An architecture for the coordination of system management services

by V. K. Naik
A. Mohindra
D. F. Bantz

Today, system management services are implemented by dedicated subsystems, built using proprietary system management components. These subsystems are customized to automate their operations to the extent feasible. This model has been successful in dedicated enterprise environments, but there are opportunities to broaden the scope of these services to multicustomer utility computing environments while reducing the costs of providing these services. A new model suitable for utility computing is being developed to address these opportunities. This model features several new elements: (1) a repository to represent the state of the remotely managed components and of the services that manage them, (2) a repository of policies and operational constraints, and (3) a set of meta-management services that use existing management services to analyze, construct, and safely execute a complex set of management tasks on remote systems. The meta-management services manage the system management services provided by the utility—they guide and modify the behavior of the services, often as a result of the collective analysis of the state of one or more services. In this paper, we describe requirements and behaviors of such meta-management services and the architecture to provide them. We focus on the components of this architecture that enable and provide effective meta-management services in a utility environment.

In an enterprise, a large fraction of the budget spent on information technology (IT) is associated with providing system management services. The primary objective of system management services is the continual operation of servers, desktops, and laptops in the enterprise. Typical of the services provided by system management are deployment services (initial configuration, software distribution, and installation), support services (help-desk support and troubleshooting), preventive maintenance services (upgrades, backups, and virus scanning), and other administrative services (asset management, user management, and license management). Today these services are typically provided by the IT staff.

System management tasks are complex, error-prone, and training- and labor-intensive. In current practice, a system administrator invokes services manually through the facilities of one or more consoles. Typically, each console controls one service, and the system administrator is responsible for mentally mapping the desired actions into the specific service invocation syntax. Multistep service invocations are often stored as scripts whose sequence is fixed but whose parameters can be supplied at run time. Taken together, these scripts represent an enterprise-specific integration of system management components. These scripts must be constructed from scratch for each enterprise in low-level programming languages,

©Copyright 2004 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8670/04/$5.00 © 2004 IBM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

and they constitute a significant management and maintenance task in themselves.

The increasing complexity and integration of enterprise IT is reflected in the increasing complexity of system management. In order to reduce the total cost of ownership associated with system management, enterprises have begun to standardize software applications and streamline IT processes used in their enterprise.

The next step in streamlining the IT infrastructure is to deliver the standard set of services and processes through a shared infrastructure with a consolidated help desk and IT staff, or to outsource the management of the IT infrastructure to other companies that specialize in IT outsourcing. We define a *system management utility* as delivering IT services to multiple customers over the Internet from a shared infrastructure. (In the rest of the paper, when there is no ambiguity, we use the term utility to mean system management utility.) The goal of such a utility is to deliver IT services at reduced cost through standardization, automation, and leveraging economics of scale.

Effective automation in service deployment, delivery, and problem determination requires a high degree of coordination among services and the objects managed by the services. We observe that the current approaches to system management are not designed with coordination and automation as requirements as follows:

1. Services maintain information in silos and do not automatically take advantage of information gathered by other services. This situation makes any multiservice use of system management services IT administrator-centric. To perform any system management task, the system administrator has to analyze the requirements, plan an approach, execute steps, and monitor success across services.
2. System management tools and services do not automatically customize themselves to the configurations being managed. As a result, the configuration, deployment, and coordination of system management services is labor-intensive.
3. System management services and applications are enterprise-centric: most services and applications treat trust and privacy issues from an enterprise perspective.
4. System management services are configured with certain built-in assumptions about the managed systems. They are not configured to adapt to a changing environment. For example, broken dependencies or unsatisfied prerequisites are the main reason when management procedures do not complete their execution normally.

In summary, the existing services and practices do not lend themselves easily to the utility computing model. Although there is a clear need for delivering system management services using the utility model, existing system management services are not sufficient for realizing this goal.

To overcome these difficulties, today's system management services could be redesigned and re-implemented, or new technologies could be developed that would allow the use of existing services while enabling automation by some other means. We take the latter approach here and describe a method and an architecture for developing an infrastructure that is highly automated and scalable, yet adaptive to each specific managed element in its unique context. With this approach, system administrators are relieved from the burden of performing repetitious tasks and are able to focus more on complex tasks such as solution architecture and planning.

Several key contributions are presented in this paper. We describe how to use a rule-based system to model policies, profiles, and dependencies or requirements. We also describe an architecture in which a repository of rule-based objects is used for automated configuration and sequencing of multiple services. This architecture also handles management tasks related to spontaneous changes in remote systems. The organization of the objects in the repository makes it possible to reason about actions to take and which services to deploy.

In the next section of this paper, we describe objectives and requirements of a system management utility and discuss the shortcomings of existing solutions to meet these objectives and requirements. We conclude the section by describing some of the salient points of our approach. In the third section, we introduce rules and discuss how they can be used to represent policies, profiles, and requirements. The architecture of our approach is discussed in the fourth section. We present some design issues after that, and in the sixth section we describe an outline of a prototype implementation of this approach. Other work in the literature related to this work is discussed in the seventh section. Finally we conclude with a summary and some directions for future work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Delivering system management services as a utility

In this section we first discuss the objectives and requirements of a utility. We then discuss the shortcomings of current system management tools and services, and following that we describe intuitively the key ingredients needed to realize the objectives of the utility model.

Objectives and requirements of a utility. In the introductory section, we briefly described the objectives and principles used in managing systems. To offer system management services as a utility, some additional considerations must be taken into account. By its nature, a utility has to cater to a diverse set of customers, each customer having its own unique requirements. At the same time, to be successful as a viable business, the utility cannot assign expert system administrators individually to each customer. It must replicate its services economically and be able to customize them before deployment on customer systems. In particular, a system management utility has the following three key objectives:

1. Provide policy-based system management and service administration
2. Provision services to customers on demand and according to service level agreements (SLAs) with the customers
3. Leverage economies of scale

Policy-based system management means adjusting the service behavior according to some customizable policy. The policy may be customer-, user-, or machine-specific. It can also be a policy specified by the utility. An example of a policy is performing incremental backup on a daily basis and a full backup on a monthly basis. At the machine or user level, the policy might specify the exact time of the day when the backup is to be performed.

"On demand" service provisioning means being able to configure and deliver a service just in time when the need arises. "SLA-driven" means that the quality of service is maintained at a certain prescribed level. An example of an on demand service is performing a nonroutine backup just before upgrading the operating system of a machine. An example of an SLA-driven service is maintaining a particular machine configuration in a usable state 99 percent of the time during normal working hours over a specified period of time.

To satisfy the first two objectives just described, for each kind of system management service, a utility has to manage multiple types of system management services (i.e., similar service functionality offered by different brands, versions, etc.) and needs to be capable of configuring and deploying different variants of a particular service, depending on the situation at hand.

The third objective is driven mostly by pragmatic considerations. Leveraging economies of scale means being able to amortize costs efficiently over a large customer base. In other words, costs associated with system management services should grow sublinearly as a function of the number of customers, machines, and users.

Clearly, for a utility to be successful as a business concept, large-scale automation is required to achieve the three key objectives listed above. Large-scale automation in configuring and coordinating services is necessary to provide system management services from a utility.

State-of-the-art in system management technology. The Microsoft Systems Management Server (SMS)[1] and the Tivoli Configuration Manager (TCM)[2] are two widely used applications for managing systems. These applications provide mechanisms that facilitate the performing of certain management tasks by an IT administrator. Management applications typically provide mechanisms for gathering hardware and software inventory and performing software installation and distribution, license management, troubleshooting, and so on. These tools enable an IT administrator to perform management tasks from a central console. However, the management applications do not alleviate the need for an IT administrator to spend time thinking and developing steps needed to accomplish the task at hand. For instance, if an IT administrator needs to deploy a new version of software, then he or she has to develop an action plan that takes into account hardware and software prerequisites, best practices, and old versions of the software that are deployed. The seemingly simple task of deploying software translates into several interdependent steps that may vary, depending upon the configuration of each managed object.

For example, the task of distributing a software package to an endpoint (i.e., a managed object) using TCM requires execution of a 65-line shell script given in Reference 3. Before executing the script, it is as-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

sumed that the endpoint has been recognized by the TCM console. Additional scripts need to be executed to discover and integrate an endpoint into TCM. The software distribution script consists of several subtasks: checking for the software profile manager (**wlookup** command), subscribing the endpoint to the software profile manager (**wsub** command), and distributing the software package to the endpoint (**winstp** command). Successful execution of this last step implies that the software package is successfully delivered to an agent running on the managed object.

