Improve Resource-sharing through Functionality-preserving Merge of Cloud Application Topologies

Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann and Andreas Weiß
Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany

Keywords: Application Topology, Merge, Resource Sharing, Multi-tenancy, Cloud Computing, TOSCA.

Abstract: Resource sharing is an important aspect how cost savings in cloud computing are realized. This is especially important in multi-tenancy settings, where different tenants share the same resource. This paper presents an approach to merge two application topologies into one, while on the one hand preserving the functionality of both applications and on the other hand enabling sharing of similar components. Previous work has not addressed this due to the lack of ways to describe topologies of composite applications in a decomposed, formal, and machine-readable way. New standardization initiatives, such as TOSCA, provide a way to describe application topologies, which are also portable and manageable. We propose an approach, realization, and architecture enabling a functionality-preserving merge of application topologies. To validate our approach we prototypically implemented and applied it to merge a set of test cases. All in all, the functional-preserving merge is a method to support the optimization and migration of existing applications to the cloud, because it increases resource sharing in the processed application topologies.

1 INTRODUCTION

Resource sharing is, besides automation and economy of scale, the key to realize the significant cost saving promised and realized by cloud computing. Today, resource sharing is mostly discussed in terms of multi-tenancy, which is the fact that tenants, i. e. organizations or users, share one or a common set of resources (Chong and Carraro, 2006). The concept of resource sharing itself has been around for decades, as well as its problems, such as security, performance, availability, and administrative isolation (Guo et al., 2007). Increasing the utilization through resource sharing is, however, often still not applied in enterprise IT. This is reflected by the fact that in today's non-cloud data centers servers usually have a very low utilization[1].

The applications running on this underutilized data centers are no monolithic blocks. Usually, enterprise applications require middleware (including application servers, database management systems, and message queuing) installed on an operating system, which in turn requires computing, storage, networking, and other infrastructure. Working together, these components provide the business functionality implemented by the application. An application topology is a graph containing all these components as separate nodes and their relations as edges. Recent advances in the formalization and management of complex IT systems, for example, the *Topology and Orchestration Specification for Cloud Applications* (OASIS, 2012) (TOSCA) currently standardized at OASIS, are based on a fine-grained topology of the cloud application to enable their automated deployment and management. This formal description of the application's configuration, structure, architecture, and dependencies enables automated processing of these application topologies and is the technical foundation of our approach.

Based on this decomposition of the application, this paper presents an approach to merge two previously separated application topologies into one integrated application topology, while preserving the functionality of the applications. For example, consider two application topologies, each including a Web service, which runs on its own Apache Tomcat on a separate virtual machine. Our approach shows how to evaluate and, if possible, merge these two topologies into one application topology. In this example the result would be one Tomcat hosting both Web services and therefore only requires one virtual machine. Therefore, our approach is a method to increase resource sharing and support the migration of existing applications to the cloud.

[1]For example, according to (Andrzejak et al., 2002) the 80% percentile is typically utilized in the 15 to 35% range.

Binz T., Breitenbücher U., Kopp O., Leymann F. and Weiß A..
Improve Resource-sharing through Functionality-preserving Merge of Cloud Application Topologies.
DOI: 10.5220/0004378000960103
In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science* (CLOSER-2013), pages 96-103
ISBN: 978-989-8565-52-5
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

