Architecture-Level Support for Software Component Deployment in Resource Constrained Environments

Marija Mikic-Rakic and Nenad Medvidovic

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

Abstract. Software deployment comprises activities for installing or updating an already implemented software system. These activities include (1) deployment of a system onto a new host, (2) component upgrade in an existing system, (3) static analysis of the proposed system configuration, and (4) dynamic analysis of the configuration after the deployment. In this paper, we describe an approach that supports all four of these activities. The approach is specifically intended to support software deployment onto networks of distributed, mobile, highly resource constrained devices. Our approach is based on the principles of software architectures. In particular, we leverage our lightweight architectural implementation infrastructure to natively support deployment in resource constrained environments.

Keywords. Software deployment, software architecture, architectural style, software connector, multi-versioning, Prism

1 Introduction

Software deployment involves the activities needed for installing or updating a software system, after the software has been developed and made available for release. Software systems of today often have complex architectures that are characterized by large numbers of potentially pre-fabricated ("off-the-shelf") components. Over a long lifespan, these systems are likely to have many installed versions and experience numerous and frequent updates. The components that comprise these systems are more commonly being developed and released independently by third-party organizations (i.e., vendors). The vendors of components are themselves usually distributed, heterogeneous, and their locations may change over time. Due to the time-to-market pressures, component vendors frequently release new versions that need to be deployed to large numbers of sites. However, the vendors usually have no control over a majority of the systems in which their deployed components reside. Moreover, multiple versions of such systems (referred to as *target* systems below) are likely to exist in various locations simultaneously. For these reasons, human-operated deployment becomes impossible, and support for automated software deployment becomes critical. This picture has become even more complex with the recent emergence of inexpensive, small, heterogeneous, resource-constrained, highly

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 31–50.

distributed, and mobile computing platforms that demand highly efficient software deployment solutions.

If considered from the perspective of the effects of deployment on the target system, software deployment deals with at least four problems. These problems are independent of the application domain, or nature and configuration of hardware platforms on which the software is to be deployed. The four problems are:

1. initial deployment of a system onto a new host (or set of hosts);
2. deployment of a new version of a component to an existing target system;
3. static analysis, prior to the deployment, of the likely effects of the desired modifications on the target system; and
4. dynamic analysis, after the deployment, of the effects of the performed modifications on the running target system.

Several existing approaches have been aimed at providing support for one or more of these four activities. Typically, software deployment has been accomplished via large-scale "patches" that replace an entire application or set of applications (e.g., new version of MS Word or MS Office). These patches do not provide control over the deployment process beyond the selection of optional features available for a given application (e.g., optional installation of MS Equation Editor in MS Office). Some existing approaches (e.g., [5]), have addressed this problem by providing support for deployment at a finer-grain component level. However, these approaches have typically taken a configuration management perspective on the deployment problem, tracking dependencies among versions of *implemented* modules. With one notable exception [6], these approaches have rarely taken into account system architectures, their evolution over time due to the frequent component upgrades, or the relationship of the deployed multiple versions of a given system to that system's architecture. Furthermore, these approaches have often required sophisticated deployment support (e.g., deployment agents [5]) that uses its own set of facilities, provided separately from the application's implementation infrastructure, thereby introducing additional overhead to the target host. For these reasons, these approaches are usually not applicable in an emerging class of light-weight, resource constrained, highly distributed, and mobile computational environments.

In this paper, we propose an approach that attempts to overcome the shortcomings of previous work and address all four deployment problems discussed above. Our approach directly leverages a software system's architecture in enabling deployment. Specifically, we have been able to adapt our existing architectural implementation infrastructure to natively and inexpensively support deployment, both at system construction-time and run-time. Our solution is light weight and is applicable in a highly distributed, mobile, resource constrained, possibly embedded environment. A key aspect of the approach is its support for intelligent, dynamic upgrades of component versions. We have provided a graphical software deployment environment, and have evaluated our approach on a series of applications distributed across a variety of desktop, lap-top, and hand-held devices.

The rest of the paper is organized as follows. Section 2 briefly describes an example application used to illustrate the concepts throughout the paper. Section 3 summarizes the architectural style used as the basis of this work and its accompanying infrastructure for implementing, deploying, migrating, and dynamically reconfiguring

applications. Section 4 discusses our approach to supporting component deployment, while Section 5 describes in more detail a technique for supporting upgrades of component versions. The paper concludes with overviews of related and future work.

2 Example Application

To illustrate our approach, 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 a secure link 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

Fig. 1. TDS application distributed accross multiple devices.

Commanders use Palm Pilot Vx devices, while the third uses a Compaq iPAQ; the *General* uses a Palm Pilot VIIx. The *Commanders* are capable of viewing their own quadrant of the battlefield and deploying friendly troops within that quadrant to counter enemy deployments. The *General* sees a summarized view of the entire battlefield (shown); additionally, the *General* is capable of seeing detailed views of each quadrant. Based on the global battlefield situation, the *General* can issue direct troop deployment orders to individual *Commanders* or request transfers of troops among the *Commanders*. The *General* can also request for deployment strategy suggestions from *Headquarters*, based on current positions of enemy troops, mines, and the number of friendly troops at disposal. Finally, the *General* can issue a "fight" command, resulting in a battle simulation that incrementally determines the likely winner given a configuration of troops and obstacles.

