Software Architectural Support for Handheld Computing

The authors present a software-architecture-based approach to support computing on distributed, handheld, mobile, resource-constrained devices.

Nenad Medvidovic
Marija Mikic-Rakic
Nikunj Mehta
Sam Malek
University of Southern California

Software engineers and practitioners traditionally have focused on *programming in the large* (PitL),[1] the development of large-scale software systems. However, the increasing speed and capacity and decreasing costs of hardware, emergence of the Internet, and proliferation of handheld devices such as cell phones and PDAs have fueled demand for new technologies that support highly distributed computation on small, mobile platforms.

Employing handheld mobile devices in complex scenarios such as sea exploration, environmental monitoring, freeway-traffic management, fire fighting, and natural disaster damage assessment will require addressing a number of daunting challenges. This new set of challenges is more appropriately characterized as *programming in the small and many* (Prism)—software development for highly distributed, dynamic, mobile, heterogenous computation on large numbers of small, resource-constrained platforms.

Recent studies[2,3] indicate that a promising approach is to apply *software architecture* principles to the development of software systems in the Prism setting. These principles provide abstractions for representing the system's structure, behavior, and key properties.[4,5]

Architectures are generally described in terms of *components* (computational elements), *connectors* (interaction elements), and their *configurations*. An *architectural style* further defines a vocabulary of component and connector types as well as a set of constraints on combining instances of those types in a software system. Examples of styles include blackboard, C2, client-server, pipe and filter, and push-based.[5,6] Selecting an appropriate architectural style is a key determinant of a software system's success.

Software architectures provide design-level models and guidelines for composing software systems. However, to be useful in a development setting, these models and guidelines require support for implementation and evolution.[5,7] As the "Prism Challenges" sidebar describes, Prism's highly distributed, heterogenous, and mobile nature amplifies the software development demands that permeate the entire software engineering life cycle. Therefore, during the past four years we have focused on the design, implementation, and empirical evaluation of techniques for supporting architecture-based software development in the Prism setting.

Several aspects of architecture-based development—component-based system composition, explicit software connectors, architectural styles, upstream system analysis and simulation, and support for dynamism[5]—make it a good fit for Prism's demands. Our solutions for adaptable system design, efficient implementation, and tailorable execution all leverage the approach's architectural basis and were guided by four research objectives:

Published by the IEEE Computer Society

- native support for programming the architectural concepts;
- efficiency to execute on small, resource-constrained platforms;
- scalability to many devices, components, connectors, and communication events; and
- extensibility and configurability to acccommodate varying development concerns across the Prism domain.

Our goal has been to adapt PitL techniques whenever applicable, but to support Prism's unique characteristics we have developed novel solutions in three areas: explicit architectural idioms suitable for Prism software system design; a lightweight, tailorable middleware for transferring architectural decisions to effective system implementations; and runtime techniques that support continuous system evolution, redeployment, and mobility.

EXAMPLE PRISM APPLICATION

To understand the concepts underlying our approach, consider an application for distributed military troop deployment and battle simulations (TDS). A computer at headquarters gathers information from the battlefield and displays the current locations of friendly and enemy troops, vehicles, and obstacles such as mine fields. As Figure 1 shows, the headquarters computer is networked via secure links to a set of commander PDAs, which are connected directly to one another and to a large number of soldier PDAs.

Commanders control their own part of the battlefield by deploying troops, analyzing the deployment strategy, transferring troops, and so on. If the headquarters device goes out of range, a designated commander assumes the HQ role. Soldiers can only view the segment of the battlefield in which they are located, receive direct orders from the commanders, and report their status.

TDS illustrates the concept of *multiplicity* inherent in Prism. In designing the application, we used a combination of four architectural styles: client-server, push-based, peer-to-peer, and C2. In addition, we implemented TDS in three dialects of two programming languages—Sun Microsystems' Java JVM and KVM, and Microsoft's Embedded Visual C++. Finally, we deployed TDS on several types of mobile devices—Palm Pilot Vx and VIIx, Compaq iPAQ, HP Jornada, NEC MobilePro, Sun Ultra, and PC—running four different operating systems: PalmOS, WindowsCE, Windows 2000, and Solaris.

