RC24916 (W0912-039) December 7, 2009
Computer Science

IBM Research Report

Distributed and Collaborative Design of Composite Service Deployments Using Graph Transformations

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

DISTRIBUTED AND COLLABORATIVE DESIGN OF COMPOSITE SERVICE DEPLOYMENTS USING GRAPH TRANSFORMATIONS

Tamar Eilam, Michael Kalantar, Alexander V. Konstantinou

IBM T.J. Watson Research Center, Hawthorne, NY, USA
{eilamt, kalantar, avk}@us.ibm.com

ABSTRACT

The design of composite service deployments (including software, hardware, network and storage), is complicated due to multiple cross cutting concerns, both functional and non-functional, and organizational division of expertise to multiple overlapping domains. In [1], we proposed an approach for deployment design that includes the usage of an abstract requirement graph, and its refinement into a concrete (physical) "desired state" deployment topology using a set of fine grained transformation rules expressing configuration knowledge, best practices and policy. The approach reduces the amortized complexity of the deployment process, and the associated risk, and is suitable for a distributed deployment design activity, where multiple domain experts collaborate, which is the reality in large enterprises. IBM Rational's deployment modeling platform is based on this approach.

In this paper, we lay out the theoretical foundations of the graph transformation approach for distributed design of composite service deployments. We propose a formal model for deployment design based on the Double Push Out (DPO) graph transformation technique. Our formal model includes a *configuration domain* containing abstract and concrete graph classes, and a *configuration framework* associating a graph transformation set with a configuration domain. We formally define what it means for a configuration framework to satisfy *correctness, completeness* and *convergence*. We demonstrate the approach on an example configuration framework (in the area of communication networks). We prove that the example configuration framework satisfies correctness and convergence, weak completeness, but not completeness.

1. INTRODUCTION

2. BACKGROUND AND CONTRIBUTION

The science of system and configuration management is concerned with reducing the total cost of ownership associated with large IT infrastructures and optimizing their multi-objective performance. Most academic work in system management focuses on optimizing a particular aspect, such as optimizing the placement of new components, performance tuning, or managing the configuration of individual components in isolation. In our research, we focus our attention on challenges that arise specifically in deployments of *composite* applications in *distributed* environments. Some of the specific challenges are attributed to the the complex, and often implied, cross dependencies between different types of resources such as software, system and network. Large organizations suffer from extreme inefficiencies in composite deployments that manifest themselves in unexpected and unpredictable labor cost and time spent.

We attribute the complexity, cost and unpredictability to the compositional nature of the deployment: multiple dependent components must be cross configured correctly, while satisfying cross-cutting concerns such as performance, availability, and security, and obeying organizational policy. It has been shown ([2, 3]) that most Internet service failures are due to operator mistakes, where mis-configuration is the most common mistake. In [4], a formal model is offered to quantify the complexity of the configuration process and it is claimed that a main source of complexity for operators is the need to cross configure systems consistently where parameter values must be copied across distant locations (both in time and space).

Another complicating aspect that is often overlooked is the large and distributed nature of the organization responsible for composite deployments. In large enterprises, multiple domain experts are involved in the deployment design and planning process, dividing the work to multiple, often overlapping, areas. A typical organization will be divided to teams separetly responsible for software, system and network configuration. Further, the software team may be subdivided to application vs. database experts; additional teams may be responsible infrastructure security, or performance tuning. This division, while necessary, greatly complicates the process, as decisions made by one expert affect the set of valid choices for other expert. These groups have no formal way to communicate the decisions they make, and their reasoning. Large enterprises report weeks wasted due to miscommunications. Any solution to the composite deployment and configuration problem must take into account the distributed nature of the deployment design process.

In [1], we proposed a new approach to address the challenges involved in composite deployment design. Our approach is based on a semantically rich formal language to describe deployment topologies, requirements and constraints in different levels of abstractions. Abstract topologies describe the high level structure of the topology (e.g., number of tiers, and division into zones) and the associated set of requirements, such as communication and capacity. Concrete topologies map these requirements to a detailed description of software stacks, network topologies, and storage. These concrete topologies include a detailed configuration specification and can serve to guide the actual deployment.

Further, in our approach we apply a set of model transformation rules to iteratively transform an input logical topology into a realizing concrete topology. Each transformation rule maps one requirement to a resource configuration structure specification implementing it. The same set of transformation rules can be reused in multiple deployments. The deployment design process proceeds in iterations where transformation rules are selected and applied in order to consistently refine the input requirement graph.

The benefits of our approach include support for a collaborative process among the different teams involved where decisions made are formally communicated through the use of the deployment model. The amortized complexity of the deployment process is reduced and the degree of assurance and predictability is increased due to the consistent use of model transformations that formally codify organizational policy and configuration rules. The set of transformations, defined once, can be reused in multiple different design activities, in or across organizations. The set of transformations may also vary to account for organization specific policy. IBM's Rational recently released a deployment modeling tool (packaged as part of the Rational Software Architect 7.5). that is based on the principles outlined here [5].

In this paper, we develop a theoretical framework to formalize the study of our approach for composite deployment design. We develop a formal model, where we use typed (attributed) graphs to represent abstract and concrete deployment topologies. We use the Double-Pushout graph transformation technique (e.g., [6]) to formally represent model transformation rules. In our formal model, a *configuration domain* includes definitions of the classes of abstract (requirement) graphs, concrete (physical) graphs, and a *realization relationship* that maps abstract graphs to valid concrete implementations. Note that an abstract requirement graph may have multiple valid implementations. The construction of a valid implementation for a given input will depend on many different factors such as organizational policy, and situational resource state and availability. A *configuration framework* includes a configuration domain and a set of transformation rules that codify a set of valid choices involved in the deployment design. The output of a *deployment design process* is a concrete graph that is a valid implementation of an input requirement graph. The deployment design process proceeds iteratively, where at each step a graph transformation rule is applied in a certain location in the graph. Note that the process lends itself to a distributed application where different experts will apply different subsets of the rules. The current topology is used to formally communicate requirements and decisions.

Further, we define and investigate key properties of configuration frameworks; we define what it means for a set of transformations to satisfy properties of *completeness*, *convergence*, and *correctness* w.r.t. a given configuration domain. We apply our formal model in the area of communication networks, where we formally define the classes of abstract and concrete graphs, the refinement relationship and an exemplary set of transformation rules (that is by itself useful for practical network configuration planning). We investigate the properties of our network framework and prove that the transformation set satisfies correctness, convergence, weak completeness, but not strong completeness.

To summarize, the novel contribution of this paper include

– Formal model for the investigation of the graph transformation approach to deployment and configuration design.
– Formal definition of correctness, completeness and convergence properties.
– Demonstration of the approach in the area of network configuration: the NET framework.
– Proof that NET satisfies correctness, weak completeness and convergence, but not strong completeness.

The structure of the paper is as follows. In the rest of this section we elaborate on the motivation for this work, and we review related works. Next, in Section 3, we define a formal model for deployment and configuration design using graph transformations. In Section 4, we introduce a specific example: the NET CM framework, for which we formally define the classes of abstract and concrete graphs, the realization relationship, and a set of transformation rules. In Section 5, we formally define key properties of CM frameworks. Last, in Section 6, we prove that the NET CM framework satisfies correctness, weak completeness and convergence, but not strong completeness. We conclude in Section 7 with a short summary.