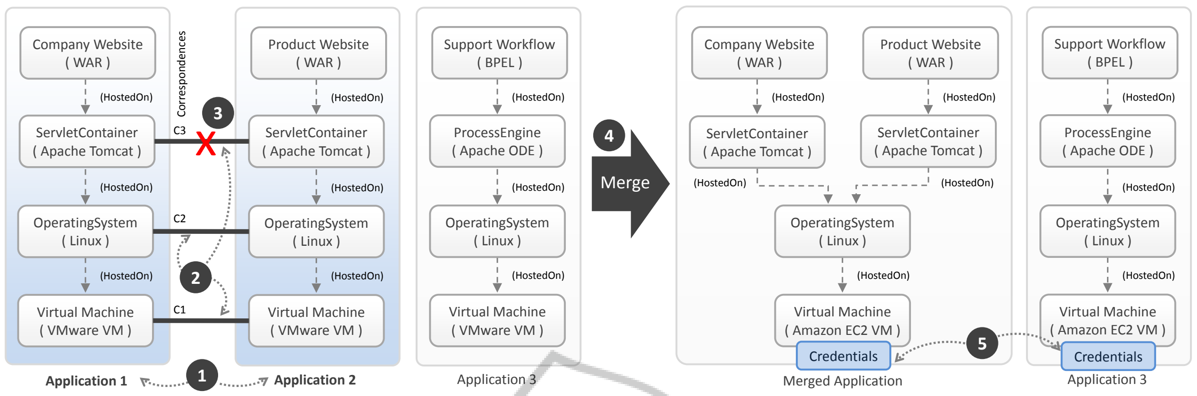
Company Website (WAR)
(HostedOn)
ServletContainer (Apache Tomcat)
OperatingSystem (Linux)
Virtual Machine (VMware VM)
Application 1
Correspondences
C3
C2
C1
1
2
3
Product Website (WAR)
Application 2
Support Workflow (BPEL)
ProcessEngine (Apache ODE)
Application 3
4
Merge
Operating System (Linux)
Virtual Machine (Amazon EC2 VM)
Credentials
5
Merged Application

Figure 1: Motivating Scenario – Migrating the three applications on the left into the cloud on the right.

This leads us to the contributions of this paper, which are threefold: (i) We propose an approach to merge two separated application topologies into one, while preserving their functionality and sharing as much of the components as possible. This approach, which is independent of a particular topology language, describes a method following the five steps: identification, matching, manual evaluation, merging, and final evaluation & deployment. A cloud migration scenario is used to extract the requirements towards a suitable approach. (ii) Based on this, we present a realization called TOSCAmerge which applies our approach to TOSCA-based application topologies. (iii) Finally, we present the architecture of TOSCAmerge, an extensible framework to enable reuse of the functionality-preserving topology merge operation.

The remainder of this paper is structured as follows: First, Section 2 presents the migration of enterprise applications as motivating scenario for our approach. The proposed method to address this scenario is described in detail in Section 3. Based on this, Section 4 shows our prototypical realization of the approach using TOSCA-based application topologies. Section 5 discusses the proposed approach and realization. Section 6 presents related work. We conclude the paper and give an outlook on future work in Section 7.

2 MOTIVATING SCENARIO

This section discusses the motivating scenario for our approach, the migration of enterprise applications to the cloud, which is used throughout the paper to illustrate our approach. In our scenario, a company has three applications, shown as topologies on the left of Figure 1. Application 1 implements the Web site of the company as Java Web application which is packaged as WAR-file. The application is hosted on an Apache Tomcat servlet container which is hosted on a Linux operating system. The operating system runs on a virtual machine provided by a local VMware deployment. The same stack is used by the company to implement their advertising Web site for their main product. The third application has the same infrastructure stack but implements the support workflow of the company as BPEL process which is hosted on an Apache ODE workflow engine.

To reduce cost, the company decides to migrate these customer-facing applications to the cloud, in this case to Amazon EC2. However, running the same supporting infrastructure three times is not efficient. To further reduce cost, the company looks for a solution to automatically merge application topologies. In addition, an important goal is to preserve the functionality provided by the applications. This is where the approach proposed in this paper is aimed to. The merged application topology on the right hand side of Figure 1 is the result of the presented approach. The detailed process from left to right is explained in the following sections.

3 APPROACH

In this section we present our semi-automated approach for merging application topologies. First, we present an analysis of the requirements in Section 3.1. Based on this, we discuss our method in detail in Section 3.2.