The script assumes that the prerequisites and other dependencies for installing the software package are met. When the script is to be applied on a mass scale, this assumption has to be verified for each and every instance before applying the script; otherwise, the software installation may not be completed successfully. However, such a verification test may depend on the type of the managed object. Thus, multiple customized verification scripts may have to be generated. The problem becomes more difficult if a set of post-installation tests are required to be performed remotely to ensure that the installation was successful or if the installed software is required not to interfere with other software installed on that system. And typically, these scripts control a single service. They do not coordinate and integrate multiple services to orchestrate them to achieve a common goal.

The preceding shortcomings are not shortcomings of TCM, but of the way in which the facilities of TCM are invoked. Similar difficulties exist when trying to accomplish the same task in SMS by using the mechanisms and wizards provided by SMS. Many products are available that offer highly specialized functions such as inventory gathering, software distribution, or event monitoring. Many IT administrators still rely on highly customized shell scripts written in scripting languages such as WSH (Windows** Script Host) and Perl to perform system management tasks that have been somewhat tuned to their environment.

Need for a new approach. As described earlier, most of the system management services available today are designed and developed in isolation. Given this state-of-the-art for the system management services, a policy-based and on demand system management utility would imply generation of one-time scripts on a case-by-case basis. Such a situation is hardly conducive to large-scale automation. From a practical point of view, it is much more desirable and attractive to automate interactions among components that can be introspected, examined for state changes, and associated with actions that can be triggered by using semantic rules. In the context of system management services, this means being able to model the managed objects and the services that manage them. These models should predict the need for a particular change in the state of the managed object and what action should be triggered to bring about that change. By associating the action with a service, a desired level of automation can be accomplished.

Our approach. We have formulated our approach by focusing on the needs described above. In particular, we have focused on a system in which the causes of changes in the state of a managed object and effects of those changes on the environment can be reasoned about. We also focus on associating actions with services so that the available services can be coordinated once the actions to be taken are determined. When desired changes are known *a priori*, this approach allows us to build action plans by analyzing the current state and known effects of available change management services.

Specifically, we model a managed object by caching selected components of its state together with constraints on that state, and we formulate rules that specify what actions to take when constraints are violated. We also model policies, profiles, and service behavior. Whenever a change is sought in a managed object, we create an action plan by using the analysis and service models. We then predict the effect of an action associated with a service on the modeled object. With this we derive a predicted state and steer the state of the actual managed object toward the desired state in a controlled manner. In this approach, effects of unanticipated behavior are managed by tracking the changes incrementally using a feedback mechanism. Moreover, whenever the system detects uncertainties, it proportionately increases the level of monitoring for feedback and analysis purposes.

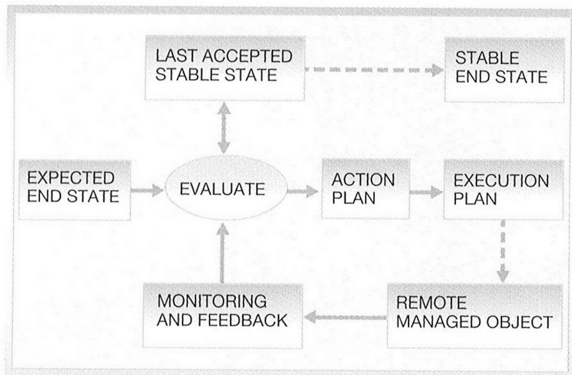
We illustrate our approach schematically in Figure 1, which depicts controlled change brought about in a remotely managed object.

Using rules

We seek a management system whose behavior can be altered by system administrators without changing the mechanisms of the management system itself. We refer to this behavior as *externally* specified,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Figure 1 Realizing a controlled change in a remote managed object



because it varies from deployment to deployment and from time to time. At the same time, the system should have some core behaviors ensuring that for any externally specified behavior the system will behave in a consistent, predictable way. We refer to this system behavior as *internally* specified, because the designers and implementers of the system make the determination of such behavior. We use rules to implement both externally and internally specified behavior. In other words, the system obeys internal rules that are not visible to its users and that maintain its consistency. The system also obeys external rules, given by its users, to alter its behavior to implement the needs of a specific deployment.

Damianou et al.[4] define a policy as "... a rule that defines a choice in the behavior of a system." Their use of the term "policy" is in accord with common usage, in which the term is used in connection with externally specified behavior. They identify several classes of policies: authorizations (you may), refrainment (you may not), and obligations (you must). Coercive policies (refrainments and obligations) are commonly referred to as "constraints." In our system, behavior-determining rules are pervasive. In order to avoid confusion, we will refer to those rules that are externally specified as implementing policies and to internal rules as implementing mechanisms. This distinction is not material, however. In both cases, rules control behavior. In fact, we think of internally specified behavior as obeying policies that are determined by the designers and implementors of the system.

In accordance with the dual role that rules play in our system, we have chosen an implementation of rules appropriate both to the external specifier and to the system designer. Rules are implemented by the Agent Building and Learning Environment (ABLE[5]). The rule language of ABLE compiles into JavaBeans**.

The following is an example of (externally specified) policy:

```
void process() using Script {
  {weekend} : invokeRuleBlock
              ("weekendPolicy");}

void weekendPolicy() using Forward {
  R1 [2]: IF (memorialDayWeekend)
              THEN {/* special case */};
  R2 [2]: IF (fourthOfJulyWeekend)
              THEN {/* special case */};
  R3 [1]: IF (!crunchTime)
              THEN {/* general case */}
  ELSE {/* special case for overworked
              folks */};
}
```

Here we see the `process()` method invoking a `Script` inference engine on a rule that is preconditioned on a Boolean variable, `weekend`, set elsewhere to indicate that the current time and day are in a weekend time period. The action of this rule is to invoke a *ruleblock* (a set of rules) whose name is `weekendPolicy`. The ruleblock is written nonprocedurally as three rules (`R1`, `R2`, and `R3`). Each rule is followed by optional numerical priority specified in square brackets. In the above example, rules `R1` and `R2` have higher priority (priority 2) than rule `R3` (priority 1). The last rule, whose condition would be very enterprise-specific, provides for an escape from normal policy in times of crisis.

Mechanisms are typically procedural, but the expression of mechanisms as rules has some important advantages even in the procedural case. Rules facilitate experimentation, tuning, and even dynamic modification. Rather than anticipate all possible eventualities by constructing an elaborate rules library, it may be advantageous to alter internal behavior temporarily by modifying existing rules. Not all mechanisms are best expressed procedurally. A (trivially) simple expression of root cause analysis is illustrated in the following nonprocedural set of rules:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
www.manaraa.com

```
void process(y) using Predicate {
  : setControlParameter(ARL.Goal,
             suspect(X));
  : cause(powerOut, power).
  : cause(cableUnPlugged, cable).
  : panel(power, no).
  : panel(cable, yes).
  : suspect(X) :- cause(X,Z),panel(Z,y).
}
```

This ruleblock uses the predicate inferencing engine to solve for X under the parameter y, which gives the state of the lights. The last rule says that the state of the lights is determined by a panel and that the cause is related to the panel. The four rules immediately preceding the last relate a cause to a panel and a panel to the state of the lights. For the parameter y = "no," the goal is reached when the local variable Z selects the power panel.

Architecture for autonomous system management

We now describe our architecture that addresses the utility requirements by using a structured application of the rule-based approach. First we discuss the forces that drive typical system management action sequences. With this discussion in the background, we present an intuitive description of the architecture, and following that we present a more formal description of the logical architecture. We then describe the salient features of the key components in some detail and the control flow among these logical components.

Architectural considerations. In a system management utility, three types of forces drive the services:

1. Component requirements
2. Computer and user profiles
3. Utility and customer policies

We consider these forces as the three dimensions of the utility. Each dimension gives rise to constraints that are not to be violated.

Any component, whether it is a hardware or a software component, has some inherent requirements that need to be satisfied in order for that component to be installed and function properly. For example, a CD-ROM (compact disk-read only memory) device requires a driver to function properly, and the driver may have a dependency on a particular version of a component in the operating system running on the computer. The component constraints arise from the design choices made by the component developer and are independent of the utility and customer policies or the computer and user profiles. We model these dependencies and requirements using special constructs of the rules.