Prism Challenges

Software engineers and practitioners have been working actively on application modeling, analysis, simulation, and semiautomated system implementation for several decades. The highly distributed, heterogeneous, and mobile nature of programming in the small and many only amplifies these problems. In addition, Prism presents a number of unique software development challenges that permeate the entire software engineering life cycle.

Resource constraints

Devices on which applications reside may have limited power, network bandwidth, processor speed, memory, and display size and resolution. Constraints such as these demand highly efficient software systems in terms of computation, communication, and memory footprint. They also demand more unorthodox solutions such as "off-loading" nonessential parts of a system to other devices.

Heterogeneity

Programming in the large is characterized by extensive standardization—for example, Java, Linux, and XML. In contrast, Prism must reconcile proprietary operating systems such as PalmOS and Symbion, specialized dialects of existing programming languages such as Sun Microsystems' Java KVM and Microsoft's Embedded Visual C++, and device-specific data formats such as *prc* for PalmOS and *efs* for Qualcomm's Brew.

Computing infrastructure

Software developers must make tradeoffs to address the computing constraints that mobile platforms impose. The infrastructures of such technologies may thus lack certain services for reasons of efficiency or through accidental omission. For example, Java KVM does not support noninteger numerical data types or server-side sockets. Likewise, typically employed techniques for code mobility, such as Java XML encoding, may be computationally too expensive to support.

Figure 1. Troop deployment and battle simulations (TDS) application distributed across multiple devices. The headquarters computer is networked via secure links to a set of commander PDAs, which are connected directly to one another and to a large number of soldier PDAs.

TDS utilizes 105 mobile devices and mobile device emulators running on PCs, with a total of 245 software components interacting via 222 software connectors. The dynamic size of the application is approximately 1 Mbyte for the headquarters subsystem, 600 Kbytes for each commander subsystem, and 90 Kbytes for each soldier subsystem.

Hierarchical composition supports multiple levels of abstraction of the system's architecture, which in turn aids the system's extensibility.

DESIGN SUPPORT

We have developed a set of software design idioms to effectively capture the characteristics of Prism application architectures. Ideally, these idioms would comprise a software architectural style,[5] but, as recent studies[8] have recognized, it is unclear which styles are most suitable for this setting. We have therefore developed Prism-SF, an architectural style framework that inherits many characteristics from previous work[5-7] but is unique because it is configurable—developers can instantiate multiple specific architectural styles from it, possibly even in a single application.

Architectural elements and composition rules

Prism-SF directly provides concepts for modeling an application's architecture. Its components maintain state and perform computations. To support scalability and extensibility, Prism-SF components cannot assume a shared address space; instead, they interact by exchanging events via master, slave, and peer communication ports.

Events. An *event* consists of a name and payload. An event's payload includes a set of typed parameters for carrying data and miscellaneous metalevel information—sender, real-time deadline, and so on. A Prism event is either a *request* for a recipient component to perform an operation, a *notification* that a sender component has performed an operation, or a *peer* event enabling symmetric communication between components. Prism-SF components send requests through master ports and receive them through slave ports, send notifications through slave ports and receive them through master ports, and send and receive peer events through peer ports.

Connectors. To further support scalability, Prism-SF *connectors* mediate interactions among components by controlling the distribution of all events. An *asymmetric* connector supports request-notification interactions, while a *symmetric* connector supports peer event interactions. Components attach to asymmetric connectors via master and slave ports and to symmetric connectors via peer ports. Prism-SF connectors can support event unicast, multicast, and broadcast semantics.

The *distributed* connectors that span device boundaries play a key role in scalability by supporting interaction among components residing on different devices. A distributed connector marshals and unmarshals application data or mobile code; it dispatches and receives events across the network; and it may perform data compression for efficiency and encryption for security.

Hierarchical composition. Prism-SF supports *hierarchical composition* of both components and connectors. This provides an effective mechanism for supporting multiple levels of abstraction of the system's architecture, which in turn aids the system's extensibility. Hierarchically composed components and connectors encapsulate their constituent subarchitectures. When performing hierarchical composition, a designer maps each composite component's external port to one port of the internal component or connector in its subarchitecture.

Instantiating Prism-SF

It is easy to instantiate Prism-SF into a number of distributed systems styles that are likely to be useful in the Prism setting.[6]

Client-server. In this type of system, client and server components communicate via synchronous asymmetric connectors—typically remote procedure calls—using request and notification (response) events. A client component usually has a single master port through which it sends requests to and receives responses from the server, while the server has multiple slave ports, one for each client.

