1 Context-Awareness

JADWIGA INDULSKA and KAREN HENRICKSEN

School of Computer Science and Electrical Engineering,

The University of Queensland and National ICT Australia (NICTA),

jaga@itee.uq.edu.au, Karen.Henricksen@nicta.com.au

1.1 INTRODUCTION

As shown in the previous chapters, the range of pervasive computing technology available to support the aging and disabled population continues to evolve, allowing for an increasing variety of wireless sensors, devices, and actuators to be deployed to produce assistive 'smart spaces'. Software applications embedded in smart spaces can intelligently assist elderly or disabled people in a variety of tasks. They can support an individual's healthcare needs and activities of daily living, while extending social interaction, environment control, and information flow to family and care givers. This needs to be achieved without compromising an individual's medical care, privacy, or security, and should allow an individual to maintain maximal independence. Variations in cognitive and physical capability also need to be considered.

Software applications for smart spaces require a degree of autonomous behavior in order to adapt to changes in operating conditions (for example, to changes in sensed data such as data about the user's current location, differences in user profiles, or changes in user priorities). Such adaptive applications are called context-aware applications, as they are able to monitor and evaluate the context in which they operate and adapt if the context changes. Due

to the variety of user activities and capabilities which should be intelligently supported, and the heterogeneity of sensors, computing devices, communication technologies and interaction protocols between devices, these applications require a rich set of context information.

In this chapter, we explore the use of context information to enable a wide variety of context-aware applications, including applications supporting independent living of elderly or disabled people. First, we explore definitions of context information and discuss its role in context-aware applications, both in general and in smart technologies for aging/disability. We also discuss the importance of formal context models and the requirements that such models should meet, and provide an overview of various approaches for modeling context information. Finally, we discuss the use of software infrastructure to support context-aware applications by assisting with tasks such as gathering and evaluating context information.

1.2 CONTEXT DEFINITIONS AND EXAMPLES

The term context is loaded with a wide variety of meanings. Various areas of computer science differ in their understandings of context, but even in the research community working on context-aware adaptive applications there is no consensus. Dey (2001) presents a survey of alternative views of context, which are largely imprecise and indirect, typically defining context by synonym or example. Common examples of context are location, time, temperature, noise level, user activity, and a plethora of information related to the computing environment, including computing devices and their characteristics, network connectivity, communication bandwidth and so on.

Dey also offers the following definition, which is now widely (but not universally) accepted in the field:

> Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is

> considered relevant to the interaction between a user and an application, including the user and applications themselves.

The meaning of "the situation of an entity" is never precisely defined by Dey, but instead is illustrated through simple examples (e.g., presence of other people near the user in the case of a mobile tour guide application). Situations can be either simple (e.g., determined from a single context fact that describes where a particular person is located) or complex (constructed from many context facts, as in the case of a situation involving a person lying motionless in the middle of a room, which is inferred from the readings of several sensors).

One of the reasons for the lack of consensus and precision in context definitions is the lack of clear separation between the concepts of *context*, *context modeling* and *context information*. While context is difficult to define, context models and information are well defined and understood. The latter two are of primary interest when constructing context-aware systems. We use the following definitions:

- The *context* of a computing application (such as a patient monitoring application) is the set of circumstances surrounding it that are potentially relevant to its execution.

- A *context model* identifies a concrete subset of the context that is realistically attainable from sensors, applications and users, and able to be exploited in the execution of the application. The context model that is employed by a given context-aware application can be specified by the application developer (although it need not be fixed at design time, but can instead evolve as required).

- *Context information* is a set of data, gathered from sensors and users, that conforms to a context model. This provides a snapshot that approximates the state, at a given point in time, of the subset of the context encompassed by the model. Context information often takes the form of a set of context facts.

Context information can be gathered from a variety of sources. These include:

- *Sensors*, including sensors embedded in the home or carried by occupants. Examples of the former are wall-mounted microphones or cameras, while the latter category includes positioning devices such as location badges. Sensors may also be logical rather than physical, as in the case of a software sensor that monitors activity on a computer according to the frequency of keystrokes.

- *Profiles* describing capabilities of hardware devices or preferences of users.

- *Applications* that report their current state, such as whether they are actively being used and for what task, for use in adaptation decisions by other applications.

- *Data fusion/interpretation services*, which combine or reason about context information to derive higher-level information, including situations of the kind described earlier.

Context sources of these types produce information of varying degrees of quality, with varying update/refresh rates. Issues related to the quality of information provided by context sources will be discussed later in Section 1.5.1.

We define a *context-aware application* as a computing application that uses context information in order to automatically adapt its behavior to match the situation. In the same fashion, a *context-aware system* is a computing system (e.g., one or more context-aware applications and supporting hardware and software) that exhibits similar adaptive capabilities. Context-aware applications are reactive applications - that is, they invoke adaptation actions in response to context changes. However, as well as being reactive, they should be pro-active in anticipation of user needs.

1.3 ROLE OF CONTEXT IN SMART TECHNOLOGIES FOR AGING, DISABILITY AND INDEPENDENCE

Context-aware applications can intelligently support users in a variety of tasks, including tasks that promote independent living of elderly or disabled people. Context-aware applications can create smart home environments, provide health-care services and also support user tasks. Their functionality may range from hiding heterogeneity of devices and communication technologies (e.g., by supporting seamless behavior regardless of device or network changes) to more complex support for independent living of the elderly/disabled. The latter may include seamless control of sensors and appliances, support for daily activities (for example, by facilitating remote help from family or local community members), health monitoring, and social interactions. Below we describe some common classes of application for independent living.

Flexible communication. Such applications provide instant communication with a family member, friend or health worker, achieved using context-aware choice of communication channel (telephone, SMS, etc.) according to the current activity, available communication devices, and preferences of the communicating people.

Support for social interactions and virtual communities. Such applications have diverse goals, but are primarily concerned with providing support for independent living in communities of elderly people and may include:

- assistance with everyday tasks by remote family or community members;
- dynamic formation of groups of people who are interested in activities such as shopping or playing chess based on their preferences and availability; and
- awareness of the activities of others in nearby homes using abstract visual representations, helping to minimize isolation and provide assurances of others' well-being.

Smart spaces. Such spaces can provide automatic configuration and reconfiguration of assistive devices and home appliances in order to adapt their functionality/behavior to user needs and preferences.

