A Framework for Ensuring and Improving Dependability in Highly Distributed Systems

Sam Malek, Nels Beckman, Marija Mikic-Rakic, and Nenad Medvidovic

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

Abstract. A distributed software system's deployment architecture can have a significant impact on the system's dependability. Dependability is a function of various system parameters, such as network bandwidth, frequencies of software component interactions, power usage, and so on. Recent studies have shown that the quality of deployment architectures can be improved significantly via active system monitoring, efficient estimation of the improved deployment architecture, and system redeployment. However, the lack of the appropriate tools for monitoring, analyzing, and effecting redeployment at the architectural level makes improving a system's deployment architecture a very challenging problem. To cope with these challenges, developers typically resort to ad hoc solutions that decrease the potential for reuse and understandability. In this paper, we first present an extensible framework that guides the design and development of solutions for this type of problem, enables the extension and reuse of the solutions, and facilitates autonomic analysis and redeployment of a system's deployment architecture. We then discuss a suite of extensible and integrated tools that help developers in realizing the framework.

1 Introduction

Consider the following scenario, representative of a large number of modern distributed software applications. The scenario addresses distributed deployment of personnel in cases of natural disasters, search-and-rescue efforts, and military crises. A computer at "Headquarters" gathers information from the field and displays the current status: the locations and status of the personnel, vehicles, and obstacles. The headquarters computer is networked to a set of PDAs used by "Commanders" in the field. The commander PDAs are connected directly to each other and to a large number of "troop" PDAs. These devices communicate and help to coordinate the actions of their distributed users. Such an application is frequently challenged by network disconnections during system execution. Even when the hosts are connected, the bandwidth fluctuations and the unreliability of network links affect the system's properties such as availability and latency.

For any such large, distributed system many deployment architectures (i.e., distributions of the system's software components onto its hardware hosts) will be typically possible. Some of those deployment architectures will be more dependable than others.

For example, a distributed system's availability can be improved if the system is deployed such that the most critical, frequent, and voluminous interactions occur either locally or over reliable and capacious network links.

Finding a deployment architecture that exhibits desirable system characteristics (e.g., low latency, high availability) or satisfies a given set of constraints (e.g., the processing requirements of components deployed onto a host do not exceed that host's CPU capacity) is a challenging problem: (1) many system parameters (e.g. network bandwidth, reliability, frequencies of component interactions, etc.) influence the selection of an appropriate deployment architecture; (2) these parameters are typically not known at system design time and/or may fluctuate at run time; (3) the space of possible deployment architectures is extremely large, thus finding the optimal deployment is rarely feasible [12]; and (4) different desired system characteristics may be conflicting (e.g., a deployment architecture that satisfies a given set of constraints and results in specific availability may at the same time exhibit high latency).

The above problem is further complicated in the context of the emerging class of decentralized systems, which are characterized by limited system-wide knowledge and the absence of a single point of control. In decentralized systems, selection of a globally appropriate deployment architecture has to be made using incomplete, locally-maintained information.

The work described in this paper builds on our previous work [10,12,13,14], where we have identified and addressed a subset of the above challenges in the context of disconnected operation. We discuss a framework that provides high-level guidelines for devising solutions addressing the challenges identified above. The framework's objective is to provide a library of reusable, pluggable, and customizable components that can be leveraged in addressing a variety of distributed system deployment scenarios. We then describe a suite of integrated tools that help us realize the framework. The tools are extensible along several dimensions and allow for: (1) inclusion of arbitrary system parameters (hardware host properties, network link properties, software component properties, software interaction properties); (2) inclusion of appropriate monitors to extract these parameters from a running system; (3) specification of desirable system characteristics (e.g., high availability, low latency, desired level of security); (4) pluggability of different algorithms targeted at improving the desired characteristics; (5) multiple visualizations of the running system and its properties; and (6) flexible support for both centralized and decentralized systems. Finally, we demonstrate our approach on both a centralized and a decentralized example scenario.

The remainder of the paper is organized as follows. Section 2 briefly outlines the related work. Section 3 presents the deployment improvement framework. Section 4 briefly describes our supporting tools and discusses the specific characteristics of the tools that make them suitable for realizing the framework. Finally, Section 5 demonstrates our approach on two example scenarios. The paper concludes with an outline of our future work.

2 Related Work

One of the techniques for improving a system's dependability is (re)deployment, which is a process of installing, updating, and/or relocating a distributed software system. Carzaniga et. al. [2] provide an extensive comparison of existing software deployment approaches. They identify several issues lacking in the existing deployment tools, including integrated support for the entire deployment life cycle. An exception is Software Dock [5], which is a system of loosely coupled, cooperating, distributed components. Software Dock provides software deployment agents that travel among hosts to perform software deployment tasks. Unlike our approach, however, Software Dock does not focus on extracting system parameters, visualizing, or evaluating a system's deployment architecture, but rather on the practical concerns of effecting a deployment.

The problem of improving a system's deployment architecture has been studied by several researchers:

- I5 [1], proposes the use of the binary integer programming model (BIP) for generating an optimal deployment of a software application over a given network, such that the overall remote communication is minimized. Solving the BIP model is exponentially complex in the number of software components, rendering I5 applicable only to systems with very small numbers of software components and target hosts. Furthermore, the approach is only applicable to the minimization of remote communication.
- Coign [7] provides a framework for distributed partitioning of COM applications across the network. Coign monitors inter-component communication and then selects a distribution of the application that will minimize communication time, using the lift-to-front minimum-cut graph cutting algorithm. However, Coign can only handle situations with two machine, client-server applications. Its authors recognize that the problem of distributing an application across three or more machines is NP hard and do not provide approximative solutions for such cases.
- Kichkaylo et al. [9], provide a model, called component placement problem (CPP), for describing a distributed system in terms of network and application properties and constraints, and an AI planning algorithm, called Sekitei, for solving the CPP model. The focus of CPP is to capture a number of different constraints that restrict the solution space of valid deployment architectures. At the same time, CPP does not provide facilities for specifying the goal, i.e., a criterion function that should be maximized or minimized. Therefore, Sekitei only searches for a valid deployment that satisfies the specified constraints, without considering the quality of the found deployment.
- In our own prior work [10,12,14], we devised a set of algorithms for improving a software system's availability by finding an improved deployment architecture. The novelty of our approach was a set of approximative algorithms that scaled well to large distributed software systems with many components and hosts. However, our approach was limited to a predetermined set of system parameters, and a predetermined definition of availability, and was not extensible to problems with different concerns. Furthermore, it did not consider decentralized systems.

