Component-oriented approaches to context-aware computing

Simon Dobson

Department of Computer Science, Trinity College, Dublin IE
simon.dobson@cs.tcd.ie

1 Motivation and goals

Context-awareness is emerging as an essential component of many user-focused software domains. It is especially integral to *pervasive* or *ambient* computing, but can be used to control the behaviour of any system that adapts to the circumstances in which it is used.

Like most new software projects, many existing context-aware systems have been constructed using object- or component-oriented programming techniques. Experience has shown that objects have difficulty in addressing some of the facets of highly adaptive and highly contextualised systems. These include:

- the need to create object views over information with significant ontological structure;
- consistently supporting multiple views of the same information at different levels of abstraction;
- the complexity of selecting and matching components and interfaces; and
- the mixing of concerns across different levels of the design space.

Many of the techniques being used *ad hoc* in context-aware applications might be better captured in tools, languages or methods; conversely new developments in infrastructure might be helpful – or not! – to the developers of context-aware systems.

Goals and organisation

The workshop was organised to address these issues. We requested short contributions in two strands:

1. Research and practitioner contributions on topics highlighting both the contributions object and component technology makes to context-aware distributed computing and the issues and shortcomings of current approaches
2. "What if we could" systems that can be used as a basis for case studies to be expanded during the workshop to drive discussion

We received a wide range of contributions covering middleware issues, interface adaptation, navigating large information spaces and embedded systems, each highlighting some particular aspect of constructing context-aware adaptive systems from components.

The workshop was divided into three parts: short presentations and two discussion groups exploring a different issue in more detail. The rest of this report summarises the contributions (section 2), explore two discussion topics (section 3), and offers some conclusions (section 4). Appendix A contains the names and contact details of the participants.

2 Summary of contributions

Three major themes were identifiable in the contributions:

- contributions focusing on the *extension of object models to provide context-awareness* within a largely standard programming infrastructure;
- work on *component selection and discovery*, and the problems inherent in performing these tasks in an open environment; and
- systems *integrating software closely with real-world artefacts* which need to provide virtual analogues of physical capabilities.

In introducing the workshop, Simon Dobson posed four questions for context-aware systems:

1. How do we engineer systems that are stable under minor perturbations?
2. How can we compose components into larger functions in a way that users can grasp clearly?
3. How do we balance information collected to inform adaptability against the privacy and security concerns arising from its abuse?
4. What does it mean for an adaptive system to be "correct", when some of its function derives from the changing environment?

Most of these questions occur in "ordinary" systems engineering with components and distributed objects[1], but are thrown into high relief by the introduction of context and adaptability. The use of components can be used to address issues of composition and correctness, but it is not sufficient for a context-aware system to behave correctly in any given context: it must also behave correctly dynamically, retaining continuity and stability for users even in the presence of adaptation.

Extending object models involves allowing objects to sense and adapt to context, without compromising the encapsulation properties of the underlying object model. Aline Senart gave an overview of the "sentient objects" model for pervasive systems. A sentient objects system is composed of a number of entities encapsulating a local view of the system's context and communicating *via* events. The event system is built around a scalable core that provides location-awareness, robust group management, and a predictive routing infrastructure for managing the sporadic disconnections of devices from the network.

From a programming perspective, sentient objects provide a local context model in which contexts are arranged into a hierarchy. An inferencing system determines which context the system is in, and triggers actions based on this. The view of context is strictly local, making sentient objects behave externally like "normal" objects hiding their context-sensitivity.

The "contextors" system described by Joëlle Coutaz and Gaëtan Rey provides a well-developed approach to "plastic" user interfaces that can be re-modelled and re-presented on a range of devices without compromising the intention or usability of the interface. The system is based around the description of abstract meta-interfaces and associated meta-data, together with a family of possible implementations. As the context of the user and the task being pursued evolve, the implementation can select a different implementation of the interface that is best able to satisfy the environmental and task constraints, and re-map the interface without losing application continuity.

Contextors can both produce and consume context and communicate *via* events. They are best regarded as context transformers – closer to actors than fully stateful objects.

The problem of component discovery is made even more complex by populations of adaptive components whose matching requirements follower a richer model. Peter Rigole presented a model of resource-aware components in which each component declares the external resources it needs in its interface. The interface then forms a contract which is satisfied and settled when the component is instanciated. The supporting middleware signs the contract to confirm that its requested resources are available, allowing better run-time confidence in and analysis of the component system. Component requirements may be parameterised by the context, allowing run-time adaptation through the re-negotiation of the components satisfying the contract or altering the contract as the task evolves.

