RC23382 (W0409-052) September 8, 2004
Other

IBM Research Report

Model-Based Automation of Service Deployment in a Constrained Environment

Tamar Eilam, Michael Kalantar, Alexander Konstantinou, Giovanni Pacifici
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Model-Based Automation of Service Deployment in a Constrained Environment

Tamar Eilam, Michael Kalantar, Alexander Konstantinou, Giovanni Pacifici
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
{eilamt, kalantar, avk, giovanni}@us.ibm.com

Abstract: Service physical topology design in large data centers is a function of many cross-cutting concerns such as service connectivity, performance, and security requirements, as well as data center resources, policies and best practices. These interdependencies represent a significant source of complexity, cost, and risk in data center management. This paper proposes a novel model-based approach to service physical topology design that reduces design complexity and allows for a separation of concerns. Service requirements are represented in terms of high-level models, and provisioning concerns as model transformation rules, resource selectors, constraints and objective functions. We use a refinement engine that automatically searches the space of service model transformations to produce a set of possible service physical deployment topologies. We discuss our prototype implementation of the service model refinement engine design and we report the results of experiments where we used our prototype to generate detailed physical deployment topology solutions for a variety of service models describing a rich set of requirements.

Keywords: data center automation, service deployment, autonomic configuration, refinement, MDA

Introduction

The design and configuration of data center physical service topologies is a highly complex process that poses significant challenges to data center operators. It requires dealing with different operational service requirements, deployment service requirements, infrastructure resource availability, as well as data center policies and best practices[18]. Expertise for these concerns is often divided across professional and organizational domains. For example, service connectivity, configuration, and performance requirements are correspondingly defined by system architects, system testers, and business users. Furthermore, the same service may be deployed in different environments, such as development, staging, testing and production, with differing sets of requirements for each environment. Data center operators must satisfy such externally defined service requirements using available resources, without violating the requirements of previously deployed services, while optimizing shared infrastructure revenue.

This paper proposes a model-based approach to automating service deployment in large and complex data centers. In our approach, service experts specify a logical service topology which is a high-level description of service component, connectivity, and configuration requirements. Concern experts express transformations and constraints over the service meta-models to codify data center policies, best practices and deployment rules. Data center operators interact with a model transformation automation system to generate a physical deployment topology from the logical service topology using an appropriate set of transformations and constraints. In this manner, the design process is viewed as a process of applying transformations to a logical service topology model to produce a physical service topology model.

We provide a refinement engine to produce a specific physical deployment topology from these high-level user inputs. We represent the logical service topology and requirements as object-relationship models, where objects represent resources, relationships represent connectivity requirements, and attributes express constraints on resources and connections. We express logical to physical topology mappings in terms of local, concern-specific, model transformations guarded by constraints. For example, a network concern transformation may replace the logical connectivity of two model nodes with relationships modeling links to a physical Ethernet switch. A service concern transformation may similarly replace a logical J2EE application server with a physical software resource, such as a WebSphere application server. Concern experts define constraints which can be used to check the validity of physical service models produced.

The result of the transformation process is dependent on the order and type of the transformations applied. In this manner, automation can be introduced as a search process over the order and type of model transformations. The search process may be further constrained by resource availability to assure that the design process produces physical topologies that can be configured in an existing data-center. This paper presents a prototype implementation of the proposed approach, and discusses its application to design of service network topologies.

The use of object-relationship models for the design and configuration of systems[27] and networks[26] has been widely adopted in industry[7]. Automating the mapping of Platform Independent Models (PIM) to Platform Specific Models (PSM) has been the basis of the Model Driven Architecture (MDA)[29]. MDA was initially developed as an abstraction layer over middleware technologies such as CORBA, J2EE, .NET, and Web Services to support automatic code generation. The advantages of an MDA approach to the separation of concerns in distributed component-based architectures were presented in [17]. Previous research has examined the application of MDA to legacy system code reverse engineering[22], manual mapping of healthcare message protocol specifications to J2EE components[23], composition of components and aspects[10], generation of BPEL workflows from ADF process diagrams using a process graph formalism[15], configuration of distributed agent communities[12], understanding and predicting the performance of component-based enterprise applications[20], as well as binding SLAs to resources[6]. Previous results have focused on the specification of appropriate domain-specific PIMs. Mapping of PIMs to PSMs has been performed manually, or through application-specific mechanisms which are often underspecified. This paper introduces a general purpose graph transformation framework where automation is generalized as a search problem. In our framework, application-specific refinement knowledge is expressed in the form of search heuristics. Previous results in application-specific models can be leveraged when expressed in our framework as meta-models, transformations, and constraints. The focus of this paper is on physical service deployment topology design, rather than runtime process management[15]. The paper further presents a novel application of MDA to the design of services with network requirements, such as firewalls, that is constrained by available infrastructure resources.

Network service design and configuration has been the subject of a significant body of work[28][13]. Alternative approaches to MDA which have been proposed can be broadly categorized as toolkit, workflow, ECA, spreadsheet, and constraint based solutions. Provisioning toolkits[21] and end-to-end automation systems[2][19][32] integrate a limited set of tools to provision a single class of environments. Frameworks to manage sequences of automation steps such as e-Utility[8] and TPM[5] provide mechanisms to specify workflows that integrate any set of tools to provision different environments. The tools are reusable between environments but the workflows are environment specific. More recently, TPM[30] introduced a new feature called Application Topologies which adopted aspects of an MDA-based approach. Application Topologies automatically refines logical application structures using application-specific transformations over available software and network resources. Our approach supports extensible refinement with improved concern separation, demonstrated over network topology refinement. ECA[9][33], spreadsheet-based[16] and planning[14] solutions have typically been applied to dynamic configuration, and have not been used in design automation. Constrained-based solutions have been applied to prediction[25], root-cause analysis [24], and configuration[11] [3]. Existing algorithms for Constraint Satisfaction are not well suited to the exploration of non-equivalent solutions in the design space. The proposed solution incorporates constraints as guards on the transformation search space.

