iManage: Policy-Driven Self-Management for Enterprise-Scale Systems

Vibhore Kumar, Brian F. Cooper[†], Greg Eisenhauer, Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{vibhore, eisen, schwan}@cc.gatech.edu

Yahoo! Research[†]
2821 Mission College Blvd.
Santa Clara, CA 95054
cooperb@yahoo-inc.com

Abstract. It is obvious that big, complex enterprise systems are hard to manage. What is not obvious is how to make them more manageable. Although there is a growing body of research into system self-management, many techniques are either too narrow, focusing on a single component rather than the entire system, or not robust enough, failing to scale or respond to the full range of an administrator's needs. In our *iManage* system we have developed a policy-driven system modeling framework that aims to bridge the gap between manageable components and manageable systems. In particular, iManage provides: (1) *system state-space partitioning*, which divides a large system state-space into partitions that are more amenable to constructing system models and developing policies, (2) *online model and policy adaptation* to allow the self-management infrastructure to deal gracefully with changes in operating environment, system configuration, and workload, and (3) *tractability and trust*, where tractability allows an administrator to understand why the system chose a particular policy and also influence that decision, and trust allows an administrator to understand the system's confidence in a proposed, automated action. Simulations driven by scenarios given to us by our industrial collaborators demonstrate that iManage is effective both at constructing useful system models and in using those models to drive automated system management.

Keywords: Policies, Self-Management, Bayesian Networks, Enterprise-Systems

1 Introduction

Consider large systems that are integral parts of an enterprise's IT infrastructure. Examples of such systems include those supporting enterprise websites, or inventory management subsystems, or even the distributed information systems supporting a company's daily operations. Administrators managing these systems are not only expected to keep them running, but in addition, many such systems must meet certain processing constraints, be highly available, offer differentiated levels of Quality of Service (QoS), meet certain Service Level Agreements (SLAs), and may be subject to unforeseen demands. Unfortunately, even the occurrence of seemingly routine events like load changes, node and link failures, software patches, or modifications of certain environmental parameters can cause such systems to behave in unexpected ways, often resulting in their failure to meet current objectives. Given these facts and acknowledging enterprises' growing reliance on their computing infrastructures, solutions must be found for system self-management. These solutions must be driven by high level business goals, be open and receptive to administrators, cope with dynamic changes in requirements and conditions, and scale from small, tightly managed individual subsystems to large company-wide support infrastructures.

The existing tools and techniques for enabling self-management of enterprise-scale systems are insufficient because they are either too general or are too specific. For instance, the state-of-the-art system management tools deployed at large enterprises include software suites like IBM's Tivoli, which is a systems management platform and HP's OpenView (now combined with Mercury), which can be used for managing large-scale systems and networks[1]. These tools are equipped with methods for system monitoring and for graphically displaying system status to administrators. However, their functionality for automated symptom determination, reasoning about symptom causes, and taking appropriate corrective actions remains rudimentary, in part due to the lack of standards and more importantly, due to the general nature of these tools. In contrast, researchers have successfully embedded self-managing capabilities into specific well-defined subsystems like database backends [28], request schedulers for multi-tier web services [8] and others. To complement the self-management work being done for specific subsystems, several researchers have been focusing on issues like policy-specification language [11], model building techniques [4] and efficient monitoring schemes [1]. Similarly, there has been some excellent research in the domain of automating specific tasks that are required for enabling self-management. A particular effort of note is the work on automated problem diagnosis, presented in [9, 32]. The work focuses on using the monitoring data gathered from a system to detect service level objective violations and correlating the violation to earlier violations for gaining useful insights. While automating subsystems and problem diagnosis is important, these specific techniques must be combined into a comprehensive framework in order to be effective for complex systems.

The goal of our research is to develop abstractions and methods that help bridge the gap between (i) the excellent progress made in the general domain of self-management, like automation of well-defined subsystems or specialized techniques for self-management tasks vs. (ii) the more general challenges posed by managing more complex and/or larger IT infrastructures and applications. Toward this end, we build on such prior work for online system management, we adopt the use of online monitoring and behavior detection tools and techniques [10], and we endorse the use of ECA policies to describe and build our self-management framework. To also address the broader management challenges posed by complex and dynamic IT applications and infrastructures, however, we propose a novel representation of the system state-space that is geared towards policy-based self-management, and we develop new techniques for dealing with the problems of *scale*, *dynamism*, *tractability* and *trust*. Tractability here refers to an administrator's ability to understand current management actions undertaken by the system and to the system's ability to expose its reasoning for those actions. To achieve the goal of system manageability, our system, iManage, offers the following tools:

[1] IBM, Tivoli, HP, OpenView and Mercury are registered trademarks of their respective owners

– *A system modeling framework* - iManage collects system parameters and metrics (collectively called *system variables*) into a single representation of the system state-space, and identifies which actions are available to change the system state.
– *A scheme for reducing the complexity of the system model* - Since a typical system model is too complex to be used or even properly constructed, our tools provide mechanisms to partition the state-space into smaller units that are easier to deal with. These *micro-models* allow us to more precisely model critical aspects of the system, and to more effectively develop policies.
– *Techniques for evolving system models* - Policies that are appropriate under one set of conditions may become invalid as operating conditions and the environment changes. iManage provides techniques for evolving system models and policies, including methods to learn new policies and incorporate human knowledge and experience to refine the policies.
– *Techniques for quantifying our confidence in a system model* - In order for our system models to be useful, the system administrator must be able to trust them. iManage associates a confidence value with each self-management policy, and allows the administrator to both understand and use this confidence value when deciding whether to let the system manage itself.

In the following section we motivate the iManage approach by describing certain subsystems and properties of the operational information system deployed by Delta Air Lines, one of our industry partners. Our interactions with the administrators and developers at that site have motivated much of this work.

1.1 Motivating Example

The Passenger Information Delivery System – PIDS (shown in Figure 1) – is a middleware developed at Delta Technology, Inc. to serve two important needs of the airline. First, it is responsible for managing the passenger data sourced from the airline's TPF mainframe. Second, it provides access to passenger information via events and service interfaces. The PIDS middleware, which according to estimates by Delta Technology processes around 9.5 billion events annually, ensures near real-time delivery of processed events to 'consumers' – programs that need to receive the events – and to a database of current booking and flight information used in activities like those in support of Delta's web site. PIDS collects data from all over the airline. While much of its information comes out of the airline's TPF-based Deltamatic Reservation and Operational Support System (OSS), additional inputs like gate information, information about weather, etc. arrive from airports throughout Delta's worldwide system. Further passenger information is provided by the reservation system. Finally, most planes generate and transmit their own landing time, which is provided to PIDS via FPES (the flight progress event system).