While all of the above projects propose novel solutions for improving a system's properties through the redeployment of software components, the implementation and evaluation of these solutions is done in an ad-hoc way, making it hard to adopt and reuse their results. Furthermore, most of these approaches are aimed at improving specific system properties, which may restrict their applicability.

Also related to our work is the research on architecture based adaptation frameworks, examples of which are [4,16]. As opposed to general purpose architecture-based adaptation frameworks, we are only considering a specific kind of adaptation (i.e., redeployment of components). Therefore, we are able to create a more detailed, and hopefully more practical framework that guides the developers in the design of their solutions.

Finally, Haas et. al. [6] provide a scalable framework for autonomic service deployment in networks. This approach does not address the exponential complexity in the selection of the most appropriate deployment, or that properties of services and hosts may change during the execution.

3 Approach

We have developed a methodology for improving a distributed system's availability via (1) active system monitoring, (2) estimation of the improved deployment architecture, and (3) redeployment of (parts of) the system to effect the improved deployment architecture. Based on this three-step methodology we developed a high-level deployment improvement framework. In this section we describe the framework's components, the associated functionality of each component, and the dependency relationships that guide their interaction. We also describe the framework's instantiation for two classes of solutions.

3.1 Framework Model

Figure 1 shows the framework's overall structure and the relationships among its six high-level components. Note that each of the framework's components can have an internal architecture that is composed of one or more lower-level components. Furthermore, the internal architecture of each component can be distributed (i.e., different internal low-level components may communicate across address spaces). The arrows represent the flow of data among the framework components.

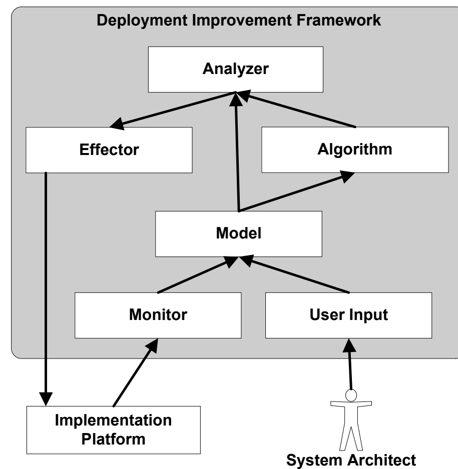

Figure 1. Deployment improvement framework.

Model. This component maintains the representation of the system's

deployment architecture. The model is composed of four types of parts: hosts, components, physical links between hosts, and logical links between components. Each of these types could be associated with an arbitrary set of parameters. For example, each host can be characterized by the amount of available memory, processing speed, battery power (in case a mobile device is used), installed software, and so on. The selection of a set of parameters to be modelled depends on the set of criteria (i.e., objectives) that a system's deployment architecture should satisfy. For example, if minimizing latency is one of the objectives, the model should include parameters such as physical network link delays and bandwidth. However, if the objective is to improve a distributed system's security, other parameters, such as security of each network link, need to be modelled.

Algorithm. Each objective is formally specified and can either be an optimization problem (e.g., maximize availability, minimize latency) or constraint satisfaction problem (e.g., total memory of components deployed onto a host cannot exceed that host's available memory). Given an objective and the relevant subset of the system's model, an algorithm searches for a deployment architecture that satisfies the objective. An algorithm may also search for a deployment architecture that simultaneously satisfies multiple objectives (e.g., maximize availability while satisfying the memory constraints).

In terms of precision and computational complexity, there are two categories of algorithms for an optimization problem like this: exact and approximative. Exact algorithms produce optimal results (e.g., deployments with minimal overall latency), but are exponentially complex, which limits their applicability to systems with very small numbers of components and hosts. On the other hand, approximative algorithms in general produce sub-optimal solutions, but have polynomial time complexity, which makes them more usable.

In terms of centralization, there are also two classes of algorithms: centralized, which are executed in a single physical location, or decentralized, which are executed on multiple, synchronized hosts. In Section 5, we describe examples of both centralized and decentralized algorithms in more detail.

Analyzer. Analyzers are meta-level algorithms that leverage the results obtained from the algorithm(s) and the model to determine a course of action for satisfying the system's overall objective. In situations where several objective functions need to be satisfied, an analyzer resolves the results from the corresponding algorithms to determine the best deployment architecture. However, note that an analyzer cannot always guarantee satisfaction of all the objectives. Analyzers are also capable of modifying the framework's behavior by adding or removing low-level components from the framework's high-level components. For example, once an analyzer determines that the system's parameters have changed significantly, it may choose to add a new low-level algorithm component that computes better results for the new operational scenario. Analyzers may also hold the history of the system's execution by logging fluctuations of the desired objectives and the parameters of interest. System's execution profile allows the

analyzer to fine-tune the framework's behavior by providing information such as system's stability, work load patterns, and the results of previous redeployments.

Monitor. To determine the run time values of the parameters in the model, a monitor is associated with each parameter. The monitor is implemented in two parts: a platform-dependent part that "hooks" into the implementation platform and performs the actual monitoring of the system, and a platform-independent part that interprets and may look for patterns in the monitored data. For example, it determines if the data is stable enough [14] to be passed on to the model. We will discuss an example of this in Section 5.

Effector. Just like monitors, effectors are also composed of two parts: (1) a platform-dependent part that "hooks" into the platform to perform the redeployment of software components; and (2) a platform-independent part that receives the redeployment instructions from the analyzer and coordinates the redeployment process. Depending on the implementation platform's support for redeployment, effectors may also need to perform tasks such as buffering, hoarding, or relaying of the exchanged events during component redeployment.

User Input. Some system parameters may not be easily monitored (e.g., security of a network link). Also, some parameters may be stable throughout the system's execution (e.g., CPU speed on a given host). The values for such parameters are provided by the system's architect at design time. We are assuming that the architect is able to provide a reasonable bound on the values of system parameters that cannot easily be monitored. Furthermore, the architect also must be capable of providing constraints on the allowable deployment architectures. Examples of these types of constraints are location and collocation constraints. Location constraints specify a subset of hosts on which a given component may be legally deployed. Collocation constraints specify a subset of components that either must be or may not be deployed on the same host.

3.2 Framework Instantiation

Figure 2 shows our framework's instantiation for a centralized system. Centralized systems have a *Master Host* (i.e., central host) that has complete knowledge of the distributed system parameters. *Master Host* contains a *Centralized Model*, which maintains the global model of the distributed system. The *Centralized Model* is populated by the data it receives from *Master Monitor* and *Centralized User Input*. The *Master Monitor* receives all of the monitoring data from the *Slave Monitor*s on other hosts. Once all monitoring data from all *Slave Host*s is received, the *Master Monitor* forwards the monitoring data to the *Centralized Model*. Each *Slave Host* contains a *Slave Effector*, which receives redeployment instructions from the *Master Effector*, and a *Slave Monitor*, which monitors the *Slave Host*'s *Implementation Platform* and sends the monitoring data back to the *Master Monitor*. Finally, the *Master Effector* receives a sequence of command instructions from the *Centralized Analyzer* and distributes the redeployment commands to all the *Slave Effector*s.

