The Journal of Systems and Software 83 (2010) 972–989

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An architecture-driven software mobility framework

Sam Malek[a,*], George Edwards[b], Yuriy Brun[c], Hossein Tajalli[b], Joshua Garcia[b], Ivo Krka[b], Nenad Medvidovic[b], Marija Mikic-Rakic[d], Gaurav S. Sukhatme[b]

[a] Department of Computer Science, George Mason University, Fairfax, VA, USA
[b] Computer Science Department, University of Southern California, Los Angeles, CA, USA
[c] Computer Science & Engineering, University of Washington, Seattle, WA, USA
[d] Google Inc., 1333 2nd Street, Santa Monica, CA, USA

ARTICLE INFO

Article history:
Received 1 December 2008
Received in revised form 25 August 2009
Accepted 1 November 2009
Available online 4 November 2009

Keywords:
Software architecture
Mobility
Quality of service analysis
Robotics

ABSTRACT

Software architecture has been shown to provide an appropriate level of granularity for assessing a software system's quality attributes (e.g., performance and dependability). Similarly, previous research has adopted an architecture-centric approach to reasoning about and managing the run-time adaptation of software systems. For mobile and pervasive software systems, which are known to be innately dynamic and unpredictable, the ability to assess a system's quality attributes and manage its dynamic run-time behavior is especially important. In the past, researchers have argued that a software architecture-based approach can be instrumental in facilitating mobile computing. In this paper, we present an integrated architecture-driven framework for modeling, analysis, implementation, deployment, and run-time migration of software systems executing on distributed, mobile, heterogeneous computing platforms. In particular, we describe the framework's support for dealing with the challenges posed by both logical and physical mobility. We also provide an overview of our experience with applying the framework to a family of distributed mobile robotics systems. This experience has verified our envisioned benefits of the approach, and has helped us to identify several avenues of future work.

1. Introduction

As the global computing infrastructure transitions from an emphasis on personal computers to mobile and embedded devices, ensuring the quality of complex distributed software systems remains an essential focus of research in software engineering and, particularly, software architecture. Software quality is measured in terms of *quality attributes*, such as performance and dependability, that are identified and prioritized by system stakeholders. In a mobile environment, system parameters such as network reliability and throughput are far less predictable than in static environments. Moreover, new quality attributes such as energy consumption would also need to be taken into account in the design and construction of these systems. Thus, for systems distributed on mobile hardware devices, such as smart phones and wearable computers, evaluating software quality is even more challenging than for traditional systems.

It has long been acknowledged that software architecture provides an effective foundation for the quality assurance of large, complex systems (e.g., Abowd et al., 1995; Medvidovic and Taylor, 2000; Clements et al., 2002; Malek et al., 2007). The key underpinning of our work is the observation that an explicit architectural focus can also be instrumental in facilitating mobile computing (Chan and Chuang, 2003; Ciancarini and Mascolo, 1998; Medvidovic et al., 2003; Malek et al., 2005b; Malek et al., 2006). Architecture-driven approaches to quality assurance use *architectural abstractions* – software components, connectors, communication ports, events, etc. – to manage complexity and leverage *architectural styles* to enforce constraints and promote desired system characteristics. Analogously, architecture-driven approaches to mobility enable system migration and adaptation during run-time in a controlled fashion by employing architectural constructs as the units of mobility.

While existing research (Chan and Chuang, 2003; Ciancarini and Mascolo, 1998; Sousa and Garlan, 2002), including our own (Medvidovic et al., 2003; Malek et al., 2005b; Malek et al., 2006), has verified the advantages of an architecture-centric approach in the development of mobile software systems, in practice, the adoption of such approaches has been limited. We argue that this is due to the lack of a comprehensive support for architecture-based development of mobile software systems. In other words, the majority

* Corresponding author.
E-mail addresses: smalek@gmu.edu (S. Malek), gedwards@usc.edu (G. Edwards), ybrun@usc.edu (Y. Brun), tajalli@usc.edu (H. Tajalli), joshuaga@usc.edu (J. Garcia), krka@usc.edu (I. Krka), neno@usc.edu (N. Medvidovic), marija@google.com (M. Mikic-Rakic), gaurav@usc.edu (G.S. Sukhatme).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.11.003

of existing architectural research approaches and industrial tools have dealt with providing point solutions that address particular mobility concerns. As a result, the developers in the mobility setting have faced some difficulties with fully embracing architectural abstractions as the foundation for modeling, analyzing, implementing, monitoring, and adapting the system. Moreover, the discrepancies between the existing tools and techniques diminish some of the key advantages associated with taking an architecture-centric approach.

To better illustrate the current shortcomings and motivate the problem, let us consider a scenario in which a software developer uses an architectural modeling tool (Childs et al., 2006; Dashofy et al., 2005; Edwards et al., 2007) to design a system and analyze its quality attributes. Since the majority of mobile middleware platforms do not provide adequate support for the implementation of architectural abstractions (Malek et al., 2005b; Malek et al., 2007), the developer is forced either to implement them through a combination of low-level programming language constructs (e.g., variables, collections, classes), or to misuse other middleware constructs (e.g., implement a connector as a middleware component). Performing such a complex mapping between constructs with different semantics and levels of granularity promotes *architectural erosion* (Perry and Wolf, 1992). In turn, the analysis performed on the architectural models becomes useless, as one cannot be certain of the fidelity of the implemented system with respect to the models.

The above example highlights only one set of problems that could arise due to the lack of complete life-cycle support for architecture-based development of mobile software systems. In this paper, we present and evaluate an integrated framework that aims to alleviate the shortcomings of the existing point solutions.[1] Specifically, our framework comprises:

- a tailorable *model* of mobile software architectural abstractions (Medvidovic et al., 2003; Edwards et al., 2007), mobile hardware platforms on which the software executes (Mikic-Rakic et al., 2004), and system quality requirements that are of particular importance to mobile systems (Mikic-Rakic et al., 2008);
- an extensible suite of architectural *analysis* techniques for mobile systems, including scenario-driven system simulations (Edwards and Medvidovic, 2008) and determination of effective deployments based on quality requirements (Mikic-Rakic et al., 2004, 2005; Malek et al., 2005a);
- a middleware platform (Malek et al., 2005b) targeted at architecture-centric *implementation* of mobile software, and an accompanying facility for stateful and stateless run-time *migration* of software components (Carzaniga et al., 1997);
- a continuous *monitoring* and architectural *awareness* methodology for detecting execution-condition changes in mobile software systems (Tisato et al., 2000); and
- a facility for *(re)deployment* and run-time *adaptation* of a software system distributed among a set of mobile hardware hosts (Malek et al., 2007; Mikic-Rakic et al., 2008).

With the exception of mobility support, about which we have only hypothesized in the context of system deployment in Mikic-Rakic and Medvidovic (2002) and Mikic-Rakic et al. (2008), the individual elements of the above framework have been published previously. This paper describes and illustrates those aspects of our framework that are pertinent to mobility. Moreover, the main contribution of our work is the manner in which they are combined to provide complete architecture-driven mobility support.

The framework is broadly concerned with the challenges mobility presents. We model the impact of physical mobility on the system's resources, such as network connectivity and battery power. We use simulation and analytical models to assess the degradation of quality attributes due to movement of devices and employ run-time adaptation to mitigate such problems. Note that since the framework has no explicit control over the actual movement of devices, we do not model the movement, but rather its impact on the system. However, if necessary, we believe the framework could be extended to model these aspects of mobility as well. We model logical mobility in terms of changes to the system's *deployment architecture* (Malek, 2007) – a representation of the system's software architecture superimposed on its hardware configuration and network topology. By adopting an architecture-based approach to development and adaptation, we avoid architectural erosion due to logical mobility. At run-time, we optimize the software system's quality attributes by finding a new deployment architecture and effecting it through logical mobility. Finally, the framework addresses other concerns in the mobile setting, such as heterogeneity of platforms and efficiency of implementation.

Our experiences with applying the framework on several mobile software systems have been very positive. For evaluation, we elaborate in detail on one such experience dealing with a family of mobile robotics systems, provide quantitative data that summarizes the results obtained in other real-world and synthesized examples, and qualitatively compare the framework with existing architectural frameworks.

The remainder of the paper is organized as follows. Section 2 details the challenges of building mobile systems and the framework's objectives in mitigating them. Section 3 provides a high-level overview of the framework, its accompanying tool suite, and how they are integrated with one another. Sections 4–8 describe the framework's support for mobility modeling, analysis, implementation, monitoring, and adaptation, respectively. Section 9 presents an overview of our experience to date with the framework, with data drawn primarily from the domain of mobile robotics. Section 10 relates this approach to existing work. We conclude the paper with the discussion of challenges that are guiding our ongoing work.

2. Challenges and objectives

As already alluded to in the previous section, mobile setting presents a number of unique software development challenges that permeate the entire software-engineering life-cycle:

Fluctuating execution context. Mobile software systems are characterized by their unknown operational profiles and fluctuating execution contexts. Since the properties of such systems (e.g., network connectivity, bandwidth, and energy consumption) constantly change at run-time and unanticipated events occur, an accurate analysis of the system's quality attributes is often not feasible at design-time.

Constrained resources. Mobile devices often have limited power, network bandwidth, processor speed, and memory. Constraints such as these demand highly efficient software systems in terms of computation, communication, and memory. They also demand unorthodox solutions, such as off-loading or migrating parts of a system to other devices.

Heterogeneity. Traditional computing increasingly relies on standard methods of representing data, computation, and communication, the best example of which is the SOA technology standards (i.e., XML, SOAP, WSDL) (Weerawarana et al., 2005). In contrast, mobile technologies remain largely proprietary. Engineer of such systems must reconcile proprietary operating systems such

[1] Note that our notion of *framework* is consistent with the term *architectural framework* as defined in IEEE 1471 and ISO/IEC 42010 standards (Maier et al., 2001; ANSI/IEEE, 2007). Our framework consists of several view points (e.g., deployment, dynamic, static), modeling languages (e.g., xADL, FSP), and is accompanied by a tool suite for specification and analysis.

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.

Peculiar infrastructure. The computing infrastructure of mobile platforms often lack certain services due to efficiency purposes. Since manufacturers cannot accurately determine a priori which capabilities are more important than others, they make ad hoc decisions that may significantly impact the development of software for those platforms. For example, Java KVM does not support noninteger numerical data types or server-side sockets. Similarly, typically employed techniques for code mobility, such as Java XML encoding, is computationally too expensive and hence rarely supported.

The above challenges have directly motivated the development of our framework. Our approach is based on five overarching principles that delineate the framework's objectives:

Efficiency of the implementation. The framework should enable the development of efficient software systems that can be deployed on resource-constrained mobile platforms.

Coping with heterogeneity. The framework should alleviate heterogeneity of both implementation (e.g., ability to deal with the variations in the system substrate) and analysis (e.g., energy consumption vs. latency) in this setting.

Flexibility and extensibility. The framework should facilitate the development of flexible software systems that can be adapted at run-time. Moreover, the framework itself should be extensible such that it can be customized to the domain-specific characteristics of each mobile application domain.

Context awareness. The framework should support context awareness to detect and react to changing conditions.

Architectural support. The key objective, and one that sets our work apart from existing research, is that the framework should support design, analysis, implementation, adaptation, and mobility at the architectural level.

3. Framework overview

Fig. 1 shows an overview of the framework's accompanying tool suite. Each of the framework's components has been realized using a combination of architecture-centric tools, which are integrated to provide comprehensive support for architecture-driven mobility. XTEAM supports modeling and analysis activities at design-time, while DeSi supports the same activities at run-time. Prism-MW is a middleware platform with extensive support for architecture-based development of mobile software systems. In this section we provide an overview of the leveraged tools and elaborate on their integration into the architecture-centric mobility framework. In the following sections, we will describe the individual components of the framework in more detail.

XTEAM assists in designing mobile software systems and XTEAM's models can be used to simulate those systems before their initial deployments. XTEAM's simulation capability assesses architectural decisions with respect to their QoS trade-offs and allows the architect to make informed decisions and select a proper architecture prior to deployment. Once the architect selects an architecture, XTEAM can generate (1) DeSi's initial underlying model for run-time analysis and deployment, (2) application-specific code to realize the models, and (3) Prism-MW configuration files to optimize the system's quality attributes.

Unlike XTEAM, DeSi furnishes a run-time model of the system that reflects the dynamic state of the architecture. In particular, the notion of software deployment (i.e., the location at which a software component executes), which has a significant impact on a mobile system's QoS, is elevated to the forefront. As further detailed in this paper, the framework models logical mobility in terms of changes to the system's deployment architecture. DeSi provides several logical-mobility-analysis algorithms that swiftly explore the large space of possible deployments and find a (near-)optimal architecture for the system. If an optimal solution is not available, DeSi's mobility analysis capability and XTEAM's simulation capability coordinate to select a preferable solution as follows: first, DeSi's algorithms find a small number of candidate deployments that may pose competing trade-offs; and second, XTEAM simulates the candidate deployments to develop a fine-grained assessment of the QoS trade-offs and select the most preferable deployment.