3.1 Requirements

Based on the motivating scenario, introduced in Section 2, and a literature study we identified the following requirements for a solution merging application topologies:
Functionality Preservation and Correctness. The functionality of the applications which have been merged

must not be altered. However, the non-functional properties may be different due to changes in the supporting infrastructure. The result, i. e. the merged application topology, must be syntactically and semantically correct. To facilitate this, we demand the input application topologies also to be syntactical and semantically correct. In particular, they have to include all the components, configurations, and relations required to operate the merged application properly.
Open World Assumption. A viable solution for merging application topologies must not assume that there is only a limited and well known set of semantics represented by the nodes and edges in the application topology. The evolution of cloud offerings, software systems, and hardware are changing too fast to restrict the approach upfront to a certain set of types, i. e. a closed world. Thus, the approach must be able to deal with various kinds of nodes and relationships.
Algorithm. The implementation must produce deterministic results, terminate for any valid input, and have a reasonable computational complexity to allow its application also to large application topologies.

3.2 Topology Merge Method

For functionality-preserving merging of application topologies we propose a method which defines five sequential steps, presented in the following subsections: (i) Identify Applications to be Merged, (ii) Matching, (iii) Manual Evaluation, (iv) Merging, and (v) Final Evaluation & Deployment.

Some details of the method depend on the language used to describe application topologies, which is explicitly noted in the respective step of the method. In Section 4, our prototypical implementation *TOSCAmerge* proposes a concrete realization of this method based on TOSCA.

3.2.1 Identify Applications to be Merged

The first step of our approach is identifying the applications to be merged. Good candidates to realize savings through increased resource sharing are large application topologies with common components in the supporting infrastructure. However, the details of the actual identification is out of scope of this paper.

Motivating Scenario. The company may decide that only application 1 and application 2 shall be merged and moved to the cloud, because the availability of the two Web sites is lower prioritized than the availability of their support workflow. This decision is represented by (1) in Figure 1.

3.2.2 Matching

After the applications to be merged are identified, step two identifies possibilities to merge nodes and edges of the application topologies. In the following, we regard a pair of two application topologies T_1 and T_2. The result of this matching step is a set of *correspondences*. Correspondences are binary, undirected overlay edges between two nodes or two edges of the application topologies. One node may have multiple correspondences to different other nodes, this is resolved during merging (see step four).

Matching Nodes. During matching, each pair of nodes (n,m) with $n,m \in T_1 \cup T_2$ is evaluated if the nodes correspond with each other and, therefore, could be merged. We write $n \circ m$ if a correspondence exits between n and m. The existence of a correspondence is evaluated by three checks, which must be all passed to establish a correspondence:
(i) If the types of the nodes are the same, a merge might be possible. How types are defined and which rules for their comparison have to be applied depends on the language used to describe application topologies. Only nodes of the supporting infrastructure, such as hardware, operating systems, and middleware, should be matched. The actual business logic is provided by the nodes representing the application, such as the Java WAR file in Figure 3, which are excluded. This is done because the application logic cannot be merged in a generic way. In contrast, the supporting infrastructure can be merged in a generic way, because it mainly consists out of standardized components configured to operate the application logic. The configuration of the supporting infrastructure must also be considered as some kind of application logic, crucial to operate the application as intended, as presented in the next steps.
(ii) Thereafter, the functional properties of the nodes are checked. For instance, the configuration of a node is a functional property. Functional properties are specific for the respective node type and therefore must be handled by type-specific logic. For example, it is not possible to determine in a generic way if the values *32* and *64* for the property *CPU architecture* are compatible. Some software products may offer versions for both, 32-bit and 64-bit systems, some not. Therefore, our approach uses plugins to integrate the type-specific logic which checks if two nodes (or edges) can be merged or not.
(iii) Nodes may also have non-functional properties (NFP), for instance, the availability of a node. Similar to functional properties, non-functional properties have to be processed by type-specific logic. If the topology language has an inheritance, sub-graph, or grouping mechanism, the NFPs of all parents also must

be considered. In this case, the accumulated NFPs of all parent constructs must also be compatible, e. g. if there is a NFP on the whole topology.

