Automatic Realization of SOA Deployment Patterns in Distributed Environments

William Arnold, Tamar Eilam, Michael Kalantar, Alexander V. Konstantinou, and Alexander A. Totok

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

Abstract. Deployment patterns have been proposed as a mechanism to support the provisioning of SOA-based services. Deployment patterns represent the structure and constraints of composite solutions, including non-functional properties, such as performance, availability, and security, without binding to specific resource instances. In previous work [1], we have presented a formal mechanism for capturing such service deployment patterns using models. Our pattern models define abstract connectivity and configuration requirements which are then *realized* by an existing or planned infrastructure. Realization mapping is used to enforce policies, and is materialized at deployment time. In this paper we extend that work to address the problem of *automatic pattern realization* over a given infrastructure. We first formalize the problem and present three variations of increasing power and complexity. We then present a variation of a search-based graph isomorphism algorithm with extensions for our pattern model semantics. Next, we show that our worst-case exponential complexity algorithm performs well in practice, over a number of pattern and infrastructure combinations. We speculate that this is because deployment topologies represent heavily labeled and sparse graphs. We present a number of heuristics which we have compared experimentally, and have identified one which performs best across most scenarios. Our algorithm has been incorporated into a large deployment modeling platform, now part of the IBM Rational Software Architect (RSA) tool [2].

1 Introduction

From the perspective of an SOA deployer, the service layer specifying service hosting, connectivity, and binding can often be viewed as the tip of an iceberg. SOA services are typically implemented as components of distributed application platforms supported by large and complex middleware containers. These containers are often dependent on other remote middleware servers for messaging, database management, and authentication. The servers on which these containers execute must be monitored, secured, and audited and therefore have their own connectivity requirements. Server communication capabilities are constrained by the topology of the networks to which they are connected. The fact

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 162–179, 2008.

that many of these communication paths are interdependent through layering [3], but often separately managed, presents a great challenge to SOA deployers [4].

Deployment patterns [5,6,1,7,8] have been proposed as an answer to the complexity of SOA deployment and the often subtle and difficult to quantify interactions and trade-offs between functional requirements, performance, security, and availability. There are several ways in which deployment patterns prove to be helpful. First, they simplify service deployment by codifying *best practices*. Second, they capture complex interdependent resource configurations that collectively achieve certain *non-functional* infrastructure properties, such as high availability, scalability, and security. These properties can then be used to satisfy non-functional requirements (NFRs) of the services being deployed. Third, deployment patterns capture intrinsic properties of composite services, while allowing them to be deployed in different environments, such as development, testing and production.

In our recent work [1] we presented a novel approach to formally capturing SOA deployment patterns. Our deployment patterns represent abstract deployment topologies specified at various levels of abstraction. They capture the structure and constraints of a composite solution, without bindings to specific resources, and without specifying provisioning actions. Deployment patterns are instantiated by deployers through *realization* of patterns in deployment topologies, representing existing or desired state of the infrastructure. Using our deployment platform, we enable non-expert users to safely compose and iteratively refine deployment patterns, resulting in fully specified topologies with bindings to specific resources. The resulting desired state topology can be validated as satisfying the functional service requirements, while maintaining the non-functional properties of the pattern.

In this paper, we turn our attention to the problem of *automatic pattern realization*, which is a function that produces a *valid realization* of a pattern in a given target infrastructure. There are several use cases for pattern autorealization. First, it can greatly simplify the work of service deployers: if a composite solution's hosting requirements are represented as a deployment pattern, the task of solution deployment can potentially be reduced to running a simple automatic pattern realization wizard performing automatic resource selection. Second, it can be used for *compliance* purposes to verify that a given infrastructure conforms to certain organizational constraints, policies and best practices, which are captured in deployment patterns. Third, it can enable deployment *impact analysis*: the ability to plan deployment changes by playing "what-if" scenarios and assessing the planned changes. In addition, automatic pattern realization that supports *infrastructure reconfiguration* can be used for infrastructure provisioning, to drive reconfiguration of the infrastructure to conform to the pattern structure and constraints.

Our approach to the automatic pattern realization problem is based on the observation that it is reducible (with some variations) to the subgraph isomorphism problem [9]. We view deployment topologies as *labeled graphs* (augmented with constraints), and infrastructure reconfiguration actions as *graph edit*

operations [10]. We propose to use (modified) *graph matching* algorithms [9] for the pattern realization that does not allow changes to the target infrastructure, and *error-correcting graph matching* algorithms [10] for the situation where infrastructure reconfiguration is necessary. We present an algorithm for automatic pattern realization for the case, where no changes are allowed to the target infrastructure. We implement the algorithm in a large deployment modeling platform [1] and analyze its performance on a set of real-life deployment scenarios. We show that the algorithm's performance depends on the heuristic used to navigate the problem's *search tree*. We analyze performance of several such heuristics and identify one which provides good performance across a range of patterns and infrastructures. Our results show the practicality of using the algorithm, although its worst case complexity is exponential. We speculate that this is because deployment topologies represent heavily labeled and sparse graphs.

The paper is structured as follows. In Section 2, we describe our deployment pattern modeling platform. In Section 3, we formalize the problem of automatic pattern realization and introduce a number of variations. In Section 4, we present the algorithm for automatic pattern realization in a common case, where no changes are allowed to the target infrastructure. We analyze the behavior of the algorithm in Section 5. Finally, we discuss related work in Section 6, and conclude in Section 7.