2.1. Motivation

Configuration design involves multiple roles, and is complicated due to the need to incorporate knowledge from multiple management domains, understand and respect cross-domain resource configuration dependencies and constraints, and also honor and apply case-specific policy rules.

In our approach, a solution design process starts with a logical topology representing the components of the solution and their requirements. For example, Figure 1(a) shows a logical topology representing a typical three-tier Web Service with three logical nodes labeled `web`, `app` and `data` (of type `LDNode`). Each node represents one or more servers executing a defined set of applications. The annotation $filtered = true$ on the link between the `app` and `data` nodes indicates that communication between the link should be secure.

In each design iteration step, a transformation rule is applied to the current topology in order to replace an abstract requirement with a concrete structure representing a possible implementation. In many cases, there will be more than one possibility to satisfy a requirement. The process completes when the current topology is fully concrete (physical). The resulting topology describes a concrete resource configuration that satisfies all requirements in the input topology. Figure 1(b) shows a possible concrete topology that corresponds to the logical topology in the same figure. Note that there can be many different concrete topologies that realize a given logical topology.

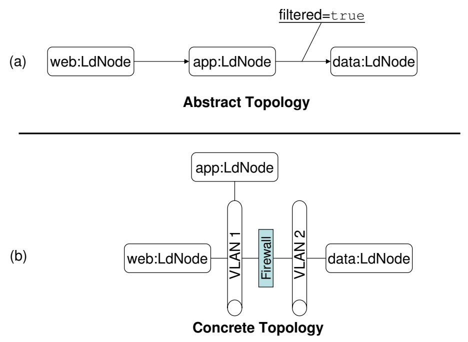

Fig. 1. Logical and Concrete topologies.

Hereafter, we informally present a set of transformations that can be used to construct the exemplary concrete topology (Figure 1) from the input logical topology (Figure 1). In fact, these are the same transformations that will be formally defined and explored later in subsequent sections. In our informal representation, each transformation rule contains a left hand side *pattern*, and a right hand side *transformer*. The patten is matched against the current topology to define the context, while the transformer informally illustrates the changes to the topology.

Figure 2 shows two connectivity transformations that share a common pattern. Both transformers express ways to replace the abstract requirement of two connected nodes into a sub graph that describes actual physically connectivity. The first transformer interconnects the nodes using a common VLAN. The second, interconnects the nodes using two disjoint VLANs routed through a firewall.

A third transformation, depicted in Figure 3 is used to combine two communication paths onto a single interface. The pattern identifies a logical node containing at least two connections (network interfaces). After applying the transformation, traffic over one of the connections is routed over the second connection. A firewall is inserted to ensure that the traffic pattern remains the same; that is, to prevent traffic from other nodes connected to VLAN 1 to nodes connected

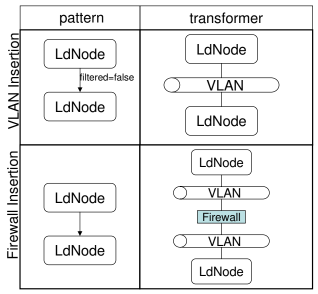

Fig. 2. VLAN insertion and firewall insertion transformations.

to VLAN 2. Prior to the consolidation, such traffic was prevented by the node itself, assuming that the multi-homed node is not routing traffic.

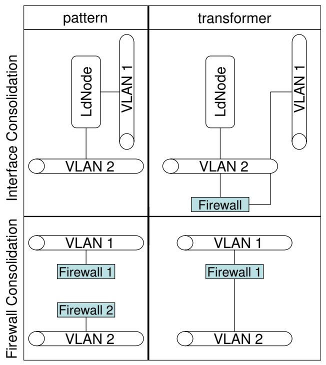

Fig. 3. Interface consolidation and firewall consolidation transformations.

Repeated application of the interface consolidation rule leads to a proliferation of firewalls. The last transformation, also shown in Figure 3, addresses this problem by converting two firewall requirements into a single firewall. In this transformation, any connections to one firewall are moved to the second firewall and the first firewall is removed from the topology. This can be done because a firewall is a multi-homed device that acts as a router in addition to filtering traffic.

Given these transformations, the aforementioned concrete topology (Figure 1(b)) can be generated from the input logical topology (Figure 1(a)) by sequentially applying (1) the VLAN insertion transformation on the `web-app` link, (2) the firewall insertion transformation on the `app-data` link, (3) the interface consolidation transformation on the `app` node, and (4) the Firewall consolidation transformation on the two firewall. See [1] for details.

2.2. Related Work

Approaches to IT deployment and configuration management can be roughly divided into two categories: (1) reactive: techniques to diagnose and potentially fix operator mistakes after they occur (2) proactive: methodology and tools for IT architects and operators to design and plan large deployments, increase the level of governance and assurance, and reduce the risks for mistakes. Clearly, both approaches are needed and are complementary. IT administration tends to be a bit more dis-organized and ad hoc in smaller companies, while in large enterprises, there is a serious attempt to put some order into the mess by adopting a clear methodology and a comprehensive set of tools for IT management. Even with the best attempt to follow a precise methodology, including pre-execution design, and review phases, techniques to discover mistakes and configuration drifts at runtime will be useful.

In the first category, many works attempt to discover, report, and potentially even automatically fix configuration errors. Many of these works focus on individual servers, irrespectful of their potential role in a larger composite service. In [7] mis-configurations in stand alone servers are diagnosed by pinpointing the instant in time where the system transitioned into an erroneous state. In [8], correctness constraints are automatically generated by analyzing a large number of Windows registries. Similarly, In [9], Windows registries of malfunctioning servers are compared against registries of healthy servers in an attempt to pin-point the problem. [10] use statistical techniques and a large sample of servers to attempt to automatically fix problems. All of the aforementioned works focus on single servers, cross dependencies and inconsistencies between servers comprising a composite service, resulting in an erroneous service behavior, are not handled. The focus of our work is to address the complexities that arise in *composite* service deployment and configuration.

Still focusing on the first approach (reactive), Validation [2] shares similarities with our work in focusing on Internet Services where intricate cross dependencies between configuration of multiple components exist. They attempt to detect operator mistakes by proposing a validation environment, to validate the new configuration state before re-configured components are migrated into the running system. [11] attempts to correct configuration errors by understanding interdependencies between components and automatically generating configuration files based on the semantic knowledge.

In contrast to the works described above, the approach described in this paper is "proactive". It focuses on providing methodology and tools for the design and planning of complex deployments such that constraints and cross-dependencies are evaluated *at design time* to create a detailed "desired state' topology specification (plan) that satisfies these constraints and dependencies "by design". A few other works followed the same path. The work described in [12], is a tool for IT architecture design based on the Architectural Description Standard [13]. While the tool is used extensively by IBM Global Services, it is manily used to describe IT architecture, and is not suitable for detailed deployment and configuration specification. Thus, it can be used to prevent only a subset of configuration mistakes and inconsistencies. SmartFrog [14], also focuses on composite deployments, and divides the deployment process into specification and execution stages. In the specification stage, the user defines an object relationship model that describes the components of the solution and their interdependencies. This model is interpreted and traversed in order to deploy the solution. The focus on a declarative object-relationship model to describe a "desired-state" is similar to our work, though the respective specification languages are quite different: in SmartFrog the specification language is Java-like, in our case, the language is based on XML and the topology can be expressed graphically. More fundamentally, SmartFrog does not discuss a way to construct the detailed "desired-state", or how to validate that indeed this specification meets all of the requirements and constraints. This paper, focuses on the collaborative and iterative process to define a detailed "desired-state" model based on a high level (logical) specification. We describe an approach to capture and leverage functional configuration rules, policy and best practices as a set of graph transformations used to iteratively construct the "desired state" topology.