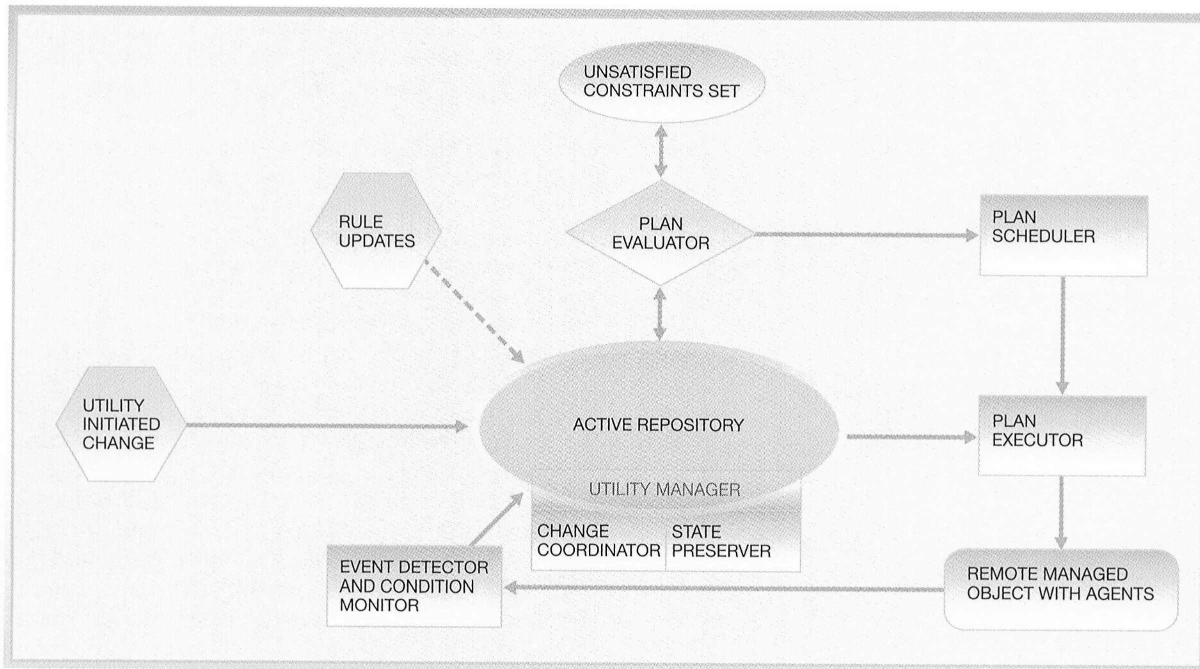
Computer and user profiles dictate how the installed components are to be configured, the look and feel as well as the behavior of the environment and of the components in the environment. These profiles give rise to constraints that need to be taken into account whenever a managed system is to be configured or reconfigured. We model these profile-specific requirements using rules.

Similarly, utility and customer-specific policies give rise to constraints, and these policies are also modeled using rules. In general, policies may be associated with every aspect of the utility. Policies may govern hardware, software, and operational aspects of the utility. An example of a hardware-related policy is that all desktops in a system should have a minimum of 512 MB (megabytes) of installed memory. An example of the software-related policy is that all desktops should have CAD (Computer Aided Design) software version 4.0 installed. An example of an operations-related policy is that each system should have a daily backup performed at 11 PM. Just as in an enterprise, policies allow IT administrators to maintain consistency and order across the organization, and policies permit the utility to plan and execute actions to keep the managed systems consistent with the policies.

Note that the constraints along each dimension can be stated independent of those in the other two dimensions. However, when a new component is to be installed in an existing environment, the constraints relate to one another and affect the state of the user or computer environment. Violations of any of the constraints can leave the environment in an inconsistent state (e.g., an existing or the newly installed component may not function or perform as expected). In fact, one of the main concerns of system management services is to eliminate conflicts and satisfy requirements. However, this concern requires a judicious choice of which services to apply, the order in which they are to be applied, and the appropriate service configurations to use. System administrators tend to spend most of their time figuring out the service configurations and the sequence in which to apply those services for each instance of a system they have to manage.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Figure 2 Logical architecture diagram for the system management utility



The architecture defined here uses rules to model constraints to plan, analyze, and execute the system management services in such a manner that the situations leading to constraint violations are avoided. With our approach, if constraints are violated, then they can be detected, and recovery actions can be planned, analyzed, and executed with a quantifiable degree of confidence. More formally, our architecture consists of a collection of active repositories (data and constraints) and modules for monitoring, analyzing, planning, evaluation, and execution with knowledge (MAPE-K). Although at a high level, the architecture has the MAPE-K structure that is common to many autonomic systems,[6,7] it extends many of the concepts to bring about cooperation among heterogeneous system management services that are not explicitly designed to cooperate with one another. This is brought about by using the active repositories and their interactions with a plan evaluator. The repositories are made active by embedding them with self-triggering rules described earlier. The rules may trigger their actions in response to externally monitored events or in response to actions of other rules.

In the following subsections, we first describe the logical architecture and then discuss the active repository, control flow, and management of changes in some detail.

Logical architecture. Shown in Figure 2 is the logical architecture diagram for the system management utility.

The Managed Object (MO) shown in the lower right corner in Figure 2 is a stand-alone system or a component running on top of another managed object, and is capable of being monitored and altered by a management service. System management services are launched by agents that run on the same platform as the MO or from another system capable of controlling the MO. Events that affect the state of the MO are detected and monitored by the Event Detector component. These events may be further filtered, based on one or more conditions and criteria.

When events and conditions of interest are observed, they are recorded into a repository of active objects. Each active object contains a rule that triggers one or more actions whenever a monitored condition is met. A rule-based action can trigger one or more events that may activate other objects in the repository, leading to more actions. In Figure 2, the re-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com
المنارة للاستشارات

pository is shown as a single logical component. An Active Repository is actually a collection of three repositories: a Component Repository, a Profile Repository, and a Policy Repository. For the sake of brevity, we refer to the entire collection of active objects and the associated mechanisms as the *Active Repository*.

A Plan Evaluator monitors the actions generated by the Active Repository. The Plan Evaluator collects actions associated with an MO and analyzes them to determine whether the actions are consistent with one another. It also performs an analysis to determine whether the application of the actions would leave the MO in an inconsistent state. Whenever it detects actions that are inconsistent with one another or whenever some actions lead to an inconsistent state, it adds the associated constraints to the Unsatisfied Constraints Set. It then tries to resolve the conflicts and unsatisfied constraints by injecting preconditions and actions and then re-evaluates the resulting plan. It iterates this process until all conflicts are resolved and the Unsatisfied Constraints Set is empty. If the iterative process does not terminate after a certain number of iterations, human intervention is sought.

The Active Repository and the Plan Evaluator have complementary roles. Together these two components make it possible to analyze and reason about an action before applying it to an MO. Undesirable results can be avoided by modifying or removing some actions in the plan before they are applied. Additional actions can be added to the plan to avoid the undesirable side effects of other actions in the plan. The analysis is carried out until a desired degree of confidence in the outcome is achieved. If, for some reason, the iterative analysis does not achieve a desired degree of confidence in a plan, the Plan Evaluator can apply the plan with fine-grained monitoring and feedback turned on so that the plan can be reversed or altered while being applied to the MO.

When the Plan Evaluator generates a plan that is safe to execute, the plan is handed over to a Plan Scheduler. The Plan Scheduler examines the plan to see whether the plan can be optimized for performance (e.g., reducing the number of reboots where possible) and determines which actions can be performed in parallel and which actions must be serialized. It also adds monitoring and feedback commands to the plan to achieve a certain desired degree of confidence in the plan.

A plan that is ready to execute is handed over to the Plan Executor. For each action in the plan, the Plan Executor either invokes an appropriate system management service or performs the action by executing corresponding system commands. The Plan Executor also primes the appropriate event monitoring agents before applying any actions. In this way, the desired state attributes and conditions are monitored and fed back to the utility. When intermediate steps of the plan execution require synchronization with the utility, the Plan Executor waits for signals from the monitoring agents or from the Active Repository before proceeding to the next step.

We note here that the ultimate plan executed by the Plan Executor may not be exactly the same as the one initially specified by the Plan Evaluator or the Plan Scheduler. The reasons are the Plan Evaluator may have generated a plan requiring intermediate feedback and the Plan Scheduler may have inserted steps requiring synchronization of the MO state information with that represented in the utility. In either case, during the course of plan execution, new events and conditions may be fed back to the utility, which may lead to reevaluation and alteration of the plan.

