Component Oriented Human Machine Interface for In-vehicle Infotainment Applications

Hemant Sharma, Dr. Roger Kuvedu-Libla, and Dr. A. K. Ramani

Abstract—**The growing complexity of In-vehicle infotainment HMI software requirements calls for the design of reusable software components, the synthesis and generation of software code. The infotainment systems are re-organizing in response to new or changing conditions in the environment. The need for self organization is often forced in user interface of infotainment applications; which are typically hosted in resource-constrained environments and may have to dynamically reorganize in response to changes of user needs, to heterogeneity and connectivity challenges, as well as to changes in the execution context and physical environment.**

In this paper, we present a lightweight component model for HMI framework, which represents an Infotainment HMI application as a set of interoperable local components. The model supports reconfiguration, by offering code migration services. We discuss an implementation of the core of HMI framework, based on the component model and evaluate our prototype.

***Index Terms*— Component Model, In-vehicle Infotainment Systems, Human Machine Interface (HMI), Logical Mobility.**

I. Introduction

Today's cars provide an increasing number of new functionalities that enhance the safety and driving performance of drivers or raise their level of comfort. Passive safety systems such as airbags were among the first to be integrated. More recently, systems for active cruise control [4] have been added, addressing aspects of active safety and directly influencing the driving process itself. Further, systems, such as night vision, will extend the drivers' sensing capabilities. In addition, Infotainment systems of general interest have become increasingly common, providing luxury facilities or enabling modern communication mechanisms. With every new generation of cars, there are more built-in infotainment functions.

Infotainment system manufacturers seeking to serve worldwide markets face a complex matrix of 15 major OEMs (car companies) with 78 major marks (brands) and perhaps as many as 600 vehicle models. Models may each have a number of trim levels; serve multiple languages and regulatory regimes (countries) [2]. To compete, the suppliers will also need to deliver new functionality; features and a fresh new look on an annual basis. This makes for an extremely complex business model and requires a mammoth engineering effort. Building Human Machine Interface (HMI) for these complex applications is difficult and code-intensive, consuming disproportionate amount of resources and adding considerably to project risk.

The current state-of-practice for developing automotive software for Infotainment and Telematics systems offers little flexibility to accommodate such heterogeneity and variation. Currently, application developers have to decide at design time what possible uses their applications will have and the applications do not change or adapt once they are deployed on an infotainment platform. In fact, In-vehicle infotainment applications are currently developed with monolithic architectures, which are more suitable for a fixed execution context.

Recent trends in automotive software design indicate that the use of prefabricated building blocks for software development is on the rise. The prefabricated artifacts are the off-the-shelf (COTS) software infrastructure and domain-specific service components that one can acquire from different vendors and integrate them to deploy large-scale software applications [1, 3]. Vehicle Navigation is a good example of such an application on In-vehicle Infotainment systems.

In this work, we exploit logical mobility [21] and components to offer self organization to Infotainment HMI applications. Logical Mobility is defined as the ability to ship part of an application or even to migrate, a complete process from one host to another. Logical mobility primitives have been successfully used to enhance a user's experience (Java Applets), to dynamically update an application (Anti- Virus software etc.), to utilise remote objects (RMI [24], CORBA [26], etc), to distribute expensive computations (Distributed.net [25]) etc. Component Models on the other hand, argue for the decoupling of a system into a set of interacting components with well defined interfaces. Components promote decomposition and reusability of software. There are numerous component models already

Hemant Sharma is Software Engineer at Delphi Delco Electronics Europe GmbH, Bad Salzdetfurth, Germany.
(e-mail: hemant.sharma @ delphi.com).
Dr. Roger Kuvedu-Libla is EMC-Competency-Leader at Delphi Delco Electronics Europe GmbH, Bad Salzdetfurth, Germany.
(e-mail: roger.kuvedu.libla @ delphi.com).
Dr. A. K. Ramani, is Professor at School of Computer Science, Devi Ahilya University, Indore, INDIA. (e-mail: ramani.scs@dauniv.ac.in).