Peer-to-peer. The peer-to-peer style defines only one component type that uses logical call-return pairs as events to communicate via symmetric, event-based connectors. A peer component can have numerous peer ports through which it sends and receives events, depending on the number of its peer relationships with other components. Peer components typically do not have master or slave ports, although it is possible to combine client-server and peer-to-peer architectural styles in a single application. Prism-SF supports such a combination naturally.

C2. The C2 style[7] is a layered network of concurrent components that communicate via neighboring connectors. C2 supports one component type similar to the peer-to-peer style, but, unlike peer-to-peer components, C2 components exchange request and notification events to communicate via asynchronous asymmetric connectors. A C2 component has single master (top) and slave (bottom) ports.

TDS architecture

Figure 2 shows a subset of the TDS architecture designed using an instance of Prism-SF. In this configuration, each component has single master, slave, and peer ports; components interact solely via connectors; the design includes both symmetric and asymmetric connectors; and symmetric and asym-

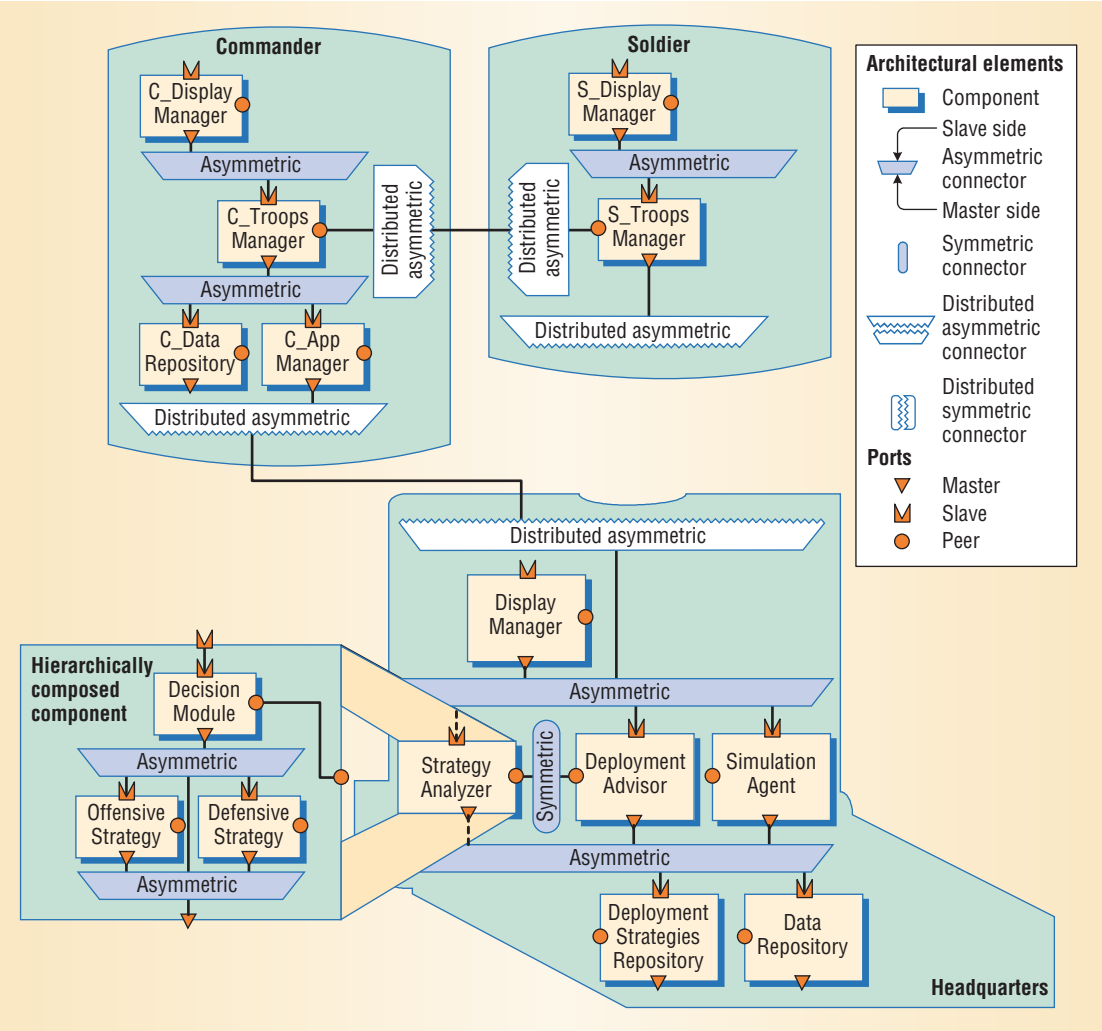

Figure 2. TDS architecture. Only single Commander and Soldier subsystems are shown. All interaction occurs with messages routed through appropriate connectors. The internal architecture of the hierarchically composed Strategy-Analyzer consists of three components and two connectors.

metric connectors cannot be connected to one another.

