Institute of Architecture of Application Systems

Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies

Tobias Binz, Frank Leymann, Alexander Nowak, David Schumm

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

BibTeX:

```
@inproceedings {INPROC-2012-21,
    author = {Tobias Binz and Frank Leymann and Alexander Nowak and David Schumm},
    title = {{Improving the Manageability of Enterprise Topologies Through
        Segmentation, Graph Transformation, and Analysis Strategies}},
    booktitle = {Proceedings of 2012 Enterprise Distributed Object
        Computing Conference (EDOC)},
    publisher = {IEEE Computer Society Conference Publishing Services},
    month = {September},
    year = {2012}
}
```


Universität Stuttgart
Germany

Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies

Tobias Binz, Frank Leymann, Alexander Nowak, David Schumm
Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

Abstract—**Software systems running in an enterprise consist of countless components, having complex dependencies, are hosted on physical or virtualized environments, and are scattered across the infrastructure of an enterprise, ranging from on-premise data centers up to public cloud deployments. The resulting topology of the current IT landscape of an enterprise is often extremely complex. We show that information about this complex ecosystem can be captured in a graph-based structure, the enterprise topology graph. We argue that by using such a graph-based representation many challenges in Enterprise Architecture Management (EAM) can be tackled through the aid of graph processing algorithms. However, the high complexity of an enterprise topology graph is the main obstacle to this approach. An enterprise topology graph may consist of millions of nodes, each representing an element of the enterprise IT landscape. Further, these nodes comprise a large variety of properties and relationships, making the topology hardly manageable by human users and software tools. To address this complexity problem, we propose different mechanisms to make enterprise topology graphs manageable. Segmentation techniques, tailored to specific use cases, extract manageable segments from the enterprise topology graph. Based on a set of formally defined transformation operations we then demonstrate the power of the approach in three application scenarios.**

Keywords-enterprise topology; enterprise topology graph; EAM; topology abstraction; segmentation; aggregation.

I. INTRODUCTION

Information technology and corresponding software systems are an important factor for the competitiveness of today's enterprises. For example, solutions like Business Process Management Systems (BPMS) help in managing the processes that drive an enterprise. BPMS support the automation of business processes and workflows and aim at structuring and optimizing reoccurring human tasks. Furthermore, through the use of new technologies the cost of business operation can be decreased, operations can be accelerated, and business-to-business relationships can be made more flexible. For instance, the efficient use of cloud computing is currently considered as one of the key success factors for enterprises [10]. However, through the increasing use of IT in almost any part of an enterprise, the management of the emerging IT landscape becomes a difficult challenge, as its complexity also increases steadily [9].

Enterprise Architecture Management (EAM) deals with the complexity of today's enterprise IT landscapes. EAM considers different layers of an enterprise, capturing the business, processes, integration, software, and infrastructure in models on different levels of abstraction to support business-IT alignment, transformation, and maintenance [6]. As EAM is being recognized as a major challenge for an enterprise, a variety of management approaches and corresponding methods has been developed. However, a commonly accepted reference model has not yet evolved from these efforts and, furthermore, the different EAM approaches vary significantly in terms of granularity and scope, as concluded by Winter et al. [7]. Another result of this study is that software tools are an important aspect in EAM to capture and visualize information. EAM tools like Iteraplan EAM [8], for example, can be used to model the topology of an enterprise IT landscape, plan transformations, and run advanced reporting functions on different aspects of such a topology. However, the information captured with these tools is often (re-)modeled manually and techniques to analyze such enterprise architecture descriptions are rather informal [7]. Due to limited human resources and the costs of maintaining such a topology, typically a high level of abstraction is applied. Hence, manually modeled topologies only map a fraction of the whole enterprise IT onto such a model. For instance, a Content Management System (CMS) can be abstractly represented as one node in a topology, or as a complex graph consisting of several dozen nodes and edges which represent components of that CMS, their relationships, and dependencies. When modeling the topology manually, such level of detail can hardly be achieved in day-to-day practice. To address the shortcomings of modeling the enterprise IT landscape manually, we advocate another option that builds on automated discovery and pre-modeled application templates that reveal the inner structure of complex architecture components. The resulting complex enterprise topology graph allows applying certain transformation operations presented in this paper to be able to address the different information needs of various stakeholders. As an example, imagine a manager who wants to know all IT-systems that are used within his department including the dependencies to other systems or departments.

The contributions of this work are (i) a formal definition of segments and two segmentation techniques, (ii) a set of transformation operations to support the management of complex and large-scale enterprise topology graphs, and (iii) different analysis strategies, which use the contributions (i) and (ii) to address relevant challenges in EAM.

The remainder of this paper is structured as follows: We present the enterprise topology graph in Section II. In Section III segments and segmentation of the enterprise topology graph are discussed. The transformation operations on the formal enterprise topology graph and its segments are presented in Section IV. Building on these fundamentals, we exemplify the approach along three analysis strategies that address the EAM concerns *Impact Analysis* in Section V.A, *Workflow Deep-Dive* in Section V.B, and *Abstract Enterprise Architecture* in Section V.C. Related work is reviewed in Section VI. In the conclusion and outlook in Section VII we reflect on the benefits and challenges of our approach and give a brief overview on future work.

II. ENTERPRISE TOPOLOGY GRAPH

An enterprise topology represents a snapshot of all services and applications in an enterprise, together with their supporting infrastructure and relations. Figure 1 shows a complete view of our approach and the focus in this paper.

A. Buiding up an Enterprise Topology Graph

The central artifact of our approach shown in Figure 1 is the enterprise topology graph. We have identified different methods to build up the topology: it can either be manually modeled, automatically discovered, imported from existing application descriptions, or it can be built up by applying an arbitrary combination of these. The manual modeling of a topology comes with the drawbacks we have already discussed in the introduction. The rather informal descriptions as well as different abstraction levels harm the creation of a holistic topology.

The modeling can be complemented with the automated discovery of an enterprise IT landscape. This is not a novel field and has been brought up over a decade ago in the context of network topology discovery. For example, Machiraju et al. [11] present an auto-discovery engine to overcome the disadvantages of manual discovery. In contrast to automated methods, Machiraju et al. characterize manual discovery as time consuming and inconsistent with a lack of reuse and the problem of distributed intelligence. Analogous to a search engine crawler that discovers the internet, topology discovery automatically analyzes the interiors of applications to get insight into its constituent parts and the relationships among them as well as the relationships to other components in the IT landscape. However, for discovering the relationships of particular applications, like a BPEL process or a Perl script, particular language-specific algorithms and processing functions are required. Such application-specific relation discovery is not state of the art.

