Software Architectural Support for Distributed, Mobile, and Resource-Constrained Environments

Nenad Medvidovic  
Marija Mikic-Rakic

Computer Science Department  
University of Southern California  
Los Angeles, CA 90089-0781 U.S.A.  
{neno,marija}@usc.edu

ABSTRACT
Over the past several decades software researchers and practitioners have proposed various approaches, techniques, and tools for developing large-scale software systems. The results of these efforts have been characterized as programming-in-the-large (PitL). A new set of challenges has arisen with the emergence of inexpensive, small, heterogeneous, resource-constrained, possibly embedded, highly-distributed, and highly-mobile computing platforms. We refer to software development in this new setting as programming-in-the-small-and-many (Prism). This paper presents an approach intended to address the challenges of Prism. The centerpiece of our approach is a software architectural style with explicit support for the needs of Prism applications: self-awareness, distribution, heterogeneity, dynamism, mobility, and disconnected operation. The style is accompanied by a set of implementation, deployment, and runtime evolution tools targeted to a variety of traditional (i.e., desktop) and mobile computing platforms. The style has been evaluated along several dimensions identified as critical in the design of distributed systems. Different aspects of its tool support have been quantified through a series of benchmarks. Our approach has been successfully applied on a number of applications. While several issues pertaining to Prism remain areas of future work, our experience to date has been very positive.

Keywords
Prism, software architecture, style, self-awareness, deployment, distribution, mobility, dynamism, disconnected operation

1 INTRODUCTION
The software systems of today are rapidly growing in size, complexity, amount of distribution, heterogeneity of constituent components, and numbers of users. We have recently witnessed a rapid increase in the speed and capacity of hardware, a decrease in its cost, the emergence of the Internet as a critical worldwide resource, and a proliferation of hand-held consumer electronics devices (e.g., mobile phones, PDAs). One can now envision a number of complex software development scenarios involving fleets of mobile devices used in environment and land-use monitoring, freeway-traffic management, fire fighting, and damage surveys in times of natural disaster [23,33]. Such scenarios present daunting challenges: effective understanding of existing or prospective configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data, numbers of data types, and numbers of devices; and heterogeneity of the software executing on each device and across devices. Furthermore, software often must execute in the face of highly-constrained resources, characterized by limited power, low network bandwidth and patchy connectivity, slow CPU speed, and limited memory and persistent storage. We refer to the development of software systems in the described setting as programming-in-the-small-and-many (Prism), in order to distinguish it from the commonly adopted software engineering paradigm of programming-in-the-large (PitL) [7].

Over the past several decades, the research on PitL has focused on supporting development and evolution of large-scale software systems, but whose degrees of distribution, heterogeneity, dynamism, mobility, and resource constraints are substantially lower than demanded by the scenarios outlined above. At the same time, a number of the challenges presented by Prism might appear quite familiar to software developers of 30-40 years ago. The resource-constrained nature of the Prism hardware platforms and often-proprietary development solutions on those platforms are reminiscent of the development world of the 1960’s and 1970’s, especially in the arena of embedded systems [19]. However, the sophistication demanded of today’s systems, their comparatively much greater size, complexity, distribution, mobility, and desired interoperability across heterogeneous platforms demand major advances over the solutions that were at the disposal of engineers in the past.

For this reason, the basic principle of our approach to supporting Prism has been to reuse the solutions of existing software engineering research and practice (i.e., PitL) whenever possible, inventing new solutions as necessary. In particular, one area from which we believe we can gain a lot of leverage is software architecture [24,32]. Several aspects of architecture-based development (component-based system composition, explicit software connectors [32], architectural styles, upstream system analysis and simulation, and support for dynamism) make it a particularly good fit for the needs of Prism. Software architecture is indeed the centerpiece of our approach. Our work interweaves several ideas in the field of architectures that we had explored in the past with a number of new ideas. In the process, we have developed several novel solutions to the Prism challenges, spanning two key areas:

1. Explicit design idioms comprising an architectural style for applications in the Prism setting. The Prism style supports

1. It should be noted that “small” refers to the size and capacity of devices on which the software executes, not the size of applications.
an explicit meta-architecture, which enables continuous architectural monitoring, analysis, and evolution. The style also natively supports distributed deployment, mobility, and dynamic reconfiguration of applications.

2. A light-weight, tailorable architecture implementation, deployment, and reconfiguration infrastructure comprising a middleware for developing Prism applications.

The key hypothesis of our work is that the architectural style and middleware in concert provide a sufficiently rich basis for developing applications “in-the-small-and-many”. This basis must be augmented in the future with support for other critical development concerns, including requirements elicitation, testing, debugging, and configuration management [10]. The style has been evaluated along several dimensions identified as critical in the design of distributed systems [9]. Different aspects of the middleware, including scalability and performance, have been quantified through a series of benchmarks. To date, our work has been applied to a number of applications executing on a variety of desktop and mobile computing platforms. Recently, Prism’s middleware has been successfully evaluated by a major industrial organization for use in one of their key embedded, dynamic, highly-distributed systems. While we do not yet possess sufficient evidence to make the claim that our approach is capable of addressing all of Prism’s challenges, our initial results are very promising.

The remainder of the paper is organized as follows. Section 2 briefly describes an example application used to illustrate the concepts throughout the paper. Section 3 describes the Prism architectural style, while Section 4 describes our middleware for implementing, deploying, migrating, and dynamically reconfiguring Prism applications. Section 5 provides an evaluation of our approach. The paper concludes with overviews of related and future work.