The Utility Manager coordinates the Active Repository and the other components of the utility. Because these components operate asynchronously and because the state of an MO may change spontaneously, events can arrive in any order. The irregular arrivals can lead to deadlocks or livelocks (i.e., one event triggering another event and the second event in turn triggering the first one, and so on). To avoid this without imposing strict synchronization requirements, coordination at a higher level is required. This function is performed by the Utility Manager. The Utility Manager has two subcomponents: the State Preserver and the Change Coordinator. The role of the State Preserver is to protect the state of an MO from spontaneous changes (i.e., changes not initiated by the utility). If the state changes spontaneously, then the State Preserver tries to ensure that no constraints are violated. If any constraints are violated, it coordinates the activities of the Plan Evaluator, the Plan Scheduler, and the Plan Executor to modify the state of the MO so that constraint violations are eliminated. The Change Coordinator, in contrast, facilitates changes in an MO state originated by the utility. The Utility Manager monitors the events entering the Active Repository. The Change Coordinator handles expected events, whereas unexpected events are handled by the State Preserver.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

We now describe the Active Repository in some detail because it is one of the key components of the architecture.

Active Repository. As stated earlier, the architecture of the Active Repository consists of three basic repositories: the Component Repository, the Profile Repository, and the Policy Repository.

The Component Repository contains a representation of all components—both hardware and software—that may be used in any of the managed systems. Associated with each component are rules representing requirements, dependencies, and knowledge-base assertions for each stage in their life cycle. The rules are updated along with the requirements and the knowledge base during the lifetime of a component. New components are added to the repository as the utility expands its support.

The Profile Repository contains customer and user profiles. Typically, each customer has a profile. This profile is inherited by the users belonging to that customer. Individual users may extend or override some of the attributes in the customer profile. In addition to specifying the look and feel and the expected behavior of the computing environment, a user profile may also specify SLAs, such as the maximum number of outages or the acceptable maximum duration of an outage during which a certain basic configuration may not be available to the user. If an SLA is violated, a penalty may be imposed on the service provider. Such penalties are also specified in the profile. Specifications in the profile give rise to constraints. These constraints are translated into rules that indicate how a particular machine or a user's computing environment is to be managed and how to prioritize available alternatives. The rules govern the manner in which system management services are to be applied during normal operations as well as whenever a change is to be applied.

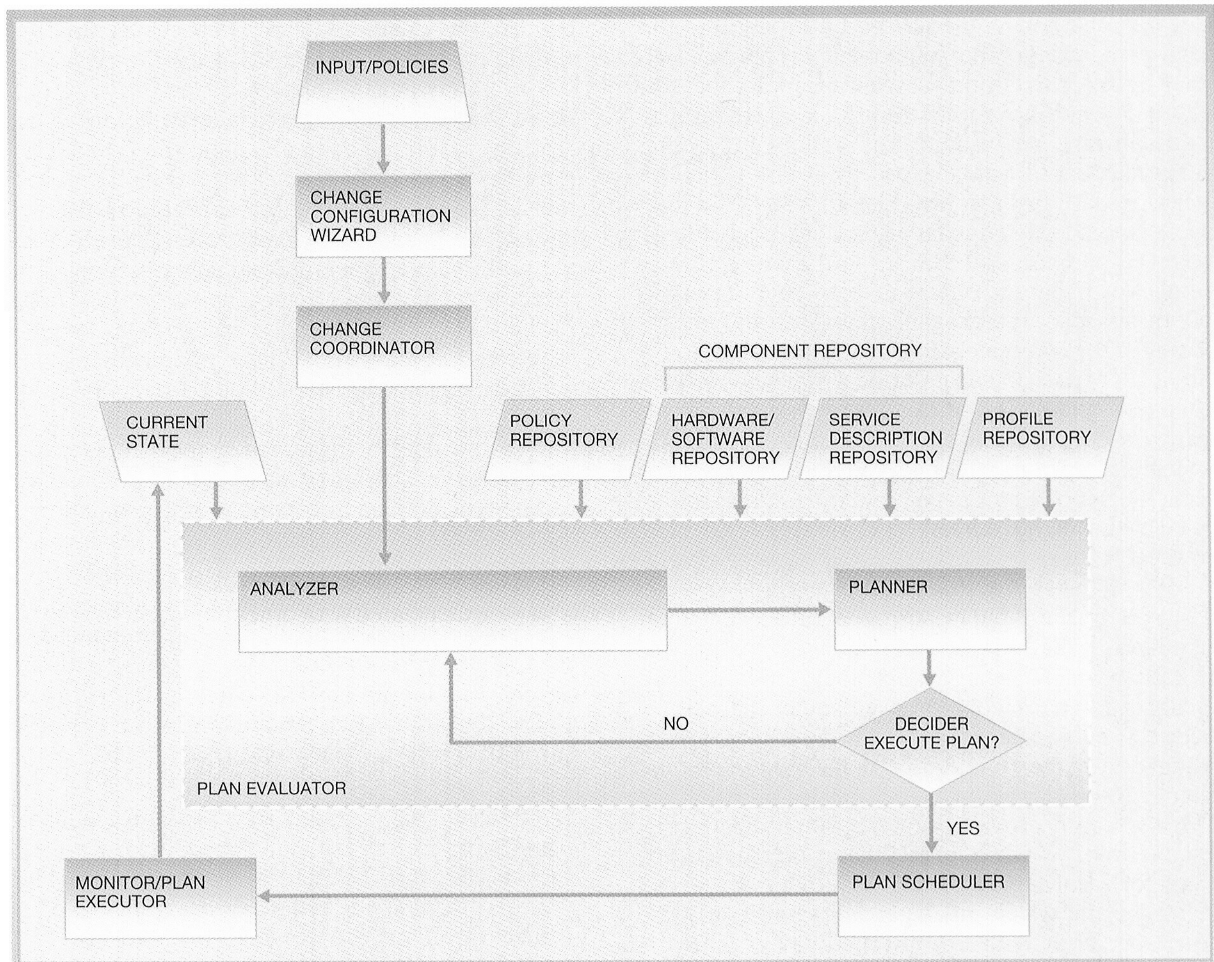
The Policy Repository contains representations of customer-specific policies as well as utility-wide policies. One example of a customer policy is how often a new version of a software package is to be rolled out to its users. An example of a utility-wide policy is denial of support for certain types of software packages or certain system configurations. As in the case of the other two repositories, these policies give rise to constraints that are represented as rules indicating how a particular machine or a user environment is to be managed.

The organization and structure of the Active Repository is such that no explicit and monolithic state representation of the managed objects is necessary. When a new component is to be installed on a managed object, that component is first modeled in the Active Repository by creating an instance object (i.e., its presence in the environment of the MO is simulated). The instance is created by using a template in the Component Repository for that component with built-in constraints. Before actually installing the component, information is gathered to validate whether the associated constraints can be satisfied. If necessary and possible, the state of the MO is adjusted to satisfy the constraints. The component is installed after verifying that all the associated constraints can be satisfied. Conditions satisfying the constraints are recorded in the object representing the component in the Component Repository. The constraints may be revalidated periodically by explicit probing or by inferring from routinely monitored events. Whenever validations fail, constraint violations are raised. The objects modeling the components are associated with other objects in the Active Repository, including the objects representing policies, profiles, and other components.

Control flow. We explain the control flow in the utility architecture by using an example in which a utility-originated change is brought about in an MO. An example of such a change is a new policy for a customer environment that mandates use of a new CAD software package for all employees in the customer organization. The net effect of this policy change would be for the utility to initiate distribution and installation of the new CAD package, ensuring the stability of each machine after the change. Figure 3 shows the schematic of the corresponding control flow. As mentioned earlier, in the Component Repository the utility maintains the constraints for the hardware and software components and the actual state information. In addition, it maintains information on administrative and maintenance services to be applied on each MO and the frequency with which those services are to be applied. The associated rules trigger application of the services. When an administrative or maintenance service is applied, the resulting state change is reflected back to the Component Repository. Thus, for example, from the information stored in the Component Repository, it is straightforward to determine when a file system on a particular machine was backed up or when the next virus scan is scheduled to be run on a machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com
المنارة للاستشارات

Figure 3 Control flow schematic for the systems management utility



A Change Configuration Wizard is provided to enable utility administrators to inject desired changes to either an individual system or to an entire customer environment. The Change Configuration Wizard interfaces with the Change Coordinator subcomponent of the Utility Manager which, in turn, interfaces with the other components in the utility. The Change Configuration Wizard verifies the availability of the relevant software packages, bug fixes, and so forth. It then initiates the change by forwarding the information to the Change Coordinator. The Change Coordinator takes a snapshot of the context of the relevant portion of the Active Repository and injects an object representing the change. In the case of the CAD software package, for example, the Change Coordinator instantiates an object to represent the CAD package and the associated constraints governing its successful installation and operation. It also updates the Profile Repository by adding a new constraint to the profile to trigger the installation of the CAD package.