Figure 3 shows our framework's instantiation for a decentralized system. Unlike a centralized software system, a decentralized system does not have a single host with the global knowledge of system parameters. Each host has a *Local Monitor* and a *Local Ef-*

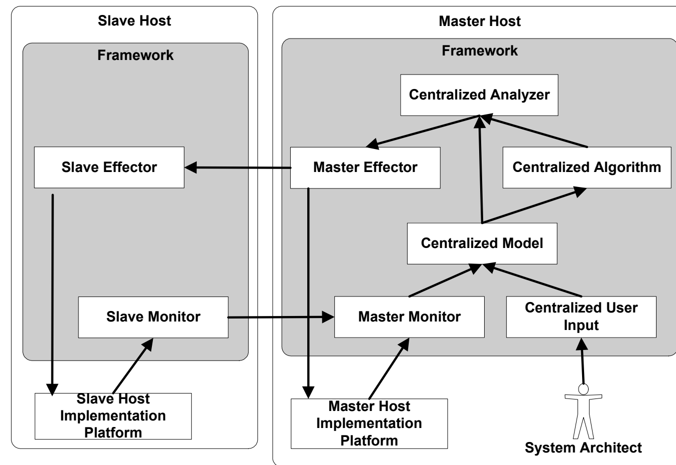

Figure 2. Framework's centralized instantiation.

fector that are only responsible for the monitoring and redeployment of the host on which they are located. Each host has a *Decentralized Model* that contains some subset of the system's overall model, populated by the data received from the *Local Monitor* and the *Decentralized Model* of the hosts to which this host is connected. Therefore, if there are two hosts in the system that are not aware of (i.e., connected to) each other, then the respective models maintained by the two hosts do not contain each other's system parameters. Each host also has a *Decentralized Algorithm* that synchronizes with its remote counterparts to find a common solution. Finally, in a similar way, the *Decentralized Analyzer* on each host synchronizes with its remote counterparts to determine an improved deployment architecture and effect it.

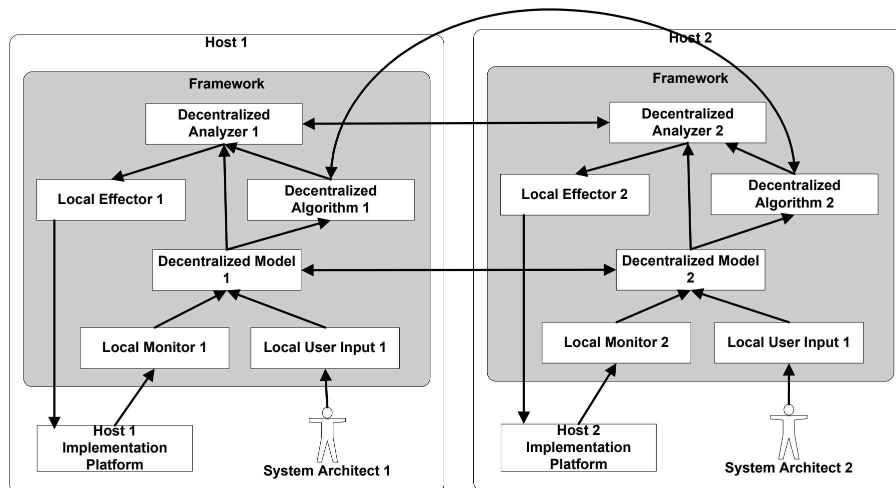

Figure 3. Framework's decentralized instantiation.

4 Tool Suite

While the framework's design is independent of any specific tool or environment, appropriate tool support facilitates the implementation, and automation, of specific deployment improvement solutions using the framework. In this section we describe two tools and their integration, which assist engineers in developing solutions that conform to the framework.

4.1 DeSi

DeSi [13] is a visual deployment exploration environment that supports specification, manipulation, and visualization of deployment architectures for large-scale, highly distributed systems. By leveraging DeSi, an architect is able to enter desired system parameters into the model, and also to manipulate those parameters and study their effects (shown in Figure 9). For example, the architect is able to use a graphical environment to specify new architectural constructs (e.g., components, hosts), parameters (e.g., network bandwidth, host memory), and values for the parameters (e.g., available memory on a host is 1MB). The architect may also specify constraints. For example, the maximum and minimum available resources, the location constraint that denotes the hosts that a component can not be deployed on, and the collocation constraint that denotes a subset of components that should not be deployed on the same host. DeSi also provides a visualization environment for graphically displaying the system's monitored data, deployment architecture, and the results of analysis (shown in Figure 10).

Figure 4 shows the high-level architecture of DeSi. The centerpiece of the architecture is a rich and extensible *Model*, which in turn allows extensions to the *View* (used for model visualization) and *Controller* (used for model manipulation) subsystems.

Model. DeSi's *Model* subsystem is reactive and accessible to the *Controller* via a simple API. The *Model* currently captures three different system aspects in its three components: *SystemData*, *GraphViewData*, and *AlgoResultData*. *SystemData* is the key part of the *Model* and represents the software system itself in terms of the architectural constructs and parameters: numbers of components and hosts, distribution of components across hosts, software and hardware topologies, and so on. *GraphViewData* captures the information needed for visualizing a system's deployment architecture: graphical (e.g., color, shape, border thickness) and layout (e.g., juxtaposition, movability, containment) properties of the depicted components, hosts, and their links. Finally, *AlgoResultData* provides a set of facilities for capturing the outcomes of the different deployment estimation algorithms: estimated deployment architectures (in terms of component-host pairs), achieved availability, algorithm's running time, estimated time to effect a redeployment, and so on.

View. DeSi's *View* subsystem exports an API for visualizing the *Model*. The current architecture of the *View* subsystem contains two components—*GraphView* and *TableView*. *GraphView* is used to depict the information provided by the *Model*'s *GraphViewData* component. *TableView* is intended to support a detailed layout of system parameters and deployment estimation algorithms captured in the *Model*'s *SystemData*

and *AlgoResultData* components. The decoupling of the *Model*'s and corresponding *View*'s components allows one to be modified independently of the other. For example, it allows us to add new visualizations of the same models, or to use the same visualizations on new, unrelated models, as long as the component interfaces remain stable.

