Designing for palpability: Workshop at Pervasive 2007, 13-16 May 2007, Toronto, Canada. http://www.ist-palcom.org/palpable-pervasive-2007

What are we Interacting With? : Architectural Reflection for Technical Translucence in Ubiquitous Computing

Mads Ingstrup[1] and Paul Dourish[2]

[1]University of Aarhus, Denmark
ingstrup@daimi.au.dk
[2]University of California, Irvine, USA
jpd@ics.uci.edu

Abstract. A number of researchers have observed that ubiquitous applications pose new challenges to usability because their locus of interaction has moved away from the traditional computer desktop and been distributed through the everyday world. We argue that the distributed nature of ubicomp infrastructure also poses interaction challenges. Ubicomp systems are made up of collections of components only some of which the user may have access to or be able to control. We propose a solution to this problem based on architectural reflection, essentially making aspects of a system's architecture visible in its interaction model, an approach we label "technical translucence." We present a case study in which architectural reflection is used to give insight into and control over privacy in a distributed, location-based system.

Introduction

Consider driving a car: the sound of the engine, the movement through the landscape, the feel of the brakes, and the resistance in turning the steering wheel all provide valuable cues to guide you in using the car. If the car breaks down, the hood can be opened and the mechanics are available for anyone to see (if not to entirely understand.) Even before this happens, the noise from the engine, or the smoke, or the fact that nothing lights up when turning the key will tell something about what is wrong. In important ways, a car is not invisible to its user, and the ways in which it is visible and accessible are an important part of how it is used. Drawing on the work of Erickson and Kellogg [2000], we call this property "translucence;" the way the system operates provides a partial view into its structure.

By way of contrast, consider next the poor user of a prototypical ubiquitous system: it is highly dynamic and distributed, and encompasses many devices, some known to the user and some not, all or most of which are connected with wireless or otherwise invisible connections, and which in turn rely on a complex and hidden infrastructure. It is not even clear exactly how to determine when a device is part of a system, or if the notion of a system applies at all. How, then, does its user stand a chance of knowing what is going on?

One line of argument is that people don't need to know what is going on because systems that can automatically infer people's intent can always respond appropriately. Another line of argument is that people don't need to know what is going on because systems will be so seamlessly interconnected and easy to use that they will "just work." However, as we observe, our activity in the everyday world depends critically on our ability to examine and monitor it, and even people have difficulty inferring each other's intent. More particularly, several authors have noted the ways in which interaction with ubiquitous systems raises a series of problems that go beyond the traditional problems of software systems usability.

Bellotti et al. [2002] discuss a number of issues that arise in sensor-based interactive systems, including many ubiquitous computing applications. Their analysis combines experience with a wide range of ubiquitous computing prototypes with observations from social science analysis, most particularly interaction analysis for interpersonal interaction. They point out a series of ways in which the move from traditional GUI applications to ubiquitous computing systems interferes with mechanisms that GUI designers have developed to help with interactional problems. In particular, they point to the importance of questions such as "how do I know that the system is doing the right thing?," "how do I address one (or more) of many possible devices?," or "how do I effect a meaningful action?" as the basis of effective human-machine interaction, and use examples to underscore the complexity of gaining adequate answers to these questions in ubicomp context. Visibility – seeing opportunities for action, and monitoring the system's response to user actions – is at the heart of these concerns.

The particular area that motivates our interest here is control over privacy and information flows. It is regularly observed that ubiquitous computing applications often put private information at risk, relying as they often do on long-term information capture and modelling, and on the transmission of information amongst different devices that potentially belong to different administrative domains. Privacy has become an important topic in networked systems in general, and in ubiquitous computing in particular [e.g. Jiang et al., 2002; Langheinrich, 2001], and the usability of privacy and security technologies has become a topic of growing research attention [e.g. Zurko, 2005]. One strand of research in this area concerns the application of usability metrics to software components designed to provide control over privacy and security; another focuses on higher-level question of how people can act securely or understand privacy in complex technological environments.

We believe that the problems outlined by Bellotti et al. are directly relevant to the problems of privacy and information management in ubiquitous computing. In particular, we argue that those questions as fundamentally concerned with software architecture as an interaction concern – that is, that user interaction needs to be organized around understandings of the structural arrangements of components that make up the system. We introduce an approach to software architecture, based on computational reflection, which allows the dynamic configuration of a distributed system to be reflected in the interface and to become a means of both introspection and control. We illustrate this approach using an example concerning privacy in location-based systems.

The remainder of the paper is organized as follows. First, we review some related work on both privacy and the relationship between architecture and interaction. Next, we briefly introduce reflection, and the declarative approach to reflective middleware

that underlies our prototype. We then continue by presenting a case study using reflection to convey an awareness of privacy issues for end users. Following that, the application of our approach to privacy is detailed with a description of the design and implementation of a prototype system.

Background and Related Research

The classical approach to software system usability argues that the goal of interface design is to create an interactional abstraction that hides the details of system implementation from the users of the system. From this perspective, the details of implementation are largely irrelevant to the user's work, and presenting them in the interface interferes with the efficient conduct of that working activity. So, a word processor should abstract away from particular storage mechanisms and data representations, in order to present an interactional model of the working task itself rather than the means by which the task is being achieved.

However, in a range of domains, examples have arisen that question this basic approach. They suggest that, while user interfaces need to be designed in terms of task-oriented accounts of system structure and function, these accounts may inevitably have to reveal aspects of system implementation, because system implementation will unavoidably become visible in the interface.

