Tracking Configuration Changes Proactively in Large IT Environments

Manoj Soni†, Venkateswara R Madduri*, Manish Gupta*, Pradipta De*
manojsoni@gatech.edu, {vmadduri, gmanish, pradipta.de}@in.ibm.com
*IBM Research, New Delhi, India
†Georgia Institute of Technology, Atlanta, GA, USA

***Abstract*—Maintaining consistent views of components in complex IT environments is challenging due to frequent changes applied to the systems. Discovery tools, typically used for scanning the environment, can fail to gather up-to-date information. We present a technique to track changes by monitoring system events induced by a change. The system events are associated to entities in a configuration model. A change in a software component may also necessitate change to a dependent application. We track cross-product dependency by enhancing the configuration model of the service. We show the efficacy of our techniques by extending the configuration model of a complex IT environment.**

I. Introduction

Tracking changes to all entities in a large IT environment is a complex process. The best practice is to maintain a database, called the Configuration Management Database (CMDB), recording all changes to the Configuration Items (CIs). In order to keep the CMDB up-to-date, a change must be tracked, along with other affected entities, as soon as the change is performed. Since change tracking is performed by discovery tools, like IBM TADDM [1], which operates in a scan-gather mode, therefore, there may be a substantial delay before the next scan of a system to discover the change. This can lead to inconsistent view of the IT environment, leading to configuration errors if the information is critical for other activities.

This work presents a technique for fast change detection by observing system events, and associated changes in dependent entities. We monitor system events locally to identify entities which are likely to be affected. We assume that a change to a system will trigger a system event, like file system event, and by associating this event to affected entities we can track the entities which are modified. Thus, we need to build the relationship between the events and the entities a priori. We build this relationship by abstracting a production system in test environment, and applying changes on the test environment. Unlike some other techniques [2], [3], we do not expect an accurate dependency model among the entities. We are also not dependent on an accurate generation of dependency models, which has been explored in several other works [4], [5], [6], and complements our technique.

The key contribution of this work is a novel approach to complement existing discovery tools by introducing a push based component that can detect changes in a system, attach the change to specific portions of the model, and thereby, update CMDB proactively without waiting for the discovery tool to trigger.

978-1-4673-0269-2/12/$31.00 © 2012 IEEE

II. System Overview

An IT environment can be characterized by a Configuration Model (CM). A CM has two representations of a setup: a meta-model and a model. A meta-model captures existing types and relationships, while a model is an instance of a meta-model, where exact values are assigned to entities.

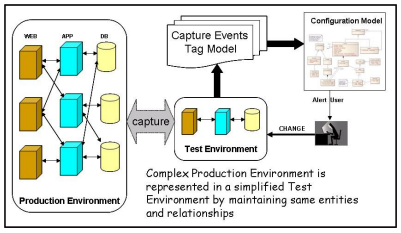

Fig. 1. System Overview

We replicate key components of a production environment in a test environment, such that both have equivalent meta-model. Fig. 1 shows how a simple test environment can capture complex three tier software architecture present in a production environment.

We add a new entity, called $Event$ type in the CM. Event type represents a system event, like file modification. Event type is associated to other entities in CM by a new relationship, called $Affects$. Affects relationship denotes that the system event modifies a model entity. The other relation, called $SameAs$, correlates two entities across software applications, thereby helping to track dependent changes.

$Affects$ relationship is characterized by two attributes: $AffectedAttribute$, and $AffectsWeight$. $AffectedAttribute$ denotes exact attributes in the CM

entity whose value may change, like the port number. $AffectsWeight$ denotes the probability of an Event type affecting the attribute.

$SameAs$ relation is characterized by three attributes: $independentAttribute$, $dependentAttribute$ and $SameAsWeight$. The $independentAttribute$ marks the field in an application whose value should remain same across other application tiers. The $dependentAttribute$ denotes the fields which are dependent on some $independentAttribute$. The $SameAsWeight$ is used to designate the importance of the connectivity while inferring whether it is a connected change. $SameAsWeight$ computation is explained later.

In our approach, software agents monitor events due to controlled changes in test environment and reports them to the "Capture Events, Tag Model" block, shown in Fig. 1. The "Capture Events, Tag Model" rediscovers the CM and compares the pre-change and post-change CMs to detect the modified model element, and associate them to Events. Cross product dependency is also determined, and the CM is augmented with $SameAs$ relationship. Finally, the knowledge is transferred from test environment to production environment.

