A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings

Marija Mikic-Rakic, Sam Malek, Nels Beckman, and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{marija,malek,nbeckman,neno}@usc.edu

Abstract. A distributed software system's deployment architecture can have a significant impact on the system's properties. These properties will depend on various system parameters, such as network bandwidth, frequencies of software component interactions, and so on. Existing tools for representing system deployment lack support for specifying, visualizing, and analyzing different factors that influence the quality of a deployment, e.g., the deployment's impact on the system's availability. In this paper, we present an environment that supports flexible and tailorable specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. The environment has been successfully used to explore large numbers of postulated deployment architectures. It has also been integrated with a middleware platform to support the exploration of deployment architectures of actual distributed systems.

Keywords. Software deployment, availability, disconnection, visualization, environment, middleware

1 Introduction

For any large, distributed system, multiple deployment architectures (i.e., distributions of the system's software components onto its hardware hosts, see Figure 1) will be typically possible. Some of those deployment architectures will be more effective than others in terms of the desired system characteristics such as scalability, evolvability, mobility, and dependability. Availability is an aspect of dependability, defined as the degree to which the system is operational and accessible when required for use [5]. In the context of distributed environments, where a most common cause of (partial) system inaccessibility is network failure [17], we define availability as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions over a period of time. In other words, availability in distributed systems is greatly affected by the properties of the network, including its reliability and bandwidth.

Maximizing the availability of a given system may thus require the system to be redeployed such that the most critical, frequent, and voluminous interactions occur either locally or over reliable and capacious network links. However, finding the actual deployment architecture that maximizes a system's availability is an exponentially complex problem that may take *years* to resolve for any but very small systems [10]. Also, even a deployment architecture that increases the system's current availability by a desired amount cannot be easily found because of the many parameters that influence this task: number of hardware hosts, available memory and CPU power on each host, network topology, capacity and reliability of network links, number of software components, memory and processing requirements of each component, their configuration (i.e., software topology), frequency and volume of interaction among the components, and so forth. A naive solution to this problem would be to keep redeploying the actual system that exhibits poor availability until an adequate deployment architecture is

found. However, this would be prohibitively expensive. A much more preferable solution is to develop a means of modeling the relevant system parameters, estimating the deployment architecture based on these parameters in a manner that produces the desired (increase in) availability, and assessing the estimated architecture in a controlled setting, *prior* to changing the actual deployed system.

In this paper, we discuss a tailorable environment developed precisely for that purpose. The environment, called DeSi, supports specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. DeSi allows an engineer to rapidly explore the space of possible deployments for a given system (real or postulated), determine the deployments that will result in greatest improvements in availability (while, perhaps, requiring the smallest changes to the current deployment architecture), and assess a system's sensitivity to and visualize changes in specific parameters (e.g., the reliability of a particular network link) and deployment constraints (e.g., two components must be located on different hosts). We have provided a facility that automatically generates large numbers of deployment scenarios and have evaluated different aspects of DeSi using this facility. DeSi also allows one to easily integrate, evaluate, and compare different algorithms targeted at improving system availability [10] in terms of their feasibility, efficiency, and precision. We illustrate this support by showing the integration of six such algorithms. DeSi also provides a simple API that allows its integration with any distributed system platform (i.e., middleware) that supports component deployment at runtime. We demonstrate this support by integrating DeSi with the Prism-MW middleware [9]. Finally, while availability has been our focus to date, DeSi's architecture is flexible enough to allow exploration of other system characteristics (e.g., security, fault-tolerance, and so on).

The remainder of the paper is organized as follows. Section2 defines the problem of increasing the availability of distributed systems, and overviews six different algorithms we have developed for this purpose. Section3 highlights the related work. Section4 discusses the architecture, implementation, and usage of the DeSi environment. Evaluation of DeSi is presented in Section5. The paper concludes with a discussion of future work.

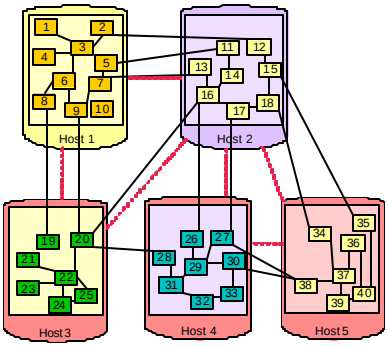


Figure 1. Example deployment architecture. A software system comprising 40 components is deployed onto five hosts. The dotted lines represent host interconnectivity; the filled lines represent software component interaction paths.

