Institute of Architecture of Application Systems

Automated Discovery and Maintenance of Enterprise Topology Graphs

Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{lastname}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings {INPROC-2013-50,
    author = {Tobias Binz and Uwe Breitenb{\"u}cher and Oliver Kopp
              and Frank Leymann},
    title = {{Automated Discovery and Maintenance of Enterprise Topology Graphs}},
    booktitle = {Proceedings of the 6th IEEE International Conference on Service
                 Oriented Computing \& Applications (SOCA 2013)},
    publisher = {IEEE Computer Society},
    pages = {126--134},
    month = {December},
    year = {2013},
    doi={10.1109/SOCA.2013.29},
 }
```


Universität Stuttgart
Germany

Automated Discovery and Maintenance of Enterprise Topology Graphs

Tobias Binz[1], Uwe Breitenbücher[1], Oliver Kopp[1,2], Frank Leymann[1]
[1]*Institute of Architecture of Application Systems*
[2]*Institute for Parallel and Distributed Systems*
University of Stuttgart, Stuttgart, Germany
lastname@informatik.uni-stuttgart.de

***Abstract*—Enterprise Topology Graphs (ETGs) represent a snapshot of the complete enterprise IT, including all its applications, processes, services, components, and their dependencies. In the past, ETGs have been applied in analysis, optimization, and adaptation of enterprise IT. But how to discover and maintain a complete, accurate, fresh, and fine-grained Enterprise Topology Graph? Existing approaches either do not provide enough technical details or do not cover the complete scope of Enterprise Topology Graphs. Although existing tools are able to discover valuable information, there is no means for seamless integration. This paper proposes a plugin-based approach and extensible framework for automated discovery and maintenance of Enterprise Topology Graphs. The approach is able to integrate various kinds of tools and techniques into a unified model. We implemented the proposed approach in a prototype and applied it to different scenarios. Due to the vital role of discovery plugins in our approach, we support plugin development with a systematic testing method and discuss the lessons we learned. The results presented in this paper enable new ways of enterprise IT optimization, analysis, and adaptation. Furthermore, they unlock the full potential of past research, which previously required manual modeling of ETGs.**

***Keywords*-Enterprise Topology Graph; Enterprise IT; Discovery; Maintenance; Crawling**

I. Introduction

Due to the increasing importance of IT in enterprises, there is a growing need for detailed technical insight – especially into the dependencies between applications and their components. Knowing which components exist, in which configuration, and what their relations are, enables new ways to analyze, optimize, and adapt IT. This insight is currently often lacking, which causes problems with severe negative impact on the business when making changes in IT. In practice, dependencies between components are still uncovered by "pulling the plug" of one server and analyze the impact. To tackle this, the concept of Enterprise Topology Graphs (ETG) as a technically fine-grained, formal, graph-based model to depict snapshots of the complete enterprise IT have been introduced [1]. An ETG includes all kinds of components, from business processes and services to their implementation and infrastructure, connected by typed edges representing the relations between the components. ETGs have already been applied to adapt [2], analyze [3], manage [4], and optimize enterprise IT, for example, by improving the ecological sustainability of business processes [5] and consolidating duplicate components [1]. Our current research uses ETGs to migrate existing enterprise applications to the Cloud [2], [6]. Migration and the other fields of application require a depth and breadth of information not available in a unified model before. To enable these fields of application, the research question addressed by this paper is: *How to discover and maintain a complete, accurate, fresh, and fine-grained Enterprise Topology Graph?* Such models can be discovered manually, but this is a time-consuming, costly, and error-prone task [7]. Existing solutions are either restricted to a particular scope (e. g., discovery on the level of network [8], [9], storage [10], [11], application configuration [7], or application [12]), focus on other aspects than analyzing, optimizing, and adapting IT (e. g., monitoring, license management, vulnerability scanner), address a different level of granularity (e. g., enterprise architecture documentation [3], [13], [14]), or do not discover instance information at all (e. g., software architecture reconstruction [15] or application topologies in deployment automation solutions [16], [17]). An approach covering enterprise IT from business process to infrastructure offering an extensible framework for existing solutions that enables developers to provide plugins has not been created yet. In this paper, we present an approach to automate ETG discovery and maintenance, i. e., crawling all software and hardware components used in an enterprise's IT and keeping the discovered ETG up-to-date. Similar to search engine crawlers discovering the Internet, the basic idea of this approach is that component-specific plugins discover the ETG. The goal is to automate most parts of the discovery using our framework and let domain-experts define the logic how to extract the information from the components as well as their dependencies. Our framework orchestrates discovery plugins, integrates information from a large variety of sources, and reconciles data to improve the quality of the discovered ETG. This enables the integration of existing solutions into a unified model.

