On the Way to Pervasive Computing

Mehdi Jazayeri
Technische Universität Wien
mehdi.jazayeri@tuwien.ac.at
www.infosys.tuwien.ac.at

Abstract
Many predict that the computing environment of the future will be characterized by the presence of numerous invisible sensors and computing elements, autonomously interacting with each other to dynamically construct and provide services to users that enter and leave the environment. The software challenges to turn such pervasive or ubiquitous computing environments into reality are enormous, to say nothing of the hardware and social challenges. These challenges question many of the assumptions we are used to making and many of the solutions we are used to applying in our current software systems. In this talk, I will review some of the work of my group in the areas of software components, security and access control, and device independent Web access and then analyze where our solutions are lacking and must be adapted for pervasive computing.

Keywords: pervasive computing, ubiquitous computing, software components, component-based software engineering, access control, security, device heterogeneity, service discovery, service formation

1. Introduction

There are predictions that the world of the future will be characterized by computing and sensing elements embedded invisibly in the environment: Not only all manufactured appliances will have computing capabilities but also roads and walls and clothes and jewelry, and almost everything else will have some level of computing and communication capability. Such pervasive and ubiquitous computing elements offer vast opportunities for new kinds of software applications. While hardware and even networking advances are being made rapidly, enormous software challenges lie ahead for making these dreams come true. Traditional software engineering has emphasized a disciplined approach that attempts to document the characteristics of the environment and all runtime behavior of the software to produce a self-contained application. This is at odds with the pervasive environment of the future which is characterized by inherent volatility and unpredictability. When a new device enters a room for the first time, it must detect possible useful partners for interaction and devices already in the room must detect the new member and plan on possible collaboration. This fundamental change in the environment calls for a complete rethinking of the software development processes as we know them today.
Back in 1991, Mark Weiser, the acknowledged founder of pervasive computing, wrote, "The technology required for ubiquitous computing comes in three parts: cheap, low-powered computers that include equally convenient displays, software for ubiquitous

applications and a network that ties them all together." [Weiser91]. Today, more than a decade later, computers and networking are cheaper and more available, but many software engineering challenges still lie ahead of us to exploit and "tie" these capabilities together. And, as usual, as new hardware technologies and capabilities become available, software must struggle to keep up with the advances. A problem that is particularly challenging for software researchers and developers in pervasive computing is that the problems are "systems" problems. Since pervasive computing deals with the "tying together" of the dispersed, embedded entities, it is not enough to introduce a stand-alone application or solution: any new concept, idea, service, or application must be able to integrate, inter-operate, and cooperate with existing services and infrastructure. Any experiment to evaluate a new idea necessarily requires the existence or construction of an elaborate environment. Constructing such an environment involves many design decisions and tradeoffs that force us to question many of our fundamental assumptions and approaches in software engineering ranging from requirements and design to implementation and deployment. In this paper, I give an overview of the issues in software for pervasive computing, review some of my group's recent work that contributes to the area and offer some ideas on how these point solutions may be merged together to solve the system problem of constructing a "pervasive computing environment."

In Section 2, I present several pervasive computing scenarios to help identify and motivate the infrastructure services required to support pervasive applications. In Section 3, I review some of the work of my group that we plan to use for constructing a pervasive computing infrastructure for experimentation. In Section 4, I discuss some fundamental tradeoffs in software engineering of pervasive applications. Section 5 concludes the paper by giving an outlook of the future.

2. Pervasive computing scenarios

The possibilities for applications of pervasive computing are only limited by our imagination. Many imaginative scenarios have been expounded in the literature. Such scenarios help in recognizing the many possibilities and also help to identify research goals. Some scenarios are visionary ones that aim to establish long-term goals, the parameters of the needed research, and the interdependence of the research in pervasive computing with that of other domains. The less visionary ones are used to define short- to mid-term research or product-development projects. Some scenarios are created for advocating the importance of the field and drumming up support. Let us consider first three visionary scenarios falling in three different application domains: home, automotive, and health.

A home application scenario. You arrive at home, tired after a day's work. The house identifies you and unlocks the door. Recognizing the fatigue on your face, the home entertainment system starts playing your favorite relaxing music, and replaces the artwork on the electronic wallboards with soothing images of nature. As you enter the bathroom, you are asked if a warm bath should be started. As you enter the kitchen, the display on the refrigerator door suggests a light meal and a specific recipe on the basis of the available ingredients in the refrigerator and fitting your mood. If you

accept the suggestion, step-by-step instructions are displayed, helping you through the steps of the recipe. Once you place the bowl on the stove, the temperature is automatically adjusted to control the cooking time on the basis of your decision whether to take a bath.

