Unity: Experiences with a Prototype Autonomic Computing System

David M. Chess, Alla Segal, Ian Whalley, Steve R. White
IBM Thomas J. Watson Research Center
{chess,segal,inw,srwhite}@us.ibm.com

Abstract

The behavior of a system results from the behaviors of its components, and from the interactions and relationships among them. In order to create computing systems that manage themselves, we will need to design both the behaviors of the individual elements, and the relationships that are formed among them. This paper describes a research project called Unity, carried out at IBM's Thomas J. Watson Research Center, in which we explore some of the behaviors and relationships that will allow complex computing systems to manage themselves; to be self-configuring, self-optimizing, self-protecting, and self-healing. The four principle aspects of Unity that we will examine are the overall architecture of the system, the role of utility functions in decision-making within the system, the way the system uses goal-driven self-assembly to configure itself, and the design patterns that enable self-healing within the system.

1. Introduction

The vision of autonomic computing [1] is of a world in which computing systems manage themselves to a far greater extent than they do today. It is a world, in particular, where interacting sets of individual computing elements regulate and adapt their own behavior in order to respond to a wide range of changing conditions with only high-level direction from humans.

The behavior of a system results from the behaviors of its components, and from the interactions and relationships among them. In order to create computing systems that manage themselves, we will need to design both the behaviors of the individual components, and the relationships that are formed among them. This paper describes a research project called Unity, carried out at IBM's Thomas J. Watson Research Center, in which we explore some of the behaviors and relationships that will allow complex computing systems to manage themselves; to be self-configuring, self-optimizing, self-protecting, and self-healing. The four principle aspects of Unity that we will examine are the overall architecture of the system, the role of utility functions in decision-making within the system, the way the system uses goal-driven self-assembly to configure itself, and the design patterns that enable self-healing within the system.

2. The structure of Unity

The essential structure of Unity follows that outlined in [1] and [2]. The components that make up the Unity system are implemented as autonomic elements; system components that manage themselves and deliver services to humans and to other autonomic elements. In our approach, every component of a system is an autonomic element. This includes computing resources such as a database, a storage system, or a server. It also includes higher-level elements with some management authority, such as a workload manager or a provisioner. And it includes elements that assist other elements in doing their tasks, such as a policy repository, a sentinel, a broker, or a registry. In the Unity project we are particularly interested in the properties that all the subtypes of autonomic elements have in common.

Each autonomic element is responsible for its own *internal* autonomic behavior: for managing the resources that it controls, and for managing its own internal operations, including self-configuration, self-optimization, self-protection, and self-healing. Each element is also responsible for forming and managing the relationships that it enters into with other autonomic elements in order to accomplish its goals: the *external* autonomic behavior that enables the system as a whole to be self-managing.

The autonomic elements in Unity are implemented as Java™ programs, using the Autonomic Manager Toolset [3]. They communicate with each other using a variety of Web Service interfaces, including both standard OGSA [4] interfaces and additional interfaces that we and other workers have defined for autonomic elements. An important principle of the system is that no other means of communication between the elements is permitted; there are no back doors or undocumented interfaces between the elements. This principle allows us to completely specify the interactions between the elements in terms of the interfaces that they support, and the behaviors that they exhibit through these interfaces.

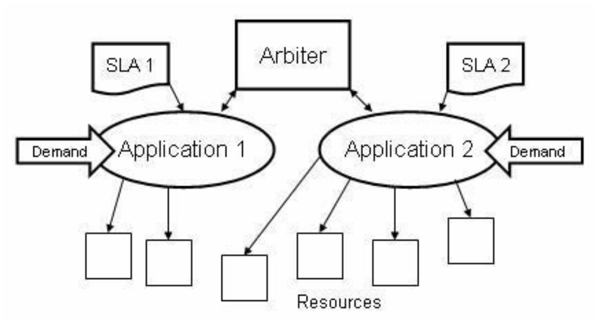


Figure 1. Unity scenario

The IT scenario that the Unity system is currently set up to address involves resource allocation between application environments, as illustrated in Figure 1. A finite pool of resources must be allocated between two or more applications, where each application provides some service for which there is a time-varying level of external demand. The performance of each application depends on the demand being placed on it, and the amount of resources allocated to it. Each application is governed by a Service Level Agreement (SLA), along the lines described in [5], which describes the rewards or penalties associated with various possible behaviors of the system. The overall success of the system depends on the performance of each application relative to the governing SLA.