The Change Coordinator then submits the newly created context, including the injected object, to the Plan Evaluator. This context represents a model of the system as it would be with the intended change in place.

The Plan Evaluator has three components: the Analyzer, the Planner, and the Decider. The Analyzer is responsible for analyzing the effects of policies and profiles governing the environment where a change

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

is to be brought about. It is also concerned with the effects of the change on the state and configurations of other managed objects in the same environment. For this, the Analyzer uses the information in the relevant portions of the Component, Profile, and Policy Repositories. In our current example, the Analyzer determines whether the installation of the new CAD software may require uninstalling an existing version, updating a database software component to a new release prior to installing the new CAD software, and so on. Thus, a single high-level change could result in several changes to the current state, some acceptable and others unacceptable. An example of an unacceptable state change would be updating of the component to a new release that would break an existing business-critical application. This change is a violation of the existing policy that all critical software must be installed and installed software must be properly configured. The Analyzer collects the set of changes and violations that are implied by the intended change and then forwards them to the Planner.

The role of the Planner is to generate a sequence of actions, called the Action Plan, to overcome the violations or to bring about the required state changes. For each violation, the Planner suggests an action to overcome the violation. To arrive at such a suggestion, the Planner may make use of domain-specific knowledge databases. Constraints in the Active Repository provide priority information that the Planner uses as advice in evaluating alternative actions for resolving violations. The Planner also assigns specific system management services to perform specific actions. For this the Planner relies on the information in the Services Repository, which is a special subrepository of the Component Repository. The Services Repository contains descriptions of services that are capable of performing actions in the Action Plan. For instance, if the Action Plan requires that a backup of the system be done prior to installing a new software package, the Planner selects a service that is most suitable for performing a backup operation on a particular system. The suitability of a service may be determined by other constraints in the Services Repository such as a policy constraint or a profile-related constraint.

As the Planner creates an Action Plan, specific services are tentatively bound to actions. These services may have their own dependency and configuration requirements. Thus, although the Planner has a plan to resolve the initial set of violations, the specific actions chosen by the Planner need to be analyzed again for policy, profile, or dependency and requirements violations. For example, invocation of a certain service may require installation of a service client. But this client installation may be incompatible with other software on that MO. Thus, the plan needs to be analyzed again to see whether it can be executed without generating any new violations. After a plan is generated by the Planner, the Decider determines whether the plan needs further analysis. If that is the case, it sends the plan back to the Analyzer. The Analyzer and Planner iterate in this manner until the plan generates no new violations. The Decider decides when to terminate the iterations. In the CAD software package installation example, iterations between the Analyzer and Planner would generate a plan consisting of performing a backup followed by an uninstall of the old version of the software, a download of the update to the database component, an installation of the update to the database component, and finally download and installation of the new CAD software package.

In the preceding discussion, we have described the interactions among the Analyzer, Planner, and Decider as totally autonomic. This may not be always possible with the available information. In such cases, a human utility system administrator can easily be introduced into the decision-making loop.

The finalized plan generated by the Plan Evaluator is first forwarded to the Plan Scheduler and then to the Plan Executor. The control flow between the Plan Evaluator and Plan Scheduler is straightforward, as is the control flow between the Plan Scheduler and the Plan Executor. In addition to the specific actions to be taken, the plan handed to the Plan Executor may also consist of expected observations or state changes in the MO, at the end of one or more actions in the plan. These conditions will be observed, verified, and reported back to the utility to update the context and the Active Repository to reflect the actual state of the MO. As the Plan Executor starts to invoke actions, the desired observations and verifications are made and reported back to the utility. Once the Plan Executor is done with the plan, the Change Coordinator dissolves the context and allows the actual changes to be reflected in the Active Repository. If the observed changes do not match the expected changes, the plan provides remedial actions for the Plan Executor to take. For example, one such action may be to undo the last set of actions and then reapply them. If the desired changes are not achieved after a certain number of attempts, the Plan Exec-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

utor sends the plan back to the Utility Manager and waits for the arrival of the next plan.

Putting it all together. We now summarize how the components of the utility manage desired and undesired changes to an MO so that the MO remains operational.

To keep the discussion simple, we assume that initially the MO is in an acceptable working condition and represent its state by [AcceptableState]. Recall that the state of an MO is the collective state of all components represented by the MO. As long as there are no changes to this state, the MO remains in the acceptable working condition. Changes to the [AcceptableState] can occur either because of the changes sponsored by the utility or because of spontaneous changes to the MO. All changes to the MO that are not initiated by the utility are spontaneous. They include device failures, bug manifestations, and user-initiated changes.

In the case of utility-sponsored changes, the following steps lead to the generation of an Action Plan:

```
{Utility Initiated Changes}
    ← ChangeConfigWizard(Administrator Input)

[PredictedState] ←
   ChangeCoordinator([AcceptableState],
                     {Utility Initiated Changes})
```

Note that the utility-sponsored changes are yet to take place on the MO. At this point, the Change Coordinator hands over the [PredictedState] to the Analyzer for determining constraint violations.

In the case of spontaneous changes, the utility observes the changes via monitored events. Thus,

```
[ObservedState] ← StatePreserver
           ([AcceptableState], {Monitored Events})
```

Changes in the MO are observed by the State Preserver after they have actually taken place. At this point, the State Preserver hands over the [ObservedState] to the Analyzer for determining constraint violations.

In either case, the [PredictedState] or the [ObservedState] is handed over to the Analyzer. The Analyzer is not aware of whether the state is observed or predicted. We represent this simply as [MO_State]. The Analyzer, Planner, and Decider iterate over the following loop, which produces an Action Plan without any constraint violations. Initially, the {Action Plan} set is empty.

```
{Constraint Violations}
    ← Analyzer([MO_State]);

do while ({Constraint Violations} is empty)
    {Antidote Actions} ← Planner([MO_State],
                {Constraint Violations});

    {Action Plan} ← {Antidote Actions}
                + {Action Plan};

    [MO_State] ← [MO_State]
                + {Antidote Actions};

    {Constraint Violations}
                ← Analyzer([MO_State]);

od
```

The resulting [MO_State] is the desired end state for the MO, and it is expected to be realized after applying the actions in the {Action Plan} set. A suitable service (along with appropriate parameters) is bound to each action in the {Action Plan} set. However, the steps in applying the services are not optimized. Optimization is performed by invoking the Plan Scheduler, which eliminates unnecessary steps (e.g., duplicate backups or reboots between unrelated actions) and streamlines the application of services so that the actions are applied in an optimized manner. The Plan Executor executes the {Action Plan} by invoking the services as mandated by the plan. The Plan Executor and the Change Coordinator work together to realize the expected [MO_State]. Otherwise, corrective actions are taken to reach a new desired end state. Once the MO is at a desired end state, that state is treated as the new [AcceptableState].

Design considerations

There are many design issues that require careful consideration in realizing an implementation of this architecture. Some of these issues are motivated by performance considerations and others by the state of the art of the available technology. In the following subsections, we address some of these design issues.

Active Repository. In our architecture, the Active Repository plays a key role in representing policy, profile, and dependency-related state information. It also acts as a driving force that prompts other components to perform a "what if" type of analysis. Once a plan for a change is decided, it helps in tracking

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

the actual changes until there is a convergence between the desired state and the actual observed state of an MO. To realize these facets of the Active Repository, we explicitly chose a design that allows us to represent policies, profiles, and components as abstract entities from which we derive instances to model observed and monitored information as well as to represent associations among entities to indicate satisfaction of constraints.

When an MO is configured to satisfy the profile, policy, and dependency constraints, one or more objects are created in the Active Repository. Information is stored in these objects to indicate how the constraints are satisfied. Collectively these objects maintain the dynamic and static state information about each MO. Note that only the constraint-specific state information is stored and tracked. Because of the distributed nature of the environment and the dynamic nature of the MO state, we require only that the Active Repository track the state of an MO closely but not in real time.