Multi-modal assistive technologies and home appliances. Context-sensitive adaptive interfaces can assist individuals by providing alternate modes of interaction with common devices such as phones, microwaves, and televisions.

Health care applications. These applications can provide health monitoring, accident monitoring, behavioral trend monitoring and cognitive health monitoring. Other health care applications provide reminders for eating, taking medications, and a variety of other tasks and activities.

1.4 DEVELOPING CONTEXT-AWARE APPLICATIONS AND SYSTEMS

Although context-aware applications have the potential to support independent living in innovative ways, as shown by the example applications just discussed, current deployments are limited. One of the challenges inhibiting the development and deployment of context-aware applications is their complexity. Context-aware systems involve a variety of cooperating hardware and software components, including sensors, additional devices such as actuators and user input/output devices, and various pieces of application software, all communicating via wired or wireless networks.

In early context-aware systems, each application was directly responsible for communicating with sensors, interpreting the sensor outputs to determine the context, and adapting itself (or the environment, via actuators). An example application that follows this approach is shown in Figure 1.1. The application provides monitoring of activity in the home, and generates alerts when abnormal circumstances arise. The application can accept explicit input from the user via a ‘panic button’, which forces an alert to be produced. How-

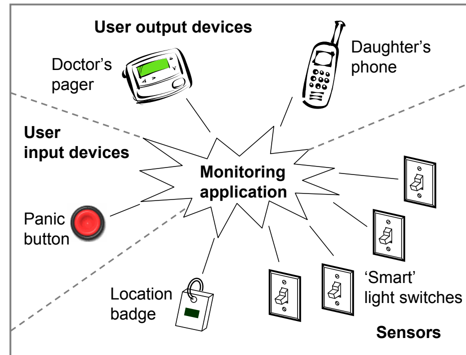

Fig. 1.1 Architecture of a monitoring application for independent living.

ever, the application also continuously tracks the user's activity via sensors embedded in the home or carried by the user. These include:

- a location badge (i.e., an Active Bat (Harter et al., 2002) or similar device), which can estimate the user's position in the home; and
- 'smart' light switches, which can report their current status (i.e., on or off).

As the application has been developed in a naïve way, without using an explicit context model or supporting software infrastructure (both of which will be discussed further later in this section), it communicates directly with the sensors and processes the raw sensor outputs to determine when alerts are required. This entails remote communication using a variety of networking protocols, polling sensors at regular intervals, handling sensor errors, and interpreting sensor outputs (for example, mapping positioning information to symbolic locations, such as names of rooms in the house). In addition, the application incorporates algorithms that learn patterns of 'normal' versus 'abnormal' behavior. When abnormal behavior is detected, such as the user remaining motionless for a long time or not switching lights off at the usual time, the application sends alerts to appropriate output devices, such as a family member's mobile phone or (in serious cases) a doctor's paging device.

In this example system, the software used to interface with the sensors and the algorithms used to interpret the sensor outputs are directly incorporated into the application software, making these components difficult to modify, extend, share or reuse. For example, adding new hardware, such as embedded microphones for enhanced activity tracking, involves substantially extending the application software, while developing a second 'smart home' application, such as an appliance controller that switches devices on or off based on the user's location and activity, involves duplicating part of the monitoring functionality. Moreover, applying this software engineering approach to applications that involve many types of sensors or sophisticated interpretation algorithms can easily yield large and complex pieces of software.

For these reasons, context-aware applications are today developed using a variety of abstractions and supporting pieces of software infrastructure that can be shared between applications. Typically, these include explicit, high-level context models, which provide a uniform and integrated description of context information gathered from various sources, as well as software components that populate the context models by gathering and interpreting information on behalf of applications. Figure 1.2 shows how these components modify the architecture of the monitoring system described earlier. In the new architecture, the application queries a high-level context model, stored in a context repository, to determine the user's symbolic location and the status of all light switches in the house, rather than obtaining separate low-level information from each of the sensors. The interpretation of the sensor outputs occurs in separate software components which feed information into the context repository. These components, together with the context repository, make up the software infrastructure that supports the application. This architecture eliminates the need for the application software to handle sensor errors or implement multiple networking protocols for communicating with sensors. In addition, the context model can be extended when new sensors are added to the home, and can be shared by any number of context-aware applications. Note that, in this case, the software infrastructure is quite sim-

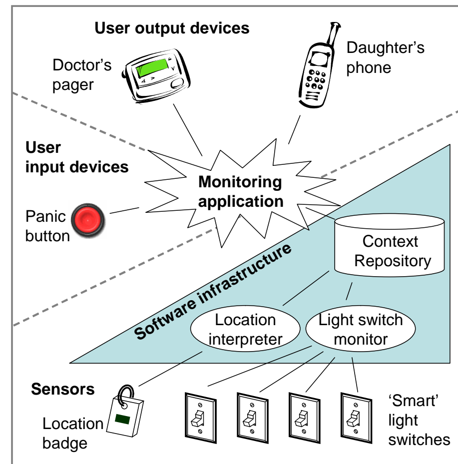

Fig. 1.2 Architecture of a monitoring application for independent living when a context model and basic software infrastructure are incorporated.

ple. More sophisticated forms of infrastructure will be described later in this chapter.

The following sections review the current state-of-the-art in context modeling (Section 1.5) and software infrastructure for context-aware systems (Section 1.6).

1.5 CONTEXT MODELING

Context modeling aims to produce a formal or semi-formal description of the context information present in a context-aware system. As discussed in Section 1.2, this context information can include information derived from sensors as well as a variety of other sources, such as context-aware applications and user profiles. By providing a uniform description of types of context information, as well as run-time instantiations of the types (context facts), context information from various sources can be easily combined, queried, and reasoned about. This promotes sharing and exchange of information between applications, and provides information representations that are straightfor-

ward for applications to process compared to other formats such as streams of raw sensor output.

A variety of context modeling approaches have been recently proposed. These vary according to the prior technologies or standards with which they are aligned and the aspects of the software engineering process they address. For example, some modeling approaches are aligned with World Wide Web Consortium (W3C) standards such as the Web Ontology Language (OWL) (McGuinness and van Harmelen, 2004), while others are based on database modeling techniques. Similarly, some aim to support requirements analysis - i.e., identification of which kinds of context information are needed for a particular context-aware application, and with what level of quality - while others are intended for run-time representation and reasoning. This section provides a discussion of some general requirements for context modeling approaches, followed by a survey of some of the approaches that have been proposed.

