Automatic Policy Rule Extraction for Configuration Management

Ron Banner*, Omer Barkol*, Ruth Bergman*, Shahar Golan*, Yuval Carmel†, Ido IshHurwitz†, and Oded Zilinsky†

*HP Labs, Haifa, Israel
Email: omer.barkol@hp.com
†HP Software, Yahud, Israel
Email: yuval.carmel@hp.com

Abstract—We propose a new IT automation technology for configuration management: automatic baseline policy extraction out of the Configuration Management Data Base (CMDB). Whereas authoring a configuration policy rule manually is time consuming and unlikely to realize the actual state of the configurations in the overall organization, this new approach summarizes the de-facto policies from the data. IT staff, instead of authoring the policy rule, is required to simply validate the automatically extracted policy. Our technology applies data-mining to organization’s configuration assets in the CMDB, and automatically identifies repeating structures of compound configurations. Based on these repeating structures, we build policy rules for compound configuration items. The heart of our technique is a new distance measure we introduce between the configuration assets, whose computation is reduced to a minimum-cost flow problem.

I. INTRODUCTION

Configuration Management practices in large IT-organizations are moving toward a policy-driven process, in which IT assets are managed uniformly throughout the organization. New products from HP’s Business Technology Optimization (BTO) business unit provide technology support for this process. These products, in conjunction with configuration item data in the CMDB, allow a practitioner to author policy rules for composite configuration items, after which the tool aids the user to analyze and monitor the organization. The practitioner can now analyze the percentage of the organizational assets that adhere to the policy, and find non-conforming configuration items. It can suggest changes to such non-conforming assets, or automatically initiate these changes. Clearly, once a policy is in place, new configuration management tools hold great power. Authoring such policies, however, is a daunting task. IT practitioners typically have responsibility to a specific set of configuration items, and, thereby, a limited view of the overall organization. No individual knows how configuration items are managed throughout the organization. As often occurs in practice, there is a risk with a tool like this that no-one will fill in the content, and, hence, the organization will not enjoy the benefits that the tool can provide. We propose to finesse the issue of policy authoring, by mining the organization’s configuration item data to automatically extract policy rules. Practitioners can deploy this tool and quickly get most of the configuration items mapped according to the different standards. this capability can also help in the use case of acquisitions or on boarding new clients for MSP’s.

The technical challenge we face is finding patterns of Composite Configuration Items (CI). We can coarsely break the problem down to three main sub-tasks:

• Computing the similarity between composite CIs
• Finding frequent patterns of CIs, and
• Generating a baseline policy.

Composite CIs are comprised of tree structures. Thus, we approach each of the above sub-problems as a problem on trees, and leverage prior work. The problem of computing similarity between CIs becomes similarity between trees, which is commonly studied in the setting of tree edit distance algorithms. Tree edit algorithms have been used to solve problems in molecular biology [21], XML document processing [13], [19], [14] and other disciplines. The prevalent definition of edit distance for labeled ordered trees was proposed by [17], and allows three edit operations on nodes - delete, insert, and relabel. For unordered trees the problem is known to be NP-hard [3]. For ordered trees, on the other hand, polynomial algorithms exist, based on dynamic programming techniques [17], [23]. Several researchers have identified restrictions to this definition of edit distance [21], [22], [10], [16]. CI similarity represents a unique set of constraints for tree-editing:

• To preserve CI structure, delete and insert operations do not apply to single nodes, rather they apply to complete sub-trees. For example, in Figure 1 an ejbmodule must be the child of an j2eeapplication. One cannot delete the j2eeapplication and add the ejbmodule as a child to the j2eedomain - the parent of the j2eeapplication.
• It is possible to change some attributes of a CI in a relabel operation, but not to change its type. Thus the distance computation between individual nodes must compare attributes of the CIs.
• CIs are unordered.
• The match between children of two CIs is not one-to-one. For example, a j2eedomain may be comprised of any number of j2eeapplications. We would not want to consider two j2eedomains very different if one contains five j2eeapplications, while the other contains fifty. Thus,
multiple children on one side may be mapped to a single child on the other side, and vice versa. On the other hand an NT server with one CPU is very different from an NT with four CPUs. Thus, there is a penalty on multiple assignments, which depends on the CI type. These constraints guide the design of our CI edit distance measure. The constraints on the delete and insert operations enable us to utilize a top-down methodology for computing the edit distance similarly to [16]. On the other hand, we cannot employ dynamic programming to match between child nodes, because it assumes a ordered, one-to-one match. Instead, we define a multiple-assignment, which is new to the best of our knowledge. We reduce this assignment to a minimum cost flow problem, which we solve using a successive shortest path algorithm in polynomial time [15]. The complete tree edit distance is also polynomial.