developed and discussed in the literature [5,6,7,8,9,10], offering various services such as transactions and concurrency control and which have been used to represent systems as a collection of either local or remote components. The novel contribution of this paper is threefold: We argue for the advantages that self organization brings to automotive infotainment software systems and how this compares to other approaches. We develop and discuss a lightweight component model that uses logical mobility to offer self organisational abilities to user interface for infotainment systems. Finally, we present implementation of core of HMI framework based on the component model and evaluate it.
The paper is structured as follows: In the following section an overview of related research is provided. Section 3 describes the component model for HMI framework. Section 4, gives an overview of architecture of core of HMI framework. In section 5, we describe the HMI framework prototype and summarize its performance. Section 6 describes an example HMI based on the framework. In section 7, we elaborate the future activities and finally conclude the paper.

II. Background and Motivation

Drivers of vehicles operate in highly dynamic environments or contexts. Existing context aware systems use context such as task at hand, location, user preferences and device capabilities [11, 12, 13] to deliver relevant information to the user. The relevance of the information is relative to a particular circumstance or context.

An In-vehicle Infotainment device is usually connected to the vehicle network as well as to GPS network. Further it not only, may have WiFi or cellular connectivity, but also interface to various ad hoc networks using the infrared or Bluetooth interfaces. The potential for interaction with its environment is great. However, the system only provides limited HMI primitives for this. The result is that such devices are still seen as stand-alone and independent system, which interacts mainly to offer static services—interaction with their environment and peers is either not considered or is very limited. Thus, although physically mobile, they are logically static systems.

This HMI interaction model in current infotainment systems has various disadvantages: There is little code sharing between applications running on the same device. There is no framework providing higher level interoperability and communication primitives for HMI service applications running on different devices. HMI Applications are monolithic, composed of a single static interaction interface, which makes it impossible to update part of HMI structure. The procedure needed to host third party dynamic service applications is difficult.

A component-based approach using logical mobility primitives would have several advantages:

- Decomposition of applications as interoperable components would allow for updating individual parts, rather than replacing the application completely.
- Componentization would promote sharing of implementations at runtime, which preserves limited resources of mobile devices.
- Logical mobility primitives would facilitate discovery and retrieval of components existing on any host that is in reach, in a peer-to-peer fashion.

A component model could support the removal of infrequently used components when the system is running out of resources. The components could be transparently retrieved from peers or a centralized host when needed again.

There is a substantial body of work on self-organizing, self healing, and adaptable systems, component deployment and middleware systems.

Beanome [15] and Gravity [16] are component models built on top of the Open Services Gateway Initiative (OSGi) Framework [17]. OSGi is a commercial framework for the Java platform that allows service providers to deliver services to consumer devices attached to a residential network and to manage those devices remotely. DACIA [20] is an adaptable distributed component based system for groupware applications that allows for the reconfiguration of the system in the event of user mobility.

Lime [21] is a mobile computing middleware system that allows mobile agents to roam to various hosts sharing tuple spaces. PeerWare [23] allows mobile hosts to share data, using logical mobility to ship computations to the remote sites that host the data.

The FarGo-DA [22] distributed component model uses logical mobility to allow disconnected operations. As such, when a FarGo component is disconnected, it has a number of options to allow the remote reference to remain valid.

III. Component Model

The term component model refers to a description of components and a component infrastructure that abstracts from the details of a concrete implementation, such as the exact format of the component executable. The goal in doing so is to understand the choices that can be made in the design of component architecture, without getting distracted by machine-specific or platform specific details. This is especially important in automotive systems, as it is believed that the diversity of process and architectures will mean that many different implementations of any given component model will be needed.

