Submitted for publication

Coordinating Adaptations in Self-managing Systems

An-Cheng Huang, Shang-Wen Cheng, Peter Steenkiste, David Garlan, and Bradley Schmerl
School of Computer Science, Carnegie Mellon University
{pach, zensoul, prs, garlan, schmerl}@cs.cmu.edu

***Abstract*—*Self-managing systems are able to dynamically adapt to changes in the environment and user needs without human intervention, thus reduce the administrative overhead of system management. Many self-management modules are being developed to provide different self-management capabilities, and being able to reuse these existing modules will greatly reduce the necessary efforts for developers of self-managing systems. However, this requires infrastructure support for plugging different self-management modules into the same system and coordination mechanisms for resolving potential conflicts among adaptations proposed by these modules. In this paper, we propose a self-managing system architecture that addresses these two requirements. Using a case study, we show how multiple self-managing modules can work together on top of a shared infrastructure that provides low-level system access functionalities. We then focus on the issue of coordinating conflicting adaptations and propose coordination mechanisms based on observations from the case study.*

I. Introduction

The cost of managing computing systems is becoming a major issue as computing systems increasingly operate in heterogeneous environments, consist of diverse and distributed elements, and need to deal with changing user needs. To address this problem, an emergent research direction is to develop self-managing systems that can automatically adapt themselves to run-time environment changes according to high-level policies specified by human developers.

Many research projects are developing self-managing systems that focus on different aspects of self-management. For example, some may focus on maintaining the overall system performance, some tune the run-time parameters of individual system elements, and others react to element failures in the system. A natural question is: can we use these existing self-managing systems as modules to develop self-managing systems that can handle multiple aspects of self-management? We believe this can be achieved by (1) providing infrastructure support so that existing self-management modules can be plugged into a system, and (2) providing coordination mechanisms to resolve any potential conflicts among adaptations proposed by different self-management modules in a system.

In this paper, we propose a self-managing system architecture that fulfills the above requirements. First, let us describe our model of self-managing systems.

A. Self-management model

A self-managing system has been characterized as a collection of Autonomic Elements (AEs) [1]. An AE consists of one Autonomic Manager (AM) and the managed element(s). In such a system, each AM performs adaptations within its AE according to high-level policies established by developers and to interactions with other AMs.

This model can be expanded by adding *system-wide self-management*. In contrast to element-level adaptations such as changing the run-time parameters of the managed element, system-wide self-management performs higher-level adaptations such as inserting a new AE into the system, removing an old one from the system, and composing a collection of AEs to deliver the desired functionality. Therefore, in this model, there are two types of AMs. An *Element-level AM* (EAM) is the AM in the original model and is usually integrated with the managed element. A *System-wide AM* (SAM) performs system-wide adaptations to produce an optimal configuration of AEs according to high-level policies specified by developers.

We envision that various SAMs will be developed with different capabilities (e.g., managing the whole system vs. a few elements) and foci (e.g., performance vs. security concerns). The developer of a self-managing system can select the appropriate SAMs and plug them into the system to add system-wide self-management capabilities. Examples of SAMs under development include Libra [2], which finds the globally optimal configuration for a self-managing system, and Rainbow [3], which adapts to system and environment changes.

To summarize, a self-managing system may consist of multiple EAMs and SAMs. Each AM in the system monitors the system state, evaluates a high-level objective specified by developers, and acts on the managed system/element(s) accordingly. Such a system can in turn be used as an element in a higher-level system.

B. Application example

Let us use a video conferencing system to illustrate self-management at both the element and system scopes. As shown in Figure 1, the system is self-managed by two system-wide AMs. SAM1 focuses on dynamically finding a globally optimal (minimizing the total latency) configuration of three types of elements—video conferencing gateway (VGW), handheld proxy (HHP), and end-system multicast [4] proxy (ESMP)—for the users (who have different devices and conferencing applications). SAM2 has a more targeted focus—when the running VGW fails, SAM2 finds a lowest-cost VGW to replace it. On the other hand, at the element level, the three types of elements are also self-managing, i.e., each element includes an EAM. The ESMPs dynamically adjust the overlay multicast