The TDS application provides an effective platform for illustrating our ideas. It has been designed, implemented, and deployed using the approach described in this paper. In the instance of TDS shown in Figures 1 and 2, sixteen software components deployed across the five devices interact via fifteen software connectors.

3 Architectural Basis for Software Deployment

We are using a software architectural approach as the basis of our support for deployment. *Software architectures* provide high-level abstractions for representing structure, behavior, and key properties of a software system [18]. They are described in terms of components, connectors, and configurations. Architectural *components* describe the computations and state of a system; *connectors* describe the rules and mechanisms of interaction among the components; finally, *configurations* define topologies of components and connectors. Software *architectural styles* involve identifying the types of elements from which systems are built, the characteristics of these elements, the allowed interactions among the elements, and the patterns that guide their composition [15]. The specific architectural style upon which we are relying in this work is Prism [10].

3.1 Architectural Style

The Prism style is targeted at heterogeneous, highly distributed, highly mobile, resource constrained, possibly embedded systems. In formulating the Prism style, we have leveraged our extensive experience with the C2 architectural style, which is intended to support highly distributed applications in the graphical user interface (GUI) domain [19]. Prism-style *components* maintain state and perform application-specific computation. The components may not assume a shared address space, but instead interact with other components solely by exchanging messages via their three communication *ports* (named *top*, *bottom*, and *side*). *Connectors* in the Prism 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.

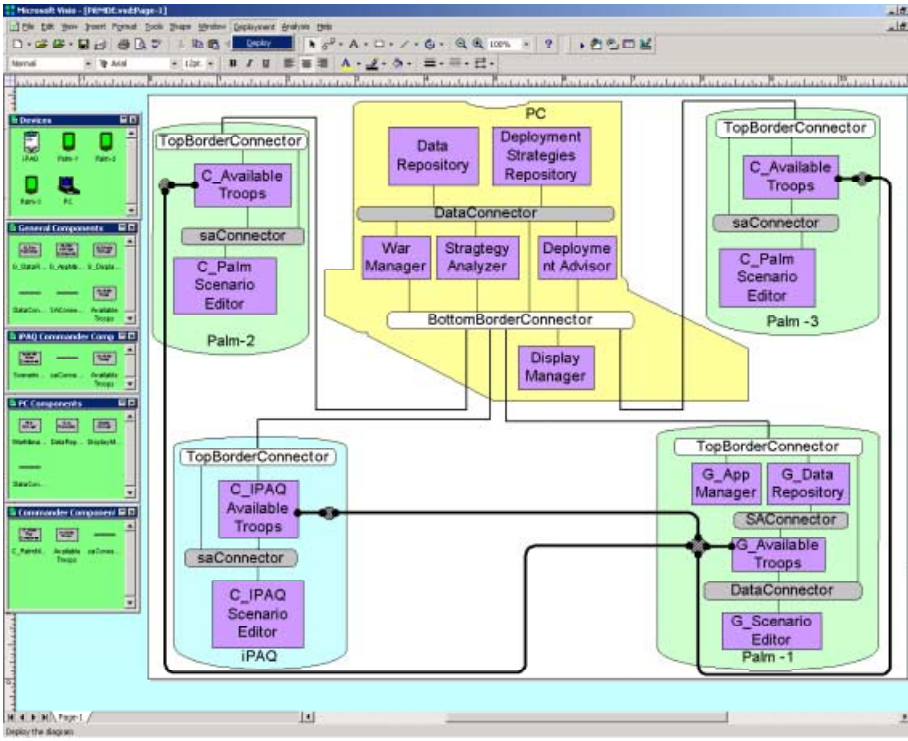

Fig. 2. Architecture of the TDS application, displayed in the Prism deployment environment (Prism-DE). The unlabeled circles connecting components across the hand-held devices represent peer connectors.

A message in the Prism style is either a *request* for a component to perform an operation, a *notification* that a given component has performed an operation and/or changed its state, or a *peer* message used in direct (peer-to-peer) communication between components. Request messages are sent through the top port, notifications through the bottom port, and peer messages through the side port of a component. The distinction between requests and notifications ensures Prism's principle of *substrate independence*, which mandates that a component in an architecture may have no knowledge of or dependencies on components below it. In order to preserve this property, two Prism components may not engage in interaction via peer messages if there exists a vertical topological relationship between them. 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* on the *iPAQ* and *G_AvailableTroops* on the *Palm-1*, meaning that they may communicate via peer messages.

3.2 Software Connectors

A Prism-style connector does not have an interface at declaration-time; instead, as components are attached to it, the connector's interface is dynamically updated to reflect the interfaces of the components that will communicate through the connector. This "polymorphic" property of connectors is the key enabler of our support for deployment, and run-time reconfiguration. Prism distinguishes between two types of connectors. *Horizontal* connectors enable the request-notification type of communication among components through their top and bottom ports, while *peer* connectors enable peer-to-peer communication among components through their side ports. The Prism style does not allow a peer and a horizontal connector to exchange messages; this would, in effect, convert peer messages into requests/notifications, and vice versa.