The proposed architecture offers significant advantages over existing solutions. Automatic weaving of concerns reduces the amortized complexity of provisioning, through reuse of provisioning transformations and constraints in the design of different services, over varying infrastructures. The process permits concern experts to define provisioning transformations and constraints in their area of expertise, shifting complexity away from the end service deployer, resulting in reduced staffing requirements. Model-refinement replaces the process of writing and adapting workflows, which is currently manual, intensive and error-prone with an automated process that is guarded by higher-level operational and policy constraints. Automatic constraint verification prevents operational, performance, and security failures during service maintenance. Utilization of infrastructure resources will be significantly improved by the automated refinement process. Application Service Providers may employ sophisticated objective function analyzers to select revenue optimizing topologies out of the several alternative physical service topologies which satisfy all constraints.

The rest of this paper is organized as follows: Section 1 presents the proposed service design automation system. Section 2 describes the SPiCE prototype which implements our automated model refinement process to determine valid service deployment topologies. Section 3 presents an application of the prototype to service provisioning focusing on network configuration. Finally, Section 4 concludes this paper and outlines future research directions.

1 Service Deployment Automation

When deploying multi-tiered web applications data center operators follow a two phase process which we call the *design phase* and the *configuration phase*. In the *design phase* operators map the logical service requirements into a physical service topology. In the *configuration phase* operators implement the physical service topology by selecting and configuring a set of specific data center infrastructure resources. For example, a Web Service[4] may be logically represented by a HTTP server tier, backed by a J2EE application tier and processing data from an SQL relational tier. Logically, each tier will be associated with service-specific content such as HTML and JSP pages, EJB and Servlet classes, as well as SQL tables and log sizes. In the design phase, these logical service tiers will be mapped into physical resources, such as Apache HTTPd 2.0 connected to a IBM WebSphere 5.0 cluster and IBM DB2 8.1, installed on Intel x86 servers running RedHat Linux AS2.1. The logical to physical mapping must maintain service requirements, such as Sun J2EE v1.3 support, as well as resource requirements, such as product and version compatibility matrixes. The service may also be associated with security and reliability requirements. In such cases, the design phase will need to introduce firewalls, redundant network connections, as well as replication to the physical service topology. Given a physical topology design of a logical service, the configuration phase involves the procurement, installation and configuration of the required resources.

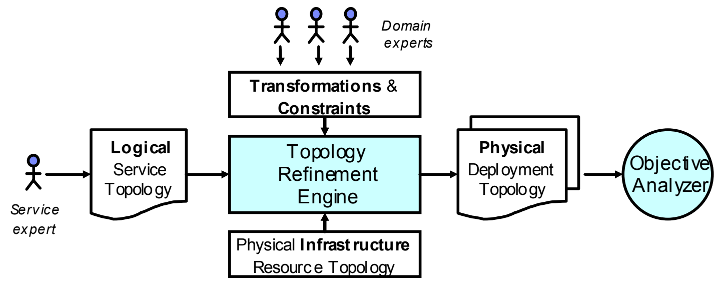

Domain experts
Transformations & Constraints
Logical Service Topology
Topology Refinement Engine
Physical Deployment Topology
Objective Analyzer
Service expert
Physical Infrastructure Resource Topology

Figure 1: Service Deployment Automation Process

This section will present the components, inputs and outputs of our service design automation process, which is depicted in Figure 1. The main component of this process is a *Topology Refinement Engine*. The refinement engine performs the automated MDA mapping from a *logical service topology* to a *physical deployment topology*. The service topology is a high-level description of the service components, connectivity, and configuration requirements, defined by a service expert. The physical topology is a description of how the service can be implemented in a given data center infrastructure. The general-purpose engine transforms the input model by accessing a repository of model *transformations* and *constraints* defined by service concern experts. The refinement engine searches through the space of transformations to generate physical topologies that can be implemented with a set of available data-center *physical infrastructure resources*. The physical topologies generated must satisfy the constraints defined by concern experts, in addition to the logical service requirements. Alternative validated physical topologies may then be compared using an objective analyzer which can evaluate cost, relative performance, and configuration time. The remainder of this section will present details of each of the components, inputs and outputs of this process.

1.1 Logical Service Topology

A logical service topology, or *service topology* for short, represents a high level specification for a service. While a service topology must contain sufficient information for a solution to be architected, it will typically be high-level and will not fully specify all of the details of an underlying physical implementation.

This service abstraction will permit alternative physical implementations. The "best" implementation will depend on the resources that are available and the policies of a given service provider. We represent a service topology as a object-relationship model which is an instance of a *service meta-model*. The nodes represent logical components and the edges define required relationships between the components. Node and edge attributes define constraints on the resources and their relationships.

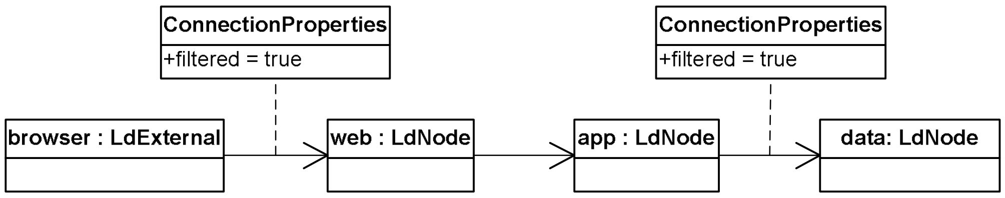


Figure 2: Sample Service Topology

Figure 2 shows an example of a service topology. This topology specifies a typical three tier Web Service with three logical nodes labeled `web`, `app` and `data` (of type `LdNode`). Each node represents a set of servers executing a defined set of applications. For the purposes of this example, we do not show application details. In addition to the logical nodes comprising the service, we show an additional node labeled `browser` of type `LdExternal` (a subtype of `LdNode`) which represents an external entity. We use the type `LdExternal` to account for the connectivity specifications of requirements on unmanaged external sources. In this example, external resources can only connect to the web node. Further, the attribute `filtered` on a communication path indicates that it is a requirement to restrict communication.