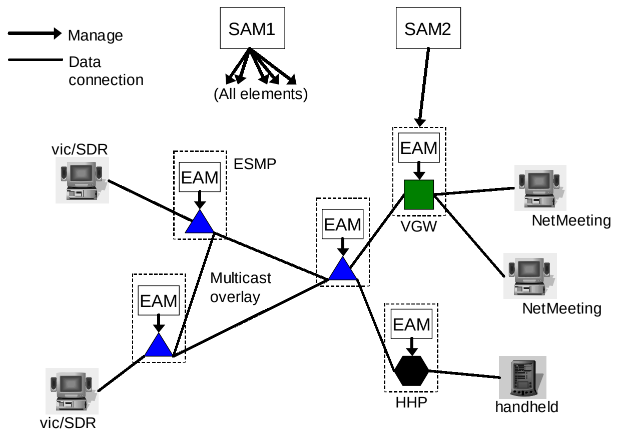

Fig. 1. A self-managing video conferencing system

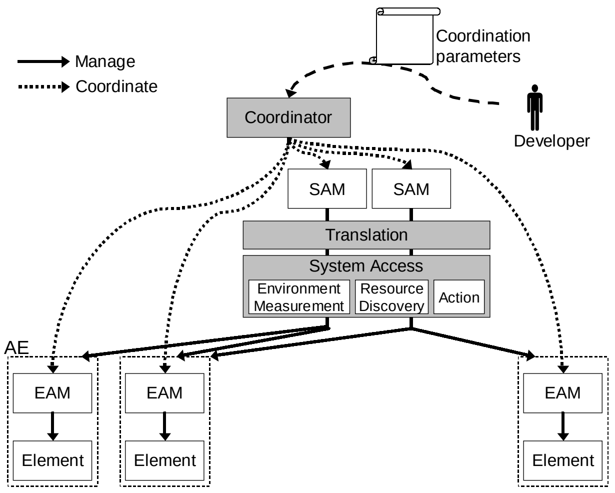

Fig. 2. Self-managing system architecture

tree based on latency and bandwidth performance. The VGW adjust the frame rate according to available bandwidth when forwarding video streams. The HHP encrypts the data if the handheld user is connecting from an open wireless network.

C. Dimensions of self-management

From the above example, we see that the adaptations performed by different AMs in the system may potentially conflict with each other. For example, when the VGW fails, SAM2 may find a replacement that is different from what SAM1 would use in a globally optimal configuration. Similarly, the ESMPs may be adjusting the overlay while SAM1 attempt to use a different set of ESMPs. Therefore, to integrate such a self-managing system, it is necessary to coordinate the adaptations of different AMs in the system.

To better understand when and how adaptations need to be coordinated, we categorize AMs along the following three dimensions.

Granularity: We identify three different granularities of self-management. (1) *Element-level*: An EAM performs self-management within its AE, e.g., the elements in the video conferencing example. (2) *Global system-wide*: A SAM performs self-management on the whole system, e.g., SAM1 finds the optimal composition of AEs in the video conferencing system. (3) *Local system-wide*: A SAM performs self-management on a small subset of system elements, e.g., SAM2 finds a new VGW to replace a failed one in the video conferencing system.

Time scale: AMs may be invoked at different time scales to perform self-management. Some AMs are “periodic” (e.g., SAM1 is invoked to look for the optimal configuration every x seconds), and we roughly categorize them into *long-term* (long interval between invocation) and *short-term* (short interval). Other AMs are invoked in response to certain events (e.g., SAM2 is invoked when the VGW fails), and we call them *triggered* AMs. Note that a global AM is likely to be long-term since its adaptations will likely incur a higher overhead (monitoring global system state, optimization, etc.) and cause more significant disruptions (e.g., replacing many elements). Similarly, short-term and triggered AMs may be invoked frequently, so they are likely to be local or element-level.

Concern: AMs may focus on different concerns about the system. For example, some AMs try to improve the performance of the system, and others try to ensure the security. Examples of other concerns include cost, availability, etc.

These three properties are closely related to the coordination of adaptations performed by different AMs. Granularity determines whether two AMs may conflict “spatially”, i.e., operate on the same element(s). Time scale determines whether two AMs may conflict “temporally”, i.e., operate at the same time. When two AMs conflict, the relative importance of their concerns will determine how they should be coordinated.

