An Extensible Framework for Autonomic Analysis and Improvement of Distributed Deployment Architectures

Sam Malek Marija Mikic-Rakic Nenad Medvidovic

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

ABSTRACT

A distributed software system's deployment architecture can have a significant impact on the system's properties, which depend on various system parameters, such as network bandwidth, frequencies of software component interactions, 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 a common framework for improving a system's deployment architecture has resulted in ad hoc solutions. In this paper, we present an extensible framework that guides the design and development of solutions to this problem, enables the extension and reuse of the solutions, and facilitates autonomic analysis and redeployment of a system's deployment architecture.

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 effective than others in terms of the desired system characteristics such as scalability, evolvability, mobility, latency, security, and availability. 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., 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 may fluctuate at run time; and (3) the space of possible deployment architectures is extremely large, thus finding the optimal deployment is rarely feasible [6]. Furthermore, different desired system characteristics may be conflicting. For example, 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 emerging class of decentralized systems, which are characterized by the limited system-wide knowledge and lack of a known, single point of control. Selection of a globally appropriate deployment architecture has to be made in a decentralized fashion, using partial, local information.

In our previous work [4,6,7,8] we have identified and addressed a subset of the above challenges in the context of disconnected operation. 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. Although effective in improving system availability in the context of disconnected operation, our methodology was not directly extensible to include system characteristics other than availability.

The work described in this paper builds on our previous experience to address the above challenges. We have developed an extensible framework for analyzing and improving distributed deployment architectures via run time redeployment. The framework is extensible along several dimensions: (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; and (5) flexible support for both centralized and decentralized systems. The framework's objective is to provide a library of pluggable, reusable, and customizable components that can be leveraged in addressing a variety of distributed system deployment scenarios. Although this work is still in progress, our experience with the framework has been promising. We illustrate the design and implementation of two different instances of the framework.

The remainder of the paper is organized as follows. Section 2 briefly outlines the related work. Section 3 presents the deployment improvement framework. Sections 4 discusses our supporting tools that facilitate the realization of the framework. Section 5 discusses some of our experience with the framework to date. The paper concludes with an outline of the future work.

2 RELATED WORK

The problem of improving a system's deployment architecture has been studied by several researchers, including I5 [1], Coign [2], Sekitei [3], and our own prior work [4,6]. While all of these 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 these results.

Related to our work is the research on architecture based adaptation frameworks, examples of which are [9,10]. As opposed to general purpose architecture-based adaptation frameworks, we are only considering a specific kind of architecture-based 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.

3 THE FRAMEWORK

In this section we describe our deployment improvement framework's components, the associated functionality of each component, and the dependency relationships that guide their interaction. We will also describe the framework's instantiation for two classes of solutions developed with the goal of improving deployment architectures.

3.1 Framework Design

Figure 1 shows the framework's overall structure and the relationships among its 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.

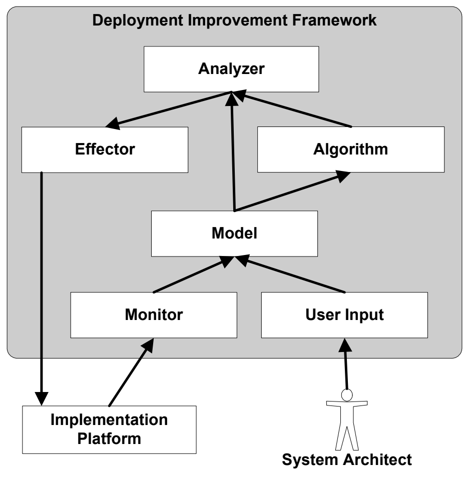

Figure 1. Deployment improvement framework.

Model. This component maintains the representation of the system's deployment architecture. The model is composed of four parts: hosts, components, physical links between hosts, and logical links between components. Each of these parts of the model 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 modeled 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 modeled.

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 general categories of algorithms: exact and approximative. Exact algorithms produce optimal results (e.g., deployments with minimal overall latency), but are exponentially complex, which limits their applicability only to systems with very small numbers of components and hosts. On the other hand, approximative algorithms in general produce suboptimal 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. Each monitor contains an implementation platform-dependent part that hooks into

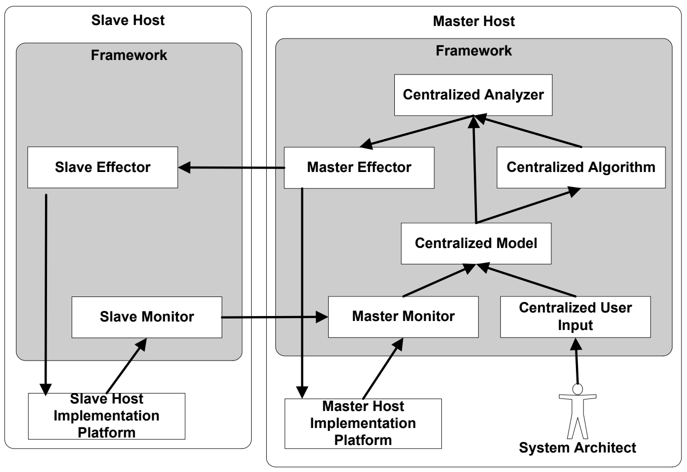

Figure 2. Framework's centralized instantiation.

the platform and performs the actual monitoring of the system. The monitored data is passed to an implementation platform-independent part that determines if the data is stable enough [8] to be passed onto the model. The monitor provides a threshold variable that is set to determine the level of fluctuation acceptable for the monitored data to be considered stable.