Other works such as [15], and [16] focus on techniques to automatically generate provisioning workflows to automate deployments from a "desired state" specification. [15] generates the workflow using topological traversal of the graph assuming that temporal dependencies are included. In [16] we use AI planning techniques to match the desired state model to any given set of automation building blocks providing that they declare their requirements and affects. In contrast, this paper focuses on the construction of the "desired state" topology specification. The detailed topology that is the output of the distributed design process can be used as input to the workflow generation phase for automatic workflow generation (e.g., as we describe in [16]).

Last, cfengine [17] and LCFG [18] are two examples of technologies that attempt to automate the IT configuration. cfengine is based on a directive language that describes how a large class of machines should be configured. Similarly, LCFG is based on a declarative description of some aspects of the installation. These works are best suited for cases were a large number of individual resources have to be configured in a similar manner. The focus is on starting servers and installing software, not the configuration of installed software components. Both cfengine and LCFG have no semantic understanding that a set of resources comprise a higher level composite service, thus there is no way to infer and drive cross-configuration of components necessary for the correct behavior of the service as a whole.

Graph Grammar. Graph Grammer originated in the 60's and was applied significantly in multiple areas such as software engineering, pattern recognition, and even biology.

Multiple techniques as been proposed such as the Double-Pushout (DPO) approach, the node labeled controlled (NLC) approach. For a comprehensive overview of these and other graph transformation techniques see [6]. Numerous works focus on development of general purpose or specialized tools for graph transformations (e.g., PROGRESS and AGG) - see [19] for a comprehensive overview of these and other tools.

A substantial amount of research applies graph transformations in diverse areas. In software engineering research, graph transformations are used in the design of a software component model used in the design, development and life cycle of software (see, e.g., [20], [21], [22]). In [23] graph transformations are used to manage versioning of software under development. In [24], graph transformations are used to formally codify the permissible set of configuration changes at runtime. We are not aware of any work that attempts to use graph transformations in the design of IT deployments.

3. FORMAL MODEL

In this section we define a formal model for the multi-domain investigation of software and system deployment and configuration design using graph transformations. We demonstrate the approach in the area of network connectivity configuration design.

3.1. Preliminaries

Our graphs are node and edge labeled.

Graphs

Given two fixed alphabets Ω_V and Ω_E for node and edge labels, respectively, a *(labeled) graph (over* (Ω_V, Ω_E)*)* is a tuple $G = \langle G_V, G_E, s^G, t^G, l_v^G, l_e^G \rangle$, where G_V is a set of vertices (or nodes), G_E is a set of edges (or arcs), s^G, and t^G: $G_E \to G_V$ are the *source* and *target* functions, and $l_V^G : G_V \to \Omega_V$ and $l_E^G : G_E \to \Omega_E$ are the node and edge labeling functions, respectively. A *graph morphism* $f : G \to G'$ is a pair $f = \langle f_V : G_V \to G'_V, f_E : G_E \to G'_E \rangle$ which preserve sources, targets, and labels. The category having labeled graphs as objects and graph morphisms as arrow is called **Graph**. We interchangeably refer to node labels as *types*. We later extend the definition of our graphs to include attributes.

Given a category C and two arrows $b : A \to B, c : A \to C$ of C, a triple $\langle D, g : B \to D, f : C \to D \rangle$ is called pushout of $\langle b, c \rangle$ if (1) [Commutativity] $g \cdot b = f \cdot c$, and (2) [Universality] for all objects D' and arrows $g' : B \to D'$ and $f' : C \to D'$, with $g' \cdot b = f' \cdot c$, there exists a unique arrow $h : D \to D'$ such that $h \cdot g = g'$ and $h \cdot f = f'$.

Graph Transformations

For a comprehensive overview of graph transformations the reader is referred to [6]. We will be using one of the most common graph transformation techniques called *Double pushout (DPO)* (see [6]). In the Double Pushout (DPO) approach for graph transformations, a graph transformation rule is defined as a graph production, as follows. A *graph production* $p : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is composed of a *production name* p, and a pair of injective graph morphisms $l : K \to L$, $r : K \to R$. The graphs L, K, and R, are called the *lhs* (left hand side), *interface*, and *rhs* (right had side), respectively.

A double-pushout (DPO) diagram is a diagram as in figure 4, where 1 and 2 are pushouts. Given a rule p, the corresponding direct transformation on a graph G is denoted by $G \overset{p,m}{\Rightarrow} H$, where $m = (m_1, m_2, m_3)$ is termed the *context*.

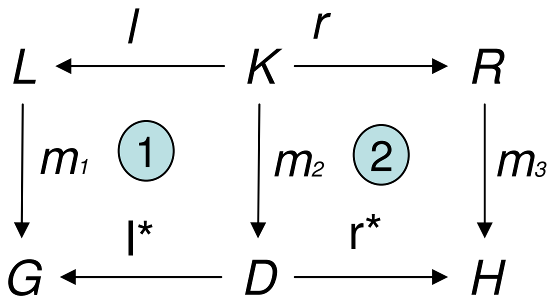
$$\begin{tikzcd}
L \arrow[d, "m_1"] \arrow[dr, phantom, "1"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "m_2"] \arrow[dr, phantom, "2"] & R \arrow[d, "m_3"] \\
G & D \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

Fig. 4. Double Pushout diagram

Intuitively, L specifies a sub-graph in the source graph G that is replaced when the production is applied with R, where K serves as an interface. Thus, to apply a production p to a graph G we find a match $m : L \to G$; we delete nodes and edges in $L \setminus K$; and we glue nodes and edges in $R \setminus K$ to produce a resulting graph H.

Formally,

Definition 1. *Given a graph G, a graph production* $p : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$*, and a match* $m : L \to G$*,* a direct derivation from G to H using p based on m *exists if and only if the diagram in Figure 4 can be constructed, where both squares are required to be pushouts in the category* **Graph**. *In this case we write* $G \overset{p,m}{\Rightarrow} H$.

For simplicity, for our needs, a simpler set theory definition may be used. Assume that all graphs are defined over the same alphabet (Ω_v, Ω_E) (so that $L \cup R$ is defined).

Definition 2. *A graph transformation rule can be represented as a production* $p : L \to R$*. Note that* K *can be reconstructed as* $L \cap R$*, thus can be omitted. A direct transformation from* G *to* H *using rule* $p : L \to R$ *is given by a* *graph morphism $m : L \cup R \rightarrow G \cup H$, called* occurrence*, such that*

1. *$m(L) \subseteq G$ and $m(R) \subseteq H$, (i.e., the left hand side is embedded into the pre-state and the right-hand side into the post-state).*
2. *$m(L \setminus R) = G \setminus H$, (i.e., the part of G matched against the parts of L not in R is deleted)*
3. *$m(R \setminus L) = H \setminus G$ (i.e., the parts of R not in L are added).*

The resulting graph H is determined up to isomorphism by the rule $p : L \rightarrow R$ and the occurrence m. Figure 5, is an example of a graph transformation rule, and its application to a graph G to produce a graph H (the use of attributes will be explained later in this section).