In the remainder of this paper, we introduce our architecture for self-managing systems, describe a case study, and propose mechanisms for coordinating adaptations in self-managing systems according to observations from the case study.

II. Self-managing system architecture

We propose the self-managing system architecture shown in Figure 2. Its three main parts—*coordinator*, *system access infrastructure*, and *translation layer*—are described below.

First, as already discussed, AMs may have different granularities, time scales, and concerns and may want to adapt the system in different and potentially conflicting ways. The coordinator receives “proposed” adaptations from the AMs and determines which ones should be executed and which ones should be rejected such that the system as a whole exhibits coherent behaviors. Later we will describe our initial design of the coordination mechanisms.

Secondly, AMs need to access the managed elements/system to gather information for decision making and to perform self-management actions. This is simple for EAMs since they are tightly integrated with the managed elements. However, SAMs cannot access the elements/system directly. If each SAM must have its own implementation of system access functionality, it would increase the development cost of SAMs and also introduce potential conflicts (e.g., different SAMs measure the same property differently). Therefore, we have identified three commonly recurring mechanisms—environment measurement, resource discovery, and action—that are needed by SAMs to obtain information and effect changes. We propose a shared system access infrastructure that provides these mechanisms so that every SAM does not need to reimplement the system access functionality. This also ensures that every AM uses the same low-level system access mechanisms.

Finally, the translation layer is necessary since each SAM may model the system/elements at different levels of abstraction (e.g., one SAM identifies the request queues within an element while another only identifies that element as a network host). When a SAM needs to access the system, the access operation is transformed by the translation layer so that the system access infrastructure can understand it (e.g., translating an abstract element identifier to the actual IP address and port).

Next, we present a case study of integrating a self-managing system using this proposed architecture.

III. Case Study

We have developed a self-managing video conferencing system similar to the one depicted in Figure 1. The current system prototype can establish and maintain a video conferencing session for users who have different devices and conferencing applications. As described in Section I-B, there are three types of system elements: VGW, HHP, and ESMP. We also have two system-wide AMs: Libra [2] (as SAM1) and Rainbow [3] (as SAM2). Libra is responsible for finding the globally optimal configuration of elements for the users, and Rainbow reacts to ESMP failure by finding the lowest-cost ESMP to replace the failed one.

We have built a shared infrastructure that provides the basic system access functionalities. We use the global network positioning (GNP) [5] approach to provide network latency estimates between network nodes. Network-sensitive service discovery (NSSD) [2] is used for finding suitable system components. Wrappers are used to provide a common interface for effecting changes in the system.

We have designed and implemented a translation infrastructure that consists of a translation repository and individual translators. The repository contains mappings between the abstract models used by the SAMs and the actual system (e.g., mapping an abstract element to a combination of an IP address and a port). The SAMs communicate with the system access infrastructure through the translators, which uses the mapping information stored in the repository to translate between the SAMs' high-level representation and the system-level representation used by the system access infrastructure.

The current prototype demonstrates that a shared system access infrastructure and a translation layer allow independent AMs to work together in the same system. On the other hand, compared to the system described in Section I-B, the scope of coordination in our current system is more limited, mostly due to limitations of the component implementations. For example, among the three elements, only ESMP has self-management capabilities, and replacing the running VGW requires user intervention, making it infeasible to invoke Libra periodically to find the optimal configuration.

As a result, we currently use a simple pattern of coordination: Libra is only invoked to construct the initial configuration. Once the conferencing session starts, the ESMPs will dynamically adjust the overlay among them. If an ESMP fails, Rainbow will override ESMPs' adaptations by replacing the failed ESMP with a new one. This new set of ESMPs will then look for the best overlay, and so on.

We are now studying the coordination issues by introducing more interesting coordination patterns into the system. In the remainder of this paper, we describe our current design of the coordination mechanisms.