As depicted in Fig. 1d, a mobile system implemented on top of Prism-MW provides support for (1) monitoring through context awareness and reflection and (2) redeployment through adaptation and component migration. DeSi uses these two middleware facilities to collect run-time information from the system, assess its architecture, and, when necessary, redeploy the components.

Prism-MW is a highly configurable architectural middleware (e.g., it provides the ability to change the size of the event queue, size of the thread pool, etc.), which allows for optimization with respect to efficiency and performance. Both these properties are considered critical requirements in the mobile setting. As shown in Fig. 1, XTEAM may automatically generate the optimal configuration for a system using the middleware's configuration capabilities and the resource requirements obtained through simulation.

4. Mobility modeling

A significant focus of software engineering research has been codification of design abstractions that allow engineers to represent and reason about complex systems at a high-level. To this end, software architecture researchers have arrived at a canonical set of architectural design constructs: components, connectors, communication ports, interfaces, events, and topologies (or configurations). Furthermore, specific prescribed uses of these constructs, via design heuristics or constraints, result in architectural styles (e.g., client–server, peer-to-peer), which are the key design idioms in software engineering. These design elements and idioms have been shown to be highly useful in practice, and constitute the basis of our architectural approach to modeling mobile systems.

4.1. Design-time mobility modeling

Architectural models developed in the early phases of system design allow engineers to ensure desired properties in a system by experimenting with alternative designs and codifying design rules. Evaluation and refinement of architectural choices is much easier and less costly on a system model or prototype than on a fully constructed system. Software engineering research has produced a substantial variety of modeling and analysis technologies with different features and goals (Medvidovic and Taylor, 2000), but few of these address the domain-specific concerns of mobile systems. Our approach integrally involves explicit software architecture models and analyses for mobility. To this end, we have adapted our extensible environment, XTEAM (Edwards et al., 2007; Edwards and Medvidovic, 2008), in order to allow an engineer to:

1. represent the structure and behavior of a mobile software system's architecture,
2. associate the different software architectural elements with the mobile hardware hosts on which they will execute, and
3. provide mechanisms for applying mobility-related analysis techniques.

XTEAM provides a metamodeling environment for architects to define domain-specific extensions to the canonical set of architec-

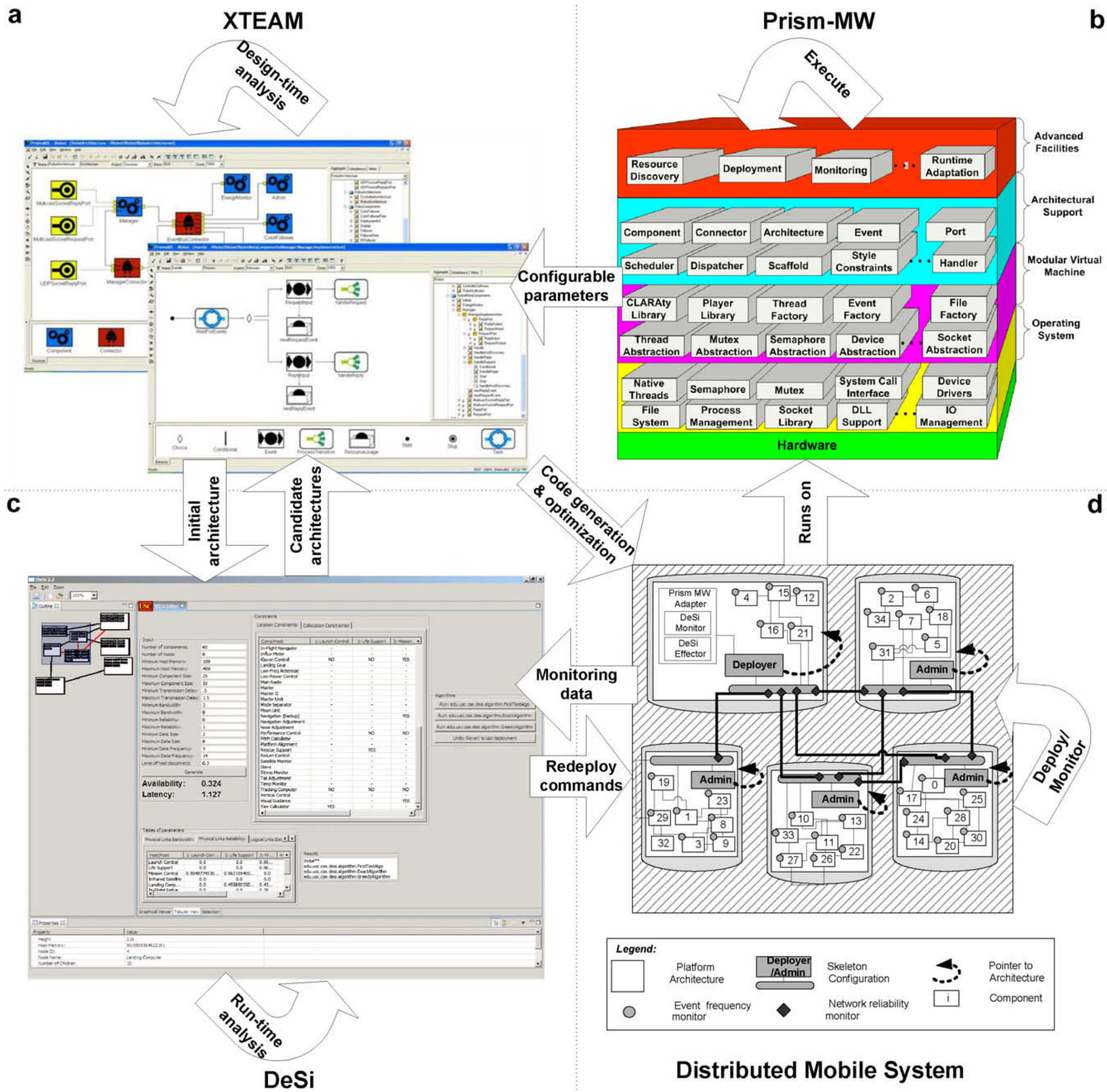

Fig. 1. Overview of the framework's accompanying tool suite: (a) XTEAM, (b) Prism-MW, (c) DeSi, and (d) a hypothetical distributed mobile software system.

tural design constructs. We have developed modeling language extensions in XTEAM that allow architects to capture information that is of particular concern for mobile systems. These language extensions, in turn, enable mobility-related analysis and, as will be detailed in Section 6.3, can also automate generation of implementation code.

Domain-specific language extensions in XTEAM may be additional properties of, or constraints on, the canonical architectural constructs or they may be entirely new constructs. For example, we found it necessary in embedded and mobile settings to extend the set of abstractions to include the notion of hardware hosts and their relationship to software components. We have also added properties and constraints to software components to ensure their compliance with the capabilities available on the target platform.

We have developed reliability, energy-consumption, and architectural-style XTEAM language extensions for mobile systems.

4.1.1. Reliability

Mobile systems operate in unpredictable environments. Wireless networks are inherently less reliable than wired networks because of such factors as interference. As a result, mobile applications must be robust in the face of network disconnection and capable of gracefully dealing with intermittent and degraded network links. The XTEAM language extension for reliability permits architects to specify the probability of disconnection, utilize stochastic models of available network bandwidth, and model mitigation strategies.

4.1.2. Energy consumption

Unlike traditional software systems that have an abundant, uninterrupted power supply, mobile devices must run on battery power. The energy usage of software components in mobile applications has a critical impact on system longevity. In order to use energy-consumption estimates (along with other concerns) in weighing design options, design models must capture system parameters required for power-usage analysis. Our XTEAM language extension for energy consumption, shown in Fig. 2, allows architects to easily include these parameters in their models. The parameters shown are defined by the energy consumption estimation technique proposed by Seo et al. (2008); we refer the reader to (Seo et al., 2008) for detailed explanations of these parameters. This technique estimates the energy consumption of a software component as the sum of its communication energy cost (due to exchanging data over a wireless network) and computational energy cost (due to manipulating data locally). Communication energy cost is a function of the size of the data exchanged and a set of platform-specific coefficients. These platform-specific coefficients are captured as properties of the host element in Fig. 2 and are used to calculate the communication energy costs for all components assigned to that host. Computational energy cost is calculated by "profiling" the service interfaces of each component to produce a characterization of the energy cost of invoking the interface. For example, some interfaces consume energy proportional to the size of the input. The properties of the Interface element in Fig. 2 capture the energy-cost characteristics of each interface.

4.1.3. Architectural style constraints

While mobility of software components enables dynamic adaptation to changing operational contexts, as well as other desirable behaviors, it must be constrained to ensure that run-time configurations remain consistent with the architects' intentions. One way our middleware platform achieves this is through architectural style-based constraints: individual architectural elements are tagged with specific roles, and constraints on their behavior are automatically enforced by the middleware based on those roles. Style-based roles may be included in architectural models using our XTEAM modeling extension for styles, and automatically generated implementations include this information so that it may be used by the middleware platform.

We have similarly implemented extensions for modeling other quality attributes of the software, such as performance, memory usage, and resource allocation. Fig. 1a depicts the structural view of a system's architecture along with the behavior of one of its components, modeled in XTEAM. The structural and behavioral views of an architecture are accompanied in XTEAM by parameter lists that represent the desired characteristics of the software (e.g., a component's anticipated memory usage) and hardware (e.g., a host's available memory) elements.

4.2. Run-time mobility modeling

A key observation underlying our framework is that mobility at the architectural level can be treated as a special case of a change to the system's *deployment architecture* (i.e., allocation of the system's software components to its hardware hosts). The deployment architecture of a software system has a significant impact on its QoS. For example, a service's latency can be improved if the system is deployed such that the most frequent and voluminous interactions among the components involved in delivering the service occur either locally or over reliable and capacious network links. Therefore, a redeployment of the software system via migration of its components may be necessary to improve its QoS.

To be able to analyze a mobile software system at run-time, one needs to model not only the system's software architecture, but also the system's execution context, which may include the hardware and network characteristics. Each of these elements may be associated with arbitrary parameters. The selection of a set of parameters to be modeled depends on the criteria (i.e., QoS objectives) that a system's deployment architecture should satisfy. Finally, the system users' usage of the functionality (i.e., services) provided by the system, and the users' QoS preferences (i.e., utility) for those services may change over time. Therefore, we also need to model the system's services, users, and users' QoS preferences. Note that our notion of a user is very general and could be interpreted as either an end-user, the software architect, or another software client.

We model a distributed software system as:

1. A set of hardware nodes (hosts) with the associated parameters (e.g., available memory or CPU on a host), and a function that maps each parameter to a value.
2. A set of software components with the associated parameters (e.g., required memory for component's execution or JVM version), and a function that maps each parameter to a value.

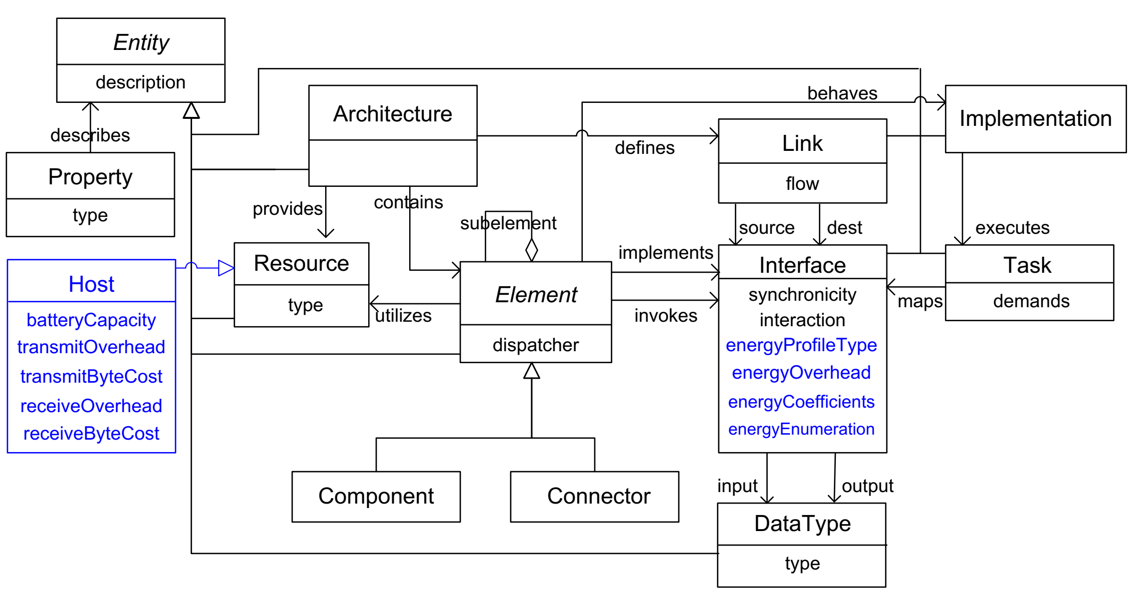

Fig. 2. A fragment of the XTEAM metamodel, with extensions that capture energy-consumption properties highlighted.