The various autonomic elements in the system must cooperate in order to optimize the overall system performance relative to the set of SLAs in effect. They do this by discovering resources and forming and maintaining relationships as we will describe, using the defined Web Service interfaces.

3. The components of Unity

As described above, Unity is structured as a set of individual autonomic elements. In this section we will briefly describe each of these elements; later sections will discuss important features of the elements in more detail.

Each application environment in Unity is represented by an **application environment manager** element, which is responsible for the management of the environment, for obtaining the resources that the environment needs to meet its goals, and for communicating with other elements on matters relevant to the management of the environment. One key responsibility of an application manager is to be able to predict how an increase or decrease in the resources allocated to the application environment would impact the environment's ability to meet its goals.

In the current Unity implementation, we have written application environment managers for typical web service requests directed to a set of servers by a workload driver or by IBM's WebSphere Edge Server, for applications parallelized through IBM's Topology Aware Grid Services Scheduler, and for our own test applications.

The **resource arbiter** element is responsible for deciding which resources from the finite pool should be assigned to which application environment. It does this by obtaining from each application environment an estimate of the impact of various possible allocations, and calculating an optimum (or probable optimum) allocation, as described in more detail below.

In the current Unity configuration, the resources being allocated are individual servers. Each server is represented by a **server** element, which is responsible for (among other things) announcing the server's address and capabilities in such a way that possible users of the server can see them.

Each host computer that is capable of supporting autonomic elements is represented by an **OSContainer** element, which accepts requests from other elements to start up certain services, certain further autonomic elements. In the current system, a host computer that is capable both of functioning as an application server and a host for other autonomic elements is represented by both a server element and an OSContainer element; it may eventually turn out to be sensible to merge these two into one.

The **registry** element, based on the Virtual Organization Registry defined in [4], enables each element to locate the other elements with which it needs to communicate, as described below. Its function is analogous to registries in multi-agent systems (see for instance [6]).

The **policy repository** element supports interfaces that allow the human administrators of the system to enter the high-level policies that guide the operation of the system. We will describe utility-function based policies below; other policies control simpler aspects of the system's operation, such as whether a particular server is available for use or reserved for testing.

The **sentinel** element supports interfaces that allow one element to ask the sentinel to monitor the functioning of another. If the monitored element is ever found to be unresponsive, the sentinel notifies the element that requested the monitoring. The sentinel takes part in the self-healing cluster pattern described below.

Finally, the **solution manager** element represents the "solution" as a whole (the entire set of application environments, resources, and so on) to the outside world, and is responsible for any bootstrapping and maintenance issues that apply to the entire solution.

3.1 User interface

In addition to the autonomic elements listed above, Unity also has a user interface that allows an administrator to observe and direct the system. The user interface is a web application consisting of a number of servlets, portlets, and applets, built using IBM's Integrated Solutions Console, an interface framework that is itself built on WebSphere Portal technology. It communicates with the autonomic elements in the system through the usual defined programming interfaces; it has no privileged access to any component. It would therefore be possible to create replacement or alternative user interfaces for Unity without altering any other part of the system.

The Unity user interface allows the user to define high-level policies and utility functions and enter them into the policy repository. It polls the registry and the autonomic elements at regular intervals to obtain current performance values for each application environment, and allows the user to examine the performance of the application environments in the system and the current state of each autonomic element.

Rather than a user interface for any single autonomic element, the Unity UI is a system-wide management interface; if necessary or desirable, it would also be possible to construct user interfaces to specific autonomic elements in the system. One of the goals of Unity is to explore user-interface design patterns in autonomic systems and to study, for instance, the relationship between element-specific user interfaces and broader system interfaces.

4. Utility functions for resource allocation

When the Unity resource arbiter needs to consider changing the current allocation of resources, it queries the known application environment managers. The content of the query is essentially "There are N units of resource that could potentially be allocated to you; for each possible number of units 0 to N, please estimate how well you would do if allocated that many units of the resource".