2 Background

2.1 Problem Description

The distribution of software components onto hardware nodes (i.e., a system's software *deployment architecture*, illustrated in Figure 1) greatly influences the system's availability in the face of connectivity losses. For example, components located on the same host will be able to communicate regardless of the network's status; components distributed across different hosts might not. However, the reliability (i.e., rate of failure) of connectivity among the hardware nodes on which

Model

Given:

(1) a set C of n components ($n = |C|$) and three functions $freq: C \times C \to \ldots$, $data\_vol: C \times C \to \ldots$, and $mem_{comp}: C \to \ldots$

$$freq(c_i, c_j) = \begin{cases} 0 & \text{if } c_i = c_j \\ \text{frequency of communication between } c_i \text{ and } c_j & \text{if } c_i \neq c_j \end{cases}$$

$$evt\_size(c_i, c_j) = \begin{cases} 0 & \text{if } c_i = c_j \\ \text{avg size of data } c_i \text{ and } c_j \text{ exchange} & \text{if } c_i \neq c_j \end{cases}$$

$mem_{comp}(c)$ = *required memory for c*

(2) a set H of k hardware nodes ($k = |H|$) and three functions $rel: H \times H \to \ldots$, $bw: H \times H \to \ldots$, and $mem_{host}: H \to \ldots$

$$rel(h_i, h_j) = \begin{cases} 1 & \text{if } h_i = h_j \\ 0 & \text{if } h_i \text{ is not connected to } h_j \\ \text{reliability of the link between } h_i \text{ and } h_j & \text{if } h_i \neq h_j \end{cases}$$

$$bw(h_i, h_j) = \begin{cases} \infty & \text{if } h_i = h_j \\ 0 & \text{if } h_i \text{ is not connected to } h_j \\ \text{bandwidth of the link between } h_i \text{ and } h_j & \text{if } h_i \neq h_j \end{cases}$$

$mem_{host}(h)$ = *available memory on host h*

(3) Two functions that restrict locations of software components $loc: C \times H \to \{0,1\}$ $colloc: C \times C \to \{-1,0,1\}$

$$loc(c_i, h_j) = \begin{cases} 1 & \text{if } c_i \text{ can be deployed onto } h_j \\ 0 & \text{if } c_i \text{ cannot be deployed onto } h_j \end{cases}$$

$$colloc(c_i, c_j) = \begin{cases} -1 & \text{if } c_i \text{ cannot be on the same host as } c_j \\ 1 & \text{if } c_i \text{ has to be on the same host as } c_j \\ 0 & \text{if there are no restrictions on collocation of } c_i \text{ and } c_j \end{cases}$$

Problem

Problem:

Find a function $f: C \to H$ such that the system's overall availability

A defined as $A = \dfrac{\sum_{i=1}^{n} \sum_{j=1}^{n} (freq(c_i, c_j) \cdot rel(f(c_i), f(c_j)))}{\sum_{i=1}^{n} \sum_{j=1}^{n} freq(c_i, c_j)}$ is maximized, and the following four conditions are satisfied:

(1) $\forall i \in [1,k] \; \forall j \in [1,n] \quad f(c_j) = h_i \;\Big|\; \sum_j mem_{comp}(c_j) \leq mem_{host}(h_i)$

(2) $(\forall i \in [1,k] \quad \forall j \in [i+1,k])$ $\left(\begin{array}{l} (\forall l \in [1,n] \quad \forall m \in [l+1,n]) \\ \text{where } f(c_l) = h_i \wedge f(c_m) = h_j \end{array} \right)$ $\sum_{l,m} data\_vol(c_l, c_m) \leq effective\_bw(h_i, h_j)$ where *data_vol* and *effective_bw* are defined as follows:

$$data\_vol(c_x, c_y) = freq(c_x, c_y) * evt\_size(c_x, c_y)$$

$$effective\_bw(h_x, h_y) = rel(h_x, h_y) * bw(h_x, h_y)$$

(3) $\forall j \in [1,n] \quad loc(c_j, f(c_j)) = 1$

(4) $\forall i \in [1,n] \quad \forall j \in [i+1,n] \quad$ *if* $(colloc(c_i, c_j) = 1) \Rightarrow (f(c_i) = f(c_j))$ *if* $(colloc(c_i, c_j) = -1) \Rightarrow (f(c_i) \neq f(c_j))$

In the most general case, the number of possible functions f is k^n. However, note that some of these deployments may not satisfy one or more of the above four conditions.

Figure 2. Formal statement of the problem.

the system is deployed may not be known before the deployment and may change during the system's execution. The frequencies of interaction among software components may also be unknown. For this reason, the current software deployment architecture may be ill-suited for the given state of the "target" hardware environment. This means that a *redeployment* of the software system may be necessary to improve its availability. The critical difficulty in achieving this task lies in the fact that determining a software system's deployment architecture that will maximize its availability for the given target environment (referred to as *optimal deployment architecture*) is an exponentially complex problem.