3. A set of physical network links with the associated parameters (e.g., available bandwidth, reliability of links), and a function that maps each parameter to a value.
4. A set of logical interaction links between software components in the distributed system, with the associated parameters (e.g., frequency of component interaction, average event size), and a function that maps each parameter to a value.
5. A set of services, and a function that provides values for service-specific system parameters. An example service-specific system parameter is the number of component interactions resulting from an invocation of a single service (e.g.,find the best route to the disaster area).
6. A set of QoS dimensions, and a function that quantifies a dimension (e.g., security) for a given service in the current deployment.
7. A set of users, a utility function that denotes a user's preference for a QoS dimension of a service, and a pair of threshold values that together determine the relative importance of each user.
8. A set of resource and locational constraints, and a function that, given a constraint and a deployment architecture, determines if the constraint is satisfied. An example of a resource constraint is that the total required memory for executing components on a host should not exceed the available memory on that host. A component's locational constraints specify the hardware hosts on which it can be deployed.

Some elements of the model are intentionally left "loosely defined" (e.g., system parameter sets, QoS set). These elements correspond to the many and varying factors that are found in different distributed application scenarios. We leverage DeSi (recall Fig. 1c) to specify these loosely defined elements of the model. DeSi supports specification, manipulation, and visualization of mobile software system's run-time architecture (Mikic-Rakic et al., 2004).

Note that physical mobility directly impacts the system's execution context, which, as discussed further in Section 5.2, instigates the run-time mobility analysis of the system. We formulate the problem of logical-mobility-analysis in terms of the model presented above as follows: given the current deployment of the system, find an improved deployment, such that the users' overall utility (i.e., satisfaction with the QoS delivered by the services) is maximized and all of the locational and resource constraints are satisfied. In the most general case, the number of possible deployment architectures is $|H|^{|C|}$, where H is the set of hardware hosts and C is the set of software components. However, some of these deployments may not be legal, i.e., they may not satisfy some of the constraints.

5. Mobility analysis

Architectural models have been shown to provide an appropriate level of granularity for analysis (Edwards et al., 2007; Perry and Wolf, 1992; Shaw and Garlan, 1996). Our framework provides extensive support for architecture-based analysis. In this section, we first describe the framework's support for design-time analysis, and afterwards elaborate on the framework's support for run-time analysis.

5.1. Design-time mobility analysis

Analysis of the quality properties of software designs allows software architects to weigh trade-offs and compare design alternatives. Frequently, design decisions intended to improve one quality metric may come at the expense of other quality considerations. For example, instantiating back-up replicas of a component may improve the perceived availability of services provided by that component, but also consumes more computational resources. Determining the right balance between competing concerns requires rigorous, quantitative analysis of the design options.

Our framework implements several types of design analysis that are of particular importance in the mobile systems domain. These analyses are implemented using XTEAM's scenario-based simulation generator (Edwards et al., 2007; Edwards and Medvidovic, 2008). The simulation generator transforms architectural models into executable simulations that run on an open-source discrete event simulation engine. We customized the simulation generator to include logic that leverages our mobility modeling extensions to monitor specific behaviors and calculate metrics of system-quality properties.

The specific analytic techniques used in our framework were obtained from the software architecture and mobility research literature (Rolia and Sevcik, 1995; Cheung et al., 2008; Seo et al., 2008). Each technique was evaluated for accuracy by the developers of the technique, so we do not provide additional evaluation of the individual techniques here. Some of the design analyses techniques implemented by our framework are:

1. *Performance.* The performance modeling extensions are utilized to create a Layered Queuing Network (LQN) simulation. LQNs can be used to calculate performance metrics, such as latency, throughput, and utilization (Rolia and Sevcik, 1995). Our LQN simulation measures and records the end-to-end latency of each request–response interaction. The observed latencies may depend on numerous factors, including the load applied to the system, the computational resources available, the size of data sets, or other stochastic factors.
2. *Reliability.* The reliability modeling extensions are leveraged by a Hidden Markov Model (HMM) simulation to calculate component reliability (Cheung et al., 2008). Component reliability is defined as the percentage of time the component spends in a normal operational mode. Potential faults specified in the architectural model occur according to the probability defined by the architect. Additionally, the architect can model the recovery actions for mitigating a fault, such as instantiating back-up components or changing the system deployment. Consequently, effects of different faults and recovery actions can be assessed.
3. *Energy consumption.* The energy consumption modeling extensions permit the use of an energy consumption estimation technique (Seo et al., 2008). This technique defines equations that calculate the energy used by the executing software based on a number of application-specific and platform-specific parameters. The total energy cost is the sum of the computational energy cost, due to CPU and memory usage, and the communication energy cost, due to sending and receiving data over a wireless network.

5.2. Run-time mobility analysis

As mentioned earlier, we represent logical mobility in terms of its effect on the system's deployment architecture. In turn, run-time mobility analysis in our approach deals with the problem of determining and maintaining a good deployment for a software system on a set of mobile hosts. Our framework uses run-time redeployment of software components to alleviate potential degradations in QoS due to physical mobility of devices. For instance, consider a scenario where due to movement of devices the network throughput connecting the devices decreases, making it difficult for software components to interact with one another. One way to tackle this problem is to redeploy the software components, such that the frequently communicating components are (temporary) collocated.

The redeployment problem is an instance of multi-dimensional optimization problems, characterized by many QoS dimensions, users and user preferences, and constraints that influence the objective. Our goal has been to devise reusable algorithms that provide highly accurate results across application scenarios. An in-depth study of the generally applicable strategies resulted in several algorithms (Malek, 2007), where each algorithm is suitable to a particular class of systems or mobility scenarios. This allows the architect to run the algorithm that is most appropriate in the given context. For brevity, below we describe two approaches that pose significant trade-offs: Mixed-Integer Linear Programming (MIP; Wolsey, 2000), and genetic algorithm.

5.2.1. MIP

The first step in representing our problem as an MIP problem is defining the decision variables, $x_{c,h}$, which correspond to whether component c is to be deployed on host h (i.e., $x_{c,h} = 1$) or not (i.e., $x_{c,h} = 0$). The next step is defining the constraints (e.g., the combined required component memory cannot exceed the available memory on a host). Finally, we need to define the objective function (e.g., maximize availability, minimize latency). Unfortunately, the objective function is a quadratic function (due to the multiplication of two decision variables). Hence, the problem is by default a Mixed-Integer Non-Linear Programming (MINLP) problem. Since there is no known algorithm for solving an MINLP problem optimally (Wolsey, 2000), we transform the MINLP problem into MIP by adding new "auxiliary" variables (Wolsey, 2000). By leveraging appropriate heuristics (e.g., variable ordering Wolsey, 2000), it is possible to cut down the search space. While MIP problems can be solved optimally in principle, doing so remains computationally expensive in most cases. Our MIP algorithm may also be used in calculating the optimal deployment for mobile systems with extensive locational constraints, which, as discussed in Section 4.2, could significantly reduce the size of the search space. In such cases, it may be beneficial to invest the time required for the MIP algorithm, in order to gain maximum possible overall QoS utility.

5.2.2. Genetic

Unlike the MIP algorithm, which needs to finish executing before returning a solution, a genetic algorithm may find an improved solution before it has completed execution. Moreover, genetic algorithms can execute in parallel on multiple processors with little overhead, making them desirable in mobile and resource-constrained settings. In a genetic algorithm, an individual represents a solution to the problem. In our problem, an individual is a string of size $|C|$ that corresponds to the deployment mapping of a system's software components to hosts. Mutating an individual corresponds to changing the deployment of a few components in a given system. To evolve populations of individuals, we define a fitness function that evaluates the quality of each new individual. This function returns zero if the individual does not satisfy the resource and locational constraints; otherwise it returns the overall utility achieved by the deployment that corresponds to the individual. The algorithm improves the quality of a population in each evolutionary iteration by selecting parent individuals with a probability that is directly proportional to their fitness values.

The algorithms presented above attempt to find a deployment that achieves the maximum utility (i.e., optimal) based on the system's execution history. However, in mobile settings, it is often more desirable to select a near-optimal solution that is least vulnerable to fluctuations in the system's resources (e.g., changes in the network bandwidth). In such cases, DeSi's mobility analysis capability and XTEAM's simulation capability coordinate to select a suitable solution as follows: first, DeSi's analytical algorithms find a small number of candidate deployments with competing trade-offs; second, XTEAM simulates the candidate deployments to develop a fine-grained assessment of the QoS trade-offs in the face of resource fluctuations; and third, the simulation results is used to identify the deployment that is least impacted by the changes in the system's resources.

6. Mobility implementation support

The results of the architectural analysis (recall Section 5) are valuable only if the actual implementation of the software system corresponds to the architectural models used for the analysis. In other words, maintaining consistency between the software system's architectural model and its implementation is of utmost importance. This is particularly true in mobile and pervasive settings where the configuration of the software may have to change significantly at run-time in order to deal with the changes in the system's context. The fact that in our approach we change the software system's deployment architecture to improve its QoS further underscores this issue.

An obstacle in maintaining the desired relationship between a software system's architecture and its implementation is that the two rely on different abstractions (Malek et al., 2005b; Malek et al., 2006). Architects often design their systems using high-level constructs (e.g., components, connectors, ports), while programmers implement those abstract constructs using low-level programming language constructs (e.g., pointers, arrays, classes, variables).

To address this disconnect, we have developed an *architectural middleware* platform, called Prism-MW (Malek et al., 2005b), that provides implementation-level modules for representing each architectural element, with an API for creating, manipulating, and destroying the element. These abstractions enable a direct mapping between an architecture and its implementation. Since the middleware constitutes an integral part of our approach and has directly enabled us to satisfy some of the framework's key capabilities, we provide an overview of its underlying design with a focus on its mobility characteristics and point the interested reader to (Malek et al., 2005b; Malek et al., 2007) for other details.

6.1. Mobile architectural middleware

Fig. 3 shows the class design view of Prism-MW. *Brick* is an abstract class that represents an architectural building block. It encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition, removal, and reconnection. A distributed application is implemented as a set of interacting *Architecture* objects. *Event*s are used to capture communication in an architecture and are exchanged via *Port*s. *Component*s perform computations in an architecture and maintain their own internal state. Each component can have an arbitrary number of attached ports. When a component generates an event, it places copies of that event on its appropriate ports. Components may interact either directly (through ports) or via connectors. *Connectors* are used to control the routing of events among their attached components. Like components, each connector can have an arbitrary number of attached ports. Components are attached to connectors by creating a link between a component port and a single connector port. Connectors may support arbitrary event delivery semantics (e.g., unicast, multicast, broadcast).

In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at run-time. This property of components and connectors, coupled with event-based interaction, provides

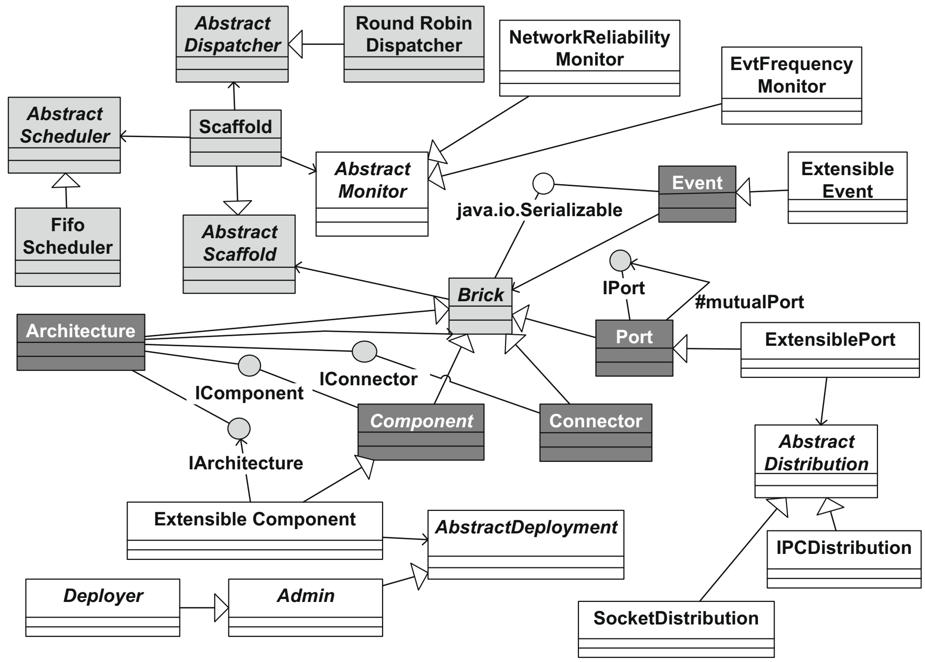

Fig. 3. UML class diagram of the design of Prism-MW.

the underpinning of our support for run-time system adaptation and mobility, as will be further discussed in Sections 6.2 and 8.