Matching Edges. Edges are matched similar to nodes to generate correspondences. However, only edges connected to at least one node with a correspondence are processed. Evaluating if a correspondence between two edges exists requires that the type-specific logic implementing the semantics of the edge is provided as plugin, similar to the type-specific plugins for nodes. In particular, edges may prevent certain node correspondences. For example, two corresponding nodes may be connected by an *anti-colocation* edge. This means that two nodes must reside on different physical infrastructure to increase availability. As a consequence, the nodes must not be merged.

Motivating Scenario. After the company decided to merge the applications providing the Web sites, the matching step automatically generates the correspondences between these two applications. The result is, that the virtual machines, the operating systems, and the two servlet containers are candidates to be merged. This is denoted by (2) in Figure 1.

3.2.3 Manual Evaluation

The result of the matching step is a set of correspondences between two nodes or two edges. However, we do not think that all decisions can be made by the implementation or its plugins. Therefore, this step allows an architect to evaluate the correspondences with respect to the application and enterprise architecture, as well as the responsible software developer with respect to the internals and hidden assumptions of the applications. This allows incorporating available knowledge of these experts with minimal time investment, because an initial identification and subsequent merging is done by the implementation automatically. The result of this step is a set of correspondences stating which nodes and edges should be merged.

Motivating Scenario. In our scenario the architect decides to run the two web applications on different Web servers, therefore, he removes the respective correspondence. The removal of this correspondence is denoted by (3) and the red *X* in Figure 1.

3.2.4 Merging

This step processes the correspondences found during matching, which may have been manually adapted by the previous manual evaluation step. Merging of nodes and edges is done in three steps:
(i) Merging the two nodes/edges connected through a correspondence by calling the respective type-specific logic to merge the functional and non-functional definitions of the node. Similar to the matching step, the type-specific logic must be provided by a plugin.
(ii) If applicable, reconnect the edges connected to one of the merged nodes. The correspondences are also reconnected accordingly; except that the correspondence whose two nodes/edges have been merged is removed.
(iii) Finally, the two old nodes/edges, which are at this point neither part of an edge nor of a correspondence, are removed from the topology.

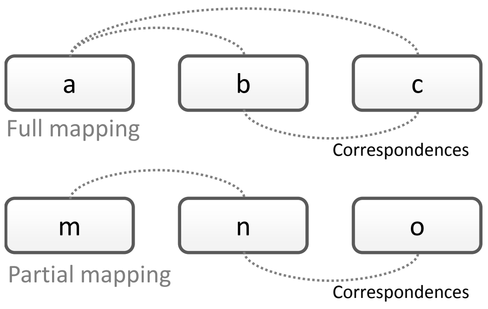

Figure 2: Two example sets of possible Correspondences between three nodes.

If a node has multiple correspondences, as depicted in the two topologies in Figure 2, additional considerations are required: Figure 2 shows at the top a set of three nodes forming a *full mapping*, i. e. each node has an correspondence with each other node. In this case, all nodes of this set can be merged into one single node, because each node has been matched with each other node. If there is only some kind of *partial mapping* for a set of three nodes, as exemplary shown at the bottom of Figure 2, only a part of the nodes can be merged. This is because nodes *m* and *o* are, for some type-specific reason found during matching, not capable of being merged, therefore, we infer that *o* also cannot be merged with a merged node $m+n$ and vice versa *m* cannot be merged with a merged node $n+o$. In this example it is only possible to merge two of the nodes. The generic rule is defined as follows: Let *M* be the set of nodes which has been merged into node *m*, if any, and *N* be the set of nodes merged into node *n*. Then, nodes *m* and *n* can be merged if and only if $\forall x \in N\ \forall y \in M : x \circ y$. This is the same for edges and other grouping concepts supported by the respective topology language.