Another way that we propose to discover an application landscape is to use existing application descriptions of deployed applications. This, however, assumes that the application that is known to be used in the enterprise has already been modeled using a nonproprietary format, for instance provided by the application vendor. In the context of cloud computing recently a specification for application models and their management has been proposed for standardization, which is very relevant to this aspect. TOSCA [5], the Topology and Orchestration Specification

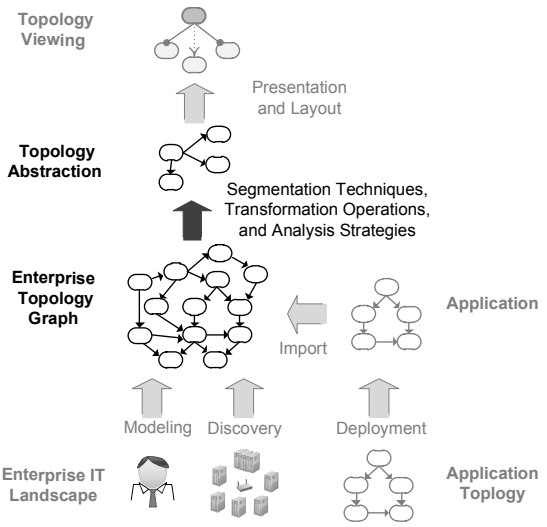

Figure 1. Overview of the enterprise topology approach. The areas this paper is focusing on are highlighted in black.

for Cloud Applications, describes a format to model an application, its constituent parts, and their relationships. The aim of the specification is to ease cloud portability, enable automated deployment of software on an enterprise scale, and to ease and automate the management of applications. We argue that such application models can be another way to build up the enterprise topology graph by importing the information and thus delivering a representation that is more detailed than manual modeling and more precise and easier to handle than by using discovery only. Further, the import of models increases reuse and fosters consistency, for example, when importing the model of a CMS multiple times to represent different deployments of that software in different data centers of a company.

Without going deeper into the challenges and methods of enterprise topology discovery and application model description and importing, we can state that the resulting enterprise topology graph may be very complex, containing a number of nodes which is not manageable without appropriate, automated abstraction techniques. These techniques should support analysis and management of the topologies relevant to different information demands and viewpoints of human users. For example, adjusting the level of granularity to abstract from technical details or focusing on a particular part of the enterprise topology graph that represents a particular business unit. The techniques we propose consider enterprise topology graph abstraction, aggregation, and segmentation, that can be coupled together to form complex analysis strategies, serving different information needs and levels of abstraction. The identification and description of these operations forms the main focus of this paper. Other related aspects like topology viewing, language-specific topology discovery, and the import of application models are ongoing work.

B. Formalization of Enterprise Topology Graph

The enterprise topology graph is a formalized graph which we use within this paper to describe snapshots of an enterprise IT. The graph-based representation enables the

definition of formal operations on enterprise topologies in order to reach the abstraction we are aiming for. Additionally, the formalization enables the application of proven graph algorithms to address particular enterprise architecture management problems.

The formal model was defined in our previous work [15], in combination with a search operation that allows querying the graph, and does not represent a contribution of this paper. To make this paper self-contained, we briefly describe the conceptual model of an enterprise topology graph – depicted in Figure 2. The core element of the enterprise topology graph is represented by the *entity* component which subsumes the *nodes* and *edges* of the graph. Each of the generic entities has a type and a number of entity properties. The node types and edge types are structured as tree, which later on allows us to abstract types from a lower to a higher level. A segment refers to a subset of the enterprise topology graph by referencing a number of its entities. The operations, strategies, and objectives on the right side of Figure 2 represent the abstraction capabilities we enable on the enterprise topology graph. We define transformation operations on segments and/or entities which can be composed into more complex transformation operations. The operations are used to implement analysis strategies which fulfill certain management objectives.

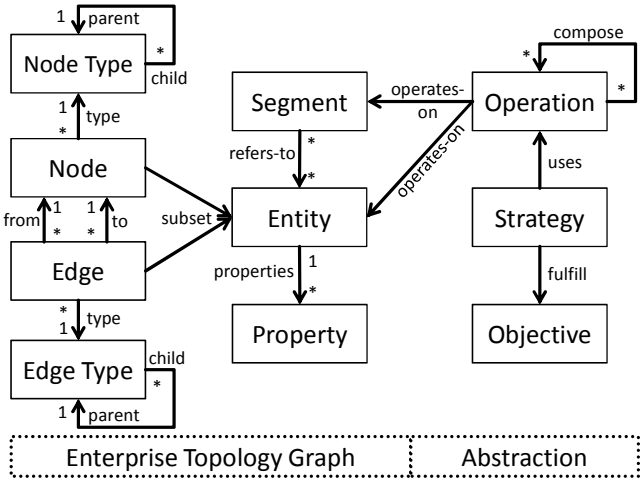

Figure 2. Conceptual model of the enterprise topology graph (left) and the support for abstraction (right).

Formally, we define the set of nodes N and the set of edges $E \subseteq N \times N$. An edge is a binary, directed, typed relation between two nodes. The set of entities subsumes both, nodes and edges, i.e., $Entities = N \cup E$.

The sets $NodeTypes$ and $EdgeTypes$ hold the types which are assigned by the function $type$ to the nodes and edges. Types are structured in a global tree, which is defined by the two functions $parentTypes$ and $childTypes$. In addition, users can define custom type trees with a different structure than the global type trees. We will show examples for custom type trees later in this paper.

$$Types = NodeTypes \cup EdgeTypes$$
$$type: Entities \rightarrow Types$$
$$parentType: Types \rightarrow Types$$
$$childTypes: Types \rightarrow Types,$$
$$p \mapsto \{c | parentType(c) = p\}$$

A property is a key-value-pair which is associated with an entity. Valid keys of a property are all URIs, which are represent by the set $PropertyKeys$. The valid property values are included in the set $PropertyValues$, which are all strings. The function $properties$ assigns a set of key-value-pairs to an entity. The $property$ function returns a single property value for an entity and property key. The label of an entity is stored as property with a well-defined property key.

$$Properties = PropertyKeys \times PropertyValues$$
$$properties:$$
$$Entities \rightarrow Properties,$$
$$e \mapsto \{(k, v) | k \in PropertyKeys \wedge v \in PropertyValues\}$$
$$property:$$
$$Entities \times PropertyKeys \rightarrow PropertyValues,$$
$$(entity, key) \mapsto \{v | (key, v) \in properties(entity)\}$$

Based on the preceding definitions the enterprise topology graph is a 5-tuple of the set of nodes N, edges E, and $Types$, as well as the functions $properties$ and $type$:

$$EnterpriseTopologyGraph = (N, E, Types, type, properties)$$

C. Example of an Enterprise Topology Graph

We provide a small extract of an enterprise topology graph in Figure 3. This graph consists of a *Web service*, an *application server*, and a relational database management system (*RDBMS*) node. The two outgoing edges of the *Web service* represent the *hosted-on* relation to the *application server* and the *depends-on* relation to the *RDBMS*. In addition, the *Web service* has a property *name*.

$$N = \{WS, AS, DB\}$$
$$E = \{(WS, AS), (WS, DB)\}$$
$$NodeTypes = \{WebService, AppServer, RDBMS\}$$
$$EdgeTypes = \{hosted\text{-}on, depends\text{-}on\}$$
$$type = \{(WS, WebService), (AS, AppServer),$$
$$(DB, RDBMS), ((WS, AS), hosted\text{-}on),$$
$$((WS, DB), depends\text{-}on)\}$$
$$properties = \{(WS, ('name', 'StockService'))\}$$

As mentioned in the introduction we do not focus on topology viewing and, therefore, use an ad-hoc notation as defined in the legend of Figure 3.