1.2 Physical Infrastructure Resource Topology

The refinement engine constrains the physical topologies generations to ones that can be implemented using available data center infrastructure resources. The physical infrastructure resource topology is a detailed description of the current configuration of the hosting infrastructure. To simplify the resource selection process, infrastructure resources are themselves represented using a model. Furthermore, the infrastructure meta-model is the same as the physical topology meta-model. This approach relieves the refinement engine from having to perform discovery of heterogeneous configuration repositories and support configuration schema conversion. In the *infrastructure topology*, nodes represent actual physical resources in the hosting infrastructure such as servers. They may also represent logical resources that result from the configuration of another resource such as virtual networks (VLANs). Edges represent the operational and configuration dependencies between resource instances. Node and relationship attribute values describe their current configuration. In addition to the configuration of the infrastructure resources, additional information such as the current use or ownership of the resources is maintained so that it is possible to identify available resources.

1.3 Physical Deployment Topology

The output of the service design automation process is a physical deployment topology, or *physical topology* for short. A physical topology expresses how infrastructure resources should be configured to implement an input service topology. It identifies specific resources and their required configuration. The physical topology can be used as input to a configuration automation tool that will determine the type and order of configuration operations. It is beyond the scope of this paper to discuss how such configuration is determined and performed. Like the service and infrastructure topologies, a physical topology can be described with an object-relationship model. In this case, the objects, relationships and attribute values represent desired, or goal, values.

Figure 3 shows three simplified physical topologies for the service topology of Figure 2. For readability, representative icons are used instead of generic UML objects. In all of the examples, the logical nodes are implemented using a single server. In the first example, each of the servers that make up the service is connected, via an isolated VLAN to a common multi-homed firewall. In the second example, two dual-homed firewalls are used. Furthermore, some servers have multiple interfaces in this solution. In the final topology, a single firewall is used to implement both of the filtered connections. However, the non filtered connections are implemented through common subnet instead of through the firewall as in example (a). It should be noted that these are not the only possible configurations.

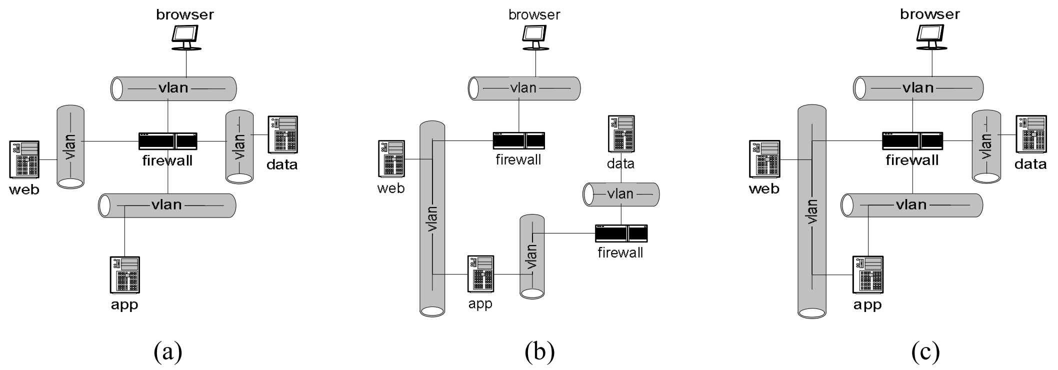


Figure 3: Sample physical deployment topologies for the same service

1.4 Topology Transformations

To create a physical deployment topology, the refinement engine applies *transformations* to the service topology. Concern experts will design these transformations to transform the logical elements in the service topology into elements that can be directly mapped to physical elements in the infrastructure. For example, logical connectivity between two logical nodes may be realized via a common VLAN, a router or a firewall; a single transformation would be created for each approach. A data center may choose to support all or any combination of these methods. Note that a transformation may only create intermediate results which need to be further transformed. For example, if a (logical) router is used to implement a connection, there may be additional transformations that define how the router is actually implemented: options might include a hardware router, a multi-layer switch, or a server running a daemon. Each transformation acts locally, hence a sequence of transformations may have to be applied to have the desired effect.

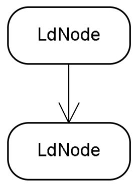
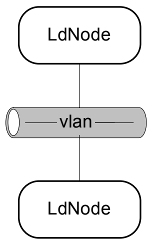
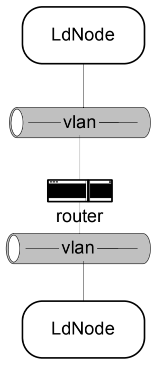
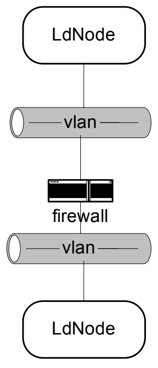
Pattern	VLAN insertion	Router insertion	Firewall insertion
			
Guards on match: `c=Connection`	**`not`** `c.filtered`	**`not`** `c.filtered` **`and`** **`not`** `c.sameSubnet`	**`not`** `c.sameSubnet`

Figure 4: Pattern with associated transformation examples

Each transformation acts locally. To identify locations in a topology where a transformation can be applied, each transformation is associated with a *pattern*. A pattern defines a sub-topology on which a transformation might act. A sample pattern is shown in the left panel of Figure 4. This pattern identifies locations in which two logical nodes communicate. Application to the sample service topology of Figure 2 would result in three matches: (`browser`$\rightarrow$`web`), (`web`$\rightarrow$`app`) and (`app`$\rightarrow$`data`). Each transformation is expressed in terms of the changes that take place to the sub-topology matched by the pattern. Figure 4 shows three example transformations that realize logical connectivity. Each uses the same left-hand-side pattern. In the VLAN insertion transformation, a VLAN is inserted between the logical nodes. In the router insertion transformation, two VLANs and a router are inserted. Finally, in the firewall insertion transformation, the logical connectivity is replaced by two VLANs and a firewall. Collectively, these transformations are referred to as *connective replacement transformations*.

Just because a sub-topology matches the pattern of a transformation does not mean that the transformation is valid. For example, if the logical connection between two nodes has an attribute `filtered` set to true, the VLAN insertion transformation is not a valid physical manifestation of the service requirements. To address this, we introduce *guards* on transformations. Guards are Boolean functions over a pattern match that must evaluate to true to execute the transformation. Example guards for the transformations in Figure 4 are shown below each transformation. For example, the guard on the VLAN insertion transformation prevents the transformation from being applied when the connection is filtered.