IV. Coordinating self-management adaptations

In our architecture, all AMs in the system perform self-management adaptations by continuously *monitoring* the element/system state, *evaluating* the observed state to determine the best action, and performing the *action* to change the state. How different adaptations should be coordinated depends on the relations between them. For example, given adaptations A_1 and A_2, if A_1 observes the system state at time t_1 and performs an action at t_2, and A_2 observes at t_3 and acts at t_4 ($t_1 < t_2 < t_3 < t_4$), then A_1 and A_2 are independent, i.e., they do not conflict with each other. Similarly, if A_1 observes element e_1 and changes e_1, and A_2 observes element e_2 and changes e_2, they are also independent. On the other hand, if A_1 and A_2 both observe element e_1, but A_1 changes e_2 while A_2 changes e_3, then A_1 and A_2 do not directly conflict with each other; however, executing both of them may eventually cause problems. Finally, if A_1 and A_2 want to change the same element at the same time, then only one of them should be executed. Therefore, conceptually, the coordinator can construct a "conflict graph" based on the relations among adaptations. Such a graph identifies clusters of adaptations where adaptations within the same cluster conflict with each other, and adaptations from different clusters are independent of each other. Next, we look at two important relations—*temporal* and *spatial* relations—among adaptations and how they affect the conflicts between adaptations.

A. Temporal and spatial relations

Let us look at these relations more formally. Each adaptation performed by an AM can be seen as a combination of a "read" (monitoring and evaluation) and a "write" (action) operations on the system state. Given an adaptation A, we define the following notations.

• $r(A)$ and $w(A)$ represent the time when A reads from the system state (observes) and the time when A writes to the state (acts), respectively. Note that we assume the state is changed at time $w(A)$, but in reality, it may take some time for all changes to propagate through the system.

• $M(A)$ is the set of elements read (monitored) by A. For example, if A reads the load of element e_1, then $e_1 \in M(A)$; similarly, if A reads the latency between e_1 and e_2, then $\{e_1, e_2\} \subseteq M(A)$.

• $M'(A)$ is the set of elements "indirectly" read by A. For example, if A reads the video quality on e_1, and the video quality is affected by the frame rate output by e_2, then $e_2 \in M'(A)$.

• Similarly, $C(A)$ and $C'(A)$ represent the sets of elements written (changed) and indirectly written by A, respectively.

- $R(A) = M(A) \cup M'(A)$, i.e., the "read set" of A.
- $W(A) = C(A) \cup C'(A)$, i.e., the "write set" of A.
- $RW(A) = R(A) \cup W(A)$.

Given two adaptations A_1 and A_2, there are two kinds of temporal relations:

(1) If "$r(A_1) < w(A_1) < r(A_2) < w(A_2)$", then A_1 and A_2 are *temporally independent*.

(2) If "$r(A_1) < r(A_2) < w(A_1) < w(A_2)$" or "$r(A_1) < r(A_2) < w(A_2) < w(A_1)$", then they are *temporally dependent*.

If two adaptations are temporally independent, they can both be executed without conflicting with each other. (Note that they may still conflict at a higher level, e.g., one wants to increase security, thus decreasing performance, while another wants to do the opposite. We consider this a policy-level issue that can be addressed using the coordination patterns described later.) On the other hand, if they are temporally dependent (i.e., both reads precede both writes), their spatial relation will determine whether they conflict with each other. There are four kinds of spatial relations between A_1 and A_2:

(1) $RW(A_1) \cap RW(A_2) = \emptyset$: A_1 and A_2 are *spatially independent*, so both can be executed without conflicting with each other.

(2) $R(A_1) \cap R(A_2) \neq \emptyset$, $W(A_1) \cap RW(A_2) = \emptyset$, and $W(A_2) \cap RW(A_1) = \emptyset$: A_1 and A_2 are *read dependent*. If the coordinator allows both adaptations to be executed, they will not conflict with each other in terms of the elements they change. However, in some cases, this may cause long-term problems. For example, suppose one AM performs an adaptation that reduces the frame rate output of the video server in response to a drop in available bandwidth, and another AM performs an adaptation that reduces the codec quality at an intermediate video transcoder in response to the same problem. If both adaptations are executed, the system as a whole may overcompensate for the bandwidth problem.

(3) $W(A_1) \cap R(A_2) \neq \emptyset$ and $W(A_1) \cap W(A_2) = \emptyset$: A_1 and A_2 are *read-write dependent*, i.e., A_1 will change the state that A_2 is based on. As a result, if both adaptations are allowed to be executed, A_2 becomes *stale*. Although A_1 and A_2 do not make conflicting changes to the system state, it is likely that, given the new state, A_2 may no longer be necessary or may require some modifications.