An automotive scenario. You are driving to a distant area for a two-week vacation. You plan to explore the area by car during your stay. On the way, your engine detects an irregularity in the operation of the motor. It flashes a warning sign to you cautioning you to drive more carefully and starts a diagnostic procedure. It detects the offending part and sends a report to your car manufacturer. The manufacturer locates the garage nearest to your location and dispatches the part to the garage from a nearby depot. The garage sends a replacement car to meet you on the highway. In the meantime, the police have been notified to look for possible traffic tie-ups in your area. The car display directs you to a specific parking area off the highway to meet the driver that delivers your replacement car. You continue on your vacation and the driver takes your car to the garage. Once the car is repaired, it will be delivered to your hotel, using the information from your itinerary posted by your travel agent, aided by the positioning system in your car.

A health scenario. Your health monitor, possibly installed in your watch, notices that your blood sugar is suddenly elevated. It starts to monitor other conditions more closely to uncover possible causes and potential problems. It contacts your medicine cabinet to ensure that you have adequate medicine in your home and contacts the pharmacy for additional medicine if necessary. It reports the change to your electronic medical record and sends a short message to your doctor, informing the doctor of your current location and evening plans in case the doctor needs to contact you. Back at home, the kitchen notifies the refrigerator to mark all sweets in your home off-limit for you and suggests a healthy recipe for the evening!

These scenarios may appear far-fetched, although they rely mostly on known technologies. The same technologies may be used today for more near-term scenarios. In our European project called MOTION, we are building an infrastructure and tool to support the following scenario:

A mobile teamwork scenario. A multi-national organization has many sites around the world and employees who travel frequently. Traditional working groups, with well-defined management and reporting structures are being augmented with, and in some cases replaced with, ad hoc teams formed for a well-defined, focused, short-term purpose. For example, a particular design flaw is suspected in a product, such as a mobile phone, that is deployed in the field. A team must be formed from potential expert employees around the world to examine, discuss, and propose a solution to the problem. Virtual meetings must be held to support communication and problem solving. Some members may be traveling and have access to limited computer and display capabilities. They may be in different time zones. The infrastructure must support access to large amounts of information and documents, synchronous and detached communication among people, and event notification. For example, one member may need to be notified as soon as some particular document becomes

available or another member comes on-line. Many of the needed technologies to integrate such a system, such as chat and instant messaging are already available. But they need to be integrated in new ways to meet the demands of the new environment. The pervasive computing environment is characterized by heterogeneity (of devices and networks) and unpredictability and lack of structure of communication patterns.

A review meeting scenario. An activity that many people are involved in is what we may generically call "review meetings." Examples of such review meetings are inspections and code reviews in software engineering, and program committee meetings for the planning of scientific conferences. A review meeting has a well-defined, structured, agenda and is preceded by a sometimes lengthy preparation phase and succeeded by a follow-up phase. In the preparation phase appropriate documents are distributed to the meeting participants for review and analysis, with specific instructions on what to analyze. All participants must complete their tasks before the meeting. Once the preparation phase is completed, the participants come together in a possibly virtual meeting to discuss the review subject. In a follow-up phase, the meeting "secretary" coordinates the completion of all the action items decided at the meeting. A pervasive computing application could provide a wide-range of services including the monitoring of progress and its notification to the participants during the pre- and post-phases, coordinating the actions of the participants, detecting the arrival of the participants at the meeting, designing appropriate seating arrangements at the meeting depending on the roles of the participants, maintaining and possibly adjusting the agenda during the meeting on the basis of the meeting progress and interim results, providing access to some documents and denying access to others, providing communication facilities both for group and bilateral communications. The application helps in both accessing data and managing the process.

These latter two scenarios are closer to today's technology but can clearly be extended to exploit the more sophisticated pervasive computing technologies as they become available. They may be viewed as stepping stones to future pervasive computing environments and applications.