In addition to the characteristics of hardware connectivity and software interaction, there are other constraints on a system's redeployment, including the available memory on each network host, the required memory for each software component, the size of data exchanged between software components, the bandwidth of each network link, and possible restrictions on component locations (e.g., a component may be fixed to a selected host, or two components may not be allowed to reside on the same host). Figure 2 shows a formal model that captures the system properties and constraints, and a formal definition of the problem we are addressing. The mem_{comp} function captures the

required memory for each component. The frequency of interaction between any pair of components is captured via the *freq* function, and the average size of data exchanged between them is captured via the *evt_size* function. Each host's available memory is captured via the mem_{host} function. The reliability of the link between any pair of hosts is captured via the *rel* function, and the network bandwidth via the *bw* function. Using the *loc* function, deployment of any component can be restricted to a subset of hosts, thus denoting a set of allowed hosts for that component. Using the *colloc* function, constraints on collocation of components can be specified.

The definition of the problem contains the criterion function *A*, which formally describes a system's availability as the ratio of the number of successfully completed interactions in the system to the total number of attempted interactions. Function *f* represents the exponential number of the system's candidate deployments. To be considered valid, each candidate deployment must satisfy the four stated conditions. The first condition states that the sum of memories of the components deployed onto a given host may not exceed the host's available memory. The second condition states that the total volume of data exchanged across any link between two hosts may not exceed the link's *effective bandwidth*, which is the product of the link's actual bandwidth and its reliability. The third condition states that a component may only be deployed onto a host that belongs to a set of allowed hosts for that component, specified via the *loc* function. Finally, the fourth condition states that two components must be deployed onto the same host (or on different hosts) if required by the *colloc* function.

2.2 Algorithms

In this section we briefly describe six algorithms we have developed for increasing a system's availability by calculating a new deployment architecture. A detailed performance comparison of several of these algorithms is given in [10].

Exact Algorithm. This algorithm tries every possible deployment, and selects the one that has 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 $O(k^{n-m})$.

Unbiased Stochastic Algorithm. This algorithm generates different deployments by randomly assigning each component to a single host from the set of available hosts for that component. If the randomly generated deployment satisfies all the constraints, the availability of the produced deployment architecture is calculated. This process repeats a given number of times and the deployment with the best availability is selected. As indicated in Figure 2, the complexity of calculating the availability for each valid deployment is $O(n^2)$, resulting in the same complexity of the overall algorithm.

Biased Stochastic Algorithm. This 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 for every deployment, the complexity of this algorithm is $O(n^2)$.

Greedy Algorithm. This algorithm 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 availability 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 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 four 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 $O(n^3)$ [10].

Clustering Algorithm. This algorithm groups software components and physical hosts into a set of component and host *clusters*, where all members of a cluster are treated as a single entity. For example, when a component in a given cluster needs to be redeployed to a new host, all of the cluster's member components are redeployed. The algorithm clusters components with high frequencies of interaction, and hosts with high connection reliability. Clustering can significantly reduce the size of the redeployment problem; it also has the potential to increase the availability of a system. For example, connectivity-based clustering in peer-to-peer networks improves the quality of service by reducing the cost of messaging [15].

Decentralized Algorithm. The above algorithms assume the existence of a central host with reliable connections to every other host in the system. This assumption does not hold in a wide range of distributed systems (e.g., ad-hoc mobile networks), requiring a *decentralized* solution. Our decentralized redeployment algorithm leverages a variation of the auction algorithm [13], in which each hosts 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 (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. In each bid message, the bidding agent also sends additional local information, including its host's network reliability and bandwidth with neighboring hosts. 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. If the winner has enough free memory and sufficient bandwidth to host the auctioned component, then the com-

ponent is redeployed to it and the auction is closed. If this is not the case, then the winner and the auctioneer attempt to find a component on the winner host to be traded (swapped) with the auctioned component. The complexity of this algorithm is *O(k*n)*.

3 Related Work

This section briefly outlines several research areas and approaches relevant to our work on DeSi: software architectures, disconnected operation, software deployment, software visualization, and visual software environments.

Software architectures provide high-level abstractions for representing structure, behavior, and key properties of a software system [14]. They are described in terms of *components*, which describe the computations and state of a system; *connectors*, which describe the rules and mechanisms of interaction among the components; and *configurations*, which define topologies of components and connectors. DeSi leverages an architectural model of a distributed system, including its deployment information. In our approach, a component represents the smallest unit of deployment.

Disconnected operation refers to the continued functioning of a distributed system in the (temporary) absence of network connectivity. We have performed an extensive survey of existing disconnected operation approaches, and provided a framework for their classification and comparison [11]. One of the techniques for supporting disconnected operation is (re)deployment, which is a process of installing, updating, or relocating a distributed software system.

Carzaniga et. al. [1] 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 lifecycle. An exception is Software Dock [4], which has been proposed as a systematic framework that provides that support. Software Dock is a system of loosely coupled, cooperating, distributed components. It provides software deployment agents that travel among hosts to perform software deployment tasks. Unlike DeSi, however, Software Dock does not focus on visualizing, automatically selecting, or evaluating a system's deployment architecture.