(4) $W(A_1) \cap W(A_2) \neq \emptyset$: A_1 and A_2 are *write dependent*, i.e., they attempt to make conflicting changes (changing the same element(s)). Only one of them can be executed.

B. Simplified coordination model

From the above analysis of the temporal and spatial relations between adaptations, we see that coordinating adaptations comprehensively is difficult for two particular reasons. (1) Since AMs adapt at different time scales, the coordinator cannot know when each AM will want to perform its actions; therefore, the coordinator may need to wait indefinitely (during which time no adaptations can proceed) to find all temporally dependent adaptations. (2) For read dependent and read-write dependent adaptations, it will be difficult for the coordinator to know whether they will in fact conflict with each other eventually. Therefore, as a first step, we adopt a simplified coordination model as follows.

The coordinator divides time into *coordination windows*. AMs monitor and evaluate the system state independently and propose the desired actions to the coordinator. The coordinator assumes that all proposed adaptations within each window are based on the same system state, i.e., they are temporally dependent (since all their writes are still pending). At the end of each window, the coordinator examines the spatial relations among all proposed adaptations within the window and groups the adaptations into sets such that adaptations from different sets are spatially independent, and adaptations within each set are read, read-write, or write dependent. Such a grouping is an approximation of the actual "conflict graph". Finally, from each set the coordinator selects one adaptation and informs the proposing AM to execute it (i.e., always allow only one adaptation in read, read-write, and write dependent cases).

Of course, the assumption above will not always hold, e.g., one adaptation based on the same state as adaptations in window i does not propose an action until window $i+1$ because the evaluation takes too long. In this case, the late adaptation will be rejected since it is based on stale state. Therefore, each adaptation should be associated with a time stamp of the system state that it is based on so that the coordinator can handle stale adaptations. Another issue to be resolved is the selection of an appropriate size for the coordination window. If the window is too large, the effectiveness of self-management is decreased (e.g., increased reaction time). If it is too small, many temporally dependent adaptations will end up in different windows, resulting in many stale adaptations. Furthermore, coordination windows can be dynamic and adaptive. For example, when an event occurs that triggers adaptations, the coordinator, expecting to receive proposed adaptations in response to the event, can preemptively end the previous window and start a new one. In addition, since these triggered adaptations are likely quick repairs, the coordinator can reduce the window size to improve the reaction time for these triggered adaptations.

C. Coordination patterns

Using the above model, after grouping adaptations into spatially independent sets, the remaining task of the coordinator is to select one adaptation from each set of spatially dependent adaptations. This selection is determined by the *coordination pattern* that the developer integrating the system wants to enforce, which depends on both the concerns of the individual AMs and any additional concerns of the developer. For example, if security is more important than performance, the developer may want to enforce a pattern where a security-based adaptation always takes precedence over a performance-based one if they conflict with each other. We now use the video conferencing system in Figure 1 to illustrate several examples of common patterns and how a developer can specify the desired pattern to the coordinator.

(1) *Compete*: When AMs have "comparable" concerns, their adaptations can be compared, and the best one selected. For example, SAM1 proposes an action that reduces the latency by 100 ms, and an ESMP proposes to change its connection to reduce the latency by 60 ms. In this case, the coordinator can