Definition 3. *A* sequential derivation *(over $\mathcal{G}$) is either a graph G (called an* identity derivation *and denoted by $G : G \overset{*}{\rightarrow} G$), or a sequence of direct derivations, $\rho = \{G_{i-1} \overset{p_i}{\Rightarrow} G_i\}_{i \in 1 \cdots, n}$ such that p_i is a production of $\mathcal{G}$ for all $i \in 1, \cdots, n$. In the last case, the derivation is written as $\rho : G_0 \overset{*}{\Rightarrow}_{\mathcal{G}} G_n$, or simply $\rho : G_0 \overset{*}{\Rightarrow} G_n$. If $\rho : G \overset{*}{\rightarrow} H$ is a (possibly identity) derivation, then graphs G and H are termed the* starting *and the* ending *graphs of ρ, and will be denoted by $\sigma(\rho)$ and $\tau(\rho)$, respectively. The* length *of a sequential derivation ρ is the number of direct derivations in ρ, if it is not identity, and* 0 *otherwise. The* sequential composition *of two sequential derivations ρ_1 and ρ_2 is defined if and only if $\tau(\rho_1) = \sigma(\rho_2)$, and denoted $\rho_1; \rho_2 : \sigma(\rho_1) \overset{*}{\Rightarrow} \tau(\rho_2)$ (and is obtained by identifying $\tau(\rho_1)$ with $\sigma(\rho_2)$).*

3.2. Formal Model for Deployment and Configuration Design with Graph Transformations

A *configuration management domain* (*CM domain*, in short) identifies a class of *abstract* graphs, representing logical deployment structure and requirements; a class of *concrete* graphs, representing concrete deployment implementation possibilities; and a *refinement relationship* identifying the valid concrete implementations for each abstract requirement graph. Note that for an abstract requirement graph, there could be any number (incl. 0) of valid concrete implementations. Graphs in the abstract class are valid inputs to the deployment and configuration design process, and graphs in the concrete class are possible outputs.

Formally,

Definition 4. *A* configuration management (CM) domain *(in short* domain*) $\mathcal{D}$ is a tuple ($\Omega_V = \Omega_V^A \cup \Omega_V^C, \Omega_E = \Omega_E^A \cup \Omega_E^C, \mathcal{G}^{\mathcal{A}}, \mathcal{G}^{\mathcal{C}}, \Gamma_{\mathcal{D}}$), where, $\mathcal{G}^{\mathcal{A}}$ ($\mathcal{G}^{\mathcal{C}}$, resp.) is a class of graphs termed* abstract *(*concrete*, resp.) over the node alphabet Ω_V^A (Ω_V^C, resp.) and edge alphabet Ω_E^A (Ω_E^C, resp.). The* refinement relationship *$\Gamma_{\mathcal{D}} \subseteq \mathcal{G}^{\mathcal{A}} \times \mathcal{G}^{\mathcal{C}}$ identifies valid concrete implementations. For graphs $G \in \mathcal{G}^{\mathcal{A}}$ and $H \in \mathcal{G}^{\mathcal{C}}$, we say that H* is a valid implementation of *G iff* $(G, H) \in \Gamma_{\mathcal{D}}$.

Definition 5. *A* configuration management framework *is a pair $(\mathcal{D}, \mathcal{T})$ where, $\mathcal{D}$ is a configuration management domain as defined above, and $\mathcal{T}$ is a set of productions (also termed* transformations*), representing configuration rules, defined over the class of all graphs with alphabet Ω_V and Ω_E.*

As discussed in Section 1, the motivation for the formal model stems from the need to support collaboration between domain experts in the area of deployment design and planning. A user in a role of an architect may define an abstract model representing a deployment on a high level, including requirements, and constraints. The architect does not have to understand the specifics of the infrastructure such as situational resource availability, network topology, or cost constraints.

At this point there may be multiple possible deployments that satisfy all of the requirements. To design a concrete deployment, multiple design decisions have to be made, such as selection of the specific middleware vendor and hardware technology, defining the physical network topology, and the configuration.

Large organizations usually have different experts responsible for defining the network configuration, allocating resources, and defining the configuration of the middleware elements. Each such domain expert will have to make design decisions based on policy, cost, resource availability and so on. In multiple cases, a decision of one domain expert will affect the valid choices available for a different domain expert (e.g., due to version competability issues, selection of a particular product and version will limit selection choices of other products and versions in different parts of the solution).

In our model, we formalize the collaborative design process using a set of transformation rules. Domain experts define transformation rules in their area of expertise to capture different aspects that have to be taken into account in the decision making process, and different implementation choices. The transformation rules are reused in multiple deployment design activities. Each transformation rule, when applied, changes the state of the model and the sub set of applicable transformations. The interaction between the domain experts is formalized through the use of the model that serves as an interchange format. Usage of transformation rules ensures consistency, and traceability, and reduces the risk of mistakes. At the end of the process a graph in the concrete graph class is reached that describes a detailed deployment.

Note that the set of transformation rules may and likely to vary between organizations, and so is the definition of what is considered a valid implementation for a given deployment requests (formally, the refinement relationship). Clearly, it is important to ensure that the set of transformations used is correct w.r.t. a given refinement relationship, namely, applying transformations to a logical graph will only result in concrete graphs that are considered valid implementations of the abstract graph. The correctness property, as well as other properties such as convergence and completeness, will be formally defined and investigated in subsequent sections.

The *deployment design activity* is formally defined as follows.

Definition 6. *Given a configuration management framework, $(\mathcal{D}, \mathcal{T})$, the* deployment design activity *over $(\mathcal{D}, \mathcal{T})$ is concerned with iteratively identifying a sequence of direct derivations $\rho : G \overset{*}{\Rightarrow} H$, for an input abstract graph $G \in \mathcal{G}^{\mathcal{A}}$, where $H \in \mathcal{G}^{c}$.*

Incorporating Attributes

It has been noted (e.g., in [25]) that graphs and graph transformations have limited applicability to software development design (and other areas) without extending the graph definitions to include attributes. The paper [25] presents and discusses the theory of attributed graphs, and graph transformations. For our needs, it is enough to consider a some how limited treatment of attributes in graphs.

In this paper, attributes may be associated with nodes, and edges. In our domain graphs, attributes are assigned values. In a production rule, attributes may take parameters, may be associated with constraints, or functions producing their value set.

Consider Figure 4, representing a Double Pushout construct. Attributes on nodes and edges on the lhs graph L have the affect of further restricting valid matches (m_1). In particular, constraints associated with attributes must be evaluated to $true$ on the corresponding attribute values in G. For a valid match, the attributes in L act as parameters that take their values from the corresponding attributes in G.

In the rhs graph R attributes are associated with functions that are exercised to generate new attribute values. If an attribute in R is assigned a value, then this value is interpreted as the singleton function. Note that we assume that attributes can be uniquely identified, thus a match implicitly produces a mapping between the respective attributes in the source and target graphs.

As an example, Figure 5(a) is a definition of a production, where one of the nodes is associated with an attribute x and a constraint $x \leq 10$. The rhs graph R defines a function $x' = a + 2$ associated with x that will be used to generate a new value. Figure 5(b) shows how the production is applied to a graph G to produce a graph H. First, a valid match is found (also satisfying the constraint on x) and a is assigned the value 1; second, the nodes and edges in $L \setminus K$ are deleted; third, the nodes and edges in $R \setminus K$ are added; last, the new value of x is calculated.