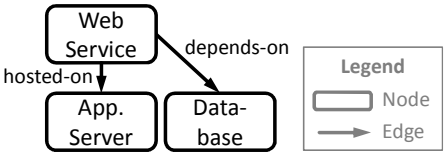

Figure 3. Visualized example enterprise topology graph.

III. SEGMENTATION

Segments are used to refer to a part of the enterprise topology graph based on user selection or logical, organizational, or physical criteria, for example. Segments reference a subset of the nodes and edges in an enterprise topology graph. Segments may also overlap and changes to

the entities contained in one segment are immediately reflected in all other segments containing this entity. As segments are just references, changes also have an effect on the enterprise topology graph. Following this definition, the whole enterprise topology graph, a single node, or a single edge are segments which ensures general applicability of the operations defined later. We define a segment as follows and $Segments$ as the powerset containing all segments:

$$Segment = (SN \subseteq N, SE \subseteq E, Types, type, properties)$$
$$Segments = 2^{Segment}$$

In addition, we define the border of a segment to contain the entities which connect the segment to its outside. This includes the edges crossing the segment border, i.e., the edge between a node in the segment with a node outside of the segment, and nodes which are target or source of an edge that is not part of the segment.

$$border: Segments \rightarrow Entities,$$
$$S \mapsto \{(f,t) | (f,t) \in SE \wedge (f \notin SN \vee t \notin SN)\} \cup$$
$$\{n | n \in SN \wedge (\exists (n,t) \in E: t \notin SN \vee \exists (f,n) \in E: f \notin SN)\}$$

In the following, we will first discuss the properties of segments and afterwards present two segmentation techniques used to define higher level analysis strategies. In the future, further segmentation techniques are likely to be defined, such as one based on clustering algorithms, logical operations, or based on a selection language.

A. Properties of Segments

To apply existing graph algorithms to segments we define possible properties of segments to be able to specify pre- and post-conditions of these algorithms. Segment properties denote what kind of segment the transformation operations and analysis strategies expect as input and which properties they assure for their output.

(i) Connected: All entities of the segment are connected, i.e., no isles exist. When checking if the segment is connected we regard the segment as undirected graph, i.e., to prove connectedness an algorithm may walk in the opposite direction of a directed edge. For a formal definition how to check if a graph is connected see [16].

(ii) Acyclic: Some graph algorithms cannot operate on cyclic graphs or less efficient algorithms must be used. There are various ways to check directed graphs for cycles, e.g., the one described in [16].

(iii) Complete: Let SN and SE be the set of nodes and edges of segment S and let E be the set of edges of the enterprise topology graph. A segment is complete if for each pair of nodes $(f,t) \in SN \times SN$ in S applies: If these nodes are connected through an edge in the enterprise topology graph, i.e., $(f,t) \in E$, then this edge must also be in segment S, i.e., $(f,t) \in SE$. Completeness is defined in the context of enterprise topology graphs and therefore differs from other completeness definitions in graph research, like [16].

(iv) Single-Entry-Single-Exit (SESE): A connected segment S has the property SESE if exactly two entities are in its border, i.e., $|border(S)| = 2$, and one entity is crossing the border into S and the other entity is crossing the border out of S. This is validated for edges (1) and nodes (2) differently: (1) Let $(f,t) \in border(S)$ be an edge in the border of S, then the edge (f,t) is crossing the border out of S if the from-node is in the segment and the to-node is not, i.e., $f \in S \wedge t \notin S$. (2) A node $n \in border(S)$ is crossing the border out of S if there exits exactly one edge to a node outside of the segment, i.e., $\exists! (n,t) \in E: t \notin S$. The validation for entities crossing the border into the segment is analogously to the definitions before.

(v) Backtraceable: A segment is traceable back to the enterprise topology graph if no operation changed the segment in a way that a clear mapping to the enterprise topology graph was lost, for example, when removing a node. When a segment is backtraceable it is possible to maintain links into the enterprise topology graph and to apply changes made on segments back into the enterprise topology graph. This property is true for all newly created segments and maintained if all the operations applied to this segment conserve backtraceability.

B. Technique #1: Node Neighborhood

When working on a particular node of the enterprise topology graph, often only a limited segment surrounding this node is of interest. Therefore, this segmentation technique creates a segment of a node's *neighborhood* to enable the stakeholders to focus on their current task.

Let f be the node whose neighborhood should be determined, let $R = \{f\}$ be the result segment, let $depthOut$ and $depthIn$ be the number of edges the algorithm follows to create the segment, and let $TypesOut \subseteq EdgeTypes$ and $TypesIn \subseteq EdgeTypes$ be the set of types to restrict the edges to be followed. If a type set is empty, the set of all edge types is assigned to it, i.e., for $TypesOut$: $if\,(|TypesOut| = 0)\; TypesOut = EdgeTypes$. Further, let E be the set of edges and N be the set of nodes of the enterprise topology graph. For each node we store its depth with respect to node f to decide if another edge should be followed or not using $depthmapOut: N \rightarrow int$ and $depthmapIn: N \rightarrow int$. Then, this algorithm is executed:

```
depthMapOut(f) ← 0
while |R| grows do
  foreach node ∈ R do
    if depthMapOut(node) < depthOut then
      foreach (node, to) ∈ E do
        if to ∉ R ∧ type((node, to)) ∈ TypesOut then
          depthMapOut(to)
                ← depthMapOut(node) + 1
          R = R ∪ {(node, to), to}
    # Analogically for depthIn and TypesIn
```

C. Technique #2: Structural Matching

In enterprise topology graphs it is important to locate segments having a certain structure. This might be of interest, for example, when organizations want to determine the occurrence of an application or in consolidation scenarios. In [15] we presented an algorithm to search in enterprise topology graphs which enables the definition of queries as segments. This algorithm maps search in enterprise topology graphs to the problem of sub-graph isomorphism and applies the VF2 algorithm presented in

Cordella et al. [17]. The structural matching segmentation technique is based on this search function. It is invoked with one segment as search space, which may also be the whole enterprise topology graph, and a second segment as search query for which all isomorphic sub-graphs are determined.