The scenarios, both short-term and long-term ones, share some important characteristics that help identify the requirements for the needed standards and infrastructures of pervasive computing. Strong network connectivity, both locally and globally, is clearly necessary. The interconnectivity that was advanced by the Internet will need to extend "pervasively." Everything and everyone will need to be connected. In some cases, networks must be able to adapt to frequent changes and in some cases they must meet stringent reliability and dependability requirements. While you may tolerate the loss of access to a Website temporarily, or even permanently, you cannot be as forgiving if the connection is between your pacemaker and your doctor's office. While mobile computing, which relies to some degree on pervasive computing, assumes intermittent connectivity, and even detached computing, many applications of pervasive computing require guaranteed connectivity.

In some cases, such as in the home application scenario, the environment may be designed with all connecting devices anticipated in advance. In other cases, such as a

virtual meeting scenario, the environment is highly heterogeneous and connecting devices cannot be anticipated. Security and privacy properties have to be assumed: as more and more information becomes available electronically, the need for safeguarding the security and privacy of the information becomes even more imperative but also complicated. For example, I might want my doctor to be informed of my blood sugar level but not of my dinner plans for the evening. The systems must provide means for paying for services. Since so much happens behind the scenes, invisibly to the user, new kinds of payment models will be needed. Services must be highly personalized to avoid inundating the user with unneeded information or burden the user with lengthy search and retrieval scenarios. Finally, the environment is required to exhibit highly intelligent behavior, anticipating user needs and providing sensible services without waiting for user requests.

In the next section, I will explain some of these basic requirements in more detail, derive more concrete software engineering requirements, and present some of my group's work towards meeting those requirements.

3. Infrastructure for pervasive applications

The scenarios presented in the last section show a range of applications of pervasive computing. These scenarios motivate the need for new kinds of architectures and approaches to software engineering for the needed applications. The architectures must enable high interconnectivity, highly dynamic interactions among diverse components, and a high level of coordination among different and heterogeneous, possibly independently-designed activities. An infrastructure for pervasive computing must therefore enable components to recognize, communicate with, and coordinate with each other. Indeed, an application may be constructed from independent components. The first question raised in this environment is then an old one: what characteristics should a software component have? We will first address this issue. Next we consider the issue of interactions among components. While the infrastructure should enable free and rapid interaction among components, not all accesses should be allowed. We consider the issues of access control and architectures that support access control in applications. Finally, we address the problems of addressing device heterogeneity in the environment.

3.1 Software components for pervasive computing

The pervasive computing environment forces us to face the need for components and their boundaries more clearly. Pervasive services will have to be composed from individual "components" residing in the large number of heterogeneous computing elements. The hardware environment itself will force a natural boundary between components. This may be the most clear-cut definition of a component. A component will be an independently deployable piece of software that resides on one hardware element and provides a service element. Of course, there may be more than one component on each hardware element. Just as Web Services are emerging in the Web computing infrastructure as a "component", some form of component will have to emerge in the pervasive computing infrastructure. Perhaps this will also be a Web

service, accessible through a URL? More likely, it will be a peer service with a well-defined peer protocol.

The most striking characteristics of software components in the pervasive computing environment are the need to deal with heterogeneity and the need for dynamic (ad hoc) adaptation to, and interaction with, communicating components. Current component models are homogeneous in the kind of components they support. Components are basically of the same "size" and "power." For example, JavaBeans components are for desktop environments while Enterprise JavaBeans are for server and enterprise-wide components. To make application development manageable, we probably need a single component model that is "scalable" in the sense that it supports the development of components of various granularity, components that can reside in tiny computing elements, such as wearable computers, but can also grow to take advantage of resource-rich computing elements such as laptop machines. This is currently a real problem, even for a single programming language. Java has several versions of its JVM for resource-poor and resource-rich environments. You need to think differently about the component model if you are designing for a smart phone or for a laptop. In a scalable component model, at least the modeling and analysis techniques must span a range of granularities.

Our work on software components started with basic language-specific components [Jazayeri95]. While language-specific components are still important, the pervasive environment requires us to also deal with heterogeneous components. The work of Johann Oberleitner [Oberleitner01] deals with the heterogeneity of component models. He has designed and built the Vienna Component Framework (VCF) that captures the essential characteristics of different component models such as COM, CORBA, JavaBeans, Enterprise JavaBeans, and simple X-Windows components. The VCF provides foundational support for component-based software engineering (CBSE). It is used as the lower layer of a CBSE environment called the Component Workbench [Oberleitner02]. The Component Workbench provides transparent access to each component model and allows applications to be built from components coming from different component models. It also supports the user in maintenance activities such as replacing components with other components. For example, if an application is to be moved to an environment where a needed component is not available, that component can be replaced with an equivalent component in the new environment. Consider an application that is built in the MS Windows environment and it uses the Internet Explorer as a component. In moving the application to a UNIX environment, the browser can be replaced by a Mozilla or Opera browser. The Component Workbench supports this replacement by helping the user match interfaces, methods, and attributes from one component to another component.