This architecture consists of single Headquarters, Commander, and Soldier subsystems. DataRepository maintains a model of the system's overall resources: terrain, personnel, tank units, and mine fields. StrategyAnalyzer and DeploymentAdvisor analyze and suggest deployments of friendly troops, respectively. SimulationAgent incrementally simulates battle outcomes based on the current situation. DeploymentStrategiesRepository stores the strategy and deployment rules. C_TroopsManager and S_Troops-Manager allocate and transfer resources and periodically update the state of resources. Finally, DisplayManager provides the application's user interface.

IMPLEMENTATION SUPPORT

Prism-SF provides design guidelines for composing large, distributed, decentralized, mobile systems. Prism-MW, a lightweight architectural middleware, supports implementation of these guidelines. Although related to several existing middleware platforms,[2] Prism-MW has characteristics specifically geared to the Prism domain.

Implementing architectures in Prism-MW

Prism-MW comprises an extensible framework of implementation-level classes representing an application's architectural elements, their properties, and their composition rules. It is easily configurable to support style-specific characteristics in applications. Programmers construct Prism application architectures by reusing the appropriate Prism-MW classes or extending them with application-specific details.

Prism-MW supports architectural abstractions by providing classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. Figure 3 shows the Unified Modeling Language class design view of Prism-MW. The green-colored classes constitute the middleware core, which, to ensure compactness, contains only eight classes and six interfaces. To use Prism-MW's basic features, an application developer only needs to be familiar with four of the classes, shaded dark green.

Brick is an abstract class that encapsulates common features of its subclasses—Architecture, Component, and Connector. Architecture records the configuration of its constituent components and