UML [12] is the primary notation for the *visual modeling* of today's software systems. UML's deployment diagram provides a standard notation for representing a system's software deployment architecture. Several recent approaches extend this notation via stereotypes [3,7]. However, using UML to visualize deployment architectures has several drawbacks: UML's deployment diagrams are static; they do not depict connections among hardware hosts; and they do not provide support for representing and visualizing the parameters that affect the key system properties (e.g., availability). For these reasons, we have opted not to use a UML-based notation in DeSi.

There are several examples of *visual software development environments* that have originated from industrial and academic research. For example, AcmeStudio [16] is an environment for modeling, visualizing, and analyzing software architectures. Environments such as Visual Studio [8] provide a toolset for rapid application development, testing, and packaging. In our context, the role of the DeSi environment is to support tailorable, scalable, and platform-independent modeling, visualization, evaluation, and implementation of highly distributed systems. For these reasons we opted for using Eclipse [2] in the construction of DeSi. Eclipse is a platform-independent IDE for Java with support for plug-ins. Eclipse provides an efficient graphical library (Draw2D) and

accompanying graphical editing framework (GEF), which we leveraged in creating visual representations of deployment architectures in DeSi.

4 The DeSi Environment

In this section, we discuss the architecture, implementation, and typical usage of the DeSi environment. We focus on the key architecture- and implementation-level decisions and the motivation behind them.

4.1 DeSi's Architecture

The overall architecture of the DeSi environment adheres to the model-view-controller (MVC) architectural style [6]. Figure 3 depicts the architecture. 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. Each is discussed in more detail below.

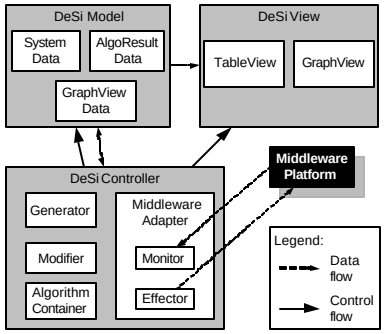


Figure 3. DeSi's architecture.

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 parameters outlined in Section2.1: 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, possi-

bly third-party, system implementation, deployment, and execution platform (depicted as a "black box" in Figure 3). 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 (recall Section2.2) 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.

The above components allow DeSi to be used to generate automatically 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 runtime 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 in Section4.2.3.

4.2 DeSi's Implementation

DeSi has been implemented in the Eclipse platform [2] using Java 1.4. DeSi's implementation adheres to its MVC architectural style. In this section, we discuss (1)the implementation of DeSi's extensible model, (2) the visualizations currently supported by DeSi, and (3) its capabilities for generating deployment scenarios, assessing a given deployment, manipulating system parameters and observing their effects, and estimating redeployments that result in improved system availability.

4.2.1 Model Implementation

The implementations of the *SystemData* and *AlgoResultData* components of DeSi's *Model* are simple: each one of them is implemented as a single class, with APIs for accessing and modifying the stored data. For example, *AlgoResultData*'s implementation provides an API for accessing and modifying the array containing the estimated deployment architecture, and a set of variables representing the achieved availability, algorithm's running time, and estimated time to change a given system from its current to its new deployment.

The *GraphViewData* component contains all the persistent data necessary for maintaining the graphical visualization of a given deployment architecture. It keeps track of information such as figure shapes, colors, placements, and labels that correspond to hardware hosts, software components, and software and hardware links. In our implementation of *GraphViewData*, each element of a distributed system (host, component, or link) is implemented via a corresponding *Figure* class that maintains this information (e.g., each host is represented as a single instance of the *HostFigure* class). Com-

ponents and hosts have unique identifiers, while a link is uniquely identified via its two end-point hosts or components.

The *GraphViewData* component's classes provide a rich API for retrieving and modifying properties of individual components, hosts, and links. This allows easy runtime modification of virtually any element of the visualization (e.g., changing the line thickness of links). Furthermore, the information captured inside *GraphViewData* is not directly tied to the properties of the model captured inside *SystemData*. For example, the *color* property of *HostFigure* can be set to correspond to the amount of available memory or to the average reliability with other hosts in the system.

4.2.2 View Implementation

The *TableView* component of DeSi's *View* subsystem displays the *Deployment Control Window*, shown in Figure 4. This window consists of five sections, identified by panel names: *Input*, *Constraints*, *Algorithms*, *Results*, and *Tables of Parameters*.