Controller. DeSi's *Controller* subsystem comprises four components. The *Generator*, *Modifier*, and *AlgorithmContainer* manage different aspects of DeSi's *Model* and *View* subsystems, while the *MiddlewareAdapter* component provides an interface to a, possibly third-party, system implementation, deployment, and execution platform (depicted as a "black box" in Figure 4). The *Generator* component takes as its input the desired number of hardware hosts, software components, and a set of ranges for system parameters (e.g., minimum and maximum network reliability, component interaction frequency, available memory, and so on). Based on this information, *Generator* creates a specific deployment architecture that satisfies the given input and stores it in *Model* subsystem's *SystemData* component. The *Modifier* component allows fine-grain tuning of the generated deployment architecture (e.g., by altering a single network link's reliability, a single component's required memory, and so on). Finally, the *AlgorithmContainer* component invokes the selected redeployment algorithms (examples of algorithms will be presented in Section 5) and updates the *Model*'s *AlgoResultData*. In each case, the three components also inform the *View* subsystem that the *Model* has been modified; in turn, the *View* pulls the modified data from the *Model* and updates the display.

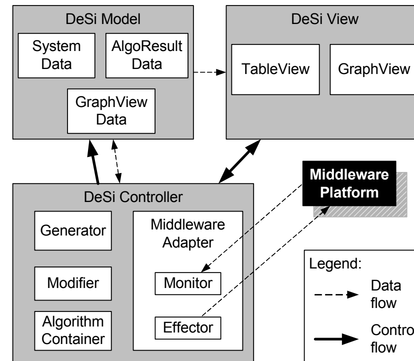

Figure 4. DeSi's architecture.

The above components allow DeSi to be used to automatically generate and manipulate large numbers of hypothetical deployment architectures. The *MiddlewareAdapter* component, on the other hand, provides DeSi with the same information from a running, real system. *MiddlewareAdapter*'s *Monitor* subcomponent captures the run-time data from the external *MiddlewarePlatform* and stores it inside the *Model*'s *SystemData* component. *MiddlewareAdapter*'s *Effector* subcomponent is informed by the *Controller*'s *AlgorithmContainer* component of the calculated (improved) deployment architecture; in turn, the *Effector* issues a set of commands to the *MiddlewarePlatform* to modify the running system's deployment architecture. The details of this process are further illuminated below.

4.2 Prism-MW

Prism-MW [11] is an extensible middleware platform that enables efficient implementation, deployment, and execution of distributed software systems in terms of their architectural elements: components, connectors, configurations, and events [17]. For brevity, Figure 5 shows the elided class design view of Prism-MW. *Brick* is an abstract

class that encapsulates common features of its subclasses (*Architecture*, *Component*, and *Connector*). The *Architecture* class records the configuration of its components and connectors, and provides facilities for their addition, removal, and reconnection, possibly at system run-time. A distributed application is implemented as a set of interacting *Architecture* objects, communicating via *DistributionConnector*s across process or machine boundaries. *Component*s in an architecture communicate by exchanging *Event*s, which are routed by *Connector*s. Finally, Prism-MW associates the *IScaffold* interface with every *Brick*. Scaffolds are used to schedule and dispatch events using a pool of threads in a decoupled manner. *IScaffold* also directly aids architectural self-awareness by allowing the run-time probing of a *Brick*'s behavior, via different implementations of the *IMonitor* interface.

To support various aspects of architectural self-awareness, we have provided the *ExtensibleComponent* class, which contains a reference to *Architecture*. This allows an instance of *ExtensibleComponent* to access all architectural elements in its local configuration, acting as a meta-level component that can automatically effect run-time changes to the system's architecture.

In support of monitoring and redeployment, the *ExtensibleComponent* is augmented with the *IAdmin* interface. We provide two implementations of the *IAdmin* interface: *Admin*, which supports system monitoring and redeployment effecting, and *Admin*'s subclass *Deployer*, which also provides facilities for interfacing with DeSi. We refer to the *ExtensibleComponent* with the *Admin* implementation of the *IAdmin* interface as *AdminComponent*; analogously, we refer to the *ExtensibleComponent* with the *Deployer* implementation of the *IAdmin* interface as *DeployerComponent*.

As indicated in Figure 5, both *AdminComponent* and *DeployerComponent* contain a reference to *Architecture* and are thus able to effect run-time changes to their local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. With the help of *DistributionConnector*s, *AdminComponent* and *DeployerComponent* are able to send and receive from any device to which they are connected the events that contain application-level components (sent between address spaces using the *Serializable* interface).

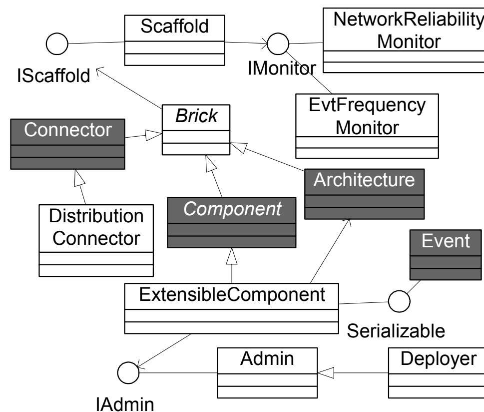

Figure 5. Elided UML class design view of Prism-MW. The four dark gray classes are used by the application developer. Only the relevant middleware classes are shown.

4.3 Tool Support for the Framework

To integrate DeSi with Prism-MW, we have wrapped *Monitor* and *Effector* components of DeSi (shown in the *Middleware Adapter* of Figure 4) as Prism-MW components that are capable of receiving *Event*s containing the monitoring data from Prism-MW's *DeployerComponent*, and issuing events to the *DeployerComponent* to enact a new deployment architecture. Once the monitoring data is received, DeSi updates its own system model. This results in the visualization of an actual system, which can now be analyzed and its deployment improved by employing different algorithms. Once the outcome of an algorithm is selected by the *Analyzer*, DeSi issues a series of events to Prism-MW's *DeployerComponent* to update the system's deployment architecture.

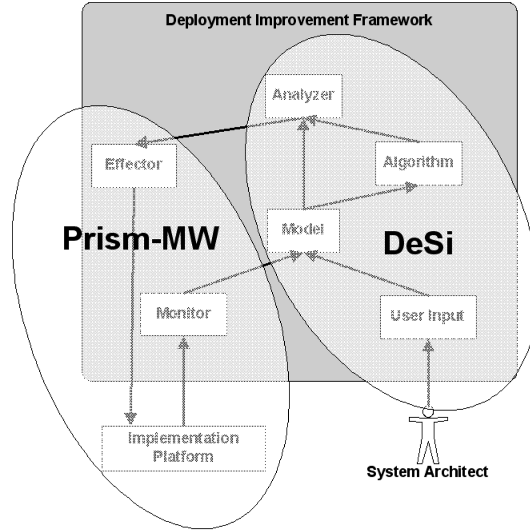

Figure 6. Realization of the framework via integration of Prism-MW and DeSi.