Maomao Wu presented a version of the standard Microsoft COM component framework extended with interface meta-data and rules for composition, using the CLIPS engine for inference and matching. A number of factors make component composition harder in context-aware systems including the increased intelligence and adaptability of the individual components, their increased autonomy, and the time- and safety-criticality of some applications.

Otso Virtanen addressed the problem of terminal integrity, ensuring that an adaptive end-point device retains the capabilities and assurances necessary to remain integrated into a full network. Components are downloaded onto the device using a secure protocol from a component broker, which manages the dependencies of the various components sets.

In an extensible and dynamic environment, however, it is impossible to guarantee the correct functioning of *every* combination of components. Incompatible combinations are recognised at the device, flagged and reported to the component broker to prevent the propagation of clashes across the system. This improves the integrity of users' devices, since faulty compositions are less likely

to propagate once a problem has been detected. It also allows "roll-back" of devices to component populations known to be stable.

Dhouha Ayed described an extension of the CORBA component model to express assembly descriptions of components. An assembly descriptor includes information allowing the choice of components to be constrained by location, structure, type and so forth, and may be linked to a discovery service to facilitate more adaptive component location. A rule-based selection system and adapter-filter architecture allows components to be connected in the presence of minor inconsistencies in their interfaces.

Many context-aware systems include a close correspondence between physical objects and informational activities, means that the underlying context model is more closely aligned to individual objects rather than to situations. Trung Dung Ngo reported on embedding sensing and processing into LEGO bricks. Connecting bricks together provided automatic composition, to (for example) use a sensor to directly control a motor. While limited in the complexity of possible compositions, the system is easy enough for children to build with – and indeed the children can work out how the bricks work together themselves, with only a minimal explanation. Other applications include cognitive rehabilitation and other sensory integration therapies.

Micael Sjölund described SensAid, a tablet PC platform for experimenting with mixed physical and virtual entities. A room containing artefacts augmented with RFID tags allows physical objects to have a corresponding virtual presence carrying their meta-data and allowing interaction through the SensAid desktop (or tablet-top) application.

3 Discussions

As seeds for discussion, the workshop addressed two issues arising from the submissions:

1. How is adaptability best represented, presented and understood, both by users and developers?
2. How can designers of middleware and platforms help support the continuity of user experience in the face of adaptation while keeping the complexity of applications development under control?

3.1 Adaptation

Adaptability means changing some aspect of a system's detailed behaviour while keeping the gross behaviour of the system consistent. From a contextual systems perspective, adaptability means matching behaviour to changes in environment, task, user population, preferences or some other factor; from a component systems perspective, it means selecting and/or configuring the component set to provide the optimum behaviour. There are any number of design solutions that can be applied to this process, with the design space being governed by the issues (among others) discussed below.

Context as process *versus* context as data Many systems use a context model based around *context-as-data*: the model stores a representation of the state of the world as seen by the application and its sensors, which is then used to inform the behaviour of applications. Adaptability comes from the way in which components respond to changes in the context model, or from how they are selected and interconnected based on it. This approach tends to lay stress on responses to the changing environment.

An alternative view is to adopt a model of *context-as-process*, where the context is the task the user is involved with – a task inferred from the sensor (and other) data. This can be used to provide a stronger sense of continuity, in that the task (and hence the users' goals) are more clearly articulated, and can therefore lay stress on the responding to the changing task priorities[2].

There does not seem to be a clear-cut case for either view as being more powerful or natural. While using the context-as-process model potentially offers a more holistic an continuous view around which to structure adaptation, such models are often more error-prone trough having to infer the task from sensor data. Conversely the context-as-data view can fragment adaptations by not retaining the "flow" of a task-level interaction. Integrating both views in the same model may lead to conflicts between different levels.

Are the components context-aware? Another dimension in systems design is where in the design the adaptability resides. The design space lies between two extremes:

1. A static collection of components is constructed to handle the task, with each component adapting itself to the changing context
2. Each component presents a fixed behaviour, and the collection of components is changed according to the environment by some external agent

The systems described in the presentations fell into both categories, but with an emphasis on the former.

An important notion in this area is that of open- *versus* closed-adaptive systems, deriving from the work of Rick Taylor and others (*e.g.* [3]). A closed-adaptive component provides a set of adaptive behaviours itself that can be selected; an open-adaptive component accepts new behaviours from outside. Closed-adaptive systems are less flexible over the long term but possibly more reliable, stable and secure than open-adaptive components.

Open-adaptive components offer a better long-term solution for choreography and re-purposing, but would accentuate the need to pay careful attention to fallback and roll-back behaviours if an adaptation proved unsuitable.

Choosing the "right" components In either of the above cases there is an issue in choosing the correct components for a given situation – a decision that may be repeated over the application's lifetime.