The contribution of this paper is threefold: (i) We propose an extensible approach to discover and maintain ETGs. (ii) To realize this approach, we present (a) an architecture, (b) plugin framework, (c) scheduling algorithm, and (d) concept for reconciliation (i. e., consolidation, integration, and deduplication) based on which we implemented the

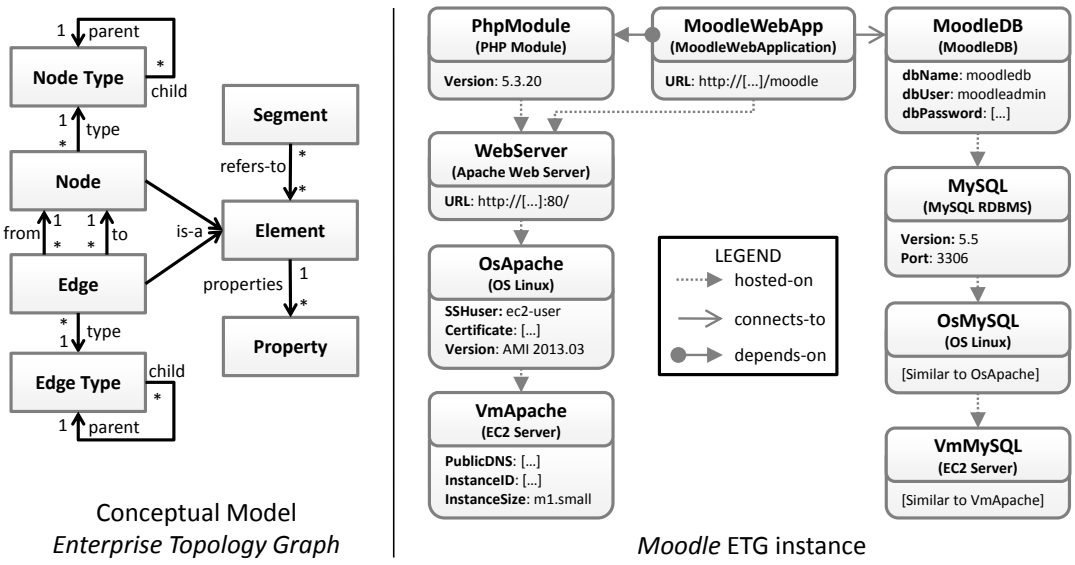

Figure 1. Enterprise Topology Graph conceptual model on the left and an extracted ETG segment on the right (Node Types are depicted in parentheses under its name, following the notation of [18]).

discovery framework. (iii) As plugins are important for a successful discovery, we present a systematic testing method as well as the lessons we learned while implementing plugins for our validation. We evaluate the approach by applying it to different scenarios.

The remainder of this paper is structured as follows: Section II provides the fundamentals by introducing Enterprise Topology Graphs. Our approach and framework are presented in Section III. Section IV discusses plugin implementation and systematic testing. We validate and evaluate our results in Section V and discuss related work in Section VI. Section VII concludes the paper and gives an outlook on future work.

II. Enterprise Topology Graphs

An *Enterprise Topology Graph* (ETG) [1] is a graph capturing a fine-grained technical snapshot of the complete enterprise IT, i. e., including all processes, services, software, infrastructure, and Cloud offerings as nodes, as well as their relations and dependencies as edges. Figure 1 shows the conceptual model of ETGs on the left. The semantics of nodes and edges are defined by types. *Node Types* represent the semantic of a component, e. g., a Web service, Web server, database, virtual machine, or network component. *Edge Types* represent the semantic of the logical, functional, and physical relationships between nodes, e. g., that a node is hosted on another node, has a dependency, or communicates with that other node. ETGs use the type system of the TOSCA standard [16], which allows users to add new types, as well as refining types through derivation. An *Apache Web Server* node type, for example, is derived from the type *Web Server*, *connects-to* is the parent of the edge types *JDBC connection* and *VPN tunnel*. To attach further information, *properties* may be attached to nodes and edges. For example, runtime information, configuration, or implementation details. The set of properties is defined by the element's type.

The right-hand side of Figure 1 depicts an ETG segment representing the school management and learning application *Moodle*, which runs on a LAMP stack (Linux, Apache, MySQL, PHP). For illustration purposes, this is only a small extract of an ETG representing the whole enterprise IT, which may have many thousands of nodes. The Figure shows that the ETG is able to depict enterprise IT as decomposed, fine-grained model, instead of a set of monolithic black boxes.

III. Discovery Approach and Framework

This section presents our approach and framework for ETG discovery. Starting from the requirements presented in Section III-A, we introduce the approach in Section III-B and describe the framework's architecture in Section III-C. Section III-D explains how the discovery plugins are scheduled and Section III-E shows how this information is reconciled.