There are hundreds of variables associated with the PIDS system that capture the current state of the PIDS servers, the current load conditions, client specific metrics and several others. Some of these variables are enumerated in Table 1. A system administrator manages the system by virtue of having the ability to modify some of these variables, examples including the number of client service threads or the number of workflow service threads. More specifically, such modifications of state variables constitute the set

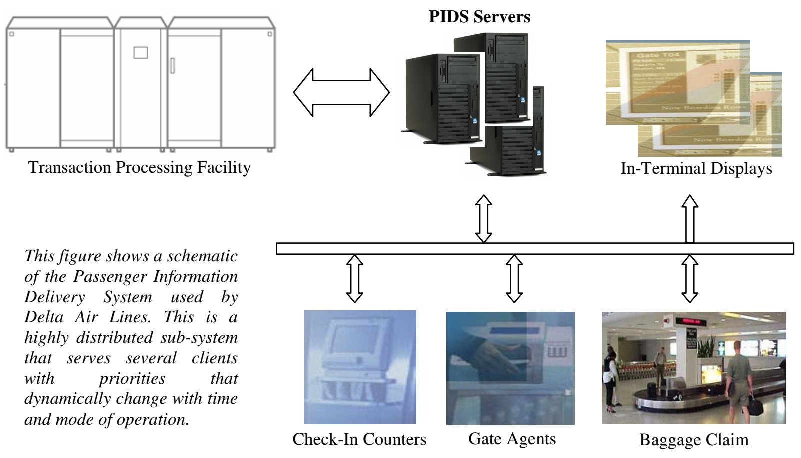
PIDS Servers
Transaction Processing Facility
In-Terminal Displays
This figure shows a schematic of the Passenger Information Delivery System used by Delta Air Lines. This is a highly distributed sub-system that serves several clients with priorities that dynamically change with time and mode of operation.
Check-In Counters
Gate Agents
Baggage Claim

Fig. 1. Some Interactions in the PIDS Middleware System

of actions allowed for managing the system. The actions of a system administrator to respond to an event (like increased workflow processing delay) are based on his wisdom (mental model of the system behavior) and the prevailing conditions (values of different variables representing the current state). However, partial (and sometimes complete) failures of PIDS middleware are not uncommon, often resulting in delayed and/or canceled flights, and eventually leading to loss of revenue. Such failures can be attributed to the scale of the PIDS middleware and to the dynamic load conditions posed by the application domain.

The above example justifies our focus on the issue of scalability when designing our self-management framework. Moreover, in order to deal with the dynamic load conditions experienced by the PIDS middleware one must make use of self-management techniques that can continuously evolve. Finally, our interaction with the system administrators running the PIDS middleware motivated the need to keep the humans in the self-management loop and in control of the adaptation actions which translated to the requirements for tractability and trust.

1.2 Road Map

The rest of this paper is organized as follows. In Section 2 we present an overview of the overall approach, introduce the system state-space model used by the iManage framework and describe the requirements for policy enablement. Section 3 focuses on the specifics of our approach by describing the algorithms and techniques used by our framework, these include - the partitioning algorithm, the model building technique and the specifics of policy learning, adaptation and the confidence attribute. In Section 4 we present the evaluation of our techniques. Section 5 discusses the related work and finally, we conclude in Section 6 with some open problems for further research in this area.

Table 1. Some variables associated with the PIDS middleware

Variable	**Description**
	Global Variables
E2EL	The end-to-end latency introduced by the processing workflow.
ELPP	The average queuing delay at individual PIDS processing nodes.
CLIE	Number of cache access clients being served at any time.
ETTR	Expected time to recover from a failure.
EDRR	Events dropped in last 100,00,000 events.
CSTH	Client service threads at individual PIDS processing nodes.
WSTH	Workflow service threads at individual PIDS processing nodes.
NGAG	Number of active boarding gates
NBCA	Number of active baggage claim
NCIC	Number of active check-in counters
NOVR	Number of active overhead displays
	Gate Agent Variables
TTFD	Time to flight departure
DEST	Identifies whether a flight is domestic or international
NPAS	Number of passengers scheduled to board the flight

2 Overview & State-Space Model

In this section we present an overview of our solution approach, which is followed by a formal description of iManage's system state-space model and thereafter we present the requirements for policy-enablement of an enterprise-system. The formal model is used in the following sections to formally describe the various algorithms and techniques used by the iManage framework.

2.1 Solution Overview

The iManage framework for policy-driven self-management of enterprise-scale systems provides an abstraction of a system state-space, where each axis represents an identifiable system variable (e.g., end-to-end delay, throughput, etc.). The state-space model specifically identifies two sets of variables - one set contains the variables that determine the operational status of the system and the other set contains the variables that can be modified to affect the state of the system. The first set is used for specifying the goals or SLAs and the second set is used to determine 'actions' that later become part of ECA policies for the system. In order to manage the system one needs to establish a model that connects the set of action variables to the set of goal variables. However, given the scale of the state-space for enterprise-scale systems and the fact that the system can exhibit different behaviors in different state sub-spaces, modeling the state-space is not straight-forward. The iManage framework utilizes a novel state-space partitioning scheme to deal with the problems of scale and heterogeneous system behavior. iManage then makes use of tree augmented naive Bayesian networks or TANs to build 'micro-models' for each partitioned sub-space that results from the state-space partitioning algorithm. As a result, the system model becomes a collection of the 'micro-models' constructed for each sub-space. In case some goal violation is detected, the system model is consulted to arrive at new values for the set of action variables. In terms of policy the goal violation becomes the *event*, the value of system variables at the time of violation become the *condition* and the assignment of new values to the set of action variables becomes the *action*. Since, probabilistic models are used to arrive at a solution in case of goal violation, even the suggested policy actions are associated with a certain probability of bringing the system to a state of non-violation, this probability acts as the confidence attribute for the policy. A policy is enforced only when the confidence attribute exceeds the threshold set by a system administrator.

2.2 System State-Space Model

The following convention is used to describe the system state-space model. We use boldface capital letters such as, $\mathbf{V}, \mathbf{V}_\phi$ to denote sets, and assignment of values to variables in these sets are denoted by regular capital letters such as $\mathtt{V}_1, \mathtt{V}_2$. Similarly, we use boldface lower case letters such as, $\mathbf{v}_i, \mathbf{v}_j$ to represent variables that occur in the sets, and regular lower case letters such as, $\mathtt{v}_1, \mathtt{v}_2$ denote specific values taken by those variables.

We consider a system whose state can be represented by a set $\mathbf{V}$ of n variables $\{\mathbf{v}_1, ..., \mathbf{v}_n\}$, which are not necessarily independent. Out of these n variables the system's operational status (like failed, stable, unstable, etc.) can be determined by using only a subset $\mathbf{V}_\phi$ (an example of such variable would be the delay experienced by the users of an enterprise's website) of the state variables in $\mathbf{V}$. Therefore, $\mathbf{V}_\phi$ is the set of variables of interest as far as the system's operational status is concerned.