On top of the architectural support, a middleware platform intended for use in the mobile setting needs to satisfy the efficiency, heterogeneity, flexibility and context-awareness requirements, which represent some of the framework's overarching principles (recall Section 2). To satisfy these requirements, the middleware was designed from conception to be highly extensible while keeping its core unchanged via its use of abstract classes and interfaces. To that end, the core constructs (*Component*, *Connector*, *Port*, *Event*, and *Architecture*) are sub-classed via specialized classes (*ExtensibleComponent*, *ExtensibleConnector*, *ExtensiblePort*, *ExtensibleEvent*, and *ExtensibleArchitecture*), each of which has a reference to a number of abstract classes (Fig. 3). Each *AbstractExtension* class can have multiple implementations, thus enabling selection of the desired functionality inside each instance of a given extensible class. If a reference to an *AbstractExtension* class is instantiated in a given extensible class instance, that instance will exhibit the behavior realized inside the implementation of that abstract class.

In support of these requirements we extended our original design of Prism-MW in the manner depicted in Fig. 1b. The *Architectural Support* layer that was discussed earlier in the context of Fig. 3 forms the centerpiece of the middleware. To satisfy the efficiency and heterogeneity requirements, instead of accessing the system resources (e.g., threads, sockets) directly, architectural facilities leverage the substrate layer, which is called *Modular Virtual Machine (MVM)*. MVM is composed of three parts: *resource abstractions*, *implementation*, and *factories*. Resource abstractions provide a common API that is leveraged by the higher middleware layers as well as application developers to produce platform-independent code. An example of a resource made available via resource abstraction is a thread. A resource abstraction is realized via its implementation, which may use OS- or hardware-specific libraries. Resource abstractions are managed via their corresponding factories. For example, a thread factory may manage the number of threads that could be created in the system.

For efficiency, we have developed a memory management facility in MVM for memory pooling, which pre-allocates various objects (e.g., event, mutex, semaphore, etc.) from the heap when the middleware starts up. This in turn allows us to efficiently access the pool when an object of a particular type is required, and release it back to the pool when it is not needed any longer. We are thereby able to reduce the overhead of memory allocation to a simple pointer operation. Since all of the architectural constructs are treated as resources and are pre-allocated from the memory pool, we are able to estimate a system's resource consumption from its software architectural models (even at design-time). This in turn allows us to analyze and inspect the impact of architectural changes on resource usage. This level of control is important in mobile systems that are typically resource-constrained.

For heterogeneity, we insulate the architectural layer from the idiosyncrasies of the underlying system via MVM's resource abstractions. As an example consider the support MVM provides for threads. C++ developers typically have to use the OS's support for threads, and to rely on OS-level semaphore or mutex libraries for thread synchronization. To remove this dependency on the OS, we developed thread, mutex, and semaphore abstractions and the corresponding implementations in the MVM layer. Other resource abstractions were provided similarly. For a given target host, the executable image of MVM is created by building the MVM source code with the appropriate implementation files included.

Finally, we have leveraged Prism-MW's extensible design to satisfy the flexibility and context awareness requirements. As depicted in Fig. 1b, a number of advanced run-time functionalities (e.g., monitoring, remote deployment, and run-time adaptation) are developed by extending the core Prism-MW facilities. We will describe the details of these facilities in Sections 6.2, 7 and 8.

6.2. Component mobility

Our objective in the development of logical mobility support in Prism-MW has been to keep the implementation consistent with the architectural model, and thus minimize the possibility of architectural erosion due to extensive dynamism in this setting. For that, we have realized support for mobility at the level of system's component. In turn the mapping between the decisions made during run-time analysis (e.g., a new deployment architecture) and the changes required for effecting them becomes straightforward. Prism-MW has facilities to support both stateless and stateful mobility of components. The process of stateless migration can be described as follows. The sending *Admin* packages the migrant element into an *ExtensibleEvent*: one parameter in the event is the compiled image of the migrant element itself (in the Java version of the middleware this corresponds to a collection of Java class files, while in the C++ version this corresponds to a Dynamic Link Library); another parameter denotes the intended location of the migrant element in the destination subsystem's configuration. The *Admin* then sends this event to its *Connector*, which forwards the event to the attached remote *Connector*s. Each receiving *Connector* delivers the event to its attached *Admin*, which reconstitutes functional modules (i.e., components and connectors) from the event, and invokes the *IArchitecture*'s *add* and *weld* methods to insert the modules into the local configuration.

The technique described above provides the ability to transfer code between a set of hosts. As such, the stateless technique is useful for performing initial deployment of a set of components and connectors onto target hosts. Cases that require run-time migration of architectural elements need the migrant elements' state to be transferred along with the compiled images of those elements. Additionally, the migrant element may need to be disconnected and deleted from the source host (if the element's replication is not desired or allowed). We provide two complementary techniques for stateful mobility: one serialization-based and one event stimulus-based.

The serialization-based technique relies on the existence of Java-like serialization mechanisms in the underlying PL. Instead of sending a set of compiled images, the local Admin possibly disconnects and removes the (active) migrant elements from its local subsystem (using the *IArchitecture*'s *unweld* and *remove* methods), serializes each migrant element, and packages them into a set of *ExtensibleEvent*s, which are then forwarded by the *Connector*. *Admin* on each receiving host reconstitute the architectural elements from these events and attach them to the appropriate locations in their local subsystems.

If the serialization-like mechanism is not available, we use the event stimulus-based technique: the compiled image of the architectural element(s) to be migrated is sent across a network using the stateless technique. In addition, each event containing a migrant element is accompanied by a set of application-level events needed to bring the state of the migrant element to a desired point in its execution. Once the migrant architectural element is received at its destination, it is loaded into memory and added to the architecture, but is not attached to the running subsystem. Instead, the migrant element is stimulated by the application-level events sent with it. Any events the migrant element issues in response are not propagated, since the element is detached from the rest of the architecture. Only after the migrant architectural element is brought to the desired state is it welded and enabled to exchange events with the rest of the architecture. While less efficient than the serialization-based scheme, this is a simpler, PL-independent technique that is natively supported in Prism-MW. At the same time, the memory cost of the event stimulus-based technique may be large if the quantity and size of events needed to update the state of a component are large.

6.3. Code generation

In order to leverage architectural models to automate system construction, XTEAM provides a Prism-MW code generator that synthesizes C++ source code for the Prism-MW platform. As noted before, this not only reduces development effort, but also ensures that the implemented architecture matches the modeled architecture, which is essential when design decisions are based on analyses of that architecture.

The Prism-MW code generator synthesizes source code to the greatest extent possible, depending on the level of detail included in the architectural model. Some architectural models may include only definitions of component types and their interfaces; in this case, the code generator only produces skeleton code for each component. If the interactions between components are also captured in the architectural model, then significant amounts of glue-code can also be generated. When component behaviors are modeled in sufficient detail, the Prism-MW code generator may be able to create complete component implementations, including the application business logic. However, many architectural models may omit the details of complex algorithms and operations encapsulated within components, if these are not significant from an architectural perspective.

The Prism-MW code generator is capable of producing configuration files that are optimized for efficiency (i.e., resource usage). These configuration files utilize the modeling extensions for mobile systems described in Section 4.1. For instance, the middleware's MVM layer (recall Section 6.1) allows applications to specify whether resources are created on-demand or through a pool, and when pooling is used, applications may control the sizes of pools for different types of objects. Architectural models may include decisions about resource allocation, which are represented via the mobility modeling extensions, in turn allowing configuration files that control this aspect of the middleware to not only be automatically generated, but also to be optimized with respect to resource usage (e.g., use the least amount of memory).

7. Context awareness and monitoring

Transparency (i.e., hiding distribution, location, and interaction of distributed objects) is considered to be one of the cornerstones of engineering distributed software systems (Tanenbaum and van Steen, 2006), as it allows for the management of complexity associated with the development of such systems. In modern distributed systems, transparency is often achieved by employing a middleware platform. While some of its aspects are beneficial in mobile computing (e.g., support for heterogeneous data serialization), transparency has been shown to suffer from major shortcomings when applied extensively in this setting (Tanenbaum and van Steen, 2006; Capra et al., 2003). For instance, the concept of location, which is often abstracted away completely by traditional middleware platforms, becomes a first-class concern that needs to be readily available for mobile software systems. Therefore, mobile and pervasive setting calls for *context awareness* (i.e., the ability to detect changes in certain crucial, potentially external, parameters and adjust accordingly) (Capra et al., 2003; Schilit et al., 1994; Julien and Roman, 2002).

To support various aspects of awareness, Prism-MW provides support for *architectural reflection* (Tisato et al., 2000) via meta-level components. A meta-level component is implemented as an *ExtensibleComponent*, which contains a reference to the *Architecture* object via the *IArchitecture* interface and allows the component's instances to make run-time changes on the system's local (sub)architecture. The *ExtensibleComponent* class can also have references to abstract classes that provide specific (meta-level) functionality (see Fig. 3). The role of components at the meta-level is to observe and/or facilitate different aspects of the execution of application-level components. At any point, the developer may add meta-level components to a (running) application. Meta-level components may be welded to specific application-level connectors to exercise control over a particular portion of the architecture. Alternatively, a meta-level component may remain unwelded and may instead exercise control over the entire architecture via its pointer to the *Architecture* object. The structural and interaction characteristics of meta-level components are identical to those of application-level components, eliminating the need for their separate treatment in the middleware.

To date, we have augmented *ExtensibleComponent* with several extensions (Malek et al., 2007). Below we describe in detail a particular type of a meta-level component that we have developed for monitoring system's properties and component redeployment. We describe the component's support for monitoring below, and present its support for run-time redeployment and mobility in the next section.

In support of monitoring, Prism-MW provides the *AbstractMonitor* class associated through the *Scaffold* with every *Brick* (shown in Fig. 3). This allows for autonomous, active monitoring of a *Brick*'s run-time behavior. We have developed several implementations of the *AbstractMonitor* class, two of which are shown in Fig. 3: *EvtFrequencyMonitor* records the frequencies of different events the associated *Brick* sends, while *NetworkReliabilityMonitor* records the reliability of connectivity between its associated connectors and other, remote connectors using a common "pinging" technique. Fig. 1b depicts the instrumentation of a software system's architectural elements with the above monitoring facilities.

A meta-level components, called *Admin*, is used to collect and assess the monitored data. An *Admin* component is an *ExtensibleComponent* with the implementation of *AbstractDeployment* installed on it (recall Fig. 3). Since the *Admin* component on each device contains a pointer to its *Architecture* object, it is able to access the architectural elements in its local address space and obtain their monitored data. An *Admin* also serves as a *gauge* (Garlan et al., 2001) for assessing the data that is collected. For instance, in our framework an *Admin* determines when the monitoring data on an associated device has become stable. Afterwards, it forwards the data to DeSi, which then uses the collected data to populate its model with the actual system properties. At this point, one of the algorithms provided by DeSi (recall Section 5.2) could be executed for improving the system's QoS. Finally, the result of analysis is reported back to the individual *Admin*s, which coordinate the redeployment of the system between different hosts in the manner described in the next section.

8. Architecture adaptation and deployment

As discussed previously, the nature of mobile systems mandates frequent changes in the system parameters and possibly the provisioned QoS. As such, to accommodate the new conditions, the system's architecture and deployment may have to be changed (recall Section 5). The architectural analysis may prescribe changes that could range from small, localized (e.g., installation of a new component, replacement of an old component) to system-wide (e.g., changing the architectural style, new deployment architecture) adaptations of the software. In either case, support for both remote component deployment (i.e., remote installation) and component mobility via migration are crucial. We elaborate on the former below and describe the framework's support for the latter in the next section.

Prism-MW allows components to exchange *ExtensibleEvent*s, which may contain computational elements (components and connectors) as opposed to data. Additionally, *ExtensibleEvent*s implement the *Serializable* interface (as shown in Fig. 3), allowing their dispatching across address spaces. These two properties, along with the middleware's reflection capability (recall Section 7), form the basis for component mobility in systems built on top of Prism-MW.

In order to deploy the desired set of architectural elements onto a set of target hosts, we assume that a skeleton configuration is preloaded on each host (the shaded objects in Fig. 1d). The skeleton configuration consists of an *Architecture* object that contains an *Admin* component and an associated connector with several *DistributionEnabledPort*s (i.e., *ExtensiblePort*s with the appropriate implementation of *AbstractDistribution* installed on them) attached to it. *Admin* component's role in monitoring a mobile software system was described in the previous section. Similarly, since the *Admin* component on each device contains a pointer to its *Architecture* object, it is capable of making run-time changes to its local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. *Admin*s are able to send and receive the *ExtensibleEvent*s containing application components from any device to which they are connected.

As depicted in Fig. 1d, *Admin*s and *DeSi* collaborate as follows:

1. *Deployer*, a designated *Admin* component in charge of coordination, receives a new deployment architecture from DeSi. *Deployer* then sends events to inform each *Admin* of its new local configuration and of the remote locations of software components required for making changes to its configuration.
2. Each *Admin* determines the difference between its current and new configurations, and issues a series of Prism-MW Events to remote *Admin*s requesting the components that are to be deployed locally. If devices that need to exchange components are not directly connected, the relevant request events are sent to the *Deployer*, which then mediates their interaction.
3. Each *Admin* that receives an event requesting its local components to be deployed remotely, detaches these components from its local configuration, serializes them, and sends them as a series of events via its *Connector* to the requesting device.
4. A recipient *Admin* reconstitutes the migrant components from the received events and invokes the appropriate methods on its *Architecture* object to attach the received components to its configuration.