A classic example is provided by Greenberg and Marwood [1994] in their exploration of synchronous collaborative editing. They analyse a range of consistency control mechanisms for synchronous editing, and show the ways in which the details of these mechanisms "bleed through" into the user interface. Strong locking, for instance, is revealed in the interface when a user attempts to modify a section of the document that is currently locked by another user. Transaction semantics are revealed in the interface through the temporal properties of updates. Optimistic concurrency control may reveal itself through roll-backs. Greenberg and Marwood convincingly demonstrate that the implementation and the user interface cannot be entirely isolated from each other.

In a related example, John and Bass [2001] describe how the choice of supporting *cancel* in an application has wide implications for its software architecture. Cancelling stops an action in progress and reverts its consequences. It must thus be supported by all components involved in executing that action, and arguably also require new and specialized components [John and Bass, 2001].

Similar issues have been encountered in research into usable privacy and security. While one strand of work in this area has attempted to apply usability metrics to evaluate and potentially redesign existing software mechanisms for privacy (such as password authentication [Adams et al., 1997], key exchange [Balfanz et al., 2004], or email encryption [Whitten and Tygar, 1999]), an alternative approach has looked more broadly at the ways in which people can "act securely." The fundamental insight here is that security and privacy cannot be defined absolutely; rather, they are relationships between needs and settings. In these cases, then, what is needed as a means by which people make informed decisions about the consequences of their actions. Like driving a car, this does not require a fully detailed understanding of the intricate inner workings of the system (the internal combustion engine, computer-controlled

ignition timing, etc); rather, we take the view that, by becoming aware of the relationship between action and response over time, people can learn how changes in their behaviour result in changes in the system and its response, and so develop ways of relating their needs to system action, and system action to consequences. In other words, as in the cases described above, what we find here is that the internal arrangements of system components, and the configuration of those components, is a relevant piece of information in determining appropriate user action; "internal" properties of the system are relevant to end users, rather than things that should be (entirely) shut away. Erickson and Kellogg [2000] point to the importance of what they term "social translucence" in helping people make technologies work appropriately; we argue here for a form of technological translucence in which people can understand and assess information systems.

Given the inherently distributed nature of ubiquitous computing applications, we believe that some degree of infrastructure translucency can particularly be of value in ubicomp applications. However, two particular problems emerge, from a technical standpoint. The first is the wide range of components and circumstances that people might encounter in the course of using a ubicomp system. The second is that these components arise independently, being produced, deployed, and administered by different groups. In this case, ad hoc solutions to system translucence are unlikely to be successful, and we need some more principled and systemic way of providing such facilities. We have turned to computational reflection as a mechanism by which this view into system internals can be provided.

Architectural Reflection

A reflective system is "a system which incorporates structures representing (aspects of) itself. We call the sum of these structures the self-representation." [Maes 1987, p. 148]. The self-representation can be causally connected with the system it represents, such that changes in either the system or the model of it will cause a corresponding change in the other.

Reflection has its origins in the artificial intelligence community, where it was posited as a mechanism by which systems might be able to reason about their own action [Smith, 1982]. Incorporated into a version of the Lisp programming language, it was adopted more generally as a way to allow programmers to build adaptable and flexible programming systems [Budd, 2002; Forman, 1995; Kiczales et al., 1991]. More recently, reflective approaches have been used to provide related forms of flexibility in domains such as database management [Klas and Schrefl, 1995; Tan et al. 2003; Barga and Pu, 1996] or distributed systems [Stroud and Wo, 1995].

Of late, research has begun into architectural reflection, wherein the model a system maintains of itself is a model of its runtime software architecture. This has been successfully used to make toolkits, frameworks and middleware more flexible [Kon et al., 2002; Capra et al., 2002; Blair et al., 2001; Dourish, 1998], or to make systems more dependable by enabling self-repair [Oreizy et al., 1999; Dashofy et al., 2002; Garlan et al., 2003; Garlan and Schmerl, 2004].

In going from reflection in programming languages and databases, over toolkits and middleware onto to architectural reflection, we witness not only a change in the functions reflection is used to accomplish, but also in the domain being reflected about. So whereas reflection with metaclasses in an object oriented language is concerned with representing the constructs—classes, objects, methods and so forth—used to specify computation in these languages, architectural reflection is concerned with representing systems.

Using reflection at the system level is most suitable for the purposes of concern in this paper. Further, because the architectural level of abstraction is high, it leads models of relatively low complexity, and is concequently used by architects to communicate with not only developers but also other stakeholders in a system. For these reasons we built on recent work with architectural reflection in ubiquitous systems, and used the Architecture Query Language (AQL) and evaluation engine [Ingstrup and Hansen, 2005; Ingstrup, forthcoming] in our prototype. However before introducing that, it is necessary to take a brief look at the concepts in terms of which a system's runtime architecture can be described.

Runtime Architecture Ontology

Structural aspects of architecture are described with concepts such as *components, connectors,* and *interfaces* which by now are fairly well established in the software architecture community. The particular model used in AQL is very similar to that of the xADL architecture description language [Dashofy et al., 2005], and is illustrated in figure 1. Below we describe each of the four concepts.

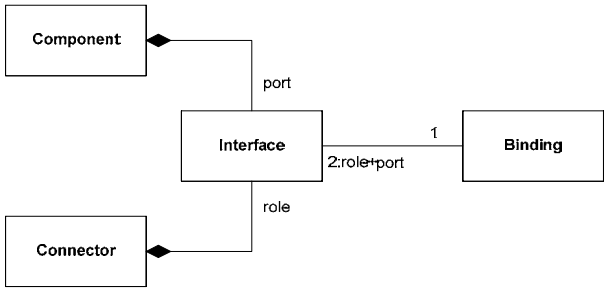

Fig. 1. Ontology of runtime architecture.