The *Input* section allows the user to specify different input parameters: (1) numbers of components and hosts; (2) ranges for component memory, frequency and event size; and (3) ranges for host memory, reliability and bandwidth. For centralized deployment scenarios we provide a set of text fields for specifying the properties of the central host. The *Generate* button on the bottom of this panel results in the (random) generation of a single deployment architecture that satisfies the above input. Once the parameter val-

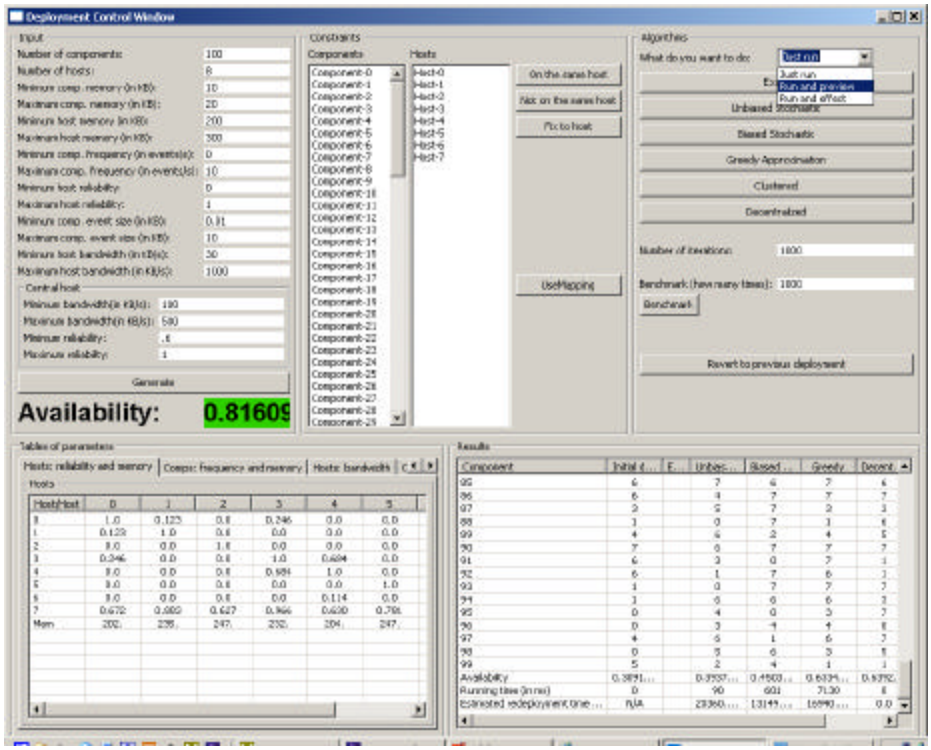


Figure 4. DeSi's Deployment Control Window

ues are generated, they are displayed in the *Tables of Parameters* section, which will be discussed in more detail below.

The *Constraints* section allows specification of different conditions for component (co-)location: (1) components that must be deployed on the same host, (2) components that may not be deployed on the same host, and (3) components that have to be on specific host(s). The three buttons on the right side of the *Constraints* panel correspond to these conditions. Consecutive clicks on the *Use Mapping* button, located on the bottom right side of the panel, enable and disable the above three buttons.

The *Algorithms* section allows the user to run different redeployment estimation algorithms by clicking the corresponding algorithm's button. There are three provided options for running the algorithms (depicted in the drop-down menu at the top of the *Algorithms* section in Figure 4): (1) *Just run*, which runs the algorithm and displays the result in the *Results* panel, (2) *Run and preview*, which runs the algorithm displays the results both in the *Results* panel and in the graphical view (discussed further below), and (3) *Run and effect*, which, in addition to actions described in option 2, also updates *SystemData* to set the current deployment to the output of the selected algorithm. The latter action can be reversed by clicking on the *Revert to previous deployment* button.

We have provided a benchmarking capability to compare the performance of various algorithms. The user can specify the number of times the algorithms should be invoked. Then, the user triggers via the *Benchmark* button a sequence of automatic random generations of new deployment architectures and executions of all algorithms.

The *Results* section displays the outcomes of different algorithms. For each algorithm, the output consists of (1) a new mapping of components to hosts, (2) the system's achieved availability, (3) the running time of the algorithm, and (4) the estimated time to effect the new deployment. Section 4.2.3 details the calculation of these results.

Finally, the *Table of Parameters* section provides an editable set of tables that display different system parameters. The user can view and edit the desired table by clicking on the appropriate tab from the tab list. The editable tables support fine-tuning of system parameters.

The goals of the *GraphView* component of DeSi's *View* subsystem were to (1) allow users to quickly examine the complete deployment architecture of a given system, (2) provide scalable and efficient displays of deployment architectures with large numbers of components and hosts, and (3) be platform independent. To this end we used the Eclipse environment's GEF plug-in, which consists of a library for displaying different shapes, lines, and labels and a facility for runtime editing of the displayed images.

GraphView provides a simple API for displaying hosts, components, and links. For example, displaying two connected hosts requires two consecutive calls of the *createHost* method, followed by the *createH2HLink* method. Figure 5 illustrates a deployment architecture with 100 components and 8 hosts. Network connections between hosts are depicted as solid lines, while dotted lines between pairs of hosts denote that some of the components on the two respective hosts need to communicate, but that there is no network link between them. Clearly, the existence of dotted lines indicates a decrease in the system's availability. Therefore, just by observing the number of dotted lines one can reason about the quality of a given deployment architecture.