While the above protocol is not applicable to all classes of systems (e.g., highly decentralized, disconnected, and ad hoc mobile systems), other mechanisms intended for such systems can be developed in a similar manner, via different implementations of *Admin* and *Deployer* components.

9. Evaluation

We have a broad experience applying the overall framework to several distributed and highly heterogeneous systems (Malek et al., 2006, 2007; Malek, 2007). In each case, we investigated the practicality of the framework in meeting the unique challenges presented by mobile computing. In this section, we describe our experience applying the framework to a family of mobile robotics systems with a specific focus on mobility concerns. We first summarize two evaluation scenarios: environment exploration and robot-following. We then assess, in detail, each of the framework's components and their collaboration with each other separately.

Our goal in this evaluation has been to verify the framework's ability to satisfy the five objectives (principles) mentioned in Section 2. In particular, we evaluate the feasibility of software architecture-based approach to designing, analyzing, implementing, adapting, and migrating mobile systems while satisfying other traditional concerns in this setting, such as efficiency of implementation and the ability to deal with heterogeneity. In Section 10, we enumerate and discuss the key differences between our framework and other architecture-driven adaptation frameworks.

9.1. Evaluation scenarios

To scope our exposition of this work, we use mobile robotics as our target domain. Several recent approaches have taken an explicit software architectural perspective to building mobile robotics systems, resulting in reusable design and implementation frameworks (Brugali, 2007; Kramer and Scheutz, 2007; Georgas and Taylor, 2008). At the same time, each of these approaches has tended to neglect one or more of the issues we consider critical: exploration of the design space, system modeling, analysis, traceability of the key design decisions in the system's implementation, deployment, and run-time adaptation in the face of development platform mobility and heterogeneity. These are the issues that have directly motivated our framework. We describe two instances of mobile robotics systems, one of which we developed in collaboration with Bosch Research and Technology Center (Bosch RTC) over the past year. Section 9.1.1 describes a mobile robotics system that explores and maps an unknown environment – a common scenario for robotics applications in domains such as emergency response, planetary science, and defense. Section 9.1.2 describes several variations of a collaborative group of mobile robots autonomously navigating a path while avoiding collisions with various obstacles – a scenario that is representative of robotic applications in domains such as supply-chain logistics and transportation.

The applications for the two scenarios involved varying numbers of mobile hosts, components, and quality objectives, which provided a broad evaluation medium of the framework. We used three mobile hardware platforms: iRobot Create platforms enhanced with the eBox 3854 embedded PC, laptops, and Compaq iPAQ PDAs. These hardware devices ran five different operating systems: the eBox embedded PCs ran Fedora Linux, the laptops ran Windows XP and Windows Vista, and the PDAs ran GPE 2.6 Linux and Windows CE. To control the iRobot mechanical devices, we relied on three different libraries: the Player library (implemented in C), the Create Open Interface Library (implemented in C), and our custom-built iRobot driver (implemented in Java). This enabled us to evaluate two versions of Prism-MW: the Java version running on the JamVM and the GNU C++ version running on a virtual machine we developed previously in collaboration with Bosch RTC. We used Player version 2.0.5, which is implemented in C but is also compatible with JavaClient2, presenting us with two options for interacting with iRobots for each implementation of Prism-MW. Finally, there were four other devices used in our scenarios: iRobot Home Bases (to dock iRobots and charge batteries during scenario execution), Creative Webcam and Logitech QuickCam cameras (controlled via the Java Media Framework, or JMF), and Sun SPOT Java-based sensors.

9.1.1. Environment exploration

The first scenario, environment exploration, involves mobile robots exploring and mapping an unknown environment with randomly-placed obstacles, as shown in Fig. 4. Five teams, each consisting of two graduate students, built unique solutions for this scenario during a 10-week, two-part project. Among the ten students, only one had prior robotics experience. The project allowed us to investigate whether a software architecture-focused development and implementation framework could (1) *simplify* the initial development, (2) enable subsequent *adaptation* of mobile systems, and (3) make the developed code more *portable* and *reusable*. All three of these objectives are critical factors in the development of mobile software systems, which are characterized by the heterogeneity of the computing substrate and often force developers to create one-off, highly coupled, and rigid software systems.

The project began before but was completed after we obtained the iRobots, which parallels what typically occurs in industrial embedded system development – the design and development of software begins before the hardware platform becomes available. The solutions were finally deployed in the heterogeneous environment described above consisting of iRobots, laptops, and PDAs. The first development phase of the exploration system utilized a virtual environment simulation. The second development iteration replaced the simulated robots with the newly obtained, real iRobots.

All five teams succeeded in preserving their designed architectures during implementation and deployment to the iRobots, demonstrating the framework's ability to prevent architectural erosion. Components that controlled the high-level behavior of the virtual robots ported seamlessly to the new hardware platforms. Although all the solutions were operational, two did not work as intended due to misuse of the iRobot's API (i.e., programming errors that fall outside of the scope of the challenges intended to be addressed by our framework). Functionally, the five developed applications were similar. Minor variations included the introduction of a component that implements the A^* algorithm for point-to-point navigation. However, the quality attributes of the solutions were observably and measurably different.

The differences in quality were a direct consequence of the significant differences in the five software architectures and deployments. The teams relied on different architectural styles, which, in turn, endowed their architectures with different characteristics. This experience verified our earlier assertions that while many software architectures may provide the same functionality, they may exhibit significantly different quality attributes.

Fig. 5 shows two example architectures that emerged from this project. The peer-to-peer solution in Fig. 5a is scalable and tolerant of host failures. On the other hand, it may experience data-consistency and synchronization problems if the events sent by peers are dropped, arrive too late or out of order, or are processed in the wrong order. The client–server solution in Fig. 5b has a central GRID component that ensures a consistent global view of the system data and avoids synchronization problems. At the same time, this component represents a single point of failure, and a potential performance bottleneck. Additionally, the stateless robots operate

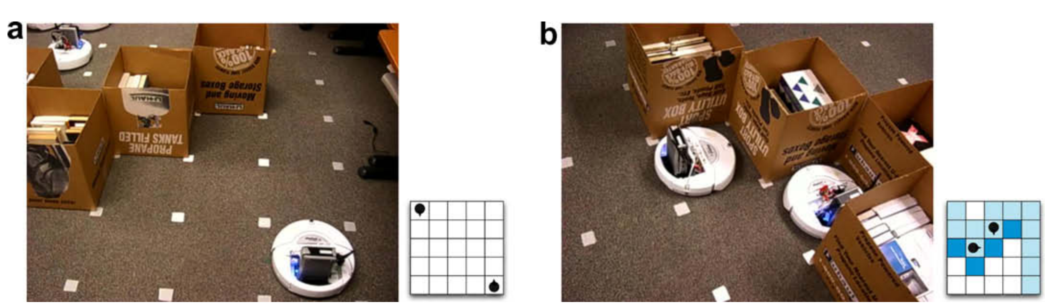

Fig. 4. Two robots are controlled remotely to map out a 5×5 grid with unknown obstacles. The initial configuration (a) is the blank map containing only the robots' positions and orientations. An intermediate configuration (b) has the majority of the grid traversed and four found obstacles.

only in the proximity of the central component, which is an undesirable property in a mobile environment.

This experience demonstrated the framework's ability to meet the heterogeneity and reusability requirements in the mobile setting. Students were able to easily include the robot control libraries as part of the middleware MVM layer (recall Section 6.1) and use them inside application-level components developed in the first phase of the project. Additionally, many of the components developed by the students were reused in our subsequent work, as we will describe in the next section.

9.1.2. Robot-following

Our findings from Section 9.1.1 encouraged further investigation of the utility of our framework in the field of mobile robotics. For example, we observed that some environment-exploration solutions were significantly more power-efficient than others and our framework could have discovered such properties during the architectural design phase. In collaboration with Bosch RTC, we used the framework to design and implement solutions to several variations of a second scenario: robot-following. The framework provided invaluable support throughout the development lifecycle: from early modeling, design, and initial analyses to subsequent implementation, deployment, monitoring, adaptation, and redeployment.

In the robot-following scenario, a platoon of mobile robots assembles autonomously and follows a leader robot along a given path (Fig. 6a). Fig. 6 depicts the execution of a representative instance of the robot-following scenario: one robot, designated as the leader, follows a line on the floor using front cliff sensors; another robot uses a mounted camera to observe and follow the leader robot; finally, a third robot uses its camera to follow the second robot. Fig. 7 shows the architecture of the leader robot with line-following components (*LineFollower* and *LineFollowerPeer*) deployed. The architectures of the follower robots are identical, except those two components are replaced with color-following components. Other scenarios not described here contain components that perform infrared-following and spacial coordinate-following in similar manners.

Along the path, the robots encounter base stations that assess the state of the robots, allow a robot to dock to recharge its battery (Fig. 6b), upload and download data to and from the robot, and even install software updates on the robot. In addition to their on-board sensors, the robots collect and process data from external Sun SPOT sensor nodes deployed throughout the environment.

For example, the Sun SPOTs can act as remote controllers to correct the orientation of robots that have lost sight of the robot in front of it.

The robots collaborate by exchanging data and, when necessary, computational components (i.e., mobile code) to enable system autonomy and adaptability. Each robot runs on-board analyses to track its health. For example, a robot that is depleting its battery too rapidly may reduce its remote communication and/or its on-board computation. A recharged robot may rejoin the convoy when it sees the trailing robot. The robots execute autonomous control components to dock and update software. Alternatively, users may issue commands from the management and control platforms (laptops and iPAQ PDAs) and receive feedback about the robots' progress (e.g., position and direction) and resource status (e.g., energy consumption) on a graphical display, as shown in Fig. 8.

Finally, an unexpected outcome of our experience was that the engineers found the framework to be helpful not only for improving QoS, but also for rapidly testing a fix to a problem during the development. This is a task that is traditionally performed manually in this setting, which our framework's redeployment and monitoring facilities can help to streamline.

9.2. Assessment of framework capabilities

Using the two mobile robotics scenarios, we evaluated each of the elements of the framework individually, as well as the ability

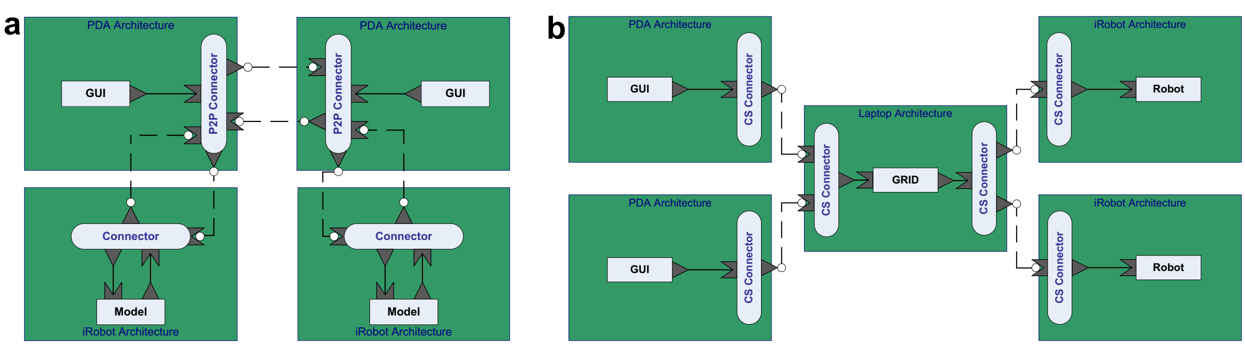

Fig. 5. Two different architectures for the environment exploration scenario, each relying on a different style: peer-to-peer (a) and client–server (b).

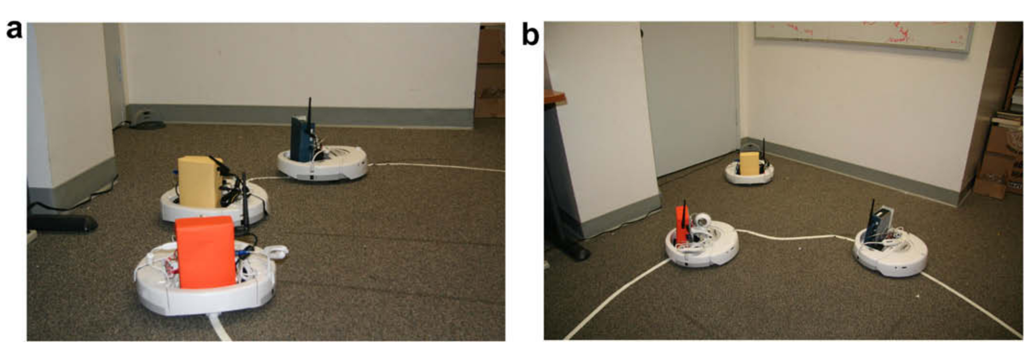

Fig. 6. (a) A platoon of three robots is following a leader and passing by a base station, shown in the top-left corner. (b) The middle robot leaves the platoon to dock with the base station.