To self-organize a configuration, we want to find frequent patterns of CIs. Again, since CIs are trees, we need an algorithm for frequent tree mining. Such algorithms [5], [1], [20], [12], [11] search for repeating subtree structures in an input collection of trees. These algorithms vary in the restrictions that the repeating structure must adhere to, and in the type of trees that are searched. For mining configuration items, we are interested in a particular tree mining scenario. In addition to the above constraints the patterns we seek are similar, but not identical. Our approach for finding CI patterns, therefore, is to compute distances using the CI edit-distance, and then to compute clusters based on the computed distance. Many efficient non-parametric clustering algorithms may be used [7]. We stress that we compare the distances between all the composite CIs, including ones that are sub-trees within other composite CIs. So, if one can view the set of composite CIs that are given as input as a forest, we regard the distance between every two sub-trees in that forest. A cluster of composite CIs at the root level illuminate configuration policies. The CI clusters of internal CIs represent the prevalent patterns of these policies.

The final problem this work addresses is, given a set of composite CIs, which we know to exemplify a particular policy, generate a baseline policy. Note that the input set of CIs may be computed by the the CI clustering algorithm, or it may be manually-selected by a user. To generate a baseline policy, we collect statistics about each CI pattern. We, then, construct a policy by adding one pattern at a time in a greedy manner, while making sure that the policy adequately covers the input set of CIs.

This paper is organized as follows. Section II defines required terms and notation. Section III overviews our approach to automatic configuration baseline policy extraction, which, as said, is comprised of the three stages: Section IV describes the distance computation for composite configuration items, Section V summarizes a pre-processing step in which statistics are gathered and parameters inferred. Section VI describes the clustering to find frequent pattern algorithm of those CIs, and Section VII describes the generation of a baseline policy from these frequent patterns. Finally, we discuss experimental results and conclude in Sections VIII and IX. The appendices contain algorithms known from the literature, which are included for the sake of completeness.

II. BACKGROUND

In this section we give some basic required definitions and notations. A composite configuration item (CI) is represented in the CMDB as a tree. An explicit composite or simple CI will be denoted by $CI$. Each simple CI $CI$ has a type $type(CI)$, and a set of attribute values, $attr_1(CI),\ldots,attr_k(CI) \in \bigotimes_{i=1}^k A_i$, where $A_i$ is a set possible values for the $i$-th attribute. For instance a composite $CI$ can be of type $NT$ and have in the $i$-th attribute, which say is “operation system”, the value “Windows-7”. It might have different children CIs, e.g., a CI of type “CPU”. When we refer to $CI$ we might consider only the simple CI (with its attributes), or the entire tree, where the CI is its root. We will use the term simple CI or composite CI in order to stress the context when it is not clear.

A composite CI, $CI$, is comprised of a tree of CIs, denoted by $T(CI)$. A tree in this context is a directed graph $G(V,E)$ where $V$ is the set of nodes and $E$ is the set of directed edges. If $(u,v) \in E$ then we say $u$ is the parent of $v$ and $v$ is the child of $u$. If further $(u,w) \in E$ with $w \neq v$, we say that $w$ is a sibling node of $v$. The root node of a tree $T$ will be denoted by $root(T)$ and the children of a node $v$ will be denoted by $children(v)$. We say that there exists a path between $u$ and $v$ if $(v,u) \in E$ or if there exist $v_1,\ldots,v_k$ such that $(v,v_1),\ldots,(v_k,u) \in E$ and for all $1 \leq i \leq k-1$, $v_i, v_{i+1} \in E$. We will then denote such a path by $v \rightarrow u$. We will sometimes traverse a tree according to some order. In that case $x_T(v)$ will denote the index of $v$ in that order of the tree $T$. If the context is clear we neglect the $T$ subscript. A vector will be denoted by $\mathbf{x} = x_1,\ldots,x_n$. 