The most widely accepted definition of a *component* is Szyperski's: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [Szyperski, 1997]. However since we are concerned with reflection, the term will in the remainder of this paper be taken to mean *runtime component*, that is, an instance of running code which is created from a deployed binary component.

Whereas components encapsulate computation, connectors encapsulate interaction between components [Shaw, 1993]. When the term was first introduced there were no

units of deployment or source code to which it corresponded in a simple way[1], and connectors remain less obvious units of design than do components. However, recently the use of first class connectors has been researched by e.g. providing language support [Aldrich et al., 2003] and a formal basis for their semantics [Allen and Garlan, 1997], and in the PalCom runtime environment [Schultz et al., 2005] support at the for explicit connectors in the virtual machine is currently being experimented with. Explicit connectors at the programmatic level is important to the purpose at hand, because it enables reflection on them, which is required to understand how components are connected.

Comment [PD1]: This is the first mention of PalCom. In fact, it may be the ONLY mention of Palcom…

Interfaces are the unit of association between components and connectors. They have a direction specifying whether they are provided or required. When a component uses a connector, it has a required interface and the connector an equivalent provided one (or vice versa) to which it is bound. A binding is thus an association of an interface on a component to an interface of opposite directionality (provided or required) on a connector.

Architecture Query Language and Evaluation Engine

As reflection has been developed as a system design principle, and as it has been applied in different domains, a range of approaches have been explored for making reflection available. In the original 3-Lisp system, the interface to reflection was primarily state-based (comprising the execution state of the interpreter) [Smith, 1982]. In CLOS, an object-oriented interface – the "metaobject protocol" – was developed [Kiczales et al., 1991].

A first concern is the level of abstraction of the interface. If it allows modifying the system at too low a level it is far too easy to end up with an inconsistent system. On the other hand, when retrieving information which is at too low a level of abstraction, the information actually desired might not be there. Knowing, for instance, a complete object-graph of a running system does not suffice to derive boundaries of the runtime components without additional information. Therefore, if the goal as in our case is to reason about a system's architecture, the reflection-interface must support the architectural level abstraction. In applications of reflection seeking to increase adaptability through reconfiguration, it is the architectural level of abstraction which as been most successful, because components, by having only explicit dependencies, lend themselves to be reasoned about in a formal manner.

For ubiquitous systems, a reflective interface could suitably be declarative as argued by Ingstrup and Hansen [2005]. Their approach is to view the system as a distributed database of reflective information, which is then accessed declaratively by issuing queries expressing what reflective data to retrieve. The queries are expressed in a dedicated Architecture Query Language, AQL, using the concepts with which runtime architecture is described. To retrieve sufficient information to build, for instance, a component and connector (C&C) diagram the following query would be issued:

[1] This is of course also true of similar concepts like component, object and function, which were not 'there' either, until someone built them.

```
SELECT cm.id, cn.id
FROM Component:cm, Connector:cn, Binding:bn
WHERE bn.cmpid==cm.id AND cn.id==bn.cnnid
```

The result of evaluating such a query against a given system is a list of tuples, each containing the objects specified in the SELECT-clause of the query. There would be a tuple for each pair (component, connector) which are connected.

This declarative fashion of accessing the reflective information is suitable for ubiquitous computing for a number of reasons. Firstly it allows a scheme where the global reflective image is only retrieved to the extent necessary. This makes it more efficient in its utilization of bandwidth than if such an image was stored centrally and had to be synchronized for every change to the system configuration [Ingstrup, forthcoming].

Secondly, the declarative nature of the interface ironically helps to keep its clients separate from the organization of the underlying system, in the sense that their reliance on assumptions about that organization must become explicit in the queries. This helps further the separation between base level and the meta level of the system, and as we will see with the prototype we have built, eases the development of generic functionality.

Usability and Architectural Reflection

Architectural reflection, then, makes a structural description of a system's architecture (in our case, its run-time architecture) available to the system as an aspect of its own state; AQL in particular provides a principled and structured manner by which this representation can be examined and manipulated, and an abstraction the potentially distributed state. As we have suggested, architectural reflection has been employed largely in support of systems design considerations such as flexibility, evolution, and robustness. However, the fundamental mechanism supporting reflection – a representation of the system's structure and activity that is available to that system – is a means also to support translucent technological structures that are available to the users of the system. The research question, then, is how an architectural description of this sort can be incorporated into the user experience in such a way as to provide meaningful and useful information relevant to end user activity.

The particular case that we have been exploring to help us answer these questions is one that arose in the work of colleagues dealing with emergency response. Their design efforts had been focused primarily on the ways in which emergency services and first responders could make use of advanced technological infrastructures in order to work more effectively and more safely. Our engagement with this work was around the privacy questions that arise in such settings.

Privacy can be considered the claim of individuals, groups or organizations to determine for themselves when, how and to what extent information about themselves is communicated to others [Westin, 1967]. Exercising that claim is precluded by many information systems today. Firstly because users have limited means with which to become aware about how their information flows through the systems they use. Secondly, because, even if they had such awareness, the means with which they can con-

trol those information flows are very crude, often coming down to the choice of whether to user the system or not.

As a specific example of the general approach laid out in the introduction, we have built a prototype in which reflection is used to visualize the interaction between flows of privacy-laden information on the one hand, and the user's constraints on what to disclose and how it can be used on the other hand. This section describes the prototype and the generic approach it incorporates, as well as how it is applied to an example system.