2 EXAMPLE APPLICATION
To illustrate the concepts introduced in this paper, we use an application for distributed, “on the fly” deployment of personnel, intended to deal with situations such as natural disasters, military crises, and search-and-rescue efforts. The specific instance of this application depicted in Figure 1 addresses military Troops Deployment and battle Simulations (TDS). A computer at Headquarters gathers all information from the field and displays the complete current battlefield status: the locations of friendly and enemy troops, as well as obstacles such as mine fields. The Headquarters computer is networked via secure links to a set of hand-held devices used by officers in the field. The configuration in Figure 1 shows three Commanders and a General; two Commanders use Palm Pilot Vx devices, while the third uses a Compaq iPAQ; the General uses a Palm Pilot VxIx. The Commanders control their own quadrant of the battlefield. The General can see a summarized view of the entire battlefield (shown) or detailed views of each quadrant. The General can issue direct troop deployment orders to individual Commanders or request transfers of troops among the Commanders. General can also request for deployment strategy suggestions from Headquarters. Finally, the General can issue a “fight” command, resulting in a battle simulation that incrementally determines the likely winner.

The TDS application has been designed, analyzed, implemented, deployed, migrated (both to streamline the application and as a result of network disconnection), and dynamically evolved using the techniques described in this paper. While its degrees of scale, complexity, distribution, and mobility may not be representative of all applications that one is likely to encounter in the Prism setting, the TDS application provides a sufficiently expressive platform for illustrating a number of Prism concepts. Specifically, several aspects of TDS embody the concept of multiplicity inherent in Prism (“many”). The application is implemented in four dialects of two programming languages (PLs): Java JVM, Java KVM, C++, and Embedded Visual C++ (EVC++). The application is deployed on five devices, four of which are mobile. The TDS subsystem on each device can run using an arbitrary number of threads of control. The devices are of three different types (Palm Pilot, iPAQ, PC), running three OSs (PalmOS, WindowsCE, and Windows98, respectively); in addition, several analysis components used in support of code mobility and disconnected operation run on a fourth platform (Sun) and OS (Unix). In the instance of TDS shown in Figures 1 and 2, sixteen software components deployed across the five devices interact via fifteen software connectors. The connectors enable two principal communication paradigms (client-server and peer-to-peer, discussed in Section 3) implemented in four different connector categories (see Section 4).

3 ARCHITECTURE-LEVEL SUPPORT
We have developed an architectural style that is capable of effectively capturing the characteristics of application architectures found in the Prism setting. The style is intended to address the key characteristics of Prism foreshadowed in Section 1: architectural monitoring and analysis, distribution, dynamism, mobility, and disconnected operation.

In formulating the Prism style, we have leveraged our previous experience with the C2 architectural style, which is intended to support highly distributed applications [34]. As indicated by an independent evaluation of network-based software architectural styles [9], several characteristics of C2 (distributed architectures, autonomous components communicating through explicit connectors, substrate independence, and dynamism) appear to be a good fit for the needs of Prism. However, support for other aspects of Prism (deployment, mobility, and disconnected operation) has to be built on top of C2’s existing facilities. Moreover, certain aspects of Prism simply cannot be supported by C2. C2 mandates that components engage in asynchronous
interactions only; this aspect of the style makes it ill suited for certain classes of applications (e.g., applications with real-time requirements). Furthermore, C2 imposes a strictly vertical topological orientation on architectures. This orientation is suited for a client-server style of interaction, but not for peer-to-peer interaction, which becomes critical as Prism applications become more widely distributed and decentralized.

For these reasons, we have chosen to use C2 as the basis of the Prism style, with several enhancements (described below) to account for the above shortcomings.

3.1 Topological Rules
3.1.1 Prism Style Rules Directly Inherited From C2
An architecture in the C2 style is modeled as a set of components, connectors, and the topology into which they are composed. C2-style components maintain state and perform computations. The components may not assume a shared address space, but instead interact with other components solely by exchanging messages via their two communication ports (named top and bottom). Connectors in the C2 style mediate the interaction among components by controlling the distribution of all messages. A message consists of a name and a set of typed parameters. A message in the C2 style is either a request for a component to perform an operation, or a notification that a given component has performed an operation and/or changed its state. Request messages are sent through the top ports, while notifications are sent through the bottom ports of components and connectors. The distinction between requests and notifications ensures C2’s principle of substrate independence, which mandates that a component in an architecture may have no knowledge of or dependencies on components below it.

3.1.2 Support for Peer-to-Peer Interaction
The Prism architectural style supports C2’s vertical topology and asymmetric communication via requests and notifications. In addition to C2’s asynchronous messages, the Prism style also supports synchronous message passing. Prism introduces a third component port (called side) and message category (called peer). Side ports and peer messages allow us to address the relative topological rigidity of C2 by supporting symmetric communication between components. In order to maintain component decoupling, the side ports exchange peer messages through “peer” connectors (unlabeled circles in Figure 2). Basic peer connectors have simple message broadcast semantics: a peer message incoming on any port is forwarded as an outgoing message through all of the connector’s remaining ports. Other routing semantics can be implemented in peer connectors as discussed in Section 4. The prism style allows direct connections, between peer connectors, as shown in Figure 2. In order to avoid transmitting the same message(s) to their destination(s) multiple times (e.g., in the case of N message broadcasting peer connectors that form a cycle), we have developed formal rules for situations in which a peer connector needs to propagate a message and those in which the connector must “trap” the message. These rules are elided for brevity. A side benefit of these rules is that components in the Prism style cannot deadlock. Coupling request-notification and peer-to-peer interactions in a single style demands that their roles and relationships be clearly delineated. This has resulted in an additional set of Prism style rules. Two Prism components may not engage in interaction via peer messages if there exists a vertical topological relationship between them. Allowing such interaction would violate the principle of substrate independence inherited from C2. For example, DataRepository on the PC and G_ScenarioEditor on the Palm-1 in Figure 2 may not exchange peer messages since one component is above the other; on the other hand, no vertical topological relationship exists between C_iPAQ_AvailableTroops and G_AvailableTroops and they may communicate via peer messages. For a similar reason, the Prism style disallows the possibility of exchanging messages between a peer and a horizontal connector (which would, in effect, convert peer messages into requests/notifications, and vice versa).