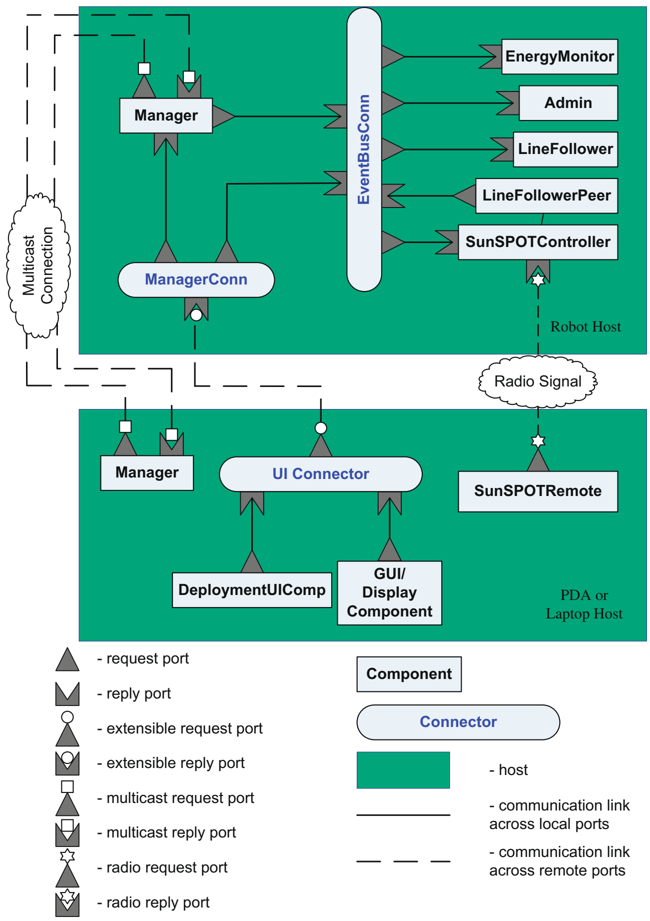

Fig. 7. A software architecture configuration deployed on a single robot and a single stationary host. The four types of ports support local interaction, point-to-point distributed interaction (extensible ports), multicast distributed interaction (multicast ports), and socket-like peer-to-peer distributed interaction (radio ports).

of the overall framework to address end-to-end system-quality goals.

9.2.1. Mobility modeling

We modeled several candidate architectures for each scenario using the architectural modeling facilities provided by XTEAM (Fig. 9 shows a partial model). The models flexibly captured the relevant system parameters and quality objectives. Moreover, we were able to easily reuse common portions of the models across different instances of the application family. The XTEAM environment allowed us to assess different configurations of the components and connectors in our system, which proved useful in making architectural decisions. For example, design-time energy-consumption analysis convinced us of the need to incorporate a new functionality in the system: when needed, robots can autonomously dock to the base stations to recharge. We used the XTEAM modeling extension for architectural styles to check selected configurations for completeness and consistency. We also performed other types of analysis (e.g., reliability and performance) which demonstrates one of the framework overarching principles, namely support for modeling and analyzing heterogeneous concerns.

Furthermore, from XTEAM, we were able to generate the underlying run-time model of DeSi, as well as the corresponding skeleton structure of the system's implementation in Prism-MW. The tightly integrated environment (e.g., model transformation and code generation capability) enabled us to alleviate the overhead associated with adopting an architecture-driven development approach.

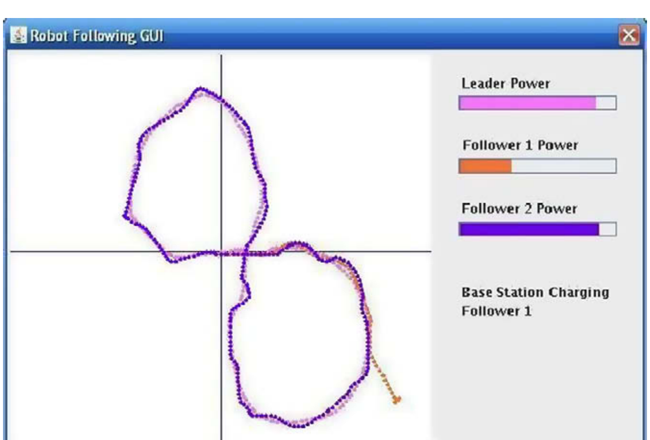

Fig. 8. Graphical display of the robot-following application. The middle robot has departed to dock with the base station.

9.2.2. Mobility analysis

During the design phase and prior to initial deployment of the software, we compared quality attributes for different designs and determined the most suitable deployments. We simulated the execution of the robotic systems in postulated scenarios and observed how quality metrics varied over time. This result corroborated our earlier empirical studies (Seo et al., 2008), and helped us to find the most suitable software architectural configuration for systems of robots.

At run-time, DeSi, which was running on the laptop, continuously analyzed the overall deployment architecture of the system. For instance, in the case of the robot-following scenario that consisted of 5 hosts and 13 software components, DeSi used its algorithms to find the best deployment among $5^{13} = 1,220,703,125$ possible combinations. The choice of algorithm (e.g., MIP versus genetic) was based on the number of location constraints specified in the model.

In our experiments, we observed more improvement in some QoS dimensions than others. For instance, since our experiments were conducted on a dedicated LAN, we noticed relatively few redeployment decisions intended to improve the system's latency and throughput. On the other hand, we observed significantly more redeployments intended to minimize the energy consumption of the battery-depleted robots. From our experiments we cannot conclude an absolute level of improvement that one should expect by using the framework, since such improvements heavily depend on the system's characteristics and the environmental setup. For example, in a controlled study (Malek, 2007) performed on a dedicated LAN, we observed the energy usage of the system could improve on average by 94% as a result of timely redeployment of the system's components. The same study showed that while improvements in energy consumption could be lower when the communication occurs on a slow network link, other QoS dimensions (e.g., latency) become a lot more sensitive to deployment decisions and could improve significantly through redeployment. Our experience with the family of mobile robotics systems corroborated these trade-offs, and served as a proof of concept that such improvements can be obtained in real-world systems.

9.2.3. Mobility implementation support

We leveraged Prism-MW to deploy the desired architecture onto the robot hardware while enforcing the constraints of specified architectural styles. Prism-MW's small footprint (less than

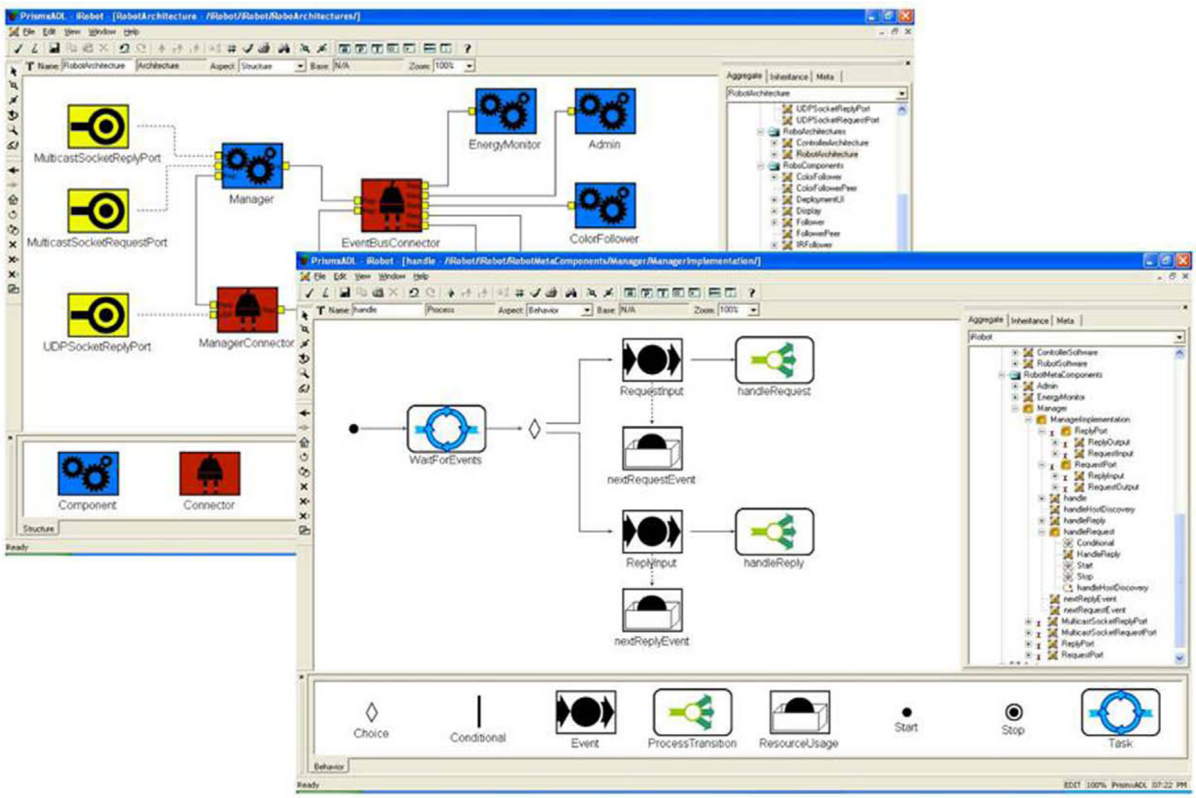

Fig. 9. Screenshots of small portions of the XTEAM model of the mobile robotics architecture.

2.3 KB) made it possible to deploy Prism-MW components both on iRobots and on significantly more resource-constrained Sun SPOTs, providing common communication mechanisms, interfaces, and protocols. On average, Prism-MW proved to be efficient, which is a key requirement for a middleware platform intended for execution on mobile devices: Prism-MW introduced less than 4% overhead on dynamic memory consumption and negligible performance overhead (around 0.5%) when compared to equivalent baseline solutions implemented in plain Java or C++.

The middleware also proved to be scalable. In one experiment, intended to evaluate the middleware's performance, a middleware instance running on one of the PDAs was configured with 10 shepherd threads and a queue of 1000 events. The system's architecture consisted of a total of 51 software components formed in a flat architecture communicating over a single connector. One software component sent 10,000 request events to 50 other components, resulting in a total of 500,000 reply events being handled in under 1.8 s. In this experiment, since the objective was to evaluate the middleware's performance, the component's did not have robotics-specific application logic.

We successfully leveraged Prism-MW's extensible design to develop a number of robotics-specific facilities. This experience, along with our previous results (Malek et al., 2005b; Malek et al., 2007), verified the practicality and flexibility of Prism-MW's extensibility mechanisms, which is particularly important in the mobile systems setting as it permits customization for heterogeneous mobile platforms. For example, Prism-MW's flexibility and extensibility allowed us to deal with robots leaving the convoy to recharge. Using Prism-MW's communication facilities, the robot departing to dock notifies its follower, causing the remaining robots to dynamically adjust their leader–follower roles to maintain their organization. As we mentioned earlier, Prism-MW supports communication even between components deployed on different platforms and developed in different programming languages.

These experiments enabled us to verify the framework's ability to satisfy some of its overarching principles of efficiency, performance, and flexibility.

9.2.4. Architecture awareness and monitoring

In the robot-following applications, the monitoring capabilities of Prism-MW and DeSi allowed users to observe and control system behavior from a management console (implemented by the *DeploymentUIComp* and *Display* components in Fig. 7). The management console allows users to issue control commands directly to application components and architectural adaptation commands such as adding or removing components. Our assessment of Prism-MW's monitoring support shows that monitoring on each host induced as little as 0.1% and no greater than 10% in memory and computation overheads. In our experiments, we monitored parameters required for performing reliability, performance, and energy consumption analysis. To further reduce the overhead of monitoring, we utilized a technique where the frequency of sampling a system resource varies based on the amount of fluctuations in that resource.

For example, the system monitored itself for failures and inefficient battery usage. An energy-efficient architecture is highly desirable because the robots typically exhaust their batteries quickly. To that end, we developed the *EnergyMonitor* component that employs the architecture-monitoring and energy-awareness facilities of our framework. *EnergyMonitor* is an implementation of *AbstractMonitor*, another instance of reliance on Prism-MW's extensibility (see Section 7). The *EnergyMonitor* component is responsible for making decisions about when to dock and charge particular robots. Additionally, as we described in Section 7, *EnergyMonitor* periodically sends the aggregated data to an instance of DeSi running on the laptop. DeSi then calculates a new deployment architecture to prolong the system lifetime through redeployment of energy-intensive components to platforms with greater capacity (e.g., laptop, docks).