In order to accurately reply to this query, the application environment manager must have two things: it must have a model of itself that allows it to predict with some accuracy how its behavior and performance would change if it were given various counterfactual amounts of resource, and it must be able to assign a single numerical quantity to the value of that behavior and performance.

The first of these things, the system model, is not a current focus of Unity; we use a relatively simple ad hoc system model most of the time, although we are beginning to experiment with more sophisticated ones.

The second of these things, the assignment of a value to a particular behavior and performance of the application, uses the utility function methodology described in [7]. Using a general utility function to compute the value of the application performance allows us to express a wider range of desired system behaviors than simpler approaches using fixed goals, and additionally allows us to choose between multiple possible system states all of which satisfy the same set of service level targets or agreements.

For instance, if each of two application environments is governed by a simple SLA that specifies a single performance-level goal, then there is no principled way to choose between resource allocations that result in both SLAs being met, or both being violated. In practice, the owner of the system will often have more detailed preferences. For instance if the "customer" for one application is an automated process that will work correctly as long as the minimal SLA goal is met, whereas the customer for the other application is a set of humans doing Web transactions, then if there are two or more possible allocations that are likely to meet both goals, the owner would prefer the one that gives the best possible response time to the human users. This is easy to represent with utility functions; without them, it would likely require special-purpose code in the resource arbiter.

The fact that utility functions are essentially mathematical objects carries additional benefits. When a high-level system policy is expressed in terms of actions to take or specific goals to be achieved, it can be challenging to decompose it into lower-level policies to be used by the components of the system. There may be no natural or automatable way to translate actions or goals at the high level into actions or goals at the next level down. When the higher-level policy is a utility function, however, it may be possible to decompose that function mathematically into utility functions for the lower-level elements which, when appropriately summed, yield the desired utility function at the high level.

5. Goal-driven self-assembly

One of the goals of the autonomic computing vision is self-configuration; autonomic elements should configure themselves, based on the environment in which they find themselves and the high-level tasks to which they have been set, without any detailed human intervention in the form of configuration files or installation dialogs.

Within Unity, we are experimenting with a technique that we call "goal-driven self-assembly". Ideally, each autonomic element, when it first begins to execute, knows only a high-level description of what it is supposed to be doing ("make yourself available as an application server", or "join policy repository cluster 17"), and the contact information (Grid Service Handle) of the registry. In a commercial-grade version of the technique, each element would also be provided with the security credentials needed to prove its identity to the other elements in the system.

When each element initializes, it contacts the registry and issues queries to locate existing elements that are able to supply the services that the new element requires in order to operate. It contacts the elements thus located, and enters into relationships as required to obtain the needed services. Once the element has entered into all the relationships and obtained all the resources that it needs to function, it registers itself in the registry, so that elements that later need the services that it provides can in turn contact it. This process is not confined to initialization time; if an element comes to need a certain service later on in its lifecycle, during operation or termination, it similarly contacts the registry to find available suppliers.

One of the key services that elements locate through the registry is the policy repository. The policy repository contains, in principle, everything that an element needs to know beyond the registry address and its own high-level role. As one of its first actions, a newly-initialized element locates and contacts a policy repository, queries it for the policies governing elements acting in its role, and uses the result of the query to make decisions about further configuration and subsequent operation. In the current Unity implementation, only some of these policies are actually stored in and retrieved from the policy repository; we intend to increase that fraction in the coming year.

Concretely, within Unity, the first elements to start are the OSContainers and the registry, which are necessary to the starting of the other elements. A bootstrap process then starts the resource arbiter, which (acting in its role as solution manager) decides what other elements need to be started and contacts OSContainers (found in the registry) to arrange for their starting. The policy repository and sentinel elements register with the registry immediately upon coming up. The resource arbiter registers with the registry, locates the existing policy repositories and sentinels, and hires a sentinel to watch each policy repository (as described below). Server elements locate and contact the resource arbiter to announce themselves as available for use, and application environment managers contact the arbiter in order to have servers allocated to them. None of the elements knows in advance where the others are located, or even in most cases how many other elements of a given kind will prove to exist.