Fig. 1. Example for a “j2ee-domain” composite CI tree. The numbers on the edges specify the multiplicity of the sub-tree, e.g., here the j2ee-domain includes 22 jdbc data-sources.
III. POLICY RULE EXTRACTION

Our complete solution for configuration baseline policy discovery is comprised of three main stages as shown in Algorithm III.1. The input of our algorithm is a set of CIs and a threshold parameter $\theta$. It then works as follows:

- Creation of a distance matrix
- Cluster to discover composite CI patterns
- Automatically generate baseline from a cluster of composite CIs

We stress that, for the sake of simplicity of expositions, the algorithms are written as if the clustering is outputting the single largest cluster of CIs and creating a policy for this cluster. Trivially, the clustering can output all clusters and then a number of policies could be produced - one for each cluster.

Algorithm III.1: GeneratePolicy($\tilde{C}I, \theta, \alpha$)

$N \leftarrow \sum_{i=1}^{n} |CI_i|$  
comment: create distance matrix

$Params \leftarrow \text{Preprocess}(\tilde{C}I)$  
$D[1 \ldots N, 1 \ldots N] \leftarrow \infty$

for $i = 1$ to $n$, $j = 1$ to $n$

do $M_D = \text{CITreeEdit}(CI_i, CI_j, Params)$

Update $D$ from $M_D$

comment: cluster the CIs

$S \leftarrow \text{NonParametricClustering}(D, \theta)$

comment: generate the policy $P$

$G_P \leftarrow \text{ComputePatternGraph}(S, \tilde{C}I)$

$P \leftarrow \text{GeneratePolicy}(G_P, \tilde{C}I, \alpha)$

return $(P)$

As one can see in Algorithm III.1 the first stage creates a distance matrix $D$ of size $N \times N$, where $N$ is the number of composite CIs including internal CIs (that is, the number of sub-trees in the forest of the input CIs). This matrix is populated by repeatedly computing a distance matrix $M_D$ which includes the distances between all the sub-trees of one composite CI $CI_i$ and the sub-trees of another composite CI $CI_j$. D is input to the clustering stage as input. We then compute a policy so that for at least $\alpha$ fraction of the input CIs the policy holds. Our next sections will elaborate as for each of the required algorithms within Algorithm III.1.

IV. DISTANCE COMPUTATION FOR COMPOSITE CONFIGURATION ITEMS

This section describes in detail the creating of the CI tree-edit distance matrix $D$. The tree-edit distance depends on the following four cost types:

- $\text{rep}(CI_i, CI_j)$ which computes the cost of replacing the simple CI $CI_i$ by the simple CI $CI_j$. This computation depends mainly on the attributes of each CI. We assume we get as input the function $\tilde{W}$ which determines the distance between two simple CIs weighing the attributes.
- $\text{mult}(CI_i)$ which computes the cost of replacing one instance of a simple CI $CI_i$ by more than one CI. We assume we get as input the function $\tilde{P}$ which gives a penalty to each type of simple CI if assigned with multiplicity.
- $\text{del}(CI_i)$ which computes the cost of deleting the CI subtree $T(CI_i)$, and
- $\text{ins}(CI_i)$ which computes the cost of inserting the CI subtree $T(CI_i)$.

As one can see in Algorithm III.1 it includes a preprocessing step to infer parameters. Explicitly, the parameters $\tilde{W}$ and $\tilde{P}$, which are necessary for the four cost functions. In Section V we present a method of obtaining parameters from the data. For the time being, to simplify the presentation of the algorithms we assume that $\tilde{W}$ and $\tilde{P}$ are part of the input. We further assume that the time to compute these four functions is independent of the size of the subtree. In our case, the cost for insertion and deletion is a constant, independent of the input value (Alternatively, the values can be pre-computed prior to the tree distance computation.)