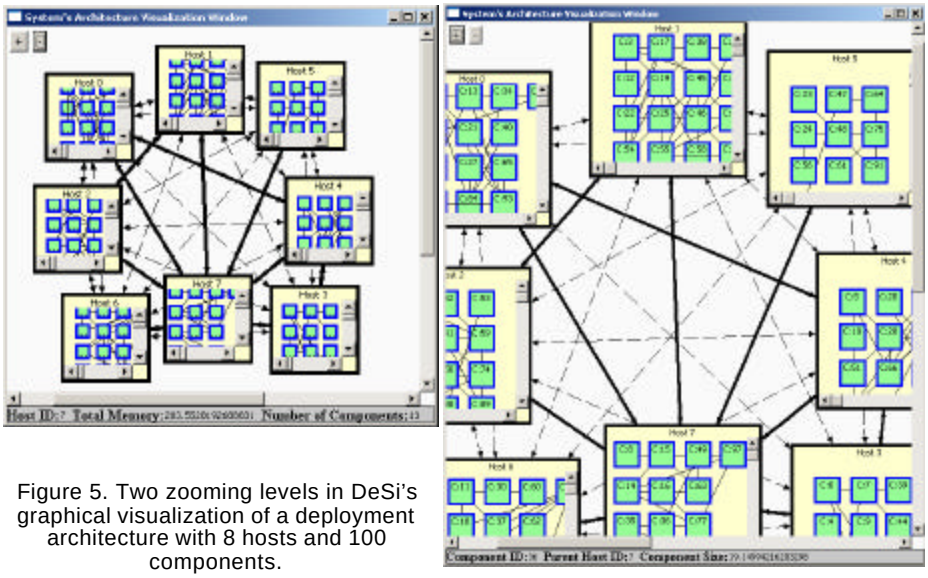
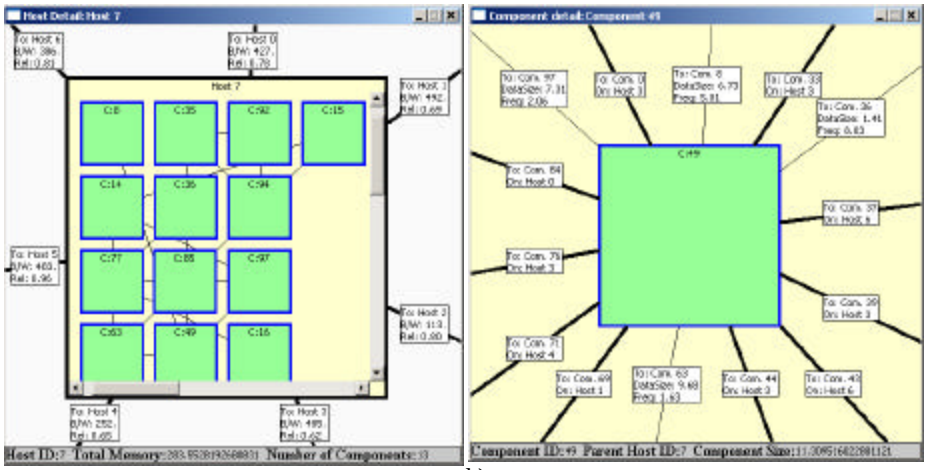


Figure 5. Two zooming levels in DeSi's graphical visualization of a deployment architecture with 8 hosts and 100 components.



a) b)

Figure 6. Detailed view of (a) a single host and (b) a single component in DeSi.

For systems with large numbers of hosts and components, visualizing the system and its connectivity becomes a challenge. For this reason, *GraphView* supports zooming in and out (see Figure 5), and provides the ability to "drag" hosts and components on-screen, in which case all relevant links will follow them. For the same reason, we do not display connections between components residing on different hosts. If such connections exist, the components will have thicker borders. A thin border on a component denotes that it does not communicate with remote components.

GraphView has several other features that allow users to easily visualize and reason about a given deployment architecture. A user can get at-a-glance information on any

of the hosts and components in the system. Selection of a single graphical object displays its information in the status bar at the bottom of the window (see Figure 5). The displayed information can easily be changed or extended through simple modifications to *GraphView* to include any (combination) of the information captured in the *SystemData* component. Detailed information about a host or component can be displayed by double-clicking on the corresponding graphical object. The *DetailWindow* for a host, shown in Figure 6a, displays the host's properties in the status bar, the components deployed on the host, the host's connections to other hosts, and the reliabilities and bandwidths of those connections. Similarly, the *DetailWindow* for a component, shown in Figure 6b, displays the component's properties and its connections to other components.