A. Requirements

The requirements steering our research have been identified based on a literature study and our past works using ETGs in different domains. In particular, Hauder et al. [19] and Farwick et al. [20] discuss challenges for enterprise architecture documentation, a related field of research. The following requirements have been identified for automated ETG discovery and maintenance:

R1: ETG Quality. The quality of the ETG is vital for all fields of application. Therefore, the first requirement is ensuring the quality of the resulting ETG. Based on Batini et al. [21], Wang et al. [22], and Farwick et al. [20],

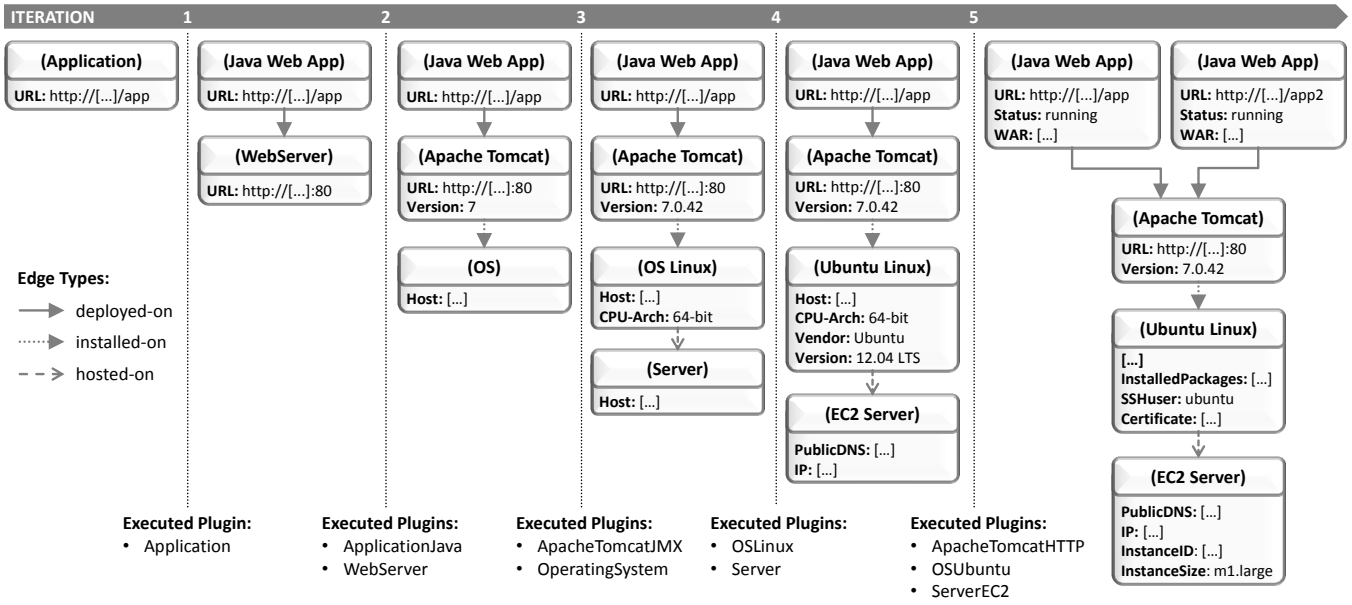

Figure 2. Growing example ETG before and after five iterations of the discovery approach. The types of the nodes are depicted in parentheses and the edge types with different kinds of arrows described in the legend on the left. Values too long to be depicted in this figure have been substituted by [...].

we define four criteria of ETG quality: (i) *Completeness*. All components of the enterprise IT and their relations are included in the ETG (cf. [20], [21]). (ii) *Accuracy*. The ETG is correct, consistent, and does not contain more or less components and relations than the enterprise IT (cf. [22], consistency in [20] and [21]). (iii) *Freshness*. The ETG does reflect the current state of enterprise IT (cf. actuality in [20], currency and timeliness in [21], part of relevancy in [22]). (iv) *Granularity*. The ETG represents the level of abstraction required for the desired field of application (cf. [20]). Putting these criteria into metrics and enable automated calculation of each metric for a given ETG is future work.

R2: Open-world Assumption. Neither the type of components, their configuration, and relations used in enterprise IT in the future, nor the evolutions in processes, architecture, software, and hardware can be predicted. Therefore, the framework must not limit the types of components and relations to be discovered. This is also an implication of using TOSCA's type system [16], which embraces the open-world assumption by foreseeing the addition and refinement of types. In addition, the framework must not require the applications to be built in a certain way or use technology of a certain vendor. This demands the approach to be *extensible* and enables domain- or enterprise-specific extensions.

R3: Integration. The integration of existing information sources, tools, and solutions must be supported (cf. challenges DC2 and TC in [19]; requirement AR1 in [20]), because there are existing solutions which provide valuable information for certain parts of enterprise IT (cf. Section VI).

R4: Update. Enterprise IT changes frequently and the ETG must reflect these changes timely to ensure up-to-date analysis results and optimization recommendations (cf. DC3 in [19]; IR1 in [20]).

R5: Minimize Operational Impact. The plugin's operational impact, i. e., the additional load caused by analyzing a production component, must be minimized (cf. DC1 in [19]).

B. Automated Discovery Approach

This section presents the iterative discovery approach implemented by our framework. The core architectural decision is that the discovery logic for the different types of components and relations is provided by type-specific plugins. A plugin can do *anything* to find out information about a component or relation. Plugins have to *pull* information from the respective component, because no component in enterprise IT should be aware of the discovery framework by *pushing* information to it: Enterprise IT components pushing change events or updates to the discovery framework would render the approach unusable for discovering *existing* enterprise IT components as well as violating the requirements *open-world assumption* (R2) and *minimize operational impact* (R5). Due to its extensible architecture, the framework is able to support all kinds of protocols (HTTP, SSH, SCP, SNMP, JMX, etc.) and is able to extract information from all kinds of data formats (XML with different schemas, text, property files, databases, console output, logs, etc.). Figure 2 shows how an ETG, starting from a given node, grows through five iterations: The application node (cf. ETG before iteration 1 in Figure 2) is provided by the user or another system, depending on the use case. For migration, the entry node (i. e., the interface or endpoint of the application, where the users engages with it) of the application to be

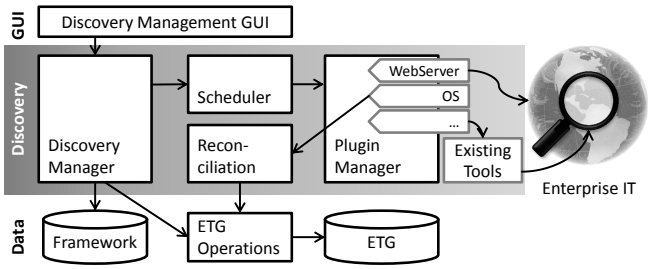

Figure 3. Discovery Framework Architecture.