DeSi and Prism-MW are directly leveraged to in realizing our framework, as illustrated in Figure 6. DeSi provides the facilities for implementing *User Input*, *Model*, *Algorithm*, and *Analyzer* components, while Prism-MW supports implementation of *Monitor* and *Effector* components. In this section we discuss our realization of each one of the framework components, and their support for pluggability, extensibility, explorability, and adaptability. These characteristics allow the tool suite to be easily tailored to the variation points that arise across different problems.

Model. We leverage DeSi's extensible model to implement the *Model Component* of the framework. DeSi's extensible representation of the system's deployment architecture makes it possible to add or remove new system properties at run-time. The model and the accompanying graphical support make it easy to configure the tool to application scenarios with different concerns and objectives. Once the appropriate model is defined and specified, it is populated with the actual data from a system. The data is provided either at run-time or at design time. Some properties are known at design time (e.g., initial deployment of the system, available memory on each host, etc.), and can be captured in architectural description of the system. To this end, DeSi has been integrated with xADL 2.0 [3], an extensible architecture description language (xADL). Properties that are not available at design time (e.g., reliability of network links, available network bandwidth) are provided by the Monitor component, discussed below.

Algorithm. DeSi provides a pluggable environment for addition and removal of algorithms that run on the model. In order to effectively support reusability and extensibility within different redeployment algorithms, we have identified the following three variation points:

- The objective function (e.g., maximizing availability, increasing security, etc.) that is specified based on the system parameters defined in the model.
- The constraints on the parameters, reflecting the limited resources in the system (e.g., available bandwidth, available memory, etc.), which need to be satisfied by the algorithms when searching for a valid solution.
- The coordination that occurs in decentralized algorithms. There are many decentralized cooperative protocols (e.g., distributed voting [8], auction-based [18]).

We have used these variation points in developing extensible and reusable algorithms in DeSi (as shown in Figure 7). Each algorithm provides an implementation of an abstract API, which is used by DeSi for interfacing with the algorithm. The algorithms are composed of a main body that denotes the algorithm's approach (e.g., greedy algorithm, genetic algorithm, etc.), an objective function, and the relevant constraint functions. Decentralized algorithms are also associated with an implementation of a coordination approach. The above methodology for developing algorithms simplifies the adoption of existing solutions to new problems. The developer first creates the model of the system (as discussed earlier) using the graphical interface, specifies the objective function based on the system parameters, and finally associates the appropriate implementations of the constraint and coordination functions with the algorithm.

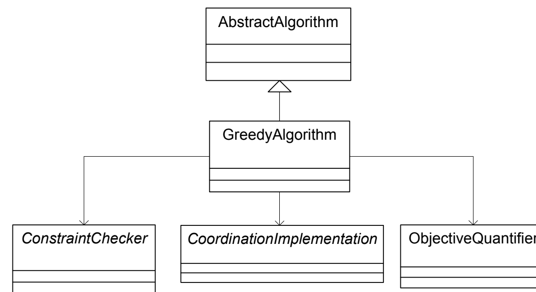

Figure 7. Class diagram of algorithm development methodology in DeSi for a greedy algorithm.

Analyzer. DeSi's visualisation of the deployment architecture and the exploratory utilities allow an engineer to rapidly investigate the space of possible deployments for a given system (real or postulated), and determine the deployments that will result in greatest improvements (while, perhaps, requiring the smallest changes to the current deployment architecture). A user can easily assess a system's sensitivity to changes in specific parameters (e.g., the reliability of a network link) and create deployment constraints (e.g., two components must be located on different hosts). However, while the analysis by a human user may be possible in small centralized systems with few objectives, it is certainly infeasible for large and/or decentralized systems with multiple (and potentially conflicting) objectives. Furthermore, given a deployment improvement problem, there are many decisions and trade-offs associated with improving the deployment architecture: scheduling the time to (re)examine the deployment architecture, selecting the algorithm(s) to run, comparing the results, resolving conflicts, determining the best result, and scheduling the time to effect the solution. Therefore, autonomic so-

lutions for the analysis and conflict resolution are needed. DeSi supports these kind of meta-level algorithms via an API for the modification of DeSi's internal architecture. The API allows for addition and removal of algorithms, modification of the model, and access to DeSi's internal data structure that holds the results of executing algorithms. Via this API, a meta-level algorithm is capable of keeping a profile of the system's history (by monitoring the system's performance), determining the best configuration for the tool, and selecting the result of the best algorithm. Furthermore, in complicated decentralized scenarios, the meta-level algorithms may leverage a decentralized negotiation technique to coordinate their actions with other remote analyzers. Some examples of different analyzers are discussed in Section 5.

Monitor. Prism-MW provides the *IMonitor* interface associated through the *Scaffold* class with every *Brick*. This allows for autonomous, active monitoring of a *Brick*'s runtime behavior. For example, the *EvtFrequencyMonitor* records the frequencies of different events the associated *Brick* sends, while *NetworkReliabilityMonitor* records the reliability of connectivity between its associated *DistributionConnector* and other, remote *DistributionConnector*s using a common "pinging" technique. A meta-level *AdminComponent* (recall Section 4.2) on any device is capable of accessing the monitoring data of its local components via its reference to *Architecture*. In order to minimize the time required to monitor the system, monitoring is performed in short intervals of adjustable duration. Once the monitored data is stable (i.e., the difference in the data across a desired number consecutive intervals is less than an adjustable value ε), the *AdminComponent* sends the description of its local deployment architecture and the monitored data (e.g., event frequency, network reliability, etc.) in the form of serialized Prism-MW *Event*s to the *DeployerComponent*. Figure 8 depicts an application running on top of Prism-MW with the monitoring and deployment facilities instantiated and associated with the appropriate architectural constructs. Our assessment of Prism-MW's monitoring support suggests that monitoring on each host may induce as little as 0.1% and no greater than 10% in memory and efficiency overheads. Note that Prism-MW's extensible design allows for addition of new monitoring capabilities via new implementations of *IMonitor* interface.