For readability and space conservation, the full formal treatment of attributes is deferred to the journal version of this paper.

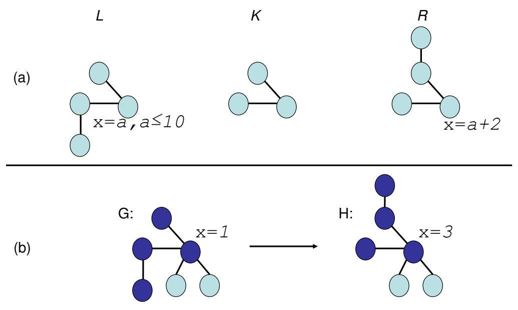

Fig. 5. Syntetic example of a production with attributes

4. NETWORK CONFIGURATION DESIGN

In this section, we show how we apply the formal model to the area (domain) of communication networks. In this example, abstract graphs represent logical communication requirements, and concrete graphs represent physical network configuration.

4.1. Definition of the NET Domain

Alphabets

We begin by defining the alphabets for our domain abstract and concrete graph classes.

- $\Omega_V = \Omega_V^A \cup \Omega_V^C$, where $\Omega_V^A = \{VNode\}$, $\Omega_V^C = \{VNode, VFW, PFW, VLAN\}$,
- $\Omega_E = \Omega_E^A \cup \Omega_E^C$, where $\Omega_E^A = \{VCON, VCONfiltered\}$, $\Omega_E^C = \{VCON, VCONfiltered, CON, REALIZE\}$.

Attributes: In our domain definition, every edge with a label in the set $\{VCON, VCONfiltered\}$ is associated with a boolean attribute $satisfied$ (taking values in the set $\{true, false\}$). Nodes of type $VLAN$ and VFW are associated with a boolean attribute $match$.

Abstract and Concrete Graph Class Definition We define the NET domain abstract and concrete graph classes.

The class $\mathcal{G}^{\mathcal{A}}_{\mathcal{NET}}$ of abstract graphs (also termed *input* or *requirement* graphs) includes all graphs over Ω_V^A and Ω_E^A where for every edge e (with labels $VCON$ or $VCONfiltered$), the attribute $satisfied$ is set to $false$, and all $match$ attributes on both VFW nodes and $VLAN$ nodes are set to $false$.

The class $\mathcal{G}^{\mathcal{C}}_{\mathcal{NET}}$ of *concrete* (also termed, *output*) graphs includes a subset of graphs over Ω_V^C and Ω_E^C satisfying the following conditions:

1. (Type rules) $VCON$ and $VCONfiltered$ edges connect nodes with labels $LDNODE$; CON edges connect nodes of type $VLAN$ with nodes in the set

$\{VFW, PFW, LDNODE\}$; $REALIZE$ edges connect VFWs and PFWs.
2. (communication-realization rule) All edges with labels in $\{VCON, VCONfiltered\}$ have $satisfied = true$
3. (firewall-realization-1 rule) All VFW nodes have $match = true$.
4. (firewall-realization-2 rule) For every pair of VFW node t and PFW node t' connected via a $REALIZE$ edge: every $VLAN$ node connected to t is also connected to t'.

As an example, consider Figure 9, where $G \in \mathcal{G}^{\mathcal{A}}_{\mathcal{NET}}$ and $H_1, H_2 \in \mathcal{G}^{\mathcal{C}}_{\mathcal{NET}}$.

The Refinement Relationship Γ_{NET}

To complete the definition of the network configuration domain NET, we define the refinement relationship Γ_{NET}. Recall that in a CM domain the refinement relationship Γ identifies for each abstract graph the set of concrete graphs that are considered valid implementations of it.

To define the implementation function Γ_{NET} we need the following preliminary definitions. An *unfiltered requirement path* in a graph $G \in \mathcal{G}^{\mathcal{A}}_{\mathcal{NET}}$ is a path between two nodes that does not contain any $VCONfiltered$ edges. A *direct connectivity requirement* exists between two nodes s, t in G if they are connected by a $VCON$ edge. For a graph $H \in \mathcal{G}^{\mathcal{C}}_{\mathcal{NET}}$, we define the *physical configuration subgraph* H^* as the subgraph of H obtained by deleting all VFW nodes (and their adjacent edges), and all $VCON$ and $VCONfiltered$ edges. Note that H^* is the portion of the graph that corresponds to the actual physical configuration of the network. An *unfiltered physical path* between two $VNode$s s, t in H^* is a path between s and t that does not contain any PFW nodes. A *physical direct connectivity* exists between two $VNode$s in H^* only if they are both connected to the same $VLAN$ node. As an example, consider again Figure 5, where graphs H_1 and H_2 are shown with their respective physical configuration subgraphs H_1^* and H_2^* (resp.).

Finally, the definition of Γ_{NET} follows. For graphs $G \in \mathcal{G}^{\mathcal{A}}_{\mathcal{NET}}$ and $H \in \mathcal{G}^{\mathcal{C}}_{\mathcal{NET}}$, let H^* be the physical configuration subgraph of H.

$(G, H) \in \Gamma_{NET}$ iff

1. For every $VNode$ $v \in G$, $v \in H$ (up to isomorphism). (i.e., no $VNode$s are deleted.)
2. A pair of $VNode$s is disconnected (no path) in H^* only if they are disconnected (no path) in G.
3. There exists an unfiltered physical path between two $VNode$s in H^* only if there exists an unfiltered requirement path in G between the corresponding nodes.
4. There exists a physical direct connectivity between two nodes in H^* only if there exists a direct connectivity requirement between the corresponding nodes in G.

Note that the definition of the refinement relationship Γ_{NET} above is concerned with maintaining the communication requirements in the concrete implementation. First, it ensures that no nodes get disconnected. Secondly, it allows for stricter implementation in some cases: a communication requirement is considered satisfied even if only filtered routes exist. Third, a communication path can be created between two disconnected nodes only if the path is filtered (the reasoning is that proper firewall rules can limit or prevent communication). Last, direct communication is allowed only if in the input graph the corresponding nodes are directly connected via a $VCON$ edge.

As an example, consider again Figure 9 where the graphs G, H_1, H_2, and the corresponding physical configuration subgraph H_1^*, H_2^* are shown. Note that both $(G, H_1) \in \Gamma_{NET}$ and $(G, H_2) \in \Gamma_{Net}$. The difference between H_1 and H_2 is that $VNode$ 3 is connected to only one $VLAN$ in H_1 permitting usage of a single homed server to implement $VNode$ 3. H_2 in contrast, mandates the use of a dual homed server for $VNode$ 3 implementation.

Definition of NET Framework

To define a CM framework $(NET, \mathcal{T}_{\mathcal{NET}})$ we now define the set of transformations $\mathcal{T}_{\mathcal{NET}}$.

$\mathcal{T}_{NET} = \{insertFW_i\}_{i=1,2} \cup \{insertVLAN, IFMerge\} \cup \{FWMerge_i\}_{i=1,2,3}$.

The transformations are formally defined in Figures 6,7, and, 8. Note the color coding used to distinguish between different node types.