A. The Composite Configuration Item Tree Edit Distance

The algorithm to compute the tree distance for composite CIs (Algorithm IV.1) is a recursive algorithm. In each step, two nodes (simple CI) and their children are considered. If the nodes are not of the same type, or one of them has no children, the case is simple. In the general case, the distance between each pair of the children is recursively computed, and the distance between the nodes along with the distance between the two sets of children is then considered. We used here the maximum of the two distances, but alternatively one can use the sum.

Algorithm IV.1: CITreeEdit($MD, T_1, T_2, p$)

$n_1 \leftarrow |T_1|, n_2 \leftarrow |T_2|$

$r_1 \leftarrow \text{root}(T_1), r_2 \leftarrow \text{root}(T_2)$

$\tilde{c}_1 \leftarrow \text{children}(r_1), \tilde{c}_2 \leftarrow \text{children}(r_2)$

if $\text{rep}(r_1, r_2) = \infty$
then $M_D(I(r_1), I(r_2)) = \text{inf}$, return

if $n_1 = 0$ or $n_2 = 0$
then $M_D(I(r_1), I(r_2)) = \max(\text{rep}(r_1, r_2),$

$\sum_{i=1}^{n_1} \text{del}(c_1[i]) + \sum_{j=1}^{n_2} \text{ins}(c_2[j]))$, return

for $i = 1$ to $n_1$, $j = 1$ to $n_2$
do $\text{CITreeEdit}(MD, c_1[i], c_2[j], p)$

$M_D(I(r_1), I(r_2)) = \max(\text{rep}(r_1, r_2),$

$\text{MinCost}(MD, \tilde{c}_1, \tilde{c}_2, p))$

return
B. The Multiple Assignment Problem

The “edit distance” of child CIs between two CIs embodies the unique constraints of this problem, as discussed in Section I. The main idea is that given two sets of child nodes in a tree, we want to match each node in one set to a node, or a sub-set of nodes in the other set, so that the cost will be minimal. The definition of the cost is designed to allow in some cases matching one-to-many with low cost, when the multiplicity of the type of the node is of lesser significance (e.g. the number of configured IP addresses for a computer). In other cases we want the cost of multiple matches to be high, when different multiplicities signifies different functionality (e.g., the number of CPUs in a computer). In that case, the “edit distance” should prefer to “delete” a CPU when moving from one set to the other, rather than matching one CPU to two CPUs in the other set. In addition, the cost of the match should account for similarity of the attributes of nodes that are matched to each other. For example, if one has two file systems one with 10GB and the second with 160GB, and the second has two file systems with sizes 20GB and 200 GB, we would like them to be assigned in that order, so that the cost of their dissimilarity would be minimal.

To find the optimal set of matches, we construct a weighted bipartite graph, where the weights are the cost for the match (or distance between the two CIs). In order to allow delete and insert operation we add two special nodes (one to each set): a “delete” and an “insert” nodes. Nodes might be assigned to more than one node, but should pay a certain penalty, according to their type. There is a variety of approaches to solve the weighted matching problem [9], [4], [2], [6]. Like many prior works, we solve the matching problem using a flow problem often known as the “successive shortest path” [4]. In essence, the successive shortest path algorithm solves the minimum cost flow problem as a sequence of shortest path problems with arbitrary link weights. To enforce the requirement that any node in each of the set must have at least one node assigned to it in the other set, we use a multi-excess formulation. Each node in the first set has excess 1 and each node in the second set has excess −1. Moreover, the edges between the two sets have capacity 1 so that pairs of nodes can only be matched only. Thus, each node is required to be matched to at least one node on the other set (or to insert/delete). In order to allow many-to-one and one-to-many matches, we add a source and a sink nodes that have a large excess, and add the cost of multiple matches on edges between the source and sink nodes and the nodes of the bi-partite graph. We will make this formal in the following, illustrating the reduction using an example in Figure 2 thereafter.

