A Classification-Based Approach to Policy Refinement

Yathiraj B. Udupi
Department of Computer Science
North Carolina State University
Raleigh, NC, 27695-8206, USA
ybudupi@ncsu.edu

Akhil Sahai, Sharad Singhal
UIM, ESSL
Hewlett Packard Laboratories
Palo Alto, CA, 94304, USA
{akhil.sahai, sharad.singhal}@hp.com

***Abstract*— Systems are typically designed based on certain high-level goals, such as performance and availability. On the other hand, during operation, usually only low level metrics (e.g., CPU utilization) are measured, and system administrators or experts use domain knowledge to implicitly map bounds on these lower level metrics such that the high-level performance goals are met. The objective of this research is to create an automated and domain independent approach to derive policy bounds on the low level metrics such that the high level goals are met. These low-level policies may be also be used for monitoring the system for goal assessment purposes. The refinement is carried out using a combination of data classification and test & development approaches. A system is deployed within a test-and-development environment and a data set containing values of low-level metrics is collected by placing appropriate workloads on the system. The policy bounds are derived by applying classification techniques on this dataset. The classification rules are further refined using statistical distributions to arrive at certain low level rules that are useful for system monitoring and to check the system health when it is deployed and running. We show the validity of our approach for an e-commerce auctioning system (RubiS).**

Keywords- policy refinement, system design, monitoring, SLA

I. INTRODUCTION

Policies can be specified for system design and to control the system behavior. Currently system administrators and other experts are provided with certain high-level goals or policies while designing systems. The experts normally apply their domain knowledge and best-practice rules or policies to design the system. These policies can be very complicated and designers depend on their past experience to arrive at good policies. During system operation, however, normally low-level metrics are monitored and maintained within bounds specified as low-level policies by system operators. Again, system operators use past experience with specific applications to determine the policy bounds necessary on these metrics to ensure that the overall system goals are met.

In Quartermaster, Singhal et al., visualized policy as the entire set of strict (enforced) constraints and desirable directives that control the behavior of a target entity towards achieving a goal [2]. By formulating policies as constraints on system behavior (as opposed to conditions that arise as a result of system operation), Quartermaster applied the concept of policy to system management in a different manner. Quartermaster considered the managed system as a set of related entities. Each entity was characterized by a set of attributes and values taken by those attributes, along with actions that were available to the management system to change the attribute values. Policies were then defined as constraints that limit the values that the attributes may take in order for the entity (and the system composed from those entities) to behave within an acceptable range.

Quartermaster applied this approach to design and configuration policies [4]. It used policies for automatic selection of resources and for generation of a design specification [1, 5], that could then be deployed. However, it is also important to monitor the system in order to ensure that the system behaves as per the design goals. The monitoring system can help assess compliance with the overall design goals and also to perform a proactive goal assessment.

In this paper, we address the question of generating low-level policies that can be used for monitoring the application as part of the design process. The end-user specifies high-level goals and during the test-and-development phase, the management system generates the low-level policies that are relevant and that may be used to proactively assess the system behavior, as well as used for subsequent design.

A typical SLA may specify various service requirements that include metrics such as availability, response time, throughput, security, and so on. A SLA is composed of a set of Service Level Objectives (SLOs). A SLO specifies what constitutes an acceptable service and is a combination of one or more constraints on component measurements. A set of example high-level SLOs defining the performance aspects of an SLA are listed below:

- *"service response time < 85 ms between 8 AM and 5PM M-F"*
- *"service availability > 99.95% between the hours of 8AM and 8PM"*
- *"overall compliance > 97% over a period of a calendar month"*
- *"transaction workload < 100 transaction/sec"*

Any system design involves many components, each potentially affecting the overall performance of the system. Hence any high level constraint specified on the system potentially relates to all low-level system components. By

defining the low-level policies for "healthy" ranges on the various low-level metrics during system design, it becomes easier for the system operator to proactively avoid potentially costly SLA violations without the need to guess proper ranges for the low-level metrics. Hence there is a need for deriving low-level policies from the high level SLAs.