IV. Transformation Operations

Transformation operations are functions and algorithms on enterprise topology graphs which facilitate certain transformations on this graph. In this section we define a set of transformation operations to build complex analysis strategies that are described in Section V. It is important to note that parameters of the operations are references to graph objects (*call-by-reference*), i.e., the enterprise topology graph is changed. If the original graph should be kept, a copy of the whole enterprise topology graph or segments thereof must be made upfront, e.g., by using the *detach* operation described in Subsection IV.G. To enable composability of the transformation operations the exact specification of the operations' side effects, pre-, and post-conditions are essential. In the following subsections we first present the possible properties of operations followed by the definition of the different transformation operations.

A. Properties of Operations

To make the effects of compositions of operations easier to understand we define two properties for operations:

(i) Non-Ambiguity: For each input the transformation operation has exactly one set of results. An ambiguous operation has possibly multiple sets of results.

(ii) Backtraceability-conserving: If a segment, to which the operation is applied to, was backtraceable before, it is also backtraceable after the particular operation has been executed. Backtraceability is also a segment property, defined in Section III.A. Thus, if a composition of operations contains at least one operation which is not backtraceability-conserving, then the composite operation will also not satisfy this property.

B. Type Abstraction

Sometimes the actual type assigned to an entity is not of interest for particular application scenarios. For example, a stakeholder is only interested in the ratio of Windows and Linux systems that are running, regardless of the concrete versions. To address such different levels of granularity, node types and edge types of the enterprise topology graph are structured as trees. This structure can be used to abstract the types in a segment into higher level types, as shown in Figure 4. The user provides a custom type tree and a set of selected types pointing into this tree. In Figure 4, the selected types for this example are highlighted in bold. If a child type of a selected type exists in the segment, it is replaced by the selected type. Types without selected type as parent or types not included in the custom type tree are not changed.

Formally, we define the backtraceability-conserving and non-ambiguous operation $abstractTypes$ as follows: Let S be the segment in which the types should be abstracted, let $typeTree$ be the root of the type tree, and let $SelectedTypes \subseteq Types$ be the set of types to which their

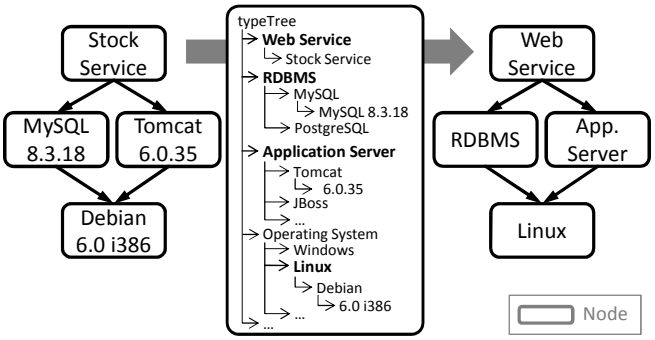

Figure 4. The detailed segment on the left is abstracted into the segment depicted on the right through the type abstraction operation. The type tree in the center shows the structure of types and the selected types in bold.

child types should be abstracted to. Then $abstractTypes$ can be implemented as:

$$\textbf{foreach } entity \in S \textbf{ do}$$
$$t = type(entity)$$
$$\textbf{while } t \neq typeTree \textbf{ do}$$
$$\textbf{if } t \in SelectedType \textbf{ then } type(entity) \leftarrow t$$
$$\textbf{else } t = partentType(t) \textbf{ end}$$

C. Entity Property Aggregation

This operation aggregates for all entities in a segment the property values associated with a certain key. For example, it can be used to determine the average utilization of the entities in a segment. The actual aggregation is done by the user-defined function $faggregate$, which is called with the set of property values and must return exactly one value. Example aggregation functions are *min*, *max*, *avg*, *count*, *concatenate*, or *sum*. These functions are restricted to be used with particular data types only, for example, *min* is restricted to Integer, Long, Single, and Double.

Let S be the segment whose entity's properties with key k should be aggregated. Then we define:

$$faggregate\colon 2^{PropertyValues} \rightarrow PropertyValues$$
$$aggregateProperty\colon$$
$$Segments \times PropetyKeys \rightarrow PropertyValues,$$
$$(S,k) \mapsto faggregate(\{property(e,k) | e \in S\})$$

D. Create and Remove Entities

The operations $newNode\colon Types \rightarrow N$ and $newEdge\colon N \times N \times Types \rightarrow E$ add a new node or edge to the enterprise topology graph, assign a type, and return the entity. To add entities to a segment which already exist in the enterprise topology graph the operations $addNode\colon Segments \times N \rightarrow \perp$ and $addEdge\colon Segments \times E \rightarrow \perp$ are used. The third pair of operations, $newNodeSeg\colon Segments \times Types \rightarrow N$ and $newEdgeSeg\colon Segments \times N \times N \times Types \rightarrow E$, create a new node or edge in the enterprise topology graph and adds them to the segment.

Removing entities is more complex, for example, removing a node may leave half-edges, i.e., edges only attached with one side to a node. Therefore, the basic removing operations do not conserve backtraceability and may render the segment disconnected. Removing nodes from a segment is done using $rmNodeSeg\colon Segments \times N \rightarrow \perp$

and removing edges by $rmEdgeSeg: Segments \times E \rightarrow \perp$. When a node or edge is removed from the enterprise topology graph using the operations $rmNode: N \rightarrow \perp$ and $rmEdge: E \rightarrow \perp$, it is also removed from all segments.

There are different strategies to facilitate the readjustment of half-edges when removing a node. We defined two of them as operations: (1) The operation $rmNodeOpaque$ removes a node by replacing it with a node of type *opaque*. With this, the structure of the segment, as well as the edges, is fully conserved. In contrast to the first operation (2) removes the node and all of its edges. For readjustment new edges of type *relation* are added to show that there has been some kind of indirect relationship between these nodes. The removed node and edges are stored as property of the new edges for documentation purposes. The operation $rmNodeKeepRelations$ is defined as follows: Let N be the set of nodes in the enterprise topology graph, let $rm \in N$ be the node to remove, let S be the segment in which rm should be removed, and let SE be the set of edges in S. Then, the sets of nodes related to rm by an edge is $toNodes = \{toNode | (rm, toNode) \in SE\}$ and $fromNodes = \{fromNode | (fromNode, rm) \in SE\}$. In the algorithm these nodes are pairwise connected by a new edge of type *relation*:

$\boldsymbol{foreach}\ (from, to) \in fromNodes \times toNodes\ \boldsymbol{do}$
  $newEdge(from, to, relation)$

Afterwards the node rm and all edges from or to rm are removed from the enterprise topology graph.