Motivating Scenario. After the correspondences were evaluated, the applications get merged as depicted by (4) in Figure 1. The result is that both operating systems are merged into one now, as well as both virtual machines. This decreases cost as, for example, only one virtual machine and operating system license must be paid for hosting both Web sites.

3.2.5 Final Evaluation & Deployment

Our approach works on the model level which implies that the merge is not done on the running application. Hence, the merged application topology can be evaluated and adjusted by responsible architects and developers, which already evaluated the correspondences in step three. For instance, this can be done by deploying the merged topology into a test environment. If the merged application has been evaluated and passed the tests, it can be rolled out.

Motivating Scenario. After merging both applications additional information for the deployment have to be added. In our motivating scenario in Figure 1, (5) denotes that the Amazon credentials are provided and, afterwards, the application is automatically deployed.

4 TOSCA MERGE IMPLEMENTATION

In this section we present our Java-based implementation "TOSCAmerge" to realize the approach presented in Section 3. The prototype uses TOSCA to describe application topologies. After an introduction into the relevant concepts of TOSCA, we present concrete realizations for the aspects of the approach depending on the language used to describe application topologies.

4.1 TOSCA

TOSCA, the *Topology and Orchestration Specification for Cloud Applications* (OASIS, 2012), is a specification currently standardized at OASIS, which can be used to describe application topologies. We call them TOSCA-based application topologies. We use TOSCA to define application topologies in our prototypical implementation[2]. A TOSCA-based application topology is defined by *Node Templates*, which represent the components of an application such as a virtual machine or Java application, and binary *Relationship Templates*, which define relations between the components represented by Node Templates, e. g. a *Apache Tomcat* Node Template may be hosted on an *Operating System* Node Template. These templates are generic and typed by reusable Node Types and Relationship Types, respectively. Types define the semantics of templates, for example, there might be a Node Type *Apache Tomcat* created by the Tomcat developer community, which is then used in different application topologies to type Node Templates. Figure 3 presents

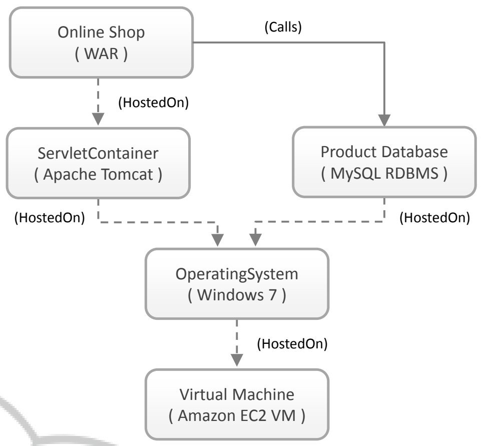

Figure 3: Example TOSCA application topology, visualized with Vino4TOSCA (Breitenbücher et al., 2012).

an example TOSCA-based application topology. The rounded rectangles denote Node Templates and the arrows Relationship Templates. The type of each template is rendered in parentheses.

TOSCA does not have a predefined set of types, but enables individual definition of types. Through derivation, types can refine other types, for example, the Node Type *Apache Tomcat* may be a refinement of the Node Type *Servlet Container*. Besides other information, which is not relevant for describing our approach, types in TOSCA define the properties of their templates, for example, a virtual machine may have the property IP-address and a hardware specification. TOSCA provides a recursive grouping mechanism for subgraphs of an application topology and is, for example, able to define policies on all the contained nodes, relationships, and groups. Describing application topologies is only one aspect of TOSCA. Based on the topology of the cloud application, TOSCA's goal is to provide portability of the application's management and operation, as well as increasing automation (Binz et al., 2012a). Due to the portability offered by TOSCA, the application may even run on different TOSCA-compliant runtimes in different environments, which supports our migration scenario.

4.2 Type-specific Logic

One requirement towards our approach (c.f. 3.1) is that the number of types and their semantics must not be restricted. We opted for a plugin-based architecture to support arbitrary types: In case a type should be supported, a respective plugin has to be written.