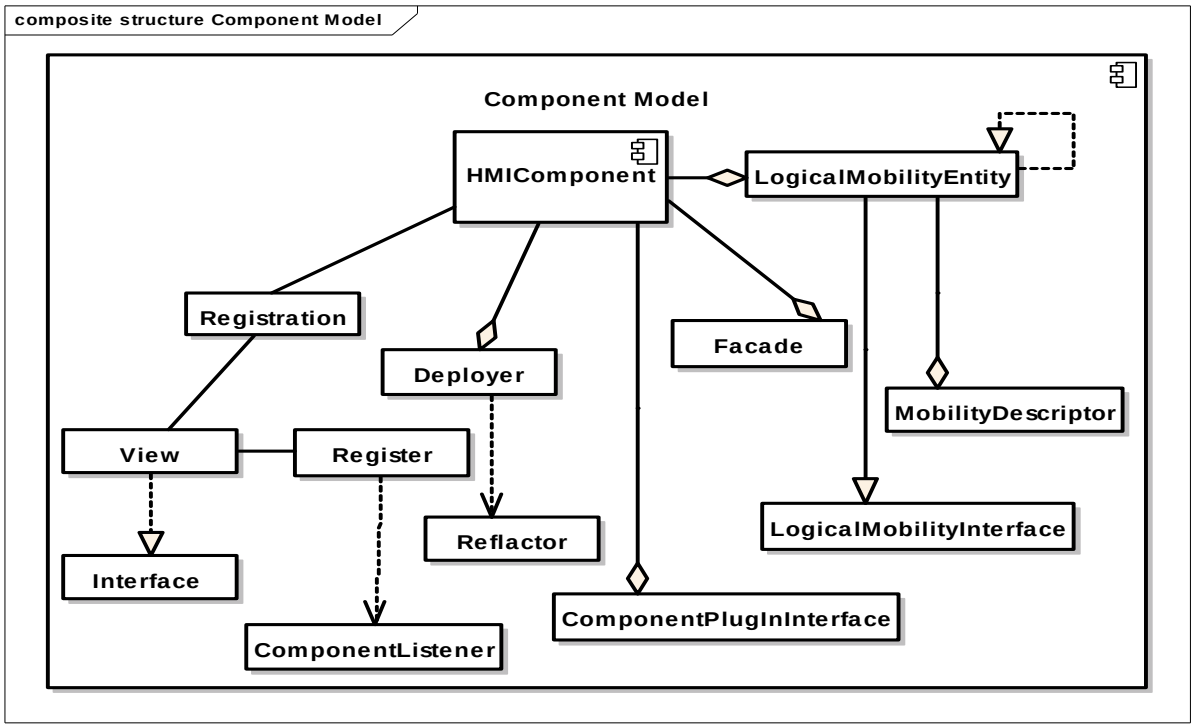

Figure 1: HMI Framework Component Model

A. Component Meta-Model Overview

The component metamodel, as shown in Fig. 1, is a Meta Object Facility (MOF) [18]-compliant extension of the UML metamodel [19]. It builds upon and extends the UML concepts of Classifier, Node, Class, Interface, Data- Type, and Instance. The most novel aspect of the component model is the way in which it offers distribution services to local components, allowing instances to dynamically send and receive components at runtime.

The component metamodel is a local, or in process, reflective component metamodel for HMI applications hosted on infotainment platforms. The model uses logical mobility primitives to provide distribution services and offers the flexible use of those primitives to the applications; instead of relying on the invocation of remote infotainment services via the vehicle network. The HMI framework components are collocated on the same address space. The model supports the remote cloning of components between hosts, providing for system autonomy when application service connectivity is missing or is unreliable. As such, an instance of HMI framework, as part of HMI application, is represented as a collection of local components, interconnected using local references and well-defined interfaces, deployed on a single host. The model also offers support for structural reflection [15] so that applications can introspect which components are available locally, choose components to perform a particular task, and dynamically change the system configuration by adding or removing components.