Background and Motivation

The particular case that we have been investigating is drawn on ongoing research into the use of advanced ubiquitous and sensor-based technologies for crisis management and emergency response, in settings such as accidents and natural disasters. In these cases, we are interested in how information technologies can enhance the work of the emergency services, either by providing them with more effective means to coordinate their own work, or new ways to understand the settings in which they will perform their work.

Consider the case of a fire. In many "high-tech" settings, the buildings in which a fire might break out are ones saturated in technologies – mobile and desktop computers, cell phones, motion detectors, etc. Can we take advantage of these technologies in order to provide fire services with an understanding of where people are in the building, so that they can target their resources appropriately?

Our colleagues who have been working on these problems have been investigating the use of location-based technologies to support situational awareness for first responders and those responsible for managing and coordinating emergency services. Cell phones, for example, are potentially very valuable as sources of information about people density in a building. Mobile phones can support a range of location mechanisms including cell tower triangulation [Otsason et al., 2005], Bluetooth proximity sensing, GPS location tracking, and, in the future, WiFi access point triangulation. These different technologies have different characteristics, both in terms of the underlying technologies and in terms of the potential privacy issues associated with them. The question which we posed as part of this work was, how can the end users of a location service of this sort understand and control the ways in which their information might be being used?

Our approach is as follows. We employ architectural reflection to model the flows of information through the system. We annotate a traditional model with information about the kinds of information that each component requires and the sorts of service that it can provide. In particular, we are concerned with the constraints on these services and the relationship between, on the one hand, "quality of service" that the system can provide and "quality of privacy" that a user might require. Since a single system may support different needs (e.g. end user location services as well as situational awareness applications for emergency responders), we find the architectural framework useful as a way to understand the relationships between different requirements and different services. What the architectural model can provide, then, is a series of "knobs" by which users can control the forms of information that they might provide

to the system and hence the kinds of service that it might be able to provide in response.

User Interface

A screen shot of the user interface of our prototype is shown in figure 2. It contains a set of controls, and a visualization of the runtime architecture of our example system.

The three sliders at the top make up the knobs with which the user can specify the following aspects of how his or her location may to be used:

- *Location Precision.* The precision with which the user's location is disclosed. When set to *p*, it is only guaranteed that the user is located somewhere within a circle of radius *p* centred at the given coordinates, which may be specified with any accuracy (number of digits). Various techniques for randomization as a function of *p* may be imagined to ensure that the given coordinates cannot be assumed as precise as they are accurate.
- *Update Interval.* The minimum time that must pass between two requests for the user's location.
- *Storage Time.* The maximum time a location may be stored.

In the diagram below the sliders, the small boxes represent connectors, and the bigger ones components. Interfaces are not shown explicitly, but an edge between a component and a connector represents a binding between an interface on that component and one on that connector.

The system is a collection of components that depend on location and other privacy-laden properties to provide a service to the user, to an emergency control center or to other components in the system. It consists of following components. A *location service* with which the user manages his or her location information, and which enforces the user's privacy settings. The location held by that service may come from any source such as a GPS component on the user's cell phone or laptop, or from a phone company doing localisation by triangulation with cell phone towers. The concrete method of localisation could be used to constrain the range of e.g. location precision, but that is not implemented in the current prototype.

In order to illustrate the mechanisms at work, we include other services provided in a location-based system, including a Friend Finder and a Route Planner. The Situation Awareness service is one whose primary user is not the person carrying the device, but rather the emergency service coordinator. It would serve a crisis management center in case of a major incident, to help in e.g. determining how many persons are within a given area of the emergency site (*Person Counter*), or provide a general map of dispatched resources (*Location Map*).

Through this interface, users may experiment with how a change in the quality of privacy [Tentori et al., 2005] they specify affects the quality of service provided to them. To do so, users specify their required quality of privacy by stating e.g. the maximum precision with which they allow their location to be known. Such a specification imposes constraints on the flow of privacy-laden information in the system. Given the constraints and a representation of the system obtained through reflection, it is possible to compute which parts (components and connectors) of the system are af-

fected by a given change. Whenever a component either stops working or suffers a decrease in the quality of service it can provide, due to the user's changed settings, that component turns red in the user interface. Users are thus able get an impression of the effect of their changes.

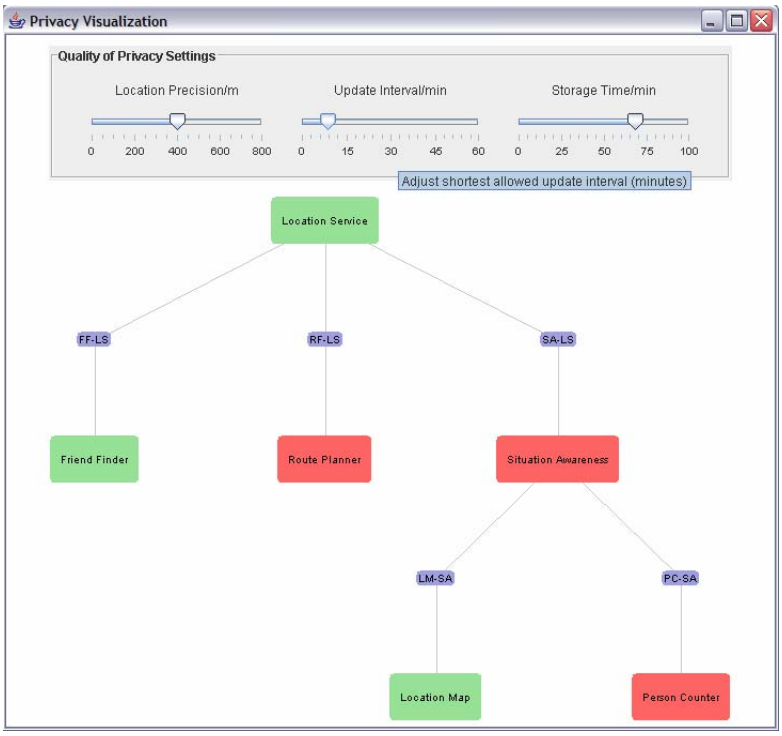

Fig. 2. The user interface of the prototype. The text box under the two rightmost sliders is a tool tip text for the mouse pointer (not shown) over the middle slider.