discovered and migrated must be provided. Based on this ETG, our framework is started and in each iteration one or more plugins are executed. The executed plugins of each iteration are listed at the bottom of Figure 2. The *WebServer* plugin, for example, is able to extract the type and version of the Web server (iteration 2 in Figure 2). Depending on the information the plugin is able to discover, it may change the type, modify properties, or add new nodes and edges to the ETG. For example, the operating system node is created by the *WebServer* plugin, which, however, is not able to determine additional information. After creating the OS node, it is connected to the *Tomcat* node with an edge of type *installed-on*. By using different types for edges and the possibility to attach properties to them (not shown in this example), rich and fine-grained semantics can be added to the ETG. The *OS Linux* node gets processed in iteration 4 by the *OS Linux* plugin, which determines that the Linux derivate is *Ubuntu* as well as its version. Therefore, the type is changed to *Ubuntu Linux* to enable processing of the *OS Ubuntu* plugin. The Ubuntu specific plugin then adds information such as the installed packages in iteration 5. The implementation of plugins is discussed in Section IV.

Plugins are able to work together by refining and extending each other's information, as shown before. The different plugins are loosely coupled through the ETG and their cooperation works solely based on the types and properties of the nodes and edges in the ETG. This is possible because the ETG type system defines the properties which can be assigned to a node/edge as well as their semantics. Therefore, one plugin *knows* where to store the information it discovered and another plugins *knows* the meaning of a property when processing the respective node/edge. Plugins are orchestrated by the scheduler discussed in Section III-D.

C. Framework Architecture

The framework's architecture (shown in Figure 3) is split into three layers: *GUI* (graphical user interface), *discovery*, and *data*. The right-hand side of Figure 3 represents the *real world* enterprise IT to be discovered. The data layer stores the Enterprise Topology Graphs as well as framework information such as available plugins, ongoing discoveries, and their configuration. In the discovery layer, the *discovery manager* controls the overall discovery as well as the user interactions. The actual discovery is done by the plugins on the right hand side of Figure 3. The *plugin manager* is responsible for maintaining the available plugins and invoking them. Between the discovery and plugin manager, the *scheduler* implements an algorithm deciding in which order the plugins are invoked (cf. Section III-D). The access to the ETG is encapsulated by the *ETG Operations*, which offer basic operations such as getting, adding, editing, and deleting nodes or edges as well as complex operations such as subgraph isomorphism. Operations of plugins on the ETG are reconciled (cf. Section III-E) before they are applied.

D. Scheduling

The scheduler controls the iterative discovery process and decides when and on which ETG node a plugin is executed. For this task, the scheduler maintains four kinds of data: (i) Each plugin registered in the *Plugin Manager* provides a list of node types it is able to process. Based on this information, the scheduler maintains the function $compatiblePlugins$ which maps each type to a prioritized (i. e., ordered) list of plugins able to process this type. The order of the list can be configured in the framework based on statistics or experience, e. g., to put the plugins discovering the information required for the current use case first. Let $Types$ be the set of all node types and $Plugins$ the set of all registered plugins, then we define $compatiblePlugins : Types \rightarrow 2^{Plugins}$. (ii) Plugins define their required inputs explicitly through topology queries. Topology queries are executed on the ETG and return nodes, edges, or a property of a node/edge. For example, a plugin using an SSH connection to discover information might require the host, user name, and SSH certificate of the hosting operating system. The topology queries are made available to the scheduler through the function $inputs$: Let $Queries$ be the set of possible topology queries, then we define $inputs : Plugins \rightarrow 2^{Queries}$. (iii) For each node and edge a version counter is stored, which is incremented if the type or a property of a node or edge is changed. We define the function vc, which represents the counter values of all nodes and edges at time $t \in \mathbb{N}_0$ as $vc : \mathbb{N}_0 \rightarrow (Nodes \cup Edges \rightarrow \mathbb{N}_0)$. (iv) The time of the last execution of a plugin on a node is stored in $le : (Plugins \times Nodes) \rightarrow \mathbb{N}_0 \cup \{\perp\}$, where $\perp$ states that this plugin was not executed on this node before.

In each iteration, Algorithm 1 (see next page) determines the plugin to be executed next. The main *for all* loop (Line 1) iterates over the compatible plugins of the currently processed node type. Line 2 executes the topology queries on the ETG to determine the values of the plugin's inputs. Given the query results, the plugin may decide if it is able to operate in a meaningful way (Line 3). Plugins may, for example, check if an input (i. e., query result) is empty or does match the expected format or granularity. If there is no time of last execution for this plugin on the given node (Line 6)

Algorithm 1 $nextPlugin(node \in Nodes,\ t \in \mathbb{N}_0)$

```
1: for all plugin ∈ compatiblePlugins(type(node)) do
2:     Run all queries in inputs(plugin)
3:     if plugin cannot operate on query results then
4:         continue with next plugin
5:     end if
6:     if le(plugin, node) is ⊥ then
7:         return plugin
8:     end if
9:     if vc(t)(node) ≠ vc(le(plugin, node))(node) then
10:        return plugin
11:    end if
12:    for all query ∈ inputs(plugin) do
13:        r ← query result (node, edge, or property)
14:        if vc(t)(r) ≠ vc(le(plugin, node))(r) then
15:            return plugin
16:        end if
17:    end for
18: end for
19: return ⊥
```

it is the next plugin to be executed (Line 7). Then it is determined whether the node (Lines 9-10) or the plugin's inputs (Lines 12-17) changed since the last execution of the plugin by comparing the version counters of the last execution with the current version counter. If yes, the plugin is returned and executed. The computational complexity of Algorithm 1 is primarily tied to the number of compatible plugins (Line 1) and secondarily to the number of queries (Lines 2 and 12), i. e., the inputs required by the respective plugin. Potentially, the number of compatible plugins and queries may be arbitrarily large. However, based on our experience, we found both numbers are usually lower than five in practice.