Many of the participants work in this area. The main approaches involve adding meta-data to component interfaces – something that should probably be emphasised and standardised as part of the evolution of component frameworks.

The decision process itself is very subtle, since most systems will not have access to all the information that they could potentially use. Several systems are rule-based, although it was recognised that "crisp" logic may not be the ideal way to deal with the inherent uncertainties.

An uncertain process must face the possibility that a decision is made incorrectly. The impact of wrong decisions can vary, from preventing the system working (with possible loss of data) to generating a system that works sub-optimally. Systems that select component populations dynamically are able to recover from non-fatal compositions by re-selecting as soon as the problem arises. More static composition systems require more explicit recovery strategies.

Semantic compatibility Component selection depends on a degree of semantic compatibility between the various components, either inherent or introduced through adapters.

Again, one may take a static on dynamic position on the issue. The extreme static position is to assume that interface meta-data is an infallible guide; the extreme dynamic position is that consistency checks are continuously required to ensure that the components are working together. What is important, however, is to embrace the fact that compatibility is an issue in the on-going evolution of systems and to address it in the basic structure of the design.

Complex components *versus* choreography Another aspect of component selection involves the relative "weight" or "intelligence" of components. Components may perform a single, simple function, or they may perform a larger set of functions. A good example is two ways of building a messaging system: using components that each support a single messaging protocol (e-mail, SMS, IM), or as a single messaging component offering all protocols. Both are valid decision decisions, and both are orthogonal to the issue of whether the components offer adaptive interfaces to their services.

Using large populations of small components involves quite complex co-ordination (sometimes referred to as "choreography"), but allows very fine-grained adaptation. Larger components allow simpler plumbing but force the system developer to take larger "chunks" of functionality.

This argument is common in all component- or object-based systems. It has a particular impact for contextual systems in which the component composition decisions are uncertain and may be hard to undo.

One part of the design space that is definitely concerning is large components offering adaptive interfaces internally. There is a real risk that the component will be selected for functional reasons (*i.e.* it offers messaging functions), but will not offer the best adaptive interface functions. Separating these two concerns is therefore an important goal for populating a component framework.

Autonomous *versus* human-in-the-loop One of the goals of pervasive computing is to be able to offer some services autonomously, matched to context. This includes adapting the user interface. The question is whether, and to what extent, services can or should be provided without user intervention, and how to integrate the human into the loop in a natural fashion.

This question divides into two parts. At an interface level it may be possible (and indeed preferable) to provide for user "guidance" of adaptations where possible. At the system level, imprecise reasoning may require disambiguation by the user. In effect this is a "whole system" issue rather than being tied to components *per se*.

Observability of relevant state in the interface Closely related to the above is the inclusion of relevant state in the interface. It is evident that human decision-making relies on presenting the information needed for the decision; it also seems to be the case that autonomous or semi-autonomous adaptations should be indicated in the interface, either before ("this change may happen") or after ("this is what just happened").

3.2 Continuity and complexity

If the "software crisis" challenges our ability to build correct software for desktop and server-based systems, then context-aware systems raise the bar even higher. Engineering such systems means that we need to identify the areas of context-awareness that generate additional complexity, and develop ways to keep it under control.

Complexity has a user dimension as well as a programmer one. A system that adapts too often may appear to "flicker" and prevent users forming a coherent mental model of its behaviour and services. This is why it is important to mirror the users' notions of the continuity of an interaction across modifications in detailed behaviour.

A number of major issues affecting complexity are discussed below.

Retain task models at run-time Maintaining continuity of interaction across adaptation relies either on knowing that adaptations inherently preserve continuity or on being able to intervene to preserve the experience. The former is difficult to conceive of in an open system; the latter is considerably simplified by keeping task models available at run-time.

"Introspective" systems architectures are moderately common, occurring on a small scale in reflective programming languages and on a large scale in facilities management systems such as Tivoli. It is far less common to encounter run-time descriptions of the use cases or tasks the system addresses. However, having an abstract, machine-readable description of the task being supported by a system makes it far easier to ensure that adaptations "maintain the place" in the workflow.

Separate functions from interface Interface adaptations form a large part of the adaptations a system makes to changes in context – although by no means all the possible adaptations involve interface changes.

The separation of function from interface is a common one in many systems, being the basis for the Model-View-Controller (MVC) design pattern. Maintaining this separation into adaptive and multi-modal interfaces – although challenging – is essential to allow tested functionality to be re-used under different usage models.

This design constraint lends itself well to open-adaptive and choreography-based component strategies, since one may select base behaviour and then provide multiple changing interfaces without modifying it.