Formally, our assignment problem is given the two sets of children CIs \( \hat{c}_1 \) and \( \hat{c}_2 \), the assignment maps each \( c_1[i] \) to zero or more elements of \( \hat{c}_2 \); similarly, zero or more elements of \( \hat{c}_1 \) may be mapped to each \( c_2[j] \). There is a cost \( d(c_1[i], c_2[j]) \) of assigning \( c_1[i] \) to \( c_2[j] \). This cost correspond to the dissimilarity between the CIs. There is a penalty, \( P \), for assigning any CI to zero elements. In addition, there is a penalty \( P_{type} \) for multiple assignments to an element of type \( type \). This penalty is accumulated for every assigned element except the first one.

To match the elements of \( \hat{c}_1 \) with elements of \( \hat{c}_2 \), we generate the following labeled graph \( G(V, E, Cost, Cap, Exc) \). Where \( Cost \) and \( Cap \) are the cost and capacity labels for each edge, and \( Exc \) is an excess value assigned to each node. Recall that we get as input the \( Params \) which includes \( P \) which gives a penalty to each type of simple CI if assigned with multiplicity. Let \( P > 1 \) be some constant penalty. The set of nodes and their excess are defined by \( V = \{ s, t, del, ins \} \cup V_1 \cup V_2 \) where the first 4 nodes are special nodes (source, sink, delete and insert) and for each \( i \in \{1, 2\} \), \( V_i = \{ c_1[i], \ldots, c_1[n_i] \} \).

The excess are

- \( Exc(s) = |V_1| + |V_2| \).
- \( Exc(t) = -2|V_1| \).
- \( Exc(del) = Exc(ins) = 0 \).
- for each \( v \in V_1 \), \( Exc(v) = 1 \).
- for each \( v \in V_2 \), \( Exc(v) = -1 \).

The set of edges and their cost and capacity labels are defined as follows.

- For each \( v \in V_1 \), \( e = (s, v) \in E \), \( Cost(e) = P_{type} \) and \( Cap(e) = \infty \), where \( type = type(c_1[j] = v) \).
- for each \( v \in V_2 \), \( e = (v, t) \in E \), \( Cost(e) = P_{type} \) and \( Cap(e) = \infty \), where \( type = type(c_2[j] = v) \).
- for each \( v \in V_1 \), \( e = (v, del) \in E \), \( Cost(e) = P \) and \( Cap(e) = 1 \).
- for each \( v \in V_2 \), \( e = (ins, v) \in E \), \( Cost(e) = P \) and \( Cap(e) = 1 \).
- \( e = (s, ins) \in E \), \( Cost(e) = 0 \) and \( Cap(e) = \infty \).
- \( e = (del, t) \in E \), \( Cost(e) = 0 \) and \( Cap(e) = \infty \).
- for each \( v \in V_1 \) and \( u \in V_2 \), \( e = (v, u) \in E \), \( Cost(e) = MD(c_1[j] = v, c_2[k] = u) \) and \( Cap(e) = 1 \), which corresponds to the distance-similarity between the two CIs.

Denote by \( Reduce \) the procedure described above, of reducing our assignment problem to the multiple-assignment minimum-cost-flow problem, by creating the input graph \( G \), and denote by \( MinCostFlow \) the minimum-cost-flow algorithm itself with the minimal cost as output, (see [15], Chapter 9.7 and in Appendix A) we perform the following algorithm:

\[ G \leftarrow Reduce(M_D, c_1, c_2, params) \]
\[ return (MinCostFlow(G)) \]

In the example of Figure 2 we have two hosts with CPUs, file systems and IP addresses as their children CIs.

So, in our notation we have:

- Set of \( N_1 = 9 \) elements \( c_1 = \{ CPU0, CPU1, CPU2, CPU3, C.; D.; E.; IP1, IP2 \} \)
- Set of \( N_2 = 10 \) elements \( c_2 = \{ CPU0, CPU1, C.; D.; E.; N.; U.; IP1, IP2, IP3 \} \); with number of elements
V. Parameter Inference

This section presents a method of computing the cost functions, defined at the beginning of Section IV. The preprocessing step, gathers statistics from the input Configuration Item data. This stage can be performed off-line and on a larger data set than the set to be later worked on. We assume that we have CIs of different types (e.g., host, cpu, etc.). Let \( \{ \text{type}_1, \text{type}_2, \ldots, \text{type}_r \} \) be the set of all types in the data-set and \( A_1, \ldots, A_i \) be the set of all possible attributes. During the pre-process stage two sets of parameters are inferred:

- **Attribute weights.** Attribute weights are set for each CI type. Attribute weights are used to ignore some non-relevant attributes, and enable more informative attributes to influence the distance. For example, if almost all CIs agree on a single value, or alternatively almost each CI has a different value for a certain attribute, it cannot distinguish between similar and non-similar CIs. This insight leads us to give high weights to attributes with moderate entropy values. Thus, statistics are gathered for each attribute \( \text{attr}_j \) counting the different values that appear in the data. For example, e.g. Windows-7: 245, Windows-Vista: 101, Unix: 7, etc.). Finally, for each \( i \in [r], j \in [l] \) we output \( w_{ij} \), which is heuristically computed as follows:
  - If almost all (more than 90%) of the CIs of type \( \text{type}_i \) have the same value for \( \text{attr}_j \) then \( w_{ij} = 0 \).
  - If the CIs of type \( \text{type}_i \) have many different values for \( \text{attr}_j \) (number of values is more than 10% of appearances) then \( w_{ij} = 0 \).
  - One can assign negative and positive additional domain knowledge into the system, e.g., attributes of certain types can get always value 0 (e.g., dates or IP addresses) or special attributes, such as 'Name', might get high value (say 10).
  - All other attributes get \( w_{ij} = 1 \).

For each type, weights are normalized to sum up to 1. CIs of different types are assumed to have an infinite distance. Alternatively, attribute weights may be provided to the algorithm. In practice, we combine this statistical approach with some domain knowledge in order to produce the weights.

- **Repetition penalty.** The repetition penalty is set for each CI type. The main idea is to look at the number of CIs of some type that appear together in a composite CI. If that number varies greatly, e.g., consider IP addresses assigned to a server, then the penalty for repetition should be small. If, on the other hand, that number is small, e.g., consider the number of CPUs in a server, then the penalty for repetition should be large. Thus, we collect statistics about repetition count of each CI type, and compute the variance of the distribution of the repetition counts. The repetition penalty influence the cost for making multiple assignments, which in turn will tend to make CIs with different repetition type more distant, especially if the repetition penalty is high. For example, a host with 1

We note that the successive shortest path has in general a pseudo-polynomial complexity. Yet, in our case we augment one unit of flow at every iteration, which would amount to assigning one additional pair of nodes. Consequently, if we let \( N \) denote the number of CIs, the algorithm would terminate within \( N \) iterations and require polynomial running time.

In practice we have noted that many of the children CIs might be identical in all their values. In such a case, we combine all the identical twins into one big node. In that case we update the excess of this new node to be of absolute value that is equal to the number of siblings that this big node represents. It is easy to see that this is equivalent to the solution with separate nodes. This significantly improve the performance of the algorithm on real data.
CPU vs. a host with 4 CPUs.

To be more precise, the preprocessing algorithm is as follows:

**Algorithm VI.1: PREPROCESS(CI)**

\[
\begin{align*}
\tilde{W} & \leftarrow \text{SetAttributeWeights}(CI) \\
\tilde{P} & \leftarrow \text{GeneratePenaltyValues}(CI) \\
\text{return } (\tilde{W}, \tilde{P})
\end{align*}
\]

The algorithm SetAttributeWeights is straightforward from the description above. The algorithm for the penalty representation is as follows:

**Algorithm VI.2: GENERATE PENALTY VALUES(CI)**

\[
\begin{align*}
\text{Hist}[1 \ldots \tau] & \leftarrow \emptyset, \text{ where Hist}_i = (Hist_{i,1}, Hist_{i,2}) \\
\text{for each } CI \in CI, \text{ for each } v \in T(CI), \\
\text{for each } i \in [\tau], \\
\text{do } h_i = |\{u \in \text{children}(v) \mid u \text{ is of type type}_i\}| \\
\text{if } h_i \in Hist_{i,1} \\
\text{then replace } (h_i, k) \in Hist_i \text{ with } (h_i, k + 1) \\
\text{else add } (h_i, 1) \text{ to Hist}_i \\
\text{for each } i \\
\text{do } P_i \leftarrow 1/(1 + \text{Variance(Hist}_i)). \\
\text{return } (\tilde{P})
\end{align*}
\]