Algorithm 2 does the overall discovery for the current point in time t. If each compatible plugin was executed on each node without discovering anything new, i. e., vc did not change and thus $\perp$ was returned, the discovery terminates.

Algorithm 2 $discover()$

```
1: while additional information has been discovered do
2:     for all node ∈ Nodes do
3:         if nextPlugin(node, t) ≠ ⊥ then
4:             execute nextPlugin(node, t)
5:         end if
6:     end for
7: end while
```

The presented scheduler is efficient, because only plugins which might discover additional information are executed. This is ensured through defining the plugin's inputs and using version counters to detect changes relevant for a plugin.

Maintaining an existing ETG (cf. requirement *R4*) is started manually or periodically, because the discovery framework is not informed about changes in enterprise IT. Our approach observes enterprise IT from the *outside*, i. e., the components are not aware of the discovery framework. Maintaining existing ETGs uses the same plugins and scheduler as used for the initial discovery. An update is initiated by deleting the history of past executions (le is deleted, but not the ETG itself), which makes the plugins run again on the respective nodes. However, when processing an existing ETG fewer plugin executions are required. In addition, plugins can take additional measures to reduce the effort of their execution, e. g., only check if the component still exists and validate if the data changed. For example, the Tomcat plugin, which added the installed applications to the ETG during the last discovery, updates the ETG by checking whether the set of installed applications changed. Based on this, it adds or removes application nodes as well as the respective *hosted-on* edges to the Tomcat node.

E. Reconciliation

Variations in naming, format, and granularity of information discovered by different plugins, as well as duplication of nodes and edges found by different plugins independently, reduces the quality of the ETG. To prevent this (cf. requirement *R1 ETG Quality* in Section III-A), reconciliation consolidates, integrates, and deduplicates data in the ETG. For example, if the same Web service is invoked by two applications, the resulting ETG might contain two Web service nodes, including their whole stacks.

The basic idea of the reconciliation concept is to always have a *clean* ETG and check the operations changing the ETG whether they keep the ETG clean or not. Therefore, we have to answer the questions: (i) How to determine the equality of nodes and edges? (ii) How to reconcile this situation? Benefiting from the common data model and strong typing in the ETG, question (i) is tackled by specifying the identity of a node or edge on a per type basis. For each type, one or more properties are defined as identifier, similar to primary keys in relational databases. For the Apache node in our example (cf. Figure 1) this is the URL, i. e., host and port. However, if the same server node is one time added with its domain name (e. g., "example.org") and another time with its IP address (e. g., "1.2.3.4"), a simple identifier does not work. One solution to determine equality would be to resolve the domain name to an IP. Due to the open-world assumption (cf. *R2*) this requires type-specific processing, because the semantics of the nodes, edges, and properties are type-specific. For all adding or modifying operations on the ETG, the reconciliation components uses either the generic identifier or, if available, the type-specific logic to determine whether the respective node or edge is already contained in the ETG. If a duplication has been found, the two nodes or edges must be merged to conserve the discovered information of the existing and the duplicate node. This brings us to question (ii): Merging nodes or edges is done by

merging their properties, which also requires type-specific knowledge. For example, how to merge two nodes with the *version* property values "7" and "7.0.42", which have been discovered by different plugins? By knowing the semantics of the properties, the second, more detailed, version number would be selected. The type-specific logic is supplied by *Reconciliation Plugins* in the reconciliation component of the architecture (cf. Figure 3).

IV. Implementing Discovery Plugins

One key of the described framework are plugins discovering different IT components. The framework tries to take as much of the heavy lifting from the plugin developer, so that the domain-expert can focus on the type-specific logic to extract the relevant information. To simplify the development process of plugins, this section discusses lessons learned (Section IV-A), the implementation of a plugin (Section IV-B), and how to test and improve plugins (Section IV-C).

A. Lessons Learned

This section discusses the lessons we have learned while implementing 21 plugins using Java: (i) Common functionality, such as making an SSH connection, downloading files, and formatting URLs are used by many discovery plugins and, therefore, extracted as reusable helpers. With a growing number of helpers, the effort to create new plugins reduces progressively. (ii) Different plugins may determine the same information (e. g., version number) in different granularity (e. g., Tomcat version "7" or version "7.0.42"). For each property of a node or edge, its type documents the syntax and semantics. This enables plugins to overwrite the less precise value. In case it is sensible to hold a high level and detailed representation of a property, an additional property should be introduced. For example, Tomcat's compile parameters and code revision are more precise, but cannot replace the version number. (iii) In terms of plugin granularity, we decided to implement many small plugins instead of monolithic blocks covering multiple types. (iv) Our approach does not restrict how plugins gather their information, neither the sources nor the way they are retrieved. Based on analyzing components existing in enterprise IT and our experience in developing plugins, we identified the following six classes of information sources: Component configuration, component communication, component code, monitoring and auditing, people and documentation, and IT/systems/datacenter management solutions.