The type system of TOSCA is capable of single inheritance of Node Types and Relationship Types through stating, for example, that type *Apache Tomcat* is derived from type *Servlet Container*. To use this ca-

[2]The implementation uses the specification and XML schema of TOSCA v1.0 working draft 05 (OASIS, 2012)

pability, plugins must be able to match and merge two nodes or relationships of different types, for example *Apache Tomcat* and *Servlet Container*. This is addressed by associating plugins in TOSCAmerge with two type ids (namespace + id). If the type ids are different the plugin states to be able to evaluate matches and, if applicable, merge a node of type *Apache Tomcat* with a node of type *Servlet Container*.

In Section 3.2 we explained that nodes and relationships are processed pairwise. If no plugin is available for a pair of type ids, then neither a match nor a merge of the respective nodes and relationships is possible. In other words, to establish a correspondence the respective plugin must be available to take the decisions requiring type-specific logic.

4.3 Handling of Group Templates

Group Templates are a TOSCA-specific grouping mechanism for subgraphs of TOSCA-based application topologies. They may contain nodes, relationships, and, recursively, other groups. A node or relationship contained in one Group Template, or recursively in multiple groups, must evaluate the policies (non-functional requirements) of all the groups it is contained in. This is done by calculating the *accumulated policy*, which summarizes the policies of the node or relationship and all of the groups it is contained in. The accumulated policy is processed with the same type-specific plugins that are used during matching, i. e. the number of constraints possibly preventing the creation of a correspondence possibly increases.

4.4 Handling Application Logic

Nodes representing application logic are not matched or merged, as discussed in Section 3.2. Due to the open world assumption discussed in Section 3.1, the implementation cannot know if a particular node represents application logic or not. Therefore, the implementation maintains a list of type ids to be excluded from matching and merging. New Node Types representing application logic must be added by the users of the implementation.

4.5 Architecture & Design

The architecture of TOSCAmerge is structured into four layers, as depicted in Figure 4: (i) The user-facing API, offering the merge operation to external clients through the *Merge Service Interface* to enable the integration into other tools. The *Manual Evaluation Interface* returns the identified correspondences after the matching step for manual evaluation and applies

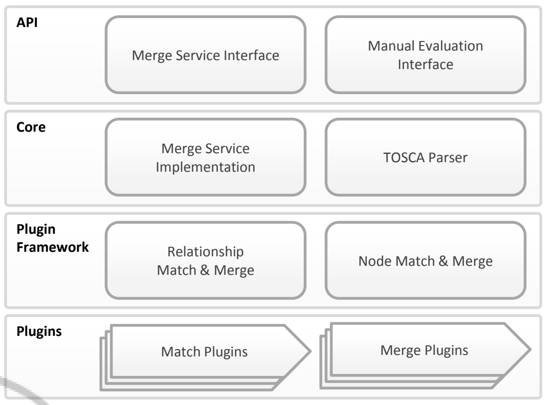

Figure 4: Architecture of TOSCAmerge framework.

the changes to the correspondences. (ii) The core implements the logic for functionality-preserving merge described in the approach, presented in Section 3. It also includes the components to read, write, and parse the involved input and result TOSCA-based application topologies. (iii) The plugin framework selects, manages, and invokes the type-specific logic, as requested by the core. (iv) The plugin layer represents the actual match and merge plugins which implement the type-specific logic. This ensures the logic for new types can be easily added to the TOSCAmerge framework and ensures extensibility.

5 DISCUSSION

In this section we discuss our generic approach and its realization based on TOSCA.

The proposed approach ensures that only valid merges of two nodes or two edges are executed. This comes with the precondition that the type-specific plugins used for merging and matching are correct. The application logic is preserved because the nodes representing business logic are ignored by the matching step and thus are never merged. Our approach focuses on the supporting infrastructure containing mainly standardized components to operate the application. Standardized does not mean that components cannot be tailored to customer's needs (Mietzner et al., 2010).