While providing an effective solution for peer-to-peer interaction, the introduction of a third component port and an additional connector type in the Prism style allows for the construction of complex configurations. This is especially true in degenerate cases in which the architect decides to rely primarily on the side component ports and peer connectors, which the style permits. We believe that the goal of a style should be to provide architects with enough guidance to arrive at effective configurations, but also enough flexibility to address a multitude of development situations and needs. A style, therefore, cannot and should not prevent the design of “bad” architectures a priori. Prism attempts to provide the balance between guidance and flexibility by coupling the design rules of the C2 style, which have been proven effective in the construction of distributed applications, with the added facilities of Prism. It is, then, the architect’s obligation to enact good design practices and apply the resulting rules most effectively.

3.1.3 Example Application in the Prism Style
Figure 2 shows the architectural configuration of the TDS application in the Prism style, distributed across five devices. The subarchitecture on the PC device maintains a model of the system’s overall resources—terrain, personnel, as well as the current and standard deployment strategies. The StrategyAnalyzer, DeploymentAdvisor, and WarManager components, respectively, (1) analyze the deployments of friendly troops with respect to enemy troops and obstacles; (2) suggest deployments of friendly troops based on their availability and positions of enemy troops and obstacles; and (3) incrementally simulate the outcome of the battle based on the current situation in the field. The subarchitecture on the Palm-1 device provides the General’s functionality, while Palm-2, Palm-3, and iPAQ provide the three Commanders’ functionalities. The G_AvailableTroops component in General’s subarchitecture is able to make direct orders (by sending peer messages through peer connectors) to the Commanders’ C_AvailableTroops components to reposition troops across battlefield quadrants.

3.2 Architectural Self-Awareness
In order to address the need for adapting highly dynamic and mobile applications to changes in the execution context, the Prism style supports architectures at two levels: application-level and meta-level. The role of components at the Prism meta-level is to observe and/or facilitate different aspects of the execution, dynamic evolution, mobility, and disconnected operation of application-level components. Meta-level components may be application-independent or application-specific. Application-level and meta-level components exist side-by-side in Prism (see Figure 4). Meta-level components are aware of application-level components and may initiate interactions with
allowing the changes. The ability of proposed runtime architectural changes, and possibly dismodels during the application’s execution, assessing the valid-

Figure 2. Architecture of the TDS application, displayed in Prism-DE, the Prism deployment environment. The unlabeled circles connecting components across the hand-held devices represent peer connectors.

them, but not vice versa. The Prism style rules apply to both component categories: meta-level components also engage in connector-mediated, message-based interactions with each other (and with application-level components).

In support of this two-level architecture, Prism currently distinguishes among three types of messages. ApplicationData messages are used by application-level components to communicate during execution. The other two message types, ComponentContent and ArchitecturalModel, are used by Prism meta-level components. ComponentContent messages contain mobile code and accompanying information (e.g., the location of a migrant component in the destination configuration), while ArchitecturalModel messages carry information needed to perform architecture-level analyses of prospective Prism configurations.

We have extensively employed special-purpose meta-level components, called Admin Components, whose task is to exchange ComponentContent messages and facilitate the deployment and migration of application components across devices (see Section 4). Another meta-level component is the Continuous Analysis component, which leverages ArchitecturalModel messages for analyzing the (partial) architectural models during the application’s execution, assessing the validity of proposed runtime architectural changes, and possibly disallowing the changes. The Continuous Analysis component is further discussed in Section 3.4.

3.3 Border Connectors

Due to the nature of the applications it is intended to support, the Prism style places special importance on connectors that span device boundaries. Such connectors, called border connectors, enable the interactions of components residing on one device with components on other devices. A single border connector may service network links to multiple devices (e.g., BottomBorderConnector on the PC in Figure 2). A border connector marshals and unmarshals data, code, and architectural models; dispatches and receives messages across the network; and monitors the status of the network links. It may also perform data compression for efficiency and encryption for security. Finally, it can monitor message traffic on each network link in support for disconnected operation as discussed in Section 3.6.

3.4 Modeling and Analysis Support

In order to support formal modeling and analysis of Prism-style architectures, we have developed a prototype architecture description language for Prism, called Prism-ADL. To this end, we have leveraged C2’s ADL (C2SADEL [21]) and tool support (DRADEL 1.0 [21]). We have extended C2SADEL with constructs for specifying Prism topologies, synchronous and peer-to-peer interactions, and dependency and degraded mode indicators (further discussed in Section 3.6). We have also extended DRADEL 1.0, for analyzing architectures specified in Prism-ADL, resulting in DRADEL 2.0. Prism’s meta-level Continuous Analysis component is a subset of DRADEL 2.0 used to assess the functional conformance and topological soundness of proposed architectural (re)configurations in the manner elaborated in [21].

3.5 Deployment, Mobility, and Dynamic Reconfiguration

The nature of Prism applications also demands support for their deployment, mobility, and dynamic reconfiguration. The Prism style natively supports this set of tasks by providing (1) meta-

level Admin Components, which are capable of exchanging ComponentContent messages that contain application-level components and perform instantiation, addition, and removal of components on a given device, (2) Border Connectors used to send and receive the application-level components to be deployed or migrated from remote hosts, and (3) strict separation of concerns between components and connectors which minimize component dependencies. Implementation-level support for these tasks is further discussed in Section 4.