5.1 Issues in self-assembly

This relatively simple explanation glosses over some potentially complex issues of bootstrapping and circular dependency. Our current system "cheats", in that the resource arbiter acts as a solution manager, contacting OSContainers to bring into being those other elements required by the system. In a more thoroughgoing version of self-assembly, which we hope to achieve in the next year, each element would be responsible for causing the instantiation of any other elements that it requires to function, if none are already available. This would allow for a dynamic and decentralized bootstrapping, more in concert with the autonomic vision. Another interesting approach would be to retain the solution manager function, and define a language for solution recipes which would tell the solution manager which elements (or at least which initial elements) to bring up to start the system operating.

A smaller-scale bootstrapping issue is that when the first OSContainer element comes up, there is not yet a registry running, so it cannot perform the registration steps described above. In our current design, each OSContainer consults its information about where the registry *should* be, and if that address turns out to be the address of a registry that the OSContainer could create, it creates it.

Similarly, no element will be able to contact a policy repository until both a registry and a policy repository have come up; this means that at a minimum both the OSContainers and the registry must be able to function at least temporarily without a policy repository, and in fact all elements should have a minimal set of default policies that suffice at least to get them through the process of waiting for a policy repository to appear, and correctly reporting the error if none ever does.

Circular dependencies, and the registry as an undesirable single point of failure, are described below, under Future Work.

5.2 Steps toward self-assembly

The phrase "self-assembly" in "goal-driven self-assembly" is meant to bring to mind the image of a box of parts, which, when thrown into the air and allowed to fall, spontaneously organize themselves into a computer, or a motorcycle, or a toaster, according to the expressed desires of the thrower. This is a relatively lofty ambition; in the near term, customers may be willing to accept, and the commercially viable technology may support, only a milder form, in which a human operator still specifies the essentials of the system's functions and relationships, and the autonomic aspects of the system are responsible only for self-configuration rather than for full self-assembly. But we consider self-assembly to be the goal, and we anticipate that eventually both customer acceptance and technological maturation will get us there.

6. Self-healing for clusters

As we mention above, one of the goals of Unity is to demonstrate and study self-healing clusters of autonomic elements. For the first version of Unity, we have implemented this style of self-healing in a single element: the policy repository.

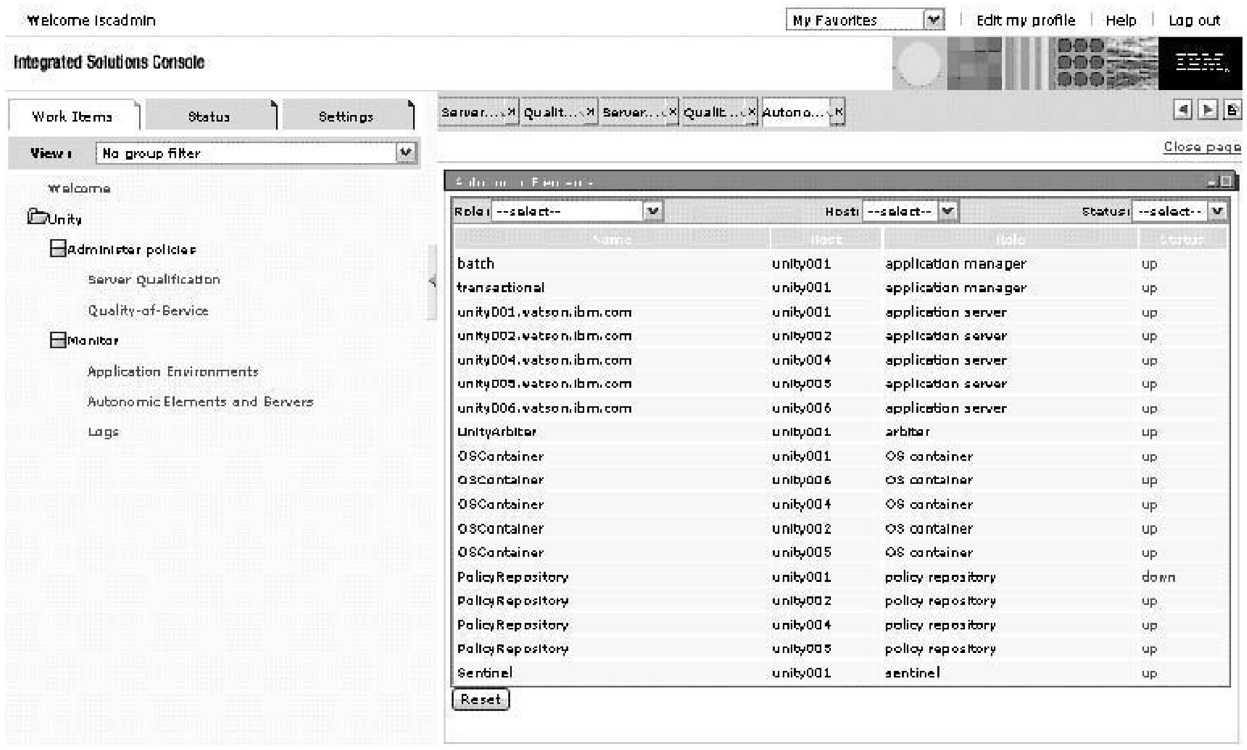
The purpose of a self-healing system is to provide reliability and data integrity in the face of imperfect underlying software and hardware. In order to provide this reliability and integrity, we have added functionality to the policy repository to support joining an existing cluster of synchronized policy repositories, and replicating data changes within that cluster.