B. Example Plugins

To show how plugins may be implemented and how they interact, we selected the discovery of a Tomcat node as a simple example of a well-known component. In this example, we start with a node of type *WebServer*, with the properties *host* (IP or domain name) and *port*, which have been discovered by another plugin before. The plugin for generic Web server sends an HTTP GET request for the root page and analyzes the response headers. If the *server* header exists and its value is known by the plugin, the type of the node is changed respectively. The node type is changed to *Apache Tomcat* in this example, because the server header was "Apache-Coyote/1.1". If the server property is empty or not available (e. g., to obfuscate the type of server due to security concerns), the plugin uses the Web server fingerprinting tool *httprint*[1] to find out the type. This tool is integrated by calling it on the command line of the machine running the discovery plugin and parsing its output (cf. requirement *R3*). To determine the exact Tomcat version, installed applications, and hosting operating system, the *Apache Tomcat HTTP* plugin tries to load the server's management and status page. Similarly, the *Apache Tomcat JMX* plugin connects to the JMX management service if the credentials are available. If accesible using HTTP or JMX, the information is extracted from the server's reply and put into the properties of the respective ETG node. The version ("Apache Tomcat/7.0.42") is put into the Tomcat node's *version* property. For each application, a node of type *Java Web App* with the property *url* as well as an edge of type *hosted-on* to the Tomcat node are created. Next, an operating system node is created and the Tomcat node is hosted on this operating system by creating the respective edge. If no further information about the operating system is available, we use *Nmap*[2] which also uses fingerprinting techniques, to determine the operating system properties type ("Linux"), vendor/derivative ("Ubuntu"), and version ("12.04 LTS"). In the following, the created nodes are further discovered and their information is completed by other plugins.

C. Systematic Testing Method

The quality of the plugins is crucial to the overall utility of the ETG and applications relying on its data. Therefore, this section presents a systematic method to develop and test plugins and quantify their quality (cf. requirement *R1* in Section III-A). Following the definition of a *systematic test* by Ludewig et al. [23], our method has five steps: (i) *Create Test Cases*. First, describe one or a set of applications in a machine-readable format to be discovered as part of the test case. (ii) *Deploy Test Cases*. To prepare the test run, the test cases are deployed into a test environment. For testing it is, therefore, preferable to use languages capable of automated deployment, such as TOSCA [16], [24], CloudFormation, or Chef. (iii) *Test Discovery*. In this step, the discovery framework discovers the test application. (iv) *Compare*. After the discovery has finished, the discovered ETG (i. e., the application instances) is compared with the original topology of the test case (i. e., the application models). Based on

[1] http://www.net-square.com/httprint.html
[2] http://nmap.org/

this, the two test metrics *recall* and *precision* are calculated, following their definition in data mining [25]. The *recall* metric denotes the fraction of the nodes and edges in the discovered ETG which were modeled in the test case (and deployed). The *precision* metric denotes the fraction of the properties in the discovered ETG which have been expected. Automating the metric calculation is an open issue for future research. (v) *Evaluate*. In the end, the results are analyzed and, if the results are not sufficient, the plugins are adapted. Afterwards, one returns to step (ii) to rerun the test and check if the metrics were improved.

V. Validation and Evaluation

We evaluated the discovery approach presented in this paper in terms of feasibility and general applicability, extensibility, and economics as well as the fulfillment of the requirements of Section III-A:

(i) Feasibility and General Applicability. To proof the discovery approach and the framework's feasibility and general applicability, we discovered four scenarios of different size (up to 1,060 nodes) from different hosting environments (local machine, Amazon EC2, Microsoft Azure). This has been done using 21 different kinds of discovery plugins, ranging from *EC2 Server* (infrastructure) to *PHP application* (software) and *Active MQ* (middleware) to *BPEL* (process). The fact that it was possible to implement this diverse set of plugins indicates that our approach is able to deliver on its goal to cover a wide range of components in enterprise IT.

For plugin development we applied the systematic testing method presented in Section IV-C with the four aforementioned scenarios as test cases. Following the testing method we did a number of iterations to develop and improve the plugins. Moreover, we were able to cover each scenario with the resulting ETG being complete and accurate, according to requirement ETG Quality (*R1*).

To understand the relation between the number of nodes discovered and the discovery time, we used the largest of our four scenarios in terms of nodes and split it into 10 equally sized segments. We did 10 test runs starting with the discovery of the first segment and adding one segment after each test run. Figure 4 depicts the relation of the number of nodes in the discovered ETG (x-axis) to the discovery time in seconds (y-axis). Although the split of the ETG into segments was artificial, the numbers indicate that the discovery time increases linearly with the number of nodes in the ETG. These measurements also point out that the discovery time, for our scenarios and the implemented set of plugins, is in a range which seems to be well suited for practical applicability.

Another scenario is the discovery of the school management and learning application *Moodle* from a larger virtualized environment to enable its migration. This is a scenario of the German government-funded project *CloudCycle*[3], which has the goal to provide security- and privacy-aware Cloud services for schools and to migrate their existing applications. We have been able to discover Moodle including its complete hosting stack in a level of detail enabling us to deploy it in another environment. This shows that the discovery framework is able to discover ETGs in a quality which is suitable for the intended use cases.