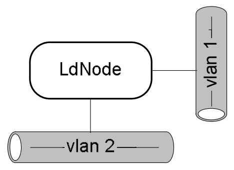
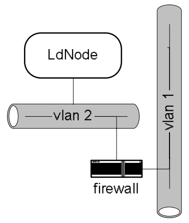
Pattern	Interface consolidation	Guard
		**`not`** `Vlan1.canReroute`

Figure 5: Interface Consolidation Transformation

A second transformation example is the *interface consolidation transformation*, shown in Figure 5. This transformation combines two communication paths onto a single interface. The pattern for this transformation identifies a logical node containing multiple interfaces by finding relationships to multiple `VLAN`s. After applying the transformation, traffic from entities on `Vlan 1` now reaches the logical node via `Vlan 2`. As a part of this transformation, a firewall is inserted between the Vlans. The firewall prevents traffic from `Vlan 2` back to entities connected to `Vlan 1.` Prior to the consolidation, such traffic was prevented by the node itself, assuming that the multi-homed node is not routing traffic.

Example Application of Topology Transformations

We show, by example, the reasoning by which the topology transformer selects and applies transformations to produce a physical topology. A more formal description can be found in Section 2.4. The refinement engine guides its selection of transformations by (a) which transformations are applicable (there are matches to its pattern and its guards are satisfied), (b) the available resources and their configuration as expressed in the physical infrastructure topology, and (c) global constraints and cost objectives.

As an example, consider the web service logical topology of Figure 2. By reviewing the set of available transformations, it is discovered that the following might be applied: firewall insertion to the matches (`browser`→`web`), (`app`→`data`), and (`web`→`app`); VLAN insertion to the match (`web`→`app`); and router insertion to match (`web`→`app`). Since there is no conflict, the firewall insertion transformation is applied twice to the matches (`browser`→`web`) and (`app`→`data`). The result of this dual application is the intermediate topology shown in Figure 6(a). There are three possible transformations that can be applied to realize the logical connection (`web`→`app`) as a physical connection. Suppose that the VLAN insertion transformation is selected. It may be selected randomly or might be selected because it appears to be the lowest cost solution. The result is shown in Figure 6(b). Now that all of the logical connections have been replaced, a possible topology has been identified if servers can be found to take the roles of each of the logical nodes. To this end, the infrastructure topology is searched. For the sake of argument, assume that the only available servers have only a single network interface. In this case, no server can be found to replace the logical `app` node. Consequently, a strategy to reduce the number of required interfaces is sought. The interface consolidation transformation achieves this. By reference to Figure 5, the transformation can be applied to the `app` node resulting in the topology of Figure 6(c). At this point, sufficient server resources, each with a single interface, can be found to implement this solution (results of the node replacement transformation are not shown). Note that the current solution is a non-optimal in terms of cost because it used 3 firewalls. If the firewalls have multiple interfaces, they could be consolidated using another transformation. If there was a cost objective or a policy that sought to minimize the number of firewalls, such a transformation would be used producing the configuration in Figure 3(c).

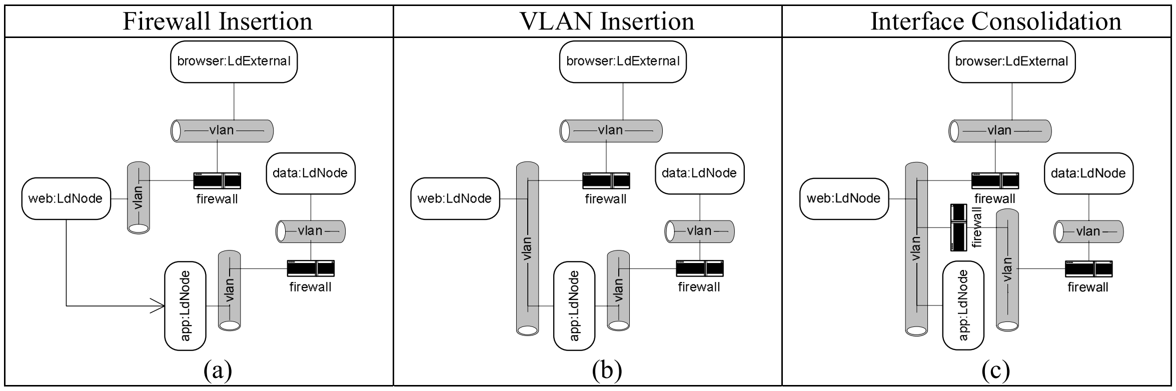


Figure 6: Example application of a sequence of transformations to a service deployment topology

2 SPiCE Prototype

To study the feasibility of the model-based approach to automating the design phase of service deployment we have built a prototype of the topology refinement engine. We call this prototype SPiCE for Service Plan Composition Engine (SPiCE). Our current prototype focuses on the network configuration aspects of service deployment. The refinement engine is a search engine which controls the transformation process. This engine receives three inputs: a service topology, an infrastructure topology, and a *transformation repository*, a collection of transformation rules and constraints. The search engine refines the service topology using the elements in the transformation repository. Its output is a set of physical topologies that represent possible network configurations with infrastructure resource bindings that can be used to deploy the service over the infrastructure.

The SPiCE prototype consists of five core components depicted in Figure 7. The search for valid physical topologies is controlled by the search engine layer. The search engine uses the transformer layer, to transform the input service topology into a physical topology. It uses the resource selector layer to identify resources in the infrastructure topology that will be used in the physical topology. Finally, the modeler layer provides a toolkit to create and manipulate typed object-relationship models representing the input, intermediary, and output topologies. Transformation rules and constraints are implemented by using functions provided by the modeler. A graphical user interface facilitates control and visualization of service deployment. The rest of this section discusses the SPiCE components and operations, bottom up.

2.1 Modeler

In SPiCE, object-relationship models are used to represent service, physical, and infrastructure topologies. SPiCE models are represented in-core as a collection of objects that can be linked by relationships. The modeler supports methods for instantiating, accessing, and updating in-core representation of an object-relationship model. All modeler operations are logged, permitting checkpointing and rollback of changes. Objects and relationship instances are indexed and can be queried. The modeler supports functions for importing and exporting model instances to and from XML documents.