9.2.5. Architecture adaptation and deployment

Prism-MW's dynamic redeployment capabilities allowed seamless adjustment of robot functionality at run-time. When software components or hardware devices (e.g., cameras) failed, the system detected these failures and took appropriate action to alleviate the situation, demonstrating the practical applications of the run-time adaptation capabilities of the framework. For instance, the system automatically deployed new software components to replace failing versions, or adapted the existing software to use a backup hardware device, such as an infrared sensor to replace a failed camera. When the robots' battery power diminished, the redeployment analysis suggested a new deployment architecture with energy-intensive components deployed on the docking base. The framework then initiated a redeployment of the system.

The adaptation capabilities and software updates rely on a running skeleton configuration that corresponds to the *Admin* and *Manager* components in Fig. 7. DeSi is used throughout the system for: (1) initial deployment of the system (e.g., the *ColorFollower* and *LineFollower* components are initially deployed to corresponding robots), (2) adaptive redeployment (e.g., when the leader robot goes to a docking station, DeSi is used to redeploy the *LineFollower* component to the new convoy leader) and (3) pushing software updates (e.g., besides recharging the battery, the docking stations feed the software updates to the robots).

The applications were initially deployed on the robots via Prism-MW's stateless component mobility support. This resulted in the class files of 6 components with a total size of 56 KB to be transmitted to the robots. Afterwards, 6 Prism-MW events were sent from DeSi to the Admin component for enacting the system's initial architecture. The total amount of time required for the system's initial deployment was 1.43 s.

In scenarios that required redeployment or replacement of software components, the framework leveraged Prism-MW's stateful component mobility support to adapt the system at run-time. While the actual overhead of redeployment (in terms of the time that the system is unavailable) depends on the size of software components and the characteristics of network links, in the cases we encountered, this overhead was not prohibitive. For instance, on average, the mobility analysis resulted in two components to be serialized and migrated and such changes required 0.25 s. for completion. Finally, note that the overhead may also depend on the application-specific characteristics of the system. We encountered such a situation when as a result of redeploying the *ColorFollower* component, the communication with the robot's camera had to be reinitialized, resulting in an additional delay of 0.93 s. When such delay was not desirable, we specified a locational constraint to prevent the component from redeployment.

10. Related work

There are four general categories of research relevant to our work: architecture-based implementation and evolution, quality attributes of software architectures, software mobility technologies, and development frameworks for mobile robotics systems. We provide an overview of the most relevant previous works from these areas below, and outline the differences between them and our approach.

Previous work has investigated the development and evolution of a software system at the architectural level. ArchJava (Aldrich et al., 2002), an extension of Java, ensures that the implementation conforms to architectural constraints. However, it lacks explicit support for mobility, beyond what is provided in the Java language. ArchJava also does not have any constructs to support quality assessment of different architectures, or any tools for aiding and optimizing the system's deployment. Aura (Sousa and Garlan, 2002) is an architectural style and supporting middleware for ubiquitous computing with a special focus on context awareness, and context switching. Although Aura supports component mobility and recognizes the importance of QoS in ubiquitous applications, it makes several simplifying assumptions (e.g., that the different QoS are independent from one another, which is clearly not the case for many QoS). Aura is thus only applicable to limited classes of applications in the embedded setting.

We categorize the techniques available for assessing the properties of a software system at the architectural level into two groups: design-time and run-time analysis.

With respect to design-time analysis, we consider Prediction-Enabled Component Technology (PECT) (Hissam et al., 2002) and Cadena (Childs et al., 2006) to be most closely related to our work. PECT is a proposed framework for the integration of component technologies and analysis technologies (Hissam et al., 2002). A PECT can be used to determine the emergent properties of a highly complex assembly of software components when certain characteristics of the individual components can be certified. We are not aware of an application of PECT to the mobile computing domain. Cadena is an extensible environment for modeling and development of component-based architectures (Childs et al., 2006). Cadena also provides an integrated model-checking infrastructure, Bogor, which enables automatic verification of the logical properties of a system, such as event sequencing. However, Cadena provides little support for the implementation of additional types of non-functional analysis.

The most relevant previous work in run-time analysis deals with the quality of a software system's deployment architecture. I5 (Bastarrica et al., 1998) proposes the use of binary integer programming for generating a deployment of a distributed application that minimizes the overall remote communication. As such, I5 is computationally expensive and does not provide support for other QoS. I5 also does not deal with run-time reconfiguration, but assumes all the parameters for determining the optimal deployment are stable and known a priori. Kichkaylo et al. (2003) provide a model for describing a distributed system in terms of the constraints on its deployment, and an AI planning algorithm for solving the model. This approach does not provide approximative solutions for large application scenarios or deployment and run-time facilities for deployment.

Mobility technologies are also related to our work. XMIDDLE (Mascolo et al., 2002) is a data-sharing middleware for mobile computing. It allows applications to share data encoded as XML with other hosts to access the shared data when disconnected from the network and reconcile data inconsistencies. Lime (Murphy et al., 2001) is a middleware that provides a coordination layer that can be exploited for designing applications that exhibit either logical or physical mobility. Lime is specifically targeted at the complexities of ad hoc mobile environments. MobiPADS (Chan and Chuang, 2003) is a middleware that supports active deployment of augmented services for mobile computing. It allows dynamic adaptation to support configuration of resources and optimize the operations of mobile applications. While Prism-MW may include features and exhibit characteristics that are similar to those provided by some of the above technologies, unlike them, it provides native implementation facilities for software architecture-based development and adaptation in a manner that is suitable to mobile systems. Finally, unlike our framework, the above mobility technologies do not explicitly support capturing relevant parameters that affect a system's QoS and do not provide analysis facilities to improve system deployment.

Over the past several years, a number of technologies for developing robotics software systems have emerged. Several such technologies are surveyed in (Kramer and Scheutz, 2007; Brugali, 2007; Georgas and Taylor, 2008). They typically provide higher-order development support for robotics software, in the form of integrated development environments, libraries, frameworks, and middleware. Unlike our approach, these technologies are not generally applicable to other domains and they do not attempt to explicitly address the challenges posed by mobility. Sykes et al. (2008) developed a 3-layer architecture that uses formalized goals to control a robotics system. The highest layer in the architecture generates reactive plans from goals, the middle layer assembles and configures the components according to the plans and execute the plans, and the lowest layer contains the deployed components. To support architecture-based implementation, Sykes et al. used Backbone, a language that allows implementation of components as Java objects (McVeigh et al., 2006). Backbone allows instantiation and interconnection of components and can modify an architecture at run-time, similarly to our Prism-MW's capabilities. The architecture can handle some application failures by replacing components or choosing a new plan step when application reaches an unexpected state. Future work includes plans to perform run-time re-planning from different goals at the highest layer; however, this work focuses on system self-management and not mobility concerns. Georgas and Taylor (2008) applied the Knowledge-Based Architectural Adaptation Management (KBAAM) (Georgas and Taylor, 2004) approach to robotics applications. The architectural model manager represents the system architecture model with connectors and components as first-class entities. The architectural run-time manager implements and executes the model and collects run-time data for the architectural adaptation manager, which, in turn, uses that data and environmental data to evaluate preset policies to determine the need for adaptation. The model manager can then adapt the model accordingly, and inform the run-time manager to change the system architecture at run-time. This framework leverages several languages and tools, e.g., xADL 2.0 (a highly extensible XML-based ADL) for architectural modeling, extended xADL schema for policy support, and ArchStudio 3 for run-time architecture implementation and evaluation. Again, this work did not focus on QoS and mobility concerns.

11. Conclusion

Software architecture provides useful abstractions, techniques, and tools for designing and organizing software systems. Further, software architecture is particularly important in the case of complex pervasive and mobile software systems. In this paper, we have presented a software architecture-based framework that provides complete support for the entire life-cycle of a mobile software system. The framework consists of an integrated tool suite that allow for design, analysis, implementation, deployment, and run-time migration of mobile software systems. At design-time, the framework is used to assess the quality attributes of the system's initial architecture. Once a good initial architecture is determined, the framework's architectural middleware and code-generation facility ensure that the implemented system satisfies the architectural requirements. At run-time, the framework copes with the challenges posed by the highly dynamic nature of mobile systems through continuous monitoring and calculation of the most suitable architecture. If a better architecture is found, the framework adapts at run-time the software, potentially via component mobility.

Our experience with applying the framework to a family of distributed mobile robotics systems has been positive. The framework helped us to improve the quality of the software built within the robotics domain and make the process of designing, implementing, and maintaining that software more efficient. In particular, it allowed us to alleviate the challenges posed by mobility that developers often face in this setting. We are currently in the process of transitioning the methodology and the technologies underlying this work to our industrial collaborators at Bosch and anticipate a number of additional, interesting challenges as a result of their application of the framework to new settings.

Our experience has also suggested several remaining challenges that form avenues of our future work. One challenge pertains to a class of mobile systems that are long lived. These systems require a solution that is itself adjustable. For instance, while the system's architects may choose a computationally expensive redeployment strategy initially (e.g., a precise but inefficient redeployment algorithm), during the system's execution they may be forced to switch to light-weight system monitoring and fast, though less precise, redeployment calculations. In other words, the framework would have to be able to reflect on itself and adjust accordingly. Another factor that must be taken into account is the amount of downtime, or downgraded QoS, the system will experience in order to migrate a component. In fact, sometimes a suboptimal deployment architecture may be preferable if it can be deployed more quickly.

While our framework is not the first software architecture-based development framework, it is the first to offer complete life-cycle support for the development of mobile systems. Our framework focuses on the most challenging of those concerns that engineers encounter when developing mobile systems, though more concerns remain. This framework is a first step toward an integrated start-to-finish tool suite for development of mobile systems.

Acknowledgements

This work is partially funded by: contract W9132V-07-C-0006 with US Army Geospatial Center; NSF grants 0820060, 0820170, 0312780, and 0120778 (as part of the Center for Embedded Network Sensing (CENS)); Bosch; and a gift from the Okawa Foundation.

References

Abowd, G.D., Allen, R., Garlan, D., 1995. Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology 4 (4), 319–364. doi:10.1145/226241.226244.

Aldrich, J., Chambers, C., Notkin, D., 2001. ArchJava: connecting software architecture to implementation, In: Proceedings of the 24th International Conference on Software Engineering (ICSE02), ACM New York, NY, USA, pp. 187–197.

ANSI/IEEE standard 1471 and ISO/IEC 42010, 2007. <http://www.iso-architecture.org/ieee-1471>.

Bastarrica, M.C., Shvartsman, A.A., Demurjian, S., 1998. A binary integer programming model for optimal object distribution. In: Proceedings of the Second International Conference on Principles of Distributed Systems.

Brugali, D., 2007. Software Engineering for Experimental Robotics (Springer Tracts in Advanced Robotics). Springer-Verlag, New York, Inc., Secaucus, NJ, USA.

Capra, L., Emmerich, W., Mascolo, C., 2003. Carisma: context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering 29 (10), 929–945.

Carzaniga, A., Picco, G., Vigna, G., 1997. Designing distributed applications with mobile code paradigms. In: Proceedings of the 19th International Conference on Software Engineering, Boston, MA, pp. 22–32.

Chan, A.T.S., Chuang, S.N., 2003. MobiPADS: a reflective middleware for context-aware mobile computing. IEEE Transactions on Software Engineering, 1072–1085.

Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L., 2008. Early prediction of software component reliability. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, ACM, New York, NY, USA, pp. 111–120. doi:10.1145/1368088.1368104.

Childs, A., Greenwald, J., Jung, G., Hoosier, M., Hatcliff, J., 2006. Calm and Cadena: metamodeling for component-based product-line development. Computer 39 (2), 42–50. doi:10.1109/MC.2006.51.

Ciancarini, P., Mascolo, C., 1998. Software architecture and mobility. In: Proceedings of the Third International Workshop on Software Architecture, ACM New York, NY, USA, pp. 21–24.

Clements, P., Kazman, R., Klein, M., 2002. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

Dashofy, E.M., van der Hoek, A., Taylor, R.N., 2005. A comprehensive approach for the development of modular software architecture description languages. ACM Transactions on Software Engineering and Methodology 14 (2), 199–245. doi:10.1145/1061254.1061258.

Edwards, G., Medvidovic, N., 2008. A methodology and framework for creating domain-specific development infrastructures. In: Proceedings of the International Conference on Automated Software Engineering (ASE08), IEEE, L'Aquila, Italy, pp. 168–177.

Edwards, G., Malek, S., Medvidovic, N., 2007. Scenario-driven dynamic analysis of distributed architectures. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE07), LNCS, Barga, Portugal, pp. 125–139.

Garlan, D., Schmerl, B., Chang, J., 2001. Using gauges for architecture-based monitoring and adaptation. In: Proceedings of the Working Conference on Complex and Dynamic System Architecture, Brisbane, Australia.

Georgas, J.C., Taylor, R.N., 2004. Towards a knowledge-based approach to architectural adaptation management. In: Proceedings of the First ACM SIGSOFT Workshop on Self-Managed Systems (WOSS04), Newport Beach, CA, pp. 59–63. doi:10.1145/1075405.1075417.