III. Methodology

We begin with a variant of graph isomorphism problem. Two graphs G and G' are defined to be isomorphic if there is a one-to-one correspondence between their vertices and their edges, such that incidence relationship is preserved. However, if each vertex is annotated with a tag, and the tag matching is required, then the problem is harder. For our purpose, a tag is a tuple $< attribute, value >$. Our proposed *Neighborhood Closeness Algorithm (NCA)* refines the set of matching vertices, using other hints, to pin-point the correct matching vertex in the target graph. The intuition is to search connected elements for the candidate vertices from the two graphs, and assign a weight to mark the degree of match. Higher weight indicates higher degree of match between source and chosen vertex of candidate set. If there is no common attribute between two vertices, match weight is 0. Otherwise, weight is the number of attributes with identical values between the two vertices being matched. In case of a tie in weight value, components connected to source nodes are also matched against connected components of target node. Match value of the connected components is used to break the tie.

A. Enhancing Test Configuration Model

Given the CM of a test environment, it can be enhanced by running changes, and tracking its effects. In this section, we explain discovery of $Event$ types, as well as, $Affects$ and $SameAs$ relations by tracking effects of test changes.

1) Event Filter: A change on a system usually modifies a file. However, other system activities can modify files as well, therefore, merely tracking all modified files is not sufficient to isolate the effect of a change. We apply a frequency based approach, where number of times a file is modified under a change repeated number of times, is counted. Higher count implies that a change affects the file. The modification count, normalized over number of runs, gives the $AffectsWeight$.

2) Associate Events to Model Elements: The next step is to associate *Events* to a *Model element*. A naive approach to identify the model elements affected by a change would be to, run the discovery tool after a change, generate new model graph, and compare $< attribute, value >$ pairs node-by-node in Breadth-First-Search (BFS) order to detect modified node. However, discovery tools may not traverse the system components in the same order in each sweep, thereby changing the relative placement of nodes in the pre and post change configuration graphs. In order to match the nodes correctly in pre and post change CM, one must compare $< attribute, value >$ pairs of all nodes, using NCA approach. Finally, an $Affects$ relation is created from the filtered events to the affected model elements.

3) Cross-product Dependency Relationship: $SameAs$ relation captures the cross-product dependency by, first identifying model elements across products which are affected by a change in one application, and then assigning weights to the relation, denoting likelihood of a change in one model element having dependent changes. $SameAsWeight$ is computed by measuring closeness of related neighbors for both elements. Greater the weight, higher the probability of a model element affecting another model element. The technique is explained in Algorithm-1.

B. Enhancing Production Model

To enhance the production environment CM, first, relations in test environment model is exported to test environment meta-model; second, the meta-model level information is exported to production model.

1) Mapping Model Enhancements to Meta-model: There are three entities to be added to the production environment meta-model: $Events$, $Affects$ and $SameAs$ relationships. For Events, all events are added from the test environment model. For Affects relationship, the AffectsWeight and modified attributes associated with the edge are added. For SameAs relations, only the edge is included in the meta-model.

2) Adding Events: File event in meta-model represents a file path in test environment. For each file event in meta-model, we find candidate set of files in production system by the following method. For example, a file path in enhanced meta-model is

Algorithm 1 Cross-Product Dependency Discovery

Require: ModelElement, ModelAttribute.
Ensure: Create sameAs relation among ModelElements

1: **procedure** $CreateSameAsRelation(element1, attribute1)$
2: Find candidate set m from all other model elements, whose one of the attribute has the same value as $element1.attribute1$.
3: **for all** $m' \in m$ **do**
4: ns1 = getNeighbours(element1)
5: ns2 = getNeighbours(m')
6: distance=1, weight=0, matchedAttributeCount=0
7: saw = getSameAsWeight(ns1,ns2,distance,weight,matchedAttributeCount)
8: extendCM(element1,attribute1,m',attribute2,saw)
9: **end for**

10: **procedure** getNeighbours(element)
11: Returns all 1-hop neighbors in the transformed graph along with merged elements and itself.