The Vienna Component Framework captures the common characteristics of different component models by providing a uniform type system across component models. It supports an event-based communication mechanism for the interconnection of components. It also offers an architectural description language that describes the composition of an application in terms of components. The architectural description can be moved to different environments and instantiated in terms of the available components. The components may come from different component models. An

interesting application of this architectural description for the pervasive computing environment is the ability of devices to carry an application's architectural description and instantiate it in a new environment. In this way, portable service descriptions may be exchanged among devices and environments.

One of the key problems of building applications out of components—component-based software engineering—is what to do if the component you need is not available in the catalogs you have. Clearly, no catalog will have every component that an application developer needs. But, often, there will be a related component, or one that is "almost" the one needed. There are several possible paths to take in this case. One is for the developer to modify the related component to make it fit his needs. This approach defeats the purpose of component-based development because of the fundamental reason that it breaks the separation of concerns between component development and component usage. A more effective approach is to automatically "adapt" the existing component to the need of the application. Ideally, with automatic adaptation, the component developer can provide a minimal catalog of components but the user gains the benefits of a larger catalog. The goal in the component work reported in [Jazayeri95] was to use generic programming to build powerful yet minimal catalogs. Thomas Gschwind's dissertation [Gschwind02] concerns the topic of automatic component adaptation and introduces a particular kind of adaptation called "type-based adaptation."

Type-based adaptation relies on a repository that stores pre-built adapters and a meta-description of the transformations that the adapters perform. Modern languages such as Java, and modern component models such as Corba Components support strongly-typed components and provide mechanisms for querying the type of the component at run-time. Type-based adaptation exploits these features to automate the adaptation process. The programmer writes the adapters, the system decides when an adapter is needed to be applied. More importantly, the process implemented by Gschwind can determine when it is necessary to chain several existing adapters together to effect an adaptation that is more powerful than any one existing adapter can do by itself. This ability to chain adapters together greatly increases the power of the process and requires many fewer adapters to be written by the programmer directly. The Component Workbench uses type-based adaptation to support the replacement of components from one component by components from another model. Type-based adaptation is also a good fit for pervasive computing environments because the devices that need to communicate with each other and the communication protocols they use are not known a-priori. Since users of such devices do not want to deal with incompatibilities, the adaptation has to be performed automatically.

The Vienna Component Framework and type-based adaptation are clearly two important ingredients for dealing with the heterogeneity of components that we will face. But they are only preliminary steps towards meeting the wide heterogeneity and dynamicity we expect to face pervasive computing environments. Components will have to adapt dynamically and compose dynamically. Versioning and legacy issues associated with such dynamically evolving services and their components will pose

enormous challenges for software engineers. We also will need to be able to deal with very tiny components residing in small embedded devices.

An interesting way to exploit component adaptation to deal with the volatility and unpredictability of pervasive computing is to maintain an adapter repository at each site. When a new device enters the site, the adapters in the repository can be used to help it match the interface requirements of available services. It is also possible for the new device to offer some new adapters to the repository or download some adapters from the repository to carry away. Other types of adapters may be used to control the behavior of incoming components to respect the requirements of the host environment, for example in terms of security. Yet other adapters may be used to provide more limited versions of existing services to untrusted arriving components.

For anyone working on components, the classic papers of Parnas on software design [Parnas72, Parnas76, Parnas79] are required reading.

3.2 Device heterogeneity

The basic premise of pervasive computing—everything connected—guarantees heterogeneity at all levels: infrastructure, hardware, software, and people. All kinds of devices must be supported. Perhaps in some specific application scenarios it is possible to restrict the kinds of devices that are supported but, in general, the environment must anticipate the existence of a wide variety of devices. If we consider devices used by the user to interact with the system, they can range from standard ones such as laptops, PDAs, and phones, to emerging ones such as those embedded in clothing and eyeglasses. The variety of available devices has several implications. One is the kind of input-output devices: textual and graphic input-output will not be the only forms of human-machine interaction. Audio, visual, and other sensory modes of communication will be prevalent. Another implication is the requirement that the environment must be prepared to adapt to the device currently used by the user. For example, if the user is requesting information and he is currently driving, the retrieved data should be relayed to him with an audio message through the car radio.