Furthermore, we associate the system with a set $\mathbf{A}$ of m action interfaces $\{\mathbf{a}_1, ..., \mathbf{a}_m\}$, such that an instance $\mathtt{a}_1$ of action interface variable $\mathbf{a}_i$ represents an action that can be invoked on the system. The invocation of an action $\mathtt{a}_1$ on a system state $\mathtt{V}_1$ is denoted by $\Omega(\mathtt{a}_1, \mathtt{V}_1)$, which possibly translates the system to a new state. The effect of invocation of action $\mathtt{a}_1$ on an instance of a system state-space variable $\mathtt{v}_1$ is similarly represented using $\omega(\mathtt{a}_1, \mathtt{v}_1)$. The above discussion is used to arrive at the following definition of a deterministic action-variable pair.

Definition 1. *A tuple $(\mathbf{a}_i, \mathbf{v}_i)$ is said to be a deterministic action-variable pair if $\omega(\mathtt{a}_j, \mathtt{v}_k)$ is known for all instances $(\mathtt{a}_j, \mathtt{v}_k)$ of $\mathbf{a}_i$ and $\mathbf{v}_i$.*

The set of all deterministic action-variable pairs of a system constitutes the set $\mathbf{D}$, and the set of all state-space variables that occur in any tuple in the set $\mathbf{D}$ constitute the set $\mathbf{V}_\alpha$, also called the set of actionable variables. The following lemma holds for all members of the set $\mathbf{V}_\alpha$.

Lemma 1. *If $\mathtt{v}_1$ and $\mathtt{v}_2$ are two possible values of the state-space variable $\mathbf{v}^i_\alpha \in \mathbf{V}_\alpha$ then there exists an instance $\mathtt{a}$ of $\mathbf{a}_i$ such that $(\mathbf{a}_i, \mathbf{v}^i_\alpha) \in \mathbf{D}$ and $\omega(\mathtt{a}, \mathtt{v}_1) = \mathtt{v}_2$.*

In order to manage a system, and affect its status, one needs to be able to deterministically modify the value of variables contained in $\mathbf{V}_\phi$. However, we only know of ways to deterministically modify the value of variables contained in $\mathbf{V}_\alpha$. Therefore, if one could discover a function χ that maps the space of variables of interest, $\mathbf{V}_\phi$ to the space of actionable variables, $\mathbf{V}_\alpha$ then one would be able to manage the system as described next. Let, $\mathtt{V}^{current}$ represent the current state of a system and $\mathtt{V}^{current}_\phi$ and $\mathtt{V}^{current}_\alpha$ represent values of the corresponding sets of variables $\mathbf{V}_\phi$ and $\mathbf{V}_\alpha$. Now, if the system needs to be translated to a new feasible state such that the variables of interest take the value $\mathtt{V}_{\phi}^{goal}$, then one should be able to determine $\mathtt{V}_{\alpha}^{goal}$ using the function χ and then use the set $\mathbf{D}$ to determine the actions required to change the value of variables in $\mathbf{V}_{\phi}$ from $\mathtt{V}_{\alpha}^{current}$ to $\mathtt{V}_{\alpha}^{goal}$.

Note that the set of variables in $\mathbf{V} - (\mathbf{V}_{\phi} \cup \mathbf{V}_{\alpha})$ are not redundant and as we shall see in Section 3.1 they play an important role in determining the function χ. An example of such variable would be a measurement of number of disk-operations - such a variable is usually not a member of $\mathbf{V}_{\phi}$, which is used to determine acceptable system operational status; and this metric, in general, cannot be deterministically affected by allowed system actions (e.g. allocating another disk-array). However, such variables may give hints about the actions to be taken to remedy a certain problem.

To put the above discussion in context, such a system model can be readily applied to the example discussed in Section 1.1. For example, the list of variables, enumerated in Table 1, constitute the set $\mathbf{V}$ of state variables for the PIDS system. The set of variables $\{E2EL, CLIE\}$ are the variables of interest as far as the operational status of the PIDS middleware is concerned and therefore constitute the set $\mathbf{V}_{\phi}$ (this corresponds to two of the several requirements imposed on the PIDS middleware -*the processing workflow should not introduce a delay of more than 1 second* and *the system should be able to handle 3000 concurrent requests from the clients*). The set $\mathbf{V}_{\alpha} = \{CSTH, WSTH\}$ constitutes the set of variables that have action associations.

Limitations. In the above discussion we assumed that all the variables that constitute the system state-space are known. This is not true for several systems where due to considerations like monitoring overhead and complexity some of these variables might not be monitored. However, the probabilistic modeling techniques used by our framework are able to perform sufficiently well even when some of the variables are not listed as members of the system state-space, or are not monitored by the system. One must note that failure to include some important state-space constituents may lead to a system model which might not be manageable.

The second limitation arises from the fact that the function χ might return multiple possible instances of the set $\mathbf{V}_{\alpha}$ corresponding to the goal state represented by $\mathtt{V}_{\phi}^{goal}$. For example, if $\mathtt{V}_{\phi}^{goal}$ corresponds to reduction in end-to-end delay for a three tier web-server then there may exist multiple actions like increasing the number of front-end servers or upgrading the backend database server that may lead to reduction in end-to-end delay. Our probabilistic techniques will suggest the solution which has the highest probability of resolving the problem without any guarantees about the efficiency or optimality of the solution. This opens up the possibility of a difference between 'manageable' and 'efficiently' or 'optimally' manageable system. However, in this paper we will limit ourselves to the concept of manageability.

2.3 Enabling Policies

There are certain requirements that should be met by any system to become eligible for policy-driven self-management. Firstly, the system should be able to measure and export the current value of variables that constitute the state-space for the system. One can think of this as 'dials' on a control dashboard used for managing a very large system. Secondly, the ability to modify some of the variables is also central to the idea of policy enablement. One can similarly think of this capability as the 'knobs', which can deterministically change the value displayed on some 'dials'. In terms of our system state-space model, the variables represented by 'dials' are the variables in the set $\mathbf{V}$. The variables which have an associated 'knob' constitute the set $\mathbf{V}_\alpha$. The 'knobs' can in turn be used to take actions specified using $\omega(\mathtt{a}_i, v_\alpha^j)$.

In order the enable policies in a policy-ready system we need to have a way for representing the policies, mechanisms that discover and learn policies at runtime, ways to enforce policies and techniques for keeping the policies updated for the current system environment. The following sub-sections briefly describe our approach to handling these issues. Some of these issues will later be dealt in detail in Section 3

Policy Specification - We use a modified form of the well accepted event-condition-action (ECA) format for specifying the policies. The ECA specification is very useful when it comes to enforcing policies for any system. We however, extend the specification to include a confidence-attribute that is related to the probability of the policy having a desired effect when the action specified as part of the policy is taken under appropriate conditions. The *event* in our policy description is a change in the value of some variable(s) in $\mathbf{V}_\phi$. The *condition* that triggers the action associated with the policy is specified over the set of variables in $\mathbf{V}$. The *action* is similarly specified as the modification in the value of some variables contained in $\mathbf{V}_\alpha$.