Representing Quality of Privacy

Constraints on how privacy sensitive information can flow through the system are represented as quality of privacy (QoP) specifications. A QoP specification of information exchange specifies both *what* privacy-laden information is exchanged (e.g. location), and what *quality* (e.g. precision) that information has. The reflective mirror of each component or connector in our prototype includes a QoP specification. That is, it specifies both what privacy-laden information is used and supplied, and the quality with which that information is required or provided. Specifically, all this is represented at the granularity of interfaces on components and connectors. In the prototype they are thus annotated with a specification of the *required* or *provided* quality of privacy. The distinction between *provided* and *required* corresponds to the direction of

the interface, i.e. whether the component/connector that it is part of requires or provides it.

The QoP specification in the prototype takes the form of a list of (property=value) pairs. For example, (location-precision=100m) which, if the interface is provided, tells that the particular component only provides information about a person's location down to 100 meters. If the interface is instead required, the (property=value) specifies that the component must know the location with a precision no less than 100 meters in order to work properly.

When users change their settings, the values of the tree sliders are used to set the privacy data on the outgoing interfaces on the *Location Service.* The quality of privacy information available to the *Location Service* is assumed high enough that the information forwarded to other components is equal to that set by the user.

Determining Effects of a Change in QoP

Whenever the QoP settings are changed by a user, it is computed what privacy-laden information can reach each component given the interconnection specified in the component and connector (C&C) graph, and the privacy specifications on each component and connector's outgoing interfaces. Knowing what privacy-laden information reaches each component, and of which quality, makes it a simple matter of comparison between available and required information to determine if a component or connector is obstructed (and hence should be coloured red in the user interface).

The algorithm for computing the flow of information through the C&C graph is optimistic; it assumes that each component forwards all information available to it, less that which is forbidden by its outgoing (provided) interfaces. As an example assume that a component C has a required interface I_r annotated with (location-precision=10m) and a provided one, I_p, annotated with (location-precision=500m). If the connector to which C is bound through I_r provides location information with a precision of 5 meters, then C is not obstructed by lack of location information of sufficient quality. However, any connector bound to I_p may not assume location information retrieved through I_p to be of greater precision than 500 meters (though it might still get higher quality information through other pathways between itself and the *Location Service* in the C&C graph).

So, referring to figure 2, even though the *Situation Awareness* component might not be able to provide its service at the desired quality (in this case because it requires an update interval of 5 minutes or less), this does not necessarily preclude the *Location Map* component from functioning properly (having no requirement on the update frequency) even though it depends on the *Situation Awareness* component.

The algorithm for computing which components and connectors are obstructed given what information is available at the origin (the *Location Service* in the sample system) is specified below. The input is the directed C&C graph, with one node per component or connector, and one edge per binding. The direction of an edge is determined by the direction of the two interfaces associated by the binding it represents.

```
void computeObstructed(Node origin, DiGraph g){
   FifoQueue Q;
   Node current, target;
   Edge e;
```

```
    PrivacyData from, to, tmp;
    current := origin;
    for (n in g.nodes())
       n.iCount := 0;
    Q.insertLast(current);
    while (current:=Q.removeFirst()){
       for(e in current.OutEdges()){
          target := e.getTargetNode();
          from := e.getFromInterface().getPrivacyData();
          to := e.getToInterface().getPrivacyData();
          tmp := maximalquality;
          tmp.constrainFrom(current.getAvailablePrvData();
          tmp.constrainFrom(from);
          target.setAvailablePrvData(tmp):
          if (!to.lessThan(tmp))
             target.setObstructed(true);
          target.iCount++;
          if (target.iCount == target.inEdges)
             Q.insertLast(target);
       }
    }
}
```

Additional care is required if the C&C graph contains cycles, because the algorithm relies on the body of the outer loop to make nodes available in the queue after computation of the contribution from all their ingoing edges in previous iterations. In the presence of cycles, the flow through each edge can be computed repeatedly until a stable state is reached.

Implementation Notes

The architecture of the prototype is shown in figure 3. It is implemented in Java, using JDK 1.5.

The AQL Evaluation Engine supplies the architectural data about the system. In the prototype that data is imported from an XML file in xADL format [Dashofy et al., 2005]. It uses a schema that extends the core xADL language construct *InterfaceInstance* to specify a *PrivacyInterfaceInstance* type which is annotated with privacy information in the form of a *PrivacyData* element. *PrivacyData* then contains *LocationPrecision*, *UpdateFrequency*, and *StorageTime* elements, each simply holding a number.

The visualization component uses AQL to retrieve the architectural data on which the visualization is based. When started, it issues the following query to the AQL evaluation engine:

```
SELECT b.role, b.port, b.cmpid, b.cnnid, cm.description
FROM Component:cm, Binding: b
WHERE cm.id==b.cmpid
```

This results in a list of tuples, each holding the set objects specified in the first line of the query. There is a tuple for each binding, which is needed here. Note that the query that was shown in the section on AQL cannot be used, since it only yields information on which pairs of components and connectors are connected. To get the correct result we need to take into account *all* bindings between each component-connector pair. The result of evaluating that query is then used to generate a graph in the format used by the Prefuse graph visualization and interaction toolkit [Heer et al., 2005]. We tried

several different layout algorithms, and found the layout done by the DOT tool in the Graphviz package [Gansner and North, 2000] to be most satisfactory.