B. Component Model Elements

Components

The framework components encapsulate particular functionality, such as, for instance, a user interface, a service advertisement protocol, a service, a graphics framework, or a widget library. The components separate interfaces and implementations. A component is implemented by one or several HMI framework classes. It can implement one or more interfaces, called facades (a term inherited from the CORBA component model [26]), with each facade offering any number of operations. A metamodel for components that are going to be deployed across autonomous domain boundaries needs to ensure that interfaces that have once been defined cannot be changed.

Each framework component implements at least one facade, the Component façade [14]. The purpose of this facade is to allow an application to reason about the component and its attributes. This permits access to the properties of the component by retrieving, adding, removing, and modifying attributes. The component facade also contains a constructor, which is used to initialize the component, and a destructor, which is used when removing the component from the system.

Containers

The central component of every HMI application is the container component. A container is a component specialization that acts as a registry of components installed on the system. As such, a reference to each component is

available via the container. The container component implements a specialization of the component facade that exports functionality for searching components that match a given set of attributes.

An adaptive system must also be able to react to changes in component availability. For example, a *media player* interface for *iPOD* must be able to reason about which streams it can decode. Hence, the container permits the registration of listeners (represented by components that implement the *ComponentListener* facade) to be notified when components matching a set of attributes given by the listener are added or removed.

To allow for dynamic adaptation, the container can dynamically add or drop components to and from the system. Registration and removal of components is delegated to one or more registrars. A registrar is a component that implements a facade that defines primitives for loading and removing components, validating dependencies, executing component constructors, and adding components to the registry.

C. Distribution and Logical Mobility

An HMI application built using the framework can reconfigure itself by using logical mobility primitives. As different paradigms can be applied to different scenarios, our metamodel does not build distribution into the components themselves, but it provides it as a service; implementations of the framework metamodel can, in fact, dynamically send and receive components and employ any of the above logical mobility paradigms.

We consider four aspects of Logical Mobility: Components, Classes, Instances, and Data Types; the last is defined as a bit stream that is not directly executable by the underlying architecture [21]. One such, the Logical Mobility Entity (LME), is defined as an abstract generalization of a Class, Instance, or Data Type. In the framework component metamodel, an LMU is always deployed in a Reflective component. A Reflective component is a component specialization that can be adapted at runtime by receiving LMUs from the framework migration services. By definition, the container is always a reflective component, as it can receive and host new components at runtime.

D. Component Life Cycle

The HMI framework supports a very simple and lightweight component life cycle. When a component is passed on to the container for registration by loading it from persistent storage, using a *Deployer*, etc., the container delegates registration to a registrar component. The registrar is responsible for checking that the dependencies of the component are satisfied, instantiating the component using its constructor, and adding it to the registry. Note that the component facade prescribes a single constructor. An instantiated component can use the container facade to get references to any other components that it may require. A component deployed and instantiated in the container may be either enabled or disabled. The semantics of those and the initial state of the component depend on the component implementation. The functionality needed to manipulate the state of the component is exported by the component facade.

IV. Architecture of Core HMI Component Framework

The aim of HMI Framework core in general is to provide higher level interaction primitives than those provided by the vehicle infotainment network and infotainment service system as a layer upon which HMI applications are then constructed. In doing so, the framework hides the complexities of addressing distribution, heterogeneity, and failures.

The core uses the adaptation primitives defined by the component model to build a flexible and adaptable platform for flexible user interface development. Hence, while describing the design of the core, we also validate the metamodel by showing how it can be used to build a complete HMI application, which offers dynamically adaptable services.