3.6 Support forDisconnected Operation

Due to the nature of mobile devices, their network connections are intermittent, with periods of disconnection. A goal of our work is to maximize the availability of an application during disconnection. There are three techniques for achieving this goal: rerouting of remote invocations through other, “live” links, and migration or replication of components to a local host before the disconnection occurs. We leverage the topological constraints of the Prism style in all three cases. In the interest of space, we will not discuss rerouting techniques here. Instead, we focus on component replication and migration.

The set of components to be migrated and/or replicated is chosen such that it maximizes the autonomy of the local subsystem during disconnection, stays within the memory constraints posed by the device, and can be migrated within the time remaining before disconnection occurs. Our approach proposes migration of components when they are not required on the source host(s) during disconnection, and replication in all other cases. Replication requires the ability to synchronize, after the connection has been restored, any updates made to the different replicas of a component during disconnection. In order to simplify the synchronization of component replicas upon reconnection, Prism-ADL attaches a degraded mode indicator to each operation exported by a component, as discussed in Section 3.4. The indicator reflects an operation’s dependence on component state: some operations do not depend on component state and are fully accessible during disconnection (allowed); other operations are delayed until the connection is restored; finally,
three key pieces of information are needed, as discussed below.

A. Message Frequencies. Our goal is to minimize the message traffic that would go through the disconnected link(s). This is accomplished by migrating and/or replicating the components responding to these messages from remote host(s) to the local host. The Border Connectors have the capability to monitor message frequencies on each network link they control.

B. Dependency. Dependency of an operation ($d_i$) is defined as the number of operations provided by other components in the system needed for this operation’s completion. Migration of components whose operations have low $d_i$ values (“independent components”) is likely to reduce the message traffic present on the network link, while migration of “dependent components” may in fact increase the message traffic along the link. Prism-ADL associates a dependency tag with each operation provided by a component.

C. Benefit of Migration. For each candidate component the benefit can be estimated by comparing the component’s static description with the list of messages exchanged across the link. For all operations invoked on the component as a result of these messages, the benefit (initially zero) is calculated as follows:

$$\text{Benefit} \leftarrow \text{Benefit} + f_i \times (1 - d_i)$$

where $f_i$ is the frequency of the message and $d_i$ is the dependency factor of the corresponding operation. This formula states that component benefit may increase only in the case of independent operations; the benefit remains unchanged for operations that depend on exactly one external operation; it decreases in all other cases.

In order to select the best set of components for migration, we employ a variant of the 0-1-Knapsack algorithm [4]. Namely, given the benefit and required memory of $N$ components, we select a subset of the components that maximizes the total benefit $TB$ (as the sum of benefits of individual components) and stays within the total available memory TAM on the given device. This algorithm is solvable in $O(N*TAM)$. It assumes that the benefits of individual components are mutually exclusive, thus becoming an approximation in the case of highly-coupled components. However, the algorithm guarantees that the actual benefit of the resulting migration set is at least $TB$: the benefit of migrating two or more components that share a communication link is greater than or equal to the sum of their individual benefits due to the message traffic along their (migrated) link. A more exact solution is obtained by using the minimum $k$-cut algorithm [5] (where $k$ would represent the number of devices), with memory as an additional constraint. However, this algorithm runs in exponential time, which may be computationally too expensive if the number of candidate components and/or hosts is high.

In addition to the above three parameters, the ability to migrate a set of components to a device before disconnection occurs

will, of course, also depend upon the network bandwidth and the time to disconnection (which may be known in the cases of anticipated disconnection [35], or estimated based on historical data in the cases of unanticipated or sudden disconnection [22]).

4 IMPLEMENTATION-LEVEL SUPPORT

As described so far, Prism provides stylistic guidelines for composing large, distributed, mobile systems. For these guidelines to be useful in a development setting, they must be accompanied by support for their implementation. To this end, we have developed a light-weight, extensible architecture implementation infrastructure. The infrastructure comprises a middleware called Prism-MW [22,25] that supports the key elements of the style (e.g., architectures, components, connectors, messages) and their characteristics (e.g., a message has a name and a set of parameters). An application architecture is constructed using this middleware by extending its classes with application-specific detail. Prism-MW also provides a set of meta-level components that facilitate various aspects of Prism applications (deployment, mobility, dynamic reconfiguration, and disconnected operation). The middleware has been implemented in several PLs: Java JVM and KVM, C++ and Embedded Visual C++ (EVC++), and Python.

4.1 The Middleware

A subset of Prism-MW design is depicted in Figure 3. The classes shown are those of interest to the application developer. Multiple components and connectors in an architecture may run in a single thread of control (Component and Connector classes), or they may have their own threads (ComponentThread and ConnectorThread classes). Component and ComponentThread classes are abstract; a meta-level or application-level component must be subclassed from them and must provide the component’s specific functionality. On the other hand, connectors provide application-independent interaction services and may be directly instantiated and used in an application. The Architecture class records the configuration of its constituent components and connectors, and provides meta-level facilities for their addition, removal, replacement, and reconnection, possibly at system runtime. A distributed application, such as TDS, is implemented as a set of interacting Architecture objects. Finally, iScaffold is an interface exported by every Brick (component, connector, or entire architecture). iScaffold directly aids architectural self-awareness by allowing probes and monitors of the runtime behavior of a Brick, further discussed below.