For example, the Active Repository contains an instance of the MO for each desktop or server system managed by the utility environment. This representation is shown in Figure 4, using the UML** (Unified Modeling Language) specification. The MO contains information about the hardware, software, and the operational state and associated constraints. For each monitored attribute, the hardware state contains information on maximum capacity, average, and peak utilization, safe operating zone, actions to take in case of exceptions, and so on. The software state contains information about the operating system and a set of installed applications, along with their static and dynamic dependencies and associated set of attributes to monitor. It also contains information on life-cycle stages of the MO and the spatial and temporal dependencies within each stage and during life-cycle transitions. The operation state contains information about the state of the services needed to manage the system such as backup, virus check, and software installation and update services. Information is stored to indicate the frequency with which a service is to be applied, the last time the service was applied, and any related status information. Service description entities provide information on the types of services available and the software products that need to be installed to make that service available. Specific service instances maintain information on the spatial and temporal dependencies among themselves and among the managed objects.

Each MO is also associated with a customer object that contains information about the location where the managed system resides. Each customer contains the state of one or more of the customer's users. One user may be associated with one or more MOs that is used to perform a day-to-day job. Further, associated with each user, customer, and MO are one or more policy objects contained in the Policy Repository.

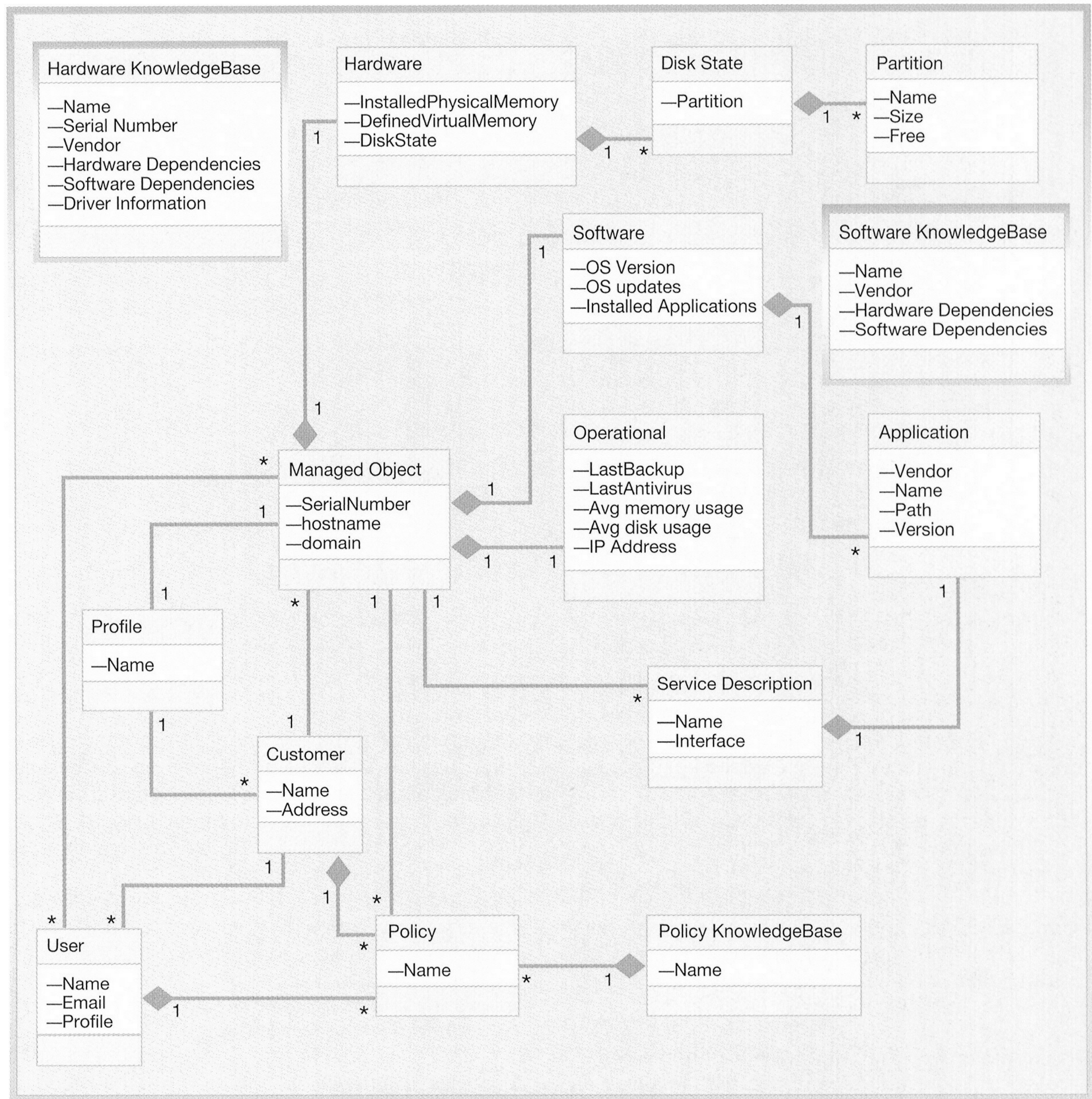
Bootstrapping the Active Repository. So far we have described the Active Repository without describing how it is bootstrapped or how it is configured so that it can respond to changes. There are three different types of activities: (1) populating the repository with objects representing configuration information, constraints, and rules, (2) bootstrapping an object into the Active Repository to represent a real MO that is subject to policy- and profile-related constraints in addition to the configuration constraints, and (3) triggering an event in the Active Repository.

The Active Repository must be populated with rules. These rules are updated as new information and knowledge are gathered. Automated software agents may be used to gather information available in various knowledge bases and incorporate it in the Active Repository. Rules are added and maintained by domain experts using specialized wizards and tools. Because events trigger rules that trigger actions, after the Active Repository is operational, it must be quiescent whenever new rules are added to the repository.

When an MO is to be represented in the Active Repository, first a representative object is created from a template with constraints that are yet to be satisfied. For example, when a new system is introduced in a customer environment, a representative object for that system (possibly with a bare minimum of state information) is injected into the repository. Because the state information associated with the object is incomplete, one or more of the constraints associated with the object remain unsatisfied. This triggers rules associated with the relevant components, profiles, and policies resulting in actions. These actions may, in turn, lead to triggering of other rules and their associated actions. This chain reaction results in an Action Plan to properly configure the newly introduced system. The Action Plan will be carried out by a set of services to be deployed in a certain order to configure the machine to its desired state. As the configuration is completed, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Figure 4 UML specification for the MO representation in the Active Repository

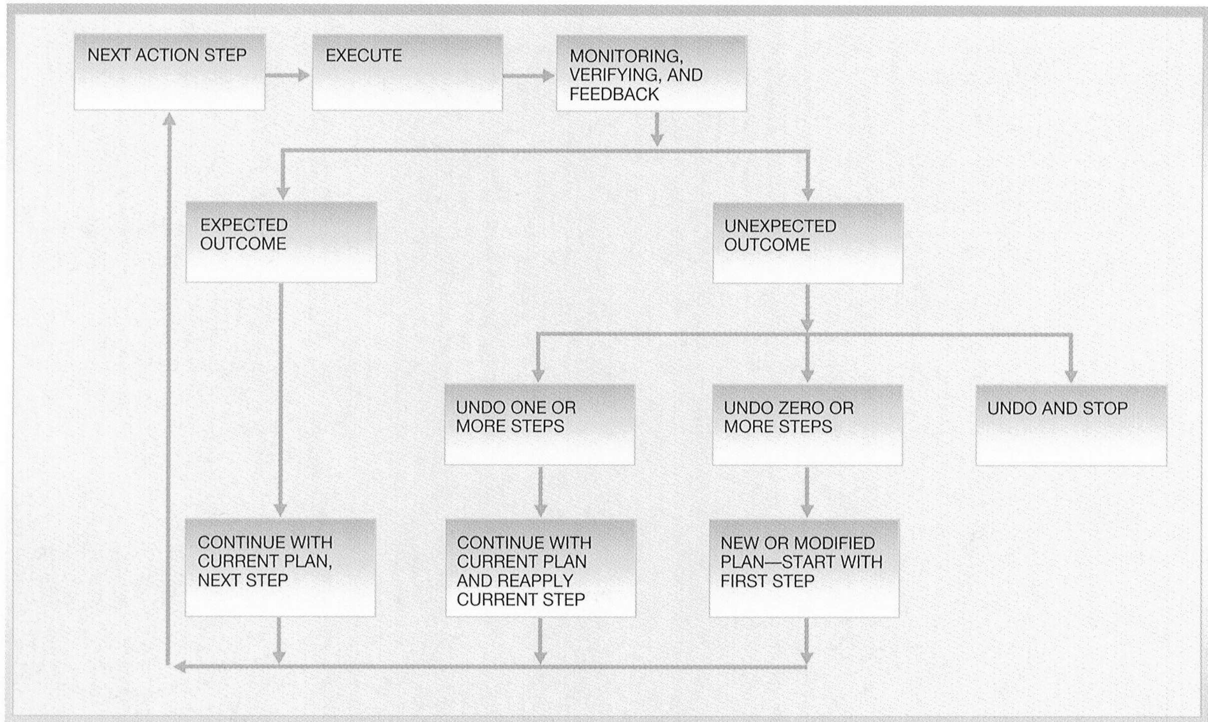


object in the Active Repository is updated with the observed state information.