It is also necessary for the system as a whole to detect the failure of one of the elements making up a cluster, and to create a new element in order to replace the failed one. Care and consideration must be given to where (that is to say, upon which host machine) this new element should be create—Unity currently assumes, for example, that two elements in the same cluster should not be hosted on the same machine, and that elements in a cluster should not be instantiated on machines that have previously hosted failed elements in that same cluster.

6.1. Policy repository clustering features

In order to support clustering, certain new operations were added to the policy repository element. The first of these changes is the most obvious—whenever a new or modified piece of policy data is received by one of the policy repositories in the cluster, it is sent to all the other repositories in that same cluster. In this way, each policy repository always has a consistent (to within a few seconds) view of the policies. It should be noted that the algorithm currently employed for this process does not have transactional integrity, and race conditions can lead to desynchronization in rare conditions; we intend to address this in the near future, probably either by applying known algorithms for transactional integrity and data synchronization, or by backing the policy repository with a pre-existing product that already features this type of data replication.

Another feature required for this self-healing pattern is less immediately apparent—elements in the Unity system ensure that they are apprised of changes to their policies by subscribing to those policies in the policy repository. In the standard OGSI [8] notification pattern, a single OGSA service (the subscriber) subscribes to a given Service Data Element on a single other OGSA service (the publisher)—in our case, the publisher would be the policy repository. In the event of that policy repository failing, while its data is still safe and available from the other policy repositories in the cluster, the subscriber is left with no subscription, and will never be notified of subsequent policy changes. Consequently, a modified subscription system was created, in which the subscriptions themselves (including the identity of the subscriber, the class of data subscribed to, and the member of the cluster currently responsible for servicing the subscription) are part of the data replicated between elements of the cluster. When a member of the cluster fails, all the subscriptions that it was servicing are still recorded in the state data of the surviving cluster members, and by reassigning those subscriptions to a surviving member, the system can continue providing notifications to the subscribers.



Name	Host	Role	Status
batch	unity001	application manager	up
transactional	unity001	application manager	up
unity001.watson.ibm.com	unity001	application server	up
unity002.watson.ibm.com	unity002	application server	up
unity004.watson.ibm.com	unity004	application server	up
unity005.watson.ibm.com	unity005	application server	up
unity006.watson.ibm.com	unity006	application server	up
UnityArbiter	unity001	arbiter	up
OSContainer	unity001	OS container	up
OSContainer	unity006	OS container	up
OSContainer	unity004	OS container	up
OSContainer	unity002	OS container	up
OSContainer	unity005	OS container	up
PolicyRepository	unity001	policy repository	down
PolicyRepository	unity002	policy repository	up
PolicyRepository	unity004	policy repository	up
PolicyRepository	unity005	policy repository	up
Sentinel	unity001	sentinel	up

Figure 2. Part of the Unity user interface, showing the autonomic elements in the system after one member of the policy-repository cluster has failed and been replaced.

6.2. Sentinel features

The sentinel used in Unity is fairly simple, and is designed explicitly for monitoring OGSA services. When the sentinel is asked to monitor a target service, the sentinel will thereafter periodically read some of the standard (mandatory) Service Data Elements from that target service in order to determine whether or not that target service is still functioning. The sentinel makes this discovered information (whether or not the target is still available) available to the requesting service via Service Data. The requesting service is expected to either subscribe to the Service Data Element in question, or to read it periodically by some other means.