The Prism style directly supports connectors that span device boundaries. Such connectors, called *border connectors*, enable the interactions of components residing on one device with components on other devices (e.g., *BottomBorderConnector* on the *PC* in Figure 2). A border connector marshals and unmarshals data, code, and architectural models, and dispatches and receives messages across the network. It may also perform data compression for efficiency and encryption for security. A *Border Connector* may facilitate communication via requests and notifications (horizontal border connector) or via peer messages (peer border connector).

3.3 Architectural Modeling and Analysis

Prism 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 deployment, execution, dynamic evolution, and mobility of application-level components. Both application-level and meta-level components obey the rules of the style. They execute side-by-side: meta-level components are aware of application-level components and may initiate interactions with them, but not vice versa.

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 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 (e.g., during deployment).

We have extensively used special-purpose components, called *Admin Component*s, whose task is to exchange *ComponentContent* messages and facilitate the deployment and mobility of application-level components across devices. 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 run-time architectural changes, and possibly disallowing the changes.

In support of this task, we have recently added architecture description language (ADL) support to Prism. We have extended our existing ADL (C2SADEL [11]) and

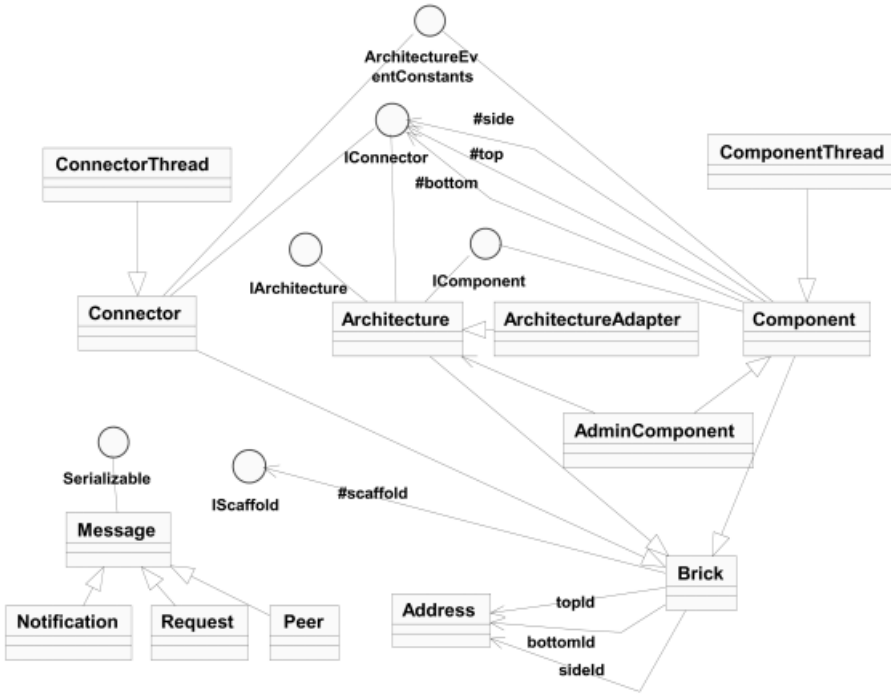

Fig. 3. Class design view of the Prism implementation framework.

tool support (DRADEL [11]) with a set of new capabilities for modeling and analyzing Prism-style architectures. Specifically, the *Continuous Analysis* component is built using a subset of DRADEL to assess the proposed architectural (re)configurations. That assessment is carried out by matching the interfaces and behaviors (expressed via component invariants and operation pre- and post-conditions) of the interacting components [11].

3.4 Architectural Implementation

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 architecture implementation infrastructure. The infrastructure comprises an extensible framework of implementation-level modules representing 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 then constructed by extending the appropriate classes in the framework with application-specific detail. The framework has been implemented in several programming languages: Java JVM and KVM [20], C++ and Embedded Visual C++ (EVC++), and Python.

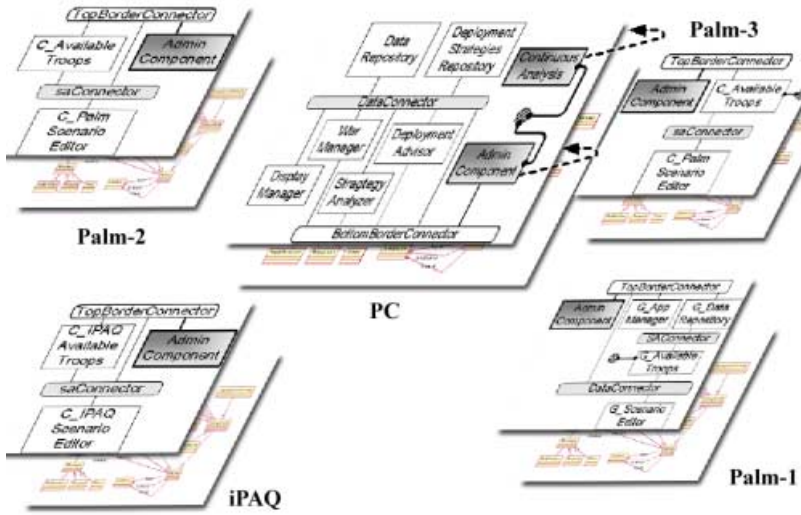

Fig. 4. Layered construction of an application using the Prism implementation framework. The application is distributed across five devices, each of which is running the framework. Meta-level components (highlighted in the figure) may control the execution of application-level components via Prism messages or via pointers to the local *Architecture* object (shown in the subarchitecture on the *PC*).