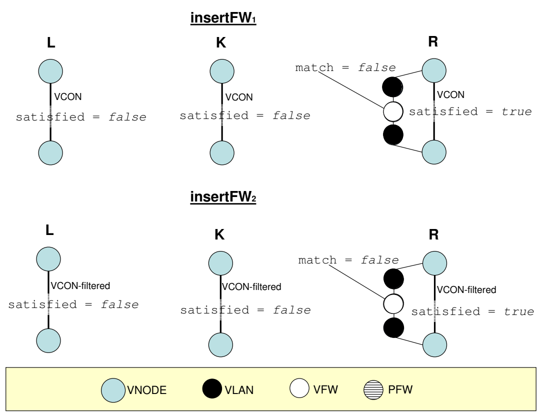

Fig. 6. The $\{insertFW_i\}_{i=1,2}$ family of transformations

The purpose of the $InsertFW_i$ transformation group is mapping connection requirements (expressed as a $VCON$ edge or a $VCONfiltered$ edge) to an implementation that includes a firewall. The purpose of the $insertVLAN$ transformation similarly, is to implement a connection requirement simply by using a virtual Local Area Network (LAN). Note that these transformations give the user the freedom to chose if to implement a virtual connection with no explicit requirement for a firewall ($VCON$ edge) with or without a firewall.

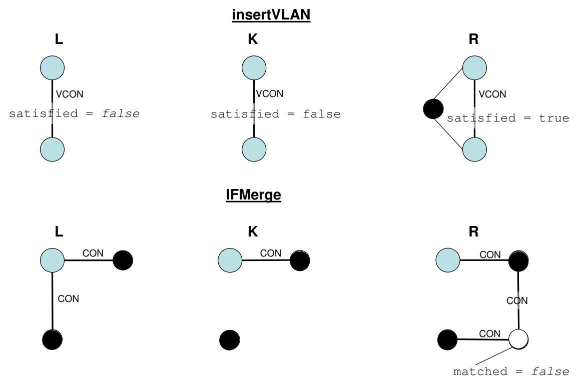

Fig. 7. The $insertVLAN$ and $IFMerge$ transformation

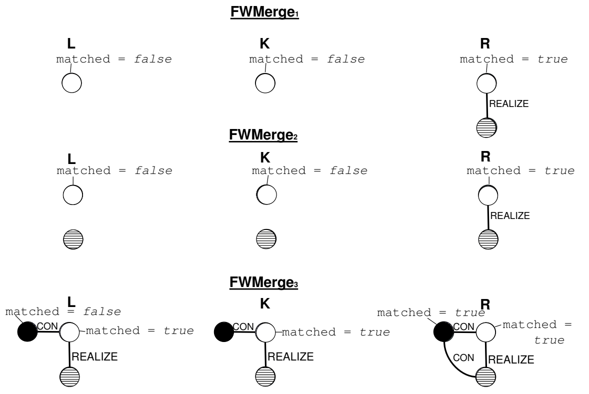

Fig. 8. The $\{FWMerge_i\}_{i=1,2,3}$ family of transformations

Also note the use of the attribute $satisfied$ to prevent multiple applications of the same transformation on the same context.

The purpose of the $IFMerge$ (read: interface merge) transformation is to allow usage of the same network interface to implement two different connections. The transformation will have a side affect of creating a route between *all* nodes connected to the two virtual LANs. The addition of a firewall on the route between the virtual LANs is needed to prevent undesirable connectivity as a result of this side affect. The $IFMerge$ transformation is necessary, since in a given infrastructure the number of interfaces per server is bound (usually it is a small number between 1 to 3).

Last, the $FWMerge_i$ transformation group maps virtual firewalls (VFW) to physical firewalls (PFW). A VFW node represents a requirement for firewall usage. In practice, a single firewall can be used to filter multiple different connections. The set of $REALIZE$ edges define the mapping between virtual firewall requirements (VFW) and actual firewalls to be used (PFW). $FWMerge_1$ adds a new PFW node to the graph; $FWMerge_2$ maps a virtual firewall requirement to an existing physical firewall (to allow multiple virtual firewall requirements to be implemented with a single physical firewall); the goal of $FWMerge_3$ is to connect every $VLAN$ adjacent to a VFW also to the realizing PFW (so that H^* will have all the information). The $matched$ attribute ensures that a virtual firewall is realized by at most one physical firewall.

As an example, consider Figure 9. The graph H_1 can be constructed from G by applying the following sequence of transformations: (1) $InsertVLAN$ on edges x and z, (2) $InsertFW_1$ on edge y, (3) $IFMerge$ on $VNode$ 3, (4) $FWMerge_1$ on VFW 5, (5) $FWMerge_2$ on VFW 6 (and the PFW added in the previous step), (6) three applications of $FWMerge_3$ to connect all $VLAN$ nodes to the PFW node.

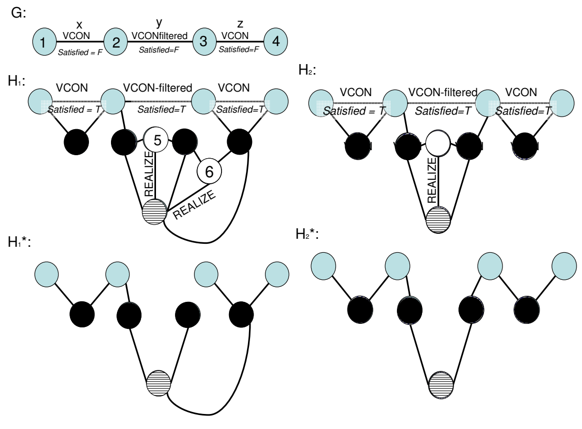

Fig. 9. Example of applying transformations in the NET domain

5. FRAMEWORK PROPERTIES

In this section, we define some useful properties for the investigation of CM frameworks. Specifically, we define what it means for a set of transformations to satisfy correctness, completeness, and convergence relative to a given CM domain.

Let $\mathcal{D} = (\Omega_V = \Omega_V^A \cup \Omega_V^C, \Omega_E = \Omega_E^A \cup \Omega_E^C, \mathcal{G}^\mathcal{A}, \mathcal{G}^\mathcal{C}, \Gamma_\mathcal{D})$ be a CM domain, and $\mathcal{T}$ is a transformation set over the class of graphs with alphabets Ω_V and Ω_E.

To effectively support a deployment design activity, it is necessary that the set of transformations used, will only lead to valid concrete implementation. Formally,

Definition 7. *A transformation set $\mathcal{T}$ satisfies* correctness *w.r.t. a CM domain $\mathcal{D}$ iff for every sequential derivation $G \overset{*}{\Rightarrow} H$, where $G \in \mathcal{G}^\mathcal{A}$ and $H \in \mathcal{G}^\mathcal{C}$, $(G,H) \in \Gamma_\mathcal{D}$.*

Another question of interest, is if the transformation set is strong enough to be able to generate *all* valid solutions. Formally,

Definition 8. *A transformation set $\mathcal{T}$ satisfies* completeness *w.r.t. a CM domain $\mathcal{D}$ iff for every $(G,H) \in \Gamma_\mathcal{D}$, there exists a sequential derivation $G \overset{*}{\Rightarrow} H$ over $\mathcal{T}$.*

In many cases, it is enough if the transformation set is able to generate at least one valid solution, in case such solution exists. Formally,

Definition 9. *A transformation set $\mathcal{T}$ satisfies* weak completeness *w.r.t. a CM domain $\mathcal{D}$ iff for every $(G, H) \in \Gamma_{\mathcal{D}}$, there exists $H' \in \mathcal{G}^{\mathcal{A}}$ where $(G, H') \in \Gamma_{\mathcal{D}}$, and a sequential derivation $G \overset{*}{\Rightarrow} H'$ over $\mathcal{T}$.*

The next property deals with characteristics of sequential derivations. For our usage of graph transformations it is sometimes desirable to have a transformation set that cannot produce infinite sequential derivations. Formally,

Definition 10. *$\mathcal{T}$ satisfies* convergence *w.r.t. a CM domain $\mathcal{D}$ iff there are no infinite sequential derivations $\rho \in \mathcal{T}$, where $\sigma(\rho) \in \mathcal{G}^{\mathcal{A}}$.*

This means that for any input graph, and for any valid application of transformations, eventually, a graph H is reached where there are no applicable transformations. Note that H may or may not be a member of $\mathcal{G}^{\mathcal{C}}$ (the later case is termed *dead end*).

