LANCASTER
UNIVERSITY

Workshop on Software Engineering Challenges for Ubiquitous Computing

June 1st – 2nd 2006
Lancaster University

Supported by UKUbiNet

Edited by Gerd Kortuem

SEUC 2006 Workshop Programme

Papers Not Presented

SEUC 2006 Workshop Programme

Session 1

Engineering for real – The SECOAS project

I.W.Marshall (University of Kent), **A.E.Gonzalez** (University College London), **I.D.Henning** (University of Essex), **N.Boyd** (Salamander Ltd), **C.M.Roadknight** (BT plc), **J.Tateson** (BT plc), **L.Sacks** (University College London)

Introduction

SECOAS was funded by the DTI as part of the Envisense centre (pervasive computing for natural environments) within the Next Wave Technologies and Markets initiative. The objective was to deploy a sensor network to monitor sedimentation processes at small scales in the area of Scroby Sands just off the coast at Great Yarmouth, Norfolk. Scroby Sands is the site of a wind-farm, and the DTI business case was based on improving the monitoring and impact assessment of offshore infrastructure. The research motivation from the academic side was to demonstrate the potential of a range of collegiate AI ideas that the group had previously simulated [1,2,3].

Successful deployment of a sensor network in a natural environment requires the devices to survive for long periods, without intervention, despite the fact that the conditions encountered are very likely to be antithetical to electronic devices. This means that any deployed devices must be both robust and able to deal with partial failures gracefully, without requiring large amounts of power. The project team believe that embedded AI is a good solution to enabling adaptation to failure and power management, since many AI algorithms are known to be tolerant to partial inputs and noise, and our own simulations had shown this to be true of the particular algorithms we were aiming to test. In other words the approach was likely to lead to reasonably robust software. On the other hand it is not possible to prove properties of this type of algorithm, and simulations can never capture the full complexity of reality (and are thus only indicative). It is therefore necessary to test ”in situ”, by undertaking a real deployment. SECOAS was designed to combine the software expertise of Kent, UCL and BT with the hardware expertise of Plextek, Essex and Salamander, and provide robust hardware and a challenging scenario that would represent a good test of the AI based approach.

What Happened

During the life of the project there were 3 full trials of sensing nodes, an initial deployment of one node (measuring Pressure, Temperature, Turbidity and Conductivity) for one week, an early deployment of 5 nodes for 2.5 weeks, and a final deployment of 10 nodes for 2 months. In all cases the AI algorithms performed well, and further tests are certainly justified. However none of the tests allowed an exhaustive characterisation of the software performance since the rate of failure of the nodes was significantly higher than expected. During the initial deployment (intended as a technology trial) no problems were observed. During the 5 node deployment one node failed completely and one node failed after 24 hours. Both failures were due to water ingress at an unplanned cable joint, introduced during deployment for operational reasons. No other hardware failures were observed. These trials gave the team confidence that the hardware was reasonably robust and had a good chance of surviving the planned final deployment of 2 months, providing the equipment was deployed as intended. In the final trial however 4 nodes were destroyed through external intervention, 2 nodes failed as a result of water ingress down a weak antenna cable (that was not intended to be submerged), and the remaining 4 nodes lasted for only 4 weeks. As a result the statistics generated by the recovered data samples are not sufficient to make conclusive claims about the software performance (remember the measured parameters exhibit long range dependency as a result of the turbulent flows, so very large sample sizes are needed). However the project did return some interesting oceanographic data, and some useful lessons for pervasive system engineering. These lessons are briefly outlined in the next section.

Lessons

A major difficulty faced by the project was aligning the language and methodologies of the software team and the hardware team. There does not appear to be any established literature on hardware /software co-design of extended systems of this nature, and the team had to create its own ad-hoc solutions (we spent a lot of time engaged in cross-disciplinary training). The difficulty was most clearly expressed close to deadlines when the software and hardware engineers had no mechanisms for dealing with the instability of each others outputs.

A second related difficulty was enabling full understanding of the limitations and failure mechanisms of the hardware and software across the whole team. This was most clearly expressed at module interfaces, where the software engineers tended to assume module clocks were as accurate as required, and the hardware engineers assumed the clocks would operate in spec. It turned out during the first multimode trial that neither group was correct, and more sophisticated interface protocols were used in the final trial.

A third key problem was the need to consider unintended interactions with non-project participants (such as local fishermen). The team did not start with sufficient expertise in this area.

Clearly for the future it is necessary to develop a methodology that systematizes approaches and solutions to these and similar problems.

We acknowledge the Support of the DTI and our industry sponsors (BT, Salamander and Plextek) and the contributions of many other team members during the life of the project. We particularly note the assistance of UEA with the oceanographic aspects of the work.

[1] A Novel Mechanism for Routing in Highly Mobile Ad-hoc Sensor Networks. J Tateson and I W Marshall. In Willig Karl and Wolisz, editors, *Wireless Sensor Networks*, number 2920 in LNCS, 2004.
[2] A Weekly Coupled Adaptive Gossip Protocol for Application Level Active Networks. I Wokoma, I Liabotis, O Prnjat, L Sacks, and I Marshall. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 02)*, Monterey, California, 2002.

[3] Emergent organization in colonies of simple automata. IW Marshall and CM Roadknight. In J Kelemen and P Sosik, editors, *Advances in Artificial Life*, number 2159 in Lecture Notes in Artificial Intelligence, pages 349-356. Springer Verlag, 2001.

Analysing Infrastructure and Emergent System Character for Ubiquitous Computing Software Engineering

Martin Randles, Prof. A. Taleb-Bendiab
m.j.randles@2004.ljmu.ac.uk a.talebbendiab@ljmu.ac.uk

School of Computing and Mathematical Sciences, Liverpool John Moores University, Liverpool, L3 3AF, UK

Introduction

A complete and rigorous understanding of the behaviour and nature emanating from complicated interconnected systems provides a major technological challenge, at present. In ubiquitous computing the character and programming model of the individual components is well understood. However this is little help in predicting or allowing for the behaviour of the macro-scale entire system or for protecting against the cascading failure subsequent to some apparently harmless device breakdown. These failures are of greater concern due to the increasing application of ubiquitous computing systems in critical real world situations. Thus the future engineering requirements, of these systems, must involve a model that encompasses the simple behaviour and actions of the individual ubiquitous computing devices and the emergent large scale behaviour of the complex interconnected system. To this end this work proposes using known results; applicable to engineering predictable macro-scale behaviour in complex systems, together with associated (most likely aggregated) cognitive systems to monitor and influence the system behaviour and evolution. Additionally this will be presented in a formalism that specifies the low scale programming models of the individual components whilst simultaneously providing the deliberative functionality necessary for the cognitive systems to reason analyse and influence the whole system. This in turn leads to future work in identifying and assessing further characteristics, dimensions and metrics applicable to such systems. This formal account is then easily translated into software code.

Engineering Behaviour

Current methodologies are adept at engineering system microscopic features. So, for instance, embedded devices conform and function according to clearly defined rules. They interact and exchange data according to some engineered set of regulations. However there is no analogous macroscopic engineering focus. It is, of course, necessary to specify the low level functioning and interactions of the ubiquitous networked devices. However, for organisation via emergent system features, the lack of any coherent methods to explicitly deal with macroscopic system properties is a serious shortcoming.

Techniques to Better Serve Ubiquitous Systems

It is necessary to consider suitable architectures and formalisms in which to express, analyse and represent ubiquitous systems. In this way it is possible to understand and deal with the consequences emanating from both the bottom-up device interactions and the top down influenced macro scale behaviour. Thus an observer hierarchy is proposed. That is, at any level of system granularity, a simple observer module will be extant monitoring some portion of the system. This observer can act to keep the autonomy of the participants bounded as well as influence the system's members towards micro-scale actions that are known to tend the system towards some macro-scale self-organising point. In addition such an arrangement displays self-similar properties that are in general not present for large scale complex systems.

For reasons of scalability (no state space enumeration), correctness property specification (through deduction) and knowledge utilisation it is proposed that a propositional account of a system is best. In this way the programming model of the embedded devices can be described, the norms and bounds affecting the interactions and operations are easily enacted whilst large scale behaviour can be monitored and deliberated upon through the logical entailment process. The Situation Calculus has been used throughout this work as the first choice propositional medium.

Emerging Network Characteristics

The point of promoting self-organising behaviour is to divest some of the control of the system to itself. That is, if the behaviour is conforming to some set organisational model then, observation need not be as stringent or highly programmed. Scale free (SF) systems represent a starting point from which to look at systems exhibiting some form of organisational behaviour that cannot be envisioned from the small scale interactions. The main properties of these systems, as detailed in the literature, are: SF systems have scaling (power law) degree distributions, they can be generated by certain stochastic processes (the most widely regarded being preferential attachment), SF networks consist of highly connected hubs, SF networks are generic in that they are preserved under degree preserving rewiring, they are self-similar and are universal in that there is no dependence on domain specific data. There is still much confusion over which of these properties is necessary and/or sufficient to entail the others. The SF property is variously said to occur when there is a scaling or power law distribution present or when generation occurs based on incremental growth and preferential attachment. There is no rigorous definition for scale-free behaviour. There is also no clear comprehension of how certain properties and mechanisms lead to such effects as power law tail distributions. Some researchers expound Self-Organising Criticality (SOC) or Edge of Chaos states as forces that tend to migrate systems towards this behaviour. Here at a certain threshold a bifurcation point exists between a safe, predictable states and complete chaos. The existence of power laws is said to represent a signature for such states. Other researchers stress more the role of optimised design in producing these effects. Systems optimized to be highly tolerant to perceived threats, known as The Highly Optimized Tolerance or Heuristically Organised Trade-off (HOT) systems, lead to power laws in the tails of degree distributions. Here systems are designed to optimise in the presence of constraints and uncertainty. It has been observed that the high performance and robustness of the optimised design, taken with the uncertainty the systems were designed for, leads to extreme sensitivity to additional uncertainty not recognised in the original design. It seems SOC type systems are less susceptible to rule changes than HOT systems but this is compensated for in less overall robustness. It is noted that complexity appears to be derived from some involved and deeply embedded trade-offs between robustness and uncertainty. These are fundamental conservation principles that may, in future, prove as important to ubiquitous system design as matter energy and entropy are to physical systems.

Example Application

This approach is especially relevant when considering planetary wide architectures that may occur as peer-to-peer links through ubiquitous computing devices. In this case the observer model may comprise a sensor-actuator overlay. The approach proposed here gives high assurance for both the systems development and the life time management because of the appeal to mathematical logic. Additionally complexity and scalability is handled via the self-organising principles and the behaviour entailing propositional account. Such a sensor-actuator overlay may be specified and engineered for use within the Planetlab environment where sensors may be added, edited or discovered to conform to a model for self-organisational robustness via the observer model.

Software Considerations for Automotive Pervasive Systems

Ross Shannon, Aaron Quigley, Paddy Nixon
{ross.shannon, aaron.quigley, paddy.nixon}@ucd.ie

Abstract

The pervasive computing systems inside modern-day automobiles are made up of hundreds of interconnected, often replaceable components. These components are put together in a way specified by the customer during manufacturing, and can then be modified over the lifetime of the automobile, as part of maintenance or upgrading.

This flexibility means that system implementers cannot know in advance which of a wide variety of configurations they are programming for, and so the software system must be designed in a way that is agnostic of implementation details.

1 Introduction

Many modern automobiles contain hundreds of embedded microcontrollers [2]. The automobile industry has seen a shift towards the use of more on-board technology and, as such, is becoming increasingly software-dependant. From sophisticated navigation systems to computer-controlled driver-assistance safety systems and in-car multimedia and entertainment, the amount of software written for cars is increasing rapidly.

These systems work in concert across the Controller Area Network (CAN) [1], seamlessly passing data from the sensory system [3] of the car (constantly measuring factors like speed, in-car temperatures and rainfall), to the actuator system, which will perform actions like augmenting the operation of the breaking mechanisms, maintaining air-conditioning and controlling the audio-visual system.

Though embedding multiple microcontrollers is more cost-effective and facilitates more reuse than designing a central control system of powerful microprocessors, there is an associated cost in additional software complexity. Many components in these automobiles are designed to be replaceable to ease future maintenance of the vehicle. This means that a new component will often have a different feature-set to the component it replaces. Separate components need to be able to work together despite not always being aware of each other's capabilities. It is also likely that this modularity will give rise to a market for cheaper non-OEM components.

The requirements for such hardware and software are poorly defined and poorly understood. [6] Components must expose their interface to the rest of the system, and find suitable points where they may "hook in" to the existing system, integrate unobtrusively, and make use of and extend its functionality.

2 Component Integration

2.1 Modularity

In modern-day automotive design, cars are made to modularised, so that a customer may outfit a car to his own specifications. This means that any vehicle could come in hundreds or thousands of possible configurations, each with their own functionality and internal dependencies.

High-end models will have additional functionality, but use many of the same hardware components across the product line. For instance, a high-end model may have additional logic to control the windscreen-wipers based on a rainwater sensor at the front of the car, whereas drivers without this feature will have to engage the wipers manually. An upgrade to the car's Body Electronic Control Unit (ECU) might make this functionality available later in the car's life.

Alongside this, further features can be purchased and added to the car once it has left the factory, which should integrate seamlessly into the existing pervasive system. Consider the dashboard-mounted GPS unit. Hardware interfaces are provided so that these modules can be added to the vehicle, but oftentimes the system designer will also want to make use of this new functionality from within the current software system, if it is made available. For example, a mapping program positioned in the car's central control console which previously prompted the user to manually enter their location each time they wanted to use it can now query the GPS module automatically. Similarly, the GPS unit itself would like to have access to the car's built-in text-to-speech program so that it can provide aural feedback to the driver.

2.2 Feature Discoverability

The challenge for the designers of software within this ubiquitous system is that there is never any guarantee which components are installed at a time inside the automobile. This necessitates strong capability-checking before any code can be executed.

However, this only covers the gamut of modules that the designers knew about as they were building the system. New modules (from other manufacturers) will have capabilities that the system designers hadn't considered. For new features to integrate and be made available to the rest of the system, feature *discoverability* must be made a priority.

The hardware and software parts of a module should be thought of as a single entity, with a single interface. [4] When a new module is connected, it is required to make contact with a central directory server within the car's internal network, which will keep track of the services being provided by components within the car. This facilitates modules which would like to use each other's services being put in contact.

2.3 Ease of Integration

Adding a module to an automotive pervasive system is different than adding a new device to a standard computer system. In general, non-critical hardware components in a computer system are not expected to work together. However, in the case of automotive systems the ease of integration and extensibility of the shipping system are two major selling points.

It is for these reasons that we feel the programming paradigm of Aspect-Oriented Programming (AOP) [5] to be suitable for programming automotive pervasive systems. The hardware and software modules being added to the automobiles should already overlap in functionality as little as possible.

Ideal cross-cutting concerns present themselves, like all devices wishing to direct feedback to the driver through the automobile's central console. Similarly, many aspects of the car's safety system (tyres with pressure sensors, headlight sensors, proximity sensors) will all need access to the braking mechanism. AOP allows these concerns to be centralised, independant of the number of components that pass information to the safety system, where it is collated and acted upon.

3 Conclusion

Software engineering for automotive systems introduces new challenges and new opportunities. Unobtrusively integrating a new component requires all existing elements of the system to be alerted of the new features it supports. The new component also needs to publish a list of their capabilities to a central service within the automobile, so that other modules that would like to make use of them are able to do so.

Aspect-Oriented Programming is an ideal programming paradigm to help in solving these problems, as it allows disparate components to advise each other on desired behaviour without requiring that the components know many details about the component's implementation.

References

[1] R. Bannatyne. Controller Area Network Systems Continue to Proliferate Through Low-cost Components. *Electronic Engineering Times*, Mar 2004.

[2] R. Bannatyne. Microcontrollers for the Automobile. *Micro Control Journal*, 2004.

[3] W. J. Fleming. Overview of Automotive Sensors. *Sensors Journal, IEEE*, 1(4):296–308, 2001.

[4] J. Hennessy. The Future of Systems Research. *Computer*, 32(8):27–33, 1999.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[6] A. Möller, M. Åkerholm, J. Fröberg, and M. Nolin. Industrial grading of quality requirements for automotive software component technologies. In *Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th IEEE International Real-Time Systems Symposium*, 2005 Miami, USA, December 2005.

SEUC 2006 Workshop Programme

Session 2: Programming

Ambient-Oriented Programming: Language Support to Program the Disappearing Computer

Jessie Dedecker[*] Tom Van Cutsem[*]
Stijn Mostinckx[†] Wolfgang De Meuter Theo D'Hondt
Programming Technology Laboratory
Department of Computer Science
Vrije Universiteit Brussel, Belgium
jededeck | tvcutsem | smostinc | wdmeuter | tjdhondt@vub.ac.be

1. INTRODUCTION

The past couple of years, pervasive and ubiquitous computing have received more and more attention from academia and industry alike. Wireless communication technology and mobile computing technology have reached a sufficient level of sophistication to support the development of a new breed of applications. Mobile networks will play an important role to realize ubiquitous computing scenarios because through cooperation of different devices context can be derived. For example, a meeting can be automatically detected by a mobile phone by identifying a video-projector in a room and ask if it is beaming a video signal. This small example already shows that the cooperation between devices plays an important role to derive the a device's context. Hence, realizing the vision of ubiquitous computing entails the construction of a distributed system.

At the software-engineering level, we observe that thus far, no general stable, robust and standard ubiquitous computing platform has emerged. Moreover, although there has been a lot of active research with respect to mobile computing middleware [14], we see little innovation in the field of programming language research. Although distributed programming languages are rare, they form a suitable development tool for encapsulating many of the complex issues engendered by distribution [2, 3]. The distributed programming languages developed to date have either been designed for high-performance computing (e.g. X10 [5]), for reliable distributed computing (e.g. Argus [13] and Aeolus [20]) or for general-purpose distributed computing in fixed, stationary networks (e.g. Emerald [10], Obliq [4], E [15]). None of these languages has been explicitly designed for mobile networks. They lack the language support necessary to deal with the radically different network topology.

In this paper we take the position that a new breed of programming languages is needed to deal and manage the complexity that arises from the novel hardware constellation used to realize the vision of ubiquitous computing. In our approach we started from analyzing important phenomena exhibited by such hardware constellations. Based on this analysis we derived characteristics for a novel computing paradigm, which we have named the *ambient-oriented programming* paradigm [7], to program such systems.

2. HARDWARE PHENOMENA

Based on the fundamental characteristics of mobile hardware, we distill a number of phenomena which mobile networks exhibit. There are two discriminating properties of mobile networks: applications are deployed on *mobile* devices which are connected by *wireless* communication links with a limited communication range. The type of device and the type of wireless communication medium can vary, leading to a diverse set of envisaged applications. Devices might be as small as coins, embedded in material objects such as wrist watches, door handles, lamp posts, cars, etc. They may even be as lightweight as sensor nodes or they may be material objects "digitized" via an RFID tag[1]. Devices may also be as "heavyweight" as a cellular phone, a PDA or a car's on-board computer. All of these devices can in turn be interconnected by a diverse range of wireless networking technology, with ranges as wide as WiFi or as limited as IrDA.

Mobile networks composed of mobile devices and wireless communication links exhibit a number of phenomena which are rare in their fixed counterparts. In previous work, we have remarked that mobile networks exhibit the following phenomena [7]:

Volatile Connections. Mobile devices equipped with wireless media possess only a limited communication range, such that two communicating devices may move out of earshot unannounced. The resulting disconnections are not always permanent: the two devices may meet

[*]Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)
[†]Author funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen"

[1]Such tags can be regarded as tiny computers with an extremely small memory, able to respond to read and write requests.

again, requiring their connection to be re-established. Quite often, such transient disconnections should not affect an application, allowing both parties to continue with their conversation where they left off. These volatile disconnections do expose applications to a much higher rate of partial failure than that which most distributed languages or middleware have been designed for.

Ambient Resources. In a mobile network, devices spontaneously join with and disjoin from the network. The same holds for the services or resources which they host. As a result, in contrast to stationary networks where applications usually know where to find their resources via URLs or similar designators, applications in mobile networks have to find their required resources dynamically in the environment. Moreover, applications have to face the fact that they may be deprived of the necessary resources or services for an extended period of time. In short, we say that resources are *ambient*: they have to be discovered on proximate devices.

Autonomous Devices. In mobile wireless networks, devices may encounter one another in locations where there is no access whatsoever to a shared infrastructure (such as a wireless base station). Even in such circumstances, it is imperative that the two devices can discover one another in order to start a useful collaboration. Relying on a mobile device to act as infrastructure (e.g. as a name server) is undesirable as this device may move out of range without warning [11]. These observations lead to a setup where each device acts as an autonomous computing unit: a device must be capable of providing its own services to proximate devices. Devices should not be forced to resort to a priori known, centralized name servers.

Natural Concurrency Due to their close coupling to the physical world, most pervasive applications are also inherently event-driven. Writing correct event-based programs is far from trivial. There is the issue of concurrency control which is innate in such systems. Furthermore, from a software design point of view, event-based programs have very intricate, confusing, control flow as they are not based upon a simple call-return stack.

As the complexity of applications deployed on mobile networks increases, the above unavoidable phenomena cannot keep on being remedied using ad hoc solutions. Instead, they require more principled software development tools specifically designed to deal with the above phenomena. For some classes of applications – such as wireless sensor networks – such domain-specific development tools are emerging, as can be witnessed from the success of TinyOS [12] and its accompanying programming language NesC [8].

3. AMBIENT-ORIENTED PROGRAMMING

In the same way that referential transparency can be regarded as a defining property for pure functional programming, this section presents a collection of language design characteristics that discriminate the AmOP paradigm from classic concurrent distributed object-oriented programming. These characteristics have been described earlier [6] and are repeated in the following four sections.

3.1 Classless Object Models

As a consequence of argument passing in the context of remote messages, objects are copied back and forth between remote hosts. Since an object in a class-based programming language cannot exist without its class, this copying of objects implies that classes have to be copied as well. However, a class is – by definition – an entity that is conceptually shared by all its instances. From a conceptual point of view there is only one single version of the class on the network, containing the shared class variables and method implementations. Because objects residing on different machines can autonomously update a class variable of "their" copy of the class or because a device might upgrade to a new version of a class thereby "updating" its methods, a classic distributed state consistency problem among replicated classes arises. Independent updates on the replicated class – performed by autonomous devices – can cause two instances of the "same" class to unexpectedly exhibit different behaviour. Allowing programmers to manually deal with this phenomenon requires a *full* reification of classes and the instance-of relation. However, this is easier said than done. Even in the absence of wireless distribution, languages like Smalltalk and CLOS already illustrate that a serious reification of classes and their relation to objects results in extremely complex meta machinery.

A much simpler solution consists of favouring entirely self-sufficient objects over classes and the sharing relation they impose on objects. This is the paradigm defined by prototype-based languages like Self [18]. In these languages objects are *conceptually* entirely idiosyncratic such that the above problems do not arise. Sharing relations between different prototypes can still be established (such as e.g. traits [17]) but the point is that these have to be explicitly encoded by the programmer at all times[2]. For these reasons, we have decided to select prototype-based object models for AmOP. Notice that this confirms the design of existing distributed programming languages such as Emerald, Obliq, dSelf and E which are all classless.

3.2 Non-Blocking Communication Primitives

Autonomous devices communicating over volatile connections necessitate non-blocking communication primitives since blocking communication would harm the autonomy of mobile devices. First, blocking communication is a known source of (distributed) deadlocks [19] which are extremely hard to resolve in mobile networks since not all parties are necessarily available for communication. Second, blocking communication primitives would cause a program or device to block long-lasting upon encountering volatile connections or temporary unavailability of another device [14, 16]. As such, the availability of resources and the responsiveness of applications would be seriously diminished.

[2]Surely, a runtime environment can optimise things by sharing properties between different objects. But such a sharing is not part of the language definition and can never be detected by objects.

Non-blocking communication is often confused with asynchronous sending, but this neglects the (possibly implicit) corresponding 'receive' operation. Non-blocking reception gives rise to event-driven applications, responsive to the stream of events generated by spontaneously interacting autonomous devices. We thus conclude that an AmOP language needs a concurrency model without blocking communication primitives.

3.3 Reified Communication Traces

Non-blocking communication implies that devices are no longer implicitly synchronised while communicating. However, in the context of autonomously collaborating devices, synchronisation is necessary to prevent the communicating parties from ending up in an inconsistent state. Whenever such an inconsistency is detected, the parties must be able to restore their state to whatever previous consistent state they were in, such that they can synchronise anew based on that final consistent state. Examples of the latter could be to overrule one of the two states or deciding together on a new state with which both parties can resume their computation. Therefore, a programming language in the AmOP paradigm has to provide programmers with an *explicit representation* (i.e. a reification) of the communication details that led to the inconsistent state. Having an explicit reified representation of whatever communication that happened, allows a device to properly recover from an inconsistency by reversing (part of) its computation.

Apart from supporting synchronisation in the context of non-blocking communciation, reified communication traces are also needed to be able to implement different message delivery policies. A broad spectrum of such policies exists. For example, in the M2MI library [11], messages are asynchronously broadcasted without guarantees of being received by any listening party. In the actor model on the other hand, all asynchronous messages must eventually be received by their destination actor [1]. This shows that there is no single "right" message delivery policy because the desired delivery guarantee depends on the semantics of the application. Reifying outgoing communication traces allow one to make a tradeoff between different delivery guarantees. Programming languages belonging to the AmOP paradigm should make this possible.

3.4 Ambient Acquaintance Management

The combination of autonomous devices and ambient resources which are dynamically detected as devices are roaming implies that devices do not necessarily rely on a third party to interact with each other. This is in contrast to client-server communication models where clients interact through the mediation of a server (e.g. chat servers or white boards). The fact that communicating parties do not need an explicit reference to each other beforehand (whether directly or indirectly through a server) is known as distributed naming [9]. Distributed naming provides a mechanism to communicate without knowing the address of an ambient resource. For example, in tuple space based middleware this property is achieved, because a process can publish data in a tuple space, which can then be consulted by the other processes based on a pattern matching basis. Another example is M2MI [11], where messages can be broadcast to all objects implementing a certain interface.

We are not arguing that all AmOP applications must necessarily be based on distributed naming. It is perfectly possible to set up a server for the purposes of a particular application. However, an AmOP language should allow an object to spontaneously get acquainted with a previously unknown object based on an intensional description of that object rather than via a fixed URL. Incorporating such an acquaintance discovery mechanism, along with a mechanism to detect and deal with the loss of acquaintances, should therefore be part of an AmOP language.

4. POSITION

The defining characteristics of the AmOP paradigm influence the structure of the applications written in the paradigm. For example, the non-blocking communication characteristic results in an event-driven programming model. Programs written based on such an event model are usually complex because the code is cluttered with callbacks. Callbacks force the developer to encode the continuations of an application manually and this complicates the code enormously. Our position is that languages with explicit support to deal with these hardware phenomena are a necessary first step to reduce the complexity of software that enables ubiquitous computing. For this reason our research focusses on finding language features that reduce the program complexity brought by the different hardware phenomena. An AmOP language that serves as a laboratory to experiment with such language features has already been reported on [7].

5. REFERENCES

[1] Agha, G. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. Programming languages for distributed computing systems. *ACM Comput. Surv. 21*, 3 (1989), 261–322.

[3] Briot, J.-P., Guerraoui, R., and Lohr, K.-P. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys 30*, 3 (1998), 291–329.

[4] Cardelli, L. A Language with Distributed Scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Press, pp. 286–297.

[5] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2005), ACM Press, pp. 519–538.

[6] Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., and De Meuter, W. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), ACM Press.

[7] Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., and De Meuter, W.

Ambient-oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)* (2006), D. Thomas, Ed., Lecture Notes in Computer Science, Springer. To Appear.

[8] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language: a holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2003).

[9] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan 1985), 80–112.

[10] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.

[11] Kaminsky, A., and Bischof, H.-P. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM Press, pp. 72–73.

[12] Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E. A., and Culler, D. E. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of the first Symposium on Networked Systems Design and Implementation (NSDI 2004)* (March 29-31 2004), USENIX, pp. 1–14.

[13] Liskov, B. Distributed programming in Argus. *Communications Of The ACM 31*, 3 (1988), 300–312.