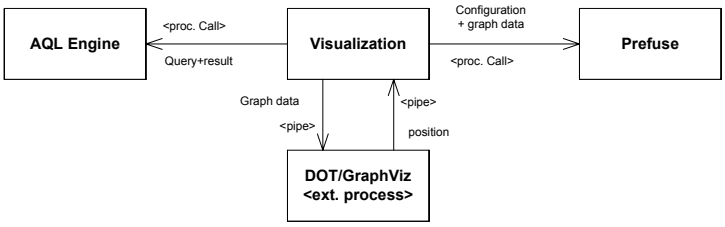

Fig. 3. Architecture of the prototype system.

Discussion

We have focused on privacy as an area where technical translucence can be of especial value. Certainly, this is a critical consideration in a range of ubicomp systems. Both the nature of these systems – ad hoc coalitions of distributed components that cross administrative boundaries – and the challenges of effective privacy – situational assessments of immediate need rather than normative distinctions between "private" and "public" – lend themselves to this approach. However, we feel that this approach has wider promise.

One area of current investigation is end-user customization and composition of services. The diversity of settings into which ubicomp devices are deployed suggests that we need to look beyond "static" approaches to design, and provide support for the appropriation of interactive systems. Essentially, this suggests that "design" does not end when a technology is deployed, but continues into use [Kyng and Henderson, 1991]. This is an argument that has been developed and explored deeply in the Participatory Design community, but which we can also see very much at work in studies of the adoption of mobile and ubiquitous systems [Ito and Okade, 2005].

These examples help speak to a common concern that arises when presenting this work, which is the extent to which an approach of this sort requires that end users of technical systems are required to understand internals that "ought" to be hidden from view. We argue that this is to misconstrue the forms of invisibility appropriate for ubicomp technologies. As in our example of driving a car, the invisibility of the engine is not a literal invisibility, but rather a form of invisibility-in-use; that is, the internal operation of the engine and the mechanics of the car are "sunk into" everyday practice. Similarly, the goal of our use of architectural reflection here is not to require explicit meditations upon the internal structure of a system, but rather to allow the system's action to be perceptually linked to accounts of how it is that action came about. What we see this approach as visualizing is not simply structure but rather opportunities for action.

Related and Future Work

The use of reflective architectures as a basis for interactive systems was first explored by Dourish and colleagues [Dourish, 1995; Dourish et al., 1996], as an extension of Rao's application of reflection to user interface technologies [Rao, 1991]. Rao was concerned principally with the software toolkits from which user interfaces and window systems were constructed, and used reflection as a means by which application semantics could be used to customize window system operation, essentially revising or augmenting architectural decisions made by system developers and hidden behind traditional abstraction barriers. The approach described by Dourish et al. also uses reflection to render abstraction barriers translucent, but does so in order to reveal aspects of system operation to end users so that they can match the system's behavior to their immediate needs, and make informed assessments of the ways in which the system is responding to their actions.

At the architectural level, computational reflection is primarily employed as a means to provide technological flexibility, particularly in support of goals such as flexibility, extensibility, and evolution. Here, however, we are following a different path, using reflection as a means to provide a view onto the internal behavior of software systems [Eisenberg, 1996]. From a user perspective, some have suggested using related approaches in order to manage end-user adaptation [e.g. Randall, 2003]. Our work is closer in spirit, perhaps, to that of people such as Vaghi et al [1999] who attempt to make aspects of system-internal behaviors visible in the interface.

A number of researchers have suggested that privacy is potentially particularly amenable to this approach. Good et al [2005] point to the importance of being able to make informed decisions about information flow and information management in everyday interaction, while Dhamija and Tygar [2005] explore the ways in which visual techniques can help to make potential phishing attacks visible to end users. In the domain of ubiquitous computing, the problems of privacy are somewhat different, and are related to the problems described by Bellotti et al [2005] – problems of making sense of systems that are assembled from components and information flows that are not apparent.

The details of the particular case examined here are motivated by real concerns. It is drawn from an ongoing inquiry into technological support for emergency response, in which the needs of first responders and situation controllers may differ from those of the subjects of the system. Further, this particular case depends on people making themselves available to the system just in case their information is needed, even though most of the time it will not be. This is an interesting case for monitoring and assessment, then. However, while motivated by these needs, the specific details we have incorporated into our prototype are simply for the purposes of exploring the underlying model and providing a compelling example. Our most important piece of further work, then, is to scale this approach up to systems of greater complexity and to develop a more accurate model of the privacy parameters and service requirements. At the same time, we need also to determine whether the level of description is adequate for the users of complex ubicomp applications. That said, we return to the example of driving the car, and note that the sound of the engine and the feel of the steering provide useful clues for drivers even when their model of the operation of their car's engine is either scant or downright wrong. Hybrid engines, computer-

controlled ignition, and other innovations in car engine design mean that the model most drivers hold of their car's operation is only loosely connected to its real behavior – and yet, the model still provides an effective guide for interpretation of manifest signals, when those signals are tightly coupled to the operation of the system. This suggests that, as long as the reflective link maintains a connection between the system's behavior and the reflective representation, the accuracy of that model need not be a constraint upon the adequacy of the representation as a tool for end-user introspection.