A subset of the Prism framework's UML class diagram is shown in Figure 3. The classes shown are those of interest to the user of the framework (i.e., 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*). 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 run-time. A distributed application, such as TDS, is implemented as a set of interacting *Architecture* objects as shown in Figure 4.

The first step a developer (or tool generating an implementation from an architectural description [11]) takes is to subclass from the *Component* or ComponentThread framework 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 directly (e.g., *Continuous Analysis* component in Figure 4).

4 System Deployment

Our support for deployment addresses problems 1 and 3 stated in the Introduction (initial system deployment and static analysis prior to deployment) by directly leveraging the Prism implementation infrastructure. We have developed a custom solution for deploying applications instead of trying to reuse existing capabilities (e.g., [5]) because of the facilities provided by the Prism style (*ComponentContent* messages, *Admin Component*s, meta-level architecture) and the light weight of the Prism implementation framework, which is critical for small, resource constrained devices (e.g., Palm Pilot, which has 256KB of dynamic heap memory). We use the same infrastructure for the initial deployment of a system onto a new host (or set of hosts) and for deploying a new version of a component to an existing target system.

4.1 Basic Requirements

In order to deploy the desired architecture on a set of target hosts, we assume that a skeleton (meta-level) configuration is preloaded on each host. The configuration consists of the Prism implementation framework, with an instantiated *Architecture* object that contains a *Border Connector* and an *Admin Component* attached to the connector. The skeleton configuration is extremely lightweight. For example, in our Java implementation, the skeleton uses as little as 8KB of dynamic memory. Since the Prism framework, *Architecture* object, and *Border Connector* are also used at the application level, the actual memory overhead of our basic deployment support (i.e., the *Admin Component*) is only around 3KB.

The *Admin Component* on each device contains a pointer to its *Architecture* object and is thus able to effect run-time changes to its local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. *Admin Component*s are able to send and receive from any device to which they are connected the meta-level *ComponentContent* messages through *Border Connector*s. Each *Admin Component* can request the components that are to be deployed in its subsystem's architecture. Finally, each *Admin Component* has the knowledge of the set of components that are available in the local configuration via a pointer to the *Architecture* object.

4.2 Deployment Process

The *current* configuration of a system describes its current topology. The *desired* configuration represents a configuration that needs to be deployed. If there is a difference between the current and desired configurations, the deployment process will be initiated. The information about the current and desired configurations can either be stored on a single host (*centralized* ownership) or each subsystem may have the knowledge of

its current and desired configurations (*distributed* ownership). In the first scenario, the *Admin Component* on the host storing the descriptions of configurations will

initiate the deployment on all hosts. In the second scenario, each *Admin Component* will initiate the deployment on its local host.

```
architecture TDS is {
     component_types {
      component PC_StrategyAnalyzer is extern {
         C:\spec\PC_StrategyAnalyzer.Prism;}
      ...
      component C_AvailableTroops is extern {
         C:\spec\C_AvailableTroops.Prism;}
      ...}}
   architectural_topology {
      component_instances {
         pcStratAnalyzer : PC_StrategyAnalyzer;
         pcDataRepository : PC_DataRepository;
         cAvailableTroops: C_AvailableTroops;
         cAvailableTroops1: C_AvailableTroops;}
      connector_instances {
         BottomBorderConn : RegularConn;}
      peer_connector_instances {
      PeerCon : PeerConn;}
      connections {
         connector BottomBorderConn {
               top  pcStratAnalyzer;
               bottom pcDataRepository;}}
      peer_connections{
      peer_connector PeerCon {
      side cAvailableTroops, cAvailableTroops1;}}
}}
```

Fig. 5. Partial architectural specification of the TDS application in the Prism ADL. Individual components are specified in separate files denoted by the extern keyword.

In order to support centralized ownership of the application's architecture, the skeleton configuration on the central host should also contain a *Continuous Analysis* component. The centralized *Continuous Analysis* component has the knowledge of the current and desired configurations for all subsystems. Figure 5 shows the partial description of the TDS application configuration used by the central *Continuous Analysis* component.

Distributed ownership of the application's architecture requires that the skeleton configuration on each host contain a *Continuous Analysis* component (attached to the local *Border Connector*), which is aware of its subsystem configuration. Each *Continuous Analysis* component is capable of analyzing the validity of architectural configurations either by performing the analysis locally or by requesting the analysis of a given configuration remotely (e.g., on a more capacious host that can perform such analysis). Additionally, the *Continuous Analysis* component is capable of storing the desired (but not yet deployed) local configuration.

```
add(DataRepository: source PC): PC
add(DeploymentStrategiesRepository: source PC): PC
add(DataConnector: source none): PC
add(C_IPAQAvailableTroops: source local): iPAQ
add(C_IPAQScenarioEditor: source PC): iPAQ
add(SaConnector: source none): iPAQ
weld(DataRepository,DataConnector): PC
weld(DeploymentStrategiesRepository,DataConnector): PC
weld(C_IPAQAvailableTroops,SaConnector): iPAQ
weld(TopBorderConnector,C_IPAQAvailableTroops): iPAQ
weld(SaConnector,C_IPAQScenarioEditor): iPAQ
peerWeld(G_AvailableTroops,SideBorderConnector):Palm-1
```

Fig. 6. Partial description of the configuration created by Prism-DE for the TDS application. Component source and destination devices are shown. Sources are denoted as "none" in the case of connectors that use the base implementation (asynchronous message broadcast) provided by the Prism framework.