[14] Mascolo, C., Capra, L., and Emmerich, W. Mobile Computing Middleware. In *Advanced lectures on networking*. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[15] Miller, M., Tribble, E. D., and Shapiro, J. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing* (April 2005), R. D. Nicola and D. Sangiorgi, Eds., vol. 3705 of *LNCS*, Springer, pp. 195–229.

[16] Murphy, A., Picco, G., and Roman, G.-C. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (2001), IEEE Computer Society, pp. 524–536.

[17] Ungar, D., Chambers, C., Chang, B.-W., and Hölzle, U. Organizing programs without classes. *Lisp Symb. Comput. 4*, 3 (1991), 223–242.

[18] Ungar, D., and Smith, R. B. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (1987), ACM Press, pp. 227–242.

[19] Varela, C. A., and Agha, G. A. What after java? from objects to actors. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7* (Amsterdam, The Netherlands, The Netherlands, 1998), Elsevier Science Publishers B. V., pp. 573–577.

[20] Wilkes, C., and Leblanc, R. Rationale for the design of aeolus: A systems programming language for an action/object system. In *Proceedings of the IEEE CS 1986 International Conference on Computer Languages* (New York, Oct. 1986), IEEE, pp. 107–122.

A Top-Down Approach to Writing Software for Networked Ubiquitous Systems

Urs Bischoff
Lancaster University, UK

Abstract — **First implementations of ubiquitous computing systems have shown promising results for a business environment. By integrating computing resources into real physical objects we can move the execution or monitoring of business processes closer to where the actual process happens. This can reduce cost and reaction time. Despite this promise there is a lack of standardised protocols. This makes it difficult for an application developer to implement business applications on top of a given network. We argue that a top-down approach to writing applications is useful. We propose a high-level language that can specify a ubiquitous network's global behaviour. A compiler can automatically generate device-level executables from this global specification.**

I. What are we talking about?

Writing software can be a challenging task. Especially when we are dealing with applications in a complex environment. The ubiquitous computing scenario provides us with such an environment. This position paper focuses on an essential part of a software design process for this environment; it is about implementing applications for an environment of ubiquitous computers.

We are interested in applications in a business environment. Nowadays IT-solutions in this environment are based around a powerful backend infrastructure that collects all necessary input data in order to execute a centralised process. The ubiquitous computing vision has changed this architecture. By embedding computers into the actual physical environment, into the physical products, we can push the computation into the network. The execution or monitoring of business processes can be done closer to where the actual process happens. This can reduce cost and reaction time.

By embedding computers and sensors into objects we can make these objects "smart"; accordingly we call them *smart objects*. Our vision is a world of these smart objects and other more powerful devices (e.g. PDAs) that can provide useful services to a business. We refer to such a network as a *smart object system*.

What do we need to make a deployment of such a network feasible? We need a network that can execute the required services. If an application developer wants to deploy services in the network, then they should not need the expert knowledge of the system designer.

II. What are the problems?

The network we are addressing is highly distributed. It mainly consists of embedded computers (i.e. smart objects). Other devices (e.g. PDAs, PCs or RFID-tags) can interact with this system; they can use it or provide services for the system. The resulting network is very heterogeneous.

Compared to traditional distributed systems (e.g. [1]) we face slightly different problems. Efficiency, for example, is defined in different terms. Throughput and latency do not play the major role in smart object systems. Because the wireless devices are normally battery powered, minimum energy consumption is an important design criterion. Dealing with this kind of problems requires a lot of low-level knowledge. This can be a challenging task for an application designer.

Because of the young nature of the field, there are not any well-established protocols for communication and collaboration between devices. It is difficult to define common protocols, because we are dealing with a variety of totally different hardware platforms. They range from passive RFID-tags to high-end PCs. It is hard to establish a "narrow waist", or a common layer, that all devices can implement. There is no "TCP/IP" for smart object systems.

In a smart object system the network as a whole is in the centre of concern. The user is interested in the results of a running service. Knowing the individual node(s) that execute(s) this service is of less interest. This is in contrast to a more traditional point of view that focuses on a single device in a distributed environment. The ubiquitous computing vision of a large number of "invisible" devices strengthens the network-centric or global view in contrast to the node-centric one.
Evaluation (at all stages) is difficult. We are dealing with new technology that allows for fundamentally new applications. Domain experts of the business environment may be unable to make use of this new potential. It is important to find a way to close the gap between domain knowledge and the way new technology can make use of this knowledge.

There is a lack of usable and stable programming abstractions or middleware solutions. Experts are needed to design the whole system (hardware and software) from scratch. It can be challenging for an application developer who is an expert in the actual business environment to implement or adapt applications.

III. The Top-Down Approach

Communication protocols and applications are traditionally organised in different layers [2]. This encourages the design of device-based software on top of established layers. This approach works well in a traditional network where we can find these well-established layers (e.g. TCP/IP). We described the problems of establishing common layers in a smart object system. Furthermore, we showed that the network as a whole as opposed to single networked devices is in the centre of concern. In contrast to this bottom-up approach we favour a top-down approach.

We propose an abstraction that allows an application developer to write network services in a high-level language. This definition of a service does not have to specify where and how this service is executed in the network; it only specifies what the network as whole has to do and what results the user expects.

Related projects identified that many business processes are described in form of rules [3]. Rules are very natural - they can be understood by both humans and computers. We therefore developed a rule-based language for smart object systems.

In the following we show a simple example. For illustration purposes we have simplified the language. In this scenario we want to implement a service that can detect when a storage room is too hot for products that need to be chilled.

```
SPACE(storage), TIME(SIMULTANEOUS) {
   STATE tooWarm :- product(X),
                    hasToBeChilled(X),
                    hot().
}
```

The network is in a state *tooWarm* if there is a product *X* that needs to be chilled and the storage room is hot. This rule has a spatial and a temporal constraint. The rule is only valid in a certain region and at a certain time. In our example, the rule is restricted to a region called *storage*. A rule consists of a goal (*tooWarm*) and several conditions (e.g. *product(X)*). The temporal constraint in our example specifies that all conditions have to be valid simultaneously in order to satisfy the whole rule.

An application developer might have to specify several rules. In the example above there is a condition called *hot()*. However, it does not say how warm a room has to be in order to be hot. A rule specifying that it is hot if the temperature is more than 25 degrees centigrade has to be defined:

```
SPACE(storage), TIME(SIMULTANEOUS) {
   hot() :-  temperature(X),
             X>25.
}
```

Similar rules are defined for *product(X)* and *hasToBeChilled(X)*.

Each device in the network exposes an interface; it specifies the device's capabilities and properties. A device that can measure the room temperature, for example, defines a capability *temperature(X)* in its interface. Other devices have different capabilities. In order to provide the whole service, they have to collaborate.

By using the temporal and spatial constraints we can define the global behaviour of a network. The declarative nature of this language allows us to separate the definition from the execution of a service. Another advantage of this declarative language is its implicit parallelism; rule conditions that could be executed in parallel can be easily extracted.

Analogously to a compiler for a single device application we can have a process that can translate the rule-based service description into a distributed application for the given network. The network is given by all the interfaces of the devices in the network.

The service description does not specify where and how a service has to be executed; it could be centralised or fully distributed. The choice is left to the translation process. This process has to decide which nodes require what knowledge and how they can collaborate in order to provide the specified service. The translation can be influenced by a translation policy; minimising overall energy consumption is one example of such a policy. This translation process is the biggest challenge of this top-down approach.

IV. Conclusion

In order to make ubiquitous networks more accessible to application developers we need programming abstractions. It is important to find a trade-off between a problem specific language and a very generic abstraction. We proposed a rule-based language. Rules are widely used to specify business processes; the translation of such a natural language rule into an application is straightforward. Furthermore, the domain expert is familiar with rules. By hiding the complexity of the underlying network the application developer can focus on the correctness of the service rules. Evaluation time can be reduced.

The proposed top-down approach is suitable for heterogeneous networks. This approach does not require all nodes to have the same common communication protocols. We still require them to be able to communicate with each other. However, we do not build applications on top of certain common layers.

By automating the translation process the application developer does not have to deal with communication, synchronisation or other low-level optimisation problems which make distributed application complex and error-prone.

References

[1] The Message Passing Interface (MPI) Standard. Available: http://www-unix.mcs.anl.gov/mpi/

[2] Open Systems Interconnection – Basic Reference Model: The Basic Model. ISO/IEC 7498-1, 1994.

[3] M. Strohbach, H.W. Gellersen, G. Kortuem and C. Kray. Assessing Real World Situations with Embedded Technology. In *Proceedings of the Sixth International Conference on Ubiquitous Computing (Ubicomp)*, 2004.

Context-Aware Error Recovery in Mobile Software Engineering

Nelio Cacho[1], Sand Correa[2], Alessandro Garcia[1], Renato Cerqueira[2], Thais Batista[3]

[1]Computing Department, Infolab21, Lancaster University – UK

[2]Computer Science Department, PUC-Rio – Brazil

[3]Computer Science Department, UFRN – Brazil

1. Motivation

The recent advances on mobile computing have enabled the development of a wide variety of pervasive mobile applications, which are able to monitor and exploit dynamically-changing contextual information from users and surrounding environments. However, with mobile software systems becoming applicable in many sectors, such as ambient intelligence, manufacturing, and healthcare, they have to cope with an increasing number of erroneous conditions and satisfy stringent dependability requirements on fault tolerance, integrity, and availability. Hence a deep concern to engineers of dependable mobile systems is how to properly deal and handle errors in the presence of contextual changes. Dependable systems traditionally detect errors caused by environmental, software, and hardware faults, and employ error-recovery techniques to restore normal computation.

Exception handling and *replication techniques* are complementary fundamental means to design and implement error recovery in software systems. Exception handling [2] is a forward-recovery technique that provides abstractions and mechanisms to respectively model and treat erroneous states, such as *exceptions*, *handlers*, *exception interfaces*, *control of the abnormal control flow*, *exception propagation*, and so forth. Replication is a backward-recovery technique based on the concept of replicated software component, that is, a component with representations in more than one host [5, 6]. In this position paper, we first discuss (Section 2) the impact of context-awareness on the characterization and handling of erroneous conditions in mobile applications. The idea is to understand why and to what extent the traditional abstractions and mechanisms from exception handling and software replication techniques need to be rethought and adapted to the engineering of dependable mobile applications. Based on this initial discussion, we draw some software engineering challenges for such applications (Section 3).

2. Is it an Error? How to Handle it? Well… It Depends on the Context

Contextual Errors. Based on our previous experience on the design of pervasive mobile applications [1, 3], we have found that the characterization of a plethora of erroneous conditions have a direct impact from contextual system changes. For example, a software component state may be considered an error in a given contextual configuration, but it may be not in others. In addition, several concurrently thrown events relative to contextual information in different devices can occur in a mobile application, which together means the occurrence of a more serious abnormal situation. For instance, fire occurrences affect many sensors which can throw different concurrently exceptions that individually represent one localized problem but should be handler as a more severe, general fault by all the mobile devices located at the impacted physical region.

Implications of Contextual Changes on Error Recovery. The importance of exploiting contextual changes in mobile systems seems not only restricted to the characterization of errors themselves, but also in the way they need to be handled by the error recovery techniques. First, error propagation needs to be context aware since it needs to take into consideration the dynamic system boundaries and the frequent variation of device contexts such as fluctuating network bandwidth, temperature changes, decreasing battery power, changes in location or device capabilities, degree of proximity to other users, and so forth. Second, contextual information also may impact on the selection of error handlers and the determination on which and how many software replicas need to be executed.

Asynchrony and Openness Issues. With physical mobility, network failures are one of the greatest concerns once wireless communication is susceptible to frequent disconnections. To address this issue, asynchronous communication mechanisms become the main pattern of device interactions. However, traditional mechanisms for exception handling and replica management in distributed systems, such as transactions and checkpointing techniques, rely often on the utilization of synchronous communication protocols. Additionally, the openness characteristic of several context-aware mobile systems complicates this scenario. Such characteristic leads to increased unpredictability of erroneous conditions, and involves a diversity of devices with different capabilities which in some situations are not aware about certain errors in dynamically-changing environments.

3. Engineering Dependable Context-Aware Systems: Research Challenges

This section discusses how the influence of context-awareness on error detection and handling brings new software engineering challenges to designers of exception handling (Section 3.1) and replication techniques (Section 3.2).

3.1. Context-Aware Exception Handling

Context-awareness seems to indicate that the handling of exceptional situations in mobile applications is more challenging, which in turn makes it impossible the direct application of conventional exception handling abstractions and mechanisms used in sequential and distributed systems [4]. First, since error propagation needs to be context-aware (Section 2), new software engineering abstractions are necessary to allow designers to define proper exception handling scopes in the structuring of their mobile applications. For example, exceptions may need to be dynamically propagated to different groups of devices in a given physical context so that they are collectively handled by them. Second, execution of exception handlers, selection of exception handling policies, and exceptional control flows often need to be determined according to user or environmental contexts. Third, as the characterization of an error itself may depend on the context, there is a need to define proper abstractions to support the definition of such contextual exceptions. In addition, several concurrently thrown exceptions can occur in a context-aware application, which actually mean the occurrence of a more serious abnormal situation (Section 2). Thus, a monitoring system should be able to collect all those concurrent exception occurrences and resolve them so that the proper exception is raised.

The openness feature creates an unawareness of the devices in terms of the existing exceptions, in which software was developed by different designers, would not be able to foresee all the exceptions provided by the current context. As a result, there is a need to explore the mobile collaboration infrastructure when an exceptional context is detected by one of the peers. Thus, severe exceptions should be notified to other mobile devices even when they have not registered interest in that specific exception. In other words, the exception should be proactively raised in other mobile collaborative devices pertaining to the same contextual boundaries. In order to handle the unforeseen exceptions, the receiver should start a collaborative activity to search for an adequate handler for this received exception and handle it according to the contextual conditions.

3.2. Context-Awareness and Replication Techniques

Traditionally, replication techniques have been applied explicitly and statically at design time. As a consequence, the designer of the application is responsible for identifying which services are critical to the systems and making decisions about what should be made robust and which strategies should be taken. New pervasive mobile applications, however, are much more dynamic, leading to scenarios where it is difficult to identify in advance the critical services and their software components. Moreover, the concept of criticality itself changes over time as context changes. As a result, a software component that is critical at a moment can lose this property a moment later. Since replication has a high cost, it is important to update de number of replicas in the system as the criticality of the software components and services change. Another important issue that may vary over time, as context changes, refers to the strategy applied to the replication. Replication techniques can be active, in which all replicas process all messages, or passive, in which only one replica processes all input and periodically transmits its current state. Each replication strategy has its advantages and disadvantages and the suitability of one strategy to the other is context dependent. As a result, to cope with scenarios where context environment changes constantly, we need to explore new approaches in which the system itself can dynamically identify the most critical components in a given context and dynamically performers the needed adaptation on the number of copies and the replication policy. Another key issue is how to provide the application developers with high-level programming abstractions to specify the relationship between the context information and fault-tolerance policies.

References

[1] Cacho, N. et al. Handling Exceptional Conditions in Mobile Collaborative Applications: An Exploratory Case Study. Proceedings of the 4th Workshop on Distributed Mobile Collaboration, WETICE.06, Manchester, UK, June 2006.

[2] Goodenough, J. B. Exception handling: issues and a proposed notation. Communications of the ACM 18, 12 (Dec. 1975), 683-696.

[3] Damasceno, K. et al. Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case. Proceedings of the 5rd SELMAS Workshop@ICSE, May 2006.

[4] Xu, J. et al. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In Proc. 25th FTCS, 1995.

[5] Anderson, T., Lee, P. Fault Tolerance: Principles and Practice. Prentice-Hall, 2nd edition, 1990.

[6] Guessoum, Z., Briot, J., Charpentier, S. Dynamic and Adaptative Replication for Large-Scale Reliable Multi-Agent Systems. Proceedings of the 1st SELMAS Workshop@ICSE. May 2002.

Towards Effective Exception Handling Engineering in Ubiquitous Mobile Software Systems

Nelio Cacho[1], Alessandro Garcia[1], Alexander Romanovsky[2], Alexei Iliasov[2]

[1]Computing Department, Infolab21, Lancaster University – UK

[2]Computing Science School, University of Newcastle upon Tyne - UK

1. Motivation

Resilience through Exception Handling. Exception handling techniques [3,5] provide the fundamental abstractions and mechanisms for constructing resilient software systems. They ensure system modularity in the presence of faults by offering software engineering abstractions for (i) representing erroneous states of software systems and their internal modules as *exceptions*, (ii) introducing *scopes* for handling exception occurrences, (iii) encapsulating exception handling activities into *handlers*, and (iv) explicitly specifying *exceptional interfaces* of modules. In addition, exception handling mechanisms promote more reliable programming by hiding from programmers a plethora of complexities relative to error handling. First, they provide explicit support for both systemic and application-specific exception detection. Second, they implement disciplined deviation from the normal control flow to the exceptional one, thereby automatically searching for and activating suitable handlers upon the occurrence of exceptional states. Once handlers are successfully executed, the system is seamlessly returned to its normal operation. Third, they tend to support simple basic solutions by being tightly coupled to the abstractions and mechanisms of the underlying programming paradigm.

Evolution of Exception Handling Mechanisms. Since the publication of seminal papers on exception handling [3,5], this area has received considerable attention from researchers across different communities, such as Software Engineering, Programming Languages, Dependability and Distributed Systems. Exception handling techniques for sequential programs were been initially designed to support the abstractions and mechanisms mentioned above. Their designs have also been systematically enhanced in order to promote their integration with mainstream development paradigms, such as object-orientation [6], and characteristics of modern applications, such as concurrency and distribution [7]. For instance, exception mechanisms in distributed concurrent systems rely on advanced transaction mechanisms in order to both cope with erroneous conditions in multi-thread collaborations and support proper modular units for error confinement, handler attachments, exception resolution, and exception handling scopes.

Why is Exception Handling in Ubiquitous Computing Challenging? There are several middleware systems available nowadays for the development of ubiquitous mobile applications. Their underlying architectures rely on different coordination models, such as tuplespaces, pub-sub mechanisms, and computational reflection. However, such middleware systems rarely provide appropriate support for exception handling. Treatment of exceptional conditions in mobile applications offers a number of challenges due their stringent requirements of openness, asynchrony, context-awareness, and increased unpredictability [1,2,4]. The dynamically changing collaboration contexts need a more flexible exception handling approach. Devices working collaboratively may fail to perform specific actions, which may in turn affect other participants in the collaboration. This is why exception handling cannot rely on traditional techniques such as transactions or sequential exception handling. In addition, exception interfaces, exception resolution, search for handlers, and error propagation policies need to adapt according to the frequent changes in the environmental and collaboration contexts. The existing solutions are too general and not specific to the characteristics of the coordination techniques used. Typically they are not scaleable because they do not support clear system structuring using modular units corresponding well to the exception handling scopes in such dynamic environments.

2. Research Challenges

Based on the motivation described above, in the last two years we have performed a number of exploratory studies [1,2,4] and derived a number of research questions which are guiding our ongoing and future work. Some of the problems and potential directions for solutions are sketched in the following.

A. Ensuring resilience of ubiquitous software through exception handling engineering. This work will rely on introducing a set of exception handling abstractions supporting development of resilient ubiquitous applications. These abstractions will fit the specific characteristics of the ubiquitous systems and typical applications and scenarios: system openness and context awareness, component mobility and autonomy, and a large variety of errors which need to be tolerated. This work will enrich the current engineering methods with the basic abstractions, such as exceptions, exception handlers, exception propagation, and nested scopes, to be used by developers to design well-structured resilient ubiquitous applications which interlink in an intuitive and effective ways separated descriptions of the normal and abnormal system and component behaviour and states. As an example we will look into understanding of how errors need to be handled in the mobile collaborative applications. To the best of our knowledge, there is no systematic study that investigates: (i) the intricacies of collaborative error handling in the presence of physical mobility, and (ii) how mainstream coordination techniques, such as pub-sub and tuple-spaced mechanisms, are appropriate to implementing robust, mobile collaborative applications.

B. Development of a middleware supporting these abstractions in the typical paradigms used for developing ubiquitous systems. Developers of the traditional ubiquitous middleware do not pay enough attention to developing specialised and effective exception handling facilities. The typical assumption is that to handle exceptions at the application level the developers will be using the standard middleware services. This is clearly a mistake. The functionally related to exception handling are very specific and implementing them using general services not tailored to these specific characteristics typically causes more errors and makes system even less resilient. These services need to include exception propagation to all parties involved in handling, managing nested scope structures, concurrent exception resolution, changing the mode of exception from the normal one to the abnormal and back after successful recovery, detecting and raising a common set of predefined exceptions typical for ubiquitous systems.

C. Development of the earlier architectural models explicitly incorporating the exception handling abstractions and ensuring seamless model transformation. It is well understood nowadays that exception handling is typically a global design issue, rather than a particularly system property to be dealt only at the implementation stage. To ensure resilience of the ubiquitous systems they need to be architected from the earlier steps of their development by incorporating exception handling features into their component, connectors, and configurations. This needs development of the specialized architectural solutions (patterns, styles and sets of standardized dedicated architectural elements) and modelling techniques (architectural views, model specification, and model transformations). The existing fault tolerance architectures are not directly applicable in the context of ubiquitous computing as they typically support hardwired static solutions and do not allow context awareness of fault tolerance.

D. Aspect-oriented solutions for integrating exception handing into ubiquitous applications. One of the general motivations for employing exception handling in the development of resilient applications is to improve software modularity in the presence of erroneous system states [5]. The code devoted to exception detection and handling should not be intermingled with the implementation of normal system activities. The situation is potentially more complicated in mobile ubiquitous systems where an exceptional condition may be a result of concurrent application-specific events in code embedded in multiple mobile devices; as a consequence, the code needed to check invariants is typically scattered all over the mobile programs. We have performed an empirical study [8] on the suitability of aspect-oriented programming (AOP) to promote improved modularization of exception handling in distributed systems. Although AOP techniques have mostly worked well to lexically separate error handling code from normal code, traditional pointcut languages seem not scale up in distributed, ubiquitous environments. For example, we think that a state-based joinpoint model and a corresponding specialized pointcut language (such as, the one defined in [9]) for exception detection can improve the modularization of error handling in ubiquitous mobile applications. They can potentially provide enhanced mechanisms to quantify exceptional context-specific conditions over sets of hosts and mobile devices.

References

[1] Cacho, N. et al. Handling Exceptional Conditions in Mobile Collaborative Applications: An Exploratory Case Study. 4th DMC Workshop, 2006.

[2] Iliasov, A., Romanovsky, A. CAMA: Structured Communication Space and Exception Propagation Mechanism for Mobile Agents. ECOOP-EHWS 2005, 19 July 2005, Glasgow.

[3] Goodenough, J. B. Exception handling: issues and a proposed notation. Commun. ACM 18, 12 (Dec. 1975), 683-696.

[4] Damasceno, K. et al. Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case. 5rd SELMAS Workshop@ICSE, May 2006.

[5] Parnas, D., Würges, H. Response to Undesired Events in Software Systems. Proc. of the 2nd International Conference on Software Engineering. San Francisco, California, USA, pp. 437 - 446, 1976. IEEE-CS.

[6] Garcia, A., Rubira, C., Romanovsky, A., Xu, J. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. Journal of Systems and Software. 59(2001), 197-222.

[7] Xu, J. et al. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In Proc. 25th FTCS, 1995.

[8] Filho, F. et al. Aspects and Exceptions: The Devil is the Details. Submitted to Foundations on Software Engineering (FSE.06), November 2006.

[9] Navarro, L. et al. Explicitly Distributed AOP using AWED. Proceedings of AOSD.06, Bonn, Germany, March 2006.

SEUC 2006 Workshop Programme

Session 3: Formal Methods

Towards Rigorous Engineering of Resilient Ubiquitous Systems

Alexander Romanovsky
University of Newcastle upon Tyne
UK
alexander.romanovsky@ncl.ac.uk

Kaisa Sere
Aabo Akademi University
Finland
kaisa.sere@abo.fi

Elena Troubitsyna
Aabo Akademi University
Finland
etroubit@abo.fi

1. Introduction

Ubiquitous systems should smoothly weave themselves into our everyday life. In general, ubiquitous systems are complex computing environments integrating a large number of heterogeneous devices. Such computing environments can be considered as complex decentralized distributed systems composed of loosely-coupled components asynchronously communicating while providing system services. Since the system components are typically autonomous and mobile, the systems as such are open and dynamically reconfigurable. To ensure reliable provision of system services in presence of component mobility, dynamically changing configuration and inherent unreliability of communicating channels, we should aim at designing resilient systems, i.e., the systems which are able to withstand unpredictable changes and faults in their computing environments.

2. Resilient Ubiquitous Systems

Ensuring predictable and correct behaviour of ubiquitous systems is an open issue as the developers of these systems are facing overwhelming complexity coming from various sources. Generally speaking, this complexity is caused by a large size, openness and distributed nature of such systems. Let us observe that while complexity is currently perceived as a major threat to *dependability* of computing systems, our society puts a high level of *reliance* on ubiquitous systems, trusting them to perform a wide range of everyday critical functions. Indeed, these systems are rapidly becoming business- and safety-critical.

Traditionally managing system complexity is achieved via abstract modelling, decomposition and iterative development. In our work we focus on creating models and development techniques for designing resilient ubiquitous systems. For instance, we specifically focus on explicit modelling of mechanisms for coping with system impairments (faults, errors and failures), since a resilient system needs to be constructed and shown to be predictably tolerant to the faults, which are most harmful and most likely to affect the overall system service.

Fault tolerance is an acute issue in designing resilient ubiquitous systems. First of all, this is due to the fact that complexity and openness of these systems make it *impossible* to avoid or remove faults altogether. Moreover, fault tolerance is needed because of a high heterogeneity of components, modes of operation and requirements, causing *mismatches*. Furthermore, the growing involvement of the *non-professional* users requires improved fault tolerance.

Complexity of ubiquitous systems makes it extremely difficult for the developers to design appropriate and effective fault tolerance measures. Such systems are susceptible to tangled erroneous conditions caused by combinations of errors, their complex interference, as well as errors caused by malfunctioning infrastructures and misbehaving hosts, agents and people.

The worst thing the developers should be doing is including fault tolerance measures that make systems more complex and more error-prone.

3. Challenges in Engineering Ubiquitous Systems

Widely used ad-hoc techniques for engineering ubiquitous systems provide cheap and quick solutions for constructing various “lightweight” applications, but cannot ensure the appropriate level of system resilience when used in designing complex business- or safety-critical applications. To guarantee a high degree of resilience of such applications, we need to employ *rigorous methods* *for system engineering* – the methods, which support a disciplined development process based on formal modelling and reasoning about system correctness. This is why one of the main challenges in engineering complex ubiquitous systems is the creation of advanced models and methodologies for *resilience-explicit* rigorous design. This will include development of the generic design and modelling patterns and tools supporting the rigorous methods.

The methods and models defined will need to support a *systemic approach* to development. While adopting such an approach we will model the system together with the relevant part of its environment and decompose it into the software- and hardware-implemented parts later in the development process. This approach should be especially suitable for resilience-explicit development of ubiquitous systems since it will allow developers to clearly express the essential properties of the system without imposing constraining design decisions early in the design process.

In particular, we have identified a number of specific issues which resolution will contribute to the creation of a viable methodology for rigorous engineering of resilient ubiquitous systems:

– ensuring interoperability of independently developed components in formal construction of ubiquitous systems

– design and formal modelling of advanced mechanisms for tolerating faults typical for ubiquitous systems

– development of application-level fault tolerance techniques, such as flexible open exception handling mechanisms suitable for the open asynchronous and anonymous systems

– formal modelling and development of resilient context-aware ubiquitous systems.

4. Research on Ubiquitous Systems in RODIN Project

Currently we are involved in a FP6 STREP-project on Rigorous Open Development Environment for Complex Systems - RODIN http://rodin.cs.ncl.ac.uk. Among the methodological and technological issues addressed by RODIN we are actively working on creating a methodology for rigorous development of fault tolerant agent systems. As a case study we develop an implementation of Ambient Campus – a multi-agent application facilitating studies of students via mobile hand-held devices. Our implementation will be obtained through the rigorous modelling and development.

To attain this goal we have identified an initial set of the abstractions to be used for developing resilient ubiquitous applications. They support locations, scopes, agents, roles, exception propagation, etc. Then we have proposed a collection of development patterns supporting rigorous development of fault tolerant multi-agent systems in the B Method. In particular we have demonstrated how formal development allows us to ensure interoperability of independently-developed agents. Moreover, we have created an approach to rigorous stepwise development of the distributed fault tolerant middleware for multi-agent systems.