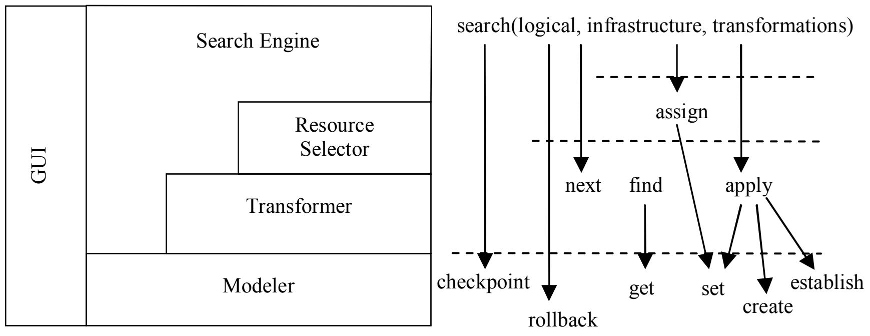


Figure 7: SPiCE components and their representative operations

In addition, the modeler supports a dynamic type system, allowing programmatic creation of namespaces containing object and relationship types, and meta-model views over namespaces. The current implementation supports multiple object type inheritance, but is limited to supporting binary relationship types only. The modeler type system is external to the Java type system in which it is implemented. Meta models are used to define the permissible types correspondingly to the input logical deployment topology, and the output physical deployment topologies models. Two meta-model roles are recognized by the system for this purpose: *input meta-model,* and *output meta-model.* They can be defined in XML files by the users of SPiCE and be given as input parameters for a new search.

2.2 Model Transformer

The Model Transformer layer includes an extensible set of transformation rules and constraints as well as some core functions to manage the set. Each transformation rule is composed of a pattern, a set of guards and a transformer. The pattern is used to produce a set of locations in a model, the guards are used to filter this set of locations, and the transformer describes how to replace the location with a different sub model structure. Interfaces are defined that encapsulate the roles of each one of these elements; each element is realized as a Java class that implements the corresponding interface.

A *pattern* is formally defined as a function that produces a set of bindings termed *matches*, for a given model. For example, a pattern may match any two objects which are instances of type `LdNode`, and are related with a `connects` relationship, as depicted on the left panel of Figure 4. Applying such a pattern on the service topology Figure 2 of would result in three matches: (`browser`→`web`), (`app`→`data`), and (`web`→`app`). Match nodes and relationships can be labeled, to facilitate use by transformers. A large class of patterns, termed *model based patterns,* can, themselves, be represented as models. The set of matches is a set of isomorphic sub-models (in the service topology). SPiCE supports model based patterns a pattern implementation that uses a generic graph matching algorithm[31] to find matches.

A *guard* is formally defined as a Boolean function on model locations in the form of conditions over the values of attributes of nodes and relationships in this location. Guards are used to filter a set of locations returned by a pattern match. They are needed in order to express the policy that governs applications of transformers. For example, a data center security policy may dictate that whenever there is no stated requirement to connect two application modules on the same broadcast domain, a firewall should be used. This policy can be expressed by associating suitable guards with the connection replacement transformations described above in Section 1.4.

A *transformer* is a function from a model and a match to a new model. That is, it transforms the sub-model defined by a location into a new sub-model thereby transforming the entire model. Every transformer is associated with a pattern that is used to identify all of the locations in the model on which the transformer can apply. A transformer may further be associated with a set of guards that filter the set of matches returned by the transformer's pattern. SPiCE applies a transformer on a model location only if (1) it is a match for the pattern associated with the transformer, and (2) all guards associated with the transformer are evaluated to `true` on the model location.

A *constraint* is formally defined as a Boolean function on models. Whilst guards are used to express a local condition, constraints are global and express policy and best practices requirements that cannot be associated with any single transformer. An example of a global constraint may be a condition on the number of resources of a certain type or their total cost. SPiCE uses the constraints to filter physical topologies that are reached in the course of the search.

A physical topology is a *valid output* of SPiCE iff (1) it can be reached from the input service topology by applying a sequence of transformers subject to guards and (2) all constraints are evaluated to `true` on this physical topology.

A *transformation repository* is a collection of transformation rules and constraints. SPiCE supports multiple transformer repositories; each different repository provides a means by which different implementation strategies and different policy constraints can be expressed. Transformer repositories are defined in XML by enumerating the set of transformation elements via reference to its implementing Java class, by defining associations between elements, such as an association between a transformer and a set of guards, and by defining model based patterns by including a definition of the associated model.

2.3 Resource Selector

The Resource Selector layer provides functions to map resources, represented as nodes and their corresponding relationships, in intermediary or physical topology models, to infrastructure resources, represented in an infrastructure topology. This mapping ensures that the output physical topology is a deployment solution that can be realized with the resources available in the infrastructure. For example, a physical topology requiring a dual-homed server cannot be realized in an infrastructure with only single-homed servers.

There are several challenges involved in resource selection. First, we observe that not all elements describing resources in a physical topology can be mapped to existing infrastructure resources. Virtual resources, such as VLANs way be created in the course of deploying a service, thus it is not necessary that they exist in the infrastructure prior to deployment. In such cases, however, it is necessary to ensure that the capacity to create such resources exists. There may be practical or policy based limits to how many of a particular type of resource can be created. The second issue concerns interdependencies between resources that must be taken into account in the selection process. For example, there is a dependency between a server and its set of link interfaces. An independent mapping of the set of servers and the set of link interfaces will not guarantee that servers with an appropriate number of link interfaces are available for the deployment; the `contains` relationship between the server and its set of link interfaces must be respected in the mapping. However, not all relationships must be respected by selection; some relationships in a physical topology represent a *desired configuration state* that can be created in the course of provisioning. For example the `contains` relationship between a VLAN and a switch port represents a desired configuration that should be created dynamically and programmatically by reconfiguring the switch. The set of 'hard' relationships that must be respected when selecting resources depends on the infrastructure, the available management tools, and on data center best practices and policy.