Effector. Once a new deployment architecture is selected by one of DeSi's algorithms based on the monitoring data supplied by Prism-MW, DeSi informs the *DeployerComponent* (recall Section 4.2) of the desired deployment architecture, which now needs to be effected. The effecting process requires coordination among different hosts (e.g., ensuring architectural consistency, synchronization, etc.), which is an implementation platform-independent task. Prism-MW's support for coordination is implemented in its *Admin* and *Deployer Component*s:

- The *DeployerComponent* sends events to inform *AdminComponent*s of their new local configurations, and of the remote locations of software components required for performing changes to each local configuration.
- Each *AdminComponent* determines the difference between its current and new configurations, and issues a series of events to remote *AdminComponent*s requesting the components that are to be deployed locally. If devices that need to

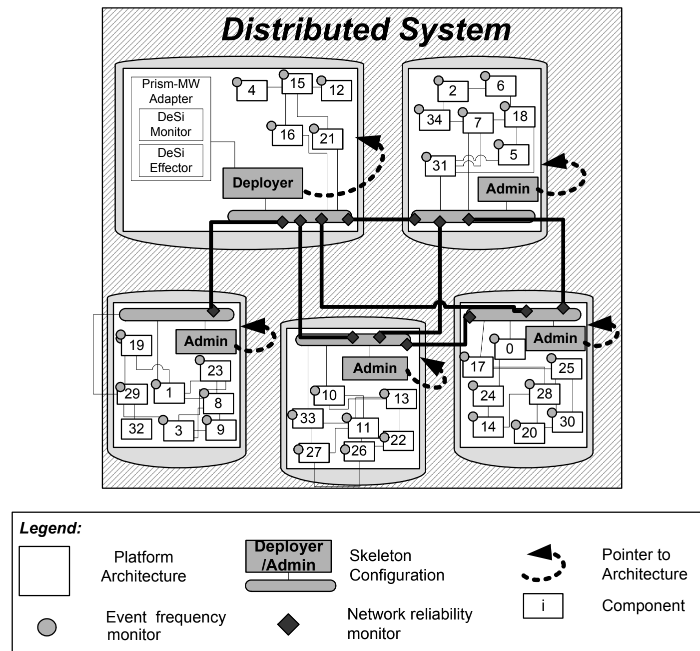

Figure 8. An example of a distributed system running on top of Prism-MW that is monitored.

exchange components are not directly connected, the relevant request events are sent to the *DeployerComponent*, which then mediates their interaction.

- Each *AdminComponent* that receives an event requesting its local component(s) to be deployed remotely, detaches the required component(s) from its local configuration, serializes them, and sends them as a series of events via its local *DistributionConnector* to the requesting device.
- The recipient *AdminComponents* reconstitute the migrant components from the received events and invoke the appropriate methods on its *Architecture* object to attach the received components to the local configuration.

Other coordination techniques can also be incorporated into Prism-MW in a similar manner via different implementations of the *DeployerComponent* and *AdminComponent*.

User Input and Visualization. Once the monitoring data is gathered from all the hosts, the user may invoke one of DeSi's visualization windows to explore the system's deployment architecture and its relevant parameters. Figure 9 shows the table-oriented page of the DeSi editor. This page is divided into five sections. In the *Parameters* table, the properties of every host, component, or link within a software system can be viewed and modified, e.g., to assess the sensitivity of a deployment architecture to specific parameter changes. In the *Constraints* panel, the user can specify different constraints on component locations (e.g., fixing a component to a selected host). Using the set of but-

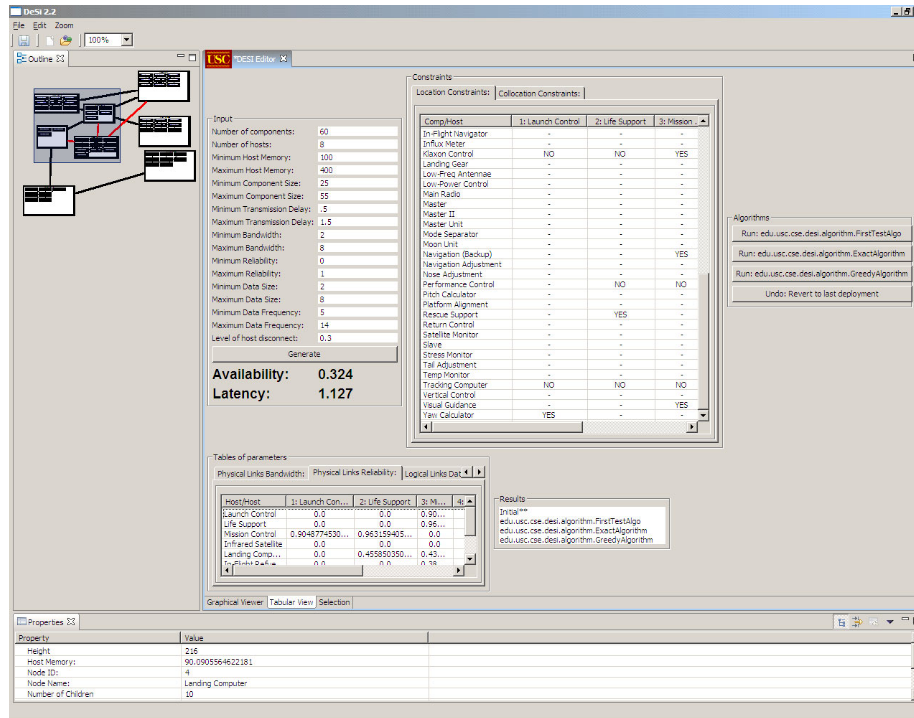

Figure 9. DeSi's editable tabular view of the system's deployment architecture.

tons in the *Algorithms* panel, different algorithms can be invoked and the results displayed in the *Results* panel. Figures 10a and b show the graph-oriented page of the DeSi editor. The thumbnail view in the upper left displays the entire architecture at once and allows users to quickly navigate to any of its portions. Since our framework can support large distributed systems with many hosts and components, DeSi supports the ability to zoom in and out on a visualized system. Hosts are depicted as white boxes while software components are depicted as shaded boxes. The solid black lines between hosts represent physical (network) links and the thin black lines between components represent logical (software) links. At the bottom of the screen, the property sheet allows users to view or modify the properties of the link, host, or component that is currently selected. Components can also be "dragged-and-dropped" from one host to another. In this way, a user can manually create a new system deployment and analyze its effect on system properties (e.g., availability, latency, etc.).

5 Example Scenarios

In this section, we describe our experience with the implementation of both the centralized and the decentralized instantiation of the framework targeted at (1) maximizing a distributed system's overall availability, and (2) minimizing the system's overall latency.