In parallel with our work on top-down approaches to the development of ubiquitous systems, we put forward bottom-up, model-checking approaches. We have identified typical behavioural patterns of ubiquitous systems and propose model-checking techniques optimized for these patterns.

Our ongoing work contributes not only to the methodology of developing resilient ubiquitous systems but also to the creating the RODIN tool platform. By exercising the platform in modelling multi-agent systems we enhance its capability in dealing with novel complex computational phenomena, such as resilient ubiquitous systems.

Controlling Feature Interactions In Ubiquitous Computing Environments

Tope Omitola
University of Cambridge, Computer Laboratory
15 JJ Thomson Avenue Cambridge CB3 0FD UK
too20@cam.ac.uk

1 Abstract

Ubiquitous computing depicts a world where several electronic objects are embedded everywhere in the environment and made available to people using a wide variety of user interfaces. These objects need to co-operate and interact to help users achieve their goals. These interactions will lead to a high proportion of **feature interactions**. A **feature interaction** is some way in which a feature modifies another feature in the overall system's behaviour set, thus affecting overall system's reliability and dependability, and therefore users' trust in the system. This paper describes a solution to the **Feature Interaction** problem. We take advantage of the ontological descriptions of these electronic objects, and provide a language of interaction used to specify objects' intended behaviour; and we use a deductive system which detects possible feature interactions before they occur, and takes ameliorating actions.

2 Introduction

Some of the properties of the elements of a ubiquitous computing environment are:

- the devices will be of limited storage, processing, display and battery power capabilities
- the computing environment will be open [1], i.e. there will be a continual entry, exit, and reentry of devices and applications from disparate sources
- the resources will vary in their availability, their quality, and their reliability
- the devices and resources will be increasingly aware of their context, including their location, and their relationships with the wider world (both real and virtual)
- applications will be built by dynamically composing disparate sub-systems together
- as the devices, resources, and applications have variable characteristics, they will need to interact to provide overall system behaviour, and therefore some of them will need to have shared access to resources
- creation of new applications and codes in the form of rules
- and, there will be a very high potential of feature interactions and unexplored consequences.

Although each component may work very well on its own, composing them together will lead to a high proportion of (unwanted) **feature interactions**[1]. A **feature** is any part of a specification having a self-contained functional role, and a **feature interaction** is some way in which a feature or features modify or influence another feature in the overall system's behaviour set. This influence takes many forms depending on the nature of their composition. Features might interact by causing their composition to

[1]More information on the feature interaction problem and its effects can be found at [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

be incomplete, inconsistent, nondeterministic, or unimplementable. Or, the presence of a feature might simply change the meaning of another feature with which it interacts. Effective solutions to a system's feature interaction problems should improve that system's dependability and reliability.

3 Towards Solving the Feature Interaction Problem

It was argued in [13] that the interactions of electronic objects of the physical world of PDAs, sensors, RFIDs, and other embedded systems of an open Ubiquitous Computing environment will be based on **policies** and **rules**. A **policy** may express such diverse characteristics as transactionality, security, response time, pricing, etc. For example, a policy of a service may specify that all interactions be invoked under transaction protection, that incoming messages have to be encrypted, that outgoing messages will be signed, that responses may only be accepted within thirty seconds, and that certain operations are subject to a fee to be paid by credit card by the invoker. Examples: XDDML [14], a system independent device description language, is used to describe network components in a network/bus independent manner, and to describe devices' properties and behaviour of the applications they host. In consumer electronics, CC/PP [15] is used to describe devices' capabilities and behaviour. A **rule** expresses the relation of domains in that environment, e.g., a user may express that the sensors in the home should be activated at some particular times, or they may express rules concerning how they want service composition to be done. Such rules may be expressed textually, or by speech, or by a combination of both.

Since these open computing ubiquitous environments will be governed by policies and rules, we argue for a rule-based control system to manage and control feature interactions, and thereby to improve the reliability and trustworthiness of these environments.

4 Properties of the Rule Based Control (RBC) Architecture

(1). A non-monotonic deductive system that elicits possible feature interactions;
(2). Policies of incoming electronic objects in ontological form (using, for example, RDF [16]);
(3). A Rule language to specify intended behaviour of electronic objects interactions

5 Properties of rule specification language

(A). Be **declarative**, allowing users to express behaviour in terms of **what** to be done, and not in terms of **how** to do it
(B). Gives its users good primitives to express time and duration
(C). Good primitives to express uncertain information.
We have developed a language for such a purpose, called Rule (Language for) Feature Interaction $\mathbf{R^{FI}}$.

5.1 Informal Semantics of $\mathbf{R^{FI}}$ language

$\mathbf{R^{FI}}$ is a sorted language based on topological [17] and metric [18] logic. Its grammar and (informal) semantics are given below:
A **Rule** is of the form:

$$\begin{aligned} R^{FI} ::= {} & (E_{ID}, (E_{TEMP\_PRE} \bullet Event_{PRE}) \\ & \rightarrow (E_{TEMP\_POST} \bullet Event_{POST})\,) \end{aligned} \quad (1)$$

E_{ID} is the Identifiers expression,
$E_{TEMP\_PRE}, E_{TEMP\_POST}$ are temporal expressions,
$Event_{PRE}$ is the Events' Pre-condition expression, and $Event_{POST}$ is the Events' Post-condition expression.

5.1.1 E_{ID} Syntax and Semantics

$E_{ID} ::= (R_{ID}, R_{PRIO}, U_{ID}, U_{PRIO})$
$\mathbf{R_{ID}}$ is the rule identifier.

$\mathbf{R_{PRIO}}$ is the rule priority. Purpose: Defines the priority for the rule.
$\mathbf{U_{ID}}$ is the user identifier. Purpose: Specifies the identifier of rule creator.
$\mathbf{U_{PRIO}}$ is the user priority. Purpose: Specifies priority of rule creator.

5.1.2 $E_{TEMP\_\{PRE,POST\}}$ Syntax and Semantics

Alphabets, Terms, and Formulae

1. the temporal sort, $\mathcal{S_T}$
2. the granularity sort, $\mathcal{S_G}$, denoting the set of granularity temporal domains, such as years, months, weeks, days, minutes, seconds
3. the temporal position operator $\star_{(\phi)}$
4. the granularity, $\nabla^{granularity}$, and displacement operator, $\nabla_{displacement}$
5. the projection operator, $\Box$, and its dual, $\Diamond$
6. a derived operator, $\nabla^{granularity}_{displacement}$, where $\nabla^{granularity}_{displacement} = \nabla^{granularity}\,\nabla_{displacement}$
7. a binary predicate,$\sqsupseteq$, over $\mathcal{S_G}$, where $G_i \sqsupseteq G_y$ means granularity G_i takes precedence over granularity G_y with $years \sqsupseteq months \sqsupseteq weeks \sqsupseteq days \sqsupseteq mins \sqsupseteq secs$
8. the temporal position operator, $\star_{(\phi)}\mathcal{F}$, which evaluates $\mathcal{F}$ from temporal position $\star_{(\phi)}$
9. the projection operator $\Box\mathcal{F}$ evaluates formula $\mathcal{F}$ and $\mathcal{F}$ is true if $\mathcal{F}$ is true at all related instants, and its dual
10. projection dual operator $\Diamond\mathcal{F}$, where $\Diamond\mathcal{F} = \neg\Box\,\neg\mathcal{F}$. It evaluates to true if $\mathcal{F}$ is true in at least one related instant
11. the temporal displacement operator, $\nabla_\alpha\mathcal{F}$. Here, $\mathcal{F}$ is true at the time instant at distance α from the one specified by $\star_{(\phi)}$, where $\alpha \in Z^+$
12. the granularity operator, $\nabla^{granularity}\mathcal{F}$. $\mathcal{F}$ is true with respect to granularity G from $\mathcal{S_G}$ ($\{years, months, weeks, days, mins, secs\}$), and $years \sqsupseteq months \sqsupseteq weeks \sqsupseteq days \sqsupseteq mins \sqsupseteq secs$
13. the derived temporal granularity and displacement operator, ∇^G_α. $\mathcal{F}$ is true with respect to G and at the time instant at distance α from the one specified by $\star_{(\phi)}$, where $\alpha \in Z^+$

5.1.3 $Event_{PRE}$ and $Event_{POST}$ Syntax and Semantics

Alphabets, Terms, and Formulae

1. A set $\mathcal{R}$ of Resource (device and object) names, e.g. switch1, light1, etc.
2. A set $\mathcal{A}$ of action names, e.g. "on", "off", "up", "down", etc.
3. A serial conjunction operator, $\otimes$, which composes events together sequentially
4. A serial disjunction operator, $\oplus$, which composes events together but acts like a choice operator
5. A parallel operator $||$ which joins events together and performs them in parallel
6. A set $\mathcal{L}$ of logical connectives: $\mathcal{L} = \{\ \wedge,\ \vee\ \}$, used to connect composite events
7. The unary operator, $\neg$
8. A pair of port commands $\mathcal{P} = \{\ ?\ ,\ !\ \}$, where ? is an input port command, and ! is an output port command
9. The quantification symbols $\mathcal{Q} = \{\ \exists, \forall\ \}$ on $\mathcal{R}$

5.1.4 Some Examples of Rules

$$\begin{aligned}&(rule1, 3, too20, 5),\\&(\star_{[2006-09-28T11:00]}\,\Box\,(door!open))\\&\rightarrow\ (\star_{[2006-09-28T11:00]}\,\Box\,\nabla^{mins}_{15}\,(door?close))\end{aligned} \qquad (2)$$

If the door is open at 11:00, keep it open for 15 minutes, and close it after 15 minutes.

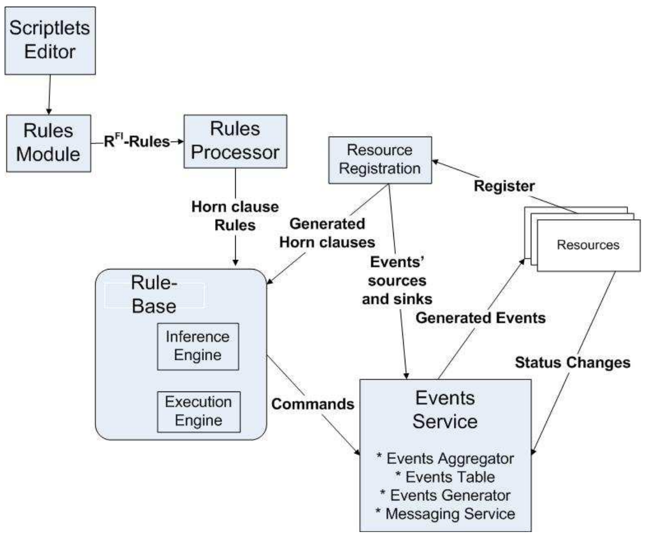


Figure 1: RBC System Architecture

$$
\begin{aligned}
&(rule2, 2, too20, 5),\\
&(\star_{[2006-09-28T11:00]} \Box \quad switch1!on)\\
&\rightarrow \; (\star_{[2006-09-28T11:00]} \quad \Box\, (\nabla_{60}^{mins}\, switch1?off)\\
&\qquad || \; (light1?on \otimes \nabla_{60}^{mins}\, light1?off)\,)
\end{aligned} \tag{3}
$$

This means, if switch1 is on at 11:00 on 28 Sept 2006 until 12:00 on 28 Sept 2006, keep light1 on during the length of that time.

6 System Mechanism of the Rule Based Controller

Fig. 1 shows the high level system architecture of the RBC. It consists of:

1. Scriptlets Editor: This allows users to create small scripts (rules) expressing intended objects' interactions. Creation of scriptlets as forms of interactions is compatible with socio-anthropological studies made on how people interact in everyday practice. For example, de-Certeau [19] showed that everyday practices depend on a vast ensemble which is difficult to delimit but which we may provisionally designate as an ensemble of procedures. The latter are schemas of operations and of technical manipulations. Procedures lack the repetitive fixity of rites, customs, or reflexes. Their mobility constantly adjusts them to a diversity of objectives. Scriptlets can be viewed as the goals and intentions of the user. Intentions are conduct controllers and good co-ordinators of actions [20]. For example, an intention to bring about a particular effect will normally give rise to one's endeavouring to bring about that effect. To endeavour to bring about some effect is, partly, to guide one's conduct and actions accordingly, thereby controlling one's conduct and co-ordinating one's actions. Scriptlets can be gestures, speech, texts, or combinations of these

2. Rules Module: contains scriptlets-converted $\mathbf{R^{FI}}$- form rules

3. Resources, represented in the system as **sensors** and **actuators**. Sensors monitor the state of the external world, i.e. the devices in the environment. They provide inputs to the system. The generated actions are used by actuators to influence and/or control the external world

4. Registration Resource Module: **Registers** resources; **Generates** Horn clauses of events' sinks and sources in resources; **Notifies** the Event Service of events' sinks and sources

5. Rules Processor: Converts rules into Horn clauses inserting them into the rule base

6. Rule Base, which consists of

   - Rules in Horn clause form
   - A Logic Inference Engine: Runs inferencing and reasoning algorithms on rules. The outputs of these reasoning algorithms are rules which have the following characteristics:
     - feature interaction-free
     - conflict-free

– satisfy safety and liveness properties

(section 6.1 has more on these)

Users are notified of satisfiability of submitted rules.

- The Execution Engine. Activates feature interaction-free, conflict-free rules turning them into commands that are sent to the Event Service

We can notice from our rule-based system that the stored knowledge (i.e. the inference engine) is orthogonal from the control mechanism (i.e. the execution engine). This separation is the proclaimed, if rarely achieved, goal of rule-based systems.

7. Events Service: Is **notified** of events originating from resources; **Receives** commands from the Execution Engine and generates events for resources; **Provides** the asynchronous messaging capability of the system.

6.1 Types of Reasoning performed on Rules

- Conflict Detection and Resolution algorithms which solve the following problems: a rule is an event structure: $E.S. =< \tau, I, R, A \cup V_i >$
τ = Priority Assignment, I = time interval (in milliseconds), R = resource,
A = action, V_i = action attributes.
A **feature interaction problem**, is defined as: For $e.s._1 \in E.S._1 \wedge e.s._2 \in E.S._2$.
$e.s._1 = [\tau_1, i_1, r_X, a_1] \wedge e.s._2 = [\tau_2, i_2, r_X, a_2]$ and a truth-value binary relation R on $i_1 \times i_2 \wedge R \in \{during, equal, overlaps, starts, ends\}$, there is a **feature interaction** if $i_1 R i_2 \wedge a_1 \neq a_2$ given the same resource r_X, and
A **conflict discovery problem** is a structure $< RuleSet_1 \times RuleSet_2, \phi >$ where ϕ is a truth-value binary relation on $RuleSet_1 \times RuleSet_2$.

- Safety reasoning on rules: A property $\mathcal{P}$ is a safety property if for all histories $\mathcal{H} = (H_0, \ldots, H_n, H_{n+1}, \ldots)$, over the RuleBase $\mathcal{B}$ and the domain of rules $\mathcal{R}$, the following holds: if $\mathcal{H}$ does not belong to $\mathcal{P}$, then some prefix $(H_0, H_1, \ldots, H_t)$ of $\mathcal{H}$ cannot be extended to any history in $\mathcal{P}$, and

- Liveness reasoning on rules: A property $\mathcal{P}$ is a liveness property if any finite history $\mathcal{H} = (H_0, H_1, H_2, \ldots, H_t)$ can be extended to an element of $\mathcal{P}$.

6.2 Overview of System Operation

- **Resources**: Resource (with its RDF) registers; Horn clauses generated for events in the RDF; Event Service is notified of events' sources and sinks in RDF.

- **Rules**: User creates scriptlets; Converted to $\mathbf{R}^{\mathbf{FI}}$ rules and later to Horn clauses; Reasonings, as enumerated in 6.1, performed on rules; Feature interaction-free, conflict-free rules inserted into Rule Base; Rules triggered by (a) temporal elements of some rules, or (b) by events coming from the environment through the Event Service.

7 Conclusion

Adequate solution to the feature interaction problem is needed to achieve the vision of seamless cooperation and interaction of electronic objects of ubiquitous computing systems. We built a rule-based control solution where a non-monotonic deductive system used electronic objects' policies and users' rules combined with powerful reasoning algorithms to elicit and handle unwanted feature interactions.

References

[1] C. Hewitt and P. de Jong: *Open Systems.* Technical Report, AIM 691, A.I. Laboratory, M.I.T. Dec. 1982

[2] L.G. Bourma, H. Velthuijsen: *Feature Interactions in Telecommunications Systems.* pub. IOS Press (Amsterdam), 1993

[3] K.E. Cheng, T. Ohta: *Feature Interactions in Telecommunications Systems III.* pub. IOS Press (Amsterdam), 1995

[4] P. Dini, R. Boutaba, L. Logrippo: *Feature Interactions in Telecommunications Systems IV.* pub. IOS Press (Amsterdam), 1997

[5] K. Kimbler, L.G. Bouma: *Feature Interactions in Telecommunications Systems V.* pub. IOS Press (Amsterdam), 1998

[6] M. Calder, E. Magill: *Feature Interactions in Telecommunications Systems VI.* pub. IOS Press (Amsterdam), 2000

[7] D. Amyot: *Feature Interactions in Telecommunications Systems VII.* pub. IOS Press (Amsterdam), 2002

[8] ISTAG Scenarios for Ambient Intelligence in 2010 Final Report Feb 2001: http://www.cordis.lu/ist/istag.htm(last viewed Aug. 2005)

[9] Manfred Broy: *Automotive Software Engineering.* Proceedings of the 25th International Conference on Software Engineering (ICSE), pp. 719-720, 2003

[10] NASA: *Concept Definition for Distributed Air/Ground Traffic Management (DAG-TM), Version 1.0.* Advanced Air Transportation Technologies Project. NASA Ames Research Center. NASA Langley Research Center, 1999

[11] J. Hoekstra, R. Ruigrok, R. van Gent, J. Visser: *Overview of NLR Free Flight Project 1997-1999.* NLR-CR-2000-227, National Aerospace Laboratory (NLR), May 2000

[12] J. Maddalon, R. Butler, A. Geser, and C Muńoz: *Formal Verification of a Conflict Resolution and Recovery Algorithm.* NASA/TP-2004-213015. NASA Langley Research Center, April 2004

[13] T. Omitola: *Building Trustworthiness Into the Semantic Grid.* Workshop on Ubiquitous Computing and e-Research, Edinburgh, UK 2005

[14] IDA - Interface for Distributed Automation: Architecture Description and Specification, IDA Group, Nov. 2001

[15] CC/PP: http://www.w3.org/Mobile/CCPP/

[16] Resource Description Framework: http://www.w3.org/RDF/

[17] N. Rescher, J. Garson: *Topological logic.* Journal of Symbolic Logic, 33, pp. 537-548, 1968

[18] R. Koymans: Specifying Message Passing and Time-Critical Systems with Temporal Logic, in *Lecture Notes in Computer Science, vol. 651, 1992*

[19] M. de Certeau: *The Practice of Everyday Life.* University of California Press, 1988

[20] P. R. Cohen, J. Morgan, M.E. Pollack: *Intentions in Communications.* MIT Press, 1990

Dependability Challenge in Ubiquitous Computing

Kaisa Sere, Lu Yan, Mats Neovius
Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland
{Kaisa.Sere, Lu.Yan, Mats.Neovius}@abo.fi

1. Background

With more than 2 billions terminals in commercial operation world-wide, wireless and mobile technologies have enabled a first wave of pervasive communication systems and applications. Still, this is only the beginning, as wireless technologies such as RFID are currently contemplated with a deployment potential of ten's of billions of tags and a virtually unlimited application potential.

Although significant R&D work has been undertaken over recent years on these systems, most of the research is still very application specific, with security and environmental applications dominating and demonstration driven. However, it is likely that more generic and comprehensive approach is required, where different stakeholders and research specialists work together to solve true systems level problems.

2. Research problem

During the past years, various projects were launched around the world on ubiquitous computing, including Georgia Tech's Aware Home, Inria's Smart Office, Stanford's iRoom, Cisco's Internet Home, Essex's Intelligent Inhabited Environments, HP's Cool Town, ATR's Creative Space, CMU's Aura, Xerox's Smart Media Spaces, IBM's DreamSpace, KTH's comHOME, Microsoft's EasyLiving, MIT's Oxygen, Philips' Home of the Future, UW CSE's Portolano, Intel's Proactive Health, UF's Assistive Smart House, Keio's SSLab, etc.

Ubiquitous computing touches on a broad array of disciplines. Though above projects in this field address various aspects of ubiquitous computing, the dependability problems of these systems have not received enough attention. Nonetheless, all of these smart objects and their applications are to be implemented into our everyday environments. Those systems should exhibit different aspects of dependability, such as, reliability, availability, safety, security, etc. Therefore, dependability has become one major prerequisite and a must for future prevalence of the commercial products based on those technologies.

3. Objective

In particular, ubiquitous systems will not be widely deployed if they require users to invest substantial amounts of time or money to keep them operating. Our goal is to provide dependability methodology and potential tools that (1) maximize the dependability of ubiquitous systems, while (2) minimizing the cost of operation (including deployment and maintenance costs).

If we succeed to make ubiquitous system deployed widely, they will become an integral part of the infrastructure upon which our society depends. (1) While some ubiquitous systems might be mission critical (e.g., health-care, transportation, etc), the dependability bottom line of these deployed ubiquitous systems are *safety-guaranteed*. (2) Other ubiquitous systems can mostly be categorized as *QoS-related dependability*: while the unavailability of a few such systems might be a mere inconvenience, the concurrent outage of a large number of systems might have broad economical consequences. For instance, it might be acceptable if a few ubiquitous home networking go down, but if a large number fail concurrently, our society might be adversely impacted.

Hence, we would like to use the term ***ubiquitous dependability*** to emphasize the difference from *traditional dependability* in control systems. In addition to inheriting most of the dependability problems from standard control systems, the special characteristics of the mobile and wireless environments upon ubiquitous systems introduce new challenges that have only begun to be studied.

4. WSN challenge

We see one important challenge in ubiquitous computing as building dependable wireless sensor/actuator networks (including RFIDs), as this is one key player as well as critical milestone of ubiquitous future: WSN (Wireless Sensor Network) is on the brink of widespread applications in logistics, transport, manufacturing, distribution, retail, healthcare, safety, security, law enforcement, intellectual property protection and many other areas.

It is our hypothesis that WSN is likely to have a long lasting impact on our society, from both technical and business perspectives. Therefore, it is vital that researchers and developers investigate and address the dependability challenges in WSN before they have a chance to cripple the promise of the ubiquitous future.

5. Our contribution and approach

We are mainly interested in middleware, service-oriented architecture and networking issues for wireless sensor networks. Sensor technologies are not considered.

As the previous work [1], we have described a formal approach to context-aware ubiquitous computing: we offer the context-aware action systems framework, which enables the correct-by-construction approach [2]. The context-aware action systems framework is supported by its associated refinement calculus [3], which gives us the formal techniques to verify the correctness of the specification and develop the system stepwise from an abstract specification to its implementation.

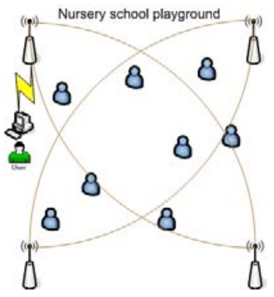


Figure 1. Smart kindergarten case study

With this formalism, we stepwise derived context-aware services for mobile applications [4], and implemented a smart context-aware kindergarten scenario where kids are supervised unobtrusively with wireless sensor networks [5]. As shown in **Figure 1**, the children are allowed to move freely in a predefined area (playground), and the supervisor is able to get the location information of all nodes (visually). When a child leaves the predefined area, the alertness level of the system increases, and the supervisor is informed. Higher alertness level implies intensified communication.

Based on the experiences gained from the case study, we outlined an abstract design framework for wireless sensor networks and provided guidelines with the intension to ease reasoning about WSN as a system and its applications [6]. **Figure 2**, deducted from Nokia's end-to-end model [7], presents a sensor network architecture that factors out the key functionalities required by applications and composes them in a coherent structure, while allowing innovative technologies and applications to evolve independently.

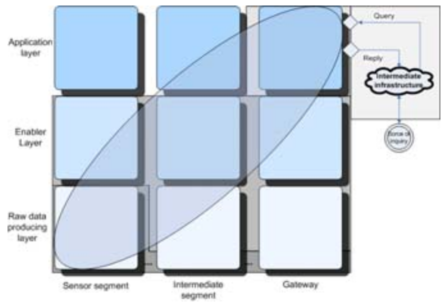


Figure 2. WSN architectural framework

Further, by taking a formal view of context-aware computing that integrates different perspectives, we look at the formal foundation and software engineering techniques for ubiquitous context-aware and context-dependent service derivation and application development, and propose a notion of synthesizing reliable complex systems from vast numbers of unreliable components, emphasizing the relationships between context and system. In particular, we developed a context-dependency model as a rigorous basis for the further development of a formal framework for design and evaluation of context-aware technologies [8].

6. Concluding remark

Dependability issues are approached using a variety of methods. Formal methods in general and verification techniques in particular are used to guarantee software correctness. These methods can also be used for the design of communication networks and hardware, but they are not enough. Here simulation and fault tolerance methods are also appropriate. In all, our approach is to combine these methods into a system design approach for dependable ubiquitous systems in different levels.

References

[1] L. Yan, K. Sere, "A Formalism for Context-Aware Mobile Computing", *Proc. Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Cork, Ireland, 2004.

[2] R. -J. Back, K. Sere, "From Action Systems to Modular Systems", *Software - Concepts and Tools*, (1996) 17: 26-39.

[3] R. -J. Back, J. Wright, *Refinement Calculus: A Systematic Introduction*, Graduate Texts in Computer Science, Springer-Verlag, 1998.

[4] M. Neovius, C. Beck, "From requirements via context-aware formalisation to implementation", *Proc. the 17th Nordic Workshop on Programming Theory*, Copenhagen, Denmark, 2005.

[5] C. Beck, *An application and evaluation of Sensor Networks*, Master thesis, Åbo Akademi, Finland, 2005.

[6] M. Neovius, L. Yan, "A Design Framework for Wireless Sensor Networks", *Proc. of the 19th IFIP World Computer Congress*, Santiago De Chile, Chile, 2006. To Appear.

[7] J. Ziegler, *End-to-End Concepts Reference Model*, Nokia, 2003.

[8] M. Neovius, K. Sere, L. Yan, "A Formal Model of Context-Awareness and Context-Dependency", *Submitted to 4th IEEE International Conference on.Software Engineering and Formal Methods*, Pune, India, 2006.

Concurrency on and off the sensor network node

Matthew C. Jadud, Christian L. Jacobsen, Damian J. Dimmich

1. INTRODUCTION

The Transterpreter is a small, portable run-time for the occam-π programming language[14, 2]. As a language, occam-π provides powerful constructs for safely managing concurrency and parallelism in a framework derived from Hoare's Communicating Sequential Process algebra (CSP), a model of concurrency with message-passing synchronisation primitives[12]. Given the fundamentally parallel nature of wireless sensor networks (WSN) and their nodes, we believe there is great value in beginning with a well-defined concurrency model for reasoning about, and ultimately authoring, software on the network. Here we introduce our run-time in light of other popular environments and languages for WSN applications, and some thoughts on possible future directions given our experiences.

2. OPERATING SYSTEMS FOR SENSOR NETWORKS

Broadly speaking, software developed for wireless sensor networks either works as a custom application tailored to the hardware, or relies on services provided by an operating system. Where an OS layer is employed, these are either open-source projects[4, 1, 11] or commercial solutions[20, 9, 19, 3]. In all cases, they provide different guarantees; for example, eCos[4], an embedded version of the Linux kernel, provides a rich, POSIX environment for programmers, while VxWorks, a proprietary run-time system, delivers hard real-time guarantees to programmers developing for small devices.

In almost all cases, these run-time environments provide an impoverished model of concurrency. Most operating systems for WSNs provide an event-based model of concurrency, and rarely provide any safety for the programmer working in this space. Race hazards, deadlock, livelock, and all the other dangers of concurrent execution face a programmer working in the constrained space of a sensor mote. While environments like TinyOS[11] and the corresponding language nesC[8] provide some compile-time race condition checks, the TinyOS concurrency model of *events* and *tasks* is difficult to reason about—either formally (using modern tools for verification) or informally (as the TinyOS programming model is rather unique).