In what we have presented here, we have explored questions of privacy and security through architectural descriptions. While this has involved annotating architectural models with information about information flows, our focus has been on those aspects of privacy that are relevant to end users. Ren and Taylor [2005] present a more comprehensive approach to the use of architectural description languages to capture and reason about the security aspects of component-oriented systems. Their concern differs from ours, in that they are concerned, first, with the internal properties of software systems, and, second, with more detailed compositional properties of systems. One area to explore is the opportunities for bringing our interactional approach to their more comprehensive descriptive model.

Conclusion

Although traditional interpretations of usability argue that implementation details should be hidden from the end users of systems, repeated experience in different domains has shown that this is an unrealistic goal. Particularly in the case of networked and distributed systems, aspects of configuration always and inevitably show through, if not directly then indirectly through breakdowns, failure modes, and temporal properties of systems. Designing as if this were not the case is at best limiting and at worst delusional. Instead, we have taken another approach. Given that some aspects of system configuration must inevitably become visible in the interface, can we take control of this visibility and explicitly turn it to our advantage? This immediately raises two questions. The first is, for what sorts of interactional experiences might one want to make aspects of system structure and behavior visible? The second is, what mechanisms can be provided to do this?

For the first, we have been focused here particularly on privacy assessment in ubiquitous computing settings. In the cases we have been considering, privacy is ripe for this sort of approach. Three properties make it so. The first is that privacy and security are properties of systems, not components; they must be understood as emergent properties of the ways in which many components are assembled to form complex wholes. The second is that privacy and security are not absolutes, but defined relative to immediate needs. The third is that, in ubiquitous computing settings such as those we have been describing, privacy and security typically cross administrative boundaries and domains, making adequate assessment particularly difficult.

In answer to the second question, we have turned to architectural reflection. This is motivated first by the fact that architectural descriptions of component interaction are a natural level at which to describe information flows, and second by the fact that ar-

chitectural descriptions will often already exist as a product of the software development process. Architectural reflection makes these descriptions an integral part of the software system itself, so that they are available for introspection, examination and, potentially, change. By augmenting an architectural description with annotations that describe information requirements and available quality of service responses, we are able to build systems in which the match between user requirements and system requirements can be dynamically explored. The architectural description provides a framework of "privacy controls."

Our goal is not simply to allow people to "trade off" privacy against function; while privacy is often thought of in those terms, we do not believe that such a reduction is straight-forward [Dourish and Anderson, in press]. What we do want to do, though, is to allow people to explore the relationship between system behavior and their own.

More broadly, we believe that architectural reflection can be extended beyond its current applications to system infrastructure flexibility and adaptability. It can also provide a "window" into system operation that allows users to assess them and to observe how the system responds to their own actions. By making ubiquitous environments legible, it helps us take important steps towards the problems of making sense of sensing-based systems.

References

Adams, A., Sasse, A.,and Lunt, P. 1997. Making Passwords Secure and Usable. People and Computers XII: Proceedings of HCI'97 (Bristol, UK), 1-19. Springer.

Aldrich, J., Sazawal, V., Chambers, C., and Notkin, D. Language support for connector abstractions. In Luca Cardelli, editor, *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP'03), volume 2743 of Lecture Notes in Computer Science, pages 74–102. Springer Verlag, July 2003.

Allen, R. and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

Balfanz, D., Durfee, G., Grinter, R., Smetters, D., and Stewart, P. 2004. Network-in-a-box: How to Set Up a Secure Wireless Network in Under a Minute. Proc. 13th USENIX Security Symposium.

Barga, R. and Pu, C. 1996. Reflection on a Legacy Transaction Processing Monitor. Proc. Reflection'96 (San Francisco, CA).

Bass, L. and John, B.E. Supporting usability through software architecture. *Computer*, 34(10):113–115, October 2001.

Bellotti, V., Back, M., Edwards, W.K., Grinter, R.E., Henderson, A., and Lopes, C. Making sense of sensing systems: five questions for designers and researchers. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 415–422. ACM Press, 2002.

Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N, and Saikoski, K. The design and implementation of open orb 2. *IEEE Distributed Systems Online*, 2(6):1–40, 2001.

Budd, T. *An introduction to Object-Oriented Programming*. Addison-Wesley, 2002.

Capra, L., Blair, G.S., Mascolo, C., Emmerich, W., and Grace, P. Exploiting reflection in mobile computing middleware. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):34–44, 2002.

Dashofy, E.M., van der Hoek, A., and Taylor, R.N. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26.ACMPress, 2002.

Dashofy, E.M., van der Hoek, A., and Taylor, R.N. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245, 2005.

Dhamija, R. and Tygar, D. 2005. The Battle Against Phishing: Dynamic Security Skins. Proc. Symposium on Usable Privacy and Security.

Dourish, P. 1995. Accounting for System Behaviour: Representation, Reflection, and Resourceful Action. Proc. Third Decennial Conf. on Computers in Context (Aarhus, Denmark).

Dourish, P. Using metalevel techniques in a flexible toolkit for cscw applications. *ACM Transactions on Computer-Human Interaction*, 5(2):109–155, 1998.

Dourish, P., Adler, A., and Smith, B.C. 1996. Organizing User Interfaces around Reflective Accounts. Proc. Reflection'96 (San Francisco, CA).

Eisenberg, M. 1996. The Thin Glass Line: Designing Interfaces to Algorithms. Proc. ACM Conf. Human Factors in Computing Systems CHI'96.

Erickson, T. and Kellogg, W. 2000. Social Translucence: An Approach to Designing Systems that Support Social Processes. ACM Trans. Computer-Human Interaction, 7(1), 59-83.

Forman, I.R. Putting Metaclasses to work: a new dimension in object oriented programming. Addison-Wesley, Reading, MA, 1995.

Gansner, E.R., and North, S.C. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, 2000. John Wiley & Sons, Ltd..