Effector. Just like monitors, effectors are also composed of two parts: (1) an implementation platform-dependent part that hooks into the platform to perform the redeployment of software components; (2) and an implementation 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 is also 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 Instantiating the Framework

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 Monitors* 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 Effectors.*

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 Effector* that are only responsible for the monitoring and redeployment on the host on which they are located. Each host has a *Decentralized Model* that contains a 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.

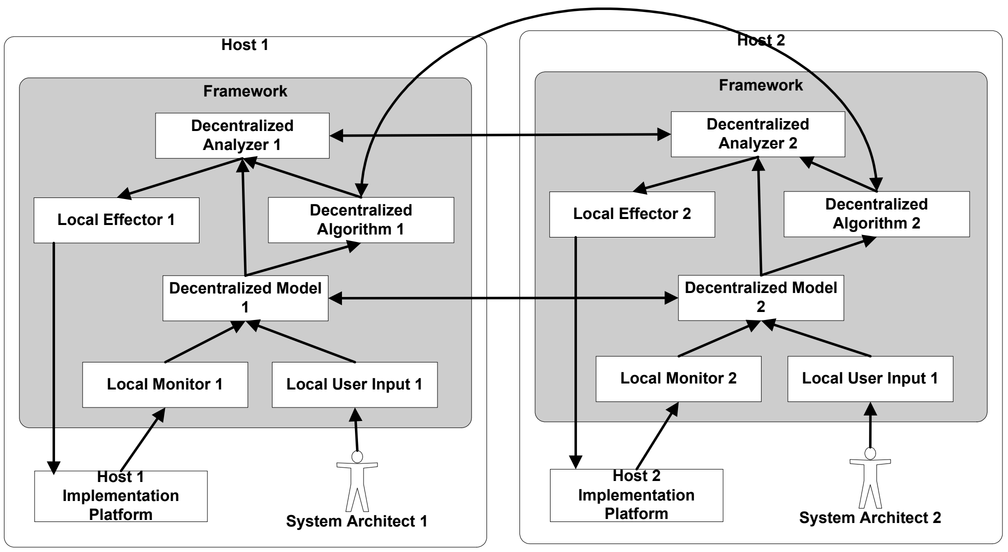

Figure 3. Framework's decentralized instantiation.

4 SUPPORTING TOOLS

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 briefly outline two supporting tools that we have leveraged in our work to date.

4.1 Implementation Platform

While the implementation platform is not an integral part of the framework, the framework depends on such a platform to provide support for monitoring the system and redeploying the components. Some of the requirements for the implementation platform are support for component-based development, event-based interaction, ability to migrate components, and ability to unintrusively monitor the system.

An example of an implementation platform that satisfies the above requirements is Prism-MW [5]. Prism-MW is a middleware platform that enables efficient implementation, deployment, and execution of distributed software systems in terms of their architectural elements: components, connectors, configurations, and events. It also provides extensible support for both monitoring and redeployment of resources at the architectural level. We have leveraged Prism-MW in implementing the above described monitor and effector components of the framework.

4.2 User Interface

DeSi [7] is a visual deployment exploration environment that supports specification, manipulation, and visualization of system parameters for large-scale and 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. For example, the architect is able to use a graphical environment to enter location and collocation constraints into the framework's model. DeSi also provides a visualization environment for graphically displaying the system's monitored data, deployment architecture, and the results of analyses. We have leveraged DeSi in implementing the *User Input* component of our framework.

5 EXPERIENCE TO DATE

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

5.1 Centralized Improvement of Deployment Architecture

In order to improve the objective of maximizing a system's availability and minimizing the latency we first created the 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 property.
- Each hosts has an available memory size property.
- Each logical link among components has a frequency of interaction and an average event size property.
- Each physical link among hosts has a network reliability, a network bandwidth, and a network transmission delay property.
- The system's model contains the location and collocation constraints, discussed in Section 3, 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 using the DeSi tool. All the modeled properties that are not entered by the user are monitored at run time and added to the model automatically. In the development of the implementation platform-dependent part of the monitor, we relied on Prism-MW's support for monitoring.

We have used three centralized algorithms, called Exact, Stochastic, and Avala [6]. 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 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 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 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 locational and collocational 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 Avala or stochastic 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 example also typically decrease the system's overall latency [6]. 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 Effectors*, which leverage Prism-MW's support for the redeployment of software components in the manner described in [8].

5.2 Decentralized Improvement of Deployment Architecture

In the development of the decentralized solution, we were able to reuse the centralized model by extending it to include the notion of "awareness". By "awareness" we denote 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 that it is aware of (i.e., directly connected to), 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 [4], that is based on an auction-based protocol to find a solution that maximize 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, described in [8].

6 CONCLUSION AND FUTURE WORK

A distributed software system's deployment architecture can have a significant impact on the system's properties. Finding a deployment architecture that 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. In this paper we have presented a design framework for analyzing and improving distributed deployment architectures, and our experience to date with the implementation of the framework for improving system availability and latency in both centralized and decentralized environments. Our initial evaluations indicate that by leveraging the framework 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 implementing and evaluating solutions for improving system characteristics beyond availability and latency. We also plan to devise mitigating techniques for situations where different desired system characteristics may be conflicting. These tasks will provide a basis for further assessment and evaluation of our framework.

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 REFERENCES

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

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

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

[5] 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.

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

[7] 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.

[8] 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.

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

[10]S. W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, N. Hu. Software Architecture-Based Adaptation for Grid Computing. *11th IEEE International Symposium on High Performance Distributed Computing* (HPDC'02), Edinburgh, Scotland, July 2002.