2.1 On models and algebras

The CSP algebra, developed by Tony Hoare, has provided a sound model for reasoning about concurrency for thirty years. A fundamentally cooperative model of concurrency, it has been adapted into the real-time space[17, 18], as well as crossed into many other paradigms. Furthermore, it was implemented in hardware in the form of the Transputer[21]; this implementation of the model (in the form of the occam2 programming language[13]) is well documented, and occam2 programs (as well as many programs written in occam-π) are formally verifiable.

Using existing documentation, we have built a "Transputer interpreter," or the *Transterpreter*[14]. This virtual machine (VM) for the occam-π programming language requires approximately 12KB of space on 16-bit architectures, and executes a concise (space-conserving) Huffman-encoded byte code. This VM is designed to be portable and executes code on all major desktop platforms, and has been ported to some PDAs, mobile phones, and other small devices. It has been most actively used at Kent in teaching robotics both on the LEGO Mindstorms and the Pioneer3 robotics platforms[6, 15].

Unlike TinyOS, Mate[16], Mantis[1], eCos, or other virtual machines intended for small devices, we have made a focus of providing well-reasoned support for concurrency. While a cooperative model of concurrency can provide some challenges in the context of hard real-time operation, it eases many other programmer tasks in the face of concurrent and parallel systems. We believe it is important to explore the use of concurrent programming languages (or languages that provide powerful and appropriate abstractions for managing parallelism and concurrency) in the embedded systems space.

3. MANAGING CONCURRENCY ON THE WSN

The occam-π programming language makes it trivial to execute processes in parallel: we simply declare them in a `PAR` block. To handle communication between processes, we make use of the CSP idioms of sending (`!`) and receiving (`?`) data over a unidirectional, blocking, point-to-point channel between two concurrent processes. To handle multiple inputs to a process at once, we ALTernate over many inputs, and deal with them as they are ready. What makes these abstractions particularly powerful is that they are not limited to a single WSN node; we might just as easily be expressing communications between a radio driver and a buffer in our software as between two processes running on two separate nodes in the network. The consistency of this model, both on a node and between nodes, is a boon (in our experience) to developers. For example, we have no need for "patterns," as described by Gray et. al. with respect to TinyOS[7]; our concurrency model already has natural concurrency patterns, and we can implement them directly in occam-π.

That said, there are challenges for principled languages and run-times. For example, hard real-time constraints are always a challenge, in any language—unless you evolve your language to explicitly support time[10]. While our run-time provides a clean abstraction for communications—which can easily be used for inter-node communications—robustly hiding the complexities and dangers of wireless communications between nodes "in the wild" is a challenging abstraction to get right. Also, developing a portable virtual machine requires trade-offs, and power consumption is one: is a byte code interpreter running on a small device too power hungry for use in the general case? Our initial tests suggest that this is not a significant concern, but there is a subject for future work and discussion.

Lastly, using a "different" language brings its own set of challenges. To leverage existing code, we provide facilities for bridging from occam-π to "foreign code" (C libraries, etc.)[5]; this is a fragile process, as the C-code may violate invariants our compiler previously guaranteed regarding safety in the face of concurrency. And change never comes freely; despite being a fundamentally unsafe tool, programmers may prefer to implement using eCos and C, as opposed to learning a new language that forces them to "think differently," regardless of the safety such a change might bring.

4. REFERENCES

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, New York, NY, USA, 2003. ACM Press.

[2] F. R. M. Barnes and P. H. Welch. Communicating Mobile Processes. In *Communicating Process Architectures 2004*, pages 201–218, 2004.

[3] T. Brusehaver. Linux in air traffic control. *Linux J.*, 2004(117):10, 2004.

[4] C. Curley. Open source software for real-time solutions. *Linux J.*, 1999(66es):33, 1999.

[5] D. J. Dimmich and C. L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press.

[6] D. J. Dimmich, C. L. Jacobsen, M. C. Jadud, and A. T. Sampson. The robodeb player/stage/transterpreter virtual machine. http://robodeb.transterpreter.org, 2006.

[7] D. Gay, P. Levis, and D. Culler. Software design patterns for tinyos. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 40–49, New York, NY, USA, 2005. ACM Press.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[9] Green Hills Software. http://www.ghs.com/products/velosity.html, 2006.

[10] K. Hammond and G. Michaelson. Predictable space behaviour in fsm-hume, 2002.

[11] J. Hill. A software architecture supporting networked sensors, 2000.

[12] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[13] Inmos Limited. *occam2 Reference Manual*. Prentice Hall, 1984. ISBN: 0-13-629312-3.

[14] C. L. Jacobsen and M. C. Jadud. The Transterpreter: A Transputer Interpreter. In *Communicating Process Architectures 2004*, pages 99–107, 2004.

[15] C. L. Jacobsen and M. C. Jadud. Towards concrete concurrency: occam-pi on the lego mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 431–435, New York, NY, USA, 2005. ACM Press.

[16] P. Levis and D. Culler. Mate: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[17] G. Lowe. Prioritized and probablistic models of timed csp.

[18] G. Lowe. Relating the prioritized model of timed csp to the timed failures model, 1992.

[19] QNX Software Systems. http://www.qnx.com/, 2006.

[20] VXWorks. http://www.windriver.com/, 2006.

[21] C. Whitby-Strevens. The transputer. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 292–300, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

SEUC 2006 Workshop Programme

Session 4: Model-based Approaches

The Self-Adaptation Problem in Software Specifications

Klaus Schmid
University of Hildesheim
Marienburger Platz 22
D-31141 Hildesheim
schmid@sse.uni-hildesheim.de

ABSTRACT

Future ubiquitous computing systems will need an unprecedented level of adaptability and autonomy. From a software engineering point of view, this desire for adaptability poses a major challenge as it is in direct opposition of the traditional requirements engineering goal of a system specification that is very precise and complete.

In this paper, we will discuss some implications of this problem and will outline some approaches to dealing with this challenge.

1. SOFTWARE ADAPTIVITY

Ubiquitous computing is not only characterized by its omnipresence, but more importantly by the demand that these individual computing nodes provide optimally adapted end-user services [1]. As a consequence, we can expect any ubiquitous computing system to be capable of several forms of adaptivity. These forms of adaptivity can be characterized as follows based on the *adaptivity drivers*:

Task-adaptivity: Even for a fixed environment and a fixed user, the specific tasks will vary from time to time. The acceptance and usefulness of ubiquitous computing systems can be greatly improved, if the system adapts its behavior to the specific tasks the user performs.

User-adaptivity: even given the same tasks, different users might have different preferences on how to get support for them (e.g., expert vs. novice). In particular, handicapped users need a different level of support.

(Physical) Environment-adaptivity: depending on the environment, the system must provide information through different channels, using different representations and so forth (e.g., in a load environment we will prefer visual input over audio input).

Function-adaptivity: depending on the other devices (and thus functional capabilities) that are around the overall service level which the integrated system will be able to provide may vary significantly. As a result, the system should be able to adapt to different functional environments.

All of these forms of adaptivity are relevant to ubiquitous systems, although to different degrees. In particular, we regard function-adaptivity as a major issue for all kinds of ubiquitous systems as in complex multi-node networks certain services may always be only temporarily available. Further, in an open system additional services may become available during its life-time. Here, we would prefer if the system would be able to take advantage of such incrementally available services.

All of these forms of self-adaptation of software systems have in common that they may lead to an explosion of the space of possible behaviors. In order to discuss the problem of how to deal with this space explosion more precisely, we may categorize adaptivity in the following ways.

First, we can distinguish different forms of adaptivity based on the *direction of adaptivity*:

- *Corrective adaptivity:* in this case a system aims at improving its current level of service quality. This possibility of improvement is desired from a change in the environment, e.g., if additional services become available.
- *Enhancing adaptivity:* in this case a system aims at improving its current level of service quality. This possibility of improvement may be triggered from a change in the environment, e.g., if additional services become available.
- *Contextual adaptivity:* in this case a system aims at adapting to a change in a context parameter. As a result of this change the current behavior is less desirable and the system aims at achieving the previous level of appropriateness by changing its behavior. This can be seen as a combination of corrective and enhancing adaptivity.

On the other hand, we distinguish different forms of adaptivity by their *range of adaptivity*:

- In *bounded adaptivity,* we can determine the range of possible adaptations systematically at development time.
- In *open adaptivity,* there is either an infinite range of possibilities (that cannot be expressed by a parameter), or we simply do not know enough about the possibilities (e.g,. new services that will become available in the future).

The following table provides an overview of the different forms of adaptivity and the relationships among them.

Drivers	**Direction**	**Range**
Task	Contextual	Bounded
User	Contextual, (enhancing)[1]	Bounded (open)[1]
Environment	Contextual,(enhancing)[1]	Bounded (open)[1]
Function	Corrective, enhancing, contextual	Bounded, open

[1] this category requires a machine learning system

2. THE SOFTWARE ENGINEERING VIEWPOINT

Traditional software engineering methods, especially software specification techniques, work on the assumption that they should produce a complete specification of the behavior of the final system, given a complete list of all possible inputs. Especially if we need to trust the final system, it is very important that it always behaves in a way that the user accepts as foreseeable and useful. This approach is at odds with self-adaptation capabilities of a ubiquitous computing system.

As a consequence of the adaptivity described above, software engineering approaches must become sufficiently sophisticated to deal with this range of flexibility of the final system. Currently we see two main approaches of dealing with this issue:

- *Variability modeling:* this approach aims at the integrated description of a whole range of systems. Both commonalities among the systems and differences are described in an integrated system.
- *Goal-modeling:* this approach aims at justifying the specification of a software system in terms of the goals the users would like to achieve with it.

We will now discuss each of these approaches and their applicability briefly.

2.1 Variability Modeling

Variability modeling is widely used in product line engineering in order to specify (and develop) various systems in an integrated manner [2, 3]. Here, commonalities and variabilities are described in a single model, while providing a mark up of variations and the context of their applicability.

As product line engineering focuses on development time variability, it is always possible to work with a bounded range of variability. Even if a new system requires unforeseen variations, we can update the specification and continue the life-cycle from there.

While these approaches are traditionally not used for runtime variability, they can be easily extended in this way. However, their major limitation from the point of view of specifying adaptivity is that they focus only on bounded adaptivity. While this is not much of a problem in the context of product line engineering, it can be a major problem in specifying adaptive systems as shown above.

2.2 Goal Modeling

Goal modeling aims at describing explicitly in a hierarchical manner the underlying goals for a software system [4]. In this way, both the system specification and its underlying rationales are derived in an integrated manner.

So far, this has only been applied in a static manner, however, these goal trees are conceptually strongly related to planning systems and theorem provers.

As a result, it could be possible to enhance this approach by explicitly providing the necessary knowledge so that a system would be able to compare different function invocations with respect to their contribution to achieving a certain goal. If the necessary knowledge would be provided at runtime, this would allow extending goal-based requirements modeling to the handling of open adaptivity.

In the context of open user or environment adaptivity, this required knowledge needs to be provided beforehand. In this case the various behaviors are not described explicitly, but they are described implicitly as the available knowledge provides boundaries on the possible behaviors.

On the other hand for functional adaptivity, additional services could provide the necessary knowledge for their integration along with the service definition. This would require a common ontology of both application and services in order to support the service integration. In [5] we discussed this approach and its prerequisites further.

3. CONCLUSIONS

In this contribution, we discussed some forms of adaptivity that can be expected from ubiquitous computing systems. We classified these forms of adaptivity and related them to different forms of modeling that could be applied to adaptivity and outlined some challenges in this area.

From our point of view, both variability modeling and goal modeling hold promises for addressing the notion of adaptivity, although for different situations. We are currently working on the integration of these approaches into requirements engineering and their extension towards addressing the challenges of ubiquitous computing.

REFERENCES

[1] M. Weiser, *The computer for the 21st century*, Scientific American, September 1991.

[2] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns;*: Addison-Wesley, 2001.

[3] K. Schmid and I. John, "A Customizable Approach to Full-Life Cycle Variability Management," *Science of Computer Programming*, vol. 53, pp. 259-284, 2004.

[4] A. Lamsweerde, Goal-Oriented Requirements Engineering: from System Objectives to UML Models to Precise Software Specifications", ICSE '03 Tutorial, Portland, 2003.

[5] K. Schmid, M. Eisenbarth, M. Grund: *From Requirements Engineering to Knowledge Engineering: Challenges in Adaptive Systems*, Workshop on Service-oriented Requirements Engineering (Soccer) at the International Conference on Requirements Engineering (RE’05), 2005, also available as Publication of the Fraunhofer Institute for Experimental Software Engineering, IESE-Report No. 118.05/E

Adapting Model-Driven Architecture to Ubiquitous Computing

Julien Pauty, Stefan Van Baelen, Yolande Berbers
K.U. Leuven, Department of Computer Science
Celestijnenlaan 200 A, B-3001 Leuven, Belgium
{Julien.Pauty - Stefan.VanBaelen - Yolande.Berbers}@cs.kuleuven.be

Deploying ubiquitous computing applications in the real world

Ubiquitous computing aims to support the user in his everyday tasks by proposing him relevant and implicit services. Such a user support imposes to deploy applications in different locations, settings and conditions. Ubiquitous computing applications are indeed deployed in and linked to distinct physical spaces.

Current software engineering practices to develop ubiquitous computing applications mix top-down and bottom-up approaches. To design an application and its accompanying services, ubiquitous computing developers and researchers use a top-down approach, by analyzing the needs of the user and how to fulfill these needs. Afterwards, the application needs to be actually developed and deployed in the target physical setting. During this phase the application developers use a bottom-up approach. They start by choosing the best technologies with respect to the target application and the physical setting wherein the application will be deployed. Afterward, the application may also have to be adapted to the limitations of the available and applied technologies.

Deploying a ubiquitous computing application in the real world implies that the application is deployed in different settings with different constraints. The aforementioned mixed approach implies that each time the application is deployed a great part of the application must be redeveloped, which leads to little software reuse, or even to systematic redevelopment from scratch. If we want ubiquitous computing to leave the laboratory settings and to become truly ubiquitous, software engineering techniques must be created in order to ease the development process and the deployment of applications in different physical and technological settings.

In this position paper, we propose to adapt the Model-Driven Architecture (MDA) approach to the ubiquitous computing environment [1] in order to significantly increase software reuse and ease application adaptation to new deployment environments. MDA decouples the applications from their execution platforms via model abstractions and model transformations. An application is initially described by a Platform-Independent Model (PIM) that does not contain any information on the final execution platform and middleware. This model is refined several times via successive transformations, each transformation integrating new constraints like persistency or redundancy. Once the PIM is enough detailed, an execution platform is chosen and the PIM is transformed into a Platform-Specific Model (PSM). This PSM is in turn transformed several times, in order to generate code, compile the application, package the application…

Adapting MDA to ubiquitous computing

Consider an application that supports an elderly person in order to help her staying home alone despite possible disease, such as dementia. This application can monitor the person's activity to detect changes in habits denoting possible aggravation of the person's diseases. In this section, we consider an application which monitors how the person sets up the table for dinner and triggers a visual or sound alarm when the person forgets a key element, such as the forks of the knifes. Such an application must be deployed in different house settings, with users that may be affected by different diseases.

The initial PIM should contain the main application logic and generic geometric logic. In our example, application logic just defines a monitoring loop that regularly checks the configuration of the table and notifies the user when it detects an incorrect table setting. Application logic can also log date and time when alarms are triggered for offline analysis of user habits. Generic geometric logic defines that the table is an area and that several objects must be present at the same time in this area, such as forks, glasses… Several elements must be also relatively located in the area: a plate should be between a knife and a fork.

The PIM is then transformed several times. Each transformation modifies the PIM to integrate user constraints and geometric constraints. These constraints are technology independent. User constraints describe user disabilities, such as deafness or blindness. For example, if the user is blind the model is transformed so that is only relies on sound output. Geometric constraints describe the geometric configuration of the place where the application will be used. For our application, this geometric configuration describes the size and shape of the table and the objects.

Once user and geometric constraints are integrated, the PIM is transformed into a PSM in order to integrate environment constraints and technology constraints. Environment constraints describe the environment where the application will be used: home/factory, indoor/outdoor, noisy, dark… For example outdoor applications may rely on mobile devices. Factory applications may need communication technologies that can resist to electromagnetic perturbations. In a home environment, technology must be as discrete as possible. In our application, the loudspeakers could be integrated into the table. Technology constraints describe the limitations of the technology, such as screen size for a user terminal or accuracy for a location sensor. Transformations take into account these limitations in order to adapt the PSM.

Technology constraints may lower the quality of service of the application. In our example, we may choose RFID or image recognition to detect the table configuration. RFID is a simpler technology than image recognition, but it is also less precise for detecting geometric configurations. It only enables the application to detect if every key element is on the table, but not to detect whether the table is correctly organized. In this case, the service provided by the PSM is a degraded version of the service defined by the PIM.

Discussion

In the previous section we have presented an adaptation of MDA to ubiquitous computing. This approach can significantly increase software reuse and ease application adaptation to new deployment environments. To reduce adaptation time towards a new deployment environment, we can choose the most elaborated PIM that is adapted to the new environment and use this model as the base for our adaptation. This results in the fact that only the remaining transformations must be performed again instead of having to develop the whole application from scratch.

If we want to save development time transformations must be automated and reused as much as possible. We have seen in the previous section that applying MDA to ubiquitous computing involves new transformations, such as taking into account technological and user constraints. Research results to solve these separated problems already exist in the ubiquitous computing community. We need now to synthesize and group them, in order to create automated and reusable transformations. For example, a lot of work has been done to adapt applications to limited screen size. We now need a generic methodology that could be used in order to create the corresponding model transformation.

New domain-specific languages also need to be developed, or existing languages need to be adapted, in order to define the different models. We need a language to define user and technology constraints, to describe geometric configurations… Several areas have been already investigated, such as user interfaces [2].

We do not claim that our proposal to adapt MDA to ubiquitous computing is neither complete nor already applicable. It is mostly intended to support discussion during the workshop. Indeed, adapting MDA to ubiquitous computing raises several questions:

- Which UML profiles and domain-specific languages should be defined in order to obtain fitted notations for ubiquitous computing?
- In which order should we perform transformations? Indeed, the transformation order has an impact on model reuse. For example, if user transformations are performed before geometric transformations, then when the user changes we must restart from the initial PIM and perform every transformation, despite the fact that the geometric transformations remain the same.
- Is the approach of gradually refining models anyhow applicable for ubiquitous computing? Maybe an orthogonal approach, in which a developer can start independently from a PIM by introducing user constraints, geometric constraints or technology constraints in parallel, integrating the 3 models afterwards into a single PSM model is better fitted.
- Is adapting MDA to ubiquitous computing a reasonable approach, or should we try to develop new techniques? On the one hand, experience can be drawn from the MDA community and tool support for MDA can be used for the development of ubiquitous computing software. On the other hand, differences between traditional MDA applications and ubiquitous computing applications may be too important to make MDA a sound approach for ubiquitous computing.

References

[1] S. Bonnet. Model driven software personalization. In *Smart object conference (SOC'03)*, 2003.

[2] J. Vanderdonckt. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In *Advanced Information Systems Engineering: 17th International Conference (CAiSE'05)*, 2005.

Efficient Modelling of Highly Adaptive UbiComp Applications

Andreas Petter, Alexander Behring, Joachim Steinmetz
University of Technology Darmstadt
Telecooperation Group, Hochschulstr. 10, 64289 Darmstadt, Germany
$\{a\_petter, behring, joachim\}$@tk.informatik.tu-darmstadt.de

Adaptive applications in ubiquitous computing depend on large sets of parameters while requiring device independency. To efficiently develop such applications, we discuss a possible solution, we call "amending models".

1 Introduction

Ubiquitous computing introduces new challenges for software engineering, among these are support for a large variety of platforms and numerous different contexts. By leveraging modeling, a great degree of abstraction from the underlying platform is achieved. By interpreting models at runtime, the abstraction is further improved. But still there is the need to go into different contexts of usage, which would have to be modeled explicitely. Therefore an approach where models are adapted to changing contexts at runtime and contexts not known at development time can be dealt with is sought after. Ideally such an approach also reduces the development efforts associated with the development of ubiquitous computing applications.

Amending Models are a possible solution to this question. They especially allow modelling for applications adapting to numerous different contexts. Amending Models address this problem by enabling an application designer to model only parts of his model and let other model elements, due to introduction or change of context, be explored and modelled automatically during runtime.

In the next section, we briefly explain what an amending model is and argue against and for it in sections 3 respectively 4, also taking into account the effects of its application. Section 5 scetches a possible setup and section 6 concludes and gives further perspectives on this field.

2 The Thesis

Building on the ideas expressed in [1] and [7], we argue for utilizing and interpreting models at runtime and amending them by leveraging *Artificial Intelligence (AI)* concepts in the adaptation process. Further, we propose that

to develop highly adaptive software for ubiquitous applications in a cost effective manner, a standard modeling approach is not sufficient. The concept of amending models is a feasible approach to adress this problem.

An *Amending Model* hereby is a graph with nodes (e.g. tasks, UI elements, ...) carrying attributes and the possibility to link the nodes through edges (e.g., in a task model, UI model, ...). An initial setup (a start model) is given by the developer. During runtime, the application is adapted by amending its model(s). This is done through applying AI algorithms and thereby, e.g., changing the links between and the usage of nodes, as well as inserting and removing nodes. The adaptation process hereby takes into account the *Amendment Context*, which includes all information that could be relevant to amend the model.

By reducing the number of different Amendment Contexts (e.g., different available sensor systems) that are modeled explicitly the adaptation process must be able to cope with by then unknown parameters. Therefore, standard model transformations alone aren't suitable to transform Amending Models. AI algorithms are a possible way of dealing with such a situation.

In order to produce models compliant with the specification of the system, validation and evaluation become an essential part of the adaptation process. This compliance could be achieved by, e.g., choosing suitable amendment methods or validation of the model.

3 Arguing against Amending Models

Validation consequently is an integral part of amending models. Unlike in current *Software Engineering (SE)*, (partial) validation must be done automatically at runtime. This *introduces new complexity and the need for special knowledge* how to write validation rules. However, if validation rules can be generated automatically from e.g. a goal model, this drawback is greatly reduced.

To reduce the number of models explored at runtime for one adaptation step, heuristics seem most suitable. *Formulating heuristics can be difficult* [4] and might result in the need for AI experts. This drawback could be reduced by improved tool-support, potential reuse of heuristics and results of further research.

Modeling might lead to *unpredictability*, which in turn results in *reduced usability*, e.g., for user interfaces [9]. Amending Models intensifies these problems, because the Amendment Context is not modeled explicitly. Besides improving the adaptation process itself, *model templates* could provide predefined model parts for certain Amendment Contexts (e.g., UIs for the most used devices) to regain predictablity. Furthermore, there has been improvement in this area, e.g., [2], [8] and [7].

4 Arguing for Amending Models

Modeling highly adaptive applications with a traditional approach implies to model every possible Amendment Context (e.g., for User Interface Adaptation in [3]). Amending Models loosen this restriction by providing an abstraction and allowing to model only the key ingredients of an application.

Design flexibility is gained through opening up a trade off between the effort of modelling elements and computational power at runtime. Closely connected to this is the decision of the developer to spend effort in modeling a specific Amendment Context (e.g., UIs for a certain device) or relying on the adapatation process.

By reducing the sheer number of elements to model, development time is reduced. By the same effect, a better overview over the application model is gained. This can improve the effectiveness of the development and reduce bugs. Consequentially, *development costs are reduced* through the application of Amending Models. Sometimes this cost reduction could be the prerequisite to start developing in the first place.

In industry software projects change requests (in all phases of the software lifecycle) are common. Especially changes affecting wide parts of an application are difficult to implement. Amending Models reduce the effort needed to implement a change request by abstraction from the Amendment Context. This abstraction also results in improved reusability. Consequently, in fortunate cases, a change request could be solved by writing a single transformation rule. Additionally, since Amending Models are interpreted at runtime, such a change might even be implemented after deployment of the application. Overall, *maintainance costs are reduced.*

Recapitulating, through Amending Models the *adaptivity of applications is improved.* An important aspect of this is improved *Plasticity* [11], denoting the size of the Amendment Context an UI stays usable in.

5 Action Example

In [1] an software architecture reference model for UI adaptivity is discussed. In the following, a brief description of a set of development methods and tools that could be used is given:

- adapted MDA (see [6]) concept and pattern
- MOF (see [10]) compliant metamodel similar to UML
- specific tools for abstract and concrete UI design
- integration of domain specific languages

A setting like this would leverage the spread of UML wihin software development companies and the concepts of MDA. The metamodel could be downsized and taylored to the specific needs of the software company using it. At the same time, it enables the usage of powerful domain specific languages. Finally Myers concerns [9] are adressed by integrating special UI design tools to reduce unpredictability. Thus setting is aimed to facilitate cost-effective development of context-aware, adaptive ubiquitous computing applications.

6 Conclusion and Outlook

We argued for *Amending Models*, which are modified and interpreted at runtime in order to adapt to different Amendment Contexts. By using this concept, typical challenges of Ubiquitous Computing are addressed and

- *development costs are reduced*, by reducing development time and improving effectiveness,
- *design flexibility is gained*, by opening up a trade-off between invested effort for detailed modeling of possible Amendment Contexts versus more abstract modeling, and
- *adaptivity of applications is improved* by reducing direct dependence of models on parameters.

Further research within this area may adress:

- identification of possible transformation processes,
- characterization of tools used for modelling with this paradigm, and
- detailed characterization of relations between different elements in this concept.

Some of these topics will be addressed within the eMode project [5].

Bibliography

[1] L. Balme, A. Demeure, N. Barralon, J. Coutaz, and G. Calvary. Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. In *EUSAI*, pages 291–302, 2004.

[2] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, and J. Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *TAMODIA 2002*, pages 18–19, Publishing House Bucharest, Romania, 2002. INFOREC.

[3] B. Collignon. Dégradation harmonieuse d'interfaces utilisateur. Master's thesis, Universite Catholique De Louvin, 2004.

[4] R. S. Engelmore (ed.). Knowledge-based systems in japan. Technical report, JTEC, Baltimore, MD, USA, 1993.

[5] EMODE. `http://www.emode-projekt.de`.

[6] J. Mukerji (ed.) J. Miller (ed.). Mda guide version 1.0.1. Technical report, OMG, 2003.

[7] J.-S.Sottet, G. Calvary, and J.-M. Favre. Towards model-driven engineering of plastic user interfaces. In *MDDAUI*, 2005.

[8] G. Mori, F. Paternò, and C. Santoro. Tool support for designing nomadic applications. In *IUI '03*, pages 141–148, New York, NY, USA, 2003. ACM Press.

[9] B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[10] OMG. Mof specification, 2002.

[11] D. Thevenin and J. Coutaz. Plasticity of user interfaces: Framework and research agenda. In *Interact'99*, volume 1, pages 110–117. IOS Press, 1999.

Model-driven design of ubiquitous interactive applications

Jan Van den Bergh
Expertise Centre for Digital Media – Institute for BroadBand Technology
Hasselt University – transnationale Universiteit Limburg
Agoralaan
3590 Diepenbeek, Belgium
Jan.VandenBergh@uhasselt.be

Karin Coninx
Expertise Centre for Digital Media – Institute for BroadBand Technology
Hasselt University – transnationale Universiteit Limburg
Agoralaan
3590 Diepenbeek, Belgium
Karin.Coninx@uhasselt.be

ABSTRACT

The ubiquitous availability of interactive applications means that the diversity of people use an increasing number of different mobile or networked computing devices with diverse but constrained input and output capabilities. This increases the interest in programs and technologies that exploit context knowledge, such as the user's location or past behavior. At the same time, the behavior of the application should be predictable and correspond to the user's expectations. In this document, we propose the usage and generation of high-level models incorporating context information for certain aspects of the design while still allowing some designer freedom in other areas. The usage of models, with roots in both human-computer interaction and software engineering, should however not divert attention away from the final users, which should be involved in the design process and provide feedback on prototypes that are generated from the specified models.