4.2.3 Controller Implementation

The implementation of DeSi *Controller*'s *Generator* component provides methods for (1) generating random deployment problems, (2) producing a specific (initial) deployment that satisfies the parameters and constraints of a generated problem, and (3) updating DeSi *Model*'s *SystemData* class accordingly. This capability allows us to rapidly compare the different deployment algorithms discussed in Section2.2. *Generator* is complemented by the class implementing the *Controller*'s *Modifier* component. *Modifier* provides facilities for fine-tuning system parameters (also recall the discussion of editable tables in Section4.2.2), allowing one to assess the sensitivity of a deployment algorithm to specific parameters in a given system.

Each deployment algorithm in DeSi *Controller*'s *AlgorithmContainer* component is encapsulated in its own class, which extends the *AbstractAlgorithm* class. *AbstractAlgorithm* captures the common attributes and methods needed by all algorithms (e.g., *calculateAvailability*, *estimateRedeploymentTime*, and so on). Each algorithm class needs to implement the abstract method *execute*, which returns an object of type *AlgorithmResult*. A bootstrap class called *AlgorithmInvoker* provides a set of static methods that instantiate an algorithm object and call its *execute* method. The localization of all algorithm invocations to one class aids DeSi's separation of concerns and enables easy addition of new (kinds of) algorithms.

Finally, DeSi provides the *Middleware Controller* component which can interface with a middleware platform to capture and display the monitoring data from a running distributed system, and invoke the middleware's services to enact a new deployment architecture. This facility is independent of any particular middleware platform and only requires that the middleware be able to provide monitoring data about a distributed system and an API for modifying the system's architecture. DeSi does not require a particular format of the monitoring information or system modification API; instead, the goal is to employ different wrappers around the *Middleware Controller* component for each desired middleware platform.

As a "proof of concept," we have integrated DeSi with Prism-MW, an event-based, extensible middleware for highly distributed systems [9]. Prism-MW provides support for centralized deployment. It provides pluggable monitoring capabilities. It also provides a special-purpose *Admin* component residing on each host. The *Admin* component is in charge of gathering the monitoring information, sending it to the central host, and performing changes to the local subsystem by migrating components. Prism-MW

also provides a *Deployer* component residing on the central host. The *Deployer* component controls the redeployment process, by issuing events to remote *Admin* components to perform changes to their local configurations. We have wrapped DeSi's *Middleware Controller* as a Prism-MW component that is capable of receiving events with the monitoring data from Prism-MW's *Deployer* component, and issuing events to the *Deployer* component to enact a new deployment architecture. Once the monitoring data is received, *Middleware Controller* invokes the appropriate API to update DeSi's *Model* and *View* subsystems. 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, *Middleware Controller* issues a series of events to Prism-MW's *Deployer* component to update the system's deployment architecture.

5 Discussion

The goal of DeSi is to allow visualization of different characteristics of software deployment architectures in highly distributed settings, the assessment of such architectures, and possibly their reconfiguration. In this section we evaluate DeSi in terms of four properties that we believe to be highly relevant in this context: tailorability, scalability, efficiency, and ability to explore the problem space. Each property is discussed in more detail below.

5.1 Tailorability

DeSi is an environment intended for exploring a large number of issues concerning distributed software systems. As discussed above, to date we have focused on the impact a deployment architecture has on a system's availability. However, its MVC architecture and component-based design allow it in principle to be customized for visualizing and assessing arbitrary system properties (e.g., security, fault-tolerance, latency). All three components of DeSi's *Model* subsystem (*SystemData*, *GraphViewData*, and *AlgoResultData)* could be easily extended to represent other system properties through the addition of new attributes and methods to the corresponding classes. Another aspect of DeSi's tailorability is the ability to add new *View*s or modify the existing ones. Clear separation between DeSi *View* and *Model* components makes creating different visualizations of the same model easy. DeSi also enables quick replacement of the *Control* components without modifying the *View* components. For example, two *Control* components may use the *GraphView*'s API for setting the thickness of inter-host links differently: one to depict the reliability and the other to depict the available bandwidth between the hosts. Finally, DeSi also provides the ability to interface with an arbitrary middleware platform.

5.2 Scalability

DeSi is targeted at systems comprising many components distributed across many hosts. DeSi supports scalability in the (1) size of its *Model*, (2) scalability of its *View*s, and (3) scalability of the *Controller*'s algorithms. *Model*s of systems represented in DeSi can be arbitrarily large since they are centralized and capture only the subset of system properties that are of interest. Another aspect of DeSi's scalability are *Controller*'s algorithm implementations: with the exception of the *ExactAlgorithm*, all of the algorithms are polynomial in the number of components. We have tested DeSi *Model*'s

scalability by generating random models with hundreds of hosts and thousands of components, and *Controller*'s scalability by successfully running the algorithms on these models. Finally, the combination of the hierarchical viewing capabilities of the DeSi *View* subsystem (i.e., system-wide view, single host view, single component view), the zooming capability, and the ability to drag components and hosts to view their connectivity, enables us to effectively visualize distributed systems with very large numbers of components and hosts.