The proposed approach provides an automated and domain independent approach to derive these low-level policies given the high-level SLA goals. We have to identify relevant low-level attributes as well as policies that define ranges within which those attributes have to be maintained for system health. The key contribution here is to identify the relevant low-level attributes and create policies that define bounds on them based on high level policies. The policy refinement is carried out using a combination of data classification and test & development approach. A system is deployed and data is collected on both the low level metrics and the high-level SLOs. Policies are then derived by applying classification techniques on this dataset. The classification rules are further refined using statistical distributions to arrive at certain low level rules that are useful both for checking system health when it is deployed and running, as well as for creating subsequent configurations.

The rest of this document is organized as follows. Section 2 describes in detail our classification-based approach to policy refinement. Section 3 provides the experimentation details and the lists collected with the help of an example scenario. Section 4 describes the related literature on policy refinement and monitoring policies and their validation. Section 5 concludes with a summary of our contributions and future work.

II. POLICY REFINEMENT: A CLASSIFICATION-BASED APPROACH

Policy refinement involves breaking the high-level SLA goals into smaller low-level policies that feature several low level system attributes and metrics. A policy specifying a constraint at a high level can be affected by several low level metrics and the challenge is to optimally constrain the low level metrics just enough to satisfy the high level metrics. For example, a high-level policy rule can specify that "*service response time < 85 ms*". This can be affected by several metrics at various tier levels in the system. Policies can be specified to constrain each of the low level metrics that affect the high level metric. But the actual thresholds for the various low level metrics can vary in different scenarios and for different possible workloads specified. The policy refinement mechanism should identify all the important low-level attributes and their policies, and refine them to specify the most optimal constraint thresholds for the relevant metrics. We explain the detailed mechanisms of policy refinement applied in this work below. The main steps are listed below and explained in detail in the following subsections and summarized in Figure 1.

1. Test and Development Phase
2. Classification Phase
3. Policy Derivation and Refinement Phase

A. *Testing and Development Phase: Data Collection and Preprocessing*

An ad hoc system configuration is first created on the basis of the given high-level SLA goals. Now we use a Test and Development environment to record values for the low-level system attributes by placing workloads on the system that are spread around the ranges of the target workload.

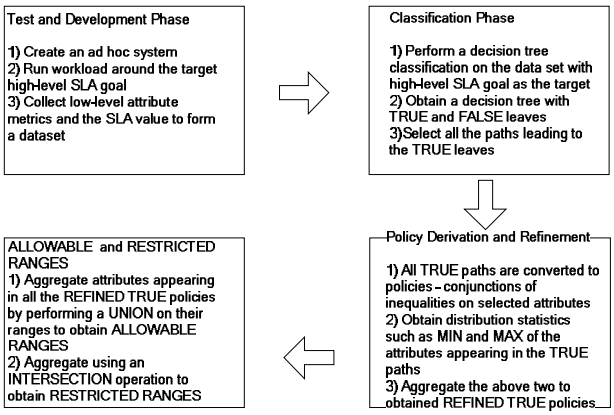

Figure 1 The policy refinement approach

Next, we preprocess the data for classification. Depending on the observed variation, a few metrics can be eliminated if they do not significantly vary for different workloads. The elimination process can be performed by looking at the distribution statistics of the variables, and is later explained with example results in Section III.

B. *Classification Phase*

This phase involves applying a classification algorithm on the dataset collected above, and deriving the policies that are useful for our purposes. In the dataset collected above, for classification purposes we include the target variable computed from the evaluation of the high-level SLA metrics. So the target variable is a Boolean value taking either TRUE or FALSE as the possible values depending on whether the high-level SLA goals are satisfied or not. Classification techniques such as decision tree classification methods are applied on the dataset for the given target. All the TRUE rules are collected which are relevant for our purposes. For example, in the case of the decision tree approach for classification, all the paths leading to TRUE leaves provide us the required rules on low-level metrics.