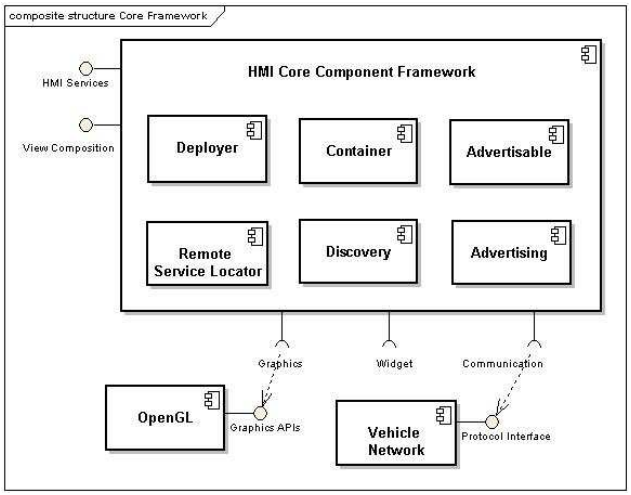

Figure 2: Core HMI Architecture Overview

The system is built on top of the OSGi middleware and provides an instance of the component container, as defined in Section 3. This container is the central aspect of every instance of the middleware system. Registered with the container are all the components that are part of the system. This includes application components (such as a Navigation application), libraries (such as route calculation), and system services (such as any registrars, deployers, service advertising, and discovery components, etc.). All components make their dependencies explicit through their properties. The core of every HMI application is the container, with every other service (including logical mobility) or application components built on top of it. Thus, even though components may build complex dependency graphs expressed via their properties, to the container, they

all implement components facades. Components can be added and removed at runtime.

The framework core along with underlying middleware provides a number of services to applications. The services themselves are seen as regular components built on top of the container. As such, they can be dynamically added and removed. In the following paragraph, we outline how the metamodel primitives are used to provide adaptable advertising and discovery services for framework components.

Components that wish to advertise their presence to the environment must implement the Advertisable facade. Examples of advertisable components include repositories, services, etc. The Advertisable facade exports a method that returns a message that is used for advertising; thus, the advertising message allows the Advertisable component to express information that it requires to be advertised. An advertising technique is represented by an Advertiser component, which is a component implementing the Advertiser facade. An advertiser component is responsible for accepting the message of advertisable components, potentially transforming it into another format and using it to advertise them. An advertiser allows components that wish to be advertised to register themselves with it to be advertised. The combination of component availability notification and advertiser registration allows an advertisable component to register with the container to be notified when specific advertisers are added to the system. The advertisable component can then register to be advertised by them.

V. PROTOTYPING AND EVALUATION

The prototype for core components of HMI Framework has been implemented using Java2 Micro Edition on the Top of *OSGi framework* [17]. The core is structured in three primary layers:

A. Interface Layer

It includes a graphical framework plug-in, service-based resources locator, and extensibility mechanisms such as plug-ins components and extension interfaces.

B. Composite Layer

The components of this layer offers application service patterns, design-to-implementation mappings, platform profiles, and extensibility mechanisms for interoperating with other infotainment applications.

C. Communication Layer

Communication layer components provide plug-in interface for distributed resources. They are responsible for dynamically registering and activating the vehicle network specific message handling protocol.