Policy Discovery - We believe that all policies cannot be specified and that the system may need to discover some policies on the fly. We use a novel state-space partitioning scheme, described in Section 3.1 to first reduce the system state-space under consideration at any instant. Then for each partition we make use of greedy algorithm to discover the most important variables from the set $\mathbf{V}_\alpha$ (i.e. the right knobs). We finally make use of Bayesian networks to build 'micro-models' of the the state-space corresponding to each partition, thereby enabling us to find the values to which the 'knobs' should be adjusted to. We elaborate on these techniques in the following sections.

Policy Enforcement - The interfaces that export the current value of system variables are continuously monitored for any changes. These changes in the value of some variables may cause some policy to evaluate its condition and if the condition evaluates to true the action specified as part of the policy is taken. In simple words, when a problem occurs (i.e. the value on some dials signals something bad) the self-management subsystem tries to (1) find the 'right-knobs' and then (2) adjusts them to some appropriate new values. The enforcement of any policy is also contingent on the confidence-attribute, which should be more than a system-wide threshold set by the system administrator. This gives the administrator a control over the degree of self-management.

Policy Refinement - Policies that are either specified or are learnt by the system may need to be changed because the conditions under which such policies are valid may change with time. An instance of this would be the addition of more nodes to the network underlying the operational information system. Such instances may lead to changes in threshold values that trigger an action specified as part of the policy. Our techniques are able to keep track of such changes in the environment and in response, they suitably modify the policies.

3 Solution Approach

The system state-space model proposed in Section 2.2 showed that $\mathbf{V}$, $\mathbf{V}_\phi$, $\mathbf{V}_\alpha$, $\mathbf{A}$, $\mathbf{D}$ and χ are the parameters that should be known for arriving at a self-management solution for a system. One can safely assume that for most of the systems the sets $\mathbf{V}$, $\mathbf{V}_\phi$, $\mathbf{V}_\alpha$, $\mathbf{A}$ and $\mathbf{D}$ are known apriori. This implies that the system variables, the variables of interest and the deterministically modifiable variables along with the ways to modify them are known. This is true for enterprise-scale systems where the system variables like number of network nodes, link capacities, etc. are known, similarly the variables of interest like end-to-end delay are also known apriori and lastly one knows of variables like allocated buffer-length at network nodes which can be deterministically modified by changing some system parameters. The problem is to find the function χ, and this means that we need to find a way to model the system. Remember that the function χ relates the variables in $\mathbf{V}_\phi$ to the variables in $\mathbf{V}_\alpha$ and the function χ can change for different values of variables in $\mathbf{V} - (\mathbf{V}_\phi \cup \mathbf{V}_\alpha)$. Once the function χ has been determined for the system state-space, one can easily find and/or adapt the actions that form part of the policy specification.

However, building a model (i.e. determining χ) for understanding the behavior of an enterprise-scale system is a tough task. This can be attributed to the fact that in such systems there are a large number of variables (e.g., bandwidth, workload, queue length at servers, etc.), each one of which can potentially affect the state of the system and more often than not these variables also interact amongst themselves. For example, in a certain sub-space of the system's state-space the bandwidth between participating nodes may be the bottleneck and any modification to the priority of processes may have little or no effect on the observed performance. The situation may similarly be reverse for some other system state sub-space where server capacity may be the bottleneck and any modification to the inter-node bandwidths may have no effect on the observed performance. The two insights that follow from the above discussion are that -

1. Finding a single function to model the entire state-space of an enterprise-scale system might lead to very crude and incorrect system models.
2. There exists system state sub-spaces where the effect of certain variables can essentially be ignored from the system model.

The above discussion motivates the need to partition the system state-space. The following sub-section elaborates on the specific requirements for the partitioning scheme and then describes the partitioning algorithm in detail.

3.1 System State-Space Partitioning

The aim of our partitioning scheme is to create system state-space partitions such that -

– the involved system variables exhibit some homogeneity in their behavior inside the partition, which is beneficial for building the system model.
– the number of 'knobs' required to manage the system within the partition is minimized, which is beneficial for the purpose of learning and adapting the actions specified as part of policy.

To incorporate the concept of partition homogeneity we create partitions such that operational states contained in the partition are close to each other. Note that partition homogeneity corresponds to macro-level states of the system, for instance in one partition the underlying network may be the bottleneck (making server capacity redundant) while in some other partition the server capacity may be the bottleneck (similarly making the network capacity redundant). In order to minimize the 'knobs' we want to ensure the partitions are created such that the 'knobs' needed in one partition are possibly not needed in the other. This corresponds to making partitions which are orthogonal to each other. The partitioning algorithm employed by our framework is described next.

The Partitioning Algorithm A system state can be defined as the binding of appropriate values to the variables contained in the set $\mathbf{V}$. The partitioning algorithm aims to partition many such observed system states to achieve the objective mentioned in the previous section. We define a partition to be a collection of observed system states. A partition inherits the sets $\mathbf{V}$ and $\mathbf{V}_\phi$ from the system state-space but the sets $\mathbf{V}_\alpha$, $\mathbf{A}$ and $\mathbf{D}$ can vary between the partitions.

Let, S be the observed operational states contained in the initial system state-space partition for which $\mathbf{D}$ defines the association of action interfaces in $\mathbf{A}$ with the variables in $\mathbf{V}_\alpha$. For simplicity the discussion here assumes that it is possible to define a measure of normalized distance between any two operational states. Techniques for doing such operations exist and interested readers may refer to well-known techniques like Mahalanobis distance [24]. We define an operator $\delta_{\mathbf{R}}$ over a pair of operational states from a partition, which finds the normalized distance between the two operational states considering only the dimensions contained in the set $\mathbf{R}$, where $\mathbf{R} \subseteq \mathbf{V}$. We also define the operation θ over a pair of operational states from a partition. The operation θ finds the number of places in which the two states differ, considering only the dimensions corresponding to the set $\mathbf{V}_\alpha$ for the partition under investigation. Finally, we define

$$\upsilon(s_1, s_2) = \eta \times \delta_{\mathbf{V}}(s_1, s_2) + \mu \times \theta(s_1, s_2) \qquad (1)$$

where, η and μ can take values from the range [0,1] and these are used to configure υ for weighted distance and orthogonality. To evaluate if we need to partition a given system state-space P, we try find a subset $\mathbf{V}'_\alpha$ of $\mathbf{V}_\alpha$ such that

$$\sum_{\forall s_i, s_j \in S} \delta_{\mathbf{V}_\alpha - \mathbf{V}'_\alpha}(s_i, s_j) \leq \varDelta_{max} \qquad (2)$$