In my group, we have taken a first step towards achieving device independent interactions in the context of the worldwide Web. The worldwide Web became popular and started its explosive growth in 1995. The number of Websites and Web pages has been growing exponentially since then. Initially, the pages were written in HTML, which is the assembly language of Web page development. Most of these pages were written for desktop display devices with relatively large screen sizes. Web page developers were excited to write pages that looked cool and exhibited fancy behavior. Soon, however, these developers faced the problem of maintaining those fancy pages and the "legacy" problem, well-known to software engineers, became also a problem of Web page developers.

A number of "Web engineering" methods have been invented to combat the development and maintenance problems of Websites. Many of these methods rely on separating the description of the content of the information from its presentation and prescribing a particular set of steps to follow in developing a Website. The XML and XSL standards were created to enable a uniform way of implementing this separation. XML helps structure Web documents and data and its accompanying XSL standard

helps to describe structures ("styles") that define presentation properties of documents. These and other standards help Web developers to structure Web sites to ease the maintenance burden.

The Web engineering methods and standards have a difficult time trying to keep pace with the rapid expansion and evolution of the Web. We ourselves have been going through several generations of our Web engineering tools and techniques since 1995 [Kirda01]. One of the areas in which Web developers have seen many changes in the Web is the introduction of many different types of "Web devices" for accessing the Web. If a Web page may be accessed from a desktop, a PDA, and a mobile phone, how is the Web developer to build the Web page, avoiding the creation of three different versions of the page? Since we may expect a constant stream of new devices to be introduced in the foreseeable future, the problem of device heterogeneity poses serious maintenance problems for Web developers in the coming years. The dissertation of Engin Kirda [Kirda02] introduces a novel way of achieving device independence in Web engineering.

Solutions to the device independence problem in Web engineering range from completely manual (the developer creates a different page for every device) to completely automatic (the "system" detects the target device and formats the page appropriately to be displayed. The technique developed by Kirda falls somewhere in the middle of this range. The developer explicitly decides on what types of devices are to be supported and classifies devices according to their display capabilities. The goal of the technique is to enable the "system" to display a page appropriately depending on the device type. The developer indicates places in the page that may require device-specific treatment. For example, the developer indicates all places in the page where page breaks are allowed. A piece of software called the "splitter" will try to fit the maximum amount of data to be displayed on the device, respecting the allowable page break boundaries. Another technique works with forms that require interaction with the user. A piece of software buffers the form's input and output appropriately. With this combination of automation, helped by developer guidance, the technique achieves good display quality on a variety of devices—an achievement often not gained by fully automatic means.

Another relevant piece of work in this area is the separation of not only content and layout, but also the logic of the application. By separating these three orthogonal aspects of the application, not only maintenance becomes more manageable, but also different parts of the application may be reused in other applications. For example, the same logic may be used to retrieve weather information and display it as a map on a large desktop display, or simple textual report on a small phone, or play on a radio. The paper [Kerer00] presents a way of separating layout, content, and logic in Web applications. The separation technique has been used to build a generic content management system for Web databases [Kerer02].

These solutions, although preliminary, appear to be necessary first steps towards sophisticated pervasive computing applications. In the future, there will be a much wider variety of devices and a higher degree of dynamic adaptability will be required. Another difference will be the application architecture. Web-based applications have a rather uniform client-server architecture. The pervasive computing applications will

be much more distributed and diverse. Peer-to-peer architectures with event-based notification will be more common.

3.3 Access control

The wide availability of services and the high mobility of users among different environments require the provision of security mechanisms to ensure the safe usage of services by legitimate users and the protection of services from unauthorized uses. Because of the wide range of services, many diverse and flexible security models and mechanisms will be needed. Either standard security mechanisms will have to be embedded in the environment and used by all applications or each application will have to build its own security mechanisms. Most likely, a combination of the two will be needed.

One of the most important aspects of security is access control, to ensure that services are only available to authorized users and those authorized users are allowed appropriate privileges. For example, a guest at a hotel may be allowed to print on the hotel's printer available in the lobby but not change the contents of the event display in the same lobby. We have been working in two directions to help application developers include sophisticated access control in their applications.