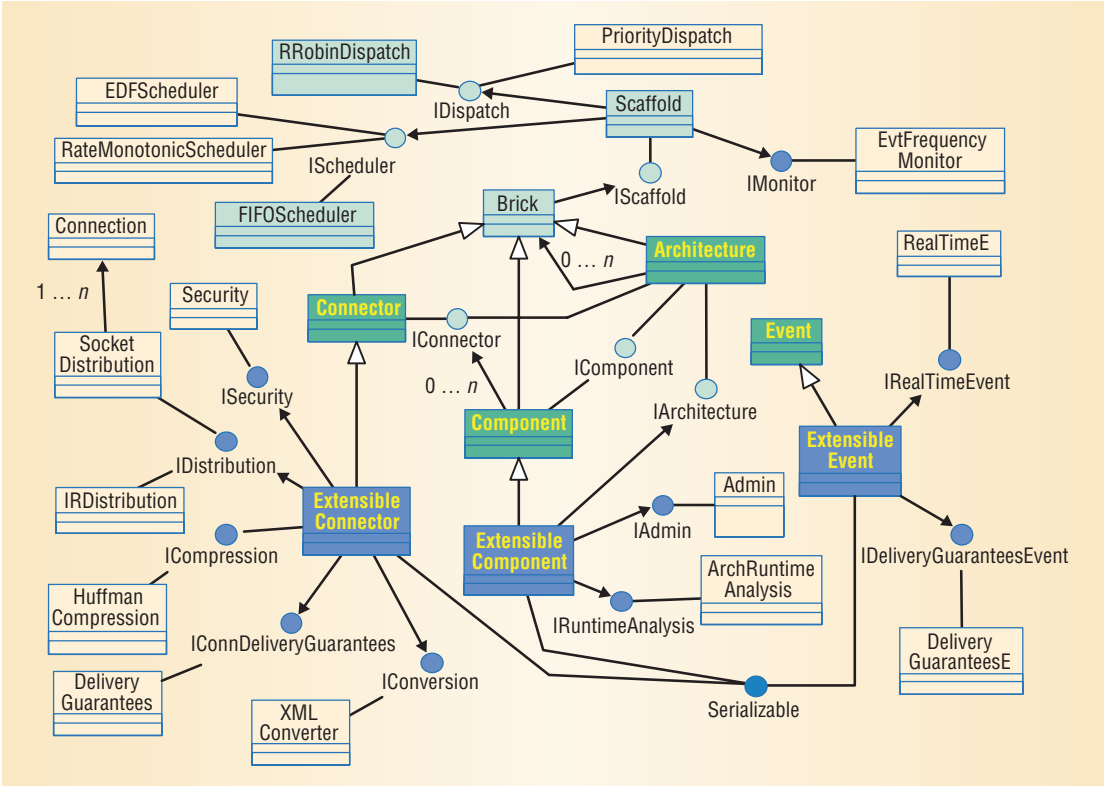

Figure 3. Unified Modeling Language class design view of Prism-MW. The green-colored classes constitute the middleware core, with those shaded dark green relevant to an application developer. Each of the blue-colored specialized classes compose a number of interfaces.

connectors and provides facilities for adding, removing, and reconnecting them, possibly at system runtime. In Prism-MW, a programmer implements a distributed application as a set of interacting Architecture objects. Components communicate by exchanging Events, which are routed by Connectors. To support extensibility to different architectural styles, the programmer can attach each component to an arbitrary number of connectors.

To further support extensibility, each Brick subclass has an associated interface. IArchitecture exposes a weld method with different implementations to accommodate style-specific composition rules. IComponent exposes send and handle methods for exchanging events with different implementations that support both asynchronous and synchronous unicast, multicast, and broadcast of events. IConnector provides a handle method for event routing; we have implemented two versions of this interface to support symmetric and asymmetric interaction. Each Architecture object implements both IConnector and IComponent, thereby allowing construction of components and connectors with internal architectures.

Prism-MW associates IScaffold with every Brick. Scaffolds schedule events for delivery via IScheduler and pool threads via IDispatch in a decoupled manner. IScaffold also directly aids architectural awareness[9] by allowing a programmer to probe a Brick's runtime behavior. Prism-MW's core provides a default FIFO implementation of IScheduler and a default round-robin implementation of IDispatch. This separation makes it possible to select the most suitable event-scheduling policy independently of the dispatching policy. In addition, decoupling dispatching and scheduling from Architecture makes it easy to compose many subarchitectures in a single application.

Extending Prism-MW

To support extensibility and configurability, Prism-MW provides explicit constructs and interfaces that accommodate different architectural styles. The extensions to the Prism-MW shown in Figure 3 provide support for numerous properties identified as relevant in Prism and other settings including architectural awareness, real-time computation, distribution, security, heterogeneity, data compression, delivery guarantees, and mobility.[2,8,10]

Prism-MW's core does not change. The core constructs Component, Connector, and Event are subclassed via specialized classes—ExtensibleComponent, ExtensibleConnector, and ExtensibleEvent, respectively—each of which composes a number of interfaces, shaded in blue in Figure 3. Each interface can have multiple implementations, shown as unshaded classes, to facilitate selecting the desired functionality inside each instance of a given Extensible class.

If an interface is installed in a given class instance, that instance will exhibit the inherent behavior in the interface's implementation. A programmer can install multiple interfaces in a single Extensible class instance. In that case, the instance will exhibit the installed interfaces' combined behavior. Each new

interface implementation requires a minimal change to the corresponding Extensible class, averaging three new lines of code.

Assessing Prism-MW

Prism applications frequently run on resource-constrained devices that have low amounts of memory and slow processing speeds. Techniques for ensuring efficient implementations of distributed systems are available, but Prism-MW presents unique challenges because it is designed to directly support architectural abstractions in highly resource-constrained settings.

Prism-MW incorporates several optimization techniques[10] including event routing based on the architectural topology and a centralized event queue with an adjustable thread pool per each address space (Prism-MW Architecture object). The result is an efficient architectural middleware that introduces minimal overhead in dynamic memory usage, is highly scalable, and exhibits good performance.