6.3. Creating and using the self-healing cluster

When the Unity system is initialized, the resource arbiter determines how many policy repositories are required (this determination is nominally made by consulting the system policy, but due to the obvious bootstrapping problem this policy is not stored in the policy repository). The resource arbiter then deploys, using the techniques described above, the required number of policy repositories (each on different hosts, as mentioned above). Each one is supplied with the address of the registry, and the role that it is to perform (including the identifier of the cluster that it should join). As each one initializes, it consults the registry to locate and contact the already registered members of the cluster and thereby join the cluster itself, using a simple serial algorithm that avoids most race conditions. The resource arbiter also contracts with the sentinel to monitor these policy repositories, and subscribes to the sentinel in order to be notified of changes to the state of the policy repositories.

From this point, whenever one of the policy repositories receives changes to the set of policies, those changes are communicated, as discussed above, to the other policy repositories in the cluster. Similarly, and also as discussed above, the policy repositories comprising the cluster exchange information about which elements are subscribers to the policy data, and to which policy data those subscribers are subscribed.

Now let us assume that the sentinel determines that one of the policy repositories in the cluster has failed—perhaps the software has suffered a failure, perhaps the network connection has been severed, perhaps the machine has simply ceased to exist. The resource arbiter will be notified (via its subscription to the sentinel) of this failure, and will decide what to do. First, it will choose one of the still-functioning policy repositories to take over the subscriptions previous handled by the failed one, and notify all cluster members of this reassignment of subscription ownership. Then, typically, it will determine that it should replace the failed policy repository—in this case, it will examine the available hosts, and select one upon which to deploy a replacement policy repository (by sending a request to the corresponding OSContainer). The policy repository is so deployed; upon initialization it consults the registry to locate the appropriate cluster, and joins the cluster by the process described above—this process includes the new policy repository receiving a copy of the current cluster state data, including all currently stored policies and subscriptions.

It will be evident that such clusters are not the final word on the subject. For example, the data replication problem is significant; a more complete solution would likely be assisted by the use of the failover and data replication features of a database management system. The method is also most effective in the case of simple single-element failures; it is not robust against network partitions or similar problems. However, even clustering patterns as simple as the one presented here offer benefits beyond failure recovery.

For example, by appropriate manipulations of the resource arbiter's decision-making routines, all the policy repositories in the cluster can be migrated to new hardware and/or software using this system. By introducing the new hardware and software, and then causing each of the legacy policy repositories to terminate in turn, new policy repositories will be created on the new hardware and/or software. This allows for routine maintenance of the underlying operating system and hardware with no interruption in service.

7. Properties of autonomic elements

From our experiences with Unity and our work on the architecture of autonomic systems, we have identified a number of properties that autonomic elements, considered as service providers, must have to enable system self-management. While we expect that our understanding of these properties will grow with further experience, we offer them here as a working draft.

First, each autonomic element must be self-managing—it must be responsible for configuring itself internally, for healing over internal failures where possible, for optimizing its own behavior, and for protecting itself from external probing and attack. This is fundamental to the approach that we use in Unity.

Second, each autonomic element must handle problems locally, where possible. If one of its input services fails to satisfy the agreed-upon SLA, it must solve the problem by requesting resolution from the input service or by finding another, more suitable service.

Third, each autonomic element must be capable of establishing relationships with the other autonomic elements whose services it uses or who use its service, and must abide by the relationships it establishes. As part of this, it must advertise its own service accurately. Otherwise, components like those we use in Unity will be unable to form correct service dependencies.

Fourth, an autonomic element must abide by its policies. It must refuse any proposed relationship that would violate its existing relationships or policies.

Further details, as well as behaviors that are recommended but not required, are available in [2].

8. Future work

Many of the features that we have implemented once, or for a single purpose, in the current Unity system could be usefully generalized. We currently support a small number of application environments; we plan to expand that number, and learn what extensions to the existing interfaces will be required by that wider range.