12: **procedure** getSameAsWeight(ns1,ns2,distance,weight,matchedAttributeCount)
13: X = (node1.a, node2.b); node1.a = node2.b; node1 $\in$ nb1; node2 $\in$ nb2; a (b), respectively, an attribute of node1 (node2)
14: weight = weight+($|X|$-matchedAttributeCount)/distance
15: **if** ($|X|$-matchedAttributeCount == 0) || (nb1 $\cap$ nb2 is nonempty) **then**
16: return weight
17: **end if**
18: matchedAttributeCount=$|X|$
19: return CalculateWeight(getNeighbours(nb1), getNeighbours(nb2), distance+1, weight, matchedAttributeCount)

20: **procedure** getNeighbours(elementset)
21: init neighbourset to Empty set
22: **for all** element $\in$ elementset **do**
23: neighbourset = neighbourset $\cup$ getNeighbours(element)
24: **end for**
25: return neighbourset

/dir1/dir2/dir3/dir4/file1 (learnt from test environment). We find all files in production environment in pre-specified product install-root directories with name *file1*. Since both production environment and test environment are deployed from same virtual appliance, therefore corresponding file path in production environment should be */dir1p/dir2p/dir3p/dir4p/file1* (depth of file path will be same in both environments). We reject files where file path depth does not match. We compare directory names at each depth in both paths and increase match count by one if they are equal. We sort all such files based on total match count, and normalize this match count for each file path thus obtained. We call this match count as $pathMatchWeight$ ($w2$).

3) Adding Affects: For each Affects relationship in meta-model, we find file event, affected model element type and affected attribute of this model element. We identify model elements and attribute in production environment model with type as identified in meta-model and we find candidate set of files in production system by the method described in Section III-B2. There can be more than one candidate model elements and more than one candidate file events for Affects relation. We apply NCA for each candidate model element and candidate file path combination (there will be a match if an attribute value is substring of candidate file path). Finally, we normalize weight of all files by maximum weight obtained. We call this weight as *eventAffinityModelWeight($w3$)*. Now, three weights - *AffectsWeight($w1$), pathMatchWeight($w2$), and eventAffinityModelWeight($w3$)* are combined to generate a compound weight for each element in candidate set. The function to combine the weights is as follows, $w1 * w2 * w3/max(w2) * max(w3)$ where $max(w2)$, $max(w3)$ are the highest weights among the candidate set. We create an Affects relationship from this file to element with the compound weight.

4) Adding SameAs: For each SameAs relation in meta-model, we identify type of elements on both side of relations and add SameAs along with attributes on both sides responsible for SameAs relation. Now in configuration model, for each pair of those model elements, we check corresponding attribute values. If they are same then we add this relation similar to meta-model and calculate the weight using Algorithm-1.

IV. Case Study

In this section, we validate the proposed technique by constructing an enhanced Configuration Model of a benchmark application, IBM Trade Performance Benchmark, popularly known as Trade6 on Linux.

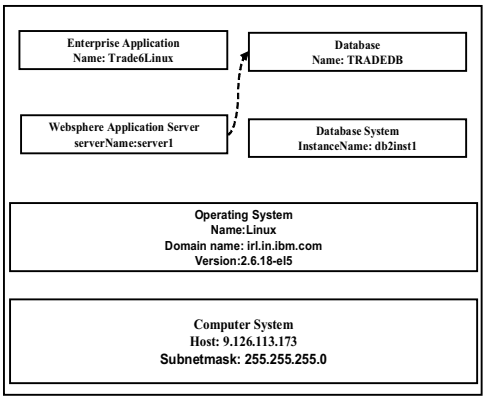

Fig. 2. Trade6Linux Virtual Appliance components

Fig. 2 shows the software component Configuration Model of Trade6 Linux solution. The WebSphere Application Server (WAS) hosts the application. WAS uses a database to store and maintain its application data. Application level dependencies are represented using directed arrows in the figure.

A. Test Environment: Generation of Enhanced Model

In test environment, a system administrator performed one change, which changed database port value. The change resulted in 13 change events in the system, which were pruned to 2, with a weight of 1.0 each by Event Filter. The CM for the changed Trade6 is gathered, and the pre and post CMs were compared using NCA to detect change in Model Element BindAddress[@=50000]. The change event elements were attached to

Composite Application Configuration	Affect Events	Affected Modelelments	True Affects relation (Weight)	False Affects relation (Weight)
$WAS+Db2$ on same machine	(e1)DBE1, (e2)DBE2	(me1)DBBA1, (me2)DBBA2, (me3)WASBA1, (me4)WASBA2	e1→me1(0.8), e2→me2(0.8)	e1→me2(0.5), e1→me3(0.4), e1→me4(0.4), e2→me1(0.5), e2→me3(0.4), e2→me4(0.4)
$WAS+DB1+DB2$ on different machines	(e1)DB1E1, (e2)DB1E2, (e3)DB2E1, (e4)DB2E2	(me1)DB1BA1, (me2)DB1BA2, (me3)DB2BA1, (me4)DB2BA2, (me5)WASBA1, (me6)WASBA2	e1→me1(0.7), e2→me2(0.7), e3→me3(0.7), e4→me4(0.7)	e1→me2(0.4), e1→me3(0.3), e1→me4(0.3), e1→me5(0.3), e1→me6(0.3), e2→me1(0.4), e2→me3(0.3), e2→me4(0.3), e2→me5(0.3), e2→me6(0.3), e3→me1(0.3), e3→me2(0.3), e3→me4(0.4), e3→me5(0.3), e3→me6(0.3), e4→me1(0.3), e4→me2(0.3), e4→me3(0.4), e4→me5(0.3), e4→me6(0.3)