Make "seams" observable A lot of pervasive computing is characterised as "seamless" interaction, characterised as meeting Mark Weiser's vision of presented the right service at the right time in the right way with minimal cognitive load[4].

However, there are many ways in which the world is inherently "seamed", and these seams should be reflected in the behaviour of a context-aware system. The key observation is that discontinuous behavioural change should occur in response to clear changes in the environment. A system may change interface mode better to match the task or environment: but if this change introduces a cognitive disruption then there should be a clear rationale for the change that the user can relate to. There are strong arguments for elevating this notion to being a fundamental design driver for tools and languages (see for example [5]).

Externalise strategies Any function that is embodied solely as code is essentially a black box that can only be manipulated according to its meta-data. While essential for core low-level tasks, higher-level co-ordination tasks can often be externalised as process descriptions. A number of emerging standards exist in this domain, often targeted at web services. While perhaps not completely applicable to pervasive computing as they stand, they offer a promising direction.

As well as task descriptions of this kind, adaptation also requires adaptation strategies that are followed to determine component configurations or reconfigurations. Again, if these strategies are articulated as descriptions rather than code, they can be analysed and manipulated by other strategies to solve conflicts or introduce additional concerns. A good example might be delaying an adaptation that will radically change the system's interaction mode until the user reaches a natural "break" in the task.

4 Conclusions and recommendations

If one had to distill three recommendations for systems design from the presentations and discussions, they would be the following:

Keep components simple Simple components allow greater flexibility in composition. The separation of user interface from function – a "contextualised MVC" – provides better adaptability than larger-grained components with hard-wired adaptation.

Externalise adaptations A systems' reaction to contextual changes is itself a subject for analysis. The strategies and decision processes used should be represented explicitly and in machine-readable form to allow second-order effects to be generated.

Retain a global view Contextual systems have an unavoidable holism, which implies taking a global view on the system and its behaviour as well as a local view per-component. This includes a view of the tasks the system is supporting and the way it is supporting them. The global view should be clearly expressed rather than being implied.

In conclusion, a wide range of techniques from object- and component-oriented software engineering are contributing strongly to the development of context-aware systems. Several extensions are needed, especially in handling dynamic component composition in the face of imperfect information, and in maintaining a continuous user experience with adaptive interfaces. Some of these extensions can be provided conservatively by adding meta-data and processing to component interfaces within largely standard frameworks; others require a hange in the way we think about, design and analyse component systems and their relationship to their operating environment.

References

1. Enmerich, W.: Engineering distributed objects. Wiley (2000)
2. Dobson, S.: Applications considered harmful for ambient systems. In: Proceedings of the International Symposium on Information and Communications Technologies, ACM Press (2003) 171–176
3. Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilliei, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14** (1999) 54–62
4. Weiser, M.: The computer for the 21st century. Scientific American (1991)
5. Dobson, S., Nixon, P.: More principled design of pervasive computing systems. In: Proceedings of Engineering for Human-Computer Interaction and Design, Specification and Verification of Interactive Systems (EHCI-DSVIS'04). LNCS, Springer-Verlag (2004) To appear.

A Participants

Name	Affiliation
Dhouha Ayed	CNRS INT-Evry, FR dhouha.ayed@int-evry.fr
Joëlle Coutaz	CLIPS-IMAG, FR joelle.coutaz@imag.fr
Simon Dobson	Trinity College, Dublin IE simon.dobson@cs.tcd.ie
Jasminka Matevska-Meyer	University of Oldenburg, DE matevska-meyer@informatik.uni-oldenburg.de
Trung Dung Ngo	University of Southern Denmark, DK dungnt@mip.sdu.dk
Gaëtan Rey	CLIPS-IMAG, FR gaetan.rey@imag.fr
Peter Rigole	KU Leuven, BE peter.rigole@cs.kuleuven.ac.be
Aline Senart	Trinity College Dublin, IE aline.senart@cs.tcd.ie
Micael Sjölund	Linköping University, SE x03micsj@ida.liu.se
Otso Virtanen	Helsinki Institute for Information Technology, FI otso.virtanen@cs.helsinki.fi
Maomao Wu	University of Lancaster, UK maomao@comp.lancs.ac.uk

Copies of all the submitted papers and presentations may be found on the workshop web site, http://www.cs.tcd.ie/COA-CAC-04/.

B Programme committee

Simon Dobson	Trinity College Dublin, IE
Paddy Nixon	University of Strathclyde, UK
Siobhán Clarke	Trinity College Dublin, IE
Achilles Kameas	University of Patras, GR
Dominic Duggan	Stevens Institute of Technology, US
Gaetano Borriello	University of Washington, US