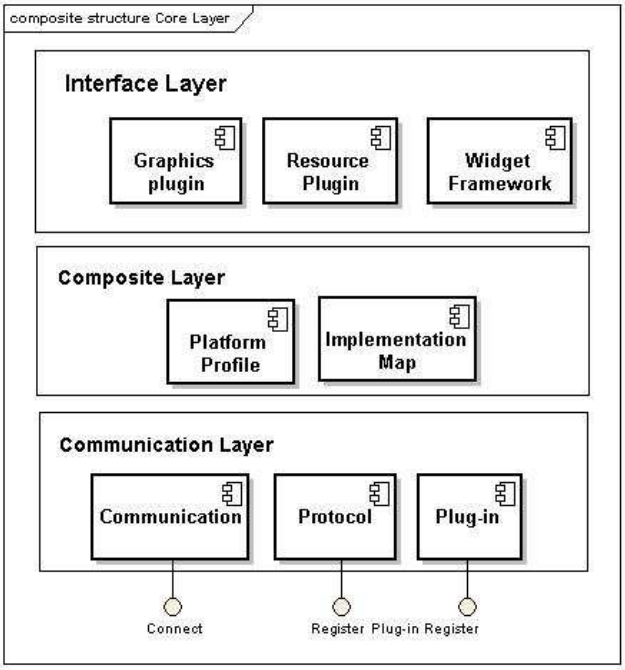

Figure 3: Layered Component View for Core of HMI Framework

Implementation of core of the framework occupies 254 kilobytes, as a compressed Java archive, and includes a launcher implementation, multicast and centralised publish/subscribe advertising and discovery components and numerous plug-in interfaces. The core contains the minimal components to design an HMI view, handle interactions on the view and establish connection between view and desired infotainment service. The core has been deployed on ARM 9 based OMAP platform along with a test HMI view to access raw GPS data. The table below summarizes the memory usage and performance figures.

Application Start-up Time	9 Seconds
Memory usage for Application	820 KB
Time to connect to GPS server	1400 ms
Time to Update HMI View	100 ms
Time to register for GPS data	657 ms

Table 1: Performance Figures

VI. THE TRAFFIC MESSAGE HMI

We have implemented a simple Traffic Message Notification application using the HMI framework core. Components that implement Traffic Message decoding and text presentation inherit message format from the radio tuner running on the same platform. As such, the Traffic HMI application uses the notification service to be notified whenever the tuner façade component that has Traffic Message attribute implemented is registered. Moreover, it uses the deployer and

the discovery components to premium Traffic message service that are found remotely. The application itself occupies 96 kilobytes as a compressed Java archive.

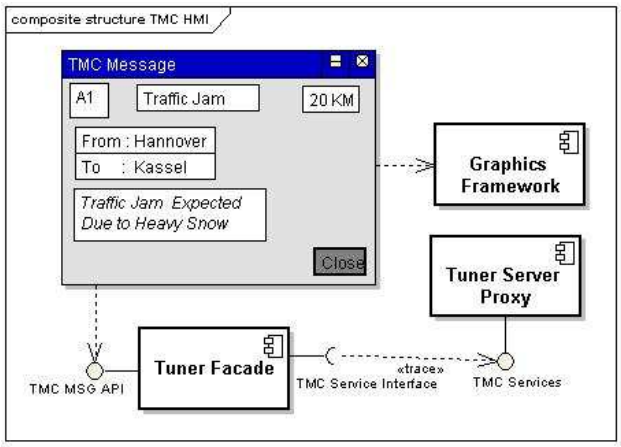

Figure 4: Traffic Message HMI Application Overview

The Traffic Message HMI demonstrates an application that uses the container to listen to the arrival of new components, adapting its interface and functionality upon new TMC service component arrival. It also demonstrates reaction to context changes, as the application monitors the discovery services for new service components and schedules them for use as soon as they appear.

VII. Conclusion

In this paper, we present an approach for building HMI for In-vehicle infotainment software applications by means of components. We enable this by using a logical mobility based component model for HMI framework. The lightweight component metamodel is instantiated as a framework system for adaptable HMI application and systems. The framework offers logical mobility primitives as first-class citizens. .

The performance of prototype version of HMI framework shall be running a test HMI application found to be adequate and the time needed to adapt was measured to be minimal. An alternative approach to using components and logical mobility would be to create a programming language that allows the specification of modular systems but that also offers built-in logical mobility primitives.

To extend this work, we are looking into:

- addressing the performance deficiencies of the component communication mechanism,
- investigating the policy issues (with regard to security) raised by the extra granularity and autonomy introduced by the component system,
- Finally providing full implementations of a component API in a Java and C++ languages.