SPiCE addresses these challenges by using a designated set of *resource selection patterns* to identify the entities (objects and relationships) that must be mapped to entities of the infrastructure. As with transformation elements, resource selection patterns are pluggable Java objects; the available set can be modified/extended to allow using different patterns, respecting different sets of 'hard' relationships, depending on the infrastructure capabilities and the data center policy. The union of the set of matches of all resource selection patterns in a given model identifies the set of elements to be mapped. This set may form a non-trivial sub-model that needs to be matched as a whole. By default, the Resource Selector uses a generic graph matching algorithm[31] in order to perform the matching against the infrastructure model. While agnostic to the set of model elements and set of resource selection patterns, the generic graph matching algorithm suffers from an exponential execution time. A second approach, also taken in SPiCE, is to code knowledge of the resource selection patterns into the mapping logic. Such an approach, while superior in performance, requires reimplementation of the mapping logic when the resource selection pattern set changes.

2.4 Search Engine

The search engine drives the search to find one or more valid output physical topologies. The inputs of a search include a service topology, an infrastructure topology, a transformer repository and an output meta-model. The search is conducted by applying *model actions* on the *current model.* Initially the current model is the input service topology. Model actions are either *model transformation* or *resource selection*. Model transformations change the structure of the current model by applying one or more transformations and resource selection map elements of the current model to elements of the infrastructure model.

At every step of the search, a model action is applied on the current model. The search engine decides what model action to apply using a *heuristic*: a function that maps a model *M* to a sequence *Seq* of model actions to be applied. When a model action is applied, the result is a new current model *M'* for which a new sequence of model actions *Seq'* is obtained from the heuristic. If the sequence is exhausted, the search backtracks, returning to the previous model state *M,* the next model action in the sequence *Seq* is applied next.

SPiCE includes a default heuristic that is agnostic to the set of transformation rules used in a search. For every transformer in the input repository, all matches of its pattern are found for which all associated guards are satisfied. Each such pair of transformation and match defines a possible model action. If the current model is *not* a physical topology, a list of all possible transformation based model actions is returned. Otherwise, the list is preceded by a model action that attempts to select all of the resources, based on the resource selection patterns, as described in Section 2.3. Thus, the last model action taken is a global resource selection action. If resource selection fails, additional transformers are applied or, when there are none applicable, the search backtracks. Since all possibilities to apply a single transformation are attempted, it is guaranteed that a deployment solution will be found if one exists. However, the heuristic is wasteful in many cases. For example, if it is known that, for a given set of transformers, the same model is reached for any order of transformer application, then many execution paths can be pruned.

In addition to the default heuristic, we experimented with heuristics that optimized the search by utilizing domain knowledge. They use knowledge of the set of the implemented transformations to prune search paths that either are guaranteed to fail, or are known to be equivalent to other paths. The heuristics are based on the following observations. First, several of the transformations are designed to address problems caused by scarce resources. Application of such transformations is delayed until the specific problems have been identified. For example, the *interface consolidation* transformation reduces the number of interfaces a server requires. Its application is driven by a lack of servers with enough interfaces or by an objective that requires a certain maximum number of interfaces. As another example, the *router (firewall) consolidation* transformation reduces the number of required routers/firewalls; its application is delayed until determining that there aren't sufficient routers (firewalls). Second, some transformations are applicable only when others have already been executed. For example, *connection replacement* transformations must precede the other transformations. In such cases, a heuristic ordering of transformations prevents pattern matching and guard evaluation that will fail and avoids transformations that will lead the search to dead ends. Finally, the order of transformation application on a set of matches may not have any impact on the set of solutions. In such cases, the heuristic applies the transformation to all matches in an arbitrary order as an atomic model action and prevents its application in other orders.

Using the above principles, two domain specific heuristics were implemented. They evaluate each model and classify it into a number of distinct states. At each state, the set of transformations and their order is clearly defined. The heuristics differ only in the order in which they attempt to apply the connectivity replacement transformations: one favors firewalls while the other favors VLANs. While these heuristics did improve the performance of the search by orders of magnitude, a detailed study of their performance is a subject of future work.

2.5 Graphical Interface

A graphical user interface, shown in Figure 8, facilitates control and visualization of the inputs and outputs of the search process. Four tabs are used to show the three inputs and the set of output physical topologies. The left window shows the filtered web-service logical topology. The center window shows an infrastructure topology with the selected resources highlighted. The right window shows a system view of one of two physical topology solutions. The user interface supports multiple model views allowing users to hide certain types, collapse relationships, and view objects as iconified graph nodes. Using the *Next* and *Previous* buttons, users can iterate over the set of physical topology outputs.

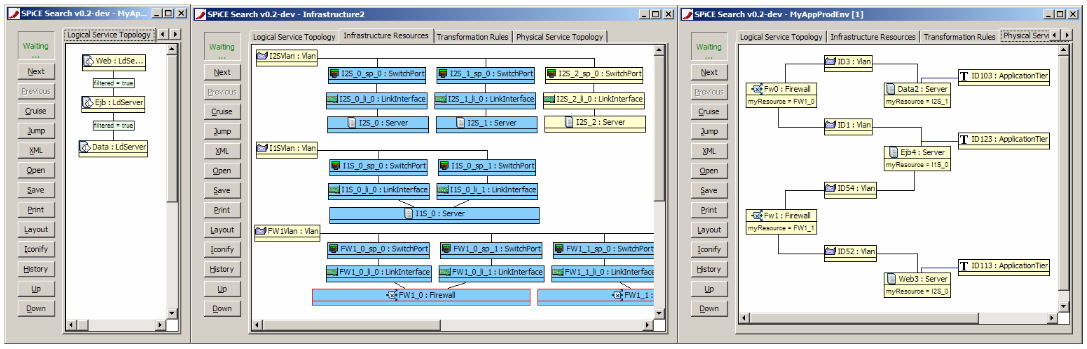


Figure 8: The SPiCE GUI showing logical, infrastructure, and physical topologies

3 Experiments

In order to experiment with the approach proposed in this paper, we chose to focus on one aspect of the deployment: network topology and configuration. First, we implemented a set of network transformation rules, some of which were described in Section 1.4. Second, we ran SPiCE on a set of examples varying two parameters: the service requirements, and infrastructure characteristics. The following network transformations were implemented: (1) ***Connection Replacement*** was described in Section 1.4 and Figure 4. Note that to simplify the presentation, load balancer, link interface and switch port elements are not show in the figure. (2) ***Interface Consolidation*** was covered in Section 1.4 and Figure 5. ***(3) Router &Firewall Consolidation*** combines two router (or two firewall) nodes in a model into a single router. To do this, all interfaces of the two routers (firewalls) are combined as interfaces on the same router (firewall). Consolidating routers (firewalls) decreases the number of routers (firewalls) needed but increases the required number of interfaces on each device. (4) ***Node Replacement*** replaces application modules (`LdNodes`) with one or more servers (and their interfaces) depending on whether or not the application module is implemented as a single server or as a cluster.