E. Segment Aggregation

If details represented in the enterprise topology graph are not important for a particular application scenario, the complexity can be reduced by aggregating a segment into a single node (e.g., a composite application) or edge (e.g., a communication or dependency channel). An example is shown in Figure 5 where the infrastructure is aggregated into a single node named *supporting infrastructure*. Creating the segment is not part of this operation; this is done by one of the segmentation techniques presented in Section III.

If needed, entity properties can be aggregated from the segment to the aggregated entity with the operation $aggregateProperties$ defined in Section IV.C.

Let S be the connected and complete segment which should be aggregated and $aggregateType \in Types$ be the type of the new aggregated node or edge. Then we define two algorithms, one to aggregate S into one node and the second to aggregate S into one edge. Both algorithms do not conserve backtraceability as defined in IV.A. After executing one of the algorithms S is not connected to the outside anymore and all nodes and edges in S are removed.

(1) $AggregateToNode$ requires $border(S)$ to contain only edges. The algorithm creates the aggregated node and reassigns all edges in $border(S)$ from the nodes in S to the $aggregatedNode$.

$aggregatedNode = newNode(aggregateType)$
$\boldsymbol{foreach}\ (from, to) \in border(S)\ \boldsymbol{do}$
  $\boldsymbol{if}\ from \notin S\ \boldsymbol{then}\ from \leftarrow aggregatedNode\ \boldsymbol{end}$
  $\boldsymbol{if}\ to \notin S\ \boldsymbol{then}\ to \leftarrow aggregatedNode\ \boldsymbol{end}$
$\boldsymbol{end}$

(2) $AggregateToEdge$ requires $border(S)$ to contain only nodes and in contrast to (1) that S is an Single-Entry-Single-Exit segment, as defined in III.A. It creates the aggregated edge between the two nodes in $border(S)$. The inner foreach-loop determines the direction of the edge between the two nodes.

$\boldsymbol{foreach}\ node \in border(S)\ \boldsymbol{do}$
  $\boldsymbol{foreach}\ (f, t) \in S\ \boldsymbol{do}$
    $\boldsymbol{if}\ f = node\ \boldsymbol{then}\ from \leftarrow node\ \boldsymbol{end}$
    $\boldsymbol{if}\ t = node\ \boldsymbol{then}\ to \leftarrow node\ \boldsymbol{end}$
  $\boldsymbol{end}$
$\boldsymbol{end}$
$newEdge(from, to, aggregateType)$

F. Filter

In some cases only certain nodes and edges are of interest, for example, one does not want to have nodes of type *OS* and its child types in a segment, as shown in Figure 6, or all entities in a segment should only have certain entity properties. For this scenario we have to distinguish between two cases: (1) filtering entities, i.e., removing certain nodes and edges from a segment, and (2) filtering properties which apply a filter to the properties of an entity.

(1) Filter Entities: Let S be the segment which should be filtered, let N be the set of nodes, and E the set of edges in the enterprise topology graph. Further, let $eval: Entities \rightarrow \{0, 1\}$ be an user-defined function which decides if an entity should be kept (=0) or removed (=1) from S. We use the previously defined operation to remove entities which renders (1) not conserving backtraceability.

Then, the algorithm for $filterEntities$ is defined as follows on the next page:

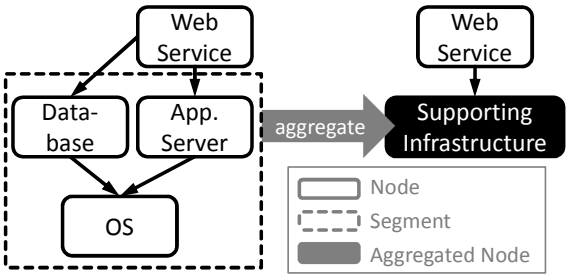

Figure 5. The user-defined segment on the left is aggregated into a single node called *supporting infrastructure* on the right.

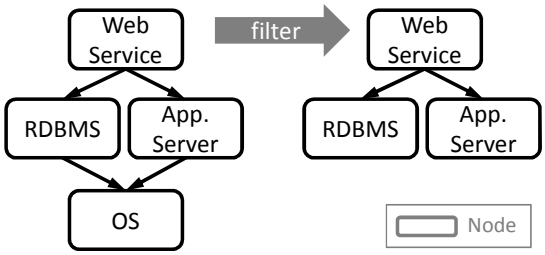

Figure 6. From the enterprise topology graph on the left, the *operating system* node is filtered. All edges are of type *hosted-on*.

$\boldsymbol{foreach}\ entity \in S\ \boldsymbol{do}$
  $\boldsymbol{if}\ eval(entity) = 1 \wedge entity \in N\ \boldsymbol{then}$
    $rmNodeKeepRelation(S, entity)$
  $\boldsymbol{else\ if}\ eval(entity) = 1 \wedge entity \in E\ \boldsymbol{then}$
    $rmEdgeSeg(S, entity)$

The exemplary implementation of function $eval$ we applied in Figure 6 filters all nodes of type OS:

$$eval\colon entity \mapsto type(entity) \in \{OS\}$$

(2) The filter for entity properties is defined in the same way as $filterEntities$ but with a different evaluation function $eval\colon PropertyKey \times PropertyValue \rightarrow \{0,1\}$, which decides if an property is kept (=0) or removed (=1). The algorithm $filterProperties$ is defined as follows:

$\boldsymbol{foreach}\ entity \in S\ \boldsymbol{do}$
  $\boldsymbol{foreach}\ (key, value) \in properties(entity)\ \boldsymbol{do}$
    $\boldsymbol{if}\ eval((key, value)) = 1\ \boldsymbol{then}$
      $property(entity, key) \leftarrow \perp$ # *Remove property*

The evaluation function enables the usage of different decision mechanisms, like regular expressions on the property key or other complex evaluations.

G. Detach Segment

In general, changes of nodes and edges are reflected in the enterprise topology graph and in all segments containing these entities, because segments only reference the entities of the enterprise topology graph. For transformations to be only reflected in the segment the respective operation is applied to, the segment must be detached before. Technically, the segment is copied and the entities are not bound to the enterprise topology graph anymore. The base of the segment is a new enterprise topology graph which contains only the nodes the detached segment refers to. Due to the fact that this is an out-of-band operation which cannot be described in our formal model, we are referring to it by its signature: $detach(Segment\ s)\colon EnterpriseTopology$

V. Analysis Strategies

An analysis strategy composes multiple transformation operations and segmentation techniques to provide the appropriate level of abstraction for the specific application scenario. During composition, the properties of segments and operations are used to determine the assurances of the resulting segments. For example, an assurance can be that changes to a resulting segment can be propagated automatically back into the enterprise topology graph.