Connectors. To support a variety of development situations in the Prism setting, we have implemented a library of connectors
Figure 4. Layered construction of an application using Prism-MW. The application is distributed across five devices, each of which is running the middleware. Meta-level components (highlighted in the figure) may control the execution of application-level components via Prism-MW messages or via pointers to the local Architecture object (shown in the subarchitecture on the PC).

Optimizations. Since it is intended to support applications on resource-constrained platforms, we have built several optimizations into Prism-MW. These include basic connectors that support both synchronous and asynchronous message broadcast, multicast, and unicast. Furthermore, we have applied our technique for component interactions across process and machine boundaries [6] in implementing border connectors. An example border connector is the infra-red (IR) connector, which enables short-range, wireless interaction of Prism-MW components across hand-held devices. In order to communicate across heterogeneous platforms, runtime environments, and PLs, we use XML as a medium for message passing. We have recently extended our connectors with support for real-time message delivery (see [22] for more detail). To ensure secure communication in a highly distributed and mobile setting, we have implemented an encryption module [2] that may be plugged into any Prism-MW connector. Finally, we have developed multi-versioning connectors (MVCs) [27] to support reliable upgrades of application functionality at runtime. An MVC allows multiple component versions to execute in parallel, such that their presence does not affect the application’s functionality. MVCs collect the execution statistics (relative correctness, reliability, and performance) and monitor the volume of incoming and outgoing message traffic for each component version. These statistics provide information about which version to retain in the system. For example, if a multi-versioned component is likely to be migrated to another device, its most desirable version need not be the most correct one, but rather the version with the lowest dependency factor.

Using the Framework. The first step a developer takes is to subclass from the Component or ComponentThread middleware classes for all components in the architecture and to implement the application-specific functionality for them. The next step is to instantiate the Architecture classes for each device and define the needed instances of thus created components, as well as the connectors selected from the connector library. Finally, attaching components and connectors into a configuration is achieved by using the weld and peerWeld methods of the Architecture class. At any point, the developer may add meta-level components, which may be welded to specific application-level connectors and thus exercise control over a particular portion of the Architecture (e.g., Admin Component in Figure 4). Alternatively, meta-level components may remain unwelded and may instead exercise control over the entire Architecture object (e.g., Continuous Analysis component in Figure 4).

Example Application. Figure 4 shows an implementation configuration of the TDS architecture distributed across five devices. Each device is running Prism-MW (depicted in the bottom planes of the five diagrams) as the local subsystem’s execution substrate. The three Palms run the Java KVM version of the middleware, the iPAQ runs the EVC++ version, and the PC runs the Java JVM and C++ versions of the middleware. As discussed above, we use XML-enabled connectors to facilitate the interaction of application components across the PL boundaries.

4.2 Deployment Support

We have integrated and extended the Microsoft Visio tool to develop Prism-DE, the Prism architectural modeling and deployment environment (see Figure 2). Prism-DE contains several toolboxes (shown on the left side of Figure 2). The top toolbox enables an architect to specify a configuration of hardware devices by dragging their icons onto the canvas and connecting them. The remaining toolboxes supply the software components and connectors that may be placed atop the hardware device icons. Once a desired software configuration is created in Prism-DE, it can be deployed onto the depicted hardware configuration with a simple button click. We currently assume that the locations of the compiled code for all the needed components and connectors are known, and specified inside Prism-DE. Additionally, Prism-DE currently assumes that the network address of each device is known; in the future, we plan to extend Prism-DE with support for automated discovery of network nodes. Finally, we assume that Prism-MW and a skeleton configuration consisting of an Admin Component and a skeleton configuration consisting of an Admin Component and a
Border Connector are preloaded on each device to facilitate the deployment.

Prism-DE creates a description of the deployment configuration (shown in Figure 5) and directly invokes the skeleton configuration on its local device. The skeleton configuration’s Admin Component waits for each device specified in the hardware configuration to connect, reads the description generated by Prism-DE, and sends appropriate messages to Admin Components residing on the connected devices. Each Admin Component receives a set of compiled code locations for components and connectors (the source parameter of the add command in Figure 5), and information about where in the architecture’s topology the components and connectors should be placed (weld and peerWeld commands). If the desired architectural element cannot be directly instantiated from the framework and is not available locally, a local Admin Component requests the element’s compiled code from the Admin Component on the device that contains it. The code is sent to the requesting Admin Component using a ComponentContent message. The requesting Admin Component instantiates the component or connector, and invokes the local Architecture object’s add, weld, and/or peerWeld methods to insert it into the local configuration.

4.3 Mobility and Dynamic Reconfiguration Support
If, during the application’s execution, a desired property is violated (e.g., as indicated by a meta-level component), the architecture may decide to reconfigure itself. For example, if TDS’s centralized Strategy Analyzer component creates a bottleneck, PC’s Admin Component may send copies of it across the network to locally perform analyses of proposed troop deployments. In the Java implementation of the framework, this amounts to the following process:

1. If necessary, the migrant component is disconnected from its attached connectors using the framework’s unweld and peerUnweld methods. In our example, since separate copies of Strategy Analyzer are being sent, PC’s Admin Component does not need to disconnect the local Strategy Analyzer.
2. The source device’s Admin Component may unload the migrant component from the local subsystem using the framework’s remove method or it may access the compiled image of the migrant component from a local file.
3. The source device’s Admin Component serializes the migrant component into a byte stream and sends it as a ComponentContent message via its Border Connector to the attached devices.
4. Once received by the Border Connectors on the destination devices, the ComponentContent message is forwarded to the Admin Component running on each device. Each Admin Component reconstitutes the migrant component from the byte stream contained in the message.
5. Each Admin Component invokes the add, weld, and peerWeld methods on its Architecture object to attach the received migrant component to the appropriate connectors (as specified in the ComponentContent message) in its local subsystem.