1. INTRODUCTION

In our work, we focuss on the design of context-sensitive interactive applications. These applications pose specific problems, not fully addressed in current methodologies. We specifically target the early stages in designing those systems. The goal is to provide models that can be understood by human-computer interaction (HCI) experts, software engineers and programmers, and customers [1]. Those models should convey enough information to generate the skeletons of context-sensitive user interfaces. We do not attempt to automate much of the detailed design, such as detailed layout and style.

We build our approach on existing models in both the software engineering and HCI community and provide extensions where we deem them necessary. In the following sections, we provide an overview about the work we performed up till now, our current and future work. We start by giving some background by the envisioned design process and how our work supports that design process. Then we introduce the different models, as well as the extensions we provided. Finally, we present some thoughts about tool support and formulate some conclusions.

[1]When direct exposure to the models is not an option, we propose to use generated prototypes.

2. APPROACH

We believe that the use of models can significantly ease the creation of context-sensitive interactive systems, while not preventing an agile methodology. The process typically starts by specifying the user's tasks and the concepts that are important to perform these tasks. These tasks and concepts can be described using well-known models and notations. The ConcurTaskTrees (CTT) notation [2] can be used for (user) task modeling, while the Unified Modeling Language (UML) [1] can be used to express the concepts and their interrelations.

The concepts can be annotated with context information to specify which concepts represent context information that will be used at runtime and whether it will be generated (automated generation) or profiled (manual input). Also the sources of the context information can be specified. The information in these models can be used to describe different contexts of use. For each of the tasks in the task model, one can specify the relevant contexts of use, whether the effect of a task is noticed by an external entity (and fed back to the system), or whether it is executed by another system, . . . Based on these "context-sensitive" models, one can generate (high-level) system interaction models and user interface models. These models can be expressed using UML. We consider semi-automatic generation of prototypes at different levels of abstraction to be crucial.

Finally, user interface designers can design the user interface guided by the information in the models, in addition to their experience and guidelines. We envision that most if not all of the models can be kept in sync by a tool, especially when only high-level design is done in the models and a high-level declarative user interface description is generated by the tool to be augmented by a stylesheet that reflects the work done by the designer.

3. HIGH-LEVEL MODELING

The proposed model notations build on existing model notations used in software engineering or the human computer interaction community.

User Task Model

The user task model is modelled using the Contextual ConcurTaskTrees notation [3] (CCTT). With CCTT we propose a hierarchical task model notation based on the CTT notation, introduced by Paternò [2].

It enhances the CTT with an extra task type: "the context task". The context task represents an activity that has influence on the context of use and as such has an indirect influence on the subsequent execution of the task. One can distinguish four different kinds of context tasks: each of the three concrete task types (user task, application task and interaction task) of the CTT can have influence on the context of use. In addition, there are also external activities that can have an influence on the execution of the tasks. These tasks are called environment tasks.

Application Model

The context and application model is represented using the class diagram of the UML [1]. Stereotypes – stereotypes are special structures to extend the meaning of UML constructs – have been defined for identifying concepts that are not part of the modelled application but belong to the context of use. The stereotype `profiledContext` is used for context information that is entered by humans, while the stereotype `detectedContext` can be used for context information that is gathered through sensors, or is processed by other software or hardware. The entities that are responsible for detecting the context information can be made by using the stereotype `contextCollector`. All these stereotypes can be applied to UML classes and are part of the CUP-profile [4], just as all other stereotypes used for the discussed models that use UML diagrams.

System Interaction Model

The system interaction model is expressed using UML 2.0 activity diagrams. The activity diagram is extended using stereotypes that correspond to the task types in the CCTT notation. These stereotypes are: `application`, `interaction` , `user`, `environment` and `task` (corresponding to an abstract task in the CCTT with a limited difference in semantics).

Abstract User Interface Model

The abstract user interface model is specified using the UML class diagram. It specifies a hierarchy of abstract interaction components that define which types of actions can be performed using a certain component. Four kinds of user interface components are possible: input components (allow a specific kind of input such as selection or free input), output components (do not allow user interaction), action components (allow the user to interact with the system scope) and group components). Each of the different types of user interface components is represented by a stereotype with an alternative representation.

Context Model

The context model specificies the applicable contexts of use within class diagrams. Each context of use is defined within a stereotyped package using the classes defined in the application model.

User Interface Deployment Model

The mapping of the abstract user interface model to concrete user interfaces is illustrated in the user interface deployment model, expressed using a specialized version of the UML deployment diagram. This model specifies the target platform and context. A stereotype `contextualNode` is defined to link a UML `Node`, representing a software or hardware platform, with a specified context.

4. AUTOMATION

For the model-driven approach to become more efficient, tool support between the user-task model and the system interaction model and between the system interaction model and the abstract user interface model is needed. Furthermore, we consider semi-automated generation of prototypes at different levels of fidelity and abstractness to be crucial for an effective development of context-sensitive interactive applications. In early stages of design, prototypes at an high level of abstraction can be used, especially when supporting hardware and/or software are not available.

5. CONCLUSIONS

We presented our approach for the design of context-sensitive interactive applications, and more specifically the user interface side of the modeling. We believe that a model-driven approach with proper abstractions and good tool-support, which integrates support for the creation and evaluation of prototypes and allows designer creativity, is vital for ubiquitous interactive applications that are easy to use and function predictably and correctly.

6. ACKNOWLEDGEMENTS

This research was partly performed within the IWT project Participate of Alcatel Bell. The research at the Expertise Centre for Digital Media is partly funded by the ERDF (European Regional Development Fund), the Flemish Government.

7. REFERENCES

[1] Object Management Group. *UML 2.0 Superstructure Specification*, October 8 2004.

[2] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer Verlag, ISBN: 1-85233-155-0, 2000.

[3] Jan Van den Bergh and Karin Coninx. Contextual ConcurTaskTrees: Integrating dynamic contexts in task based design. In *Second IEEE Conference on Pervasive Computing and Communications WORKSHOPS*, pages 13–17, Orlando, FL, USA, March 14–17 2004. IEEE Press.

[4] Jan Van den Bergh and Karin Coninx. Towards Modeling Context-Sensitive Interactive Applications: the Context-Sensitive User Interface Profile (CUP). In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 87–94, New York, NY, USA, 2005. ACM Press.

SEUC 2006 Workshop Programme

Session 5:
Engineering for Humans I

Surveying the Ubicomp Design Space: hill-climbing, fields of dreams, and elephants' graveyards

Michael B Twidale
Graduate School of Library and Information Science
University of Illinois at Urbana-Champaign
twidale@uiuc.edu

Introduction
The famous quotation from the movie the Field of Dreams seems to guide so much techno-optimism from dot-com bubble business plans to ubicomp research proposals: "if we build it, they will come". Sometimes they will. But not always. Is there a better way to explore a design space to help decide what exactly to build, rather than just picking the first idea that occurs to us and then focusing exclusively on how to make it work?

I believe that there is a way, by applying a range of high speed low cost techniques to more actively explore and analyze various design spaces. Even spending a few hours doing this can be worthwhile. My contention is that even this is rarely done. The challenge is to explore and refine those methods and to demonstrate their power and their relatively low cost. It is quite understandable that developers are somewhat skeptical of analytic techniques that appear to mostly slow down development. This is even more the case in the context of research activities where the development activity is itself part of the learning process and the requirements elicitation process.

The problem
In teaching undergraduates and graduate students with strong technical development skills I have noticed that unsurprisingly they want to build applications and start building as soon as possible. That is perfectly understandable and something I want to encourage. One learns a lot about a topic by trying to build an application. However the first idea one has is unlikely to be optimal. Once one has spent a lot of time on working on that design and emotionally committing to it, it can be very difficult to abandon it and start afresh. Available resources just may not permit it. I see such a scenario playing out time and again – an obsessive focusing on one point in the design space that was chosen arbitrarily and too early on. This creates a pedagogical problem and one that can be addressed in planning future classes. However, once noting it, I realized that it applies to many research projects too. The decision of what to build is made hurriedly and in ignorance, often as part of the rush of putting together a grant proposal and is not considered thoroughly. Very experienced researchers can draw on their previous work to make snap decisions in such circumstances and stand a good chance to locating a productive point in the design space. Less experienced researchers are unlikely to be so lucky.

In the case of novel ubiquitous computing applications, our intuitions can be wrong and even misleading. These is ample evidence from the field of CSCW research of how considerable experience in developing single user applications does not guarantee success in developing useful, usable and acceptable collaborative applications. It is most likely that the same occurs when we get up from the desktop and move around and think about workspaces, home spaces, social spaces and moving between these and others. One approach is to acknowledge the problem, but apply a Darwinian, market based solution: just create a whole host of ideas embodied in applications as fast and as productively as possible and let the marketplace of ideas decide which are successful or not. Such a method certainly works but it is very wasteful. Is a more considered approach possible without excessively slow analysis getting in the way of design creativity and innovation?

A solution
The classic, powerful solution to this issue in CSCW is to first do a detailed ethnographic study. I am all in favour of these and have participated in some as both a systems developer and as an ethnographer. However they are very slow and often are better at telling you what not to build rather than what to build. Can such approaches be supplemented by other rapid analysis techniques? Methods that can be applied in minutes or a couple of hours and integrated into rapid prototyping iterations. Below I list some that I've been looking at. Ideally I think the real answer is ways to select from and combine these at different points in a design cycle, tightly integrated with exploratory development with an idea of build in the morning, test, analyse, plan redesign in the afternoon.

Affordance Analysis
Building on the HCI concept of affordances this looks at an individual component of a use case, either how people currently do something or our new proposed device and consider what are the generic kinds of activities that that component affords, supports or enables. Anti-affordances are also considered. In this way it becomes possible to consider a slice-and-dice solution where the new design does not attempt to do everything that the old design does and more and better in all cases, but rather focuses on supporting certain activities for which it is dramatically better and integrates with the old approach for the other activities. As a very simplistic example, an exploration of the technological and social affordances and anti-affordances of cellphone use in public settings can lead to ideas such as sound output but button-press input as a way to enable conversations in public settings where listening is acceptable but speaking is not. A very simple interaction might involve pressing a button that transmits a recorded message such as "I'm in a public place. I can listen to any message you want to give me, but I can't really talk right now. If you want to ask me questions I can press buttons to say yes or no. Otherwise, I'll get back to you in X [typed in] minutes". Not exactly programming rocket science, but a very simple case of application innovation inspired by affordance analysis.

Goals, Constraints, Opportunities, Issues,
Inspired by SWOT analysis, this is a quick way of considering aspects a design space:
- Goals lists all the desirable features and uses of the envisaged application, acknowledging that some can be contradictory or lead to design tradeoffs.
- Constraints are limitations either of requirements and use or of the available technology or development environment.
- Opportunities lists advantages, most usually caused by new technologies having disruptive effects on traditional cost benefit calculations.

- Issues are things that arise in discussing potential applications and can include features of privacy, ownership, expectations, acceptability, trust etc.

Scenario Based Design, Personas, Body Storming
Building on a rich tradition in this area of trying to envisage the proposed application in actual use and then acting out these scenarios and critiquing them. The main new contribution here is to consider various forms of failure analysis, inspired by HCI cognitive walkthroughs. That is, one begins with the optimistic scenario of how the ubicomp app ought to work and so be a better way of doing tings than the traditional approach. Then at each stage of the scenario we stop and consider "what could go wrong here?" Having identified problem, maybe f learning, use, interpretation, or systems failure, we then consider how people might cope and how the design might cope, either via a redesign to prevent the error, or to mitigate its consequences or to support recovery, or to inspire a rapid exploration of a different design solution altogether.

Creativity and Bad Ideas
The methods above can be characterized as explorations around a particular point on a design space, an attempt to do local hill climbing in that space rather than sticking purely with the point in that space of the first application idea. Working with Alan Dix, we have been looking at how to inspire design creativity to look at entirely different places on the design space. On method is the consideration of 'bad ideas' – design solutions and applications that are clearly bad. By then analyzing exactly why they are bad, one does a local design space exploration very similar to the effects of the techniques above. This can then lead to new ideas for applications that are slight variants of the original bad idea, or inversion of parts of that idea. We have written a couple of papers on that work and wish to explore it further.

Rapid prototyping
This approach is mostly about analysis, but the development of prototypes is a very important part of analysis. Very rapid prototyping is helpful. It has a lot of precedent with the classic story f the developer of the palm pilot carrying a block of wood in his shirt pocket and pulling it out periodically and imagining how it could help his life. In traditional desktop application development, paper prototyping has a long tradition in both interface design, but also forms of requirements capture, especially as part of participatory design. The mapping between sheets of paper and windows on a PC is pretty obvious. What is the equivalent of paper prototyping for ubicomp? Working proofs of concept can also be highly valuable as ways to inspire analysis of what really should be built, so long as they can be put together fast enough that they do not cause a lock-in forcing the project to stick with that particular design. We have been exploring the use of mashups, wikis and other pieces of open source software as ways to produce very crude but operational elements of functionality to illustrate a design idea before committing to a particular implementation

Analysis as a process of exploring spaces
I've talked about 'the design space' as something to be explored semi-systematically as part of low cost analysis. In fact, it is impossible to fully explore this space. It is so huge it is not surprising that students cling to their first design idea. The immensity of the space can inspire a form of 'cognitive agoraphobia' making people reluctant to consider too many or any alternate design ideas for fear of getting swamped by choice and producing noting. The techniques outlined above are ways to explore some small parts of this space without being overwhelmed. However there are several spaces:

- Design space: what we can build, combining features, functionalities, interfaces
- Adoption space: what people want, might want, like, don't like
- Research space: build to learn, to think, to understand, to articulate
- Funding space: fashion, strategies, rhetoric, re-articulating, advocating

Related Work

Dix, A. Ormerod, T., Twidale, M.B., Sas, C., Gomes da Silva, P.A., McKnight, L. (2006). Why bad ideas are a good idea. To appear in Proceedings of the First Joint HCI Educators' Workshop.

Jones, M.C., Floyd, I,R. (forthcoming). Patchworks of Open-Source Software: High-Fidelity Low-cost Prototypes. In "Handbook of Research on Open Source Software: Technological, Economic, and Social Perspectives", K. St. Amant, B. Still (Eds). Idea Group, Inc.

Jones, M.C., Floyd, I.R., Twidale, M.B. (2006). Patching Together Prototypes on the Web. Submitted as a Notes paper to CSCW 2006.

Jones, M.C., Rathi, D., & Twidale, M.B. (2006). Wikifying your Interface: Facilitating Community-Based Interface Translation. Proceedings of DIS 2006.

Jones, M.C., Twidale, M.B. (2006). Snippets of Awareness: Syndicating Copy Histories. Submitted as a Notes paper to CSCW 2006

Twidale, M.B. & Jones M.C. (2005). "Let them use emacs": the interaction of simplicity and appropriation. International reports on socio-informatics 2(2) 66-71.

Twidale, M.B. & Ruhleder, K. (2004). Where am I and Who am I? Issues in collaborative technical help. Proceedings, CSCW04. 378-387.

Twidale, M.B. (2005). Over the shoulder learning: supporting brief informal learning. Computer Supported Cooperative Work 14(6) 505-547.

Twidale, M.B., Wang, X. C., & Hinn, D. M. (2005). CSC*: Computer Supported Collaborative Work, Learning, and Play. Proceedings, Computer Supported Collaborative Learning (CSCL), 687-696.

Twidale, M.B. (2006). Worrying About Infrastructures. CHI 2006 Workshop: Usability Research Challenges for Cyberinfrastructure and Tools.

Connecting rigorous system analysis to experience centred design in ambient and mobile systems

M. D. Harrison and C. Kray
Informatics Research Institute, University of Newcastle upon Tyne, NE1 7RU, UK
michael.harrison@ncl.ac.uk,c.kray@ncl.ac.uk

1 Introduction

Ambient and mobile systems are often used to bring information and services to the users of complex built environments. The success of these systems is dependent on how users *experience* the space in which they are situated. Such systems are designed to enable newcomers to appropriate the environment for the task at hand and to be provided with relevant information. The extent to which a system improves the user's experience of such environments is hence important to assess. Such a focus on experience provides an important trigger for a fresh look at the evaluation for such systems but there are other reasons too why traditional notions of usability need reconsideration:

- the impact of the environment as the major contributor in understanding how the system should work — its texture and complexity
- the possible role of location and other features of context in inferring action and as a result action may be implicit or incidental in the activities of the user — how natural and transparent this inference is.

Due to these and other problems it is difficult to assess ambient and mobile systems early in the design process. For this reason, we are investigating how to relate experience requirements to more rigorous methods of software development.

2 Eliciting user experience requirements

A conclusion that may be drawn from these differences is that the evaluation of such ambient and mobile systems must be carried out in-situ within the target environment, with typical users pursuing typical activities. The problem with this conclusion is that it is usually infeasible to explore the role of a prototype system in this way, particularly due to cost considerations or when failure of the system might have safety or commercial consequences. We therefore need methods that would enable us to establish experience requirements and to explore whether they are true of a system design before expensive decisions are made.

Eliciting experience requirements for an envisaged ambient system can be carried out using a combination of techniques. For example, it can be valuable to gather stories about the current system, capturing a variety of experiences, both normal and extreme, and visualising the experiences that different types of user or persona might have in the design. The results of this story gathering process will be a collection of scenarios that can be valuable in exploring how the new design would behave. These scenarios can be used to evaluate the design [7] perhaps using a specification of the design or using a rapidly developed prototype. Techniques such as cognitive walkthrough or co-operative evaluation can provide valuable complementary approaches to evaluation based on these scenarios [7].

In addition to scenario orientated techniques for elicitation other techniques are also valuable. Techniques such as cultural probes [5] are used to elicit "snapshot experiences" and complement these more scenario orientated approaches to the establishing and discussing of user requirements. The elicitation process here involves subjects collecting material: photographs, notes, sound recordings to capture important features of their environment. While these snippets may make sense as part of a story they may equally well be aspects of the current system that are common across a range of experiences or stories.

3 Formal analysis and experience requirements

A question then is how to make sense of these snapshots. Consider, for example, a system developed to help passengers experience a sense of place in the unfamiliar setting of an airport. One might imagine a combination of ambient displays, kiosks and mobile services for hand-held devices. They combine together to provide an environment in which passengers can obtain the information they need, in a form that they can use it, to experience the place.

Consider a situation exemplifying the kind of system that is being described. On entry to the departures hall, a sensor recognises the electronic ticket. As a result the passenger is subscribed to the appropriate flight and the passenger's context is updated to include current position in the departures hall. The flight service publishes information about the status and identity of queues for check in. A message directing the passenger to the optimal queue is received by the passenger's handheld device because the context filter permits its arrival. This information is displayed on a public display in the departures hall. When the passenger enters the queue a sensor detects entry and adds the queue identifier to the passenger information. As a result different messages about the flight are received by the passenger — this might include information about seating so that the passenger can choose a seat while waiting to check in baggage. This process continues as the passenger progresses through the various stages of embarkation.

In the context of this system a frequent flyer might be anxious about missing his flight. One could imagine in the old system that he might take a snapshot of the public display and comment that he always looks for a seat where this information is visible. He might also comment: that the flight information relevant to him is not always clearly discernible on the display; that delay information is often displayed late and is not updated so there is no sense of there being any progress.

This information is not captured well by a specific scenario because although one such situation can be captured well, the scenario does not cover all situations — that this information needs to be available whatever "path" the user takes. More formal approaches may play a role here. The challenge is whether it is possible to produce models of systems in such a way that it becomes feasible to explore experience issues in the design of these systems. In particular we claim that an approach used in earlier work that combines scenarios with property checking would be of value [2]. In such an approach model-checking may play a role in checking experience requirements. Suppose that a passenger reports that she wants to be able to access "up to date flight information" wherever she is. Properties can be specified of the model that capture this notion of up-to-dateness and used to check the model. Just as in [6] we would use the model to explore the possible paths that passengers might take to reach the flight gate.

4 Technical realisation

We envisage a software architecture to realise systems such as those illustrated here using a publish-subscribe architecture [3] coupled with a sensor interface, public displays and handheld devices as clients, and a model of context to filter published messages. Generic publish subscribe models are a relatively well established area of research [4, 1]. In general these approaches focus on features of the publish subscribe mechanism including:

- reusable model components that capture run-time event management and dispatch
- components that are specific to the publish subscribe application being modelled.

With such models it is possible to explore properties such as:

- when the passenger enters a new location, the sensor detects the passenger's presence and the next message received concerns flight information and updates the passenger's handheld device with information relevant to the passenger's position and stage in the embarkation process.
- when the passenger moves into a new location then if the passenger is the first from that flight to enter that location, then public displays in the location are updated to include this flight information
- when the last passenger in the location on a particular flight leaves it then the public display is updated to remove this flight information
- as soon as a queue sensor receives information about a passeneger entering a queue then queue information on the public display will be updated.

These properties can all be related to the experience that a user has of the system. Checking properties of the model will generate sequences that do not satisfy them. The domain expert will use this information to inspire potentially interesting scenarios. These scenarios may then be used to visualise how different personae would experience them. A potential user might be asked to adopt the persona and then to visualise the system. Paper or electronic prototypes would be used to indicate what the system would appear to be like at the different stages of the scenario.

5 Conclusion

In this brief position paper we argue that experience centred design is of particular importance where a user is situated in a dynamic environment involving an ambient and mobile system. We have outlined approaches to experience requirements elicitation and summarised an approach to analysis that is based on the use of formal methods such as model checking to identify potential problems related to the user experience and a technical realisation based on publish-subscribe models.

References

[1] L. Baresi, C. Ghezzi, and L. Zanolin. Modeling and validation of publish / subscribe architectures. In S. Beydeda and V. Gruhn, editors, *Testing Commercial-off-the-shelf Components and Systems*, pages 273–292. Springer-Verlag, 2005.

[2] J.C. Campos and M.D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8:275–310, 2001.

[3] P.T. Eugster, P.A. Felber, R. Gerraoui, and A. M. Kermarrec. The many faces of publish subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[4] D. Garlan, S. Khersonsky, and J.S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN03)*, Portland, Oregon, 2003.

[5] W. Gaver, T. Dunne, and E. Pacenti. Design: cultural probes. *ACM Interactions*, 6(1):21–29, 1999.

[6] K. Loer and M.D. Harrison. Analysing user confusion in context aware mobile applications. In M.F. Constabile and F. Paternò, editors, *INTERACT 2005*, number 3585, pages 184–197. Springer Lecture Notes in Computer Science, 2005.

[7] M.B. Rosson and J.M. Carroll. *Usability Engineering: scenario-based development of human computer interaction.* Morgan Kaufman, 2002.

Addressing Challenges of Stakeholder Conflict in the Development of Homecare Systems

Marilyn Rose McGee & Phil Gray

Department of Computing Science, University of Glasgow, UK,

1 Introduction

In this position paper we identify particular types of conflict that can arise in home care systems and consider ways in which system development methods and tools can address the satisfactory resolution of such conflict. We conclude by presenting our proposed future work.

2 Home Care Systems and Their Stakeholders

We define home care as a potentially linked set of services of either social care, health care, or both, that provide, or support the provision, of care in the home. Our focus in this paper in on technologically supported home care, in particular those that involve specialised computer systems. Such home care support can range from simple stand-alone electro-mechanical alarms installed in a person's home, perhaps to indicate a bath overflowing or a door left ajar, to systems integrated into the home's physical infrastructure [6,7] that monitor patient state, perform sophisticated analyses, deliver customised information to patients and clinicians and support communication among them.

We distinguish between

- the social and professional aspects of home care, including the people being cared for, the carers, and any external stakeholders playing a role in the care, which we call the *Network of Home Care,* and
- the technology used to support and realise the activities of the network of care, providing the means to collect, distribute, analyse and manage care related information. Such technology typically includes sensors, devices, displays, data, and networks, and computing infrastructures. Together we call this the *Home Care System.*

The key features of home care systems, from the point of view of this paper, are the following:

- Sensors provide data about the status of the cared person
- Home care can be multi-user and often collaborative
- Home care can be distributed
- Homecare System Interaction can be multimodal

Given the multi-user, multimodal, potentially collaborative and distributed nature of Home Care Systems, it is likely that the software and system solutions will produce conflicts and challenges that ubiquitous research much address.

3 Key Issues and Conflicts

3.1 Sources of Conflict

Conflicts might arise if the user(s) of the system misinterpret (a) other user(s) intentions or interactions and/or (b) the systems intentions or interactions. In order for home care systems to minimize the damage these conflicts can potentially cause, they have to be identified and described in such a way that their structure and characteristics are revealed with respect to potential resolution. What follows is not intended as a comprehensive and complete analytic model of such conflict, but merely an initial attempt to examine some examples, to illustrate their likely structure and variety.

- Shared Interaction Spaces
- Multiple care conditions
- Service quality versus user experience
- Control and use of data
- Accountability
- Volatility of behaviour and belief
- Consequences of conflict
- System failure
- Makes it hard to provide autonomic configuration
- Poor Usability

4 Conflict Identification, Negotiation and Resolution

Stakeholder conflict is a potential threat to the realisation of effective and usable home care systems. Solutions involve improving the identification, description and resolution of these conflicts. In this section, we present some initial ideas about how this might be accomplished. These potential solutions, or partial solutions, to stakeholder conflict, include technological solutions, socially or clinically negotiated, or implemented at a system design level, or some combination of these.

4.1 Technological

- Modify sensing or interaction technologies.
- Enhance the network policy languages for networks being built for homecare systems
- Develop configuration/monitoring tools that are based on these patterns of care and system models

4.2 Social and Clinical

Social and Clinical solutions can be derived from some combination of the other solutions. Where appropriate, multiple users and stakeholders are invited to feed into either or both of participatory design of the home care system and the ongoing configuration and evolution of the home care system.

4.3 System design-oriented Solutions

- Participatory design of the home care system
- Develop or augment activity, requirements and system models to enable conflicts to be identified and dealt with effectively
- Languages and prototyping tools to support system models.
- Identify and categorise patterns of care at home within these networks and ultimately develop a pattern language to support this and enable future home health care systems to be built successfully

4.4 Configuration Oriented Solutions

Instead of trying to resolve the conflicts at design time, they might be addressed by enabling the system to be configured appropriately at run-time. Given our earlier observations about the difficulty of identifying conflicts before users have experience the system, this approach is perhaps the most important but also one of the most demanding in terms of changes to home care system development. The challenge here is to make it possible to change the functionality and the interactive appearance and behaviour of the system, more or less fundamentally, at an acceptable cost to the user(s) and/or stakeholder(s).

Furthermore, while personalisation seems a key requirement for configuration oriented solutions, it can also exacerbate the problem if handled incorrectly [2]. If one stakeholder is allowed to personalise the system for themselves this may create a conflict with another. Thus personalisation has be implemented in such a way that conflicts are notified during personalisation, if possible and/or personalisation is performed in a collaborative way, enabling all relevant stakeholders to contribute to the final configuration decision.

We believe this to be a key research challenge for home care system software, requiring a solution that exploits the notion of dynamically reconfigurable self-describing components in a framework capable of supporting structural evolution and incorporating sharable components for editing and monitoring system status [e.g. 5].

5 References

[1] Barkhaus, L., & Dey, A.K. (2003). Is Context Aware Computing Taking Control Away From The User? Three Levels of Interactivity Examined. *In Proceedings of Ubicomp 2003*, 159-166.

[2] Bellotti,V., & Edwards, K. (2001). Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human-Computer Interaction*, 16, 193-212.

[3] Dey, A.K. (2001). Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1), 4-7.

[4] Dey, A.K., Mankoff, J., Abowd, G.D. & Carter, S. (2002). Distributed Mediation of Ambiguous Context in Aware Environments. *UIST 2002*, Paris, France.

[5] Dourish, P. (2004). What We Talk About When We Talk About Context. *Personal and Ubiquitous Computing*, 8, 19-30.

[6] Dowdall, A. & Perry, M. (2001). The Millenium Home: Domestic Technology to Support Independent-Living Older People. *In Proceedings of the 1st Equator IRC Workshop* 1-15; 2001, 13-14 Sept, Nottingham, UK.

[7] Mozer, M. (1998). The Neural Network House. *In Proceedings of AAAI Symposium on Intelligent Environments*, 110-114.

SEUC 2006 Workshop Programme

Session 6:
Engineering for Humans II

‘Palpability’ as an Architectural Quality

Klaus Marius Hansen
Department of Computer Science, University of Aarhus
Åbogade 34, 8200 Århus N, Denmark
Email: klaus.m.hansen@daimi.au.dk

Abstract

Arguably, a central challenge in engineering pervasive computing is that pervasive systems are pushing technical as well as use frontiers. Pervasive technology is providing new means for users to work and play often enabling profound changes in the practice of these users. On the other hand, pervasive technology is only successful to the extents that it is actually used and usable.