$$|\mathbf{V}'_\alpha| \leq f \qquad (3)$$

where $\varDelta_{max}$ is a user defined parameter that represents the maximum allowed representation error for the actionable variables and f represents the maximum number of actions that can be used to manage the system in the given partition. We employ a greedy approach for finding $\mathbf{V}'_\alpha$, i.e. we add the member of $\mathbf{V}_\alpha$ to $\mathbf{V}'_\alpha$ which causes the greatest reduction in the L.H.S. of the equation 2. We repeat the above process until the L.H.S. becomes lesser than $\varDelta_{max}$, at this point we look at the cardinality of the set $\mathbf{V}'_\alpha$ - if the cardinality is less than f we do not partition the system state-space, otherwise we proceed to partition the system state-space. The $\mathbf{V}'_\alpha$ so determined becomes the $\mathbf{V}_\alpha$

for the partition. We start by finding a pair of states s_1 and s_2 from the set of all such pairs contained in the set S such that $v(s_1, s_2)$ is maximized. The pair s_1 and s_2 acts as the seed for the two new system state sub-spaces S_1 and S_2 that will be created. We then iterate through the remaining operational states in the set S, adding the operational state s_i to S_1 if $\delta_{\mathbf{V}}(s_i, s_1) \leq \delta_{\mathbf{V}}(s_i, s_2)$. One can alternatively use the centroid of existing operational states in the evolving partitions to determine the membership. Once the two new partitions S_1 and S_2 have been created, we find the set $\mathbf{V}_\alpha$ for them using the greedy approach described above. If the criteria defined by Δ_{max} and f is not met by any partition then we repeat the above scheme for that partition. We now enumerate the advantages of the partitioning scheme for the purpose of enabling policy-driven self-management.

- *Simplifies Policy Learning*. Our approach intelligently reduces the space of possible actions that could be taken in response to an event. This greatly simplifies the process of correlating the events to actions for the purpose of determining ECA policies.
- *Assists in Problem Diagnosis*. The system might migrate through a series of system state ‘partitions’ before ending in an unacceptable state (e.g. SLA violation). The path followed by a system before a failure may contain information about the events that may have led to a failure, and can therefore assist in problem diagnosis and constructing complex policies.
- *Simplifies Problem Resolution*. If a system enters an unacceptable state during its operation then the model corresponding to the partition to which this unacceptable state belongs can possibly be used to arrive at a resolution to the problem.
- *Reducing Monitoring Overhead*. The partitions that are created by our algorithm allow us to ignore a subset of variables when the system is operating in that partition. This can potentially allow us to monitor such variables at reduced frequencies. However, we have not fully explored this possibility.

Once the system state-space has been partitioned we build a system *micro-model* corresponding to each partitioned sub-space. A system model in our framework consists of several micro-models each one of which models a sub-space of possible system states. The micro-model to be applied is determined based on the current state of the system. Since, we attempt to model only a small partition of the entire system state-space at a time we are able to build models even for systems with a very high number of variables. This makes our approach highly scalable. A similar approach was presented in [32], which made use of an ensemble of probabilistic models to detect SLO violations, and was shown to perform significantly better than the approach which used a single monolithic model to detect violations. The approach works by adding new models when the existing models do not accurately capture the current system behavior.

3.2 Building System Micro-Models

We want to create *micro-models* such that they can predict the the values for the variables in $\mathbf{V}_\alpha$ given the values for the variables in $\mathbf{V} - \mathbf{V}_\alpha$. Here we take advantage of the fact that our system state sub-spaces have a reduced dimension in terms of actionable variables. For all the variables in $\mathbf{V}_\alpha$ we exhaustively enumerate all the possible values and create a new variable $\mathbf{c}$ which can take values corresponding to such an exhaustive enumeration. For example, if $|\mathbf{V}_\alpha| = 2$ and each variable in $\mathbf{V}_\alpha$ can take 3 discrete values then $\mathbf{c}$ can take 9 distinct enumerated values. We assume that the actionable variables take discrete values and if the variable is continuous, one can use existing techniques to discretize continuous data (actually the Bayesian modeling techniques, which are referred to in this paper make use of such techniques). The system state space $\mathbf{V}$ can now be represented as $\{\mathbf{c}\} \cup (\mathbf{V} - \mathbf{V}_\alpha)$.

To find the best value from the variable $\mathbf{c}$ which helps translate the system to a desired state we resort to making using of probabilistic modeling techniques. We use a variant of Bayesian network [16] called the Tree Augmented Naive Bayes [15] or TANs to probabilistically model the system state-space. A Bayesian network is represented as an acyclic graph whose vertices encode random variables and the edges represent statistical dependence relations among the variables and local probability distributions for each variable given values of its parents. The main advantage of using a Bayesian network (or one of its variants) is that their representation provides and easy way to inspect the relationships between the involved variables. This allows an expert to embed his knowledge or the common wisdom into the self-management framework by proposing an initial model, which can be further refined using learning techniques. Furthermore, by simple inspection an expert can single out any faults in the learnt system model. Our choice for making use of TANs was driven by the fact that unrestricted forms of Bayesian network are computationally very costly to build as they need to evaluate all the dependencies amongst the set of random variables. A TAN, on the other hand allows only a tree structured dependence amongst the set of random variables (other than the class variable) and is therefore cheaper to build and has been shown to perform almost as well as the unrestricted version. A TAN model when used as a classifier is able to determine the following probability

$$p = Pr(\mathbf{x}|\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n) \tag{4}$$

for the set $\{\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n, \mathbf{x}\}$, from a given training set. The variable $\mathbf{x}$ assumes a special status in this equation and is called the class variable and the other variables are called the attributes.

To create the micro-model we designate the newly formed variable $\mathbf{c}$ as the class variable and the remaining variables, i.e. the set of variables in $\mathbf{V} - \mathbf{V}_\alpha$ are designated as attributes. The resulting micro-model is able to determine the following probability.

$$p = Pr(\mathbf{c}|\mathbf{V} - \mathbf{V}_\alpha) \tag{5}$$

The above equation determines the probability of the variable $\mathbf{c}$ taking a certain value given the values of the variables in the set $\mathbf{V} - \mathbf{V}_\alpha$. This procedure for achieving a desirable and feasible system state is as follows. Let, $\mathtt{V}^{current}$ represent the current system state and $\mathtt{V}_\phi^{goal}$ represent the new desired values for the set $\mathbf{V}_\phi$. To find the values of variables in $\mathbf{V}_\alpha$ that can possibly lead to the goal state, we create the set $\mathbf{V}' = \mathbf{V} - \mathbf{V}_\alpha$. We create an instance $\mathtt{V}^{goal}$ of the set $\mathbf{V}'$ by assigning the corresponding values from the set $\mathtt{V}^{current}$ and thereafter resetting the values corresponding to the set $\mathbf{V}_\phi$ using the values from $\mathtt{V}_\phi^{goal}$. We then use the instance $\mathtt{V}^{goal}$ to find the instance $\mathtt{c}^{goal}$ of variable $\mathbf{c}$ that maximizes Equation 5. The values of $\mathbf{V}_\alpha$ corresponding to $\mathtt{c}^{goal}$ so determined are used as the new values for actionable variables.

