart Meeting Room.* 3rd International Joint Conference on Autonomous Agents and Multiagent Systems, July, 2004. pp. 854-861.

[5] S. Helal et al., *Enabling Location-Aware Pervasive Computing Applications for the Elderly.* 1st IEEE Conference on Pervasive Computing and Communications, Fort Worth, March, 2003.

[6] A. K. Dey, D. Salber and G. D. Abowd, *A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications.* Human-Computer Interaction, 16(2-4): pp. 97-166, 2001.

[7] G. Chen and D. Kotz, *Context Aggregation and Dissemination in Ubiquitous Computing Systems*. 4th IEEE Workshop on Mobile Computing Systems and Applications, Callicoon, June, 2002.

[8] J. Pascoe, *Adding Generic Contextual Capabilities to Wearable Computers*. 2nd International Symposium on Wearable Computers, October, 1998. pp. 92-99.

[9] H. Lei et al., *The Design and Applications of a Context Service*. ACM SIGMOBILE Mobile Computing and Communications Review, 6(4): pp. 45-55, October, 2002.

[10] K. Henricksen, J. Indulska and A. Rakotonirainy, *Modeling Context Information in Pervasive Computing Systems*. 1st International Conference on Pervasive Computing. In: Lecture Notes in Computer Science, Volume 2414, pp. 167-180. Springer, 2002.

[11] A. Schmidt et al., *Advanced Interaction in Context*. 1st International Symposium on Handheld and Ubiquitous Computing. In: Lecture Notes in Computer Science, Volume 1707, pp. 89-101. Springer, 1999.

[12] T. A. Halpin, *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufman, San Francisco, 2001.

[13] K. Henricksen and J. Indulska, *Modelling and Using Imperfect Context Information*. 1st Workshop on Context Modeling and Reasoning, PerCom'04 Workshop Proceedings, IEEE Computer Society, March, 2004. pp. 33-37.

[14] K. Henricksen, J. Indulska and A. Rakotonirainy, *Generating Context Management Infrastructure from Context Models*. 4th International Conference on Mobile Data Management (Industrial Track), January, 2003. pp. 1-6.

[15] K. Henricksen, *A Framework for Context-Aware Pervasive Computing Applications*. PhD thesis, School of Information Technology and Electrical Engineering, The University of Queensland. September, 2003.

[16] A. K. Dey and G. D. Abowd, *CybreMinder: A Context-Aware System for Supporting Reminders*. 2nd International Symposium on Handheld and Ubiquitous Computing. In: Lecture Notes in Computer Science, Volume 1927, pp. 172-186. Springer, 2000.

[17] K. Cheverst et al., *Using Context as a Crystal Ball: Rewards and Pitfalls*. Personal and Ubiquitous Computing, 5(1): pp. 8-11, 2001.

[18] H. E. Byun and K. Cheverst, *Harnessing Context to Support Proactive Behaviours*. ECAI2002 Workshop on AI in Mobile Systems, Lyon, July, 2002.

[19] T. F. Paymans, J. Lindenberg and M. Neerincx, *Usability Trade-offs for Adaptive User Interfaces: Ease of Use and Learnability*. 9th International Conference on Intelligent User Interfaces, ACM Press, 2004. pp. 301-303.

[20] M. Nilsson, J. Hjelm and H. Ohto, *Composite Capabilities/Preference Profiles: Requirements and Architecture*. W3C Working Draft, 21 July, 2000.

[21] R. Agrawal and E. L. Wimmers, *A Framework for Expressing and Combining Preferences*. ACM SIGMOD Conference on Management of Data, Dallas, May, 2000. pp. 297-306.

[22] A. Newberger and A. Dey, *Designer Support for Context Monitoring and Control*. Technical report IRB-TR-03-017, Intel Research Berkeley. June, 2003.

[23] P. J. Brown, *The Stick-e Document: a Framework for Creating Context-Aware Applications*. Electronic Publishing, Palo Alto, 1996. pp. 259-272.

[24] B. Segall et al., *Content Based Routing with Elvin4*. AUUG2K Conference, Canberra, June, 2000.

[25] P. Gray and D. Salber, *Modelling and Using Sensed Context Information in the Design of Interactive Applications*. 8th IFIP International Conference on Engineering for Human-Computer Interaction. In: Lecture Notes in Computer Science, Volume 2254, pp. 317-336. Springer, 2001.

[26] T. McFadden, K. Henricksen and J. Indulska, *Automating Context-aware Application Development*. UbiComp 1st International Workshop on Advanced Context Modelling, Reasoning and Management, September, 2004. pp. 90-95.

[27] T. H. Tse et al., *Testing Context-Sensitive Middleware-Based Software Applications*. 28th Annual International Computer Software and Applications Conference, IEEE Computer Society, 2004.

[28] M. Bylund and F. Espinoza, *Testing and demonstrating context-aware services with Quake III Arena*. Communications of the ACM, 45(1): pp. 46-48, 2002.

[29] T.McFadden, K. Henricksen, J. Indulska and P. Mascaro, *Applying a Disciplined Approach to the Development of a Context-Aware Communication Application*. 3rd IEEE Conference on Pervasive Computing and Communications, Hawaii, March, 2005. pp. 300-306.

[30] J. Indulska, K. Henricksen, T. McFadden and P. Mascaro, *Towards a Common Context Model for Virtual Community Applications*. 2nd International Conference on Smart Homes and Health Telematics (ICOST). In: Assistive Technology Research Series, Volume 14, pp. 154-161. IOS Press, 2004.