C. *Policy Derivation and Refinement Phase*

This phase includes the processes of deriving policies from the output of the classification phase. In the case of a decision tree approach for classification, we derive policies from all the TRUE paths, i.e., the paths leading to TRUE leaves. These TRUE policies are a conjunction of inequalities on various attributes that are picked by the classification phase. A refinement strategy that is applied at this level uses the distribution statistics of the attributes on these TRUE paths. For all attributes in the TRUE tuples (tuples in the dataset that correspond to a TRUE high-level

SLA value), the distributions statistics such MINIMUM and MAXIMUM values are computed. The inequalities of attributes that appear on the TRUE policies are further refined by appending the MINIMUM and MAXIMUM values giving definite bounds for the attributes, resulting in the required TRUE REFINED policies. This process results in a set of TRUE REFINED policies, which can be used for monitoring system health as as for designing subsequent configurations.

D. ALLOWABLE and RESTRICTED Ranges

We provide certain further refined categories of policies by aggregating the different policies generated to arrive at rules on individual attributes. We arrive at two kinds of AGGREGATE RANGES for the attributes:

1. ALLOWABLE RANGES: The allowable ranges of attributes are derived by considering all the TRUE refined policies, and aggregating the inequalities of the attributes from each of the policies they appear in, by performing a UNION operation on their individual ranges. This composite range forms the ALLOWABLE RANGE for an attribute.
2. RESTRICTED RANGES: They are computed in a similar fashion as the Allowable ranges, but an INTERSECTION of all the individual ranges of attributes from different policies is performed to arrive at a single restricted range for the attribute.

III. EXPERIMENTATION AND RESULTS

We performed a workload analysis using the ***RUBIS*** [3] testbed – an auction platform implemented as a multi-tier application. For the current purposes, we considered a two-tier application having an apache web server tier and a mysql database tier. The RUBIS framework offers two kinds of workload mixes – a browsing mix (read-only interactions), and a bidding mix (includes 15% read-write). We can vary the workload by specifying different numbers of clients running of RUBIS, and by specifying different kinds of workload mixes. We considered three different high-level SLA goal parameters in our experiments namely: No. of Clients, Response Time, and Throughput. For the purposes of the experiments illustrated in the example below, the following SLA goals were considered: No. of Clients < = 400, Response Time < = 24 ms, and Throughput >= 17 req/sec.

We ran experiments with workloads around the above specified high-level SLA goal, varying the number of clients from 100 to 900, for four different mixes. The example results shown in the following sections are for an experiment with a bidding mix. We collected the low-level system metrics such as CPU utilization, Net_Stats (packets info), IO_Stats (bytes read and written), and the mysqladmin extended-status metrics (a total of 33 metrics). We can potentially collect many other metrics including some metrics from the apache server and other system metrics.

Table 1 provides a list of the 33 low-level metrics that we thought were relevant and that significantly affect the system behavior. The first 18 from wwwCPU till dbRxBR are the metrics that include CPU utilization, net_stats, and the io_stats of both the apache web server and mysql database tiers. The remaining 15 from Aborted_clients_psec to open_tables are a few important metrics selected from the list of mysqladmin extended-status returned values.

wwwCPU	dbCPU	wwwRBR
dbRBR	wwwRCR	dbRCR
wwwWCR	dbWCR	wwwWBR
dbWCR	wwwTxBR	dbTxBR
wwwTxPR	dbTxPR	wwwRxPR
dbRxPR	wwwRxBR	dbRxBR
Aborted_clients_ psec	Aborted_connect_ psc	Bytes_Rcvd_psec
Bytes_Sent_psec	Connections_psec	Cache_hit_ratio
Key_W_key_W_ request_ratio	Max_used_connectns	Open_files
Queries_psec	Threads_connected	Threads_created
Threads_running	Tmp_tables_created_ ps	Open_tables

Table 1 Low-level metrics collected

A. An Example Policy Refinement Scenario

We illustrate the policy derivation and refinement mechanism using an example scenario. For the purposes of this example, we consider a dataset containing the tuples and the corresponding high-level target SLA boolean values for the first 18 metrics. We perform a decision tree classification on this dataset using the Random Tree classification (RT) as well as the J48 algorithm. The results are shown in Figures 2 and 3.