1.5.1 Requirements

In order to be broadly applicable to a variety of context-aware applications and to serve as useful abstractions for software engineering as discussed in Section 1.4, context modeling approaches should meet the following requirements.

1.5.1.1 Support for imperfect context information A common problem in context-aware systems is the presence of imperfect context information. For example, problems with sensor-derived information can arise as a result of sensor failures, power shortage, noise in the environment, faulty sensor installation, or inaccuracy in the algorithms used to abstract context information from sensor outputs. Similarly, user- or application-supplied information can be subject to problems such as staleness.

Consequently, when modeling context, it is necessary to be able to represent:

- information that is incomplete;
- information that is imprecise;

- information that is ambiguous (for example, conflicting location reports provided by different location sensors); and
- quality indicators (information source, timeliness, granularity, sensor accuracy, etc.) for information that may be imprecise or erroneous. These can be used by context-aware applications to determine when to trust the available context information, and can also be used to trace faulty context information back to malfunctioning sensors, which can then be repaired or replaced.

Applications should be able to effectively query and reason about the available context information, so as to continue operating satisfactorily even when that information is incomplete, imprecise, ambiguous or otherwise imperfect.

1.5.1.2 Support for context histories Context-aware applications often require not only information about the current context, but also past or future contexts. Therefore, context modeling techniques must provide natural ways for modeling histories of information, and applications should be capable of querying and reasoning over these histories. Histories can be used to detect patterns in user behavior and predict future requirements, as seen in the monitoring application described in Section 1.4.

1.5.1.3 Support for software engineering A key role of context models is to simplify and introduce greater structure into the task of developing context-aware applications. The greatest benefit is often derived when a formal or semi-formal context model is introduced early in the software engineering life-cycle and refined incrementally over the life-cycle. Early in the life-cycle, a context model should be produced that sets out the types of context information required by the application, as well as constraints on the data, such as information quality and privacy requirements. This model can be used to evaluate the suitability of any context sensing infrastructure that is already in place and to identify any additional hardware and software requirements. In addition, it can be used to inform the design and implementation of the application.

The model can also be refined to produce a run-time context model that can be populated by various sources of context information and queried by applications. This is the model that resides in the Context Repository shown in Figure 1.2. Additionally, context models can be used as the basis for generating test cases to allow systematic testing of context-aware functionality (Henricksen and Indulska, 2006).

There are currently no context modeling approaches that address all of these software engineering issues, although some address one or more of the issues. The majority of the approaches, however, are concerned with run-time context representation, querying and reasoning, not on requirements analysis, design or testing.

1.5.1.4 Support for run-time querying and reasoning One of the most important forms of context model is the run-time model. This model is typically stored in one or more context repositories and queried by context-aware applications. Unlike context models used for analysis and design purposes, run-time models must address representational issues - that is, how to represent information at run-time so that it can be efficiently stored in a repository, queried, and reasoned over to support decision making by context-aware applications about how to react to context changes. For example, the run-time context model for the monitoring application described earlier must be capable of representing histories of location information and light switch 'on/off' events that can be used for learning patterns and reasoning about which sequences of location changes and switch events are abnormal.

The run-time model should incorporate information about the known *types* of context and their characteristics (metadata), as well as concrete pieces of context information gathered from various sources (context instances or facts). Querying and reasoning over both kinds of information should be possible. The run-time model should also be easily extensible to new types of context information. The ability for context-aware applications to query and reason

about the currently known context types (the model metadata) helps them to tolerate evolving context models and exploit newly available information.

Although the example context-aware system presented in Section 1.4 (Figure 1.2) involves only a single context repository, more complex systems may contain context information that is distributed amongst a large number of context repositories. The context representation used by the run-time context model (as well as the query and reasoning mechanisms) must therefore be able to support this type of distribution.

1.5.1.5 Support for interoperability Context-aware applications may be required to cooperate at run-time with components that were not known to the application designer, such as new applications or sensing hardware. They should be able to exchange context information with these components, which requires a form of interoperability. This can entail:

- supporting transformations between alternative representations of the same information, using a shared context modeling approach (for example, mapping between different units of measurement, or mapping one term to another equivalent term); or
- supporting transformations between different modeling approaches (for example, mapping from an ontology-based approach to an approach based on database modeling techniques).

The first problem is reasonably straightforward, provided that the modeling approach provides a way to define the required mapping rules. The second problem is more challenging. If a given pair of modeling approaches differ in terms of their expressive power, complete transfer of information between them may not be possible.

An important way to address interoperability is through standardization. By introducing standard ways to represent context information (i.e., common modeling approaches), as well as standard vocabularies and concepts for describing instances of context information, it is possible to avoid the problem of mapping between different context representations. Unfortunately, there are

currently no widely accepted standards for context modeling, although some standardization work has been carried out for particular application domains. Indulska et al. (2004) have carried out some early work on the development of a common context model to support independent living applications for the elderly.

1.5.2 Markup scheme approaches

The remainder of Section 1.5 provides a survey and analysis of a variety of context modeling techniques that are in use today. One of the earliest modeling approaches built on the popularity of markup schemes such as XML (Bray et al., 2004). A well-known example that typifies this approach is CC/PP (Composite Capability/Preference Profiles) (Klyne et al., 2004), which was standardized as a W3C recommendation during 1998-2004. CC/PP aims to support the transfer of simple context information and preferences - such as device characteristics and users' language preferences - from Web browsers to servers, in order to support dynamic adaptation of the Web pages returned by the servers to browsers. For example, information about the screen size of the user's device can be used to resize or remove images to fit.

CC/PP is based on RDF (Manola and Miller, 2004), a framework for representing information in a common graph-based format capable of representing resources and their properties. CC/PP is intended to be used as a run-time model (i.e., it does not address software engineering tasks such as requirements analysis). It principally addresses the transfer of information between software components (typically, between Web servers and clients); however, storage and querying of CC/PP information is also supported by a variety of tools (both RDF and special-purpose CC/PP tools). CC/PP supports little in the way of reasoning, as it is traditionally used only for representing simple types of information about which reasoning is not necessary.