In this section we define three analysis strategies to address selected EAM problems. Each strategy is defined in terms of its objectives, the problem it addresses, a solution sketch, and a formalization composing previously defined transformation operations and segmentation techniques.

A. Strategy 1: Impact Analysis

Objectives: The objective of this strategy is to determine the importance of certain elements in an enterprise topology. The importance, i.e., how strong an element is related to business-critical elements, may be identified by discovering the dependencies between the entities of this enterprise topology graph.

Problem: In complex enterprise topologies the consequences of node failures, i.e., of components in the IT landscape or changes of configurations are not foreseeable. This is due to the fact that it is not clear which nodes depend on another node, especially when taking into consideration all depending nodes, i.e., at arbitrary depth. For changes in Web services, for example, it is useful to have a clear insight which applications and workflows use a certain service that should be updated as well.

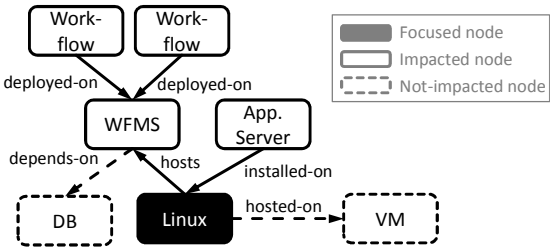

Figure 7. Sketch of *Impact Analysis* strategy showing focused, impacted, and not-impacted nodes of an example enterprise topology graph segment.

Solution: The *Impact Analysis* in enterprise topology graphs determines which nodes, named *impacted nodes*, are depending on a specific node of interest, named *focused node*. As shown in Figure 7, the result is a segment containing the focused node together with its neighborhood, the impacted nodes. Impacted nodes are connected to the focused node through a configurable number (i.e., the *depth* parameter) of edges of configurable types. The *DB* and *VM* node, depicted by a dashed line in Figure 7, exemplarily show two nodes not impacted and, therefore, would normally not appear in the result segment. This impact analysis strategy provides an implementation for the *Infrastructure Failure Impact Analysis* pattern in [20] and broadens its analysis scope to all levels of enterprise topology, not only infrastructure. Our strategy is realized by using the segmentation techniques *node neighborhood*, defined in Section III.B, together with a set of edge types defined in this strategy. Figure 7 shows the different edge types and their directions.

Formalization: This strategy configures the parameters for the node neighborhood segmentation technique, formally defined in Section III.B, as follows:

$$TypesIn = \{depends\text{-}on, installed\text{-}on, hosted\text{-}on, deployed\text{-}on\}$$
$$TypesOut = \{hosts, executes\}$$
$$depthIn = depthOut = 5$$

When applying the node neighborhood segmentation technique with this configuration and the focused node f being the *Linux* node, we get the segment of impacted nodes as depicted in Figure 7.

B. Strategy 2: Workflow Deep-Dive

Objectives: The application of this strategy aims at identifying how a specific process automation (i.e., a workflow which in most cases represents a business process or parts thereof) is mapped to its underlying infrastructure. This information might be important when, for example,

analyzing infrastructure migration possibilities or when determining the environmental impact of a business process.

Problem: For example, when tackling the challenge of Service-oriented Architecture governance [19], the analysis of services and their orchestrations in workflows plays a central role. To analyze workflows, e.g., a service orchestration based on BPEL [13] or BPMN [14], the involved services and their realization have to be considered. The ecological indicators or security properties of a workflow, for example, highly depend on the invoked Web services as well as on the middleware and infrastructure hosting them.

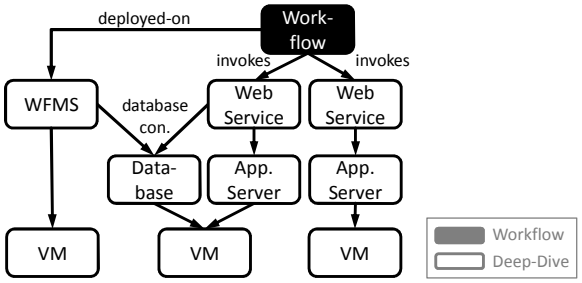

Figure 8. Sketch of *Workflow Deep-dive* strategy showing the workflow and an exemplary set of nodes included in the deep-dive. Edges without an explicit type label are of type *hosted-on*.

Solution: The *Workflow Deep-Dive* into the enterprise topology graph starts with a node representing a workflow. From this node all the outgoing edges of type *invokes* are followed to determine the set of relevant services. In addition, the infrastructure hosting the workflow is added by tracing the edges of the type *hosted-on*. These nodes build up the segment to start the actual deep-dive on, i.e., the first iteration of the result segment. In the following all nodes reachable through a *hosted-on* or *depends-on* edge from one of the nodes in the result segment are added to the result segment. For this strategy we may decide not to include network components and use a filter to remove all nodes representing network components.

Formalization: Let w be the node of type *workflow* for which the deep-dive should be determined, let E be the set of edges in the enterprise topology graph, and let R be the result segment containing the workflow deep-dive. Then, the deep-dive is determined using the following algorithm:

$$R = \{to | (w, to) \in E \\ \wedge type\big((w, to)\big) \in \{invokes, hosted\text{-}on\}\}$$
$$DeepDiveTypes = \{hosted\text{-}on, depends\text{-}on\}$$
$$\textbf{while } |R| \text{ grows } \textbf{do}$$
$$\quad \textbf{foreach } node \in R \textbf{ do}$$
$$\quad\quad \textbf{if } \exists (node, to) \in E\colon to \notin R \\ \quad\quad\quad \wedge type((node, to)) \in DeepDiveTypes \textbf{ then}$$
$$\quad\quad\quad addNode(R, to)$$

To remove all nodes assigned with the type *network* and all child types of it, we define the following evaluation function, and apply it using the $filterEntities$ operation:

$$NW = \bigcup_{i=0}^{\infty} childType^{i}(network)$$
$$eval\colon Entities \rightarrow \{0,1\}, entity \mapsto type(entity) \in NW$$
$$filterEntities(R, eval)$$

C. Strategy 3: Abstract Enterprise Architecture

Objectives: From an EAM perspective, the objective of this strategy is to simplify a given enterprise topology graph to address stakeholders that only need topology information on a high level of abstraction.

Problem: Today, enterprise architecture models are mostly created manually with the problems mentioned in the introduction. When these models are created with automated techniques like discovery and application topology import, the level of abstraction of the resulting enterprise topology graph may not be adequate for some stakeholders. In some application scenarios, a higher level of abstraction is needed.