The process described above relies on the existence of Java serialization-like mechanisms. Such mechanisms are not provided by all PLs. Furthermore, even Java implementations on certain platforms do not support serialization. We have encountered this latter limitation in Java KVM for the PalmOS. For this reason, we have considered other possible mobility techniques [20], adopting one that directly exploits Prism’s message passing: the compiled image of the migrant component (e.g., a collection of Java class files) is sent across the network as a byte stream packaged in a ComponentContent message. This meta-level message is accompanied by a set of ApplicationData messages needed to bring the state of the migrant component to a desired point in its execution (see [27] for details of how such messages are captured). Once the migrant component is received at its destination, it is loaded into memory and added to the Architecture object by the local Admin Component, but is not attached to the appropriate connectors. Instead, the migrant component is stimulated by the ApplicationData messages sent with it: the Admin Component invokes the Architecture object, which in turn spawns a thread used to issue request, notification, and peer messages to the migrant component; any messages the migrant component issues in response are not propagated, but are “trapped” by the Architecture. Only after the migrant component is brought to the desired state is it welded and enabled to exchange messages with components in the local Architecture. This technique is less efficient than the serialization-based migration scheme and using it may prove prohibitively expensive in certain situations (see Section 5 for some benchmarks on the costs associated with message passing in Prism-MW). However, the technique is simple, PL independent, and natively supported by our middleware.

4.4 Support for Disconnected Operation
We have implemented the disconnected operation facilities described in Section 3.6 in a Prism-MW’s meta-level component called Disconnection Controller. Disconnection Controllers reside on each device and collaborate in estimating the best set of migrating components by calculating their benefits of migration. We illustrate our approach in the context of the TDS application. Let us assume that General’s Palm (recall Figures 1 and 2) is going to get disconnected within a given period, and that we know how much dynamic memory remains unused on the Palm. Also, let us assume that the connection speed between the Palm and the Headquarters PC is known. The goal is to maximize the functionality of the application running on General’s Palm until the connection is restored. Depending on the time to disconnection, message frequencies, and available memory on the device, the selected set of components for migration will vary, such that different subsets of PC’s Strategy-Analyzer, WarManager, and DeploymentAdvisor components will be selected for migration [22].

5 EVALUATION
In order to assess the suitability of our approach for developing applications in the Prism setting, we have undertaken a series of tests on both our design and implementation support. As part of two graduate courses at USC, the Prism style and middleware have been used in the development of over a dozen applications.

2. Several implementation-level details of this process are elided for brevity. Also elided are the issues of ensuring application integrity during the dynamic adaptation (see [23]). Application integrity is achieved in our approach by exchanging ArchitecturalModel messages along with ComponentContent messages and leveraging Continuous Analysis meta-level components (recall Section 3.4).

3. Java KVM does not support dynamic loading of classes, forcing us to extend KVM with a third-party class loader. We have also added this support to the EVC++ version of Prism-MW using DLLs.
involving PalmOS- and WindowsCE-compatible devices, digital cameras, and motion sensors. These applications included distributed digital image processing, map visualization and navigation, location tracking, and instant messaging for hand-held devices. Additionally, we have implemented a series of benchmarking applications containing large numbers of components, connectors, and exchanged messages, and utilizing our meta-level support for mobility, dynamic reconfiguration, deployment, and disconnected operation [22]. Finally, two major industrial organizations, the first developing military ground vehicle systems and the second developing avionics systems, have completed independent preliminary evaluations of the work described in this paper. In addition, the Java implementation of Prism-MW [25] has been successfully stress-tested by the second organization for use in one of their key embedded, dynamic, highly distributed avionics systems.

The anecdotal evidence from the two graduate courses and positive reactions from the external organizations suggest that the Prism style is well suited for application development “in-the-small-and-many.” However, in order to perform a more principled evaluation of the style, we have applied Fielding’s framework for comparing architectural styles for network-based systems [9]. The results of this evaluation are summarized in Table 1. This evaluation leverages Fielding’s own evaluation of the C2 style, on which Prism is based.

### Table 1: Evaluation of the Prism architectural style

<table>
<thead>
<tr>
<th>Property</th>
<th>Support</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>++</td>
<td>Minimal component interdependencies make possible numerous optimizations (e.g., recall the discussion of message processing from Section 3).</td>
</tr>
<tr>
<td>Network Efficiency</td>
<td>+</td>
<td>Support for mobility, dynamic reconfiguration and disconnected operation can be utilized to minimize network traffic.</td>
</tr>
<tr>
<td>Scalability</td>
<td>+</td>
<td>The style does not restrict the numbers of devices, processes, threads, components, connectors, or messages.</td>
</tr>
<tr>
<td>Simplicity</td>
<td>+</td>
<td>Simple topological rules. Separation of concerns via explicit components and connectors.</td>
</tr>
<tr>
<td>Evolvability</td>
<td>++</td>
<td>Via component decoupling and flexible (e.g., multiversioning) connectors.</td>
</tr>
<tr>
<td>Extensibility</td>
<td>++</td>
<td>Via component decoupling and support for mobility and dynamic reconfiguration.</td>
</tr>
<tr>
<td>Customizability</td>
<td>–</td>
<td>Customization of individual components at runtime is not supported; wholesale component replacement is required.</td>
</tr>
<tr>
<td>Configurability</td>
<td>+</td>
<td>Via deployment and dynamic reconfiguration support.</td>
</tr>
<tr>
<td>Reusability</td>
<td>++</td>
<td>Message-based interaction mediated by connectors and substrate independence minimize coupling between components.</td>
</tr>
<tr>
<td>Portability</td>
<td>++</td>
<td>Via support for mobility and PL independence.</td>
</tr>
<tr>
<td>Reliability</td>
<td>++</td>
<td>Via meta-level services that can monitor and react to various application context changes.</td>
</tr>
</tbody>
</table>