An example snippet of an XML-encoded CC/PP profile is shown in Figure 1.3. This example shows a description of device hardware (specifically,

```
<ccpp:component>
  <rdf:Description rdf:about="http://mydomain.com/TerminalHardware">
    <rdf:type rdf:resource=
"http://www.wapforum.org/profiles/UAPROF/ccppschema-20010430#HardwarePlatform"/>
    <prf:ScreenSize>320x200</prf:ScreenSize>
    <prf:ColorCapable>No</prf:ColorCapable>
    <prf:ImageCapable>Yes</prf:ImageCapable>
  </rdf:Description>
</ccpp:component>
<ccpp:component>
  <rdf:Description rdf:about="http://mydomain.com/TerminalSoftware">
    <rdf:type rdf:resource=
"http://www.wapforum.org/profiles/UAPROF/ccppschema-20010430#SoftwarePlatform"/>
    <prf:OSName>EPOC</prf:OSName>
    <prf:OSVendor>Symbian</prf:OSVendor>
    <prf:OSVersion>1.0</prf:OSVersion>
  </rdf:Description>
</ccpp:component>
```

Fig. 1.3 An excerpt from an example CC/PP profile, describing device hardware and software.

the display capabilities) and software (operating system name, vendor and version).

Since CC/PP was initially proposed, various extensions have appeared, including CC/PP vocabularies such as the Open Mobile Alliance's User Agent Profile (UAProf) (Open Mobile Alliance, 2003) and other independent extensions for describing relatively advanced types of context, such as location, network characteristics, application requirements, sessions, and constraints on properties (Indulska et al., 2003). In addition, a similar RDF-based proposal called Comprehensive Structured Context Profiles (CSCP) (Buchholz et al., 2004) was developed to provide greater expressive power than CC/PP. CSCP lifts some of the limitations imposed by CC/PP on the structure of the profiles, bringing the expressiveness closer to the original expressive power of RDF. However, with the growing popularity of ontology standards that are capable of describing more sophisticated concepts than the markup schemes described here, such as equivalence and cardinality constraints - and thereby supporting reasoning over these concepts for purposes like consistency checking - the popularity of simple modeling approaches along the lines of CC/PP and CSCP is waning.

1.5.3 Ontology-based approaches

The distinction between the markup scheme approaches covered in the previous section and the approaches discussed in this section, which are classed as *ontology-based*, is useful but somewhat artificial. An ontology may be viewed as a comprehensive and rigorous description of a given domain, defining important terms or concepts, as well as relationships between these. However, confusion arises because this definition encompasses most of the context modeling approaches discussed here, yet they are not all classed as ontology-based. In addition, ontology standards are often encoded using markup languages such as XML, in a similar manner to CC/PP and CSCP.

To clarify, therefore, this section addresses context modeling approaches that are aligned with recent work in ontology language standardization - specifically, OWL (McGuinness and van Harmelen, 2004) and closely related precursors such as DAML+OIL (Horrocks, 2002). These modeling approaches are more sophisticated than the markup scheme approaches described in the previous section, in that they support additional concepts, such as set operators for defining classes (union, intersection, etc.), cardinality constraints on properties, and equivalence between pairs of classes or properties. Importantly, they are also based on logical formalisms that support reasoning (although the same may actually be said of RDF, on which both CC/PP and CSCP are based). Like the markup scheme approaches, the ontology-based approaches focus on run-time context modeling, not software engineering issues like analysis of required context types, their quality, and other similar characteristics.

One of the earliest ontology-based proposals, by Strang et al. (2003), was the Context Ontology Language (CoOL). CoOL structures context information according to an *Aspect-Scale-Context* (ASC) model. *Aspects* are kinds of context, such as position or temperature. Each aspect has one or more *scales*, corresponding to units of measurement. Mappings between pairs of scales within a given aspect must be defined. Each piece of context information belongs to a given scale within an aspect, and characterizes a particular *entity*.

Context information may also have associated *quality information*, which is itself context information with its own scale and aspect.

Strang et al. provide mappings of the ASC model into the OWL, DAML+OIL and F-Logic languages, and use the OntoBroker tool for reasoning about context. The types of reasoning that can be performed using this approach include reasoning to validate the consistency of a context model, map information between context models using inter-ontology relationships, and 'complete' an ontology by computing implicit relationships, such as subclass relationships.

The main strengths of CoOL are that it provides one form of support for interoperability (through defined mappings between different units/scales) and supports reasoning using widely available tools. However, this reasoning does not take into account imperfect context information (e.g., quality indicators or ambiguity).

CoOL differs from other work in ontology-based context modeling in that it introduces core context modeling constructs (aspects, scales, etc.) that are separate to the underlying ontology languages used (OWL and DAML+OIL). Wang et al. (2004) and Chen et al. (2005) take a different approach. They use the standard OWL constructs, but focus instead on creating extensible domain ontologies that define standard concepts/vocabularies that can be used for describing context.

Wang et al. propose the use of two levels of ontology to capture general concepts and domain-specific concepts, respectively. General concepts, which include entities such as locations, people, activities, and computing applications and services, are defined via a common upper ontology. The upper ontology can be extended by various domain ontologies, which define concepts relevant to specific environments such as homes or offices.

Wang et al. show that it is possible to use reasoning to derive high-level context (e.g., the current activity of the user) from lower-level context (e.g., location information derived from sensors). This requires the definition of appropriate axiomatic rules. However, OWL does not support such rules, so

they are instead represented in a separate (non-standard) format. Further, Wang et al.'s original proposal does not address quality of context information. Their later work (Gu et al., 2004) introduces extensions for modeling the derivation of context information (sensed, derived, aggregated or deduced) and relevant quality indicators. However, this work requires non-standard extensions to OWL, and therefore is not supported by OWL tools.

The work of Chen et al. (2005) is similar to that of Wang et al. in its aims, but broader in scope. Chen et al. propose a set of OWL ontologies, collectively referred to as SOUPA, that address numerous modeling issues related to context-aware applications, including context modeling, modeling of concepts from the field of intelligent agents, such as roles, beliefs and intentions, and modeling of privacy policies for controlling access to sensitive information. SOUPA builds on a variety of well-known ontologies, such as Friend-Of-A-Friend (Brickley and Miller, 2005) and DAML-Time (Hobbs et al., 2002). SOUPA also incorporates an earlier set of ontologies called COBRA-ONT, which were developed by Chen et al. (2004b) for modeling context information in smart meeting rooms.

The principal aim of SOUPA is to provide a broad coverage of common context concepts, so that application developers rarely need to define their own. This promotes interoperability. As in Wang et al.'s work, a distinction is made between core concepts (defined by the SOUPA Core ontologies) and additional domain-specific concepts (defined by the SOUPA Extension ontologies).