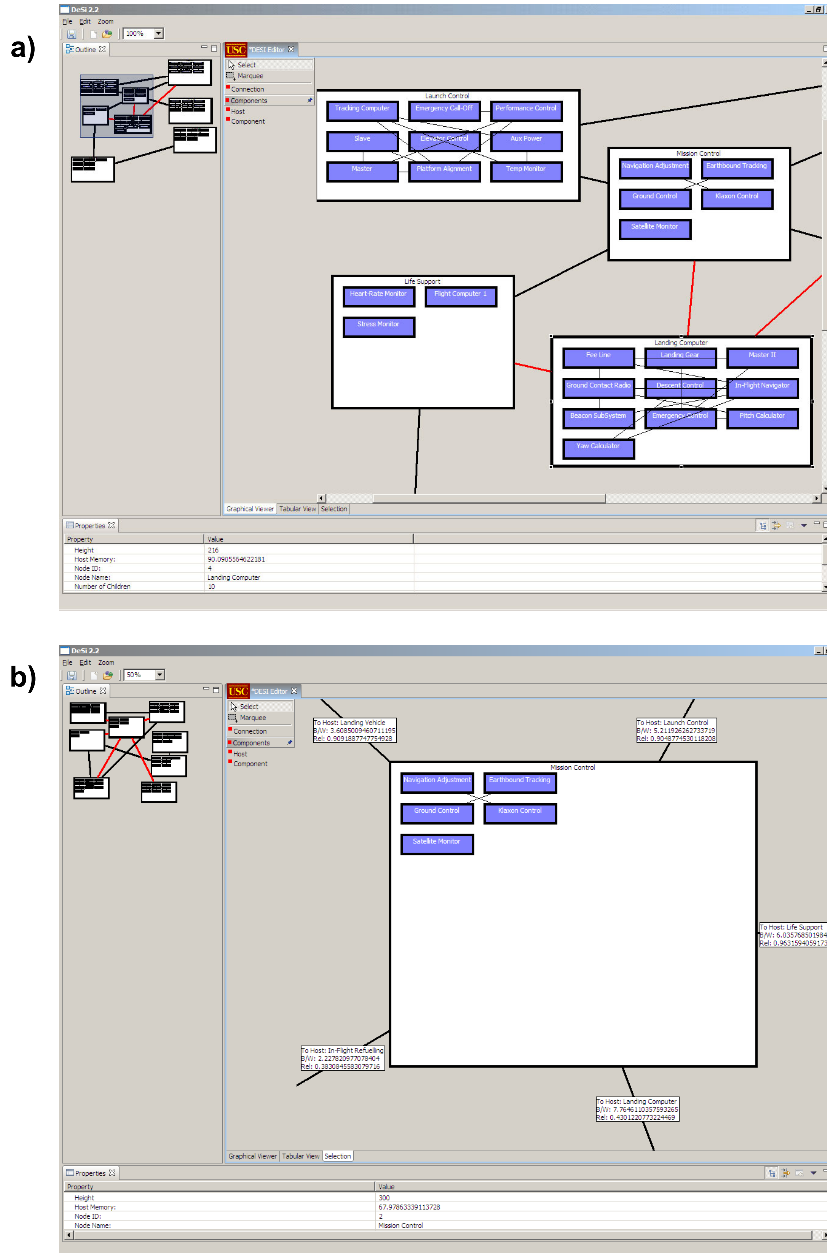

Figure 10. DeSi's graphical view of a system's deployment architecture: (a) zoomed out view showing multiple hosts; (b) zoomed in view of the same architecture.

5.1 Centralized Configuration

In order to achieve the objective of maximizing a system's availability and minimizing the latency we first created an appropriate model. The model is composed of a hierarchical structure of components and hosts that includes the following properties:

- Each component has a required memory size.
- Each host has an available memory.
- Each logical link between components is modelled with a frequency of interaction and an average event size.
- Each physical link among hosts is modelled with a particular network reliability, bandwidth, and transmission delay.
- The system's model contains the location and collocation constraints, discussed in Section 3.1, that restrict the space of valid deployments.

The values for the host's available memory, component's required size, location and collocation constraints are all entered into the model by the user via the DeSi tool. All the modelled properties that are not entered by the user are monitored at run time and added to the model automatically.

We have used three centralized algorithms, called Exact, Stochastic, and Avala [12]. The objective of all these algorithms is to maximize the system's availability by finding a deployment architecture such that the most critical, frequent, and voluminous interactions occur either locally or over reliable and capacious network links. Below we provide a high-level overview of these algorithms.

The Exact algorithm tries every possible deployment, and selects the one that results in maximum availability and satisfies the constraints posed by the memory, bandwidth, and restrictions on software component locations. The Exact algorithm guarantees at least one optimal deployment (assuming that at least one deployment is possible). The complexity of this algorithm in the general case (i.e., with no restrictions on component locations) is $O(k^n)$, where k is the number of hardware hosts, and n the number of software components. By fixing a subset of m components to selected hosts, the complexity reduces to $O(k^{n-m})$.

The Stochastic algorithm randomly orders all the hosts and all the components. Then, going in order, it assigns as many components to a given host as can fit on that host, ensuring that all of the constraints are satisfied. Once the host is full, the algorithm proceeds with the same process for the next host in the ordered list of hosts, and the remaining unassigned components in the ordered list of components, until all components have been deployed. This process is repeated a desired number of times, and the best obtained deployment is selected. Since it needs to calculate the availability and latency for every deployment, the complexity of this algorithm is $O(n^2)$.

Avala is a greedy algorithm that incrementally assigns software components to the hardware hosts. At each step of the algorithm, the goal is to select the assignment that will maximally contribute to the objective function, by selecting the "best" host and "best" software component. Selecting the best hardware host is performed by choosing a host with the highest sum of network reliabilities and bandwidths with other hosts in

the system, and the highest memory capacity. Similarly, selecting the best software component is performed by choosing the component with the highest frequency of interaction with other components in the system, and the lowest required memory. Once found, the best component is assigned to the best host, making certain that the location and collocation constraints are satisfied. The algorithm proceeds with searching for the next best component among the remaining components, until the best host is full. Next, the algorithm selects the best host among the remaining hosts. This process repeats until every component is assigned to a host. The complexity of this algorithm is $O(n^3)$.

Our framework's analyzer component automatically decides which one of the algorithms to run based on the following factors:

- The size of the architecture — For example, the Exact algorithm finds the optimal solution, but due to its complexity it can only be used for architectures with very small numbers of hosts (on the order of 5) and components (on the order of 15). Therefore, for large architectures either of the other two algorithms is used.
- The system's availability profile — Analyzer holds a record of the fluctuations in the system's availability (caused by changes in system parameters) that is used to determine when the system should be redeployed and what algorithm should be invoked. For example, the analyzer selects a more expensive algorithm to run if the system is stable (i.e., the system's availability does not fluctuate significantly). On the other hand, if the system is unstable, the analyzer runs a less expensive algorithm that could produce faster results for the immediate improvement of the system's availability.
- The system's overall latency — The algorithms used in this scenario also typically decrease the system's overall latency [12]. However, in rare situations where this is not the case, the analyzer either disallows the results of the algorithms to take effect or modifies the solution such that it does not significantly increase the system's overall latency.