Discussion. Note that the state depicted by $\mathtt{V}^{goal}$ may not exist in a real system. This is because the variables that are contained in $\mathbf{V} - \mathbf{V}_\phi \cup \mathbf{V}_\alpha$ inherit their values from the state instance $\mathtt{V}^{current}$, and it may so happen that when the system translates to the new goal state, the values for variables other than the variables of interest and actionable variables may also change. However, experimental results presented in Section 4 show that the predicted values of $\mathbf{V}_\alpha$ are mostly able to achieve the goal state. This can be attributed to the fact that attribute discretization adds some degree of tolerance causing some smaller changes not to be reflected until they occur at the points where discretization partitions the continuous data space. Another important consideration for future work may be the consideration of the magnitude of change in the values of the variables in $\mathbf{V}_\alpha$, a solution that requires smaller change in magnitude may sometimes be preferred over the most probable solution.

3.3 Policy Learning, Adaptation & Confidence Attribute

The system state space model and the micro-model play a central role in supporting the task of policy learning. The high-level directives or goal statements are described over the set of variables contained in the set $\mathbf{V}_\phi$. For example, a high-level directive like `delay` $< 20msec$ can be used to learn the corresponding policy using the procedure described next. The framework instantiates a trigger for capturing `delay` $\geq 20msec$ which acts as the *event* in terms of policy. If at any time the *event* occurs the current system state $\mathtt{V}^{current}$ is used in conjunction with system sub-space micro-model to arrive at a corrective *action*. The event, the current system state (condition) and the corrective action is recorded as a policy. Due to space limitations we cannot give the full details of policy construction here. Interested readers may refer to an extended version [20] of this paper.

In a dynamic system the micro-models may evolve with time. This may cause some learned policies to become invalid with time because the corrective actions that were determined using an earlier version of the model may not be applicable any more. Policy adaptation requires periodic evaluation of the actions specified as part of the learned policies.

The confidence-attribute associated with each policy helps us to deal with the issue of administrator's *trust* in our self-management framework. The confidence-attribute for a policy is equal to the probability p determined using the Equation 5. The system administrator can declare a threshold value to have control over the policies that will be enforced. Only the policies with a confidence-attribute greater than the threshold value are autonomically enforced by the self-management framework.

3.4 Implementation

We have implemented the system state-space partitioning algorithm in C++, which takes as input a set of data which contains actual observed values from a running (or simulated) system and the index of variables which can be modified deterministically. The user also needs to provide values for the partitioning parameters η, μ, $\mathit{\Delta}_{max}$ and f as defined in Section 3.1. The output from the partitioning algorithm is the set of partitions and the corresponding index of variables which can be modified deterministically. The output is generated in the well-known C4.5 format to facilitate further processing.

We then make use of jBNC [17], a java based implementation for the Bayesian networks to build a TAN micro-model corresponding to each partition. The TAN is then used for finding corrective actions, policy adaptation, etc.

4 Experiments

Our goal was to study the suitability of our techniques in managing large enterprise-scale systems where a large number of variables can potentially affect the state of the system. In this section, we present our findings based on simulation experiments under a variety of workloads and operating conditions. Our techniques, for instance, were able to detect bottleneck nodes in our simulation of a PIDS-like middleware and were able to avoid several SLA violations that would have otherwise occurred. We start with a description and validation of the simulator testbed, which is followed by a brief description of the workload and evaluation metrics. We present our experimental results starting from Section 4.3.

4.1 Simulator Testbed

We wanted to evaluate our techniques for self-management using applications that are representative of the ones used by large enterprises. We evaluated the possibility of using well-known benchmarks and real-enterprise applications for putting our techniques to test. However, we soon realized that the applications available to us in our lab environment (like RUBiS [26] and an implementation of industrial middleware from Delta Technology [12]) were not instrumented well enough to sense and actuate a sufficiently large set of variables, and often changing any environment parameter (like maximum number of worker threads, number of MySQL connections, memory allocations and MySQL cache size) required restarting the application for the changes to take effect. Of course, in order to use our techniques, these systems could be enhanced to provide more monitoring and dynamically tunable parameters. Furthermore, it was not possible for us to make use of such applications for a large-scale (say 500 underlying nodes) evaluation of our techniques.

In order to overcome the problems mentioned above we decided to design a well instrumented simulator for simulating a large system implementing service oriented architecture (SOA). An implementation of SOA contains a set of services running on a distributed network of nodes that can be invoked by sending a message to the service, messages may or may not be generated as a result and if the messages are generated they may be forwarded to the source, to a sink or to some other service(s). The PIDS middleware described in Section 1.1 can be implemented as a SOA. Our SOA simulator consists of four main components - server, service, network-link and client. A server represents a processing facility with a limited number of cycles per second, a limited memory, connections to other servers and ability to throttle server frequency at the expense of more power. A service represents a software which accepts certain types of messages, possibly generates some messages in response and determines the server cycles that will be used to process a certain message type given the available memory. There may be more than one service running on a server and they may have different priorities. A network-link has an associated bandwidth, delay, and cost per unit of data

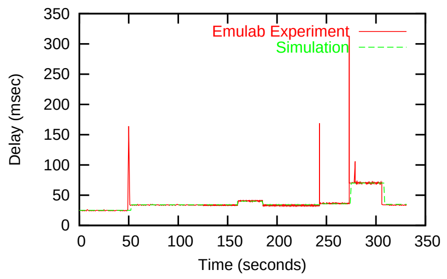

Fig. 2. Validation of the SOA simulator against an emulation at Emulab

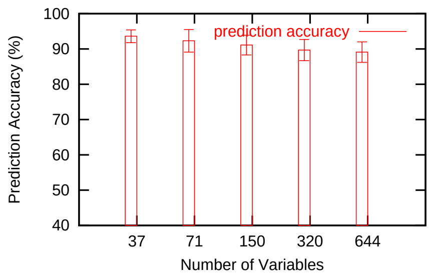

Fig. 3. iManage's scalability with number of observed system variables

transmitted. Finally, a client represents a source or a sink for the messages. An event source has a rate of generating events that can vary with time. A sink measures the incoming event-rate, the average delay for update propagation and the current delay measured over a recent window.

A simulation can be started by providing a network topology which instantiates the basic distributed network of servers and network-links, this is followed by addition of some services for processing messages and some clients. The simulation is run for a pre-specified amount of time and it dumps the state at configurable regular intervals. We make use of these state dumps to evaluate to build system models and then use these system models to arrive at policies for managing the simulated system. The network topologies for the simulations were generated using the GT-ITM [30] generator.

4.2 Simulator Validation

To validate our simulator we compared the measurement we got from our simulator with the ones we got from using the same experimental setup on Emulab [14] testbed. Our experimental setup consisted of a 13 node topology and an event processing graph that consisted of 3 sources, 2 services and 2 sinks. On the Emulab testbed we created the specified 13 node topology and then made use of the IFLOW middleware [22] to setup the event processing graph. We instantiated the same setup using our simulator. The services were configured to take a specified amount of time for processing the incoming events depending upon the incoming event type, and server load. We measured the event propagation delay between a source and a sink under a variety of variations which included events that take different processing times, variation in event rate from sources and change in event-size. The same event workload was used for both the Emulab testbed and the simulator. The measurement of event propagation delay for both the Emulab testbed and the simulator is shown in Figure 2. Our simulator was able to closely follow the behavior of real emulation testbed for the same experimental setup paving the way for simulations at a larger scale.