Correctness and functionality preservation can only be facilitated if the topology models include all the components, properties, and relations required to operate the application. Due to the fact that our approach processes the application topology only and not the executables or other information sources, lack of information in the application topology leads to incorrect and unpredictable results.

In addition, application topologies have to be decomposed into components. For example, modeling the operating system, middleware components, and the application as one single node represents a topology which cannot be processed by our approach. This fine-granular modeling is also a best practice to facilitate automated management using TOSCA. This results in application topologies with similar granularity, which is important to be able to process nodes and edges pairwise.

Our approach is able to merge two input application topologies into one result application topology. However, if more than two application topologies should be merged, the result of the first merge must be merged with a third application topology and so on. This is called *binary approach* by (Leser and Naumann, 2006) and comes with the challenge in which order to merge the application topologies. This is out of scope of this paper.

For the usage in enterprise IT projects, our approach increases automation in IT management by reducing the effort of the respective domain experts to a minimum. The responsible domain experts of the infrastructure and middleware components put their knowledge into the plugins, processing nodes and edges of the respective types. This enables reuse of domain knowledge, even shared between different enterprises, and therefore improves the return on investment for the effort spent to define and optimize the type-specific plugins. On the other hand, the effort of the architects and developers is reduced by removing the time-consuming task of identifying (matching step) and performing the merge manually. Instead, architects and developers only review and verify the correspondences identified automatically by our approach before they are processed. This is in line with the TOSCA philosophy of domain experts defining and implementing Node Types and Relationship Types which are composed to model applications (Binz et al., 2012a).

For validation, we tested our approach with more than 20 pairs of different test scenarios, covering the most important concepts provided by TOSCA-based application topologies. The TOSCA-based realization is implemented as greedy algorithm, i. e. it generates correct results, which are, however, not necessarily the global optimum. The computational complexity of the overall implementation is, also due to the implementation as greedy algorithm, quadratic with respect to the number of nodes. The algorithms and a detailed discussion of their computational complexity can be found in (Weiß, 2012).

6 RELATED WORK

This section reviews approaches for merging in other areas, like schemas, processes, and graphs in general, as well as languages to describe application topologies.

For business processes there is a large number of approaches to merge them. An algorithm to merge two *Event-Driven Process Chains* (EPC) into one, while preserving the behavior of the input processes, is presented in (Gottschalk et al., 2008). This is done by transforming the EPCs into *Function Graphs*, describing the sequence of functions, merge these graphs, and transform them into EPCs again. A similar approach is presented in (La Rosa et al., 2010), where EPCs are transformed into an *Annotated Business Process Graph*, which is also able to depict connectors and events. An approach to compose a business process from process fragments—non-complete process knowledge—is discussed in (Eberle et al., 2010). The authors define a number of basic operations to facilitate the *knitting* of fragments.

A high-level methodology to merge graphs is presented in (La Rosa et al., 2010), which first calculates a mapping between the two graphs, then merges the mapped entities, and finally foresees an optional post-processing. Our method (Section 3.2) is structured similarly. There are a number of other approaches following this high-level methodology and using correspondences to describe the result of their mapping step. A generic approach in (Pottinger and Bernstein, 2003) assumes that the correspondences are given and their merge operator merges two models to return a *duplicate-free union*. Reintegrating adapted business processes into the original version of the business process, without having an (ordered) list of the change operations done, is addressed by (Küster et al., 2008). They also foresee the manual intervention of domain experts during the process of merging.

These approaches are focused on business processes and the specifics in this area. One difference is that the number of types in business processes is usually restricted and known, which is in contrast to our open world assumption (see Section 3.1).

A general method for matching of graphs based on labels is the *edit distance* (Wagner and Fischer, 1974) or *graph edit distance* (Sanfeliu and Fu, 1983), which calculates the similarity of two labels or two graphs, respectively. This is done by counting the number of operations (remove, add, and so on) required to edit the label/graph from one to the other. For our approach the edit distance is not used, because we merge using the exact type ids (namespace + id in TOSCA), not the labels, i. e. names, of the nodes.