In addition to evaluating the style, we have performed extensive benchmark tests on Prism-MW to determine how well it addresses issues such as performance, memory footprint, and scalability [22]. In the interest of space, in this section we will only present a subset of the obtained results for the Java implementation of Prism-MW. One benchmarking application consisted of 50 identical components above and one component below a single connector. The application used a pool of 50 shepherd threads and a queue of 30 messages. 100,000 simple (parameter-less) messages were sent by the bottom component to all top components, resulting in 5,000,000 handled messages. The benchmark was performed on an Intel Pentium III 500 MHz processor with 256 MB of RAM running JDK 1.4 beta 2 on Microsoft Windows 2000. The time required to complete the exchange of messages was 4.7 seconds. Memory usage of Prism-MW, recorded at the time of architecture initialization, is 1.35 KB; such a small size is critical in resource-constrained environments for which the middleware is intended. The overhead of a “base” Prism-MW component, without any application-specific methods or state, is 0.8 KB. Memory overhead of creating and sending a single message can be estimated using the following formula, obtained empirically:

$$\text{Memory Overhead} = 0.46 + 0.02 \times \text{number of parameters}$$

The formula assumes that the parameters do not contain complex objects, but may contain simple objects (e.g., Java Integer or String). Therefore, for example, the memory overhead induced by using the Prism-MW in our benchmark application described above (assuming the message queue is full) is approximately

$$1.35 + (51 \times 0.8) + (30 \times (0.46 + (0.02 \times 0))) = 56 \text{ KB}$$

We have also assessed the memory usage of the meta-level components. Admin Components are very lightweight and at instantiation time occupy under 3 KB. Depending on the kind of analysis it has to perform (e.g., topological constraint checking or component compliance checking, or both), the size of the unoptimized meta-level Continuous Analysis component (each of which includes a third-party Prism-ADL parser) ranges between 72 KB and 95 KB. Finally, the size of the DisconnectionController component without state synchronization capabilities is about 9KB.

We have assessed the scalability of Prism-MW in the numbers of devices, components, connectors, threads, and messages. The Prism style does not require centralization, such as having a single connector in the system (unlike, e.g., CORBA [36]). For this reason, the number of devices supported by Prism-MW is unlimited in principle. We are currently developing a simulation environment that will allow us to test this aspect of scalability more easily (i.e., without having to have a large number of physical devices available). The number of components on a given device is limited and can be estimated using the following simple formula:

$$n = (M \cdot MS) / \text{ACS}$$

where $M$ is the available memory on the device, $MS$ is the memory occupied by the middleware (in this case, only 1.35 KB), and $ACS$ is the average component size. Our middleware supports as many threads as the underlying platform supports. Finally, the number of messages supported by Prism-MW is not limited by the middleware itself, but by the properties of the underlying hardware platform. This limit can be characterized by the following two parameters: (1) the maximum number of messages that can simultaneously be present in a system and (2) the rate of message delivery. The maximum number of messages is limited by the available memory on a given host (or set of hosts) and message size (discussed above), while the rate of message delivery depends on the CPU speed, the number of threads.

4. In this sense, the measure represents the minimum message overhead. Note that the possible use of complex objects as message parameters is independent of the middleware, but is an application-level decision.
threads servicing the message queue, the ratio of message produc-
tion to consumption by the components, and the network
bandwidth for messages that traverse machine boundaries.

6 RELATED WORK

Our work on Prism has been primarily influenced by four
research areas: architectural styles, middleware, code mobility,
and disconnected operation. Below we discuss the related
approaches in these four areas.

6.1 Architectural Styles

Several good overviews of architectural styles exist [9,15,32].
In particular, Fielding [9] studies architectural styles for distrib-
uted applications. Most of those styles are variants of the client-
server style and make certain assumptions that make them a
poor fit for Prism applications. These assumptions include one
or more of the following: centralized ownership of the applica-
tions; purely client-server or peer-to-peer interaction (but not
both); lack of topological guidelines for decomposing the archi-
tecture of the application and/or its major components (e.g., cli-
ents or servers); lack of support for formal architectural
modeling and analysis; and limited architectural self-awareness,
focus on mobility, and support for disconnected operation. Our
goal is to tailor the assumptions and characteristics of the Prism
style to address these issues in a principled way.

6.2 Middleware

Central to our investigation of the issues in Prism is our mid-
dleware. The research and use of middleware can be classified into
six distinct generations on the basis of the achieved level of
component reuse: (1) Module interconnection languages [7]
enabled the reuse of components implemented in a single PL.
(2) RPC and platform-neutral data representations (e.g., [3,28])
enabled distribution and reuse across PLs. (3) Platform-neutral
runtime environments and dynamic component loading (e.g.,
[13]) enabled dynamism and reuse across computing platforms.
(4) Domain-specific and GUI frameworks (e.g., [14,26])
enabled reuse across applications. (5) Provision of infrastruc-
ture services such as naming, threading, persistence, and trans-
action management (e.g., [17,30,36]) introduced the possibility
of reuse of architecture-level abstractions. (6) Reuse of archi-
tecture-level abstractions became an explicit focus of architectural
style-based frameworks (e.g., [31,34]). While it exhibits prop-
erties of middleware spanning several generations, Prism-MW is
most closely related to the sixth generation.