Events and conditions may be activated in two different ways. One way is when the utility wants to bring about a change in the state of the MO. For example, a scheduled upgrade may need to be performed or a security patch may need to be applied. In such a case, the utility initiates the change, and this change is injected into the Active Repository as a desired state change. The repository behaves as though a real state change has taken place and produces actions to resolve violations. However, the Utility Manager is aware of the trigger provided by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Figure 5 Action Plan execution with risk containment



the utility-initiated change, and it orchestrates the utility components to generate actions and plans.

Real events taking place at the MO can also activate the objects in the Active Repository. For example, the user of a system may install new software without involving the utility. The configuration change event is detected, and a condition is triggered that ultimately activates an object in the repository. The resulting plan is analyzed and evaluated in the same manner as described earlier.

Risk containment. It is important for utility administrators to know and mitigate the risks involved in executing an Action Plan. When routine changes are to be applied to a normal MO, the probability of a successful outcome of an Action Plan is high. When extraordinary changes are to be made, the outcome of the Action Plan may not be so obvious. Similarly, when changes are to be applied to a mission-critical MO, administrators may want to take extra precautions. To provide the desired level of assurance, the actions in the Action Plan can be executed in a mode that we call "risk containment mode." In this mode, actions are performed in a way analogous to a compiler that has its debugging option on, but instead of break points, monitoring and automatic verification points are set at the end of one or more action steps. At the end of a set of action steps that are deemed to exceed acceptable risk, the process inserts additional verification scripts, personalized to the service invocations they succeed and precede. In the general case, these scripts invoke additional services to validate the success of the preceding service invocation and to compute the prerequisites for the subsequent service invocation.

In our design, the Decider determines the effectiveness of the Action Plan and, based on certain predefined criteria such as success probability, prior history associated with the change, minimal downtime, and scope of changes, would either allow the Action Plan to proceed or cause the plan to be further analyzed. The Decider may also provide specific monitoring and feedback to the utility so that the changes can take place in a controlled manner. In case of the failure of any step, the Decider can take corrective actions to roll back the effects of the plan step so

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

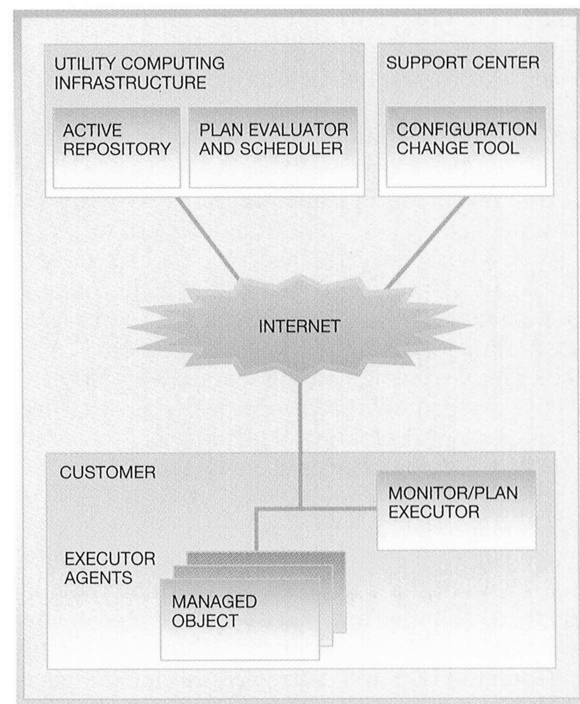
that the system is restored to a consistent state. This process is illustrated schematically in Figure 5. Note that the exact verification steps and their relative frequency can be decided dynamically.

Our goal is to apply system management services such that each state transition of the managed system leaves it in a consistent and usable state. We do not want to create an environment in which failure of an action plan results in the need to completely reinstall the operating system along with the software applications and user data. By controlling the granularity of the atomic changes that are applied to the system, we are able to ensure that the system is easily recoverable to a consistent state. We do realize that not all roll-backs are simple, and some may require reinstallation of the operating system. The architecture and design of the utility minimize the number of such cases, and, when they do occur, automation reduces the need for experts to be involved.

Implementation

We now describe a proposed implementation of the system management utility discussed in this paper. A high-level view of the prototype is shown in Figure 6. The prototype consists of three components: the utility infrastructure, the support center, and the customer environment. Several customer environments can be managed by a single utility infrastructure and support center. The utility infrastructure typically consists of a server farm that hosts deployment services (configuration and setup, and software distribution and upgrades), support services (help desk support and troubleshooting), preventive maintenance services (backups and virus scanning) and other administrative services (asset management, user management, and license management). These services are implemented using off-the-shelf products such as the Tivoli Storage Manager[8] (for backup), Tivoli Configuration Manager[2] (for configuration and software distribution), and IBM Director[9] (for event management and monitoring). The Active Repository, the Plan Evaluator, and the Plan Scheduler are collocated with these services to coordinate their activities. Components of the Utility Manager, Plan Evaluator, and Plan Scheduler are realized as Web services using the IBM Websphere* Application Server,[10] whereas the Active Repository is implemented as a database using IBM DB2*.[11] The Plan Evaluator uses the ABLE (Agent Building and Learning Environment) toolkit for rule evaluation and an inference engine. The support technicians perform system management tasks for customers using the Change Configuration Wizard, which is implemented as a Web service with a browser interface.

Figure 6 Schematic for realizing the system management utility



At the customer site, all the desktops managed by the utility are connected via a local-area network to a *management server.* The primary role of the management server is to act as a local proxy for the utility services. To ensure security of all communications between the customer site and the utility, the on-site server is connected via an outbound (customer-site initiated) VPN (virtual private network) connection to the utility infrastructure. Each managed system at the customer site contains a standard software stack that consists of the base operating system, business applications, and agents representing one or more utility services. In addition to these agents, the customer environment also hosts the client-side Monitor and the Plan Executor components of our architecture. Both the Monitor and Plan Executor subsystems are implemented as extensions to the IBM Director Server and its agents that run on individual systems.

المنارة للاستشارات
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Related work

System management is a well-established discipline with its own extensive literature (e.g., Reference 12). Much of this concerns systems, algorithms, and techniques for accomplishing a specific management goal, as in, for example, the automation of event response. In this paper, we are concerned primarily with the orchestration of management services.

A common theme in the references cited here is the introduction of integration middleware, often as proxies for a managed object or for a gateway that manages a family of related objects. This integration middleware may be passive, functioning as an adapter or format and protocol translator or, as in the examples below, active integration middleware. Active integration middleware performs such functions as event generation, detection of inconsistencies in the monitored state, and aggregation.

Aschemann and Kehr[13] focus on the information model with which management applications interact. They conclude that existing information models are too limiting and that object-oriented models are required. Management applications are written in Common Lisp and include constraint and query sublanguages. Although a constraint solver has yet to be included in their system, constraints play a major role in their thinking. Our focus is similar to theirs in several respects, but differs in that our repository is a shadow of the information maintained in autonomous subsystems, and we focus on the dynamic plug-in nature of those subsystems.

Anerousis[14] writes of Marvel, a "distributed computing environment for building scalable management services using intelligent agents and the world-wide web." The goals of Marvel are similar to ours, in that its emphasis is on integrating existing management services and exposing their data to management applications. Analogous to the ability of databases to support multiple views of their data, Marvel supports the definition of new managed objects to represent computed views of existing management information, acquired through a process of *spatial aggregation*, or grouping, and *temporal aggregation*, or the extraction of parameters from a time series. Control originates with intelligent agents, downloaded dynamically.