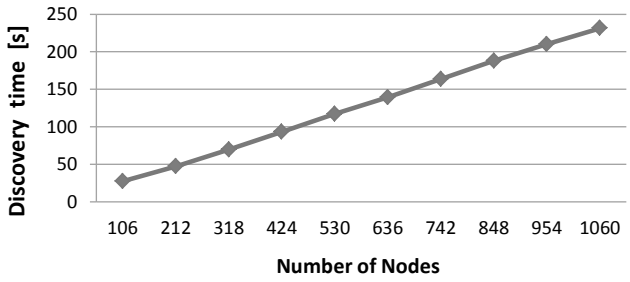

Figure 4. Relation between number of nodes and discovery time.

(ii) Extensibility. Due to the framework's plugin architecture and the ETG's extensible type system, our approach is extensible and fulfills the *open-world assumption* (cf. requirement *R2*). Plugins can do literally everything to extract information using the capabilities of Java as well as discovery logic invoked from within Java, for example, operating system functionality, executables, scripts, or Web services. We showed that existing tools, such as *Nmap* and *httprint*, can be integrated through plugins (cf. requirement *R3*). The framework ensures integration and reconciliation of the data from different sources and, therefore, maintains the ETG's quality (cf. requirement *R1*).

(iii) Economics. The goal of reducing IT operation cost is facilitated by automating a former manual, time-consuming, and error-prone task [7]. Our approach separates concerns between the discovery framework covering the tasks which can be automated and domain experts developing plugins.

VI. Related Work

In contrast to *Enterprise Architectures* (EA) [26], ETGs are technically fine grained models, i. e., address a different perspective on enterprise IT. The information captured in *Enterprise Architecture Management* (EAM) is often modeled manually, which is a time-consuming and error-prone task, and techniques to analyze these models are rather informal [26]. Thus, we focus in the following on the research to automate EA documentation, which is concerned with the creation of EA models. Hauder et al. [19] defines the research challenges in this field, which are reflected by the requirements of our approach. Farwick et al. [27] presents organizational processes to discover and maintain EAs. Buschle et al. [13] map the network scan output of the vulnerability scanner *Nexpose* onto the technology layer of an EA model. Our approach has more depth and breadth in

[3]http://cloudcycle.org/en

discovery because it covers the complete enterprise IT from infrastructure to processes (cf. ETG in Section II), discovers relations not visible in a network scan, and integrates data of different solutions as well as additional discovery logic via plugins. In [3], we showed how EA and ETG complement each other: Strategic IT decisions are made based on the EA and the technical realization is done using ETGs.

Software Architecture Reconstruction (SAR) has the goal to determine the architecture of a given software, in contrast to our broader scope to discover runtime information and relations between different software and infrastructure components. A state of the art, including a taxonomy and definition of SAR, is provided by Ducasse and Pollet [15].

Network-based discovery is applied for inventory and asset management, for example, while creating IT outsourcing contracts or to determine license numbers. Network information is used, for example, by Kind et al. [8] or Chen et al. [9] to extract traffic dependencies based on network package contents and timing. Other approaches exist for specific domains of enterprise IT, for example, by Joukov et al. for Java EE applications [12] and storage [10]. *Galapagos* [11] is a template-based approach focusing on mapping the data dependencies and storage locations of application data. There exists a multitude of tools to extract information about a certain operating system or machine, we are naming only a few here: *Httprint* and *Nmap* have already been integrated in our framework to enable identification of Web servers and operating systems, find open ports of running services, and so on. From Linux's *proc*[4] file system, various system information can be read. Multi-platform tools like *Opscode Ohai*[5] or *Puppet Labs Facter*[6] offer only a subset of information but across different platforms. Menzel et al. [28] use Ohai to extract the configuration (e.g., installed packages) of Amazon EC2 virtual machine images. On the other hand, network and systems monitoring solutions such as *Nagios*[7] or Zenoss[8] provide additional valuable information. The number of solutions providing information for particular aspects or classes of components points out the need to integrate and consolidate them, as we demonstrated with two examples.

A generic discovery framework for technical details, aimed at analyzing variations in the application configuration, is discussed by Machiraju et al. [7]. It uses template models representing the structure and properties of the application as well as how to retrieve them. The discovered model is used to analyze variations in the configuration of the application instances to simplify installation and customization. Ritter [29] presents an approach to discover the business network built on the IT, but not reflecting the technical level in depth. Similarly to our approach, *TADDM*[9] uses agent-less discovery to create an application topology and provides analytics functionality based on this data. However, TADDM only detects few well-known enterprise software packages [28] and requires templates to discover business applications. A study at our institute [30] evaluated 20 commercial and open-source products/tools for enterprise topology discovery and related fields. An approach, like ours, covering enterprise IT from business process to infrastructure offering an extensible framework for existing solutions and enabling developers to provide plugins has not been published.

[4]https://www.kernel.org/doc/Documentation/filesystems/proc.txt
[5]http://docs.opscode.com/ohai.html
[6]http://puppetlabs.com/puppet/related-projects/facter/
[7]http://www.nagios.org/
[8]http://www.zenoss.org/

VII. Conclusions

The lack of an automated way to discover Enterprise Topology Graphs (ETG) causes a lack of insight into the enterprise IT. This might lead to wrong decisions, resulting in problems with severe negative impact on the business. In this paper, we proposed an approach to discover and maintain ETGs which help to adapt, analyze, and optimize enterprise IT. The presented approach enables the integration of existing tools into a unified model by reconciling information of different sources. We implemented a framework realizing the approach and 21 discovery plugins. We validated and evaluated our results by discovering four scenarios. In the future, we envision the application of ETGs in the field of due diligence, compliance, and analysis in general. Automating the discovery of ETGs, as presented in this paper, enables the efficient usage of the previously presented and future fields of application.