To the best of our knowledge we do not know any works which are concerned with the merging of decomposed application topologies. This might be due to the fact that the concepts and way TOSCA depicts topologies is relatively new to IT service management.

7 CONCLUSIONS

In this paper we proposed an approach for functionality-preserving merging of application topologies. We first presented a motivating scenario, based on this we extracted the requirements of such a solution, and then defined a suitable methodology. Additionally, we implemented our approach using TOSCA to describe the application topologies. The discussion showed that our approach is well applicable, but requires the availability of a detailed application topology and the respective type-specific plugins.

In the future we want to research how these topology descriptions may be extracted from existing application deployments by integrating this research with our work on *Enterprise Topology Graphs* (Binz et al., 2012b). TOSCA, used in the realization of the presented approach, is not limited to the description of topologies, but in particular enables the portable management of applications. Therefore, in the future we want to look into the adaptations required in the management plans, to adapt to the changes in the topology.

ACKNOWLEDGEMENTS

This work was partially funded by the BMWi project CloudCycle (01MD11023).

REFERENCES

Andrzejak, A., Arlitt, M., and Rolia, J. (2002). Bounding the resource savings of utility computing models. *Internet Systems and Storage Laboratory, Hewlett Packard Laboratories, Palo Alto*.

Binz, T., Breiter, G., Leymann, F., and Spatzier, T. (2012a). Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(03):80–85.

Binz, T., Fehling, C., Leymann, F., Nowak, A., and Schumm, D. (2012b). Formalizing the Cloud through Enterprise Topology Graphs. In *Proceedings of 2012 IEEE International Conference on Cloud Computing*.

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Schumm, D. (2012). Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *Proceedings of the 20th International Conference on Cooperative Information Systems (CoopIS 2012)*, Lecture Notes in Computer Science. Springer.

Chong, F. and Carraro, G. (2006). Architecture strategies for catching the long tail.

Eberle, H., Leymann, F., Schleicher, D., Schumm, D., and Unger, T. (2010). Process Fragment Composition Operations. In *Proceedings of APSCC 2010*, pages 1–7. IEEE Xplore.

Gottschalk, F., Aalst, W. M., and Jansen-Vullers, M. H. (2008). Merging event-driven process chains. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008*, OTM '08, pages 418–426. Springer-Verlag.

Guo, C., Sun, W., Huang, Y., Wang, Z., and Gao, B. (2007). A framework for native multi-tenancy application development and management.

Küster, J. M., Gerth, C., Förster, A., and Engels, G. (2008). Detecting and resolving process model differences in the absence of a change log. In *Proceedings of the 6th International Conference on Business Process Management*, BPM '08, pages 244–260. Springer-Verlag.

La Rosa, M., Dumas, M., Uba, R., and Dijkman, R. (2010). Merging business process models. In *Proceedings of the 2010 On the move to meaningful internet systems*, OTM'10, pages 96–113. Springer-Verlag.

Leser, U. and Naumann, F. (2006). *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. Dpunkt.Verlag GmbH.

Mietzner, R., Leymann, F., and Unger, T. (2010). Horizontal and Vertical Combination of Multi-Tenancy Patterns in Service-Oriented Applications. *13th International IEEE EDOC Enterprise Computing Conference (EDOC 2009)*, 4(3):1–18.

OASIS (2012). *Topology and Orchestration Specification for Cloud Applications Version 1.0 Working Draft 05*.

Pottinger, R. A. and Bernstein, P. A. (2003). Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 862–873. VLDB Endowment.

Sanfeliu, A. and Fu, K.-S. (1983). A distance measure between attributed relational graphs for pattern recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-13(3):353 –362.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *J. ACM*, 21(1):168–173.

Weiß, A. (2012). *Merging of TOSCA Cloud Topology Templates*. Master thesis, University of Stuttgart.