The SOUPA ontologies support basic types of reasoning. For example, the ontologies that deal with location and time define relationships that allow reasoning to derive information such as the current occupants of a building, or the precedence/ordering of a given set of events. However, the ontologies provide minimal support for imperfect context information. Conflicting context information can be detected via reasoning and resolved manually by users; however, the ontologies do not provide any mechanisms for representing ambiguous information or quality of information.

Ranganathan and Campbell (2003) propose a very different approach that is based primarily on first order logic, but also makes use of simple ontologies expressed using DAML+OIL. They represent context information in terms of predicates, such as "Temperature(room 3231, "=", 98F)" and "Location(chris, entering, room 3231)" (Ranganathan and Campbell, 2003). The role of the DAML+OIL ontologies is to define the structure of the predicates and the types of the arguments. The ontology definitions can be used to check the validity of predicates, and also as a basis for defining mappings between different predicates, in order to support interoperability.

Context predicates can be combined to form complex logical expressions using the operators 'and', 'or' and 'not', and the universal and existential quantifiers. As there are decidability and safety problems associated with reasoning over unconstrained first order logic expressions, Ranganathan and Campbell adopt a many-sorted logic. This means that quantified expressions, such as "chris is in some location" (written $\exists_{Location} x$ Location(chris, in, x)) are evaluated only over a finite set of objects known to the system. In the example expression, this implies that only a fixed set of locations are considered as possible values for x. This ensures that evaluations always terminate, but can still introduce serious performance problems when the sets of known objects are large. For this reason, this approach for context modeling and reasoning is most appropriate for use in small smart spaces, such as meeting rooms.

Although Ranganathan and Campbell show that it is possible to store histories of predicates and corresponding timestamps in a database, their many-sorted predicate logic does not provide any special support for querying historical information. Additionally, this modeling approach does not address modeling of, or reasoning over, imperfect context information. However, as Ranganathan and Campbell point out, it is possible to layer various reasoning mechanisms, such as temporal or fuzzy logics, on top of their predicate-based representation of context. Another context modeling approach which incorporates a somewhat similar logic-based reasoning approach, but handles certain

types of imperfect context information using a three-valued logic, will be discussed in Section 1.5.5.

1.5.4 A requirements analysis model for sensed context

The context modeling approaches discussed so far address the run-time representation of context information, and, to varying degrees, run-time context querying and reasoning. In contrast, the final two approaches that will be covered are predominantly concerned with software engineering issues, focusing on structured or semi-structured techniques to support the analysis and design of context-aware applications.

Gray and Salber's proposed model for analyzing sensed context information (Gray and Salber, 2001) is considerably less formal than the modeling approaches covered thus far. It aims to support a systematic evaluation of the types of context information needed by a context-aware application and the required/anticipated characteristics of this information. Its focus is not on the constructs/concepts used to represent and reason about context information, but rather on properties of the information that are relevant from the application development perspective, such as the data representations, the information quality, the source of the information, and the set of transformations required between the source (sensor) and the high-level data representation. Gray and Salber refer to these properties of the context information as meta-attributes. They argue that the software engineering process (and its end result) can be improved by rigorously analyzing the possible design choices for these meta-attributes.

Gray and Salber provide a checklist of information quality aspects that should be considered - namely, coverage of sensors, resolution of sensor output, accuracy, repeatability/stability, frequency and timeliness. They also describe a set of related issues that must be considered in relation to particular sensing technologies - reliability, intrusiveness, security/privacy and cost. Finally, Gray and Salber relate their identified set of context meta-attributes to

the software engineering process by recommending a simple design approach consisting of the following activities (from (Gray and Salber, 2001)):

- *identifying sensed context possibilities*
- *eliciting and assessing information quality requirements*
- *eliciting and assessing requirements of the acquisition process*
- *consideration of issues of*
  - *intrusiveness, security, privacy*
  - *transformations of the data from source to "consumer"*
  - *transmission and storage*
- *eliciting and assessing sensor requirements*

Gray and Salber's model provides a good basis for analyzing the context requirements of a context-aware application and helping to ensure that potential problems, such as information quality or privacy problems, are identified as early as possible in the application development process. However, it lacks the formality to support a straightforward mapping of the context types identified at the analysis/design stage to a context model that can be stored, queried and reasoned over at run time. The following section describes a more formal context modeling approach that supports elements of analysis and design, as well as mapping to a run-time context model.

1.5.5 The Context Modelling Language

Henricksen et al. propose a context modeling approach that supports incremental development of context models, beginning during the requirements analysis and design phases of the software engineering process, and continuing through to application deployment, execution and beyond (Henricksen et al., 2005a; Henricksen and Indulska, 2006). This approach builds on the Object-Role Modeling (ORM) technique (Halpin, 2001), which is traditionally used for database modeling. ORM uses a graphical notation for creating a diagrammatic representation of relevant concepts and relationships between the concepts. In the terminology of ORM, concepts are known as entities and

relationships as fact types. Concrete instances of relationships (e.g., "Fred is located in Room 633") are known as facts.

As ORM is designed for database modeling, not context modeling, it lacks powerful ways to describe relevant meta-attributes of context, such as sources of information and quality attributes. For this reason, Henricksen et al. extended ORM with a number of special-purpose context modeling constructs (Henricksen et al., 2005a), originally introduced in an earlier context modeling notation (Henricksen et al., 2002). This extended variant of ORM is known as the Context Modelling Language (CML).