Aschemann et al.[15] describe an architecture for the dynamic discovery and configuration of managed objects using Jini**[16] technology. Monitoring and control is mediated by a *configuration service*, which contains a configuration repository, a scheduling service, protocol adapters, and most relevant here, a "rule base which contains rules and policies to preserve the consistency of the configuration." Managed objects are represented by a so-called nanny[17] that proxies for the object. The repository is active in that it can perform arbitrary processing when an event is received or when the state of an MO is changed.

Lewis et al.[18] describe a useful case study of fulfillment of customer orders for a switched ATM (asynchronous transfer mode) service. The case study expresses the solution, expressed as subscription and configuration management and a network planner, in terms of the standard TINA (Telecommunications Information Network Architecture) and TM (TeleManagement) Forum specifications.[19] This case study illustrates the application of these standards and presupposes a standards-based implementation. If broadly accepted and widely implemented standards were available for system management, this case study shows how simple it might be to build an integrated system.

Yang et al.[20] describe ViaScope, a virtual enterprise information integration system based on CORBA** (Common Object Request Broker Architecture**). Managed objects in ViaScope consist of entities, meta-data, and events, each of which has a manager. Relevant to this paper is the Information Integrating Engine, which implements numerous integration functions, including a bulletin service, access negotiation service, and integration processing. This engine permits all managed objects to have a highly functional, uniform distributed appearance.

Hong et al.[21] review the alternatives in Web-based management and give some useful examples of how they can be applied. They observe that "the main problem is that none of the existing management approaches can be used to manage all types of internet/intranet resources," precisely the problem that occupies us here.

Conclusions and future work

Any complex endeavor involves risk, and system management is no exception. System administrators may fail to translate policies into corresponding actions, communications may fail at inopportune times, hardware and software may fail, and procedures may be run in environments that do not match their prerequisites. In our environment, the system manage-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

ment utility is expected to manage thousands of customers, each with their own set of policies and rules. Even with standardization, slight variations across systems can have a large impact on the complexity of the tasks in the system management utility environment.

In this paper, we have described an approach for building a large-scale utility for managing remote systems spread over multiple customer sites. The system is designed, keeping in mind the need for a high degree of automation, so that not only can system management chores be automatically executed, but the action plans can also be derived automatically. The dynamic interactions among objects in the Active Repository in response to observed or simulated state changes in the remote managed objects allow the composition of Action Plans that in effect integrate multiple system management services. Our system is also capable of analyzing and reasoning about a proposed plan. When a plan is determined to have a degree of uncertainty about its outcome, its execution can be carried out using a finer-grained monitoring, verification, and feedback mode so that the steps taken can be reversed before undesirable effects render the system unmanageable. The Utility Manager, Plan Evaluator, Plan Scheduler, and Plan Executor together provide meta-system management services that enable controlled execution of complex management tasks.

Much more remains to be done toward realizing the ultimate goal. Ongoing challenges include growing and maintaining expert knowledge, rule authoring, dependency generation, and maintenance. We want to extend the system to cross-platform (diverse operating environments) management. Future work includes risk management, performance measurements, and effectiveness of the design in field trials.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Sun Microsystems, Inc., or Object Management Group.

Cited references

1. Microsoft Systems Manager Server, Microsoft Corporation, http://www.microsoft.com/smserver/.
2. Tivoli Configuration Manager, IBM Corporation, http://www.ibm.com/software/tivoli/products/config-mgr/.
3. *Implementing Automated Inventory Scanning and Software Distribution after Auto Discovery*, Redbook, SG24-6626-00, IBM Corporation.
4. N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "*Ponder: A Language for Specifying Security and Management Policies for Distributed Systems*," Research Report DoC 2000/1, Imperial College, London (January 2000).
5. J. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Miller III, and Y. Diao, "ABLE: A Toolkit for Building Multiagent Autonomic Systems," *IBM Systems Journal* **41**, No. 3, 350–371 (2002), http://www.research.ibm.com/journal/sj/413/bigus.pdf.
6. J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computing* **36**, No. 1, 41–50 (2003).
7. A. G. Ganek and T. A. Corbi, "The Dawning of the Autonomic Computing Era," *IBM Systems Journal* **42**, No. 1, 5–18 (2003).
8. Tivoli Storage Manager, IBM Corporation, http://www.ibm.com/software/tivoli/products/storage-mgr/.
9. IBM Director with UMS [Universal Management Server], 2.2–Overview, IBM Corporation, http://www-1.ibm.com/support/docview.wss?uid=psg1MIGR-4V2UP7.
10. WebSphere Application Server, IBM Corporation, http://www.ibm.com/software/info1/websphere/index.jsp.
11. DB2 Information Management, IBM Corporation, http://www.ibm.com/software/data/db2/library/.
12. *Journal of Network and Systems Management*, Kluwer Academic/Plenum Publishing Company, New York.
13. G. Aschemann and R. Kehr, "Towards a Requirements-Based Information Model for Configuration Management," *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, MD (May 1998), pp. 181–189.
14. N. Anerousis, "An Architecture for Building Scalable, Web-Based Management Services," *Journal of Network and Systems Management* **7**, No. 1 (1999).
15. G. Aschemann, S. Domnitcheva, P. Hasselmeyer, R. Kehr, and A. Zeidler, "A Framework for the Integration of Legacy Devices into a Jini Management Federation," *Proceedings of the Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '99), Lecture Notes in Computer Science* **1700**, Springer-Verlag, Heidelberg (October 1999), pp. 257–268.
16. See, for example, "The Jini Specifications, Second Edition," J. Waldo and K. Arnold, Editors, Pearson Education, Upper Saddle River, NJ (December 2000); see also Jini Network Technology, Sun Microsystems, Inc., http://wwws.sun.com/software/jini/.
17. G. Aschemann and P. Hasselmeyer, "A Loosely Coupled Federation of Distributed Management Services," *Journal of Network and Systems Management* **9**, No. 1, 51–65 (2001).
18. D. Lewis, C. Malbon, G. Pavlou, C. Stathopoulos, and E. J. Villoldo, "Integrating Service and Network Management Components for Service Fulfilment," *Proceedings of the Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '99), Lecture Notes in Computer Science* **1700**, Springer-Verlag, Heidelberg (October 1999), pp. 49–62.
19. G. Pavlou and D. Griffin, "Realizing TMN-like Management Services in TINA," special issue on TINA, *Journal of Network and Systems Management* **5**, No. 4, 437–457 (1997).
20. X. Yang, G. Wang, G. Yu, and D. L. Lee, "Modeling Enterprise Objects in a Virtual Enterprise Integrating System: ViaScope," *Lecture Notes in Computer Science*, **1749**, Springer-Verlag, Heidelberg pp. 166–175 (1999).
21. J. W. Hong, J. Y. Hong, T. H. Yun, J. S. Kim, J. T. Park, and J. W. Baek, "Web-Based Intranet Services and Network Management," *IEEE Communications Magazine* **35**, No. 10, 100–110 (October 1997).

Accepted for publication September 20, 2003.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com
المنارة للاستشارات

Vijay K. Naik *IBM Research Division, Thomas J. Watson Research Center, P. O. Box 218, Yorktown Heights, New York 10598 (vkn@us.ibm.com).* Dr. Naik is a research staff member in the Internet Infrastructure and Computing Utilities Department. His current research areas include distributed computing, in particular, utility computing, peer-to-peer, and grid computing. He is the author of the book, *Multiprocessing: Trade-Offs in Computation and Communication*. He has also published over 40 articles in journals and refereed conferences. Prior to joining IBM in 1988, Dr. Naik was a staff scientist at ICASE, NASA Langley Research Center. He received a Ph.D. in computer science from Duke University.

Ajay Mohindra *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (ajaym@us.ibm.com).* Dr. Mohindra has been a research staff member at IBM since 1993. He holds a Ph.D. in computer science from the Georgia Institute of Technology. His research interests include distributed systems and autonomic and utility computing.

David F. Bantz *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (bantz@watson.ibm.com).* Dr. Bantz has been a research staff member since 1972, after a short stint at a startup company. He graduated from Columbia University in 1970 with an Eng. Sc. D. degree and taught there as an adjunct professor for nearly 25 years. He has 20 issued patents. His technical interests have always been in personal computing applications and technology, and he is currently working on autonomic personal computer management.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com