RUNTIME SUPPORT

Prism-MW provides the foundation for building a number of advanced capabilities needed for Prism applications. We have focused on automated deployment, dynamic reconfigurability, and mobility by directly leveraging Prism-MW's support for incremental system composition. Connectors can dynamically add and remove communication ports as the programmer invokes the weld and unweld methods on their container Architecture object to attach and detach components. To implement this support, the connectors leverage programming language or OS facilities—for example, Java's dynamic class loading or Windows' dynamically linked libraries.

Automated deployment

Our support for deployment directly leverages Prism-MW's incremental system composition. Prism-MW's light weight is critical for resource-constrained devices. Therefore, we have developed a custom solution for deploying applications instead of trying to reuse existing capabilities such as the software dock.[11] To deploy a desired configuration on a set of hosts, Prism-MW uploads a skeleton configuration on each host consisting of an Architecture object. This object contains an ExtensibleConnector that implements mechanisms for communicating across the network.

As Figure 3 shows, Prism-MW uses sockets and infrared ports to implement these connectors. In addition, the skeleton configuration contains AdminComponent, a special-purpose, metalevel ExtensibleComponent tasked with effecting runtime changes on the Architecture. The skeleton configuration, which is less than 20 Kbytes, can directly exploit the Architecture object's API and dynamic connector interfaces to instantiate a local subsystem architecture. With the help of ExtensibleEvent's serializable interface, Prism-MW migrates components across hosts for deployment as (usually large) events.

Prism-MW directly supports architectural abstractions in highly resource-constrained settings.

Dynamic reconfigurability and mobility

Dynamic reconfigurability encompasses runtime changes to a system's configuration by adding and removing components and connectors. If a subsystem needs components and connectors that are not available locally, it must be able to migrate them from remote hosts. To support code mobility at runtime, Prism-MW uses the same basic technique as for deployment: AdminComponents exchange events that contain mobile code. For example, if a Commander device needs to assume the role of Headquarters in the TDS application, it can use Prism-MW's support for mobility to migrate the HQ-specific components and then use the local Architecture object's weld method to attach the components to the appropriate connectors in the Commander subsystem.

Prism-DE[12] is an architectural deployment and mobility environment that integrates Microsoft Visio with Prism-MW. The environment contains several toolboxes for specifying a configuration of hardware devices, OS processes, and software components and connectors for deployment. A simple button click deploys a Prism-DE software configuration onto the depicted hardware configuration. Likewise, dragging a component from one depicted process to another initiates Prism-DE component mobility.

OTHER APPLICATIONS

To date, we have developed more than a dozen applications designed using various instances of Prism-SF, implemented on top of Prism-MW, and dynamically reconfigured and redeployed using Prism-DE. These applications involve traditional desktop platforms, PalmOS and WindowsCE devices, digital cameras, and motion sensors. In addition to several variants of TDS, they include distributed digital image processing, map visualization and navigation, location tracking, and instant messaging for handheld devices.

For example, the Attention System with Multiple Cameras (ASMC) detects changes in different phys-