A simple example model specified using the CML notation is shown in Figure 1.4. This model captures types of context relevant to the monitoring application that was discussed in Section 1.4. It incorporates various types of user-supplied information (so-called 'profiled fact types'), including information about the occupants of a home, the rooms in the home, the light switches in each room, and assignments of location badges to particular people. In addition, the model contains two types of sensor-derived information ('sensed

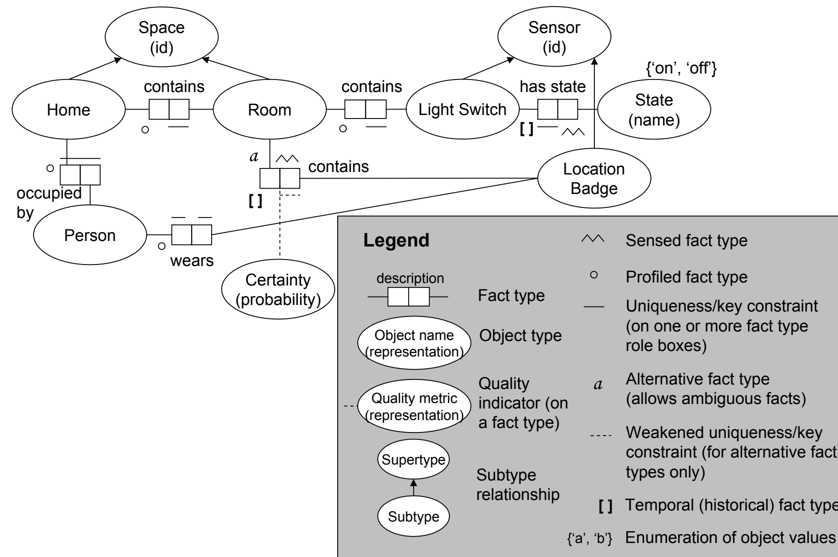

Fig. 1.4 Example context model for the monitoring application discussed in Section 1.4.

fact types'): estimated positions of location badges in the home to the nearest room, and status information for light switches. Both sensed fact types are marked as 'temporal' fact types, implying that histories of facts, marked with timestamps, should be retained in the context repository at run-time. For the former information type (badge location), uncertainty is allowed by marking this as an 'alternative' fact type. This allows two or more contradictory facts - such as "bedroom 1 contains badge 343" OR "hallway 1 contains badge 343" - to be represented whenever the positioning system is unable to more accurately resolve the location of a badge. In addition, each location fact is associated with a probability.

ORM, the database modeling approach on which CML is based, is well established as a requirements analysis technique, and therefore much has been written about the process of constructing ORM models in cooperation with experts in the domain that is being modeled and/or intended database/application users. This process can be adapted to provide guidelines for analyzing context requirements and constructing a context model using CML. In addition, there is a straightforward mapping of ORM models to relational databases. This mapping procedure can be extended to allow mapping of CML models to run-time context models stored in context repositories that take the form of enhanced relational databases supporting specialized context meta-attributes and constraints. The run-time models can be queried using either standard relational database query languages, or by evaluating pre-defined 'situations' expressed using a form of predicate logic. Situations provide basic support for evaluating ambiguous context (such as the conflicting location facts discussed above) using a three valued-logic (where situation expressions can be 'true', 'false' or 'possibly true'); in addition, situations can incorporate special forms of existential and universal quantification, in which variables are constrained by binding them according to a fact template, thereby ensuring efficient and safe evaluation.

Requirement	Markup schemes	Ontology-based approaches	Sensed context (analysis) model	CML
Support for imperfect context information	✓[a]	✓[a]	√	√
Support for context histories	✓[b]	✓[b]	×	√
Support for software engineering	×	×	√	√
Support for runtime querying and reasoning	✓	√	×	√
Support for interoperability	✓[c]	√[d]	×	×

Table 1.1 An analysis of the context modeling approaches discussed in Sections 1.5.2 to 1.5.4 with respect to the requirements introduced in Section 1.5.1.
(Key: √ = comprehensive support, ✓ = partial support, × = no support)

[a]Imperfect information can usually be represented in some form (although rarely in a very natural way), but reasoning over imperfect information is not supported by conventional tools.
[b]Can be represented, but the majority of the approaches do not define natural concepts/vocabularies for doing so.
[c]Based on the use of standard vocabularies.
[d]Based on the use of standard vocabularies and defined mappings between concepts.

1.5.6 Analysis

This survey of context modeling approaches is intended to be illustrative rather than exhaustive; however, it covers most of the well-known work in the area. Owing to the immaturity of the field of context-awareness, none of the modeling approaches has been widely adopted. CC/PP has the advantage of being standardized as a W3C Recommendation, but it is unsuitable for representing many types of information (Indulska et al., 2003), and its support for reasoning is limited. OWL-based modeling approaches are better able to support reasoning and are currently enjoying favor. However,

these predominantly address run-time issues (representation of context information, reasoning and interoperability), not software engineering tasks such as requirements analysis and design. Therefore, models of the sort presented in the latter part of the survey also play an important role. CML, discussed in the very last section of the survey, offers the particular advantage that it supports the mapping of a requirements model to a run-time model.

Table 1.1 presents a summary of the strengths of the various approaches. As none of the approaches is comprehensive in the sense that it addresses all of the requirements introduced in Section 1.5.1, hybrid modeling approaches are being considered - for instance, approaches that take elements from mature database modeling techniques, which support requirements analysis as well as efficient run-time representation and querying, and augment them with descriptions written in ontology languages, so as to enable interoperability through constructs such as equivalence definitions and concept mappings (Henricksen et al., 2004; Becker and Nicklas, 2004).

1.6 SOFTWARE INFRASTRUCTURE FOR CONTEXT-AWARE SYSTEMS

Creating models of context requirements, as well as run-time context models, are important steps in developing context-aware applications. An equally important requirement is gathering the context information specified by the model from various sources at run time. This task is usually delegated to a software infrastructure shared by a number of context-aware applications; however, the infrastructure may also handle other related tasks. This section discusses the components and functionality that may be found in a software infrastructure for context-aware systems, and then introduces some representative examples of infrastructure.

1.6.1 Reference architecture for context-aware systems

When discussing software infrastructure for context-aware systems, it is instructive to have a typical - or reference - architecture in mind, with which various solutions can be compared. Figure 1.5 shows such a reference architecture. This architecture incorporates various components that may be present in current context-aware systems - however, it is important to note that some of the components will be absent in many systems. This is either because the systems are sufficiently simple that the components are not required (for example, the monitoring application discussed in Section 1.4 does not require actuators), or because the functionality of some components is incorporated directly into the applications (as in the initial design of the monitoring system that was shown in Figure 1.1). In addition to the application components, sensors and actuators, shown at the two extremities in Figure 1.5, the reference architecture contains the following infrastructural components:

- components that: (i) assist with discovering suitable context sources (most commonly, sensors) and processing the data produced by these sources to populate applications' run-time context models[1]; and (ii) map update operations on the models back down to actions on actuators when required (layer 1);
- context repositories that provide persistent storage of run-time context models, query and update facilities, and optional support for reasoning over context (layer 2); and
- decision support tools that help applications to select appropriate actions and adaptations based on the information in the run-time context models (layer 3) .