Table 1.

Centralized Ownership	Distributed Ownership
1. The central *Continuous Analysis* component receives the desired configuration as part of an *ArchitecturalModel* message, analyzes it, and after ensuring that the configuration is valid, invokes the *Admin Component* on the local host.	1. Each *Continuous Analysis* component receives the desired configuration of the local subsystem, analyzes it, and, after ensuring that the configuration is valid, invokes the *Admin Component* on the local host.
2. The *Admin Component* packages the components (e.g., .class files in Java) to be deployed into a byte stream and sends them as a series of *ComponentContent* messages via its local *Border Connector* to the target hosts.	2. *(a)* Each *Admin Component* issues a series of requests to other *Admin Component*s (using either request or peer messages) for a set of components that are to be deployed locally. *(b) AdminComponent*s that can service the requests package the required component(s) into byte streams and send them as a series of *ComponentContent* messages via their local *Border Connector*s to the requesting device.
3. Once received by the *Border Connector* on each destination device, the *ComponentContent* messages are forwarded to the *Admin Component* running locally. The *Admin Component* reconstitutes the migrant components from the byte stream contained in each message.	
4. Each *Admin Component* invokes the *add*, *weld*, and *peerWeld* methods on its *Architecture* object to attach the received components to the appropriate connectors (as specified in the *ComponentContent* message) in its local subsystem.	
5. Each *Admin Component* sends a message to inform the centralized *Continuous Analysis* component that the deployment has been performed successfully. The central *Continuous Analysis* component updates the model of the configuration accordingly. Update to the model is made only after successful deployments.	5. Each *Admin Component* sends a peer message to inform its local *Continuous Analysis* component (see Figure 4) that the deployment has been performed successfully. The local *Continuous Analysis* component updates the model of the local configuration accordingly.
6. Finally, each *Admin Component* invokes the *start* methods of all newly deployed components to initiate their execution.	

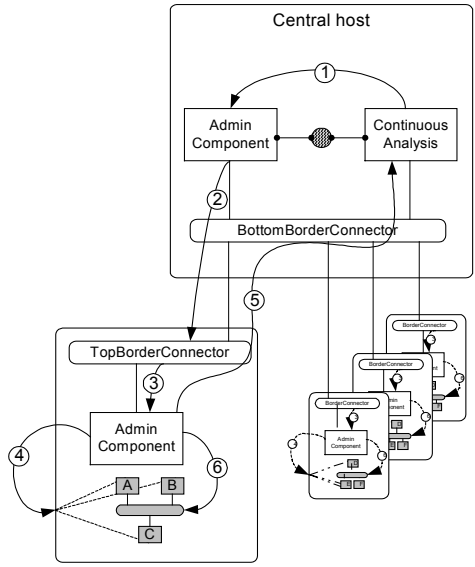

Fig. 7. Deployment process using centralized ownership.

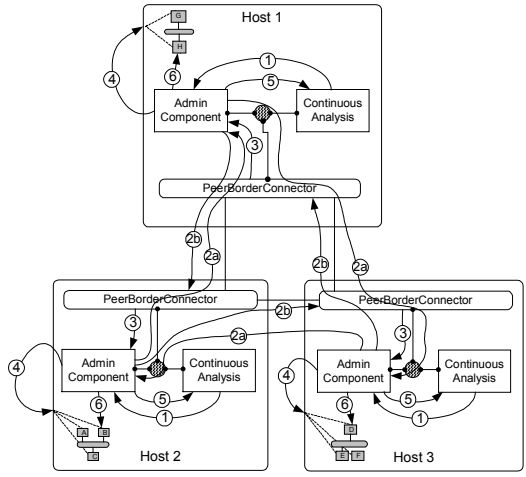

Fig. 8. Deployment process using distributed ownership.

Table 1 and Figures 7 and 8 describe the deployment process in the Java version of our implementation and deployment framework for both the cases of centralized and distributed ownership of the application's architecture.

4.3 Deployment Environment

We have integrated and extended the MS Visio tool to develop *Prism-DE*, the Prism architectural modeling and deployment environment (see Figure 2). Prism-DE con-

tains 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. Prism-DE currently assumes that the available devices and their locations are known; we are extending Prism-DE with support for automated discovery of network nodes. The remaining toolboxes in Figure 2 supply the software components and connectors that may be placed atop the hardware device icons. Once a desired software configuration is created and validated in Prism-DE, it can be deployed onto the depicted hardware configuration with a simple button click.

Prism-DE supports deployment using centralized ownership of the application's architecture. Recall that this is *not* the case with our implementation framework, which also supports decentralized ownership as discussed in Section 4.2. Our current implementation assumes that the locations of the compiled code for all the needed components and connectors are known, and specified inside Prism-DE. Each software component in the toolboxes is associated with its compiled code location and its architectural description. Once the desired configuration is created, its validity can be assured automatically: Prism-DE generates the appropriate architectural description in the Prism ADL (e.g., recall Figure 5) and invokes its internal *Continuous Analysis* component to ensure that the description is valid. Once the user requests that the architecture be deployed, Prism-DE generates a series of deployment commands as shown in Figure 6. These commands are used as invocations to the skeleton configuration on the device on which Prism-DE resides. The local *Admin Component* initiates the deployment process on all hosts as specified in the left column of Table 1.