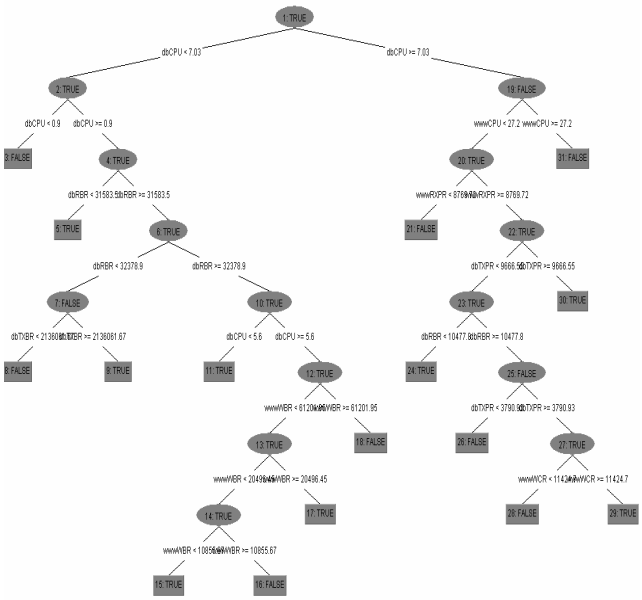

Figure 2 A decision tree generated using Random Tree classification algorithm

We observe (Figure 2) that for this example, the RT algorithm has resulted in a tree with 16 leaves out of which 8

are TRUE leaves and the rest are FALSE leaves. We are interested in the paths starting from the root node till the TRUE leaves. This results in eight TRUE policies that are conjunctions of inequalities consisting of certain relevant low-level system attributes. We compute the distribution statistics such as the MINIMUM and the MAXIMUM values of the attributes that appear in the TRUE policies, and append them in the TRUE policies to get the REFINED TRUE policies.

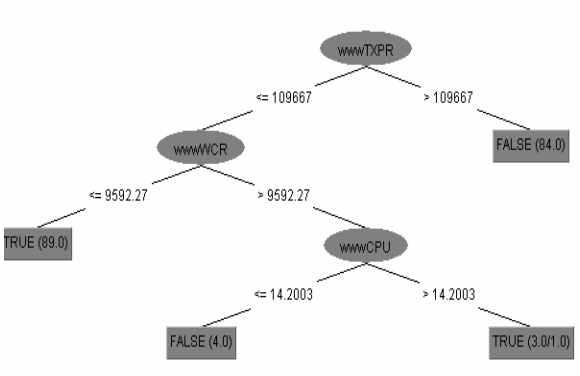

Figure 3 A decision tree generated using J48 algorithm

For example, consider the TRUE path leading to Leaf #29. After the first iteration we arrive at this UNREFINED TRUE policy:

"(dbCPU >= 7.03) AND (wwwCPU < 27.2) AND (wwwRXPR >= 8769.72) AND (dbTXPR < 9666.55) AND RBR >= 10477.8) AND (dbTXPR >= 3790.93) AND (wwwWCR >= 11424.7) => Target = TRUE"

In the next iteration, we combine attributes that appear more than once in the path and also append their distribution statistics, to get the REFINED TRUE policy:

"(7.03 <= dbCPU <= 12.434) && (3.61269 <= wwwCPU < 27.2) && (8769.72 <= wwwRXPR <= 183002.0) && (3790.93 <= dbTXPR < 9666.55) && (10477.8 <= dbRBR <= 154804.0) && (11424.7 <= wwwWCR <= 12352.3) => Target = TRUE"

By looking this policy and the other TRUE REFINED policies, we observe that we arrive at constraint rules that specify certain inequalities on the potential values those low-level metrics can take while the system is running healthy and the high-level SLA goals are being satisfied. From the set of all refined TRUE policies we can arrive at ALLOWABLE RANGES (by performing an UNION), and RESTRICTED RANGES (by performing an INTERSECTION) for all the attributes that appear in these policies. For example, Table 2 provides ALLOWABLE RANGES for two datasets (one with the first 18 metrics, and the other with the remaining 15 metrics from the list of 33 metrics provided earlier).