Programming toolkits are often also incorporated at the application layer (layer 4) to support the interactions of the application components with the infrastructural components. One responsibility of these toolkits is to handle

[1]Note that other context information may be directly inserted by users or applications.

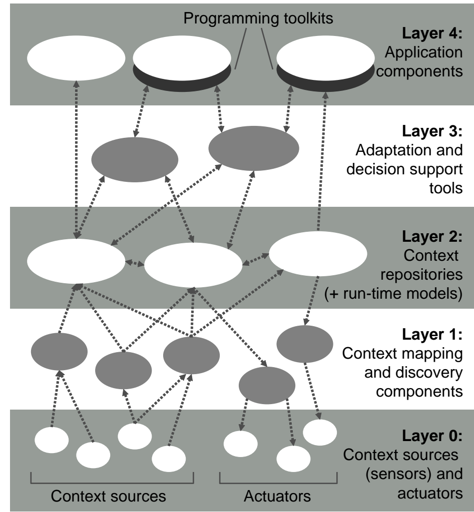

Fig. 1.5 Reference architecture for context-aware systems.

tasks such as discovery of, and remote communication with, context repositories and decision support tools.

The following sections describe examples of infrastructure proposed by the research community, positioning their functionality with respect to layers of the reference architecture.

1.6.2 The Context Toolkit

One of the pioneering software infrastructures for context-aware systems was Dey et al.'s Context Toolkit (Dey et al., 2001). The toolkit addresses primarily layer 1 issues (i.e., interpretation and discovery of context information derived from sensors). It also handles some layer 2 issues; however, it does not support explicit run-time context models of the type discussed in Section 1.5. This means that applications must directly query the software components responsible for acquiring/interpreting context information, rather than querying a single high-level context model.

The toolkit defines a set of abstract component types that can be instantiated and composed to gather context information from sensors. The component types include:

- *widgets*, which acquire information directly from sensors (effectively acting as software wrappers for sensors);
- *interpreters*, which raise the level of abstraction of context information to better match application requirements (for example, transforming the raw location coordinates reported by a positioning device to a building and room number);
- *aggregators*, which group related context information together in a single component to facilitate querying;
- *services*, which are used to invoke actions on actuators (effectively acting as software wrappers for actuators); and
- *discoverers*, which can be used by applications to locate suitable widgets, interpreters, aggregators and services.

The toolkit is implemented as a set of abstract Java objects representing the component types described above. These objects implement a simple communication protocol based on HTTP and XML to support transparent distribution of the components over multiple devices. Libraries of reusable components of each type can be created to support standard sensor and context types.

A variety of applications have been developed using the toolkit, including a home intercom built using speakers, microphones and location sensors, and an In/Out board for the office that provides basic presence awareness, also using location sensors.

1.6.3 Solar

A more recent solution by Chen et al. (2004a) addresses similar issues to Dey et al.'s Context Toolkit, but adds an operator graph abstraction for selecting/composing context sources and data fusion methods using high-level descriptions. It also adds tolerance for sensor mobility and failures. In the

operator graph model, context acquisition and processing is specified by application developers in terms of sources, sinks and channels. Here, sensors are represented by sources and applications by sinks. Operators, which are responsible for data processing/fusion, act as both sources and sinks. Similarly to the Context Toolkit, no explicit run-time context model is present.

Chen et al. have implemented support for the operator graph model in the form of *Solar*, a peer-to-peer platform that instantiates the operator graphs at run-time on behalf of applications. The Solar hosts support application and sensor mobility by buffering events during periods of network disconnection, and also address component failures by providing monitoring, recovery and preservation of component states.

1.6.4 The PACE middleware

The two software infrastructures just described address layer 1 issues and a small subset of layer 2. The PACE middleware developed by Henricksen et al. (2005b) focuses on layers 2 and 3, but also provides some support for layer 4. It consists of:

- a context management system (layer 2) that manages run-time context models, including:
  - model metadata (i.e., model definitions); and
  - context facts from layer 0 and 1 sources, user profiles and applications;
- a preference management system that provides user-customizable decision-support for context-aware applications (layer 3);
- a programming toolkit that facilitates discovery and use of context and preference repositories by context-aware applications (layer 4); and
- tools that generate components that can be used by all layers, including a flexible messaging framework for transparently transmitting context information over a variety of communication protocols (RPC-based and message-based) using custom-generated stubs produced from context model descriptions.

The context and preference management systems are the key parts of the middleware. The former provides run-time support for Henricksen et al.'s Context Modelling Language (CML) and situation-based query mechanism, both of which were described in Section 1.5.5. The preference management system provides preference-based decision support for applications, using a novel solution for specifying context-dependent user preferences. The preferences use a scoring mechanism to indicate the suitability of application actions for particular contexts. The actions are application-specific - for example, an action for a document retrieval application might correspond to selecting a given document or search term. In addition to numerical scores that capture relative preference, users can describe policies such as obligation (actions that must be taken) and prohibition (actions that must not be taken).

A benefit of this preference-based decision support is that most of the evaluation of context information is done as a side-effect of preference evaluation, rather than directly by context-aware applications. This reduces the coupling between the application and the context model, allowing the application to better tolerate changes in the model. For instance, new kinds of context information can be used simply by adding preferences that reference this information, without modifying the application.

Like the Context Toolkit, the PACE middleware has been used to implement various context-aware applications, including context-based routing of phone calls (McFadden et al., 2005) and automatic profile switching for the Nokia 6600 mobile phone.

1.6.5 Discussion

The examples of software infrastructure that have been described here are experimental solutions developed by the research community. None of them was developed particularly with independent living of the elderly/disabled in mind; however, all could be adapted for this purpose. Mature commercial implementations do not yet exist, but will be required before context-aware applications can be widely developed and deployed. In addition to mature

solutions, there is still also a need for comprehensive solutions that provide strong support for all layers of the reference architecture, plus support for other important concerns that cross-cut the architectural layers, such as privacy, security, fault tolerance and scalability. Further analysis of the current state-of-the-art in this area is provided by Henricksen et al. (2005b).

1.7 CONCLUSIONS

In this chapter we addressed the use of context-awareness to intelligently assist elderly and disabled people. Context-aware applications can address healthcare needs and support activities of daily living, and can also be used to extend social interaction, environment control, and information flow to family and care givers.