TABLE I
TABLE SHOWING DIFFERENT EXPERIMENT SETUP AND WEIGHT FOR THEIR TRUE AND FALSE AFFECTS RELATIONS

Composite Application Configuration	Model elements containing same attr value "50000"	True SameAs relation (Weight)	False SameAs relation (Weight)
$WAS+Db2$ on same machine	(me1)WASRP, (me2)DBBA	WAS→DB(0.9)	none
$WAS+DB1+DB2$ on different machines	(me1)WASRP, (me2)DB1BA, (me3)DB2BA	WAS→DB1(0.8)	WAS→DB2(0.3)

TABLE II
TABLE SHOWING DIFFERENT EXPERIMENT SETUP AND WEIGHT FOR THEIR TRUE AND FALSE SAMEAS RELATIONS

Model Element BindAddress[@=50000] with $Affects$ relations. Next we discover the candidate model elements which have at least one attribute value matching the affected attributes value, viz. 50000. The candidate model element was WebSphereJ2EEResourceProperty[@portNumber=50000]. The system computed the strength of the $SameAs$ relation between BindAddress and WebSphereJ2EEResourceProperty. $SameAsWeight$ computes to 0.9 for the configuration model.

B. Production: Model Enhancement

Production environment had two DB2 installed, whereas in test environment there was only one DB2. For each DB2 instance, user-1 an user-2 are associated with port 50000 and 60000 respectively. Production setups were varied as follows: (I)Both WebSphere and DB2 installed on same machines with WebSphere pointing to DB2; (II)One instance of WebSphere and two instances of DB2 on different machines with WebSphere pointing to one of the DB2 instances.

For each setup, Table I shows candidate set of events, affected model elements and weight for possible $Affects$ relationship, while Table II shows possible candidates for $SameAs$ relation.

In Setup-I, to transfer $Affects$ relation from meta-model to model, 2 similar events and 4 model elements were found, as shown in Table I. Weights corresponding to 8 affects relation are shown, where only two relations have high weights. Transfer of $SameAs$ relationship is also recored. In Setup-I, two similar model elements were found with one of the attribute values as 50000. The weight of the SameAs relationship among them is 0.8. Setup-II results are similar. Results in Table I and Table II show the efficacy of our approach in different cases which includes false dependencies. For false dependencies, although weight does not become zero, we can observe that false dependencies always end up with a lower weight compared to true dependencies.

V. CONCLUSION

Maintaining up-to-date view of a complex IT environment in the face of frequent changes is a challenging task. Discovery tools may lag behind in updating the states because of its pull-based mode of gathering information about the entities. We present a technique to enhance the configuration model to track changes locally, thereby keeping the model view consistent. Configuration model is enhanced by adding, (i)$Events$, which are system activities that leads to a change in a model element, (ii)$Affects$ relation that connects an event to a model element denoting the possibility of an event changing a model element, (iii)$SameAs$ relation that tracks cross-product dependency relationship. We have shown how configuration models of complex production environments can be enhanced by creating simple test environments, and the knowledge exported from test environments to production environments.

REFERENCES

[1] "IBM Tivoli Application Dependency Discovery Manager." [Online]. Available: http://www-01.ibm.com/software/tivoli/products/taddm/faq-tech.html

[2] B. Gruschke, "Integrated event management: Event correlation using dependency graphs," 1998.

[3] G. Kar, A. Keller, and S. Calo, "Managing application services over service provider networks: Architecture and dependency analysis," in *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2000.

[4] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th international conference on Autonomic computing*, 2009.

[5] S. Bagchi, G. Kar, and J. Hellerstein, "Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment," in *In Proc. 12th Intl. Workshop on Distributed Systems: Operations & Management*, 2001.

[6] A. Brown, G. Kar, and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment," in *In Seventh IFIP/IEEE International Symposium on Integrated Network Management*, 2001.