Garlan, D., and Schmerl, B. Using architectural models at runtime: Research challenges. In F et al. Oquendo, editor, *Proceedings of the European Workshop on Software Architectures*, volume3047 of *Lecture Notes in Computer Science (LNCS)*, pages 200–205, St. Andrews, Scotland, May 2004. Springer-Verlag.

Garlan, D., Cheng, S.-W., and Schmerl, B. Increasing system dependability through architecture-based self-repair. In de Lemos, Gacek, and Romanovsky, editors, *Architecting Dependable Systems*. Springer-Verlag, 2003.

Good, N., Dhamija, R., Grossklags, J., Thaw, D., Arnowitz, S., Mulligan, D., and Konstan, J. 2005. Stopping Spyware at the Gates: A User Study of Privacy, Notice, and Spyware. Proc. Symposium on Usable Privacy and Security

Greenberg, S. and Marwood, D. 1994. Real-time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'94 (Chapel Hill, NC), 207-217.

Heer, J., Card, S.K., Landay, J.A. Prefuse: a toolkit for interactive information visualization. in *CHI 2005, Human Factors in Computing Systems* (2005)

Ingstrup, M. *Declarative Architectural Reflection in Distributed Systems.* (submitted).

Ingstrup, M., and Hansen, K.M. *A Declarative Approach to Architectural Reflection*. Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture. Pittsburgh, Pennsylvania, USA. November, 2005.

Ito, M. and Okabe, D. 2005. Technosocial Situations: Emergent Structurings of Mobile Email Use. In Ito, M., Okabe, D., and Matsuda, M. (eds). Personal, Portable, Pedestrian: Mobile Phones in Japanese Life, 257-273. Cambridge, MA: MIT Press.

Jiang, X., Hong, J., and Landay, J. 2002. Approximate Information Flows: Socially-based Modeling of Privacy in Ubiquitous Computing. Proc. Intl. Conf. Ubiquitous Computing Ubicomp 2002 (Gotenberg, Sweden).

John, B.E., and Bass, L. Usability and Software Architecture. *Behaviour and Information Technology*, 20(5):329–338, 2001. Taylor & Francis Ldt.

Kiczales, G., des Rivieres, J., and Bobrow, D. 1991. The Art of the Metaobject Protocol. MIT Press.

Klas, W. and Schrefl, M. *Metaclasses and their Application: data model tailoring and database integration*. Lecture Notes in Computer Science. Springer-Verlag, 1995.

Kon, F., Costa, F., Blair, G.S., and Campbell, RH. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

Kyng, M. and Henderson, A. 1991. There's No Place Like Home: Continuing Design in Use. In Greenbaum and Kyng (eds), Design at Work: Cooperative Design of Computer Systems. Hillsdale, NJ: LEA.

Langheinrich, M. 2001. Privacy by Design: Principles of Privacy-Aware Ubiquitous Systems. Proc. Intl. Conf. Ubiquitous Computing Ubicomp 2001 (Atlanta, GA).

Maes, P. Concepts and experiments in computational reflection. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155. ACM Press, 1987.

Oreizy, P., Gorlick, M., Taylor, R. N. Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., and Wolf, A.L.,. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, may-june 1999.

Otsason, V., Varshavsky, A., LaMarca, A., and de lara, E. 2005. Accurate GSM Indoor Localization. Proc. Intl. Conf. Ubiquitous Computing Ubicomp 2005 (Tokyo, Japan), 141-158. Springer.

Randell, R. 2003. User customization of medical devices: The reality and the possibilities. Cognition, Technology, and Work.

Rao, R. 1991. Implementation Reflection in Silica. Proc. European Conf. Object-Oriented Programming ECOOP'91 (Geneva, Switzerland). Springer.

Schultz, U.P., Corry, E., and Lunk, K.V. Virtual Machines for Ambient Computing: A Palpable Computing Perspective. ECOOP'05 OT4AmI workshop, Glasgow, U.K., 2005.

Shaw, M. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *Proceedings of Workshop on Studies of Software Design*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

Smith, B.C. 1982. Reflection and Semantics in a Procedural Language. Laboratory for Computer Science Technical Report MIT-TR-272, MIT.

Stroud, R. and Wu, Z. 1995. Using Metaobject Protocols to Implement Atomic Data Types. Proc. European Conf. Object-Oriented Programming (Aarhus, Denmark), 168-189. Springer.

Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Ltd, Essex, England, 1997.

Tan, J., Ewald, C.A., Zaslavsky, A., and Bond, A. Domain-specific metamodels for heterogeneous information systems. In *Proceedings of the Thirty-Sixth Annual Hawaii International Conference on System Sciences*, Big Island, Hawaii, 2003. IEEE Computer Society.

Tentori, M. Favela, J., Gonzalez, V.M., and Rodríguez, M.D. Supporting Quality of Privacy (QoP) in Pervasive Computing. Proceedings of the Sixth Mexican International Conference on Computer Science (ENC'05), IEEE.

Vaghi, I., Greenhalgh, C., and Benford, S. 1999. Coping with Inconsistency Due to Network Delays in Collaborative Virtual Environments. Proc. ACM Symp. Virtual Reality Systems and Technology VRST'99.

Westin, A. *Privacy and Freedom.* Atheneum Press, 1967.

Whitten, A. and Tygar, D. 1999. Why Johnny Can't Encrypt: A Usability Case Study of PGP 5.0. Proc. 8th USENIX Security Symposium.

Zurko, M. 2005. User-Centered Security: Stepping up to the Grand Challenge. Proc. IEEE Annual Computer Security Applications Conference, 187-202.