Convergence is often a desirable property, but not strictly required. In some cases, if one wants to enable solutions that include unbound number of objects of a certain type, infinite sequential derivations are necessary.

6. CORRECTNESS, COMPLETENESS, CONVERGENCE IN OUR NETWORK CM DOMAIN

We investigate the correctness, completeness and convergence properties in our network domain. Specifically, we prove that the Γ_{NET} set of transformations satisfies correctness, weak completeness, and convergence, but not completeness w.r.t. the $\mathcal{D}_{\mathcal{NET}}$ CM domain. Weak completeness is satisfied even under stricter definition of $\mathcal{G}^{\mathcal{C}}$. We also reason about what is needed to satisfy completeness.

Proposition 1. *The transformation set $\mathcal{T}_{\mathcal{NET}}$ is* correct *w.r.t. the CM domain $\mathcal{D}_{\mathcal{NET}}$.*

proof.

We first observe that none of our transformations delete nodes. Thus, if a node exists in G it will also exist in any sequential derivation of G. We also observe, that no $VCON$ or $VCONfiltered$ edges are ever deleted.

The proof proceeds along the following steps. First, we define a transitive *refinement relationship* between graphs, denoted $\rightsquigarrow$. Second, we prove that if G' is a direct derivation of G, then $G \rightsquigarrow G'$. It follows: $G \overset{*}{\Rightarrow} H$ implies $G \rightsquigarrow H$. We complete the proof by showing that if $G \rightsquigarrow H$ and $H \in \mathcal{G}^{\mathcal{C}}$ then $(G, H) \in \Gamma_{NET}$.

To proceed with the proof we need the following preliminary definitions. A path in any a graph G (over alphabets Ω_V and Ω_E) between two $VNodes$ is *unfiltered* if it does not contain any VFW or PFW nodes, nor does it contain any $VCONfiltered$ edges.

Consider any two graphs G, G' over alphabet Ω_V and Ω_E (note, for both G, G' we do not assume membership in the classes $\mathcal{G}^{\mathcal{A}}$ or $\mathcal{G}^{\mathcal{C}}$). A graph G' is a *refinement* of G (denoted $G \rightsquigarrow G'$) iff the following conditions are satisfied.

1. (no disconnect) If $VNode$s s and t are connected in G they are also connected in G'.
2. (filtering) There exists an unfiltered path between two $VNodes$ in G' only if such path exists in G.
3. (no regression) If there exists a CON-path (ie, a path consisting only of CON edges) between two $VNode$s in G then there exists a CON-path between these nodes in G'.
4. (directness) Two $VNode$s are connected to the same $VLAN$ node in G' only if either they are connected to the same $VLAN$ node in G or they are connected via a $VCON$ edge in G.

The proof that the refinement relationship is transitive is left as an exercise for the reader (note that transitivity or property 4 is implied by the fact that no $VCON$ edges are ever deleted). Consider any two graphs G and G' (over (Ω_V, Ω_E)) where G' is a direct derivation of G ($G \overset{T}{\Rightarrow} G'$ and $T \in \mathcal{T}_{\mathcal{NET}}$). We prove $G \rightsquigarrow G'$. We first observe that only the transformation $IFMerge$ removes edges (it removes a single CON edge). However, this transformation has the affect of creating an alternative CON path between the two endpoints of the CON edge deleted. The "no disconnect" and "no regression" properties follow (for all of the transformations in $\mathcal{T}_{\mathcal{NET}}$). Next, we observe that only the $InsertVLAN$ transformation have the affect of connecting two $VNode$s to a same $VLAN$ node, and it is also the only one to create a new unfiltered path between two $VNode$s. Properties "filtering" and "directness" follow since the condition for $InsertVLAN$ is the existence of a $VCON$ path between the two $VNode$s (observe that the $VLAN$ node is a new node thus no undesired connectivity side affects are possible). This concludes the proof that $G \xrightarrow{T} G'$ implies $G \rightsquigarrow G'$. In fact, because of the transitivity of the refinement relationship, it follows that if $G \overset{*}{\Rightarrow} G'$ then $G \rightsquigarrow G'$.

Next, consider $G \overset{*}{\Rightarrow} H$, $H \in \mathcal{G}^{\mathcal{C}}$, and let H^* be its physical connectivity subgraph. From the above, $G \rightsquigarrow H$. First we prove $H \rightsquigarrow H^*$. Since $H \in \mathcal{G}^{\mathcal{C}}$, every $VCON$ and every $VCONfiltered$ edge has $satisfied = true$. The only transformations that set the $satisfied$ attribute is $InsertFW_i$ and $InsertVLAN$. These transformations must have been applied in the derivation sequence (since in $\mathcal{G}^{\mathcal{A}}$ $satisfied = false$). All these transformations have the affect of creating a CON path between the respective $VNode$s. Since we proved that all transformations preserve the refinement relationship the "no regression" property implies that a CON path exists between any two $VNodes$. Clearly, such CON path is not affected by the deletion of $VCON$ and $VCONfiltered$ edges. In addition, the firewall realization rules that are part of the definition of the class $\mathcal{G}^{\mathcal{C}}$ guarantee that any path containing a VFW has an alternative path going through the realizing PFW. Thus deletion of $VFWs$ does not disconnect CON paths. Properties "no disconnect" and "no regression" follow for H and H^*. Since H^* is a subgraph of H clearly "filtering" and "directness" are satisfied. It follows $H \rightsquigarrow H^*$ and thus $G \rightsquigarrow H^*$. Also observe that clearly $H^* \in \mathcal{G}^{\mathcal{C}}$.

To complete the proof we show that if for $G \in \mathcal{G}^{\mathcal{A}}$, $G \rightsquigarrow H \in \mathcal{G}^{\mathcal{C}}$ then $(G, H) \in \Gamma_{NET}$. Let H^* be the physical connectivity subgraph. An unfiltered path in H^* is guaranteed to be a unfiltered physical path. The rest of the conditions constituting Γ_{NET} follow directly from the definition of the refinement relationship.

□

Next, we show that our NET framework does not satisfy completeness. Refer to the example in Figure 10. Clearly, $(G, H) \in \Gamma_{Net}$ but H cannot be sequentially derived by applying a sequence of transformations in the set $\mathcal{T}_{\mathcal{NET}}$. Our set of transformations introduces a new $VLAN$ node for every $VCON$ edge. Consequently, the set of transformations Γ_{NET} may result in graphs that are non-optimal in the number of $VLAN$s used. If we were to extend our transformation set with an infinite number of transformations, each consisting of a lhs that is basically a clique (with increasing size) and a rhs that adds a $VLAN$ node connected to all of the $VNodes$ then this would allow re-use of $VLAN$ without compromising correctness (an example with 3 $LDNodes$ is shown in Figure 10).