References

[1] B. Hardung, T. Kölzow, A. Krüger: “Reuse of Software in Distributed Embedded Automotive Systems”. Proc. EMSOFT, 203-210, 2004
[2] J. Dannenberg, C. Kleinhans: “The Coming Age of Collaboration in the Automotive Industry”, Mercer Management Journal 18:88-94, 2004.
[3] I. Krüger, E. Nelson. K.V. Prasad: “Service-based Software Development for Automotive Applications”. Proc. CONVERGENCE 2004, 2004.
[4] BMW. ACC. http://www.bmw.co.za/products/acc/default.asp.
[5] C. Szyperski. Component Software. Addison-Wesley, 1998.
[6] M. Völter. A Generative Component Infrastructure for Embedded Systems. http://www.voelter.de/data/pub/SmallComponents.pdf 2003.
[7] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition. Addison-Wesley and ACM Press (2002)
[8] C. Pahl. A Pi-calculus based Framework for the Composition and Replacement of Components. In Proc. OOPSLA Workshop on Specification and Verification of Component -based systems, 2001.
[9] Bachman, F., Bass, L., Buhman, S., Comella-Dorda, S., Long, F., Seacord, R.C., and Wallnau, K.C. Technical Concepts of Component -Based Software Engineering. Tech. Rep. CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.
[10] D. Schmidt, The ACE ORB. http://www.cswustl.edu/~schmidt/TAO.html.
[11] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste, Project Aura: Toward distraction-free pervasive computing, IEEE Pervasive computing (2002), 22–31.
[12] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy, Modeling context information in pervasive computing systems, 1st International Conference on Pervasive Computing (Zurich, Switzerland), Springer, August 26-28 2002, pp. 167–180.
[13] Patil, S. and J. Lai. Configuring Privacy Preferences in an Awareness Application. In Proceedings of CHI 2005.
[14] S. Zachariadis and C. Mascolo. Adaptable mobile applications through satin: Exploiting logical mobility in mobile computing middleware. In 1st UK-UbiNet Workshop, September 2003..
[15] H. Cervantes and R. Hall, “BEANOME: A Component Model for the OSGi Framework,” Software Infrastructures for Component-Based Applications on Consumer Devices, Sept. 2002.
[16] H. Cervantes and R. Hall, “Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model,” Proc. 26th Int’l Conf. Software Eng. (ICSE ’04), pp. 614-623, May 2004.
[17] The OSGi Framework, OSGi Alliance, http://www.osgi.org, 1999.
[18] “Meta Object Facility (MOF) Specification,” Tech. report, Object Management Group, Mar. 2000.
[19] “Unified Modeling Language”, version 1.5, Object Management Group, http://www.omg.org/docs/formal/03-03-01.pdf, Mar. 2003.
[20] R. Litiu and A. Parakash, “Developing Adaptive Groupware Applications Using a Mobile Component Framework,” Proc. 2000 ACM CSCW, pp. 107-116, 2000.
[21] A.L. Murphy, G.P. Picco, and G.-C. Roman, “Lime: A Middleware for Physical and Logical Mobility,” Proc. 21st Int’l Conf. Distributed Computing Systems (ICDCS 21), pp. 368-377, May 2001.
[22] Y. Weinsberg and I. Ben-Shaul, “A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices,” Proc. 24th Int’l Conf. Software Eng., pp. 374-384, May 2002.
[23] G. Cugola and G. Picco, “Peer-to-Peer for Collaborative Applications,” Proc. IEEE Int’l Workshop Mobile Teamwork Support/Int’l Conf. Distributed Computing Systems (ICDCS ’02), pp. 359-364, July 2002.
[24] Sun Microsystems, Inc. Java Remote Method Invocation Specification, Revision 1.50, JDK 1.2 edition, October 1998.
[25] The Distributed.net Project. http://www.distributed.net.
[26] OMG. CORBA Component Model. http://www.omg.org/cgi-bin/doc?orbos/ 97-06-12, 1997.