**VI. CLUSTERING**

Like other data-mining applications, a suitable clustering algorithm must be efficient in both time and space. For such applications, agglomerative hierarchical clustering is typically selected. This approach to clustering begins with every object as a separate cluster and repeatedly merges clusters. A good review of agglomerative hierarchical clustering for document clustering may be found in [18], and a thorough review of data clustering methods, including hierarchical clustering is given in [7]. We use a mode finding clustering approach from [8] (for completeness it appears in Appendix B). This approach has good space and time performance because it uses neighbor lists, rather than a complete distance matrix. Neighbor lists are determined based on a distance threshold \( \theta \). The running time and memory requirement for the algorithm is \( O(N \times \text{average}(|\eta_0|)) \), where \( N \) is the number of objects to cluster and \( \eta_0 \) is the neighbor list of object \( i \). We expect the neighbor lists to be small and independent of \( N \).

**VII. GENERATING A BASELINE POLICY FROM FREQUENT PATTERNS**

This section describes algorithms for creating a policy given a set of composite CIs. The input CIs are assumed to adhere to some policy. At this point, we further assume that the CI clustering algorithm provides us the frequent pattern clusters. We invoke two algorithms to generate the baseline policy. The first algorithm, Compute Pattern Graph, computes pattern inclusions and gathers statistics about the frequency and repetition of the patterns. As shown in Algorithm VII.1, we create the graph \( G_P \), which is a hierarchical graph of the various clusters. Each cluster is represented by a node in the graph. A cluster node is linked as a parent of another cluster node if there exists a composite CI that is member of the first cluster which is a parent of a CI which is member of the second cluster. The edges are labeled by ranges. As each node can have many children that are member of the same cluster, we count these occurrences, and keep track of the minimal and maximal such multiplicities per-edge.

**Algorithm VII.1: COMPUTEPATTERNGRAPH(S, CI)**

\[
\begin{align*}
G_P(V, E, L) & \leftarrow \emptyset \\
\text{for each } S \in S \text{ add } v_S \text{ to } V \\
\text{for each } S, S' \in S \\
\text{for each } CI \in S, \\
N_{S,S'} & \leftarrow |\{CI' \in \text{children}(CI) \mid CI' \in S'\}| \\
\text{for each } S, S' \in S: L(v_S, v_{S'}) & \leftarrow (\infty, 0) \\
\text{for each } S, S' \in S: \text{ if } N_{S,S'} > 0 \\
\text{then add } (v_S, v_{S'}) \text{ to } E \\
\text{if } N_{S,S'} < L_1(v_S, v_{S'}): L_1(v_S, v_{S'}) & \leftarrow N_{S,S'} \\
\text{if } N_{S,S'} > L_2(v_S, v_{S'}): L_2(v_S, v_{S'}) & \leftarrow N_{S,S'} \\
\text{return } (G_P)
\end{align*}
\]

This algorithm works in time linear to the tree size. We use hash tables to calculate the minimum and maximum quantities of patterns.

The second algorithm, Generate Policy, utilizes a number of heuristics to build the policy from pattern paths in the pattern graph. The policy itself is actually a generalized CI in the sense that it is a tree of simple CIs with attributes. There are many ways to generate this tree out of the cluster graph \( G_P \). We present a very basic way here, which has an advantage in performance. Generally speaking, it adds part of the graph \( G_P \) in a greedy manner, as long as the support of the policy still exceeds the threshold which is given as input. We assume we have an efficient function Match which allow us to check whether a CI match a policy. At first the policy \( Pol \) is an empty graph so any CI will answer Match positively.
here the function Sort sorts the different paths based on a priority for each path based on the minimum quantity on each edge in the path (the multiplicity), the support of the path and the depth of the path.

VIII. Results

We have tested our solution on real customer data for two rather different types of configurations, both of which are quite common in practice.