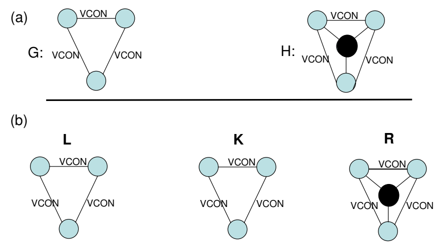

Fig. 10. Completness property counter example

Next, we show that weak completeness is satisfied.

Proposition 2. *The transformation set $\mathcal{T}_{\mathcal{NET}}$ satisfies weak completeness w.r.t. the domain $\mathcal{D}_{\mathcal{NET}}$.*

proof. We only have to apply a single $InsertFW_1$ transformation on each $VCONfiltered$ edge, and a single $InsertVLAN$ transformation on each $VCON$ edge to construct a graph $H \in \mathcal{G}^{\mathcal{C}}$, where $G \overset{*}{\Rightarrow} H$. □

More interestingly, weak completeness can be proved even if the class $\mathcal{G}^{\mathcal{C}}$ is constrained to have only $LDNodes$ with a degree smaller than a number x, for any $x > 1$, and/or the total number of PFW is bound (but not if we also put a limit on the degree of PFWs). These are very practical constraints, since most data center servers have 3 or less network interface cards. The proof is deferred to the journal version.

Last, we prove convergence.

Proposition 3. *The transformation set $\mathcal{T}_{\mathcal{NET}}$ satisfies convergence w.r.t. the CM domain $\mathcal{D}_{\mathcal{NET}}$.*

The number of $InsertFW_i$ and $InsertVLAN$ transformations in any sequential derivation $G \overset{*}{\Rightarrow} G'$ is bounded by the number of $VCON$ and $VCONfiltered$ edges in G (since each one of them sets $satisfied$ to $true$ which prevents subsequent applications). The number of $IFMerge$ transformations is bounded by the sum of degrees of all $LDNodes$ in G (formally, $\sum_{v \in G_v, l(v)=VNode} deg(v)$, where $deg(v)$ is the number of adjacent edges) since each such transformation reduces the degree of a $LDNode$ by 1. The maximum number of VFW nodes in any graph that is sequentially derived from G is bounded since it must be smaller that the number of $InsertFW_i$ rules applied plus the number of $IFMerge$ rules applied (since these are the only rules to add VFW). Similarly, the maximum number of $VLANs$ is bounded as a function of the number of $InsertFW_i$ and $InsertVLAN$ transformations applied. Last, the number of $FWMerge_i$ transformations is also bounded. $FWMerge_1$ and $FWMerge_2$ are bounded by the number of VFW, and $FWMerge_3$ is bounded by the number of $(VLAN, VFW)$ pairs.

□

.

7. SUMMARY

In this paper we proposed a theoretical framework for the investigation of the graph transformation approach for composite deployment design. We formally defined the concepts of a *CM domain* comprising abstract and concrete graph classes, and the realization relationship. We further defined the concept of a *CM framework* consisting of a configuration domain and a set of graph transformations that can be used to construct graphs in the concrete graph class. We defined what it means for a CM framework to satisfy properties of *correctness*, *completness* and *convergense*. We demonstrated the concepts with an example CM domain and CM framework, termed NET, focusing on configuration of networks for communication. We proved that NET satisfies correctness, convergence, and weak completness.

Future suggested work will explore the expressivness of the graph transformation approach for distributed deployment design and its limitations. Another interesting direction is to quantify deployment complexity reduction and error reduction using the proposed approach.

8. REFERENCES

[1] Eilam, T., Kalantar, M., Konstantinou, A., Pacifici, G.: Reducing the complexity of application deployment in large data centers. In: IM. (2005)

[2] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R.P., NGUYEN, T.D.: Understanding and dealing with operator mistakes in internet services. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI 04). (2004)

[3] OPPENHEIMER, D., GANAPATHI, A., PATTERSON, D.: Why do internet services fail, and what can be done about it? In: USENIX Symposium on Internet Technologies and Systems (USITS03). (2003)

[4] Brown, A.B., Keller, A., Hellerstein, J.: A model of configuration complexity and its applications to a change management system. In: IM. (2005)

[5] IBM: Rational Software Architect (RSA) (2008)

[6] Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation: vol. 1: Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)

[7] Whitaker, A., Cox, R.S., Gribble, S.D.: Configuration debugging as search: finding the needle in the haystack. In: OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2004) 6–6

[8] Kycyman, E.: Discovering correctness constraints for self-management of system configuration. In: ICAC '04: Proceedings of the First International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2004) 28–35

[9] min Wang, Y., Verbowski, C., Dunagan, J., Chen, Y., Wang, H.J., Yuan, C.: Strider: A black-box, state-based approach to change and configuration management and support. In: In Usenix LISA. (2003) 159–172

[10] WANG, H.J., PLATT, J.C., CHEN, Y., ZHANG, R., WANG, Y.M.: Automatic miscoguration troubleshooting with peerpressure. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). (2004)

[11] Zheng, W., Bianchini, R., Nguyen, T.D.: Automatic configuration of internet services. SIGOPS Oper. Syst. Rev. **41**(3) (2007) 219–229

[12] Abrams, S., Bloom, B., Keyser, P., Kimelman, D., Nelson, E., Neuberger, W., Roth, T., Simmonds, I., Tang, S., Vlissides, J.: Architectural thinking and modeling with the architects' workbench. IBM Syst. J. **45**(3) (2006) 481–500

[13] Youngs, R., Redmond-Pyle, D., Spaas, P., Kahan, E.: A standard for architecture description,. IBM Syst. J. **38**(1) (1999) 32–50

[14] Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., Toft, P.: The smartfrog configuration management framework. SIGOPS Oper. Syst. Rev. **43**(1) (2009) 16–25

[15] Keller, A., Hellerstein, J., Wolf, J., Wu, K.L., Krishnan, V.: The CHAMPS system: change management with planning, and scheduling. In: NOMS, IEEE Press (2004)

[16] El Maghraoui, K., Meghranjani, A., Eilam, T., Kalantar, M., Konstantinou, A.: Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In: Middleware. Volume 4290 of LNCS., Springer (2006)

[17] Burgess, M.: Cfengine: A site configuration engine. In: USENIX Computing Systems, Vol. 8, No. 3. (1995)

[18] ANDERSON, P., SCOBIE, A.: Lcfg: The next generation. In: UKUUG Winter Conference. (2002)

[19] Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999)

[20] Grunske, L., Geiger, L., Zndorf, A., Eetvelde, N.V., Gorp, P.V., D., V.: Using graph transformation for practical model-driven software engineering. Model-Driven Software Development (2005) 91–117

[21] Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. **44**(2) (2002) 133–155

[22] Stanciulescu, A., Vanderdonckt, J., Mens, T.: Colored graph transformation rules for model-driven engineering of multi-target systems. In: GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations, New York, NY, USA, ACM (2008) 37–44

[23] Parisi-Presicce, F., Wolf, A.L.: Foundations for software configuration management policies using graph transformations. In: FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering, London, UK, Springer-Verlag (2000) 304–318

[24] Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations, London, UK, Springer-Verlag (2000) 179–193

[25] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundam. Inf. **74**(1) (2006) 31–61