In this position paper, we postulate that these challenges are best attacked by pervasively applying use concepts as well as technical concepts at all levels of pervasive computing systems; from platform to middleware to application. In particular, we introduce the quality of ‘palpability’ as a quality-in-use as well as an architectural quality that needs to be addressed in a pervasive system.

1 Palpable Computing and Palpability

Pervasive computing[1] has always been a good source for interesting and relevant technical challenges in creating software systems and architectures (e.g., [6]). This stems from a complex interplay of requirements from particular applications and particular use and from general properties of these kinds of computing systems such as resource constraints, use of wireless connectivity, and mobility of devices and users. Taking pervasive computing as its outset,

Palpable computing systems are pervasive computing systems that are “graspable” both mentally and physically[2]. Palpable computing subsumes pervasive computing in that goals such as autonomy, invisibility, construction, and dynamism are complemented with user control and deference, visibility, de-construction, and support for stability [2]. Among the rationales for building and studying such systems are to be able to understand complex, pervasive systems, to be able to (de-)construct them and to handle partial failures of such systems. The PalCom IST project that explores palpable computing, has as its major goals to 1) define an open architecture for palpable computing, and 2) to create a conceptual framework for palpable computing.

Many of these concepts can be interpreted in a use as well as a technical sense, and are, if interpreted in a technical sense, congruent with good (object-oriented) software engineering practices. *Invisibility* of internals of objects is usually supported by information hiding and considered a major technique in managing dependencies in software systems. *Construction* (or composition) is the raison d’être of component-based

[1]In the context of this position paper, we will regard the terms ’ubiquituous computing’ [5], ’pervasive computing’ [3], and ’ambient intelligence’ [1] as equivalent

[2]`http://www.ist-palcom.org/`

architectures in which applications ideally may be composed from available software components.

On the other hand, some of the complementary concepts in the challenges pairs give rise to interesting issues in languages, middleware, and software architecture *Visibility*, e.g., may be in conflict with information hiding so controlled ways of "opening up" software systems are needed. In particular, if exceptions arise in the use of palpable computing systems, visibility of what has gone wrong and possibly why becomes important. Actually, in a dynamic pervasive computing world, failure cannot really be seen as exceptional. And *de-construction/de-composition* – in particular when the de-construction is not an exact inverse of a previous construction – emerges as a major and radical new issue.

2 Palpability as an Architectural Quality

In short, the goal of palpable computing is to support 'palpability' as a quality of use of pervasive computing systems built on top of the PalCom open architecture. Now the issue is how this use quality translates into product/architectural qualities of the systems and platform being constructed? As alluded to above, the characteristics of palpability have parallels at use and at platform level.

Using the "visibility/invisibility" characteristic for a specific example, invisibility on the use level may be a question of being able to access the Internet on one's laptop through a combination of Bluetooth radio (on the laptop and on a mobile phone) and GSM radio (on a mobile phone) without noticing that it is actually the mobile phone that brokers the Internet connection. Now, if this connection fails for some reason (e.g., unavailability or lack of performance), there is a need for being able to inspect the state/situation of devices, i.e., there is a need for visibility.

To realize this inspection, devices (and thus software) needs to fundamentally allow for this inspection. Ideally, we will want to know what has failed, for what reasons, what works correctly, and with which properties, etc., so that we, e.g., may choose a reasonably priced WiFi connection and use that for Internet access. Now, this example is a candidate for autonomous behavior of the pervasive system based on descriptions of the available resources and a description of the users desired trade-offs between speed and price. An example of where user control and deference would be preferable – also in a failure situation – would be a use scenario where palpable devices are used at an accident site and where the device to fail is a sensor connected to an injured person.

Again, this is an example in which the same characteristics of palpable computing needs to be supported at both use and platform level. A further example would be dynamism/stability in which the concept of ´assemblies' are explored at both levels [4]. Indeed, one may make similar arguments for most aspects of palpability leading to a speculative conclusion that palpability is a (central) concept in pervasive computing needing support both at use and architectural levels. In particular, palpability needs to be treated as an architectural quality of (palpable) systems and thus needs to be designed for, analyzed for, and constructed for. As such, palpability is representative of the more fundamental challenge of connecting design to use, here in the context of systems where technical and use innovation is radical.

References

[1] E. Aarts, R. Harwig, and M. Schuurmans. Ambient intelligence. In B. Denning, editor, *The Invisible Future*, pages 235–250. McGraw-Hill, 2001.

[2] P. Andersen, J. E. Bardram, H. B. Christensen, A. V. Corry, D. Greenwood, K. M. Hansen, and R. Schmid. An open architecture for palpable computing. In *ECOOP 2005 Object Technology for Ambient Intelligence Workshop, Glasgow, U.K.*, July 2005.

[3] G. F. Hoffnagle. Preface to the issue on pervasive computing. *IBM Systems Journal*, 38(4):502–503, 1999.

[4] M. Ingstrup and K. M. Hansen. Palpable assemblies: Dynamic service composition for ubiquitous computing. In *Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering*, 2005.

[5] M. Weiser. The computer for the 21st century. *Scientific American*, pages 94–110, July 1991.

[6] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

Human-Computer Interaction in Ubiquitous Computing Environments

David Benyon
The HCI group, School of Computing,
Napier University. Edinburgh

Human-Computer Interaction (HCI) has to respond to the demands of the new era of ubiquitous, ambient and pervasive computing. It has to deal with the challenges of large scale wireless sensor networks (Romer and Matten, 2004) and context aware applications (Lieberman and Selker, 2000). We will refer to these new and emerging computational environments as 'information spaces'. These spaces mix the real and the digital and are physically distributed across devices. They require a radical re-think of HCI because traditionally HCI has been dominated by a view that foregrounds ***a*** person interacting with ***a*** computer. Norman's classic conceptualisation of cognitive engineering and his theory of action (Norman, 1986) is an early example, but even most of the theories in Carroll (2003) focus on one person and one computer. By contrast information spaces are characterised by many people and many (possibly thousands) of devices.

Lieberman and Selker (2000) argue that there is an abstraction-context trade-off that is currently biased in favour of abstraction. Software engineering wants systems designed for re-use and for ease of maintenance but this means that the 'black box' is pitched at the wrong level. The benefits of higher levels of abstraction are outweighed by the lack of context sensitivity. Lieberman and Selker offer three criticisms of the simple input-processing-output model of HCI. User input is expensive and slow and could be much more efficient and enjoyable if the system took more account of what it knows about the user's context. Second, explicit outputs are not always appropriate or desirable. Here they point to ambient interaction that arises from work on tangible user interfaces (Fitzmaurice, et al., 1995) and other more subtle forms of output such as audio, gesture and bodily movement. Finally, the input-output feedback loop is not sequential as the traditional software model of HCI would suggest. Modern interfaces need to deal with multiple, multimodal and concurrent interactions. A single goal of a person may be to 'plan a trip', but this involves moving between several applications, and often having to re-enter information in different applications. This arises because the applications do not share the context.

An information space is some combination of things that is used by someone to provide information (Benyon, Turner and Turner, 2005). We distinguish three kinds of entity that make up information space. Agents are goal-directed. People are agents and we are increasingly developing artificial agents. Information Artefacts are objects that organise and present information. They have both a conceptual side (the information content in digital form) and a perceptual side (the interface in physical form). Information space also contains devices that allow data to be transmitted and stored. Information space is, thus, characterised by a mixture of physical and digital objects that are networked together, can sense their environment, process that information, operate and communicate with one another. Some of these networks will be autonomic; configuring themselves and communicating autonomously.

The work builds on our previous work on information spaces which has led to the notion that it is useful to see HCI as navigation of information space (Benyon and Hook, 1997; Benyon, 1998; Benyon 2001; Benyon, 2005). In this characterisation of HCI people are seen as moving through the network of agents, devices and information artefacts that make up the information space in much the same way as people move through the built environment (Benyon and Wilmes, 2003; Benyon, 2006). Information is distributed through the agents and information artefacts and people interact with this information space by storing, retrieving and transforming information. However, rarely does the information space does map 1:1 onto the activities. People have to move through the network to accomplish an activity.

This notion that information is distributed in this way has resonances with a number of other researchers such as the resources model (Wright, Fields and Harrison (2000), distributed cognition (Perry, 2003) and situated cognition (Lave, 1988). There are also similarities with Pirolli's 'information scent' where people are seen as 'informavores'; utilising our evolved food-foraging mechanisms for information gathering (Pirolli, 2003).

In information spaces we are not designing for tasks and goals, but designing spaces for interaction (Winograd, 1997). Designers can create information space sketches, showing the devices, agents and information artefacts involved in some activities. These can be more formally described using techniques such as ERMIA (Green and Benyon, 1996).

The concept of information space also means that we need to augment the traditional model of user interface architecture where the Model View Controller (MVC), or PAC (presentation, abstraction, control) model (Coutaz, 1987) has already been rendered lacking through work on tangible user interfaces (Ishii and Ullmer, 1997). They argue that the PAC model needs a physical part in addition to the PAC. However, Ishii still focuses just on the *interface*. With the notion of navigation in information space, the focus moves to the whole interaction; across devices and across time. In a recent review Dix (2003) laments that although there have been workshops addressing the challenges to these traditional HCI models, little real progress has been made.

In conclusion, consideration of the characteristics of information spaces, by which we intend to include pervasive computing environments, ubiquitous computing environments, context aware environments, ambient intelligence and wireless sensor networks, leads to a fundamental re-think of HCI. In particular we need software architectures that allow the separation of the description of the interaction and the device. We therefore require an ontology and taxonomy suitable for interaction design in these distributed, multimedia, multimodal environments and suitable theory, tools and methods to design for human interaction.

References

Benyon, D. R. (1998) Cognitive Ergonomics as Navigation in Information Space *Ergonomics* 41 (2) Feb. 153 – 156

Benyon, D. R. (2005) Information Space. In Ghaoul, C. (ed.) *The Encyclopedia of Human-Computer Interaction*

Benyon, D. R. (2006) Information Architecture and Navigation Design. In *Human Computer Interaction Research in Web Design and Evaluation*, Zaphyris, P. and Surinam, k. (eds.)

Benyon, D. R. and Wilmes, B. (2003) The Application of Urban Design Principles to Navigation of Web Sites. In *Proceedings of HCI 2003*

Benyon, D., and Höök, K. (1997) Navigation in Information Spaces: supporting the individual, In *Human-Computer Interaction: INTERACT'97*, S. Howard, J. Hammond & G. Lindgaard (editors), pp. 39 - 46, Chapman & Hall, July

Carroll, J. (ed.) (2003) *HCI Models, Theories and Frameworks*. Morgan Kaufman, Boston, MA

Coutaz, J. (1987) PAC, an object model for dialogue design. In Bullinger, H-J and Shackel, B. (eds.) Proceedings of Inteact 87. North Holland, Amsterdam.

Dix, A. (2003) Upside-Down As and Algorithms- Computational Formalisms and Theory. In Carroll, J. (ed.) HCI Models, Theories and Frameworks. Morgan Kaufman, Boston, MA

Fitzmaurice, G. W., Ishii, H., Buxton, W. (1995). Bricks: Laying the Foundations for Graspable User Interfaces. Published in the Proceedings of CHI 1995, May 7-11, ACM Press.

Green, T. R. G. and Benyon, D. R. (1996) The skull beneath the skin; Entity-relationship modelling of Information Artefacts. *International Journal of Human-Computer Studies* 44(6) 801-828

Ishii, H. and Ullmer, B. (1997) Tangible bits: towards seamless interfaces between people, bits and atoms. Proceedings of CHI 97 Conference, Atlanta, GA ACM press, New York pp 234 – 241

Lieberman, H., Selker, T. (2000) Out of Context: Computer Systems that Learn About, and Adapt to, Context. In IBM Systems Journal, Vol 39, Nos 3&4, pp.617-631

Norman, D. (1983) Cognitive Engineering. In Norman, D. and Draper, S. *User Centred Systems Design*

Perry, M. (2003) Distributed Cognition. In Carroll, J. (ed.) HCI *Models, Theories and Frameworks*. Morgan Kaufman, Boston, MA

Pirolli, P. (2003) Exploring and Finding Information In Carroll, J. (ed.) *HCI Models, Theories and Frameworks*. Morgan Kaufman, Boston, MA

Romer, K. and Mattern, F. (2004). The Design Space of Wireless Sensor Networks. Appeared in IEEE Wireless Communications, Vol. 11, No. 6, pp. 54-61, December 2004.

Winograd, T. (ed.) (1996) *Bringing Design to Software.* ACM Press. New York

Wright, P., Fields, R. E. and Harrison M. (2000) Analyzing Human-Computer Interaction as distributed cognition: the resources model. *Human-Computer Interaction*, 15(1), 1 – 42

User-Centered Task Modeling for Context-Aware Systems

Tobias Klug
Telecooperation Group
Technical University of Darmstadt
Hochschulstrasse 10
Darmstadt, Germany
lastname@tk.informatik.tu-darmstadt.de

1. INTRODUCTION

Ubiquitous computing envisions computers as such an integral part of our environment that we cease to notice them as computers. One part of this vision are context aware applications that know about their user's context and are able to intelligently support him doing his work. However, actually achieving context awareness in computer systems is an extremely complex matter.

Compared to traditional software, the context awareness idea is relatively new. Therefore existing software engineering methodologies focus on WIMP (Windows, Icons, Mouse, Pointer) interfaces. For these methodologies it is sufficient to concentrate on the system itself, because there are few if any external influences. As a result these applications are always in control of what happens. With context aware systems this is different. They focus on the real world and the system is just an add-on. Therefore, the designer needs some understanding of what is going on in the real world that might influence the system. In other words s/he needs to understand the user's context. In most cases the relevant context can be defined as the sum of all user's tasks and everything that influences their execution. In the following we will refer to this as the user's work environment.

Understanding a user's work environment is one of the most important steps when designing a context aware application. Users need to be involved in this step to obtain authentic data on how users perform their work, what their environment looks like and what their needs are. User centered design [4] is a paradigm that fits naturally in this application area as it increases the chances of user acceptance.

An important step towards understanding a user's work environment is gathering knowledge about his task and goal structures. This step is already common practice in the HCI community. HCI practitioners use techniques of the family of task analysis and modeling to analyze work processes. The resulting models are used to communicate the user's needs to the design and development teams building the application.

At runtime the system also needs to be aware of the user's environment, because it wants to support him/her in reaching a goal. To achieve that, the system needs to anticipate this goal and to know what needs to be done to achieve it. But goals and the related task structures can be very complex which makes it hard if not outright impossible to infer a user's goal without additional knowledge. Therefore, the system needs a model of the relevant tasks and goals.

It is obvious that models of the user's work environment are of help or necessary during a number of development stages from the initial concept development phase to the runtime phase. The remainder of this paper shows why adapted task models are interesting candidates for the above mentioned purposes.

2. TASK MODELING

The introduction shows that task modeling can be a valuable tool for the development of context aware systems. However the requirements for a successful integration are difficult to meet. The model must be powerful enough to be useful at runtime and at the same time simple enough to be understood by non-technical stakeholders that use it during development.

As task modeling has its origins in HCI, most existing methods originate in this discipline. Depending on their purpose the models vary in their degree of formality. Rather informal models like Hierarchical Task Analysis (HTA) [1] are used for analysis and concept development. These models are easy to understand and can be used as communication artifacts between different stakeholders during the system design. Other models like GOMS [2] and CTT [3] are more formal. These are often used for performance and offline usability evaluations. They are machine readable, but their formalisms make them unusable for any work with non-experts in task modeling.

The latter models have also been used by the model based user interface development community to (semi-)automatically construct user interfaces based on task models. So far these models have mostly been used to develop WIMP and mobile applications. They only capture a user's tasks and hardly any information about the context they are executed in. Integrating context into these models has been attempted, but these approaches describe how the task model changes in a specific context rather than describing why and how a task is influenced by this context.

Both, usability and expressiveness have not yet been achieved in one model because existing models have all been developed with a single purpose in mind. Theoretically it is possible to achieve both in a single model, because user and system are dealing with different representations of a model.

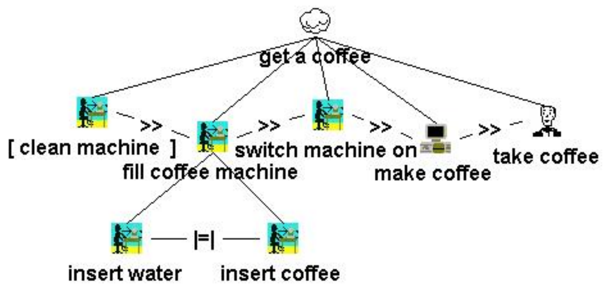


Figure 1: Coffee machine example in CTT notation.

A human is only interested in its graphical representation, whereas the system only understands the formal specification. The challenge is to find suitable representations for both parties and a mapping between these two.

3. PROPOSED MODEL

Task models and their associated context are hard to model because they form a very complex system with many relationships and constraints. Previous approaches have tried to integrate information about the task context into the same representation as the task model itself. The results are monolithic representations too bloated to be useful for any human, especially if end users are concerned. These kinds of models answer the question how tasks and context are interrelated, but not why. Let's consider the example of getting a coffee:

Alice wants to get a coffee from the coffee machine. But before she can get the coffee, she needs to fill in water and coffee. Then she starts the coffee machine and waits until the coffee is ready. If the machine was not clean before she might also need to clean the machine first.

Using CTT to model this process might look like in Figure 1. The process actually involves two interacting parties, Alice and the coffee machine. Although this model is supposed to show Alice's view on the coffee problem, it mixes her tasks with details inherent to the way the coffee machine works. This approach has several disadvantages. If the coffee machine were to be exchanged with a new device, the user process might have to be remodeled, although only the interaction depending on the coffee machine changed. Another problem is the readability of the representation. The only task Alice actually wants to perform is taking the coffee from the machine. All other tasks are only necessary because of the way the machine works.

In contrast splitting the model into one part for each entity involved in the process offers many advantages. Figure 2 shows such a model using a notation similar to UML. The upper part shows Alice's workflow from her point of view. The lower part shows a state machine modeling the behavior of the coffee machine. If the coffee machine needs external input to change state, this is indicated with the name of the task at the transition arrow.

The only link between the two models is the association of the "take coffee" task in Alice's workflow with the "take coffee" transition in the coffee machine's state machine. When Alice wants to perform this task, the coffee machine needs to be in the "coffee ready" state. If this is not the case, Alice needs to perform additional other steps like "insert water" to bring the machine into this state.

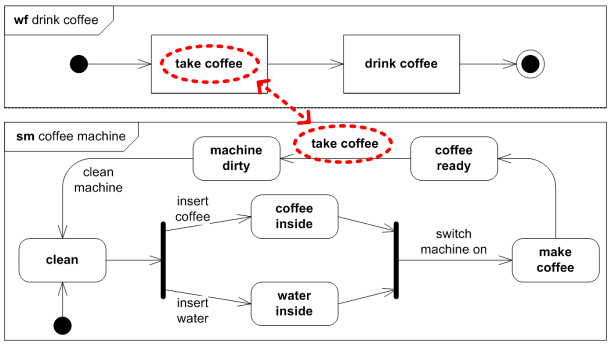


Figure 2: Coffee machine example with separate models for user and machine based on UML.

Using this approach logically separates the different entities within the process. Now, if a new coffee machine were to be purchased, only one part of the model would have to be changed and the link between the parts might have to be adapted. Additionally, the model is much more flexible, because a variety of coffee machine states is handled transparently. If the previous user forgot to clean the machine or already filled in water, The model shows what to do.

This example clearly shows that a lot of dependencies within one work environment can be deducted from its parts. As Figure 2 shows, the only manual dependency that was specified is the dependency between the "take coffee" task and the respective state of the coffee machine. All other dependencies can be derived from the combination of all parts. Together with the separation of concerns within the model the readability of the representation is improved while still maintaining the complexity for the computing system.

4. REFERENCES

[1] J. Annett. *Hierarchical Task Analysis*, chapter 2, pages 17–35. Lawrence Erlbaum, 2003.

[2] B. E. John and D. E. Kieras. The goms family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4):320–351, 1996.

[3] G. Mori, F. Paterno, and C. Santoro. CTTE: support for developing and analyzing task models for interactive system design. *IEEE Trans. Softw. Eng.*, 28(8):797–813, 2002.

[4] K. Vredenburg, S. Isensee, and C. Righi. *User-Centered Design: An Integrated Approach.* Prentice Hall, 2001.

Task-Based Development of User Interfaces for Ambient Intelligent Environment

Tim Clerckx and Karin Coninx
Hasselt University, Expertise Centre for Digital Media,
and transnationale Universiteit Limburg
Wetenschapspark 2
3590 Diepenbeek, Belgium
{tim.clerckx,karin.coninx}@uhasselt.be

Abstract

The designer of user interfaces for ubiquitous systems has to take into account the limited resources and changing contexts of use. In this paper we report on our proposal for a task-based development process for user interfaces that takes into account changes in the context of use and can react appropriately to the appearance and disappearance of services in their environment. Model-transformations and the generation and evaluation of prototypes are an important part of this development process and are supported by a custom-built tool.

1 Introduction

The shift to the increasing development of context-aware systems brings along the question whether traditional user interface development methodologies still result in systems that are sufficiently usable. Context-aware systems are more unpredictable than static systems because external context information can change both the system's and the user interface's state. This is why interactive context-aware systems require a better insight in what may happen with the user interface during the execution of the system in order to keep the system usable when a significant change of context occurs. We believe these problems can be reduced by taking two measurements: (1) restricting the possible influence of external context and available services on the user interface and (2) testing and evaluating context-changes causing changes in the user interface at early design stages until the actual deployment of the system.

Firstly, we were concentrating on how we could restrict the influence of context on the user interface by constraining influence in models. We developed a task-centred design approach to model the interaction of a context-aware system [2]. Furthermore, we developed a runtime architecture, supporting the early prototyping of the specified user interface models [3]. Afterwards, we generalized this runtime architecture to a generic one, in order to build prototypes of context-aware user interfaces during the whole development cycle of a context-aware system. In this approach we tried to reduce application specific code to be written in the user interface part of the system. In this position paper, we will give an overview of our development process, called DynaMo-AID.

2 The DynaMo-AID Process

In this section, we describe the DynaMo-AID development process supporting the development cycle for context-aware and service-aware user interfaces in providing methodologies and tool support in the design, prototyping and deployment phase of the production of context-aware and service-aware user interfaces.

The development process is prototype-driven consisting of several iterations over a prototype until a final iteration results in a deployable user interface. The process is presented in figure 1, inspired by the spiral model introduced by Boehm [1].

The process consists of four iterations which on their turn consist of four phases. Each iteration starts with the specification of an artefact, i.e. specification of models or code implementation (Artefact Construction). Subsequently a prototype is derived from this artefact and test runs are performed with this prototype (Prototyping). Afterwards this prototype is evaluated (Evaluation) in order to apply some changes to the basic artefact of the current iteration (Artefact Reconsideration). The updated artefacts are brought along to the next iteration.

The first iteration starts with the specification of models defining a user interface. Firstly, abstract models are specified describing the interaction at a high level, such as user tasks (task model). Afterwards, tools assist the user

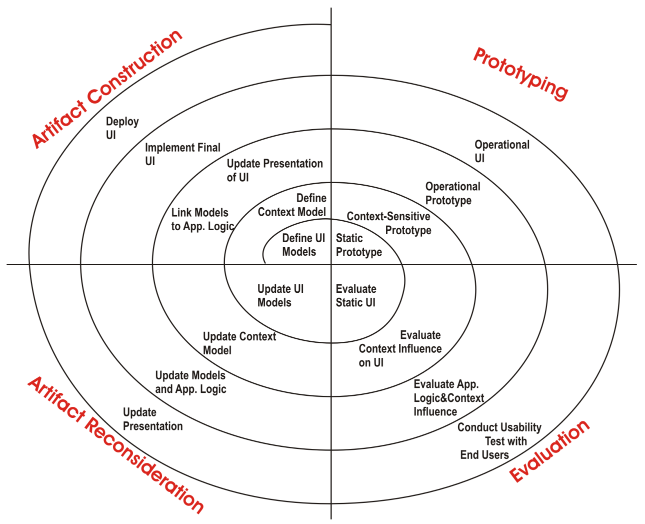


Figure 1. The User Interface Development Process

interface developer in transforming these abstract models into more concrete models (dialog and abstract presentation model) in order to make the models suitable for automated prototype generation. Next a static prototype is generated by the supporting design tool to perceive how the modelled user interface actually works. Finally, the prototype is evaluated and possible changes to the modelled UI can be carried out.

The next iteration is meant to introduce context-awareness in the user interface. Because context is only relevant when it has a direct influence on the user's task [4], we chose to attach the context model to a task model. Next a new prototype is automatically generated by the design tool. Afterwards the prototype is evaluated by the user interface developer and possible changes to the context and user interface models are allowed until the developer is satisfied with the resulting prototype.

In the third iteration the user interface models are linked to the functional core of the system. Furthermore the presentation of the user interface is altered in order to present the data provided by the functional core of the system. Next a new prototype can be rendered. This prototype is now operational and can be used to test the user interface on top of a working functional core and influenced by external context information. The prototype is evaluated, and changes can be applied.

The final iteration includes the actual implementation of the final presentation layer of the user interface. This layer can be very thin due to the architecture's modular characteristic and because the specified models are interpreted and used to support the communication between the different parts of the system. In this way, more attention can be paid to the design of the presentation of the user interface in this final iteration. The resulting operational user interface can then be tested by means of a usability test with end users in order to make final adjustments to the presentation layer. Afterwards the user interface is ready for deployment.

3 Concluding Remarks

Traditional user interface development techniques did not consider external context influence. This is why we have constructed a new, prototype-based, tool supported development process in order to model and test context influence on the user interface of the system. We believe the evaluation of prototypes throughout the whole development process is necessary considering the development of ubiquitous systems to create usable end user systems. In our work, we clearly focused on the development of context-aware and services-aware user interfaces which play an important role in ubiquitous systems.

Acknowledgements

The authors would like to thank Chris Vandervelpen and Kris Luyten for their contributions to the work described in this paper. Part of the research at EDM is funded by EFRD, the Flemish Government and IBBT. The CoDAMoS project IWT 030320 is directly funded by the IWT (Flemish subsidy organization).

References

[1] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[2] Tim Clerckx, Kris Luyten, and Karin Coninx. Dynamo-aid: A design process and a runtime architecture for dynamic model-based user interface development. In *Engineering for Human-Computer Interaction/DSV-IS*, volume 3425 of *Lecture Notes in Computer Science*, pages 77–95. Springer, 2004.

[3] Tim Clerckx, Kris Luyten, and Karin Coninx. Designing interactive systems in context: From prototype to deployment. In *People and Computers XIX - The Bigger Picture, Proceedings of HCI 2005: The 19th British HCI Group Annual Conference, September 5-9 2005, Napier University, Edinburgh, UK*, pages 85–100. Springer, 2005.

[4] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

SEUC 2006 Workshop Programme

Session 7: Platforms

Domino: Trust Me I’m An Expert

Malcolm Hall, Marek Bell, Matthew Chalmers
University of Glasgow

In ubicomp, the use of computers expands beyond work activities focused on pre-planned tasks into leisure and domestic life. People’s activities, contexts and preferences are varied and dynamic, and so system adaptation and evolution are especially important. It is difficult for the designer to foresee all possible functions and modules; and their transitions, combinations and uses. Instead of relying on the developer’s foresight, incremental adaptation and ongoing evolution under the control of the users may be more appropriate.

In Domino, the software engineer specifies basic classes and dependencies, but then the system changes its structure on the basis of the patterns of users’ activity. In this way, the architecture actively supports incremental adaptation and ongoing evolution of ubicomp systems. It supports each user in their discovery of new software modules through a context-specific collaborative filtering algorithm, and it integrates and interconnects new modules by analysing data on past use. Domino allows software modules to be automatically recommended, integrated and run, with user control over adaptation maintained through acceptance of recommendations rather than through manual search, choice and interconnection. One way of looking at Domino is to see it as a means of broadening access to and lowering the skill threshold needed for users’ adaptation of the system. For software engineers, it reduces or avoids the need to ‘see into the future’, i.e. to specify all of the contexts and permutations of use in advance. More detail about Domino can be found in a forthcoming paper [1].