5 Component Upgrade

Prism and its integrated analysis capabilities provide assurance that the specified configuration of components and connectors is valid according to the architectural specification. However, implemented components may not preserve all the properties and relationships established at the architectural level (e.g., due to accidental coding errors) [12]. This section describes a run-time approach to supporting reliable upgrades of existing component versions [2,16] and reliable deployment of new components, addressing problems 2 and 4 stated in the Introduction.

When upgrading a component, vendors try to maintain the old component version's key properties (e.g., granularity, implementation language, and interaction paradigm), while enhancing its functionality (by adding new features) and/or reliability (by fixing known bugs). However, component upgrades raise a set of questions, including whether the new version correctly preserves the functionality carried over from the old version, whether the new version introduces new errors, and whether there is any performance discrepancy between the old and new versions. Depending on the kinds of problems a new component version introduces and the remedies it provides for the old version's problems, this scenario can force a user to make some interesting choices:

- deploy the new version of the component to replace the old version;
- retain the old version; or
- deploy both the old and new versions of the component in the system.

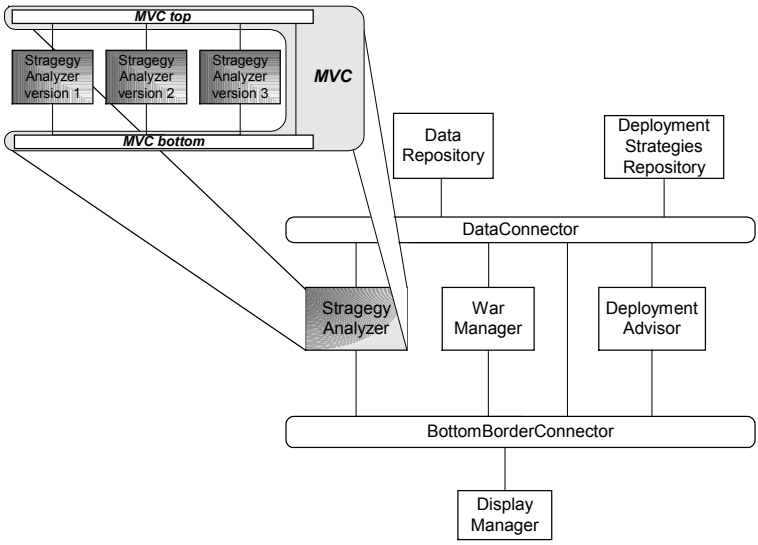

Fig. 9. Partial architecture of the TDS application. The internal structure of MVC is highlighted.

Prior to making one of these choices, the user must somehow assess the new component in the context of the environment within which the old component is running, and, once the choice is made, the running system needs to be updated.

```
Component: StrategyAnalyzer
Operations:
analyzeStrategy(int[][]):boolean;
calculateProbabilities(int[][]):int[][];
determineWinner(int[][]):String;
fight(int[][]):int[][];
```

Fig. 10. Operations of the Strategy Analyzer component.

5.1 Multi-Versioning Connectors

We illustrate our approach in the context of the example application described in Section 2. Figure 9 shows the partial architectural configuration of the TDS application. In this configuration, a *Strategy Analyzer* component provides several operations as illustrated in Figure 10. Let us assume that, after this application is deployed, we obtain two new versions of *Strategy Analyzer* that are claimed to be improvements over the old version. We would like to assess both new versions before deciding which one to deploy in our system.

Figure 9 depicts the essence of our approach: a component (*Strategy Analyzer*) is replaced by a set of its versions encapsulated in a wrapper. The wrapper serves as a connector between the encapsulated component versions and the rest of the system [13]. We say that *Strategy Analyzer* is multi-versioned and call the wrapper *multi-versioning connector (MVC)*. The MVC is responsible for "hiding" from the rest of the

system the fact that a given component exists in multiple versions. The role of the MVC is to relay to all component versions each invocation that it receives from the rest of the system, and to propagate the generated result(s) to the rest of the system. Each component version may produce some result in response to an invocation. The MVC allows a system's architect to specify the component authority [2] for different operations. A component designated as authoritative for a given operation will be considered nominally correct with respect to that operation. The MVC will propagate *only* the results from an authoritative version to the rest of the system. At the same time, the MVC will log the results of all the multi-versioned components' invocations and compare them to the results produced by the authoritative version.

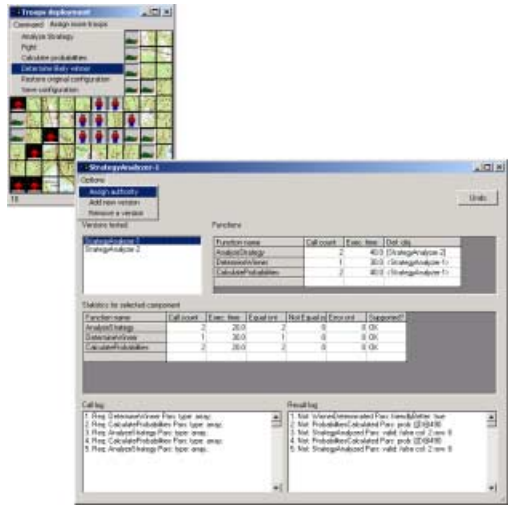

Fig. 11. MVC monitoring window (bottom) and a screenshot of the TDS application (top).