5.3 Efficiency

A goal of this work was to ensure that DeSi's scalability support does not come at the expense of its performance. As a result of developing the visualization components using Eclipse's GEF, DeSi's support for visualizing deployment architectures exhibits much better performance than an older version that was implemented using Java Swing libraries. We also enhanced the performance of the *GraphView* component by repainting only parts of the screen that correspond to the modified parts of the model. As discussed in Section4.2.2, we also provide three options for running the redeployment algorithms. This enables us to customize the overhead associated with running the algorithms and displaying their results. Finally, while the efficiency and complexity of the redeployment algorithms is not the focus of this paper, with the exception of the exact algorithm, all of the provided algorithms run in polynomial time [10].

5.4 Exploration Capabilities

The nature of highly distributed systems, their properties, and the effects of their parameters on those properties is not well understood. This is particularly the case with the effect a system's deployment architecture has on its availability [10]. The DeSi environment provides a rich set of capabilities for exploring deployment architectures of distributed systems. DeSi provides side-by-side comparison of different algorithms along multiple dimensions (achieved availability, running time, and estimated redeployment time). It also supports tailoring of the individual parameters of the system model, which allows quick assessment of the sensitivity of different algorithms to these changes. Next, DeSi provides the ability to tailor its random generation of deployment architectures to focus on specific classes of systems or deployment scenarios (e.g., by modifying desired ranges of certain parameters such as minimum and maximum frequency of component interactions). DeSi supports algorithm benchmarking by automatically generating, assessing, and comparing the performance of different algorithms for a large number of randomly generated systems. DeSi's extensibility enables rapid evaluation of new algorithms. Finally, through its *MiddlewareAdapter*, DeSi provides the ability to visualize, reason about, and modify an *actual* system.

6 Conclusion

A distributed software system's deployment architecture can have a significant impact on the system's properties. In order to ensure the desired effects of deployment, those properties need to be assessed. They will depend on various system parameters (e.g., reliability of connectivity among hosts, security of links between hosts, and so on). Existing tools for visualizing system deployment (e.g., UML [12]) depict only distributions of components onto hosts, but lack support for specifying, visualizing, and analyzing different factors that influence the quality of a deployment.

This paper has presented DeSi, an environment that addresses this problem by supporting flexible and tailorable specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale, highly distributed systems. The environment has been successfully used to explore large numbers of postulated deployment architectures. It has also been integrated with the Prism-MW software architecture implementation and deployment platform to support the exploration of deployment architectures of *actual* distributed systems.

Our experience to date with DeSi has been very promising. At the same time, it has suggested a number of possible avenues for further work. We plan to address issues such as tailoring DeSi for use in exploring other non-functional system properties (e.g., security), improving existing DeSi visualizations (e.g., planarizing the graphs of hosts and components), creating new views (e.g., visually displaying the performance of different algorithms), and integrating DeSi with other middleware platforms (e.g., different implementations of CORBA).

7 References

1. A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. Technical Report, Dept. of Computer Science, University of Colorado, 1998.
2. Eclipse. http://eclipse.org/
3. J. Greenfield (ed.). UML Profile for EJB. *Public Review Draft JSR-000026*, Java Community Process, 2001.
4. R. S. Hall, D. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *21st International Conference on Software Engineering (ICSE'99)*, pp. 174-183, Los Angeles, CA, May 1999.
5. IEEE Standard Computer Dictionary: *A Compilation of IEEE Standard Computer Glossaries.* New York, NY: 1990.
6. G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26-49, August/September 1988.
7. C. Luer, and D. Rosenblum. UML Component Diagrams and Software Architecture - Experiences from the Wren Project. *1st ICSE Workshop on Describing Software Architecture with UML*, pages 79-82, Toronto, Canada, 2001.
8. Microsoft Visual Studio. http://msdn.microsoft.com/vstudio/
9. M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *Middleware 2003*, Rio De Janeiro, June 2003.
10. M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. Technical Report *USC-CSE-2003-515*, 2003.
11. M. Mikic-Rakic and N. Medvidovic. Toward a Framework for Classifying Disconnected Operation Techniques. ICSE Workshop on Architecting Dependable Systems, Portland, Oregon, May 2003.
12. Object Management Group. The Unified Modeling Language v1.4. Tech. report, 2001.
13. D. C. Parkes, L. H. Ungar. An Auction-Based Method for Decentralized Train Scheduling. *International Conference on Autonomous Agents*, Montreal, Canada, 2001.
14. D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. Software Engineering Notes, Oct. 1992.
15. L.Ramaswamy, et. al. Connectivity Based Node Clustering in Decentralized Peer-to-Peer Networks. *Peer-to-Peer Computing* 2003.
16. B. Schmerl, and D. Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. *ICSE 2004*, Edinburgh, Scotland, 2004.
17. J. Weissman. Fault-Tolerant Wide-Area Parallel Computing. *IPDPS 2000 Workshop*, Cancun, Mexico, May 2000.