Pascal Fenkam [Fenkam01, Fenkam02b] has developed a flexible, generic, highly configurable access control subsystem to be included in applications. The subsystem supports a powerful model that consists of users, resources, and access rights. New access rights may be defined and users may be given different rights, including the right to define new access rights. An application developer may configure this component for an application, offering a restricted or a highly flexible mechanism to the application user. The component makes it easier for the application developer to add access control to an application.

Another direction we have followed to help application developers add security to applications is to design an architectural style for building secure applications. In general, inclusion of security and verifying the security properties of applications is difficult and application developers need practical techniques. Architectural styles provide guidelines for developers in building their applications. The DPS architectural style proposed in [Fenkam02] restricts the amount of security checks in an application to certain control paths in the program, thus guiding the programmer in application construction and enabling easier verification of the application's security properties.

The access control subsystem and the architectural style mentioned here are two building blocks that may be used for pervasive computing applications. But much more is needed. In fact, more flexible and powerful security models are needed to protect services that are composed of distributed service components. It is reasonable for a user to provide a password once to authenticate himself or herself. But the user should not have to provide passwords each time a new service is being contacted. This problem is known as single sign-on [Savar99]. Some kind of universal access control must be attached to the user and be checkable by all services. Many more architectural styles and design patterns will be needed to help in the building of services. In a larger sense, security is part the bigger problem of dependability.

Ensuring the dependability of services formed from service elements that are joined in dynamic and possibly ad hoc ways is a considerable challenge.

4. Tradeoffs and design decisions

As in any system design effort, pervasive computing environments offer the system designer a large number of design tradeoffs. In this section, I list two example tradeoffs that are not easy to resolve.

The first tradeoff concerns the amount of intelligence or automation that may be built into the environment or application. Clearly, the more the system can infer and do automatically on behalf of the user, the more supportive the environment becomes for the user, freeing him or her to concentrate on other important tasks. Garlan [Garlan02] calls this "distraction-free pervasive computing." But there is of course a limit to how much you want the system to do for you automatically. For example, if you book a flight to London, should the system automatically realize that your favorite play is on in London and book tickets for you, or should it ask you first? Or if your blood sugar is elevated, should the system notify your doctor or ask you first? We need models and guidelines for making such tradeoffs. These tradeoffs involve not only system design decisions but some of them may require the consideration of privacy and ethical guidelines. Certainly, different users will want to have different guidelines applied to them.

Another serious conflict in the pervasive computing environment is the interplay between openness and security. A basic premise of the environment is that components dynamically recognize each other and cooperate with each other. On the other hand, such an open environment requires strict control over resources. You may not want to offer all available services to every new element that enters the environment. What is the best way for the components in the environment to identify and qualify one another? We need a lightweight mechanism because it has to be used heavily. A reasonable idea (which still needs a lot of work!) is that the application components carry "tokens" or capabilities that authorize them to access or provide services. Once an application component has obtained a token, it uses it for its authentication. The low-level communication mechanism is responsible only for secure communication among authorized parties. But this arrangement goes against the well-accepted end-to-end system design philosophy that says that the lowest layers of the system must provide only basic, lightweight functionality.

Not only components must authenticate clients, but clients will also want to authenticate server components. Here there are many issues related to trust. Should some components be trusted more than others? If two components provide the same service, how do you select which one to use? These issues are now being investigated especially in peer-to-peer systems.

The need to combine security and access control with dynamically interacting components of varying granularity may be the biggest challenge of component composition in the pervasive computing environment.

5. Summary and conclusions

The worldwide Web fostered an environment in which more and more people are connected through a vast networking infrastructure to vast amounts of information. The pervasive computing environment expands the capabilities offered by the Web in many directions. It adds connectivity to any object in general, not only to information. It supports not only access to information but also the formation of services and processes. It provides services behind the scenes, without interrupting the users. It supports the creation of services dynamically, to adapt to user needs. I have discussed representative scenarios for pervasive computing that show some of the possibilities of pervasive computing.

To make pervasive computing a reality, a significant amount of research is still needed, much of it in software engineering. Software is an essential ingredient of pervasive computing and we need new methods and techniques to meet the new challenges. Our techniques must be able to deal with highly dynamic, highly heterogeneous environments and scale the range of tiny to powerful computing elements.