4.3 Microbenchmarks

We ran two simulations with 8 and 32 servers, each for 4 simulated hours. The two simulations dumped 71 and 227 state variables, respectively every 30 seconds. During the course of simulation we kept modifying the system conditions like the event rates from the sources, modifying the server frequencies, using alternate high or low-cost links to the destination and changing the priorities for various services running at a server. We also ran the simulation for another 20 minutes, dumping data at every 30 seconds to evaluate the accuracy of generated models. We collected 3 such sets of observations.

The first experiment focused on determining the effect of partitioning parameters Δ_{max} and f on the number of partitions that are created for a given system state-space and the average number of actionable variables that appear across the partitions. The results obtained by using one set of observation from the simulation described above are shown in Table 2. The table enumerates results from two set of simulations described above which generated 480 observed system states each. The results show that our techniques were able to significantly reduce the average number of actionable variables. For example our partitioning scheme was able to able to achieve a 90% reduction in number of actionable variables per partition for a system state-space with 31 variables. As far as the number of partitions are concerned, they are an important contributors towards the scalability of our techniques. However, a very high number of partitions may lead to partitions that may have a very sparse population leading to bad system models. The number of actionable variables are required to be low for our techniques to be effective as long as the manageability of the system partition is maintained, which can be controlled by setting a low value for Δ_{max}.

The next experiment was conducted to examine the effect of partitioning on the accuracy of the system models. To construct a single system state-space model for the set of observations collected earlier we proceeded as follows. We eliminated any $\mathbf{V}_\alpha$ from the set if it did not change its value during the simulation run. This reduced $|\mathbf{V}_\alpha|$ to ≈ 5 and ≈ 11 for the simulations with 71 and 227 variables, respectively. Notice that even with a discretization factor of 2 for each variable, the simulation with 227 variable had $2^{11} = 2048$ possible values for the variable $\mathbf{c}$. In the real world this translates to the confusion of which 'knobs' to turn to fix the system. Using the technique described in 3.2 we then constructed the single system models. For building micro-models corresponding to the partitioned sub-spaces we did not have to perform the pruning of the set $\mathbf{V}_\alpha$ as the partitioning algorithm takes care of removing the redundant members from the set $\mathbf{V}_\alpha$. The micro-models were then constructed for each of the partitioned sub-space. We used the generated models to predict the value of actionable variables given the value of other system variables from the test data set. Results reported in Table 3 show that the specialized micro-models work better than a single model at correctly predicting the values of variables in $\mathbf{V}_\alpha$.

Table 2. Effect of partitioning parameters on $|\mathbf{V}_\alpha|$ & number of partitions

		Original		**Partition**	
f	Δ_{max}	$\lvert\mathbf{V}\rvert$	$\lvert\mathbf{V}_\alpha\rvert$	avg $\lvert\mathbf{V}_\alpha\rvert$	partition count
3	0.1	71	11	2.8	5
4	0.2			4.0	3
3	0.1	227	31	2.7	7
4	0.2			3.8	5

Table 3. Comparison between the accuracies (in %age) of single and micro-models

Data Set	Model Type	
$\lvert\mathbf{V}_\alpha\rvert$	Single Model	Micro-Model
71	89.4 ± 2.8	92.3 ± 3.2
277	86.3 ± 1.9	90.7 ± 2.3

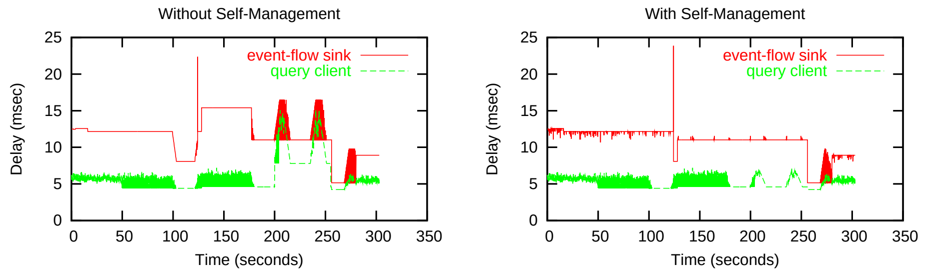

Fig. 4. Delays from the SOA simulator with and without self-management

To examine the effect of the number of observed system variables (i.e. $|\mathbf{V}|$) on the accuracy of predicting the right values for the variables in $\mathbf{V}_\alpha$ we conducted the following experiment. We used our SOA simulator to simulate systems that had 37, 71, 150, 320 and 644 variables that could be observed. Each system was simulated under varying workload conditions with appropriate corrective actions being taken at several points in the simulation. The time a system was simulated for was proportional to the number of variables being observed for that system. The smallest system with 37 variables was simulated for 1 hour simulation time. We used our techniques to build models for each of the systems and then used 10 minutes of generated test data to calculate the prediction accuracy corresponding to each model. We repeated the experiment 3 times. Results shown in Figure 3 show a slight decrease in prediction accuracy with the increase in number of observed system variables. However, the prediction accuracy only shows a linear trend in decrease as the number of variables are increase exponentially. We acknowledge that the results obtained may be highly dependent on the training set and the test data set that were generated during the simulation.

4.4 Evaluation of the Self-Management Framework

The next set of experiments was conducted to evaluate the end-to-end efficiency of our framework in managing large-scale systems. We study the impact of suggested policy actions and the confidence attribute on the end system metrics at runtime. The simulations were conducted for a system with 227 variables and consisted of 32 simulated server nodes.

The simulation setup consisted of an event-flow that contained 3 sources, 2 services and 1 sink, and 2 query response services which received a stream of queries from a co-located client. Each event-flow service was located on a separate server but shared the server with another query response service. The variables that could be modified included the priority of the event-flow service thread, the priority of the query service thread and the frequency of the server. The dynamic workload conditions were simulated by varying the event rates from the sources and the query clients. The metrics of interest included $\mathbf{delay}_{flow}$ and $\mathbf{delay}_{query}$. The goals for the simulation run were specified as $\mathbf{delay}_{flow} < 12.5msec$ and $\mathbf{delay}_{query} < 7.5msec$, and both the threads were assigned the same priority. Figure 4 shows the delay observed at the event-flow sink and at one of the query client with and without self-management. Our techniques were mostly able to avoid any violations of the specified goals. The confidence threshold for this experiment was set to 85.0%.