Once the analyzer selects the most appropriate deployment architecture, it creates the appropriate set of redeployment instructions and sends it to the *Master Effector*. The *Master Effector* then forwards the instructions to the appropriate *Slave Effector*s, which leverage Prism-MW's support for the redeployment of software components in the manner described earlier.

5.2 Decentralized Configuration

In the development of the decentralized solution, we were able to reuse the centralized model by extending it to include the notion of "awareness". Awareness denotes the extent of each host's knowledge about the global system parameters. The *Decentralized Model* on each hosts synchronizes its local model with the remote hosts of which it is aware of (i.e., to which it is directly connected), by sending streams of data whenever the model is modified.

Unlike the centralized solution, getting the user input and monitoring is done separately and independently on each host. Similarly to the centralized solution, we leverage DeSi and Prism-MW in gathering data about the system parameters.

We have used a decentralized algorithm, called DecAp [10], that is based on an auction-based protocol to find a solution that significantly improves the system's overall availability. In DecAp, each Decentralized Algorithm component acts as an agent and may conduct or participate in auctions. Each host's agent initiates an auction for the redeployment of its local components, assuming none of its neighboring (i.e., connected) hosts is already conducting an auction. The auction initiation is done by sending to all the neighboring hosts a message that carries information about a component to be redeployed (e.g., name, size, and so on). The agents receiving this message have a limited time to enter a bid on the component before the auction closes. The bidding agent on a given host calculates an initial bid for the auctioned component, by considering the frequency and volume of interaction between components on its host and the auctioned component. Once the auctioneer has received all the bids, it calculates the final bid based on the received information. The host with the highest bid is selected as the winner and the component is redeployed to it. The complexity of this algorithm is $O(k*n^3)$.

The functionality of the decentralized analyzer remains very similar to the centralized version, except that the analyzer uses either the voting or the polling protocol to decide on the appropriate course of action. Once a redeployment decision is made by the analyzers, the redeployment instructions are sent out to the *Local Effector*s, which collaborate in performing the redeployment by leveraging Prism-MW's support for redeployment.

6 Conclusion

A distributed software system's deployment architecture can have a significant impact on the system's dependability, and will depend on various system parameters (e.g., reliability of connectivity among hosts, security of links between hosts, and so on). Improving the deployment architecture such that it exhibits desirable system characteristics is a challenging problem. The lack of a common design framework for improving the system's deployment architecture exacerbates the complexity of this problem. Existing deployment approaches focus on providing support for installing and updating the software system but lack support for extracting, visualizing, and analyzing different parameters that influence the quality of deployment.

In this paper we have presented a design framework for analyzing and improving distributed deployment architectures. We also discussed the integration of Prism-MW, a lightweight architectural middleware that supports system monitoring and run-time reconfiguration, and DeSi, an environment that supports manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. In concert, Prism-MW and DeSi provide a rich capability for developing solutions that comply to the framework's rules and structure. Our experience has indicated that by leveraging the tool suite to develop solutions we are able to increase the potential for creating pluggable, extensible, and reusable components that could be used to improve deployment architectures in many different scenarios. In our future work we will focus on improving system characteristics beyond availability and latency, such as security, durability, and throughput. We also plan to devise mitigating techniques for situations where different desired system characteristics may be conflicting. There are also many

unresolved issues in the decentralized setting that we plan to focus on in the future. For example, modelling user preferences for multiple desired system characteristics in a decentralized environment, and devising decentralized algorithms for non-collaborative hosts are challenging problems. For this we will leverage utility computing techniques to determine a deployment architecture that maximizes the users' overall satisfaction with a distributed system. Furthermore, in the future we plan to extend our framework and tool suite to enhance redeployment with other strategies (e.g., caching and hoarding of data, queuing of remote calls, etc.). These tasks will provide a basis for further assessment and evaluation of our framework and the tool suite.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Numbers CCR-9985441 and ITR-0312780. Effort also partially supported by the Jet Propulsion Laboratory.

8 Reference

[1] M. C. Bastarrica, et al. A Binary Integer Programming Model for Optimal Object Distribution. *2nd Int'l. Conf. on Principles of Distributed Systems*, Amiens, France, Dec. 1998.

[2] A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. *Technical Report, Dept. of Computer Science, University of Colorado,* 1998.

[3] E. Dashofy, A. van der Hoek, and R. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *International Conference on Software Engineering (ICSE'04)*, Orlando, Florida, May 2002.

[4] D. Garlan, S. Cheng, B. Schmerl. Increasing System Dependability through Architecture-based Self-repair. In R. de Lemos, C. Gacek, A. Romanovsky, eds., *Architecting Dependable Systems*, 2003.

[5] R. S. Hall, D. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *International Conference in Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.

[6] R. Haas et. al. Autonomic Service Deployment in Networks. *IBM Systems Journal*, Vol. 42, No. 1, 2003.

[7] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. *3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, Feb. 1999.

[8] R. Kieckhafer, C. Walter, A. Finn, P. Thambidurai. The MAFT Architecture for Distributed Fault Tolerance. *IEEE Transactions On Computers*, Vol. 37, No. 4, April 1988, pp. 398-405.

[9] T. Kichkaylo et al. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. *Int'l. Parallel and Distributed Processing Symposium*, April 2003.

[10] S. Malek et. al. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. *Technical Report USC-CSE-2004-506*, 2004.

[11] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Rio de Janeiro, Brazil, June 2003.

[12] M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. *Technical Report USC-CSE-2003-515*, 2003.

[13] M. Mikic-Rakic et. al. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. *2nd Int'l Working Conf. on Component Deployment (CD 2004)*, Edinburgh, Scotland, May 2004.

[14] M. Mikic-Rakic and N. Medvidovic. Support for Disconnected Operation via Architectural Self-Reconfiguration. *Int'l Conf. on Autonomic Computing (ICAC'04)*, New York, May 2004.

[15] M. Mikic-Rakic and N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *2001 Symposium on Software Reusability (SSR 2001)*, Toronto, Canada, May 2001.

[16] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture Based run time Software Evolution. *International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[17] D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[18] C. A. Waldpurger, et. al. Spawn. A Distributed Computational Economy. *IEEE Trans. on Software Engineering*, February 1992.