We are allowing authority specification to be at the level of the entire invocation domain (e.g., for each invocation, the entire component version *v1* will be considered nominally correct). We are also supporting authority specification at the level of individual operations (e.g., component version *v1* is authoritative for *analyzeStrategy*, while *v2* is authoritative for *calculateProbabilities*). For each possible invocation, we are assuming that there is going to be exactly one component designated as authoritative.

In addition to this "basic" functionality of insulating multi-versioned components from the rest of a system, the MVC provides several additional capabilities. It allows component authority for a given operation to be changed at any point. It also allows insertion of a new component version into the system during run-time without removing the old version. The MVC can monitor the execution of the multiple component versions and perform comparisons of their performance (i.e., execution speed), correctness (whether they are producing the same results as the authoritative version) and reliability (number of thrown exceptions and failures). Furthermore, the MVC logs the execution history as a sequence of invocations of the multi-versioned component. In case a failure has occurred, this information can be used to determine which

sequence of invocations has led to the failure. The MVC also periodically records the state of each component version. The execution history and state "snapshots" can be used to roll back the execution of a multi-versioned component to any point in its past [16].

MVC's monitoring mechanism (logging of component invocations, comparisons of their results, and component state "snapshots") can help a user decide among replacing the old component version with the new, retaining the old version, and simultaneously deploying multiple versions of a single component in the system. In the first two cases, the MVC can be removed from the system to reduce the overhead introduced by its insertion. In the last case, the MVC will be retained and used to "simulate" the functionality of a single conceptual component. In that case, the monitoring can be disabled to minimize the overhead.

To date, we have primarily focused on the use of our approach in the case of component upgrades. However, the same infrastructure can be used when an entirely new component needs to be reliably deployed into a system. The wrapped component can be deployed at the desired location in the system's architecture in the manner discussed in Section 4. The component's behavior can then be assessed with minimal disturbance to the rest of the system since the MVC will be configured to "trap" all the invocations the component tries to make. Once the new component is assessed in the context of the deployed system and it is established that the component produces satisfactory results, the wrapper around it (i.e., the MVC) may be removed.

5.2 Implementation of MVC

We have directly leveraged Prism's implementation infrastructure in constructing the MVC. In particular, we have implemented three special-purpose, reusable software connectors, called *MVC-Top*, *MVC-Bottom*, and *MVC-Side*. Each connector serves as an intermediary between the multi-versioned component and the corresponding port through which the component is attached to the rest of the system. Depending on which ports are used by the multi-versioned component, one, two, or all three MVC connectors would be required to create the wrapper. Figure 9 shows one possible wrapping scenario, in which the multiversioned component is communicating using the top and bottom, but not side ports. *MVC-Top* and *MVC-Bottom* connectors encapsulate multiple versions of a component, allowing their parallel execution and monitoring. The intrinsic support of the Prism framework for dynamic addition and removal of components [10,14] is leveraged in the context of the MVC to add and remove component versions during run-time.

When a message is sent to a multi-versioned component (e.g., *Strategy Analyzer* in Figure 9) from any component below the *MVC-Bottom* or above the *MVC-Top* connectors, the corresponding connector invokes within each component version the operation that is responsible for processing that message. Even though the operation is invoked on all the installed versions, only the messages generated by the authoritative version are propagated by the two MVC connectors to the rest of the system. In our example, whenever a *determineWinner* request message is sent from the *Display Manager* component, MVC-Bottom will return to *Display Manager* only the result produced by (the authoritative) version *v2*; the results produced by versions *v1* and *v3*

are compared with those of *v2* and logged, but are not propagated to the rest of the system.

The GUI of our implementation of the MVC is shown in the bottom window of Figure 11. This window is separate from an application's UI, such as that of TDS, partially depicted in the top window. The MVC window shows the list of component versions in the upper left frame. The table in the upper right frame shows the current authority specification and the total (cumulative) execution time, in milliseconds, for each invoked operation of a selected component version (in this case, version *v1* of *Strategy Analyzer*).

The table in the middle frame displays the execution statistics for the selected component version. For each operation, the table shows the number of times the operation has been invoked, the average execution time for that operation (-1 if the operation is not implemented by the component version), and the number of times the operation produced identical and different results in comparison to the authoritative version. The table also displays the number of times an error (an exception or a failure) occurred during the execution of the operation, and whether the invoked operation is implemented by the component version.

The bottom two frames in the MVC window display the call and result logs as sequences of generated messages. Using these logs, the *Undo* button can revert the states of a given set of multi-versioned components to any point in the past. This capability is achieved by taking "snapshots" of and storing the versions' states at regular intervals and by logging each message sent to the multi-versioned component. A detailed description of the undo process is given in [16].