The current version of Domino runs on both desktop computers and mobile devices that support WiFi and run Windows, Windows Mobile or PocketPC. Each instance of the Domino system consists of three distinct parts: handling communication with peers; monitoring, logging and recommending module use; and dynamically installing, loading and executing new modules. We refer to the items that Domino exchanges with peers and dynamically loads and installs as modules. A module consists of a group of .NET classes that are stored in a DLL (Dynamic Link Library) that provides a convenient package for transporting the module from one system to another. Each Domino system continually monitors and logs what combination of modules it is running. When one Domino system discovers another, the two begin exchanging logs of usage history. This exchange allows each system to compare its history with those of others, in order to create recommendations about which new modules a user may be interested in. Recommended modules are then transferred in DLL format between the systems. Recommendations accepted by the user are dynamically installed and executed by Domino. This constant discovery and installation of new modules at runtime allows a Domino system to adapt and grow continually around a user’s usage habits. The transfer of history data and modules when Domino clients meet leads to controlled diffusion inspired by the epidemic algorithms of Demers et al. [2]. Popular modules are quickly spread throughout the community, while modules that fulfil more specific needs spread more slowly but are likely eventually to locate a receptive audience because of history-based context matching and the use of ‘wanted lists’ to find required modules.

Security is a serious problem for any system that uses mobile code that moves between different devices, with threats such as viruses and spyware. Current solutions, such as signing and sandboxes, are viable but can be computationally expensive or require a trusted third party. Instead, while we are researching the use of epidemic algorithms to also spread information about malicious modules, we are concentrating on implementing Domino in game systems. While this does not avoid problems of viruses and malware (since ‘bad’ modules could destroy a user’s game, or be used as a way of cheating) it does provide an environment for experimenting with module recommendation and the broader security issues, avoiding potential damage to users’ devices.

Our work is also influenced by *Treasure* [3], which was a mobile game used to explore the exposure of system infrastructure in a ‘seamful’ way, so that users might appropriate variations in the infrastructure. To test the Domino architecture, we developed a mobile strategy game, *Castles*, that selectively exposes software structure to users, so that they might be aware of and appropriate software modules.
The majority of the Castles game is played in a solo building mode, in which the player chooses which buildings to construct and how many resources to use for each one. Each type of building is a Domino module. The goal of this stage is for the player to create a building infrastructure that efficiently constructs and maintains the player’s army units. In order to mimic the way that plug-ins and components for many software systems continually appear over time, new modules are introduced throughout the game, as upgrades and extensions that spread among players while they interact with each other. When two players’ devices are within wireless range, one may choose to attack another.

Behind the scenes, Domino also initiates its history-sharing and module-sharing processes. When a battle commences, both players select from their army the troops to enter into battle. Players receive updates as the battle proceeds, and at any time can choose to retreat, i.e. concede defeat. At the same time, players can talk about the game, or the modules they have recently collected, or modules they have used and either found useful or discarded. With such a high number of buildings, adapters and units, there is significant variation in the types of society (module configurations) that a player may create. Selecting which buildings to construct next or where to apply building adapters can be a confusing or daunting task. However, Domino helps by finding out about new modules as they become available, recommending which modules to create next, and loading and integrating new modules that the player accepts. When new buildings and units are available to be run but not yet instantiated, we notify the user of the new additions by highlighting them in the menu of available buildings. Domino does not fully automate the process. It presents modules in a way that lets the user see them as he or she plays, find out something of their past use, and show this information to others when meeting and talking with other players. Overall, Domino complements the conversation and discussion among players about new and interesting modules, and eases the introduction of new modules into each individual system and into the community.

We have run pilot tests and offer some initial evidence from the system's use. We set up the game so that four players sat in rooms distant from one another, and out of wireless network range. We periodically moved the players between rooms, so that they passed by each other, and met up in pairs. This meant that users spent most of the time alone but periodically met up to start battles and to talk about the game and its modules, much as they might if they were walking with their phones during a normal day. Each player started with the same core set of 54 modules, plus 5 items that were unique to him or her. For example, amongst the additional items given to one player was the catapult factory. As anticipated, when players met for battle, their Domino systems exchanged usage information and transferred modules between phones so as to be able to satisfy recommendations. Thus, the catapult factory and catapult unit began with one player, but were transferred, installed and run by two of the three other players during the game. Several players who had been performing poorly because of, for instance, a combination of buildings that was not efficient for constructing large armies, felt more confident and seemed to improve their strategies after encountering other players. They started constructing more useful buildings by following the recommendations. In each of these cases, this did not appear to stem from players' conversation, but directly from the information provided by the system.

Overall, our initial experience is promising. Domino's epidemic style of propagation of modules seems to be well suited to mobile applications where users may potentially encounter others away from high-bandwidth and freely (or cheaply) accessible networks. Domino can quickly and automatically exchange log data and modules. We are preparing for a larger user trial involving non-computer scientists in a less controlled environment than the one used for our pilot. We have begun to instrument the code so as to create detailed logs of GUI activity and module handling, to feed into tools for analysis and visualisation of patterns of use.

To summarise, the Domino architecture demonstrates dynamic adaptation to support users' needs, interests and activities. Domino identifies relationships between code modules beyond those specified in code by programmers prior to system deployment, such as classes, interfaces and dependencies between them. It uses those relationships, but it also takes advantage of the code modules' patterns of use and combination after they have been released into a user community. The Castles game demonstrated its components and mechanisms, exemplifying its means of peer-to-peer communication, recommendation based on patterns of module use, and adaptation based on both module dependencies and history data. The openness and dynamism of Domino's system architecture is applicable to a variety of systems, but is especially appropriate for mobile systems because of their variety and unpredictability of patterns of use, their frequent disconnection from fixed networks, and their relatively limited amount of memory. As people visit new places, obtain new information and interact with new peers, they are likely to be interested in new software, and novel methods of interacting with and combining modules. We see this as appropriate to software engineering for ubiquitous computing, where technology is seen not as standing apart from everyday life, but rather as deeply interwoven with and affected by everyday life. In the long run, we hope to better understand how patterns of user activity, often considered to be an issue more for HCI than software engineering, may be used to adapt and improve the fundamental structures and mechanisms of systems design.

References

1 Bell, M. et al, Domino: Exploring Mobile Collaborative Software Adaptation, *To Appear in Proc. Pervasive 2006*

2 Demers A. et al, Epidemic algorithms for replicated database maintenance, *Proc. 6th ACM Symposium on Principles of Distributed Computing* (PODC), 1987, 1-12

3 Barkhuus, L. et al., Picking Pockets on the Lawn: The Development of Tactics and Strategies in a Mobile Game *Proc. Ubicomp* 2005, 358-374.

Ubiquitous Computing: Adaptability Requirements Supported by Middleware Platforms

Nelly Bencomo, Pete Sawyer, Paul Grace, and Gordon Blair
Computing Department, Lancaster University, Lancaster, UK
{nelly, sawyer, gracep, gordon}@comp.lancs.ac.uk

Introduction

We are increasingly surrounded by computation-empowered devices that need to be aware of changes in their environment. They need to automatically adapt by taking actions based on environmental changes to ensure the continued satisfaction of user requirements. This complexity, of how to handle the requirements arising from different states of the environment, how to cope when the environment changes to ensure that ubiquitous systems [1] fulfill their intended purpose poses a major challenge for software engineering. One approach to handling this complexity at the architectural level is to augment middleware platforms with adaptive capabilities using reflection [2,3,4]. These augmented middleware platforms allow us to avoid building large monolithic systems that try to cover all the possible events, by providing components enabled with adaptation capabilities. These components can then be configured automatically and dynamically in response to changes in context.

Our current research is concerned with how adaptive middleware can be exploited by analysts handling requirements for ubiquitous systems. The problem here is to identify the requirements for adaptability from the user requirements, and map them onto the adaptive capabilities of the middleware in a way that is traceable and verifiable.

Adaptive middleware

Requirements for systems to dynamically adapt to changes in their environment introduce substantial complexity. In general, it is uneconomic and poor engineering practice to provide ad-hoc solutions to complex problems that share commonalities encountered within particular domains. Adaptive middleware solutions such as GridKit [5,6] or [7,8,9] mitigate this complexity in a structured way for application developers by providing adaptation support within domains of adaptation.

Gridkit is an OpenCOM-based middleware [10] solution that is structured using a lightweight run-time component model. This model enables appropriate policies to be defined and configured on a wide rage of device types, and facilitates runtime reconfiguration (as required for reasons of adaptation to dynamic environments). Gridkit supports an extensible set of middleware interaction types (e.g. RPC, publish-subscribe, streaming, etc.), and handles network heterogeneity by layering itself over virtual overlay networks which it manages and transparently instantiates on demand. GridKit exploits a set of frameworks, each responsible for different types of middleware behavior. GridKit therefore provides the basic capability for adaptation, while adaptability requirements are encoded as rules that are consulted at run-time when a change in the underlying environment is detected [5]. By the specification of different behaviors related to different adaptability requirements, the system can be adapted without changes to the application. Although GridKit is targeted at a particular domain of application, we believe that the same adaptability mechanism is capable of supporting adaptation required in other domains.

Adaptability requirements

While software architecture has provided a technology for explicitly separating concerns in adaptive applications, requirements engineering (RE) has yet to address the problem of how to deal with adaptability requirements. Our view, echoed by Berry et al. [11], is that adaptive systems introduce conceptual levels of requirements that are orthogonal to the accepted levels of, for example, user and system requirements. In particular, requirements for adaptation are concerned with understanding how a system may either make a transition between satisfying different user requirements depending on context, or continue to satisfy the same user requirements in the face of changing context. Hence, the adaptability requirements are intimately related to, and derived from, the user requirements. Yet, they represent requirements on the satisfaction of user requirements and therefore represent a kind of meta-requirement.

We propose that RE echoes the approach taken by software architecture and imposes a clear separation of concerns between application requirements and adaptability requirements. This should have the advantage of maintaining clear traceability links between user requirements at the application level and the adaptability requirements identified by analysis and refinement of the user requirements. However, this top-down approach is insufficient since the satisfiability of the derived adaptability requirements is contingent on the adaptive capabilities of the middleware.

Again, however, software architecture provides a model that can be exploited by RE. The GridKit framework, for example, provides sets of components that can be configured for different applications using policies. As noted by Keeney and Cahill [8]: “Policy specifications maintain a very clean separation of concerns between adaptations available, the decision process that determines when these adaptations are performed and the adaptation mechanism itself”. The policies used by GridKit are rules, expressed in XML, and can be mapped cleanly onto adaptability requirements provided the requirements are developed to the appropriate level of detail and constrained by the scope of GridKit’s domains of adaptation.

Open issues

In summary, therefore: traceability can be maintained by the derivation of *adaptability requirements* from user requirements; the requirements identified as adaptability requirements are refined and verified against the capabilities of the middleware using the semantics of the policy language used to configure the middleware component frameworks; and the verified adaptability requirements are finally encoded as policy rules while the remaining *application requirements* are implemented conventionally.

While this provides a conceptual partitioning of requirements into adaptability and application requirements, there is one outstanding problem for which a solution has not yet been identified. This is that ‘traditional’ analysis methods that are (e.g.) use-case driven or viewpoint-oriented, provide ways of partitioning the requirements that are poorly suited to identifying adaptability requirements. The need for adaptation may span several use cases, for example, yet may not easily emerge as a requirement common to the uses cases it spans. It is possible to treat adaptability as a soft goal in i* [12] but even this is problematic because adaptation is not necessarily closely related to user intentionality. Investigating this problem will form the next stage of our research.

References

1. Weiser M.: *The Computer for the 21st Century*. Scientific American, 265(3), pp. 94-104, September 1991
2. Maes, P.: *Concepts and Experiments in Computational Reflection*. Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987
3. Smith B.: *Reflection and Semantics in a Procedural Language.* PhD thesis, MIT Laboratory of Computer Science, 1982
4. Kon, F., Costa, F., Blair, G.S., Campbell, R.: *The Case for Reflective Middleware: Building middleware that is flexible, reconfigurable, and yet simple to use*. CACM Vol 45, No 6, 2002
5. Grace P., Coulson G., Blair G., Porter B.: *Addressing Network Heterogeneity in Pervasive Application Environments*. Proceedings of the 1st International Conference on Integrated Internet Ad-hoc and Sensor Networks (Intersense 2006), Nice, France, May 2006
6. Coulson G., Grace P., Blair G., Duce D., Cooper C., Sagar M.: *A Middleware Approach for Pervasive Grid Environments*. UK-UbiNet/ UK e-Science Programme Workshop on Ubiquitous Computing and e-Research, 22nd April 2005
7. Capra, L., W. Emmerich, W., C. Mascolo, C., *CARISMA: Context-Aware middleware System for Mobile Applications*. IEEE Transactions on Software Engineering, Vol 29, No 10, pp929-945, Nov 2003
8. Keeney J, Cahill V., *Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Environment.* Proc. Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03), Lake Como, Italy, 2003
9. Efstratiou C., *Coordinated Adaptation for Adaptive Context-aware Applications*. Ph.D. Thesis, Lancaster University, Computing Department, 2004
10. Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J.: *OpenCOM v2: A Component Model for Building Systems Software*, Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge, MA, ESA, Nov 2004
11. Berry D.M., Cheng B.H.C., Zhang J., *The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems*, Proc. 11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ’05), 2005, Porto, Portugal
12. Yu E., *Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering*. Proc. Third IEEE International Symposium on Requirements Engineering (RE’97), Annapolis, MD. USA, 1997

wasp: a platform for prototyping ubiquitous computing devices

Steve Hodges and Shahram Izadi, Microsoft Research Cambridge
Simon Han, UCLA Networked & Embedded Systems Laboratory
{shodges, shahrami}@microsoft.com; simonhan@gmail.com

20th April 2006

The importance of prototyping

Prototyping is a powerful way of assessing the value of ubiquitous computing applications, deciding if they warrant further development, and understanding how best to do this. Indeed, putting prototypes in the hands of 'real users' is increasingly important in assessing their potential impact and relevance. Prototypes can be developed to many different levels of sophistication, but typically early prototypes are quite basic, and as the concept is refined so too is the prototype. Experience shows that each successive level of refinement requires considerably more effort than the previous. Unfortunately, today's 'real users' have very high expectations of technology. This means that they increasingly expect even prototypes to be refined and robust, and without this they often find it hard to evaluate them fairly. Our experience shows that only when a prototype is sufficiently well developed do users see past its prototypical nature, and only then do the real insights about how it can be used become apparent. Of course, developing prototypes to this level of refinement is difficult and time-consuming. This especially applies to the development of the embedded hardware which is often integral to ubiquitous computing applications, due to the electronic and industrial design requirements that accompany software development.

wasp, a new platform for prototyping ubiquitous computing devices

We are currently developing a new embedded system development platform, called *wasp*, which facilitates the effective and efficient prototyping of both hardware and firmware for ubiquitous computing applications. *wasp* (the wireless actuator and sensor platform) is specifically designed to accelerate the development of reliable, compact, physically robust wireless prototypes with good battery lifetime. There are many other prototyping tools for ubiquitous computing hardware, but in the authors' experience, these do not meet all of the above criteria, and the resulting prototypes are typically unsuitable for full-scale user trials of the technology that they demonstrate.

From a hardware point of view, *wasp* is based around a series of small modules that can be connected together physically and electrically (via a high-speed daisy-chained serial bus) to form a particular complete embedded device. The 'base' module contains an ARM7 microcontroller[1] with a USB interface, real time clock, and power regulation (including a lithium-ion battery charger). Input, output and communications devices are incorporated via additional 'peripheral' modules – this helps manage the complexity of both the hardware and the firmware, because each peripheral to the main processor has a simple and well-defined interface. Example peripherals under development are a GSM/GPRS modem, Bluetooth, basic I/O for LEDs, servos, buzzers etc., GPS and a VGA camera.[2] If a new type of peripheral is required for a specific application, then a suitable module must be developed – but because the interface is well-defined this is a relatively simple, self-contained task.

In addition to hardware support, *wasp* also provides a powerful environment for development of embedded firmware. The basis of this is a lightweight event-based (i.e. co-operative) kernel called *wasp*-OS. In many ways, *wasp*-OS is similar to TinyOS, which has become very popular in the wireless sensor network community, although it supports a number of additional features and is written entirely in ANSI C. *wasp*-OS includes a tiered hardware abstraction layer which allows performance-sensitive applications direct access to the hardware, but also provides a reasonably high-level API to hardware such as timers and I^{2}C, SPI and UART interfaces. Note that *wasp*-OS does not directly support protocols such as TCP/IP or Bluetooth which keeps the kernel light-weight

[1] We are currently concentrating on the ARM7, but *wasp* could be based around any microcontroller.

[2] Each module is one of a number of different sizes, but they are designed to connect together physically in a space-efficient manner.

and simple. Instead, these are supported through the use of peripherals that themselves have processors embedded; for example the GPRS modem module contains a microcontroller that runs a TCP/IP stack.

A prototype ubiquitous computing device built around *wasp* will therefore consist of a number of hardware ‘peripheral’ modules connected to a ‘base’ processor module. The processor on the base module runs an embedded C application in conjunction with *wasp*-OS.

Development and debugging using *wasp*

wasp also provides significant firmware development and debugging support. Since both *wasp*-OS and the code for the application itself are written entirely in ANSI C, it is possible to compile them for x86 (using Visual Studio, for example), and run them under Windows (as a single Windows process). This makes it possible to leverage the range of powerful debugging capabilities available in a PC development environment for embedded application development. Of course, a completely different binary will eventually be deployed in the embedded hardware – so there are several aspects of operation that cannot be tested in this manner (such as real-time performance). But in the authors’ experience, a large number of errors in embedded system development are of a much simpler nature, and the ability to avoid a time-consuming compile-download-test cycle combined with virtually unlimited use of tools such as breakpoints[3] is incredibly powerful.

The other major issue when running a *wasp* application on the PC (rather than the embedded target) is that the peripheral modules will not be directly available. To overcome this, a different HAL must be used. The most straightforward approach is to use a HAL that re-directs peripheral access to a simulator for each peripheral. These simulators can run on the PC as separate Windows applications, or can be combined into a single Windows app. The simulation can have a UI (e.g. simulated LEDs that ‘light up’, or simulated push-buttons that can be clicked on), or can be completely embedded (e.g. a simulated flash memory). This approach allows embedded application development to proceed before any hardware is available. However, since many embedded peripheral devices communicate using RS232, I^{2}C or SPI, it is possible to connect them directly to a PC.[4] In this case, a modified HAL is used to access the real hardware peripherals, which enables a further step of validation in the debugging process.

Following migration of the application from a desktop to the embedded hardware, the *wasp* base module may itself be connected to a PC via USB. This results in a serial port connection to the *wasp* CPU. We plan to add a simple command-line interface to *wasp*-OS which will allow the real-time operation of the embedded target to be monitored and controlled very simply using this serial interface. It may be possible to extend this command-line interface to support simple scripting, which would make it easier for non-experts to control various aspects of operation. We are also investigating ways in which information may be communicated automatically between a *wasp* device and desktop applications such as Macromedia Flash.

Summary

In summary, *wasp* is a complete platform for prototyping ubiquitous computing devices effectively and efficiently. It supports a development process that naturally integrates hardware and firmware, and leverages powerful, established debugging tools and practices common to desktop application development. *wasp* is very-much work-in-progress, but we hope that over time it will prove valuable in a wide range of ubiquitous computing applications.

[3] Embedded microcontroller development environments often limit the number of simultaneous breakpoints.

[4] Many devices to convert USB to RS232, I^{2}C and SPI are readily commercially available.

SEUC 2006 Workshop Programme

Additional Papers

Development Tools for Mundo Smart Environments

Erwin Aitenbichler
Telecooperation Group
Darmstadt University of Technology
Hochschulstrasse 10
64289 Darmstadt, Germany
erwin@informatik.tu-darmstadt.de

1 Introduction

The *Mundo* project [2] at our group is concerned with general models and architectures for ubiquitous computing systems. The present paper describes the tools created in this project to support the development of applications for smart environments.

First, we give a brief description of the overall structure of such applications. The common software basis is formed by the communication middleware *MundoCore* [1]. It is based on a microkernel design, supports dynamic reconfiguration, and provides a common set of APIs for different programming languages (Java, C++, Python) on a wide range of different devices. The architectural model addresses the need for proper language bindings, different communication abstractions, peer-to-peer overlays, different transport protocols, different invocation protocols, and automatic peer discovery. Most importantly, MundoCore serves as an integration platform that allows to build systems out of heterogeneous *services*.

Applications consist of a set of common services and application-specific services. Common services for smart environments are offered, e.g., by our *Context Server* [3] and an application server [4]. The Context Server is responsible for transforming the readings received from sensors into information that is meaningful to applications. It builds on the notion of context widgets. To facilitate the development of applications for smart environments, we have created a number of tools in addition. In the following, we describe how these tools support the different phases of software development (Figure 1).

2 Modeling Phase

WorldView is a versatile tool with functions to support the modeling, implementation, and testing phases. In the modeling phase, WorldView is used to create a spatial model of the smart space. It supports 2D models as well as detailed geometric 3D models. In the map window, the application shows the floor layout and provides an overview of the available resources and their locations. Resources include tags and sensors of different location systems, wall displays, and smart doorplates. WorldView provides an easy way to define the regions of interest that should trigger spatial events for location-aware applications. The created maps can be uploaded to the Context Server and then be accessed by context widgets in order to derive higher-level context information.

3 Development Phase

A lot of research in ubiquitous computing is conducted around possible application scenarios that can be put to daily use. Many of these scenarios are quite simple and straightforward to implement. However, many of these applications never come to life, because the whole process from development to deployment is still very complex. To write a new application, the developer typically has to start her IDE, install all required libraries, write and test code and then deploy the application on a server. For that reason, we wanted to make this development process as easy and fast as possible. An aim was to enable a wide variety of people with some basic technical background, but not necessarily with knowledge of a programming language, to create and alter applications. Simple-structured applications can be directly developed with the provided tools, while more complex applications can be prototyped.

SYECAAD: The SYstem for Easy Context Aware Application Development (SYECAAD) [4] facilitates the rapid development of context-aware applications. Applications are built using a graphic-oriented block model. The basic building blocks are called *functional units*. Functional units have input and output pins and are interconnected to form *functional assemblies*.

Functional units come in three different flavors: sensors, operations, and actors. A sensor unit either receives data directly from a sensor or preprocessed data from the Context Server. Operations perform logic or arithmetic operations, implement dictionaries, render HTML pages, etc. On the output side, actors can control the smart environment or send feedback to users. Actors can publish MundoCore events or invoke methods. This way, an application can e.g., control smart power plugs, control data projectors, send emails, send SMSs, send instant messenger messages, or display information on electronic doorplates. An electronic doorplate is a 8-inch color TFT display with touchscreen that replaces an ordinary doorplate.

SYECAAD uses a client/server architecture. The server hosts applications. Clients connect to the server and permit to control, edit and test running applications. The *Application Logic Editor* in *WorldView* is such a client to edit applications. This system addresses all the development steps described above. The first step, namely setting up the development environment, is no longer necessary, because all the application logic is centrally stored on the application server. The development environment is a client application that connects to this server. In this way, an application can be loaded from the server, displayed, edited and deployed with the click of a button.

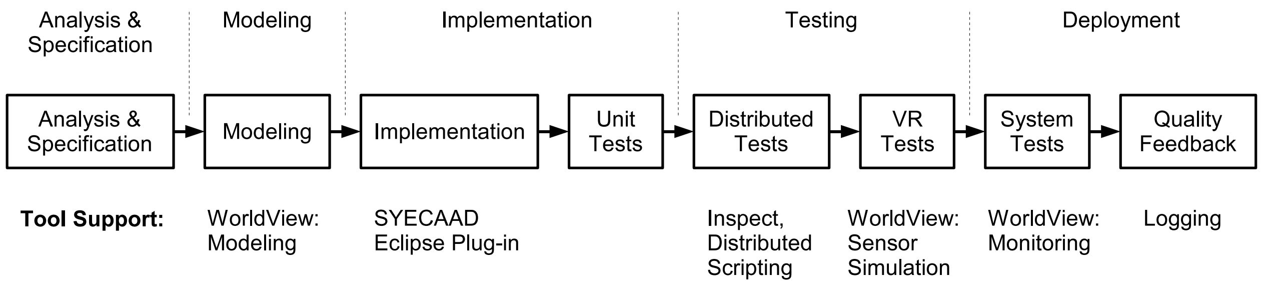


Figure 1: Tool support for different phases of software development (multiple iterations of steps possible)

Eclipse plug-in: If the standard sensor, operation, and actor blocks are not sufficient, the system can be extended by programming new blocks in Java. A plug-in for the Eclipse IDE supports the programmer with code templates and help documents.

4 Testing Phase

Implementations of abstract data types and smaller units of frameworks can be successfully tested with *unit tests*. However, to verify the correct behavior of a distributed system, it is also essential to run integration tests across multiple computers.

Distributed Tests: To conduct such tests, we implemented a *Distributed Script Interpreter*. Script server processes are started on multiple hosts in the network. The script servers and the master script interpreter also use MundoCore for communication. This enables them to automatically discover each other on startup. To run a test, the name of an XML script file is passed to the master interpreter. When the master interpreter encounters a `execRemote` instruction, it starts a new process on a remote script server and passes the text contained in the tag to the standard input of the remote process. The standard output of the remote process is passed back from the remote script server to the master interpreter and written to a log file. The shell return codes indicate if processes were successful running their sub-tests.

Inspect: The *MundoCore Inspect* tool can connect to an arbitrary remote node and manage the hosted services. The program allows to view the routing tables, list the imports and exports tables of message brokers, monitor the messages passed over a channel, view service interfaces, dynamically call remote methods with a generic client, view service configuration information and reconfigure services.

Sensor Simulation: To test applications, *WorldView* can be used to simulate certain tracking systems. In this case, the user can move around the symbols on the map and WorldView generates the same kind of events the tracking system would.

5 Deployment Phase

Monitoring: *WorldView* can also be used to inspect the running system by visualizing the events from certain event sources, like tracking systems. If a tag is physically moved around, the position of the corresponding symbol in the map view is updated in real-time.

Quality Feedback: The MundoCore middleware implements heap debugging, system resource tracking, deadlock detection, progress monitoring, and logging. Some bugs are not discovered until the system is tested with the real sensors. In this case, detailed logs provide important information for developers to fix the problem.

6 Summary

The development process of distributed, loosely-coupled, and context-aware applications raises the need for novel tools. Such tools to support the different phases of software development, i.e., modeling, implementation, testing, and deployment have been presented.

Bibliography

[1] Erwin Aitenbichler, Jussi Kangasharju, and Max Mühlhäuser. Experiences with MundoCore. In *Third IEEE Conference on Pervasive Computing and Communications (PerCom'05) Workshops*, pages 168–172. IEEE Computer Society, March 2005.

[2] Andreas Hartl, Erwin Aitenbichler, Gerhard Austaller, Andreas Heinemann, Tobias Limberger, Elmar Braun, and Max Mühlhäuser. Engineering Multimedia-Aware Personalized Ubiquitous Services. In *IEEE Fourth International Symposium on Multimedia Software Engineering (MSE'02)*, pages 344–351. IEEE Computer Society, December 2002.

[3] Marek Meyer. Context Server: Location Context Support for Ubiquitous Computing. Master's thesis, Darmstadt University of Technology, January 2005.

[4] Jean Schütz. SYECAAD: Ein System zur einfachen Erzeugung kontextsensitiver Applikationen. Master's thesis, Technische Universität Darmstadt, 2005.

Engineering Trust in Ubiquitous Computing

Sebastian Ries
Telecooperation Group
Technical University of Darmstadt
Hochschulstrasse 10
64289 Darmstadt, Germany
ries@tk.informatik.tu-darmstadt.de

1 Introduction

In Weiser's vision of ubiquitous computing (ubicomp), computers, as they are common today, vanish more and more [5]. Instead we will be surrounded by a multitude of smart items. The smart items are developed for different purposes, and therefore have different computational resources and communication capabilities. This increases the need for interaction with nearby devices. E.g., if it is not possible for an item to connect directly to the internet to get some information, it may receive this information form a nearby item, or connect to the internet via another item.