The Unity components currently self-assemble into only one overall system; we plan to add flexibility to the system so that the box of parts can come down to form various different useful wholes, closer to the ultimate dynamic vision of self-assembly. That ultimate vision will also require standard languages and taxonomies for services offered, dependencies, registry queries, and so on. We would like to evaluate other potential registry models (such as the UDDI model) for their suitability to autonomic systems. It would also be interesting to develop ways to do hypothetical self-assembly, so that the box of parts could be asked "if I were to toss you into the air and ask for an automobile, what would the result be like?". There are interesting issues in self-assembly in complex environments that may involve circular dependencies; avoiding deadlock during self-configuration will be important.

The self-healing cluster pattern that we currently use to increase the reliability of the policy repository should be able to accomplish the same goal for the other potential single points of failure in the system; the resource arbiter, for instance, or the registry. It should be noted that making the registry into a self-healing cluster will require some new invention to avoid the bootstrapping problems inherent therein.

Utility functions are a powerful and flexible way to allow systems to manage themselves. We plan to extend the use of utility functions in Unity from resource allocation to the rest of the system. The self-assembly process, for instance, could use utility functions to decide between various alternate configurations of the system. For instance, an element that could potentially form a relationship with multiple other elements to acquire a needed service could use a utility function to decide which relationships to actually form. System properties like the sizes of self-healing clusters could be derived from higher-level goals (in terms of estimated reliability, say), rather than specified directly by policy. Behaviors, such as bringing up each member of a self-healing cluster on a different host system, could similarly be derived from higher-level principles rather than hardcoded into the algorithms.

Because utility functions are so powerful and general, there are challenges in designing user interfaces that give human users and administrators useful information about them and intuitive control over them. The typical user should probably not be given the ability to sketch an arbitrary utility curve, or be expected to determine which of several possible curve shapes correctly express the value of various outcomes. Existing work on preference elicitation, such as [9], could be usefully applied to the problem of determining the right utility function in an autonomic system.

Similarly, the space of possible policies and utility functions is potentially very large, and users may need the ability to explore, with whatever degree of accuracy is possible, the likely effects of policy changes before those changes are actually made. We are working with other researchers on advanced policy and utility function tooling that would allow this sort of exploration.

Finally, we plan to replace some of the ad hoc algorithms in Unity with more robust methods. The optimization algorithm that we use in the resource arbiter, for instance, currently assumes that switching costs are zero: that moving a resource from one application to another is free. This assumption is valid only in some environments; we plan to explore more powerful algorithms that can deal with non-zero switching costs. And as noted above, the algorithms that we use for state synchronization between members of a self-healing cluster are not robust against various race conditions, and do not have transactional integrity; we plan to replace them with algorithms that do.

Unity has been a valuable platform for studying and validating our ideas about autonomic systems. We intend to expand its scope to include a wider range of functions and products, and to illuminate more of the large and interesting space of self-managing systems.

10. References

Java is a trademark of Sun Microsystems, Inc.

[1] Jeffrey O. Kephart, David M. Chess, “The Vision of Autonomic Computing”, IEEE Computer 36(1): 41-50 (2003)

[2] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart, “An Architectural Approach to Autonomic Computing,” submitted to International Conference on Autonomic Computing (ICAC-04), 2004.

[3] David W. Levine et al., “A Toolkit for Autonomic Computing”, IBM Developerworks Live, 2003.

[4] J. Nick I. Foster, C. Kesselman and S. Tuecke, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,” Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[5] Avraham Leff, James T. Rayfield, Daniel Dias, “Meeting Service Level Agreements In a Commercial Grid,” IEEE Internet Computing, July/August, 2003.

[6] E. H. Durfee,D. L. Kiskis, and W.P. Birmingham, "The Agent Architecture of the University of Michigan Digital Library", IEE/British Computer Society Proceedings on Software Engineering (Special Issue on Intelligent Agents) 144(1), February 1997.

[7] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das, “Utility Functions in Autonomic Systems,” submitted to International Conference on Autonomic Computing (ICAC-04), 2004.

[8] Open Grid Services Infrastructure (OGSI) Version 1 at http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf

[9] V. S. Iyengar, J. Lee, and M. Campbell, “Q-Eval: Evaluating Multiple Attribute Items Using Queries,” Proceedings of the ACM Conference on Electronic Commerce EC'01, October 2001.