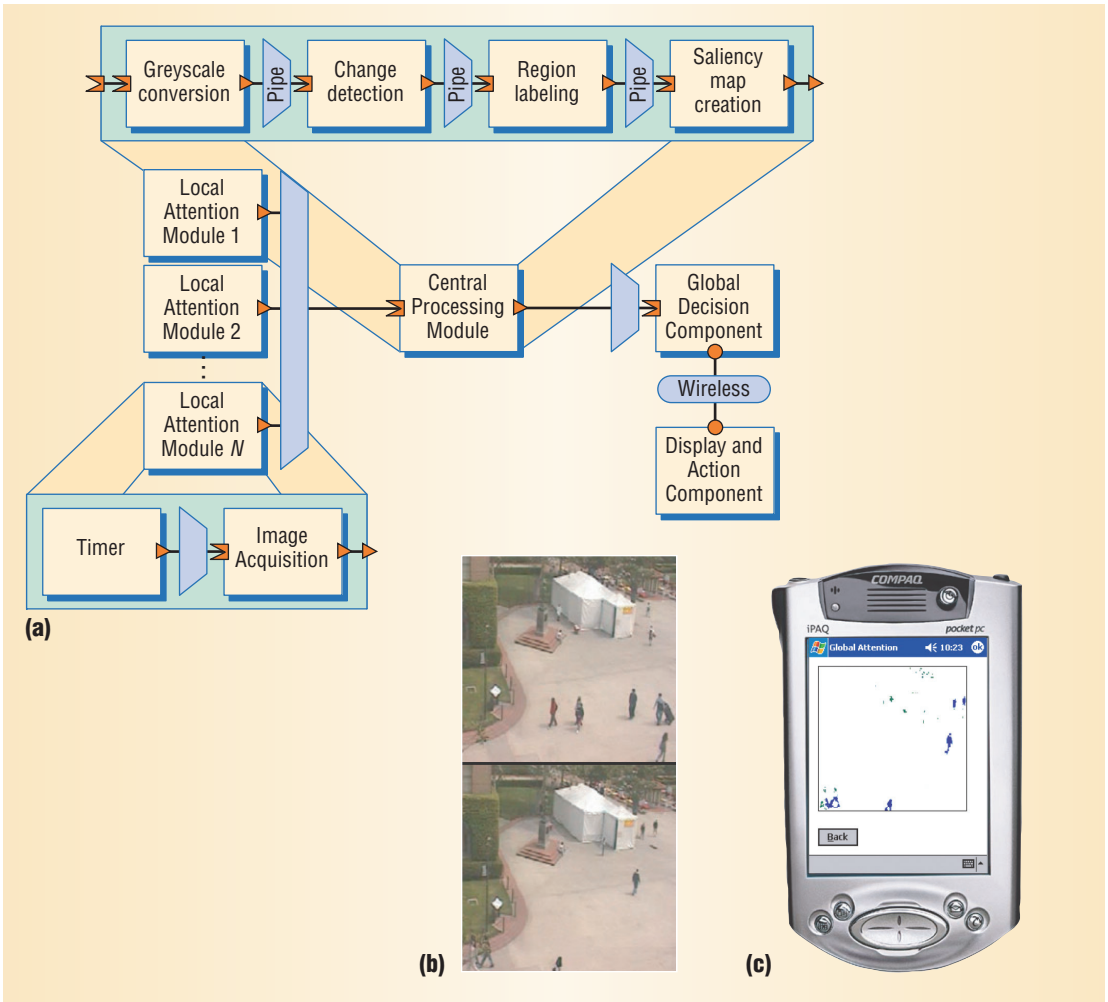

Figure 4. Attention System with Multiple Cameras. (a) The internal architectures of each LocalAttentionModule and the CentralProcessingModule are highlighted. (b) Two consecutive images taken by a camera located in a single LocalAttentionModule. (c) The resulting feature map sent via a wireless peer ExtensibleConnector to a PDA.

ical locations and executes follow-on actions such as activating an alarm in a surveillance system. As Figure 4a shows, each LocalAttentionModule observes a given location periodically and sends the obtained image to the CentralProcessingModule, which resides on a desktop device. A Timer triggers ImageAcquisition at each site. The CentralProcessingModule performs grayscale conversion, detects changes, labels the regions, and creates a saliency map. The GlobalDecisionComponent, which can reside on either a desktop or a handheld device, reports significant changes to any number of Display&ActionComponents, each of which also can reside on a desktop or handheld device.

Figure 4b shows two consecutive images captured by a single LocalAttentionModule camera. ASMC sends the selected feature map—an image containing the changes processed by the CentralProcessingModule—to the GlobalDecisionModule, which in turn forwards the most significant change to the Display&ActionComponent—in this case, the PDA shown in Figure 4c.

We used a combination of three architectural styles to design and implement ASMC: push-based inside each LocalAttentionModule, pipe and filter inside the CentralProcessingModule, and peer-to-peer between the GlobalDecisionComponent and Display&ActionComponent.

In concert, the Prism architectural style framework, Prism-MW, and Prism's runtime support provide efficient, scalable, and extensible capabilities for handheld computing. In collaboration with two major industrial organizations, we have successfully tested and evaluated our approach on several applications. We have also used it as an educational tool in courses on software architectures and embedded systems at the University of Southern California.