The overhead of MVC is linearly proportional to the number of operations of the multiversioned component, and can be calculated using the following formula:

*Mem(MVC)=1208+44*num_op* (in bytes)

where *Mem(MVC)* is the memory usage of a single MVC and *num_op* is the total number of operations provided by the multi-versioned component. For example, the overhead of a single MVC for a component with 10 operations is around 1.6KB. The overhead of MVC's GUI is 680KB since it uses an off-the-shelf GUI framework (Java Swing). While this overhead is acceptable on desk-top and even some hand-held platforms (e.g., the iPAQ), it is too expensive for devices with significant resource constraints (e.g., the Palm Pilot). We are currently developing a simplified version of the UI that is targeted for such, less capacious platforms.

6 Related Work

In addition to software architectures, discussed in Section 3, this section outlines other relevant research. Carzaniga et. al. [1] proposed a comparison framework for software deployment techniques. They identified the activities in the software deployment process and provided an extensive comparison of existing approaches based on their coverage of the deployment activities. Our approach has similar coverage of the deployment process to application management systems [1]. Below, we briefly describe two approaches most closely related to ours.

Software Dock [5] is a system of loosely coupled, cooperating, distributed components. It supports software producers by providing a Release Dock and a Field Dock. The Release Dock acts as a repository of software system releases. The Field Dock supports a software consumer by providing an interface to the consumer's resources, configuration, and deployed software systems. The Software Dock employs agents that travel from a Release Dock to a Field Dock in order to perform specific software deployment tasks. A wide area event system connects Release Docks to Field Docks. The entire infrastructure of Software Dock introduces substantial overhead, and is therefore not directly applicable for the classes of applications that need to be deployed onto resource constrained devices.

Cook and Dage [2] have developed an approach to reliable software component upgrades that is most closely related to ours. Their component upgrade framework, HERCULES, treats only individual procedures as components, allows multiple such procedures to be executed simultaneously, and provides a means for comparing their execution results. Unlike our MVC, HERCULES does not provide any support for inserting and removing component versions at system run-time, or reverting a multi-versioned component to its past execution state.

7 Conclusions and Future Work

Software deployment is a central activity in the software development lifecycle. The deployment process changes the architecture of the target systems, as well as the systems' behavior. In order to ensure the desired effects of deployment, these changes need to be analyzed and controlled. The recent emergence of inexpensive, light-weight, resource constrained, highly distributed and mobile platforms additionally demands highly efficient software deployment solutions.

This paper has presented an approach that addresses these issues. Our approach directly leverages a system's architecture in enabling deployment. We have adapted our existing architectural implementation infrastructure to natively support the four deployment activities outlined in the Introduction. Our solution is extremely light-weight and has been successfully ported to a number of desk-top and hand-held platforms. While our experience thus far has been very positive, a number of issues remain areas of future work.

The issue of trust is central to the deployment of Prism applications due to their increased distribution, heterogeneity, and (possibly wireless) communication. We believe that explicit, first-class software connectors may be used to effectively support secure deployment in the Prism setting. The connectors may be used to implement various security protocols, including authentication, authorization, encryption, certificates, and sessions [13]. To date, we have implemented an encryption module inside *Border Connectors*. We plan to extend and directly apply this capability in our deployment support in the near future.

Another critical issue associated with highly distributed, mobile, possibly embedded systems is performance [7]. 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. This includes

estimation of optimal component locations in a distributed configuration, estimation of which components should be deployed, and, finally, when the deployment should occur. We intend to integrate Prism's support for architectural self-awareness and run-time monitoring with existing tools for system resource analysis [4] in order to enable these estimations.

References

1. A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. Technical Report, Dept. of Computer Science, University of Colorado, 1998.
2. J. E. Cook and J. A. Dage. Highly Reliable Upgrading of Components. *21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.
3. F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, June 1976.
4. P. H. Feiler and J. J. Walker. Adaptive Feedback Scheduling of Incremental and Design-To-Time Tasks. 23rd *International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001.
5. R. S. Hall, D. M. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.
6. J. Kramer and J. Magee. Constructing Distributed Systems In Conic. *IEEE TSE*, Vol. 15, No. 6, 6 1989.
7. E. A. Lee. Embedded Software. Technical Memorandum UCB/ERL M001/26, UC Berkeley, CA, July 2001
8. T. Lindholm and F. Yellin. The Java Virtual Machine Specification. *Addison Wesley* 1999.
9. N. Medvidovic, et al. Reuse of Off-the-Shelf Components in C2-Style Architectures. *19th International Conference on Software Engineering (ICSE'97),* Boston, MA, May 1997.
10. N. Medvidovic and M. Mikic-Rakic. Architectural Support for Programming-in-the-Many. Technical Report, USC-CSE-2001-506, University of Southern California, October 2001.
11. N. Medvidovic, et al. A Language and Environment for Architecture-Based Software Development and Evolution. *21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.
12. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, January 2000.
13. N. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. *22nd International Conference on Software Engineering (ICSE 2000),* Limerick, June 2000.
14. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. *20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.
15. D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.
16. M. Rakic and N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *Symposium on Software Reusability*, Toronto, Canada, May 2001.
17. B. Shannon, et al. Java 2 Platform, Enterprise Edition: Platform and Component Specifications. *Addison Wesley* 2000.

18. M. Shaw, and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.
19. R.N. Taylor, N. Medvidovic, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.
20. Sun Microsystems. K Virtual Machine (KVM). http://java.sun.com/products/kvm.