compare these two actions and let SAM1 execute its action. Note that this pattern can be coupled with the developer's preference, e.g., the developer may want the ESMP's action to be selected unless SAM1's action is at least twice as good. To specify such a pattern, the developer can specify a *weight* parameter for each AM, for example, weights 1 and 2 for SAM1 and ESMP, respectively.
(2) *Override*: When the concerns of AMs are not comparable, the developer may have a strict precedence among the different concerns. For example, if cost is more important than performance, the developer may want SAM2's adaptations to always take precedence over SAM1's when they conflict with each other. This pattern can be specified by assigning a *priority* parameter to each AM (priority 1 for SAM1 and 2 for SAM2).
(3) *Divide and conquer*: Different AMs may work best under different conditions. For example, SAM2 is suitable for quick repair, and SAM1's adaptations (with higher overhead and disruption) are more cost-effective when the latency has become relatively high. Therefore, when SAM1's and SAM2's adaptations conflict with each other, the developer may want SAM2's action to be selected when the latency is below a threshold (otherwise, select SAM1's). Such a pattern can be specified using a *precondition* parameter: preconditions (latency $> x$) and (latency $\leq x$) for SAM1 and SAM2, respectively. Then each AM's adaptations will only be selected if its precondition is true.
(4) *Global objective*: There may be a global objective that the developer wants to optimize. In the previous example, SAM1 and SAM2 are both capable of repairing the system configuration when a VGW fails. However, the developer wants to minimize the disruption, so the action with a shorter disruption should be selected. Therefore, the developer should be able to specify a *global objective* parameter—in this case, the minimization of the disruption caused by adaptations—such that different AMs' adaptations are compared accordingly.
(5) *High-level constraints*: In addition to the above patterns, the developer may also have high-level constraints on whether an AM's actions should be selected. For example, the developer may impose a constraint on the disruption caused by SAM1's actions so that an action proposed by SAM1 can only be selected if the disruption it causes is shorter than x seconds. Such a high-level constraint for an AM can also be specified using the precondition parameter, for example, in this case, the developer can specify the precondition (disruption $\leq y$ seconds) for SAM1.

V. Related work

Many previous studies have designed and implemented self-management capabilities within individual elements. For example, end-system multicast endpoints that automatically modify the multicast overlay according to available bandwidth [4], a video streaming controller that adjusts the video quality based on available bandwidth [6], and a cluster frond-end that distributes request within the cluster based on request locality [7] System-wide self-management capabilities have also been studied, for example, automatically composing a series of adaptors to adapt a server's output to a client's input [8], [9], dynamically allocating resources for resource-intensive Grid applications [10], and the Libra and Rainbow projects used in this paper.

In contrast to our bottom-up approach, others have looked at a top-down approach. For example, Wolpert et al. presented an approach for deriving appropriate objectives of individual agents in a multi-agent system according to a global objective [11]. Ours approach may be more suitable when there is no explicit global objective or when the global objective is not a summary of local objectives.

VI. Conclusions

In this paper, we have proposed a self-managing system architecture that allows developers to integrate existing self-management modules into a coherent system. We have presented a case study where we developed a self-managing video conferencing system using self-managing elements and system-wide self-management modules. We have constructed a system access infrastructure and a translation layer that are necessary for different self-management modules in the system to access the managed elements through a common interface. Finally, we have described our initial design of coordination mechanisms for enforcing the desired coordination patterns based on the temporal and spatial relations between different adaptations.

References

[1] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] A.-C. Huang and P. Steenkiste, "Network-Sensitive Service Discovery," in *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[3] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste, "Using Architectural Style as a Basis for System Self-repair," in *Proceedings of the Third Working IEEE/IFIP Conference on Software Architecture*, Aug. 2002.

[4] Y. Chu, S. Rao, and H. Zhang, "A Case for End System Multicast," in *Proceedings of ACM Sigmetrics*, June 2000.

[5] T. S. E. Ng and H. Zhang, "Predicting Internet Network Distance with Coordinates-Based Approaches," in *Proceedings of IEEE INFOCOM 2002*, June 2002.

[6] P. Chandra, Y.-H. Chu, A. Fisher, J. Gao, C. Kosak, T. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Customizable Resource Management for Value-Added Network Services," *IEEE Network*, vol. 15, no. 1, Jan. 2001.

[7] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware Request Distribution in Cluster-based Network Servers," in *Proceedings of ASPLOS-VIII*, Oct. 1998.

[8] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao, "The Ninja Architecture for Robust Internet-Scale Systems and Services," *IEEE Computer Networks, Special Issue on Pervasive Computing*, vol. 35, no. 4, Mar. 2001.

[9] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko, "Automated Planning for Open Architectures," in *Proceedings for OPENARCH 2000 – Short Paper Session*, Mar. 2000, pp. 17–20.

[10] C. Liu, L. Yang, I. Foster, and D. Angulo, "Design and Evaluation of a Resource Selection Framework for Grid Applications," in *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[11] D. Wolpert, K. Wheeler, and K. Tumer, "Collective Intelligence for Control of Distributed Dynamical Systems," *Europhysics Letters*, vol. 49, no. 6, Mar. 2000.