We next conducted an experiment using the above setup to examine any unwanted behavior that may happen due to low confidence-threshold. When the confidence threshold was reduced to 75.0%, we actually observed delays at the event-flow sink that were more than the delays observed without the self-management in place. Confidence thresholds even lower than 75.0% made the system behave erratically when self-management was turned on. This was corroborated by re-examination of some of our earlier data used for prediction accuracy experiments. 90.0% of the predictions that lead to false predictions had a confidence-attribute lesser than 65.0%. These findings can be attributed to the use of probabilistic models by our framework. A low-confidence threshold means that there is possibility that a certain other assignment of values to variables in $\mathbf{V}_\alpha$ also has a high probability of occurrence. This may lead to two assignments having almost the same probability of occurrence leading to a higher chance of erroneous choice of assignment by the system.

5 Related Work

Policy Research
There has been a lot work in the domain of using policies for simplifying the management tasks associated with system administration. Over the last decade, researchers, both in academia and industry, have focused on issues like policy specification languages [11], frameworks [5, 27] and toolkits [23]. The research presented in this paper builds on the work done in the above mentioned areas and is a logical next-step, as the focus is on applying the policy-research to the management intensive domain of enterprise-scale systems. The policy research in the domain of automated network management that deals with issues like security, access control and other associated management tasks [29, 25] justifies our stand on studying the impact and application of policy research to another rich domain. More recently, researchers have started evaluating the pros and cons of applying policy research for managing IT systems at large business enterprises [7]. This research, which is in its nascent stages, promises to provide systems that will manage themselves in accordance to high-level business goals [2]. The issues concerning human expertise and policy representation have also been explored in a recent paper [18].

Autonomic Computing & Self-Managing Systems
The task of implementing self-managing systems is a multi-step process in which policies can play an important role. Policies are a way to dictate the behavior of a self-managing system. This is in line with the vision of autonomic computing - 'to design computing system that can manage themselves given high-level objectives from administrators' - as described in [19]. There has been a lot of work in the domain of enabling self-management for a wide variety of systems. The SLA-based approach to manage systems has been explored by number of researchers [31]. In our prior work, we had focused on enabling self-management capabilities for distributed data stream systems [21, 22]. Some researchers have also explored the use of rule-based self-management approach for managing applications [4]. The use of utility-functions for self-management has also been explored in specific reference to event-based systems [6] and an interesting take on aggregate utility-functions is presented in [3]. It turns out that defining

utility-functions for enterprise-scale applications is a tough task because it may not be possible to mathematically model all the factors that can potentially affect the state of the enterprise system.

Bayesian Networks & Problem Diagnosis
Bayesian networks or the Belief networks have found applicability in a number of AI domains and they represent one of the best classification tools available to researchers. A tutorial on Bayesian networks is presented in [16]. Several specializations of the Bayesian networks have been proposed in literature the most important ones being the Naive Bayes [13] and the Tree Augmented Naive Bayes or the TANs [15]. In reference to applying Bayesian networks for modeling computer systems, a very innovative approach for correlating instrumentation data to system states is presented in [9]. This was later extended in [10] to develop signatures that could be used to more efficiently correlate the SLA violations that may occur in a system. More recent work in this domain makes use of an ensemble [32] of system models for the purpose of problem diagnosis. The work presented in this paper not only detects possible violations of higher level goals by the system but also suggests appropriate corrective actions to arrive at a solution for the problem.

6 Conclusions & Future Work

In this paper we described a system modeling framework that collects the system parameters and metrics into a unified abstraction, we call the system state-space, and identifies the actions that can be used to manage the system. To deal with complex system state-spaces, typical of enterprise-scale systems, we presented techniques that can be used to reduce the complexity and to more precisely model critical aspects of the system, and to more effectively develop policies. Additionally, iManage has capabilities for dealing with dynamic environment and for letting the administrator incorporate human knowledge and experience to refine the policies. Finally, the confidence-attribute associated with the policies learnt by iManage framework allows the administrator to fine-tune the enforcement of such policies. As part of the future work we are trying to address the issues related to monitoring overhead and making use of dynamic Bayesian networks to incorporate the consideration of time into the system model.

References

1. S. Agarwala and K. Schwan. Sysprof: Online distributed behavior diagnosis through fine-grain system monitoring. In *ICDCS*, 2006.
2. S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug. Autonomic self-optimization according to business objectives. In *ICAC*, 2004.
3. A. AuYoung, L. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *HPDC*, 2006.
4. V. Bhat, M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *ICAC*, 2006.
5. M. Bhide, A. Gupta, M. Joshi, M. Mohania, and S. Raman. Policy framework for autonomic data management. In *ICAC*, 2004.
6. S. Bhola, M. Astley, R. Saccone, and M. Ward. Utility-aware resource allocation in an event processing system. In *ICAC*, 2006.

7. Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, and R. E. Strom. Utility-driven management of availability in enterprise-scale information flows. In *Middleware*, 2006.
8. A. Chand, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: System Support for Controlling and Analyzing the Execution of Multi-tier Applications. In *Middleware*, 2005.
9. I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, 2004.
10. I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP '05*, 2005.
11. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *POLICY*, 2001.
12. Delta technology home. `http://www.deltadt.com/`, as viewed on 04/22/2007.
13. P. Domingos and M. J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.
14. Emulab - network emulation testbed home. `http://www.emulab.net/`, as viewed on 04/22/2007.
15. N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.
16. D. Heckerman. A tutorial on learning with bayesian networks. Technical report, Microsoft Research, Redmond, Washington, 1995.
17. jBNC: Bayesian network classifier toolbox. `http://jbnc.sourceforge.net/`, as viewed on 04/22/2007.
18. E. Kandogan, C. Campbell, P. Khooshabeh, J. Bailey, and P. Maglio. Policy-based management of an e-commerce business simulation: An experimental study. In *ICAC*, 2006.
19. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
20. V. Kumar, B. F. Cooper, G. Eisenhauer, and K. Schwan. iManage: Policy-driven self-management for enterprise-scale systems. *Extended Version.* `http://www.cc.gatech.edu/~vibhore/inTransit/policy-ext.pdf`.
21. V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *ICAC*, 2005.
22. V. Kumar et al. Implementing diverse messaging models with self-managing properties using iflow. In *ICAC*, 2006.
23. L. Lymberopoulos, E. C. Lupu, and M. S. Sloman. Ponder policy implementation and validation in a cim and differentiated services framework. In *NOMS*, 2004.
24. P. Mahalanobis. On the generalised distance in statistics. In *Proc. of the National Institute of Science of India 12*, 1936.
25. N. Minsky. A scalable mechanism for communal access control. In *NCAC*, 2005.
26. RUBiS - home page. `http://rubis.objectweb.org/`, as viewed on 04/22/2007.
27. Web services policy framework. `http://www-128.ibm.com/developerworks/library/specification/ws-polfram/`, as viewed on 04/22/2007.
28. G. Weikum, A. Mnkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB*, 2002.
29. M. J. Wright. Using policies for effective network management. *International Journal of Network Management*, 9(2):118–125, 1999.
30. E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, 1996.
31. L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *ICSOC '04*, 2004.
32. S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *DSN*, 2005.