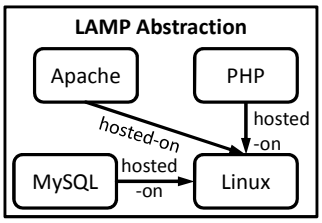

Figure 9. The structure of the LAMP abstraction, containing a Linux, Apache, MySQL, and PHP node.

Solution: The *Abstract Enterprise Architecture* strategy creates a high-level enterprise architecture from a low-level enterprise topology graph. Before this strategy can be applied, a set of abstractions is defined. For example, the *LAMP abstraction* in Figure 9 denotes that the depicted **L**inux, **A**pache, **M**ySQL, and **P**HP node should be aggregated into a single node of type *LAMP*. This strategy is applied to a user-defined segment containing, for example, the nodes of a specific datacenter. In the following we describe the steps of this strategy in detail, as denoted in Figure 10: (1) The types in the LAMP abstraction are extracted and used in (2) as the selected types of a custom type tree in the $abstractTypes$ operation. In step (2), the types of all entities in this segment are abstracted, for example, a *Debian 6.0* node is abstracted to *Linux*. (3) With the structural matching segmentation technique and the LAMP structure as search query, the enterprise topology

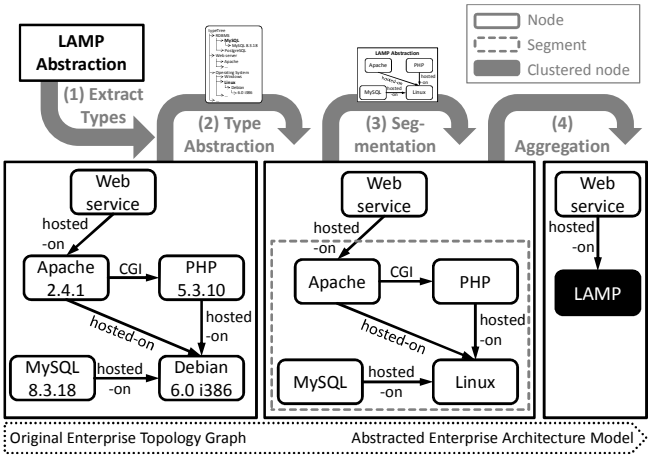

Figure 10. Sketch of the *Abstract Enterprise Architecture* strategy showing the four steps how the enterprise topology graph segment on the left is abstracted towards and enterprise architecture model.

graph with the abstracted types is segmented. The result of the segmentation is a set of segments matching to the pre-defined structure. Figure 10 depicts one result segment in the dashed box. (4) In the end, each segment is aggregated into a single node using the $aggregateToNode$ operation.

When applying a series of these abstractions we are able to extract, abstract, and automatically generate an enterprise architecture from the detailed enterprise topology graph.

Formalization: Let segment E be the section of the enterprise topology graph which should be abstracted, let segment $Abstraction$ be the LAMP abstraction, and $typeTree$ the root of the tree of node types.

(1) The set $SelectedTypes$, which is input of the $abstractTypes$ operation, is defined as $SelectedTypes = \{type(n) | n \in abstraction\}$. In the example presented in Figure 9 $SelectedTypes$ is $\{Linux, Apache, MySQL, PHP, hosted\text{-}on\}$.

(2) All entities with a type which is the child of one of the types in $SelectedTypes$ is changed to this type by the transformation operation $abstractTypes$.

$$abstractTypes(E, typeTree, SelectedTypes)$$

(3) The structural segmentation technique is used to locate all segments matching the search query $Abstraction$. For the example depicted in Figure 10, the resulting segment is highlighted by the dashed box.

$$R = StructuralSegmentation(E, Abstraction)$$

(4) To each of the resulting segments we apply the operation $aggregateToNode$:

$$\textbf{\textit{foreach}}\ s \in R\ \textbf{\textit{do}}\ aggregateToNode(s, LAMP)\ \textbf{\textit{end}}$$

VI. Related Work

Increasing complexity and corresponding abstraction techniques have been a key challenge in information technology since its early beginnings. To narrow the scope of works related to our approach we focus in the following on approaches that operate on graphs. First, we inspect a work from graph visualization dealing with a very large number of nodes and edges. We then take a closer look at works dealing with process graphs, which only contain hundreds of nodes but which have high demands on readability and semantic correctness of the outcome of the abstraction. Afterwards, we discuss the abstraction of models of software systems in the field of software engineering. Finally, we will have a look on the related work of enterprise topology graphs

Research on graph theory has a long tradition and a large set of proven algorithms with high efficiency exits. These algorithms can be made accessible to EAM through the usage of graphs to represent the enterprise IT landscape. However, a graph holding all details of an enterprise IT may contain millions of nodes and edges, demanding for graph abstraction algorithms, as we proposed in this paper. This characteristic makes approaches on visualizing large-scale graphs relevant to our work. Abello et al. [3] presented *ASK-GraphView*, a system to efficiently visualize graphs up to 200,000 nodes. The main idea of the approach is to construct a hierarchy on an arbitrary graph using a pipeline of the clustering algorithms *Peeling*, *Biconnected Components*, *Markov Cluster Algorithm*, and *Contraction*. However, labeling of created clusters is recognized as an important aspect. Maqbool et al. [12] present a generic algorithm to address this problem by using the frequency and inverse frequency of keywords contained in relevant properties like function identifiers. The work compares automatically calculated labels with labels defined by human experts for the same clusters which showed a high correspondence between both sets of labels and most of the automatically obtained labels are considered as meaningful and helpful for understanding. The aggregation operation we presented in Section IV.E can use this or similar mechanisms to create cluster labels. However, our approach addresses a much broader scope of abstractions beyond clustering and labeling.

As business processes are a crucial factor for the success of an enterprise, it is of utmost importance to have a clear understanding of these processes. Due to the increasing complexity of process models, which are mainly investigated in the field of BPM, abstraction techniques of process graphs are gaining more and more importance. For instance, Sadiq et al. [2] presented an approach to make the analysis of complex processes more efficient through the use of graph reduction rules. Many further approaches targeting the abstraction of process graphs have been proposed in the meanwhile. In [4], the graph transformations that have been applied frequently in the state of the art of process abstraction have been described. These transformations range from structural abstraction like the omission and aggregation of process structures, over to different forms of information augmentation like semantic tagging, up to viewing functions that address the visualization of a process like graphical highlighting and usage of particular shapes. Regarding the structural transformation patterns, the enterprise topology approach applies all patterns except for the *disconnected aggregation*, which describes the aggregation of unconnected or transitively connected segments. Patterns concerning the augmentation of additional information from external data sources as well as concerning the graphical visualization need to be considered in further elaboration of the enterprise topology approach.