6.3 Code Mobility

A detailed overview of existing code mobility techniques is
given in [12]. Fuggetta et al. describe three code mobility para-
digms: remote evaluation, mobile agent, and code-on-demand.
Remote evaluation allows the proactive shipping of code to a
remote host in order to be executed. Mobile agents are autono-
ymous objects that carry their state and code, and proactively
move across the network. In the code-on-demand paradigm, the
client owns the resources (e.g., data) needed for the execution of
a service, but lacks the functionality needed to perform the ser-
vice. In this paradigm, the desired component can be retrieved
from a remote host, which acts as a code repository, and then
executed on the client. As described in Section 4.3, our work
primarily supports the code-on-demand technique.

Existing mobile code systems offer two forms of mobility.
Strong mobility allows migration of both the code and the state
of an execution unit to a different computational environment.
Weak mobility allows code transfers across different environ-
ments; the code may be accompanied by some initialization
data, but the execution state is not migrated. Our approach sup-
ports both forms of mobility: strong mobility is supported
through Java-like component serialization; weak mobility is sup-
ported by the use of compiled images as migrant compo-
nents, and application-level messages as a way of bringing a com-
ponent to its desired state (recall Section 4.3).

6.4 Disconnected Operation

Ensuring availability of a system during disconnection has been
explored primarily in the domain of file systems. The approach
is to make the mobile computer less dependent on the network
by using such methods as file caching or prefetching, and lazy
writeback. Example systems such as Coda [18], D-NFS [11],
and Ficus [16] use optimistic replication for file caching, and
reconciliation of replicas to resolve conflicting updates. In opti-
mistic replication, updates can be made concurrently to differ-
ent file replicas, resulting in multiple versions of a file. To
recover from conflicting updates, after-the-fact conflict resolu-
tion (i.e., reconciliation) actions are required to recombine mul-
tiple file versions into one. Conflict resolution can be automated
[29], but it may also require the intervention of the (human)
owner of the file.

Our approach to disconnected operation is more similar in its
nature to FarGo-DA [35], which supports migration of compo-
nents as computational elements, rather than as files, in
response to disconnection. However, while FarGo-DA handles
only anticipated disconnection, our approach can also be used
in cases of unanticipated disconnection.

7 CONCLUSIONS AND FUTURE WORK

Over the past several decades software researchers and practi-
tioners have proposed various approaches, techniques, and tools
for developing ever larger, more complex systems. The results
of these efforts have shared a number of traits: system size and
complexity, possible distribution across desktop platforms,
focus on modeling and analysis before implementation, accom-
panying development environments, explicit software architec-
tures, and so on. The resulting development paradigm has been
referred to as programming-in-the-large (PitL) [7]. This paper
has presented an approach to address a new set of software
engineering challenges that have arisen with the proliferation
inexpensive, small, heterogeneous, resource-constrained, possi-
ibly embedded, highly-distributed, and highly-mobile comput-
ing platforms. While a number of the individual challenges bear
similarity to those addressed by PitL, we believe that their com-
bination and overall novelty is more appropriately described as
programming-in-the-small-and-many (Prism).

The centerpiece of our approach to Prism is an architectural
style. The style and its accompanying modeling, analysis, and
implementation tools ensure flexible component-based system
composition and interaction; efficient implementation; fine-
grained distribution and deployment; dynamic reconfiguration;
mobility of system models, data, and code; and continued avail-
ability in the face of connectivity losses. Additionally, the Prism
architectural style introduces facilities for system self-aware-
ness, which are leveraged in the development and evolution of
long lived, highly distributed, dynamically evolving systems
whose ownership is potentially decentralized. We have pro-
vided support for these capabilities in several PLs and comput-
ing platforms (both desktop and mobile). The Prism style and
tools have been evaluated both analytically and empirically.

While our experience thus far has been very positive, this work is still preliminary and a number of pertinent issues remain unexplored. Our future work will span issues such as ensuring trust in Prism applications, supporting configuration management of the many involved artifacts, and automatically discovering the hardware devices and software components available on the network at a given time. We discuss two additional areas of our most immediate interest in more detail below.

As applications are moving to highly distributed topologies, issues such as decentralized ownership need to be properly addressed. We have only begun to investigate possible solutions for supporting decentralized ownership of an application. Our current solution is that each device stores its subsystem’s architectural model. Before a dynamic change to the application is allowed, the model is analyzed to ensure the consistency of that change with the existing configuration. A limitation of this solution is that it assumes that each device has local analysis capabilities. It also induces decisions about architectural changes based solely on local information. An alternative is to allow a device to communicate its own application model to neighboring device(s) equipped with the needed analysis facilities. As a result, such devices may have access to a more complete model of the overall system’s architecture and thus may be able to perform more meaningful analyses. On the other hand, such a collaborative approach to analysis will accentuate the issues of system security and trust. We intend to study the applicability of and tradeoffs between these two alternatives.

Another critical issue associated with highly distributed, mobile, possibly embedded systems is performance [19]. To date, we have deliberately chosen to provide developer support for Prism in mainstream, general-purpose PLs. However, these PLs are typically not optimized for resource-constrained and embedded devices, inducing performance penalties that may be unacceptable for certain classes of applications. We thus intend to explore the use of optimized compilation [1] and special-purpose languages for embedded systems [19] to address this problem. Our longer term goal is to develop techniques for actively assessing Prism applications and suggesting deployment strategies that minimize network traffic and maximize performance and availability. We intend to integrate Prism’s support for architectural self-awareness and runtime monitoring with existing tools for system resource analysis (e.g., [8]) in order to enable these estimations.

8 REFERENCES
25. Prism. [http://sunset.usc.edu/~softarch/Prism/](http://sunset.usc.edu/~softarch/Prism/)