Georgas, J.C., Taylor, R.N., 2008. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In: Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS08), ACM, Leipzig, Germany, pp. 105–112. doi:10.1145/1370018.1370038.

Hissam, S.A., Moreno, G.A., Stafford, J.A., Wallnau, K.C., 2002. Packaging predictable assembly. In: Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD02), Springer-Verlag, pp. 108–124.

Julien, C., Roman, G.C., 2002. Egocentric context-aware programming in ad hoc mobile environments. SIGSOFT Software Engineering Notes 27 (6), 21–30. doi:10.1145/605466.605471.

Kichkaylo, T., Ivan, A., Karamcheti, V., 2003. Constrained component deployment in wide-area networks using AI planning techniques, in: Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS03), IEEE Computer Society, Washington, DC, USA, p. 3.1.

Kramer, J., Scheutz, M., 2007. Robotic development environments for autonomous mobile robots: a survey. Autonomous Robots 22 (2), 101–132. doi:10.1007/s10514-006-9013-8.

Maier, M.W., Emery, D., Hilliard, R., 2001. Software architecture: introducing ieee standard 1471. Computer 34 (4), 107–109.

Malek, S., 2007. A user-centric approach for improving a distributed software system's deployment architecture, Ph.D. Thesis, University of Southern California.

Malek, S., Mikic-Rakic, M., Medvidovic, N., 2005a. A decentralized redeployment algorithm for improving the availability of distributed systems. In: Proceedings of the Third International Conference on Component Deployment (CD05), LNCS, Grenoble, France, pp. 99–114. doi:10.1007/115907128.

Malek, S., Mikic-Rakic, M., Medvidovic, N., 2005b. A style-aware architectural middleware for resource-constrained, distributed systems. IEEE Transactions on Software Engineering 31 (3), 256–272. doi:10.1109/TSE.2005.29.

Malek, S., Seo, C., Ravula, S., Petrus, B., Medvidovic, N., 2006. Providing middleware-level facilities to support architecture-based development of software systems in pervasive environments. In: Proceedings of the Fourth International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC06), Melbourne, Australia, p. 2. doi:10.1145/1169075.1169077.

Malek, S., Seo, C., Ravula, S., Petrus, B., Medvidovic, N., 2007. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In: Proceedings of the 29th International Conference on Software Engineering (ICSE07), Minneapolis, MN, USA, pp. 591–601. doi:10.1109/ICSE.2007.69.

Mascolo, C., Capra, L., Zachariadis, S., Emmerich, W., 2002. XMIDDLE: a data-sharing middleware for mobile computing. Wireless Personal Communications 21 (1), 77–103.

McVeigh, A., Kramer, J., Magee, J., 2006. Using resemblance to support component reuse and evolution. In: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems (SAVCBS06), Portland, Oregon, pp. 49–56. doi:10.1145/1181195.1181206.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93. doi:10.1109/32.825767.

Medvidovic, N., Mikic-Rakic, M., Mehta, N.R., Malek, S., 2003. Software architectural support for handheld computing. Computer 36 (9), 66–73. doi:10.1109/MC.2003.1231196.

Mikic-Rakic, M., Medvidovic, N., 2002. Architecture-level support for software component deployment in resource constrained environments. In: Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD02), Springer-Verlag, London, UK, pp. 31–50.

Mikic-Rakic, M., Malek, S., Beckman, N., Medvidovic, N. 2004. A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. In: Proceedings of the Second International Conference on Component Deployment (CD04), LNCS, Edinburgh, UK, pp. 1–17. doi:10.1145/1075405.1075424.

Mikic-Rakic, M., Malek, S., Medvidovic, N., 2005. Improving availability in large, distributed component-based systems via redeployment. In: Proceedings of the Third International Conference on Component Deployment (CD05), LNCS, Grenoble, France, pp. 83–98. doi:10.1007/115907127.

Mikic-Rakic, M., Malek, S., Medvidovic, N., 2008. Architecture-driven software mobility in support of qos requirements. In: Proceedings of the First International Workshop on Software Architectures and Mobility (SAM08), Leipzig, Germany, pp. 3–8. doi:10.1145/1370888.1370891.

Murphy, A.L., Picco, G.P., Roman, G.-C., 2001. LIME: a middleware for physical and logical mobility, in: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS01), pp. 524–533. doi:10.1109/ICDSC.2001.918983.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40.

Rolia, J.A., Sevcik, K.C., 1995. The method of layers. IEEE Transactions on Software Engineering 21 (8), 689–700. doi:10.1109/32.403785.

Schilit, B., Adams, N., Want, R., 1994. Context-aware computing applications. In: Proceedings of the Workshop on Mobile Computing Systems and Applications, IEEE Computer Society, pp. 85–90.

Seo, C., Malek, S., Medvidovic, N., 2008. Component-level energy consumption estimation for distributed java-based software systems. In: Chaudron, M.R.V., Szyperski, C.A., Reussner R. (Eds.), CBSE, LNCS, vol. 5282, Springer, pp. 97–113.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ.

Sousa, J.P., Garlan, D., 2002. Aura: an architectural framework for user mobility in ubiquitous computing environments. In: Bosch, J., Gentleman, M., Hofmeister, C., Kuusela J. (Eds.), Proceedings of the Third Working IEEE/IFIP Conference on Software Architecture: System Design, Development, and Maintenance, Kluwer Academic Publishers, pp. 29–43.

Sykes, D., Heaven, W., Magee, J., Kramer, J., 2008. From goals to components: a combined approach to self-management. In: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS08), Leipzig, Germany, pp. 1–8. doi:10.1145/1370018.1370020.

Tanenbaum, A.S., van Steen, M., 2006. Distributed Systems: Principles and Paradigms, second ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Tisato, F., Savigni, A., Cazzola, W., Sosio, A., 2000. Architectural reflection. Realising software architectures via reflective activities. In: Proceedings of the Second Engineering Distributed Objects Workshop (EDO00), Springer, pp. 102–115.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F., 2005. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall, Upper Saddle River, NJ.

Wolsey, L.A., 2000. Integer programming. IIE Transactions 32, 273–285.

Sam Malek is an Assistant Professor in the Department of Computer Science at George Mason University (GMU). He is also a faculty member of the C4I Center at GMU. Malek's general research interests are in the field of software engineering, and to date his focus has spanned the areas of software architecture, distributed and embedded software systems, autonomic computing, service-oriented architectures, and quality of service analysis. The underlying theme of his research has been to devise techniques and tools that aid with the construction, analysis, and maintenance of large-scale distributed, embedded, and pervasive software systems. His research has been funded by NSF, US Army, and SAIC. Malek received his Ph.D. in 2007 from the Computer Science Department at the University of Southern California (USC) under the direction of Professor Nenad Medvidovic. His dissertation research was nominated by USC for the final round of the ACM Doctoral Dissertation Competition in 2007. He also received an M.S. degree in Computer Science in 2004 from USC, and a B.S. degree in Information and Computer Science cum laude in 2000 from the University of California, Irvine. Malek is the recipient of numerous awards, including USC Viterbi School of Engineering Fellow Award in 2004, and the USC Computer Science Outstanding Student Research Award in 2005. He is a member of the ACM, the ACM SIGSOFT, and the IEEE.

George Edwards is a Ph.D. candidate in the Computer Science Department at the University of Southern California (USC), where he is a USC Viterbi School of Engineering Fellow, an Annenberg Graduate Fellow, and a member of the Software Architecture Research Group in the Center for Systems and Software Engineering. He has also worked as a software architect at The Boeing Company and as a research intern at IBM's T.J. Watson Research Center. Edwards holds an M.S. in Computer Science from USC and a B.S. in Computer Science from Vanderbilt University. He has authored over twenty scholarly journal articles and conference papers on varied topics related to software engineering and distributed systems, and, in 2008, he received the USC Computer Science Outstanding Student Research Award.

Yuriy Brun is an NSF CRA postdoctoral Computing Innovation Fellow at the University of Washington. He received his Ph.D. degree in 2008 from the University of Southern California, as a Viterbi School of Engineering Fellow, and his M.Eng. degree in 2003 from the Massachusetts Institute of Technology. His doctoral research was a finalist in the ACM Doctoral Dissertation Competition in 2008. Brun's research interests are in the area of engineering self-adaptive systems, and, in particular, using mechanisms from nature to create engineering paradigms for robustness,

fault and malice tolerance, scalability, and security. He does (1) theoretical work on design and complexity analysis of biologically inspired algorithms and (2) software engineering work on implementing these algorithms for Internet-sized distributed systems, grids, and clouds. He is a member of the ACM and the ACM SIGSOFT.

Hossein Tajalli is a Ph.D. student in the Computer Science Department at the University of Southern California (USC), where he is a member of the Software Architecture Research Group in the Center for Systems and Software Engineering. His research interests are in the field of software architecture and self-adaptation in robotics applications. Prior to his work at USC, Hossein received his Master of Science in Electrical Engineering from University of Tehran, Iran.

Joshua Garcia received the M.S. degree in Computer Science from the University of Southern California (USC) in 2008 and the B.S. degree in Computer Science from USC in 2006. Currently, he is a Ph.D. student in the Computer Science Department at USC, a USC Viterbi School of Engineering Fellow, a member of the Software Architecture Research Group in the Center for Systems and Software Engineering, and a software engineer at the NASA Jet Propulsion Laboratory. He has also worked as a research intern and research assistant at the Southern California Earthquake Center at USC. His research interests are in the area of software architecture reconstruction, program analysis for program comprehension, software architecture for distributed systems, and software architecture for embedded systems. He is a member of the ACM, the ACM SIGSOFT, and IEEE.

Ivo Krka is a Ph.D. student in the Computer Science Department at the University of Southern California (USC), where he is a USC Provost's Fellow, and a member of the Software Architecture Research Group in the Center for Systems and Software Engineering. He has also worked as an engineering intern at Google Inc. and is a recipient of the Google Intern Scholarship. His main research interests include requirements specification, software architecture modeling and evaluation, and architectural recovery. Krka holds an M.S. in Computer Science from USC and an M.Eng. in Computing from University of Zagreb, where he was the top student of the graduating class. He is a member of the IEEE and the ACM SIGSOFT.

Nenad Medvidovic is an Associate Professor in the Computer Science Department at the University of Southern California (USC). He is the director of the USC Center for Systems and Software Engineering and a faculty associate of the Institute for Software Research at the University of California, Irvine (UCI). Medvidovic is the Program Co-Chair of the 2011 International Conference on Software Engineering (ICSE). Medvidovic received his Ph.D. in 1999 from the Department of Information and Computer Science at UCI. He also received an M.S. in Information and Computer Science in 1995 from UC Irvine, and a B.S. in Computer Science summa cum laude in 1992 from the Computer Science and Engineering Department at Arizona State University. Medvidovic is a recipient of the National Science Foundation CAREER (2000) and ITR (2003) awards, the Okawa Foundation Research Grant (2005), and the IBM Real-Time Innovation Award (2007). He is a co-author of the ICSE 1998 paper titled "Architecture-Based Runtime Software Evolution," which was recently named that conference's Most Influential Paper. His paper "A Classification and Comparison Framework for Software Architecture Description Languages" was recognized by the Elsevier Information and Software Technology Journal as the most cited journal article in software engineering published in 2000. Medvidovic's research interests are in the area of architecture-based software development. His work focuses on software architecture modeling and analysis; middleware facilities for architectural implementation; domain-specific architectures; architectural styles; and architecture-level support for software development in highly distributed, mobile, resource constrained, and embedded computing environments. He is a co-author of a new textbook on software architectures. He is a member of the ACM, the ACM SIGSOFT, and the IEEE Computer Society.

Marija Mikic-Rakic is a Senior Software Engineer and Technical Lead Manager at Google, Santa Monica. She holds a Ph.D. in Computer Science from the University of Southern California. Her research interests are in large-scale distributed systems. She is a member of the ACM and the ACM SIGSOFT.

Gaurav S. Sukhatme is a Professor of Computer Science (joint appointment in Electrical Engineering) at the University of Southern California (USC). He received his undergraduate education at IIT Bombay in Computer Science and Engineering, and M.S. and Ph.D. degrees in Computer Science from USC. He is the co-director of the USC Robotics Research Laboratory and the director of the USC Robotic Embedded Systems Laboratory, which he founded in 2000. His research interests are in multi-robot systems and sensor/actuator networks. He has published extensively in these and related areas. Sukhatme has served as PI on numerous NSF, DARPA, ONR, and NASA grants. He is a Co-PI on the Center for Embedded Networked Sensing (CENS), an NSF Science and Technology Center. He is a senior member of the IEEE, and a member of the AAAI and the ACM. He is a recipient of the NSF CAREER award and the Okawa foundation research award. He has served on many conference program committees, and is one of the founders of the Robotics: Science and Systems conference. He was one of the program chairs of the 2008 IEEE International Conference on Robotics and Automation. He is the Editor-in-Chief of Autonomous Robots. In the past, he has served as Associate Editor of the IEEE Transactions on Robotics and Automation, the IEEE Transactions on Mobile Computing, and on the editorial board of IEEE Pervasive Computing.