MYSQL (15 metrics dataset)	**CPU, IO, Net_Stats (18 metrics data set**
Tmp_tables_ created_psec : [0.01, 0.67]	**dbTXBR :** [2136061.67, 3051843.91]
key_reads_key_ read_req_ratio : [0.05, 0.24]	**wwwCPU :** [3.61269, 27.2)
key_write_key_ write_req_ratio : [0.0, 1.0]	**dbCPU :** [1.08, 12.434]
Threads_created : [1.0,31.0]	**wwwRXPR :** [8769.72, 183002.0]
Open_tables : [6.0, 48.0]	**dbRBR :** [0.0, 154804.0]
Bytes_received_ psec : [0.62, 70.67]	**dbTXPR :** [1084.2, 31681.5]
Open_files : [13.0, 62.0]	**wwwWBR :** [6917.29, 10855.67) U [20496.45, 61201.95)
Connections_psec : [0.03, 0.38)	**wwwWCR :** [11424.7, 12352.3]

Table 2 ALLOWABLE Ranges created using the RT classifier

B. Classification Algorithms

We tried different decision tree classification algorithms to arrive at different sets of policies. The factors we used to evaluate these classification approaches for our purposes are mainly the number of TRUE policies selected, and the number of attributes appearing in these TRUE policies. In addition to the above, the standard classification algorithm evaluation techniques such as number of incorrectly classified instances, error rate and so on can be used. The three decision tree algorithms we selected are J48, REP, and Random Tree. Figure 3 shows the decision tree computed on the same data set as that of Figure 2, but using the J48 algorithm. Policies using the J48 algorithm can be derived from the decision tree in a manner similar to the RT algorithm.

Algorithm	No. of Metrics	No. of TRUE Policies	Number of attributes picked
J48	33	2	3
	18	2	3
	15	2	3
Random Tree	33	4	6
	18	8	8
	15	8	9

Table 3 Number of attributes bounded by different classification algorithms

Table 3 illustrates the number of TRUE policies and the number of selected attributes for the dataset collected in the

above example. We observe that the Random Tree algorithm provides a larger number of TRUE policies and the number of attributes picked. While at first glance, this may suggest that the Random Tree algorithm is more suitable, this is not necessarily true without taking into account the error characteristics of the algorithms. For this analysis, we segmented the data into a training set (to generate the decision trees) and a test set (for testing performance of the classifiers). We used a 75-25 split and a 80-20 split of the data into the training and test sets for this analysis. Table 4 shows the results of the analysis.

Random Tree Classifier		
	75%-25% data split	**80%-20% data split**
Number of instances	45	36
Correct classifications	41 (91%)	32 (89%)
Incorrect classifications	4 (9%)	4 (11%)
Kappa statistics	0.82	0.78
J48 Classifier		
	75%-25% data split	**80%-20% data split**
Number of instances	45	36
Correct classifications	44 (98%)	35 (97%)
Incorrect classifications	1 (2%)	1 (3%)
Kappa statistics	0.96	0.94

Table 4 Classifier performance observed for the different classification algorithms

We find that the random tree classifier has a much higher error rate as compared to J48 decision tree classifier. Since J48 classifier provides better classification and lower error rates we find it to be a better classifier for policy refinement because even though it generates lesser number of policies, they will be more reliable.

The effects and relevance of these policies can be further evaluated by actually configuring a system based on the policies generated and testing for their consistency, and will be considered as future work.

C. Distribution Correlation Study

We observe the distribution characteristics of some of the variables by varying the number of clients. These distributions help us in two ways. First we can eliminate some of the attributes that do not vary with workload, and hence are not important for the purposes of the required policies. After eliminating the non-varying attributes we arrived at a list of 33 metrics. The second use of studying the distributions helps us see the patterns of attributes value variation.

From Figure 4 we see that the CPU utilization of apache web server and mysql database server are uniformly increasing with the number of users. The 9 regions in the graph correspond to the 9 different workloads with number of users being 100, 200, 300, and so on. As can be seen in the Figure, CPU utilization of both the web tier and the database tier increases with the number of number of clients. The tested classification algorithms also select the CPU utilization as an important metric to observe.