To facilitate a discussion of issues related to designing and implementing context-aware applications, we introduced definitions of context and context information and highlighted their role in context-aware applications. We described several examples of context-aware applications designed to assist elderly or disabled people, in order to illustrate both the possible functionality of such applications and the rich types of context information that many of these applications require. We also emphasized the importance of formal context models in the design and development of context-aware applications, in order to facilitate sharing of context information between applications, as well as to simplify the development and maintenance of context-aware software. We showed the requirements that such models should meet and provided a survey of current context modeling techniques. Finally, we discussed the support which software infrastructure/middleware can provide for context-aware applications in tasks such as gathering, processing and evaluating context information and supporting context-based adaptation decisions.

REFERENCES

Becker, C. and Nicklas, D. (2004). Where do spatial context models end and where do ontologies start? A proposal of a combined approach. In *UbiComp 1st International Workshop on Advanced Context Modelling, Reasoning and Management*, pages 48–53, Nottingham.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2004). Extensible Markup Language (XML 1.0) (third edition). W3C Recommendation, 4 February 2004.

Brickley, D. and Miller, L. (2005). FOAF vocabulary specification. Namespace Document, 27 July 2005.

Buchholz, S., Hamann, T., and Hubsch, G. (2004). Comprehensive Structured Context Profiles (CSCP): Design and experiences. In *1st Workshop on Context Modeling and Reasoning, PerCom'04 Workshop Proceedings*, pages 43–47. IEEE Computer Society.

Chen, G., Li, M., and Kotz, D. (2004a). Design and implementation of a large-scale context fusion network. In *1st Annual International Conference on Mobile and Ubiquitous Systems*, pages 246–255. IEEE Computer Society.

Chen, H., Finin, T., and Joshi, A. (2004b). An ontology for context-aware pervasive computing environments. *Knowledge Engineering Review*, 18(3):197–207.

Chen, H., Finin, T., and Joshi, A. (2005). *The SOUPA Ontology for Pervasive Computing.* Ontologies for Agents: Theory and Experiences. Springer.

Dey, A. K. (2001). Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7.

Dey, A. K., Salber, D., and Abowd, G. D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2-4):97–166.

Gray, P. and Salber, D. (2001). Modelling and using sensed context information in the design of interactive applications. In *8th IFIP International Conference on Engineering for Human-Computer Interaction*, volume 2254 of *Lecture Notes in Computer Science*, pages 317–336. Springer.

Gu, T., Wang, X. H., Pung, K. K., and Zhang, D. Q. (2004). An ontology-based context model in intelligent environments. In *Communication Networks and Distributed Systems Modeling and Simulation Conference*, San Diego.

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

Harter, A., Hopper, A., Steggles, P., Ward, A., and Webster, P. (2002). The anatomy of a context-aware application. *Wireless Networks*, 8(2-3):187–197.

Henricksen, K. and Indulska, J. (2006). Developing context-aware pervasive computing applications: Models and approach. *Journal of Pervasive and Mobile Computing*, 2(1):37–64.

Henricksen, K., Indulska, J., and McFadden, T. (2005a). Modeling context information with ORM. In *OTM Federated Conferences Workshop on Object-Role Modeling*, volume 3762 of *Lecture Notes in Computer Science*, pages 626–635. Springer.

Henricksen, K., Indulska, J., McFadden, T., and Balasubramaniam, S. (2005b). Middleware for distributed context-aware systems. In *International Symposium on Distributed Objects and Applications*, volume 3760 of *Lecture Notes in Computer Science*, pages 846–863. Springer.

Henricksen, K., Indulska, J., and Rakotonirainy, A. (2002). Modeling context information in pervasive computing systems. In *1st International Conference on Pervasive Computing*, volume 2414 of *Lecture Notes in Computer Science*, pages 167–180. Springer.

Henricksen, K., Livingstone, S., and Indulska, J. (2004). Towards a hybrid approach to context modelling, reasoning and interoperation. In *UbiComp 1st International Workshop on Advanced Context Modelling, Reasoning and Management*, pages 54–61, Nottingham.

Hobbs, J. R. et al. (2002). A DAML ontology of time, November 2002. http://www.cs.rochester.edu/ ferguson/daml/daml-time-nov2002.txt.

Horrocks, I. (2002). DAML+OIL: A description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25(1):4–9.

Indulska, J., Henricksen, K., McFadden, T., and Mascaro, P. (2004). Towards a common context model for virtual community applications. In *2nd International Conference on Smart Homes and Health Telematics*, volume 14 of *Assistive Technology Research Series*, pages 154–161. IOS Press.

Indulska, J., Robinson, R., Rakotonirainy, A., and Henricksen, K. (2003). Experiences in using CC/PP in context-aware systems. In *4th International Conference on Mobile Data Management*, volume 2574 of *Lecture Notes in Computer Science*, pages 247–261. Springer.

Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M. H., and Tran, L. (2004). Composite Capability/Preference Profiles (CC/PP): Structure and vocabularies 1.0. W3C Recommendation, 15 January 2004.

Manola, F. and Miller, E. (2004). RDF primer. W3C Recommendation, 10 February 2004.

McFadden, T., Henricksen, K., Indulska, J., and Mascaro, P. (2005). Applying a disciplined approach to the development of a context-aware com-

munication application. In *3rd IEEE International Conference on Pervasive Computing and Communications*, pages 300–306. IEEE Computer Society.

McGuinness, D. L. and van Harmelen, F. (2004). OWL Web Ontology Language overview. W3C Recommendation, 10 February 2004.

Open Mobile Alliance (2003). User Agent Profile Version 20-May-2003. OMA document OMA-UAProf-v2_0-20030520-C.

Ranganathan, A. and Campbell, R. H. (2003). An infrastructure for context-awareness based on first-order logic. *Personal and Ubiquitous Computing*, 7(6):353–364.

Strang, T., Linnhoff-Popien, C., and Frank, K. (2003). CoOL: A Context Ontology Language to Enable Contextual Interoperability. In *4th International Conference on Distributed Applications and Interoperable Systems*, volume 2893 of *Lecture Notes in Computer Science*, pages 236–247. Springer.

Wang, Z., Zhang, D., Gu, T., Dong, J., and Pung, H. K. (2004). Ontology based context modeling and reasoning using OWL. In *Workshop on Context Modeling and Reasoning, PerCom'04 Workshop Proceedings*, pages 18–22, Orlando. IEEE Computer Society.