To vary the service requirements, we used two different logical deployment topologies as the set of inputs. The first, *Prod*, is the simplified service topology corresponding to a 3-tiered e-business service, shown in Figure 9(a). The second, *Dev,* has the same application structure as *Prod*. However, instead of requiring filtered connections, it requires connections to be on the same subnet; it explicitly tries to minimize the use of firewalls or routers. These two examples demonstrate a common situation in which a similar service needs to be deployed in different environments, such as development and production. The security requirements for the development environment are much more relaxed, hence firewalls are not necessary.

To vary the infrastructure characteristics, we used two different infrastructure topologies. One, *Infrastructure Topology A,* is the sample shown in Figure 8. The second one, *Infrastructure Topology B,* is similar except that it does not contain the 3-interface firewall and it contains only two single-homed servers, and one dual-homed server.

We ran the SPiCE prototype to generate network topologies for all four input combinations of service and infrastructure topologies. Sample output physical deployment topologies are illustrated in Figure 9, an "iconified" view, in which a server is represented by a , a VLAN by a , a firewall by a , and an application tier by a **T**. For simplicity of the presentation, some network elements such as switch ports and link interfaces are not shown. The same figure identifies the combinations that generate each of the outputs. Some combinations produce only one valid output while others generate several. Further, some outputs, such as (a) and (b) are generated by more than one combination while others are generated by only one service topology, infrastructure topology pair.

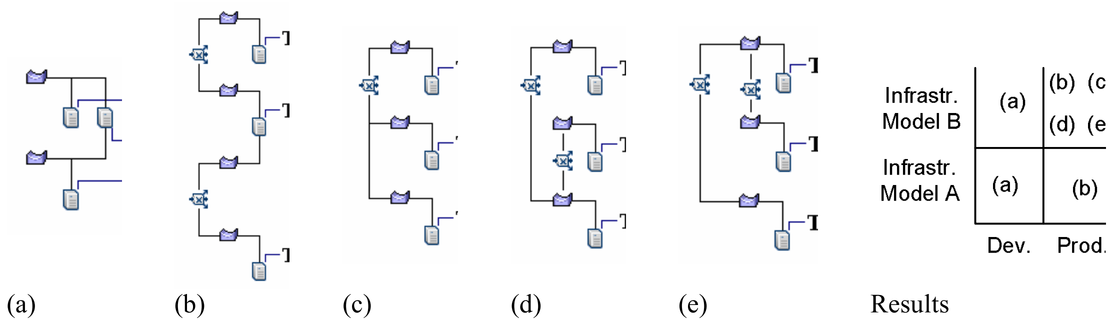


Figure 9: Unique outputs of the SPiCE refinement engine & their mappings to service/infrastructure topology pairs

SPiCE was successful in producing correct deployment topologies based on the implemented set of network transformations and constraints. Programming transformations proved to be a relatively easy task thanks to a well thought of object model and the modeler layer. Visualization, provided by the GUI of SPiCE, proved to be an important function not only to reason about the output set, but also to debug transformations in the implementation phase.

The experiments helped us identifying the following set of issues. First, many output physical topologies are actually equivalent; they differ only in the mapping to a set of identical resources. This situation is caused by reaching equivalent search states via different execution paths. For example, by applying an order-independent set of transformations in different orders. Methods need to be developed to either identify and filter equivalent results or to prevent their creation in the first place.

Another issue is the performance of the search. One source of poor performance is the generic graph matching algorithms that are used both for matching transformers' patterns, and for mapping physical topology elements to infrastructure resources. While these algorithms provide a general method that can be applied to any set of transformers, or any set of models and model elements, they suffer from an exponential execution time. The problem is less severe in the use of these algorithms to match transformers' patterns, as patterns and even service topologies are relatively small. However, using generic graph matching to map to resources of the infrastructure proved to be impractical. We were obliged to develop resource mapping logic based on resources semantics. One such method, for example, is to select servers based on the number of link interfaces they possess. This 'special case' code will need to be re-implemented whenever the set of model elements change. Another source of poor performance is the search algorithm that blindly applies transformations in different orders in search for solution topologies. A long search execution path can lead to a physical topology that is equivalent to previous a one, or that is filtered by a constraint. Transformer specific heuristics, discussed in Section 2.4 alleviate the problem but they will need to be re-implemented for a different transformer set.

4 Summary and Future Work

This paper proposes a model based approach to transform a high-level service specification into a physical deployment topology specifying how the service can be implemented on a given hosting infrastructure. The approach consists of representing deployment concerns such as service requirements, infrastructure resource availability, infrastructure capabilities, and policy using such elements as models, transformations, guards and constraints. An input service deployment topology is transformed by these transformations and by identifying available resources to produce a physical deployment topology which can then be used to drive actual deployment. The paper describes in detail SPiCE prototype topology refinement engine.

We used SPiCE to generate detailed physical deployment topology solutions for a variety of service models describing a rich set of requirements. To experiment with the proposed method, we implemented a set of *refinement elements* including model transformations, guards, and constraints focusing on network configuration. We used the SPiCE prototype, configured to work with these knowledge elements, to find deployment solutions for different combinations of service requirements sets and infrastructure characteristics, both represented as models. For each such combination, a different ,but sometimes overlapping, set of deployment topologies was correctly produced. The SPiCE GUI significantly facilitated reasoning about the produced deployment topologies as well as why and how they were produced.