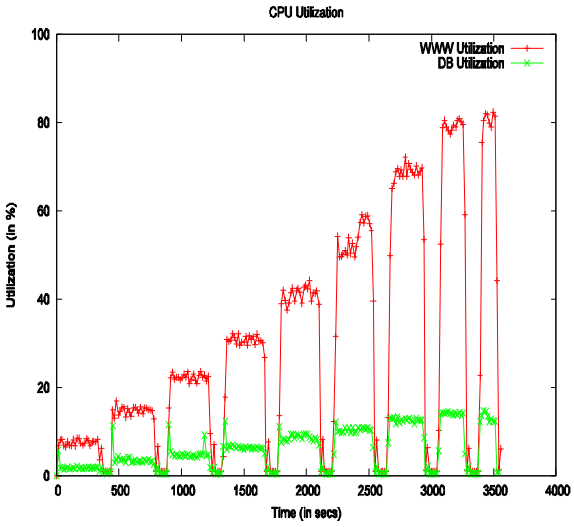

Figure 4 Plot of CPU Utilization of WWW server and DB server for varying workload

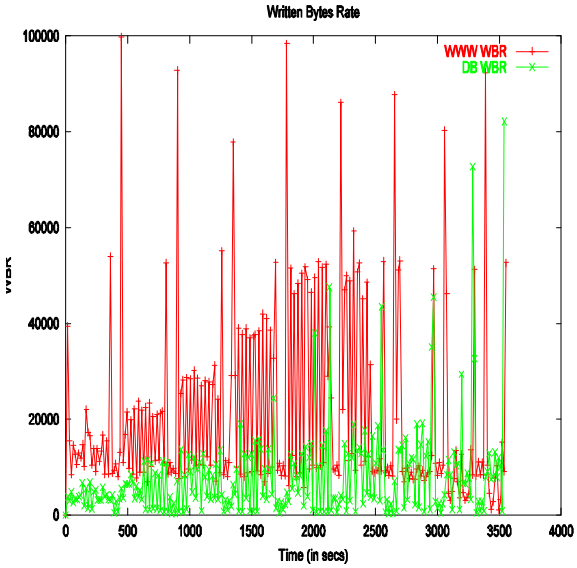

Figure 5 Plot of Written IO Bytes for WWW and DB servers for varying workload

Figure 6 shows WBR (Written Bytes Rate) as workloads are changed from 100 to 900 clients. There does not appear

to be a strong correlation of this metric with workload, and it is eliminated by the decision tree algorithm. Similarly, we also observed that attributes that do not correlate with one another also do not appear together in the generated policies. Hence we infer that the refined policies obtained using our approach are better indicators of SLA performance than policies that create thresholds on the distribution statistics of the low-level attributes independently of such correlations.

This correlation study can also be applied to perform a "bottleneck analysis" by running a bigger system for a wider range of workload and will be considered as future work.

IV. Related work

The area of policy refinement has become very important recently with a growing interest in policy-based approaches to systems management. Darimont and Lamsweerde studied formal refinement patterns for a goal-driven requirements elaboration using the KAOS language [6]. Several formal patterns were identified for refining goals into subgoals. These patterns can be reused from a library structured according to weakening/ strengthening relationships among patterns.

These formal refinement patterns became the basis for many techniques of policy refinement involving temporal logic and event calculus approaches [7, 8]. Policy refinement has been studied on several domains and researchers have considered a domain-knowledge based refinement. Bandara and others study refinement on DiffServ QoS management domain [9], while Albuquerque and others have studied policy modeling and refinement for network security systems domain [10]. In contrast, the proposed approach considers a test and development and a classification-based approach to policy refinement, and we present an automated, generic approach of refinement mainly for performance related goals.

Feather and others integrated the KAOS goal-driven specification methodology and the FLEA runtime event-monitoring system [11] to identify runtime deviations from requirements specifications [12]. Their approach is capable of monitoring the system requirements at runtime to reconcile the requirements and the system's runtime behavior. The proposed approach provides runtime monitoring policies for the system to detect any anomalous behavior and to perform a proactive goal assessment.