Furthermore, ubicomp applications need to adapt their configuration depending on the context information, like time, location, or infrastructure. E.g., the available communication partners can change with the location, and therefore some services are no longer available, but could be replaced by others.

We expect the ubicomp world to offer redundant services for all kinds of purposes. But those services may differ in quality, availability, privacy statements, and billing. The application developers will no longer be able to come up with a standard configuration which is perfect for everybody and we cannot expect the user to adapt the configuration of dozens of smart items perhaps several times a day. We need a dynamic concept which provides the user with a reasonable feeling of safety, which adapts the applications to the users preferences, and which takes advantage of the computational power available by interaction with other resources in the infrastructure.

In the following part of the paper we will show that trust is the appropriate foundation for interaction in ubicomp and how it should be implemented in applications. We believe that trust will become an important concept for applications in ubicomp environments and therefore has to be considered early in software engineering.

2 Concept of Trust

> That we get up at all in the morning is a sign of the trust we have in society and our environment. (Niklas Luhmann, 1979)

2.1 Social Trust

Trust is a well-known concept of everyday life, which simplifies complex processes. Many processes are enabled by trust and would not be operable without. Trust allows us to delegate tasks and decisions to an appropriate person, and trust facilitates efficient rating of information based on the experience with communication partners and, if applicable, on their reputation. Essential aspects of trust, besides personal experience and reputation, are the expected risk and benefit associated with the engagement of concern [4].

Trust has several features on which most researchers on this topic agree, and which are relevant if the concept is applied to ubicomp. Trust is subjective and therefore asymmetric. That is if Alice trusts Bob, we cannot make any conclusion about Bob's trust in Alice. Trust is context-dependent: It is obvious that Alice may fully trust Bob in driving her car, but not in signing contracts in her name. Trust is also dynamic. It can increase with good experiences and decrease with bad experiences, but it can also decrease if too much time has passed without any experience.

Furthermore, trust has different levels [3]. We think that it is not necessary to model trust continuous since people are better in assigning discrete categories [2], but trust cannot be modeled in a binary way.

Finally, there is the sensitive point of transitivity of trust. If Alice does not know Charly, but Alice trusts Bob in a certain context, and Bob trusts Charly in the same context, Bob can report the recommendation of Charly to Alice. She will decide what to do with this recommendation based on her trust in Bob's ability to recommend other opinions in this context and on the trust level assigned to Charly by Bob for this context. Since it seems unnatural to build arbitrary long chains, we point out that trust is not transitive in a mathematical sense.

2.2 Trust in ubicomp

In computer science there is much research ongoing on trust [1, 2]. Trust has been successfully applied to many areas, e.g. security, eCommerce, virtual communities, and the semantic web. Trust is a concept, which helps people to deal with aspects like uncertainty, limited resources and interaction with others, and this way increases the efficiency of many everyday process. Since these aspects are main issues of ubiquitous computing, it is a very promising approach to transfer the concept of trust to this area, in order to exploit the potentials of the ubicomp environment. It will not be sufficient to add some trust mechanism to ready-made ubicomp applications. The concept needs to be integrated early in software engineering.

2.3 Definition

There is much work about the concept of trust in different areas of computer science, but there is no coherent understanding of trust [2]. Although trust is a well-known concept, it is hard to define. In our definition of trust we point out the basic social foundations of trust:

Trust is the well-founded willingness for a potential risky engagement in the presence of uncertainty. Trust is based on experience, reputation as well as on the context of this engagement, including especially the expected risk and benefit.

The engagement will in general be the delegation of a task or of some information (more technically speaking, the delegation of a function or some data) from the delegator to the delegatee. For ubicomp, this means that trust is the appropriate basis for interaction between known and unknown devices described in the introduction.

3 Trust-aided computing

Our vision of trust-aided computing is about applications which explicitly model trust and use trust as basis to make decisions for risky engagements. Hence, it has direct impact on many kinds of interaction between ubicomp applications. Trust-aided applications are more than simple applications which get trusted by their users with growing experience. They model trust themselves. Therefore, they keep track of the devices they are surrounded by, they collect their experiences with those devices and information about the reputation of them. This way they can adapt to changes in the infrastructure and keep preferences for interaction with devices which are already trusted, or are reported to be trustworthy by other trusted devices. Trust-aided applications have to implement the following aspects.

3.1 Trust management

In trust management, we include the evaluation of the relevant context (e.g., time constraints, location, minimum level of expected quality of service), the evaluation of expected risk and benefit of the engagement, and the collection of opinions about the potential delegatees.

We do not want trust management to make a decision due to the credentials or certificates which a potential delegatee could present. The verification of a authentication of a certificate does neither imply that the delegator trusts the owner of the certificate, nor does it make a statement how much the delegator trusts in the binding between the identity of the one presenting the certificate and identity stated by the certificate. These parameters are not part of the trust management but of the trust model.

3.2 Trust model

The trust model aggregates the collected opinions and defines their representation. There are many different approaches to represent and aggregate opinions [2]. We identify two main requirements for opinion representation. At first, the representation has to be readable for man and machine in an easy way. This enforces a mathematical or logical model, which should be designed in a way that allows an intuitive representation for humans. Secondly, man and machine have to be able to update their own opinion according to latest experiences and preferences. From our point of view, trust depends on the overall number of experiences on single subject, as well as on the number or ratio of positive and negative experiences. Therefore, we think that trust cannot be expressed by a single value. An opinion should at least express our rating of the subject (positive, negative, or degrees inbetween) and the certainty of our rating, which should increase with the number of experiences.

Since experience is an essential issue of trust, the collection of experience and feedback from the user becomes an important aspect to be addressed by software engineering.

3.3 Decision making

The result of the decision making should obviously be the decision whether to trust or not, that is, whether to carry out the risky engagement or not. Since the user is not used to this kind of autonomous decision making (he does not yet have any trust in it, since he has no experience with it), the computed decision can be presented to the user with the possibility of manual overwrite. This seems to be reasonable, but will reduce the benefit of the automatization dramatically. Therefore, we need a way to reduce the risk of the user, without taking the benefit. The user should be able to define a set of expected results, and if the decision is part of this set, it will be carried out without any further user interaction.

4 Conclusion

Trust provides an excellent basis for delegation, which helps to cope with sparse resources and enables efficient evaluation of data presented by trustees. Trust is a concept which is able to adapt decisions to infrastructure changes, due to its dynamic nature. Furthermore, the building of trust chains seems to be the appropriate way to make interaction more reliable. In our future work, we will develop a robust trust model which can be applied to ubicomp applications and includes the social aspects of trust, especially the aspects of experience and risk. In the end, a framework should be developed which allows the explicit integration of trust in different kinds of software applications, as a first step towards trust-aided computing.

Bibliography

[1] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys and Tutorials*, 3(4), 2000.

[2] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. In *Decision Support Systems*, 2005.

[3] S. Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, 1994.

[4] M. Voss, A. Heinemann, and M. Mühlhäuser. A Privacy Preserving Reputation System for Mobile Information Dissemination Networks. In *Proc. SECURECOMM'05*, pages 171–181. IEEE, 2005.

[5] M. Weiser. The computer for the twenty-first century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999.

Enhancing Mobile Applications with Context Awareness

Jens H. Jahnke
Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria B.C. V9E2A7, Canada
jens@uvic.ca

1 Introduction

Context is an important concept in ubiquitous computing applications. Context-awareness has the potential to deliver the right information dialogues in the right situation, significantly minimizing the need for user interaction. Over the last decade, there have been many reports on CA system prototypes developed in order to explore the concept of context-awareness. While most of these systems were one-off proof-of-concepts solutions, recent research has focussed on finding systematic ways of constructing CA systems. These approaches include middleware frameworks [6], a predefined component building blocks [2], and dedicated programming languages [9] to be used for building CA applications.

Our approach is different. Rather than creating a technology to engineer CA applications "from scratch", we propose a way to enhance existing mobile applications with CA features. In other words, the development of the CA features is pushed down to some of the later stages (deployment) in the software development life-cycle. This approach follows a general software engineering approach often referred to as product-line engineering or product-line practice [5]. This approach increases re-usability of software products in deployment contexts with differing sets of contextual inputs. For example, consider a mobile application that provides employees of electricity providers with functionality to input the energy meter readings of their customers. The data to be captured includes the customer's name, address, and the current meter reading. A CA version of this application may be designed to utilize a localization service to automatically infer the customer address. However, if the CA application depends on the availability of this type of contextual information, it cannot easily be re-used in a rural area where the service is not available.

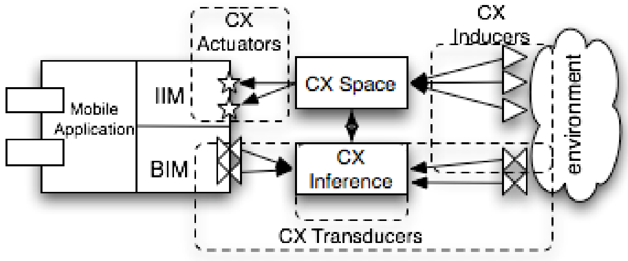
This simple example illustrates the importance of the principle of separation of concerns between the functional design of an application and its CA features. If this principle is not followed, i.e., if CA features are part of the functional design of an application, the application becomes dependent on the availability of certain contextual data provided in its deployment environment. While this kind of dependency may not be avoidable in all cases, it can be achieved in a large class of CA applications. We advocate that analysis, design and development of these applications should follow primarily conventional processes, and CA features are injected only late in the software life cycle, namely at deployment time, when the particular contextual information sources of the deployment environment are known. Another benefit of this approach is that it simplifies equipping existing applications with CA features.

2 Architecture of mobile applications with *Whiskers*

Separation of concerns has been a leading principle in the design of our framework "Whiskers". As a result, Whiskers applications are basically developed like conventional mobile applications and CA features merely represent an add-on that is added during deployment. This section describes the architecture of a Whiskers-based application.

The figure below shows a conceptual overview of this architecture. The component on the left-hand side symbolizes the mobile application. The figure particularly highlights two parts of the mobile application that interact with the Whiskers context manager, namely the Interface Information Model (IIM) and the Business Information Model (BIM). The IIM denotes the (run-time) data structure used to manage information at the user interface of the application. The BIM is the (typically persistent) data structure that contains all information maintained by the mobile application.

The introduction of the two concepts IIM and BIM indicates that we are targeting medium and thick client mobile applications with Whiskers, rather than thin-client, browser based clients. This is in contrast to our previous research [7], where the context management system is server-based. Server-side solutions are not suitable for a large class off mobile applications, which are not always "online" but rather synchronize data with servers at certain times.



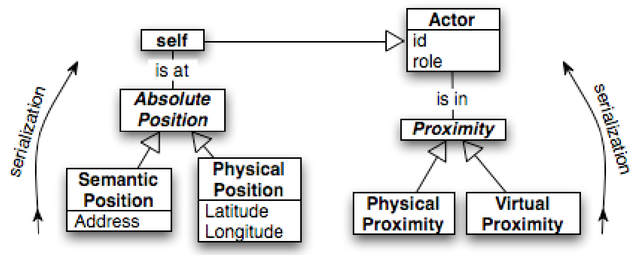
2.1 Context Actuators

The part of the Whiskers context management system that interacts with the mobile application in order to automatically set it to a particular context is called *Context Actuators* (CX Actuators). They can be realized as aspects [3] advising the nobile application to query the context fact base (*CX Space* in certain user-interface actions. If the CX Space contains relevant information about the current usage context, the CX Actuator automatically updates the corresponding data element in the application's IIM. If the user interface of the mobile application utilizes the common model-view-controller (MVC) paradigm (e.g., if it uses Java Swing), the CX Actuator will actuate a change to the corresponding model element, thus triggering an update of all its views. Consequently, the user does not have to spend effort on entering the context information herself. In the event that the system failed to determine the context correctly, the user retains her normal ability to override any results of the context detection.

2.2 Context Space

The CX Space is a run-time data structure that implements an extension of the concept of a Linda Tuple Space [4]. Contextual facts are stored in terms of *shaded* tuples, i.e., each tuple is associated with a "shade", a numerical value ranging between 0 and 1, which expresses its degree of *validity*.

The kinds of tuples that are deposited to and are read from the Context Base are defined by serializing a concept model called a *Context Ontology*. The figure below illustrates this with an excerpt of a Context Ontology that expresses concepts for *actors*, physical positions (coordinates), semantic positions (logical locations), physical proximity, and virtual proximity. The device user herself is contained as "self" in this model. Most of these concepts are self-explanatory, except, perhaps, for the concept of virtual proximity. An example is an actor who calls in on the phone; In this case, she would be considered to be in "virtual proximity".



The concept of a *shaded* tuple space provides us with a means of implementing a simple notion of *context history*: Tuples in the CX Space "fade-out" rather than being removed abruptly when sensor data changes. The fading is realized by decreasing their shade over time until it reaches a minimum threshold, at which time the corresponding tuples are forgotten permanently. Depending on the nature of the corresponding context sensors, some tuples should fade faster than others. Thus, each context inducer determines the fading speed.

2.3 Context Inducers

We refer to components that sense information about the usage context of a Whiskers-based mobile device as *CX Inducers*. Bluetooth-based or RFID-based proximity sensors, GPS positioning sensors, map-based positioning components such as Place Lab [1] are examples for CX Inducers. The instances of these components execute in a separate thread concurrently to the mobile application. They deposit sensed context data in the CX Space as a shaded tuple. As mentioned earlier, the initial shade of a tuple is chosen to express the necessity of its validity. For example, higher shades may indicate actors in close proximity to the device user, lower shades indicate actors that are not so close. Over time, tuples are "forgotten" by decreasing their shades until a minimum threshold is reached.

2.4 Context Tansducers

CX inducers may deliver data about the usage context that may be relevant but not directly usable for a given mobile application. For instance, a context sensor may deliver a geographical position in terms of physical coordinates, but the application requires the user to enter a semantic address. Another example is a context sensor that delivers the telephone number of a caller on the phone (caller ID), but the user interface of the application requires the user to identify a person by name. *Context Transducers* (CX Transducers) are software components that process current context data and derive additional relevant context information. For this purpose, CX Transducers may use device-internal or –external services. For example, a CX Transducer that translates geographical coordinates to addresses may use an external mapping service – or an internal software service such as Place Lab [1]. Tuples in the CX Space that were generated by CX Transducers are called *derived* tuples, in order to distinguish them from *intrinsic* tuples generated by CX Inducers. Like intrinsic tuples, derived tuples are shaded. However, their shade is computed based on the shades of the tuples they have been derived from.

Conclusions and current work

Many mobile applications could benefit from CA features. We have sketched a framework on how to upgrade such applications. Rather than treating context-awareness as an inherent feature that is ingrained in the design of new applications, we provide a way to add-on the CA aspect at a later stage in the software development life cycle. We are currently at a stage where we have designed the Whiskers architecture and we have experimented with a case study. Our current and future work is directed towards implementing a full evaluation prototype for a Whiskers-enabled application.

References

1. Bill, N.S., Anthony, L., Gaetano, B., William, G.G., David, M., Edward, L., Anand, B., Jason, H. and Vaughn, I. Challenge: ubiquitous location-aware computing and the "place lab" initiative *Proceedings of the 1st ACM international workshop on Wireless mobile applications and services on WLAN hotspots*, ACM Press, San Diego, CA, USA, 2003.
2. Dey, A. Understanding and Using Context. *Personal and Ubiquitous Computing*, *5* (1). 4-7.
3. Filman, R.E. *Aspect-oriented software development.* Addison-Wesley, Boston ; Toronto, 2005.
4. Gelernter, D. Generative Communication in Linda. *ACM TOPLAS*, *7* (1).
5. Jan, B. Software product lines: organizational alternatives *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, Toronto, Ontario, Canada, 2001.

10. Oleg, D., Jukka, R., Ville-Mikko, R. and Junzhao, S. Context-aware middleware for mobile multimedia applications *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, ACM Press, College Park, Maryland, 2004.

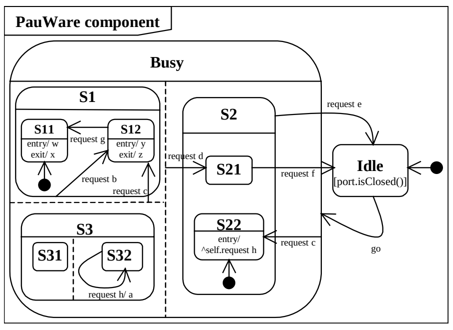
MDA-BASED MANAGEMENT OF UBIQUITOUS SOFTWARE COMPONENTS

Franck Barbier & Fabien Romeo
PauWare Research Group – Université de Pau
Av. de l'université, BP 1155, 64013 Pau CEDEX – France
Franck.Barbier@FranckBarbier.com, Fabien.Romeo@univ-pau.fr

Introduction: the Model-Driven Architecture (MDA) [1] software engineering paradigm aims at considering "models" as first-class products within a software development process. More precisely, this new approach is based on model transformation in which the generated code (in a fairly abstract or detailed form) is just an "implementation model". Concretely, MDA is strongly influenced by the Unified Modeling Language even if other modeling techniques are acceptable. MDA advocates Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs), the later resulting from transformations of the former. In this optic, it becomes natural to focus on PSMs which specifically target ubiquitous applications (see for instance [2]).
In a MDA approach, the need for model checking is often based on "executability" [3]. So, the modeling language used, if "executable", enables model simulation. Such an activity occurs at development time and encompasses, of course, model checking activities, but also testing activities if models include technical details which closely refer to deployment platform properties. In the area of ubiquitous computing, deployment platforms have special features. A relevant research statement is therefore the look for MDA concepts, techniques and tools that comply with the development and the inner nature of ubiquitous applications. For instance, if one is able to provide different executable models which correspond to distinct software component types, how then to deploy and run these models/components in wireless and mobile devices? How to protect these models/components from instable communication, a key characteristic of wireless and mobile platforms? How to endow these models/components with autonomic features (self-managing, self-healing, dynamical reconfiguration…) since controlling runtime conditions is more difficult in ubiquitous systems compared to common distributed systems? Etc.
This experiment paper proposes a MDA-compliant execution engine called *PauWare*. (www.PauWare.com/PauWare_software/). This tool is mainly composed of a Java library which enables the simulation of *UML 2 Sequence Diagrams* (*i.e.*, scenarios) and *UML 2 State Machine Diagrams*, a variant of Harel's Statecharts. The *PauWare.Velcro* sub-library is a J2ME-compliant (Java 2 Mobile Edition) tool which supports the design of the inner workings of software components by means of Statecharts.
We stress in this paper the problem of remote management of software components embedded in wireless and mobile devices and, in certain cases, the possibility of equipping such components with self-managing characteristics [4]. We think that ubiquitous software components and applications require larger management capabilities. Management relies on dedicated infrastructures like, for instance, Java Management eXtensions (JMX) [5]. Despite the availability of management standards, there are few techniques that explain how to instrument the dynamical reconfiguration of components running in remote devices. What could mean self-healing and how one may implement it? Etc.
In *PauWare*, supporting dynamical reconfiguration leads to forcing the state of a component to a well-known stable consistent "situation". For instance, in the figure below, one may go (or go back) to the *Idle* state in bypassing the "normal" behavior of the component. The stable consistent nature of a statechart is a modeled state[1], *Idle* here plus some invariants that can be

[1] Several nested and/or parallel states are also possible in conformance with all of the power offered by the Statecharts modeling technique.

checked at runtime (a port must be closed). States of components are modeled at development time but are also explicit at deployment time since models persist at runtime. The execution model of UML 2 is a run-to-completion model, meaning that component clients' requests are queued and cannot interrupt the current processing, if any, of a request.
Management services may therefore be incorporated into a configuration interface. For concrete reasons, such a facility currently relies in *PauWare* on JMX and on the Wireless Message API (WMA) for communication. Self-management is a more tricky problem. A component may aim at itself deciding to launch a self-configuring operation, "reset" for instance, namely in case of fault recovery: this is typically self-healing. We propose a (parameterized) rudimentary mechanism which is a kind of "undo". If the "autonomic" flag is turned on, a component automatically tries to roll back the current transaction (a global transition from the stable consistent context raising a problem to the immediately prior one) in case of fault detection. Roll back may succeed but it may also fail because many internal business operations (see for instance *x*, *y*, *z* and *w* in the figure below) are executed within a run-to-completion cycle. Canceling the effect of such operations is not always possible. State invariants therefore help to establish if roll back succeeded: all state invariants attached to all nested and/or parallel states must be true when returning to these states.
Conclusion: we think that ubiquitous applications and components require self-adapting capabilities. We nevertheless observe that a gap between theory and practice still remains. While the notions of self-management, self-adaptation are evident and may have several formal shapes, few means currently exist for supporting these concepts in ubiquitous platforms. In the global world of software engineering, MDA put models forward. In such a context, we exploit the power of reputable models like Statecharts to implement self-adaptation.



1. Mellor, S., Scott, K., Uhl, A., Weise, D.: *MDA Distilled – Principles of Model-Driven Architecture*, Addison-Wesley (2004)
2. Grassi, V., Mirandola, R., Sabetta, A.: *A UML Profile to Model Mobile Systems*, Proc. «UML» 2004, LNCS #3273, Springer, pp. 128-142, Lisbon, Portugal, October 11-15 (2004)
3. Mellor S., Balcer, S.: *Executable UML – A Foundation for Model-Driven Architecture*, Addison-Wesley (2002)
4. Barbier, F., Romeo, F.: *Administration of Wireless Software Components*, Proc. ETSI/MOCCA Open Source Middleware for Mobility Workshop, European Telecommunications Standards Institute, Sophia-Antipolis, France, April 6 (2005)
5. Kreger, H., Harold, W., Williamson, L.: *Java and JMX – Building Manageable Systems*, Addison-Wesley (2003)

New paradigms for ubiquitous and pervasive applications

J.Gaber
Laboratoire Systèmes et Transports
Université de Technlologie de Belfort-Montbéliard
90010 Belfort cedex, France
Tel : +33 (0)3 84 58 32 52
gaber@utbm.fr

The recent evolution of network connectivity from wired connection to wireless to mobile access together with their crossing has engendered their widespread use with new network-computing challenges. More precisely, network infrastructures are not only continuously growing but their usage is also changing. They are now considered to be the foundation of other new technologies. Related research area concerns ubiquitous and pervasive computing systems and their applications. The design and development of ubiquitous and pervasive applications require new operational models that will permit an efficient use of use resources and services and a reduction of the need for the administration effort typical in client-server networks.

More precisely, to be able to develop and implement ubiquitous and pervasive applications, new ways and techniques for resource and service discovery and composition need to be developed. Indeed, most of research works to date are based on the traditional Client-Server paradigm. This paradigm is impracticable in ubiquitous and pervasive environments and does no meet their related needs and requirements.

Ubiquitous Computing (UC) deals with providing globally available services in a network by giving users the ability to access services and resources all the time and irrespective to their location [Wei93]. An appropriate model has been proposed in [Gab00] as an alternative to the traditional Client/Server paradigm. The fundamental aspect of this model is the process of service discovery and composition. In the traditional *Client to Server* paradigm, it is the user who should initiates a request, should know a priori that the required service exists and should be able to provide the location of a server holding that service. However, ubiquitous and pervasive environments have the potential ability to integrate a continuously increasing number of services and resources that can be nomadic mobiles and partially connected. A user can be mobile or partially connected and its ability to use and access services will no longer be limited to those that she/he has currently at hand or those statically located on a set of hosts known a priori. Therefore, the ability to maintain, allocate and access a variety of continuously increasing number of heterogeneous resources and services distributed over a mixed network (i.e., wired, wireless, and mobile network) is difficult to achieve with the traditional Client-Server approaches. More precisely, most Client-Server approaches are based on hierarchical architectures with centralized repositories used to locate and to access required services. These architectures cannot meet the requirements of scalability and adaptability simultaneously. The way in which they have typically been constructed is often very inflexible due to the risk of bottlenecks and the difficulty of repositories updating. This is particularly true for the cases where some services could be disconnected from the network and new ones may join it at any time. The alternative paradigm can be viewed as opposed to the Client/Server model and is denoted as *Server to Client* paradigm (Server/Client). In this model, it is the service that comes to the user. In other words, in this paradigm, a decentralized and self-organizing middleware should be able to provide services to users according to their availability and the network status. As pointed out in [Gab00], such a middleware can be inspired from biological systems like the natural immune system (IS). The immune system has a set of organizing principles such as scalability, adaptability and availability that are useful for developing a distributed networking model in highly a dynamic and instable setting. The immune-based approach operates as follows: unlike the classical Client/Server approach,, each user request is considered as an attack launched against the global network. The immune networking middleware reacts like an immune system against pathogens that have entered the body. It detects the infection (i.e., user request) and delivers a response to eliminate it (i.e., satisfy the user request). This immune approach can therefore be considered as the opposed approach to the Client-Server one.

Pervasive Computing (PC) often considered the same as ubiquitous computing in the literature, is a related concept that can be distinguished from ubiquitous computing in terms of environment conditions. We can consider that the aim in UC is to provide any mobile device an access to available services in an existing network all the time and everywhere while the main objective in PC is to provide emergent services constructed on the fly by mobiles that interact by ad hoc connections [Gab00]. The alternative paradigm to the Client/Server one involves the concept of emergence and is called the *Service Emergence* paradigm. This paradigm can be carried out also by an inspired natural immune middleware that allows the emergence of ad hoc services on the fly according to dynamically changing context environments such as computing context and user context. In this model, ad hoc or composite services are represented by an organization or group of autonomous agents. Agents correspond to the immune system B-cells. Agents establish relationships based on affinities to form groups or communities of agents in order to provide composite services. A community of agents corresponds to the Jerne idiotypic network (in the immune system, B-cells are interconnected by affinity networks) [Jer74].

References

[Wei93] M. Weiser, “Hot topics: Ubiquitous computing”. *IEEE Computer*, October 1993.
[Jer74] N. Jerne, “Towards a network theory of the immune system”. Ann. Immunol. (Inst. Pasteur) 125C373, 1974
[Gab00] J. Gaber, “New paradigms for ubiquitous and pervasive computing”, *research report RR-09*, Université de Technologies de Belfort-Montbéliard (UTBM), Septembre 2000.

A model-based approach for designing Ubiquitous Computing Systems

Mahesh U.Patil (*maheshp@cdac.in*)

Centre for Development of Advanced Computing

Keywords

Ubiquitous Computing, Model Based Design, Embedded Systems

Abstract:

Ubiquitous computing systems are environments with ample number of typically small-networked embedded devices. At the same time these devices tend to be invisible from the view of the user by providing user interfaces through physical world, which would generally consist of a wireless infrastructure. Challenges involved in designing and implementing such ubiquitous computing systems add up to the inherent complexities present in the embedded systems thus consuming time in deploying such systems. Moreover, ubiquitous computing system demands for heavy data transfer between these embedded devices for providing the basic communication components like service discovery, service mobility, event notification and context awareness. The basic framework for such embedded devices can be generalized thus providing reusability of the design and implementation across similar requirements. The issues of testability and maintainability would be a major factor in deciding the time and cost involved in deploying such systems and thus cannot be ignored.

Model based design has emerged as a solution for the difficulties and complexities involved in the traditional method of design, implementation and verification of embedded systems. Model Based Design encompasses development phases like: design, simulation, code generation, verification and implementation. It has helped in minimizing the design and requirement errors early in the development cycle thus reducing expensive delays. Hierarchical models have allowed abstraction that hides the underlying

complexities in the system and also provides opportunity for reusability of models across similar requirements and specifications. Model based design provides a facility of integrating pre-tested and deployed legacy code with the new systems designed, thus allowing the direct use of existing modules. Moreover, such a development methodology makes it easy to distinguish between design errors and implementation errors.

To make the concept of Ubiquitous computing a reality requires a comprehensive design methodology that enables short design cycles and reusability, not compromising on the associated cost factor and maintainability. Many projects have taken the advantage of model based design in deploying complex and high performance embedded systems on time and at low cost. Model Based Design has thus become a necessity rather than a facility. The above mentioned challenges in designing ubiquitous computing could be efficiently handled using the model based design.

This position paper would thus focus on a model based approach in designing ubiquitous computing systems that would allow faster design iterations that produce desired performance, functionality and capabilities. We specifically consider a scenario of Ubiquitous Computing System that employs Wireless Sensor Networks and propose a model-based approach that would lead to **rapid prototyping, reusability and abstraction** (by using hierarchical models).