1) A set of 700 hosts, which are compound CIs. In this dataset, each of which has many children, but the depth of the CI tree is small. Figure 3 shows a simple policy rule generated from the large database. The algorithm first clusters different types of hosts. This example for one cluster of NT hosts, the policy dictates that the NT machine should have a Microsoft OS, at least 2 file systems and 4 IP service endpoints.

2) A set of 8 CI J2EE domain CIs. In this data, each compound CI contains thousands of CIs, and a complex tree structure. Figure 1 depicts the policy produced for this set. This policy rule prescribes that each J2Eedevelopment contains 22 jdbcdatasources, 3 j2eeapplications of one type and one of a different type. In this example the two types of j2eeapplications differ by the CIs contained by them. One type includes 3 different types of ejbmodule whereas the second type contains only one.

Fig. 3. Example of a policy extracted from one CI cluster that was discovered in a dataset of 700 compound CIs.

IX. Conclusions

We have introduced an automatic baseline policy extraction algorithm. Our algorithm mines CIs within the CMDB, and automatically identifies repeating structures of composite configurations. We introduce a novel distance measure between CIs, and suggest an efficient computation method by reducing the computation of edit-distance between to trees to a minimum-cost flow problem.

REFERENCES


[Algorithm VII.2: GeneratePolicy(GP, C'I, α)]

\[
GP = G_P(V, E, L)
\]

\[
n \leftarrow |\mathcal{C'I}|, r \leftarrow \text{root}(G_P)
\]

for each leaf \(v \in V : R_v \leftarrow r \rightarrow v\)

Sort(\(\{R_v\}_v\))

\[
Pol(V_P, E_P, L_P) \leftarrow 0
\]

for each \(R_v\) :

if \(|CI_v : \text{Match}(CI_v, Pol \cup R_v)| > \alpha n\)

then \(Pol \leftarrow Pol \cup R_v\)

for each \(e \in E : \)

while \(|CI_v : \text{Match}(CI_v, Pol \cup R_v)| > \alpha n\)

for \(k \leftarrow L_1(e)\) to \(L_2(e) : L_P(e) \leftarrow k\)

return \(Pol\)
APPENDIX

A. Min Cost Flow

The algorithm MinCostFlow(G(V, E, Cost, Cap, Exc)) will be the Successive Shortest Path algorithm as appears in [15] (Chapter 9.7):

Algorithm: Successive Shortest Path (G)
Input: Graph G, costs cij, capacities rij, node capacities bi
Output: flow
Let N be the number of vertices in G
Let x[i] = 0; π[i] = 0; c[i] = b[i]
Reduced cost r = cij - π[i] + π[j]
initialize the sets E = {i|e[i] > 0} and D = {i|e[i] < 0}
while E ≠ ∅ do
    select a node k ∈ E and a node l ∈ D
    determine shortest path distances d[i]
from node k to all
other nodes in G(x) with respect to the reduced costs cij
let P denote a shortest path from
node k to node l
for all i ∈ N update π[i] = π[i] - d[i]
δ = min(e[k], -e[l], min{rij, i, j ∈ P})
augment δ units of flow along the
path P
update x, G(x), E, D and the reduced costs

B. Clustering algorithm

Let d[i,j] denote the distance between X_i and X_j and define the neighborhood η_k of X_i as

η_k = {k|d[i,k] ≤ θ, k ≠ i}

where θ is a given scalar. Define the density N_i at X_i as

N_i = |η_k|

That is, N_i is number of elements in η_k. Finally, we define g[i,j] which will be used to determine the parent node of X_i

\[ g[i,j] = \frac{N_j - N_i}{d[i,j]} \]

The parent nodes of the X_i are determined in sequence with the parent node of X_i being determined as follows:

Algorithm: NonParametricClustering
Input: Distances d[i,j], θ
Output: Cluster trees C = {C_k | C_k ⊆ {0..n - 1}}
and C_k ∩ C_l = Θ

If η_k is empty, X_i is a root.
If η_k is not empty, compute g[i,j] as

\[ g[i,j] = \max_{j ∈ η_k} g[i,j] \]