V. Conclusion

We have presented an automated and a non-domain specific approach, which we believe is a generalizable approach, to derive a set of low-level policies from the given high-level goals. This approach is a combination of a test and dev methodology and a classification-based policy derivation and refinement mechanism which provides a refined set of policies that can be used for generating system design specifications and system monitoring for a proactive goal assessment. We observe that not all low-level attributes are selected by the classification algorithm, and hence we conclude that only a few selected attributes correlate with each other and hence form relevant policies for our purposes. This refinement is definitely better than providing just the distribution statistics such MINIMUM and MAXIMUM for all the low-level attributes. There are certain problems that could be associated with this approach. If the ad hoc system that we begin with is not a big enough system, for certain workload, we can observe that certain low-level attributes may become the bottleneck. To avoid such situations we propose to start our test and dev phase with a larger ad hoc system. While using such large systems our approach can be used to detect bottlenecking attributes in addition to deriving the refined TRUE policies. Currently, this approach works best for performance related high-level goals; while we believe that this approach is generalizable to other kinds of goals.

As future work, we also plan to validate our approach by redesigning the system based on the derived low-level policies and testing the results from the new system. This way we can see if the derived low-level policies are consistent. We plan to further evaluate different classification techniques to derive the best set of refined low-level policies. There is another venue, where our approach can be beneficial, namely the root-cause detection and analysis. We plan to study and evaluate the potential use of our approach to see if the derived policies can be applied for a root-cause analysis in the case of a failure of the high-level goals.

Acknowledgment

We would like to express many thanks to Mustafa Uysal, Ira Cohen, Pradeep Padala, Xiaoyun Zhu, Vijay Machiraju, Kumar Goswami, Rich Friedrich, Subu Iyer, Yuan Chen, and others at HP Labs for their continuous feedback and guidance in this project.

References

[1] Lyle Ramshaw, Akhil Sahai, Jim Saxe, and Sharad Singhal. Cauldron: A Policy-Based Design Tool. In *Proceedings of the 7th IEEE International Workshop on Policies for Distributed Systems and hNetworks (POLICY)*. June 2006. To appear.

[2] S. Singhal et al. Quartermaster: A Resource Utility System. Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management, Nice, France, May, 2005. pp 265—278.

[3] RUBIS Project. URL: http://rubis.objectweb.org/.

[4] Akhil Sahai, Sharad Singhal, Vijay Machiraju, Rajeev Joshi. Automated Generation of Resource Configurations through Policies. In *HP-Labs Report HPL-2004-55*. 2004.

[5] Akhil Sahai, Sharad Singhal, Rajeev Joshi, Vijay Machiraju. Automated Policy-Based Resource Construction in Utility Computing Environments. In *HP-Labs Report HPL-2003-176*. 2003.

[6] Robert Darimont and Axel van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *Proceedings of SIGSOFT, CA, USA*. 1996.

[7] Arosha K. Bandara, Emil C. Lupu, Jonathan Moffett and Alessandra Russo. A Goal-based Approach to Policy Refinement. In *Proceedings of POLICY*, 2004.

[8] Javier Rubio-Loyola, Joan Serrat, Marinos Charalambides, Paris Flegkas, George Pavlou, and Alberto L. Lafuente. Using Linear Temporal Model Checking for Goal-oriented Policy Refinement Frameworks. In *Proceedings of POLICY*, 2005.

[9] Arosha Bandara, Emil Lupu, Alessandro Russo, Paris Flegkas, Marinos Charalambides, and George Pavlou. Policy Refinement for DiffServ Quality of Service Management. In *Proceedings of IEEE/IFIP Integrated Management Symposium (IM'2005)*, Nice, France, 2005.

[10] Joao Porto de Albuquerque, Heiko Krumn, and Paulo Licio de Geus. Policy Modeling and Refinement for Network Security Systems. In *Proceedings of POLICY*, 2005.

[11] D. Cohen, M.S. Feather, K. Narayanaswamy, and S. Fickas. Automatic Monitoring of Software Requirements. In *Proceedings of 19th International Conference on Software Engineering*, Boston, May 1997.

[12] M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of IWSSD9*, Isobe, Japan, April