I have given examples of some work in my group that needs to be extended to satisfy the needs of pervasive computing. Pervasive computing offers not only exciting opportunities but also significant design challenges. Our approach is to re-examine our current technologies in light of the new requirements of pervasive computing. This is an evolutionary approach to attacking the enormous challenges of pervasive computing.

Acknowledgments

The work I have reported here has been developed over the years by several current and former students in the Distributed Systems Group, and supported by several projects. The people include: Robert Barta, Pascal Fenkam, Harald Gall, Thomas Gschwind, Clemens Kerer, Engin Kirda, Joe Oberleitner, and Markus Schranz. I would like to also acknowledge the support of the Hewlett-Packard Corporation, the IBM Corporation, and the Austrian National Science Foundation.

This work was performed in several projects, supported part by the European Commission, including within the ESPRIT Framework IV project no. 20477 ARES, and in IST Framework V projects EASYCOMP, MOTION, and OPELIX.

6. References

[Fenkam01]
P. Fenkam, *A Dynamic User Management Access Control for Web Sites" Distributed Systems Group*, Master's Thesis, Technical University of Vienna and Insitute for Software Technology, Technical University of Graz, 2001.

[Fenkam02]
P. Fenkam, H. Gall, and M. Jazayeri. "DPS—An Architectural Style for Development of Secure Software." In Proceedings of the Infrastructure Security Conference (Bristol, 1-3 Oct., 2002), Springer Verlag.

[Fenkam02b]

P. Fenkam, H. Gall, and M. Jazayeri, "Visual Requirements Validation: Case-Study in a Corba-supported environment," *IEEE Joint International Requirements Engineering Conference (RE '02)* (Essen, Germany), IEEE CS Press, September 2002.

[Garlan02]
D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: Toward Distraction-Free Pervasive Computing," Pervasive Computing, April-June 2002,

[Gschwind02]
T. Gschwind, *Adaptation and Composition Techniques for Component-Based Software Engineering*, PhD dissertation, Distributed Systems Group, Technical University of Vienna, 2002.

[Jazayeri95]
M. Jazayeri, "Component programming: a fresh look at software components," Proc. 5th *European Software Engineering Conference (ESEC '95)* (Sitges - Barcelona, Spain), LNCS 989, pp. 457-78, September 1995.

[Joshi01]
J. B. D. Joshi, W. G. Aref, A. Ghafoor, and E.H. Spafford, "Security Models for Web-Based Applications," CACM 44(2):38-44, Feb. 2001.

[Kerer00]
C. Kerer and E. Kirda, "Layout, Content and Logic Separation in Web Engineering," 9th International World Wide Web Conference, 3rd Web Engineering Workshop (Amsterdam, 15-16 May, 2000). LNCS 2016, Springer Verlag, May 2000.

[Kerer02]
C. Kerer, E. Kirda, R Kurmanowytsch, "WebCUS: An XML-based Web content update system," IEEE Internet Computing, August 2002.

[Kirda01]
E. Kirda, M. Jazayeri, C. Kerer, and M. Schranz, "Experiences in Engineering Flexible Web Services," IEEE Multimedia, January - March 2001.

[Kirda02]
E. Kirda, "Engineering Device-Independent Web Services," PhD dissertation, Distributed Systems Group, Technical University of Vienna, 2002.

[Oberleitner01]
J. Oberleitner, *The Component Workbench: A Flexible Component Composition Environment*. Master's Thesis, Distributed Systems Group, Technical University of Vienna, Austria, October 2001.

[Oberleitner02]
J. Oberleitner and T. Gschwind, "Composing Distributed Components with the Component Workbench," Proceedings of the 3rd Intl. Workshop on Software Engineering in Middleware, Jan. 2002.

[Parnas72]
D. L. Parnas, "On the criteria to be used in decomposing systems into modules," Communications of the ACM, 15(12): 1053–8, December 1972.

[Parnas76]
D. L. Parnas, “On the design and development of program families,” IEEE Transactions on Software Engineering, 2(2): 1–9, March 1976.

[Parnas79]
D. L. Parnas, “Designing software for ease of extension and contraction,” IEEE Transactions on Software Engineering, 5(2):128–138, March 1979.

[Weiser91]
M. Weiser, “The Computer for the 21st Century,” Scientific American, 1991.

[Savar99]
V. Samar, ‘Single sign-on using cookies for Web applications.” Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999. (WET ICE '99) Proceedings. IEEE 8th International Workshops, pp 158 -163.