Experimentation identified a number of performance and optimization issues that need additional attention. In addition to the approaches discussed above, incorporating the semantics of global constraints and objective functions into the search would help to optimize the search by providing new opportunities for pruning.

Transformations and constraints currently require programming to specify their behavior. It is an open question of how to best express these in a declarative way. This would make their expression simpler and easier. We did find it possible to express some patterns declaratively as models. However, the current definition of a model is not powerful enough to express patterns such as server with any number of interfaces. Furthermore, transformation behavior is harder to express. Other approaches such as [1] may suffice. Finally, expression of guards and constraints might be expressed in a formal expression language such as OCL. An open question is how expressive such a language must be.

Finally, the physical topologies generated by SPiCE focus on network configuration. Additional details must be added to the physical topologies to ensure that they contain sufficient information for the resources to actually be deployed. For example, firewall rules are not currently specified. In addition to augmenting the set of network transformations, more experience with transformations in other domains such as software installation and storage management is needed to validate the thesis that such transformations can be written independently of each other.

Acknowledgements

The authors would like to acknowledge John A. Pershing Jr., Lily Mummert, Guerney D. H. Hunt, Andrew Trossman, and Liana Fong who have contributed to this work with many comments and insights.

References

[1] Agrawal, A.: A Formal Graph Transformation Based Language for Model-To-Model Transformation. Ph.D. Dissertation, Vanderbilt University, 08/2004.

[2] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J.A. and Rochwerger, B.: Oceano – SLA Based Management of a Computing Utility. IM 2001.

[3] Balter, R., L. Bellissard, F. Boyer, M. Riveill, J.Y. Vion-Dury: Architecturing and Configuring Distributed Applications with Olan. Middleware 98, 09/1998.

[4] Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard: Web Services Architecture. W3C Working Group Note, http://www.w3.org/TR/ws-arch/, 11/2004.

[5] Credle, R., et al. IBM Web Infrastructure Orchestration. IBM Redbook ISBN 0738499374, 11/2003.

[6] Debusmann, M., R. Kroger, K. Geihs,: Unifying service level management using an MDA-based approach. NOMS 2004.

[7] Distributed Management Task Force (DMTF): Common Information Model (CIM) Specification, Version 2.2. 06/1999.

[8] Eilam, T., Appleby, K., Breh, J., Breiter, G., Daur, H., Fakhouri, S. A., Hunt, G. D. H., Lu, T., Miller, S. D., Mummert, L. B., Pershing, J. A., and Wagner, H.. Using a Utility Computing Framework to Develop Utility Systems. In IBM Systems Journal. 43(1):97-120, 2001.

[9] Fosså, H.: Interactive Configuration Management for Distribute Systems. Ph.D. Thesis. University of London. 05/1997.

[10] Fuentes, L., M. Pinto, A. Vallecillo: How MDA can help designing component and aspect-based applications. EDOC 2003.

[11] Goli, S. K., J. Haritsa, and N. Roussopoulos: ICON: A System for Implementing Constraints in Object-based Networks. IM 95.

[12] Gracanin, D.; Bohner, S.A.; Hinchey, M.: Towards a model-driven architecture for autonomic systems. IEEE ECBS. 05/2004.

[13] Hegerink H.G., S. Abeck: Integrated Network and System management. Addison Wesley 1995.

[14] Keller, A., J.L. Hellerstein, J.L. Wolf, K.-L. Wu, V. Krishnan: The CHAMPS system: change management with planning and scheduling. NOMS 2004.

[15] Koehler, J., R. Hauser, S. Kapoor, F.Y. Wu, S. Kumaran: A model-driven transformation method. EDOC 2003.
[16] Konstantinou, A. Towards Autonomic Networks. Ph.D. Thesis,.Columbia University. 10/2003.
[17] Marvie, R., P. Merle, J.-M. Geib: Separation of concerns in modeling distributed component-based architectures. EDOC 2002.
[18] Mehra, P.: Global deployment of data centers. IEEE Internet Computing, Vol. 6, Nr. 5, 09/2002.
[19] Moore, J., Irwin, D., Grit, L., Sprenkle, S., and Chase, J.: Managing Mixed-Use Clusters with Cluster-on-Demand. Technical Report, Duke University, Department of Computer Science, November 2002.
[20] Mos, A, J. Murphy: Performance management in component-oriented systems using a Model Driven Architecture approach. EDOC 2002.
[21] Opsware http://www.opsware.com
[22] Qiao, B. H. Yang, W.C. Chu, B. Xu: Bridging legacy systems to model driven architecture. COMPSAC 2003.
[23] RamIjak, D., J. Puksec, D. Huljenic, M. Koncar, D. Simic: Building enterprise information system using model driven architecture on Js2EE platform. IEEE ConTEL. 06/2003.
[24] Sabin, M., R. D. Russel, and E. C. Freuder: Generating Diagnostic Tools for Network Fault Management. IM 1997.
[25] Sabin, M., A. Bakman, E. C. Freuder, and R. D. Russel: Constraint-Based Approach to Fault Management for Groupware Services. IM 1999.
[26] Sengupta, S., A. Dupuy, J. Schwartz, and Y. Yemini: An Object-Oriented Model for Network Management. Object Oriented Databases with Applications to CASE. Prentice-Hall, 1991.
[27] Sloman, M. Management for open distributed processing. Distributed Computing Systems, 09/1990.
[28] Sloman, M.:Network and Distributed Systems Management. Addison Wesley. 1994.
[29] Soley, R., OMG Staff Strategy Group. Model Driven Architecture. OMG White Paper, 11/2000.
[30] Tivoli Provisioning Manager (TPM). IBM. http://www-306.ibm.com/software/tivoli/products/prov-mgr/
[31] Ullman, J.R.: An algorithm for subgraph isomorphism. Journal of the Association for Computing Machinery, 23(1):31-42, 1976.
[32] Werner Vogels and Dan Dumitriu. An Overview of the Galaxy Management Framework for Scalable Enterprise Cluster Computing. IEEE Cluster 2000, Chemnitz, Germany, 12/2000.
[33] Yemini, Y., A.V. Konstantinou, D. Florissi: NESTOR: An architecture for self-management and organization. JSAC Vol. 18, Nr. 5. 06/2000.