Acknowledgments

This work was partially funded by the BMWi project CloudCycle (01MD11023). The authors thank Jakob Krein for his part in the implementation.

References

[1] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm, "Formalizing the Cloud through Enterprise Topology Graphs," in *Proceedings of 2012 IEEE International Conference on Cloud Computing*, 2012.

[2] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch, "How to adapt applications for the Cloud environment," *Computing, Springer*, December 2012.

[3] T. Binz, F. Leymann, A. Nowak, and D. Schumm, "Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies," in *Proceedings of 16th IEEE International Enterprise Distributed Object Computing Conference*, September 2012.

[4] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Pattern-based Runtime Management of Composite Cloud Applications," in *Proceedings of the 3rd International Conference on Cloud Computing and Service Science*. SciTePress, 2013.

[9]http://ibm.co/K9IDni

[5] A. Nowak, T. Binz, F. Leymann, and N. Urbach, "Determining Power Consumption of Business Processes and their Activities to Enable Green Business Process Reengineering," in *Proceedings of 17th IEEE International Enterprise Distributed Object Computing Conference*, September 2013.

[6] T. Binz, F. Leymann, and D. Schumm, "CMotion: A Framework for Migration of Applications into and between Clouds," in *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications*, 2011.

[7] V. Machiraju, M. Dekhil, K. Wurster, P. Garg, M. Griss, and J. Holland, "Towards generic application auto-discovery," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2000.

[8] A. Kind, D. Gantenbein, and H. Etoh, "Relationship discovery with netflow to enable business-driven it management," in *Proceedings of Business-Driven IT Management*, 2006.

[9] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions," in *Proceedings of USENIX Symposium on Operationg Systems Design and Implementation*, 2008.

[10] N. Joukov, B. Pfitzmann, H. V. Ramasamy, and M. V. Devarakonda, "Application-Storage Discovery," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM, 2010.

[11] K. Magoutis, M. Devarakonda, N. Joukov, and N. G. Vogl, "Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems," *IBM Journal of Research and Development*, vol. 52, no. 4.5, 2008.

[12] N. Joukov, V. Tarasov, J. Ossher, B. Pfitzmann, S. Chicherin, M. Pistoia, and T. Tateishi, "Static Discovery and Remediation of Code-Embedded Resource Dependencies," in *Proceedings of IFIP/IEEE Symposium on Integrated Network Management*, 2011.

[13] M. Buschle, H. Holm, T. Sommestad, M. Ekstedt, and K. Shahzad, "A tool for automatic enterprise architecture modeling," in *IS Olympics: Information Systems in a Diverse World*. Springer, 2012, vol. 107.

[14] M. Farwick, R. Breu, M. Hauder, S. Roth, and F. Matthes, "Enterprise architecture documentation: Empirical analysis of information sources for automation," in *46th Hawaii International Conference on System Sciences*, 2013.

[15] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, 2009.

[16] OASIS, *Topology and Orchestration Specification for Cloud Applications Version 1.0 Committee Specification 01*, March 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html

[17] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – a runtime for TOSCA-based cloud applications," in *ICSOC*, ser. LNCS, vol. 8274. Springer, 2013.

[18] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and D. Schumm, "Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA," in *Proceedings of the 20th International Conference on Cooperative Information Systems*, 2012.

[19] M. Hauder, F. Matthes, and S. Roth, "Challenges for automated enterprise architecture documentation," in *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*. Springer, 2012.

[20] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke, "Requirements for automated enterprise architecture model maintenance - a requirements analysis based on a literature review and an exploratory survey." SciTePress, 2011.

[21] C. Batini and M. Scannapieco, *Data Quality: Concepts, Methodologies and Techniques (Data-Centric Systems and Applications)*. Springer, 2006.

[22] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of Management Information Systems*, 1996.

[23] J. Ludewig and H. Lichter, *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Dpunkt.Verlag GmbH, 2010.

[24] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 03, May 2012.

[25] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*. Springer, 2008.

[26] K. Winter, S. Buckl, F. Matthes, and C. M. Schweda, "Investigating the state-of-the-art in enterprise architecture management method in literature and practice," in *Proceedings of Mediterranean Conference on Information Systems*, 2010.

[27] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke, "Automation processes for enterprise architecture management," in *Proceedings of 15th IEEE International Enterprise Distributed Object Computing Conference Workshops*. IEEE Computer Society, 2011.

[28] M. Menzel, M. Klems, H. A. Lê, and S. Tai, "A configuration crawler for virtual appliances in compute clouds," in *IEEE International Conference on Cloud Engineering*, März 2013.

[29] D. Ritter, "From network mining to large scale business networks," in *Proceedings of the 21st International Conference Companion on World Wide Web*. ACM, 2012.

[30] S. Bahle, C. Endres, and M. Fetzer, "Evaluierung von Ansätzen zur Identifizierung und Ermittlung der Enterprise IT in Forschung und Produkten," Student Report 178, Software Engineering, University of Stuttgart, June 2013.

All links were last followed on October 8, 2013.