Our experience thus far has been very positive, but many pertinent research questions remain. Future work will span issues such as evaluating the applicability of different Prism styles to various problem domains, adding pluggable architectural-style constraint checking into Prism-MW, and extending runtime support for Prism to include different aspects of application self-healing—for example, in the face of network disconnections. ■

Acknowledgments

We thank R. Banwait, M. Bhachech, V. Jakobac, V. Kudchadkar, A. Rampurwala, E. Sanchez, V.

Viswanathan, and students from the University of Southern California's CSCI 578 and 599 courses who contributed to the development of Prism-MW and various applications on top of it. We also thank G. Sukhatme for his insights on handheld, mobile, and embedded computing. Barry Boehm and Richard N. Taylor generously provided support in obtaining the initial equipment used in our research. The work described in this article was supported by an equipment grant from Intel and is partly based upon research supported by the US National Science Foundation under grant no. CCR-9985441, DARPA/Rome Laboratory under agreement F30602-00-2-0615, and US Army TACOM.

References

1. F. DeRemer and H.H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Trans. Software Eng.,* June 1976, pp. 80-86.
2. E.A. Lee, "Embedded Software," *Advances in Computers,* M. Zelkowitz, ed., vol. 56, Academic Press, 2002.
3. J.P. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," *Proc. 3rd Working IEEE/IFIP Conf. Software Architectures,* Kluwer Academic, 2002, pp. 29-43.
4. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes,* Oct. 1992, pp. 40-52.
5. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, 1996.
6. R.T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," PhD dissertation, School of Information & Computer Science, University of California, Irvine, 2000.
7. R. N. Taylor et al., "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Trans. Software Eng.*, June 1996, pp. 390-406.
8. G. D. Abowd, "Software Engineering Issues for Ubiquitous Computing," *Proc. 21st Int'l Conf. Software Engineering,* ACM Press, 1999, pp. 75-84.
9. L. Capra, W. Emmerich, and C. Mascolo, "Middleware for Mobile Computing (A Survey)," *Advanced Lectures on Networking,* LNCS 2497, Springer-Verlag, 2002, pp. 20-58.
10. M. Mikic-Rakic and N. Medvidovic, "Adaptable Architectural Middleware for Programming-in-the-Small-and-Many," *Proc. ACM/IFIP/USENIX Int'l Middleware Conf.,* LNCS 2672, Springer-Verlag, 2003, pp. 162-181.
11. R.S. Hall, D.M. Heimbigner, and A.L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," *Proc. 21st Int'l Conf. Software Engineering,* ACM Press, 1999, pp. 174-183.
12. M. Mikic-Rakic and N. Medvidovic, "Architecture-Level Support for Software Component Deployment in Resource Constrained Environments," *Proc. Component Deployment, IFIP/ACM Working Conf.,* LNCS 2370, Springer-Verlag, 2002, pp. 31-50.

Nenad Medvidovic *is an assistant professor in the Computer Science Department at the University of Southern California's School of Engineering and a faculty member of the USC Center for Software Engineering. His research focuses on software architecture modeling and analysis; middleware facilities for architectural implementation; product-line architectures; architectural styles; and architecture-level support for software development in highly distributed, mobile, resource-constrained, and embedded computing environments. Medvidovic received a PhD in information and computer science from the University of California, Irvine. He is a member of the IEEE, the ACM, and ACM SIGSOFT. Contact him at neno@usc.edu.*

Marija Mikic-Rakic *is a PhD candidate in the Computer Science Department at the University of Southern California's School of Engineering. Her research interests are in the field of software architectures. Mikic-Rakic received an MS in computer science from the University of Southern California. She is a member of the ACM and ACM SIGSOFT. Contact her at marija@usc.edu.*

Nikunj Mehta *is a PhD candidate in the Computer Science Department at the University of Southern California's School of Engineering. His research focuses on software architectures, particularly styles and primitives. Mehta received an MS in computer science from the University of Southern California. He is a member of the IEEE, the ACM, and ACM SIGSOFT. Contact him at mehta@usc.edu.*

Sam Malek *is a graduate student in the Computer Science Department at the University of Southern California's School of Engineering. His research interests are in software architecture and engineering. Malek received a BS in information and computer science from the University of California, Irvine. He is a member of the ACM and ACM SIGSOFT. Contact him at malek@usc.edu.*