For the abstraction of software models, Selic [1] proposed a set of common abstraction patterns frequently used by software architects to make complex system models more comprehensible. The patterns concerning abstraction of structure are (1) *black box* which abstracts a complex structure to a single component, (2) *black line* which abstracts a chain of components that serve as communication channel to a single edge, (3) *cable* which abstracts multiple edges to a single one called cable, (4) *port group* which abstracts a group of communication ports of a component to a single one called port group, and (5) *platform layer* which abstracts a connected structure to a coarse-grained unit by combining use of the black box and cable pattern. Selic also discusses refinement of abstracted models as inverse to applied patterns. The patterns Selic identified also apply to enterprise topology abstraction. The transformation operations we proposed cover all patterns and extend the set proposed by Selic by filtering of components and relations, as well as by fine-granular abstraction on property level.

In [15] we evaluated the related work of enterprise topology graphs in detail. We would like to stress that in contrast to approaches describing application models [5][21] or architectural blueprints [8][22] an enterprise topology graph depicts a snapshot of the instances in the enterprise topology. When describing this relation in terms of object-oriented programming the former can be seen as classes, the latter as the objects. Therefore, an enterprise topology graph might include a number of instantiated application models.

VII. Conclusions and Outlook

The presented approach helps organizations to reduce the complexity and to improve the manageability of their enterprise topologies through the use of segmentation techniques, transformation operations, and analysis strategies. The proposed abstraction methods address current EAM needs because enterprise topology graphs that contain all components of an enterprise IT, their supporting infrastructure, and corresponding relations may consist of millions of nodes and are growing tremendously.

We argued that offering such analysis strategies contributes to increasing the efficiency and impact of EAM. However, there is some effort required to develop and tailor analysis strategies towards the application scenarios, internal policies, and IT strategy of an organization. Therefore, we propose to create a new role called *Enterprise Information Designer* to support the groups working on the enterprise architecture, a role similar to the *Information Designer* in Business Process Management [18] who specifies different views according to the requirements of different stakeholders of a business process. Enterprise Information Designers develop strategies and create appropriate segmentation techniques and transformation operations based on the information needs, requirements, and challenges of the groups working on the enterprise architecture. This new role can provide the partly lacking adaption to enterprise specifics in EAM [7]. Segmentation techniques, transformation operations, and analysis strategies could be consumed and hosted as a service by external providers, creating a marketplace for the works of Enterprise Information Designers.

We formally defined those strategies, however, they have not been applied to real world enterprise topologies yet. This is due to the fact that the semi-automated creation and especially the automated discovery of enterprise topologies is not a trivial task and prototype implementations are still ongoing work.

The proposed approach is not limited to enterprise topology graphs but also applicable to a wide variety of other graphs. We argue that the operations presented in this paper for enterprise topology graphs may also be applied to application models like *TOSCA Topology Templates* [5].

In future work we will define further analysis strategies to show the application of enterprise topology graphs to further problems of EAM.

Acknowledgment

This work was partially funded by the BMWi project CloudCycle (01MD11023) and Migrate! (01ME11055). D. Schumm would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart.

References

[1] B.V. Selic, “A Short Catalogue of Abstraction Patterns for Model-Based Software Engineering,” International Journal of Software and Informatics, vol.5, no.2, pp. 313–334, 2011.

[2] W. Sadiq and M. Orlowska, “Analyzing Process Models Using Graph Reduction Techniques,” Information Systems, vol.25, no.2, pp. 117–134, Elsevier, 2000.

[3] J. Abello, F. van Ham, and N. Krishnan, “ASK-GraphView: A Large Scale Graph Visualization System,” IEEE Transactions on Visualization and Computer Graphics, vol.12, no.5, IEEE, 2006.

[4] D. Schumm, F. Leymann, and A. Streule, “Process Viewing Patterns,” Proceedings of the 14th IEEE International EDOC Conference, IEEE Computer Socienty, 2010.

[5] “Topology and Orchestration Specification for Cloud Applications (TOSCA),” OASIS, Oct. 2011.

[6] R. Winter and R. Fischer, “Essential Layers, Artifacts, and Dependencies of Enterprise Architecture,” Journal of Enterprise Architecture, pp. 7–18, 2007.

[7] K. Winter, S. Buckl, F. Matthes, and C. Schweda, “Investigating the State-of-the-Art in Enterprise Architecture Management Methods in Literature and Practice,” MCIS 2010 Proceedings, AIS Library, 2010.

[8] Iteratec, “Iteraplan EAM,” online at http://www.iteraplan.de/en, 2012.

[9] J. Garbani, T. Mendel, and E. Radcliffe, “The Writing on IT’s Complexity Wall,” Forrester Research, June 2010.

[10] Gartner, “Gartner Identifies the Top 10 Strategic Technologies for 2011,” Press Release, 2010.

[11] V. Machiraju, M. Dekhil, K. Wurster, J. Holland, M. Griss, and P. Garg, “Towards Generic Application Auto-discovery,” HP Laboratories Palo Alto, July 1999.

[12] O. Maqbool and H. Babri, “Automated Software Clustering: An Insight using Cluster Labels,” The Journal of Systems and Software, vol.19, no.11, pp. 1632–1648, Elsevier, 2006.

[13] “Web Services Business Process Execution Language (BPEL) Version 2.0.,” OASIS specification, 2007.

[14] “Business Process Model and Notation (BPMN) Version 2.0,” Object Management Group specification, Jan. 2011.

[15] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm, “Formalizing the Cloud through Enterprise Topology Graphs,” Proceedings of the 5th International Conference on Cloud Computing (IEEE Cloud), 2012.

[16] G. Valiente, “Algorithms on Trees and Graphs,” Springer, 2002.

[17] L. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (Sub)Graph Isomorphism Algorithm for matching large Graphs,” Pattern Analysis and Machine Intelligence, IEEE Transactions, vol.26, no.10, pp. 1367–1372, Oct. 2004.

[18] D. Schumm, “Information Design for Business Process Management,” Poster in the 5th Summer School on Service Oriented Computing, 2011.

[19] B. Wolf, “Introduction to SOA governanve,” IBM developer works, Jul. 2007.

[20] S. Buckl, F. Matthes, and C. Schweda, “EAM Pattern Catalog,” online at http://wwwmatthes.in.tum.de/wikis/eam-pattern-catalog/.

[21] W. Arnold, T. Eilam, M. Kalantar, A. Konstantinou, A. Totok, “Pattern Based SOA Deployment,” Proceedings of 5th International conference on Service-Oriented Computing (ICSOC), 2007.

[22] D. Garlan, R. Monroe, and D. Wile, “Acme: An Architecture Description Interchange Language,” CASCON First Decade High Impact Papers, ACM, pp. 159–173, 2010.