2 Deployment Modeling and Validation

Our model-driven SOA deployment platform [1] supports the construction of deployment models at different levels of abstraction, ranging from *abstract models* (also termed, *patterns*) to *concrete*. Abstract models capture only intrinsic properties of a reusable deployment solution, they partially specify the configuration of resources, focusing on key parameters and structures, while leaving others to be determined at deployment time. Concrete models include detailed software stacks and configurations; valid and complete deployment models can be consumed by provisioning automation technologies (such as, Tivoli Provisioning Manager (TPM) [11]) to drive automated provisioning [12]. The platform, modeling concepts, and principles were introduced in [1], and are partially repeated here for completeness. To simplify the presentation, some definitions are simplified, where this does not affect the algorithm spirit and principles. For a complete description of the platform, modeling language, and analytic capabilities see [1].

The core model captures common aspects of deployment and configuration syntax, structure and semantics. Core types are extended to capture domain-specific information. Domain objects and links are contained in a *Topology* which is used to represent a composite solution. Figure 1 is an example of a deployment model (*Topology*). The *Unit* core type represents a unit of deployment, which may correspond to a hardware resource (x86 Server), a software product (Windows XP OS, WebSphere Application Server), or a software configuration node (J2EE Datasource). Subtypes of *Unit* group domain-specific configuration

attributes.[1] A *Unit* may represent an *installed* resource, or a resource *to be installed*. The *state* attribute on *Units* is an ordered pair which represents the *initial* and *desired* state of a *Unit*. In the example, *Windows2000Unit* and *Was6Unit* represent installed resources, and all other units are to be installed. Note that deployment topology examples used throughout the paper tend to either be abstract (not referring to specific resource types) or present low levels of SOA infrastructures: they were chosen to be just simple enough to demonstrate the idea of deployment pattern realization. We are continuing the work on modeling system and software configurations containing higher levels of SOA stacks and supporting more sophisticated considerations, such as messaging, security, high availability, etc.

Resource dependencies and requirements on other resources are represented by *Requirement* objects, contained in units. A *Requirement* is a tuple (t_r, t_l), where t_r represents that *type of required resource*, and t_l represents the *type of relationship (link)*. We define three *relationship types*: a many-to-one *hosting* relationship; a one-to-one *dependency* relationship, and a many-to-many *membership* relationship. Relationships between *Units* are represented by a *Link* object, which is a quadruple (u, r, v, t), where r is a *Requirement* contained in *Unit* u, t is the *type* of the link (i.e., *hosting*, *dependency*, or *member*), and the link connects the *Requirement* r and a target *Unit* v.

Local constraints can be defined and contained in *Units*, where the context of evaluation is the containing unit, or in *Requirements*, where the context of evaluation is the associated relationship's target unit. An example of such a local constraint is "`version` $\geq$ `1.4`" contained in a requirement of type *hosting* (contained in the *EARUnit*) for a target resource of type *J2EEContainer* (Figure 1), which restrict the version of the hosting application server. Note that *WAS6Unit* is a subtype of *AppServerUnit*, thus the constraint is satisfied in the example topology. A special *Membership constraint*, contained in a *Requirement* of type *membership*, can further restrict the multiplicity of a membership relationship instance.

Structural constraints, operating on pairs of *Units* can also be defined via a *Constraint Link*. Two common constraint links are: *Collocation* and *Deferred Hosting*. A *Collocation* constraint restricts the valid hosting of two units. It is associated with a *target type* property which determines the type of the host on which the two units' hosting stacks must converge (anti-collocation can be defined similarly). *Deferred Hosting* is a constraint that the source unit be eventually (indirectly) hosted on the target of the link.

Semantics and Validation. Topologies are evaluated against a set of *core platform validation rules*, including a core set of local constraints ($=$, $\leq$, $\geq$, $<$, $>$, ...), the two structural constraints described above, and link validity rules (multiplicity and endpoint types). The platform is extensible with (domain specific) validation rules and constraints (see [1]). Validation rules and constraints produce validation statuses of three types: *satisfied*, *undefined*, and *violated*.

[1] Some of the modeling concepts from [1] are simplified in this paper to focus on the algorithmic aspect. In particular, we collapsed the *Capability* concept into *Unit*.

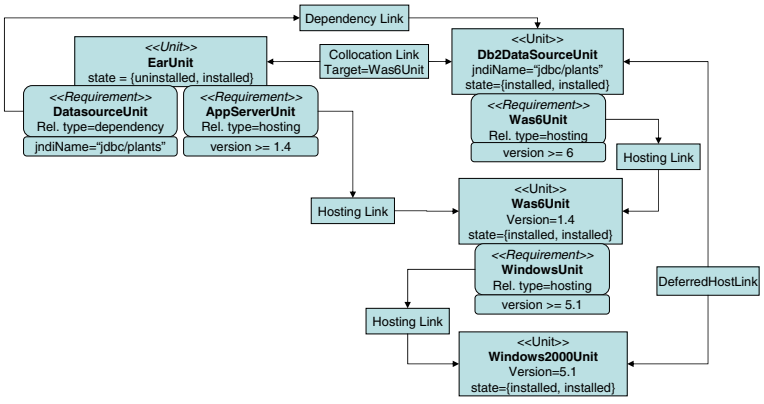

Fig. 1. An example deployment model

Undefined statuses will be generated in incomplete topologies, where there is not enough information to validate a rule or a constraint. For example, a *Requirement* that is not yet linked with a target unit, or a *Deferred Hosting* constraint connecting two units with an incomplete software stack, will both produce an *undefined* status. *Violated* statuses will be generated when units are improperly connected or when an attribute on a unit has an invalid value. For example, a link $l{=}(u, r, v, t)$ that connects requirement $r{=}(t_r, t_l)$ with a target unit v, where $t \neq t_l$ or $type(v)$ is not a subtype of t_r will produce a *violated* status. Note that a *violated* status can not be turned into a *valid* or *undefined* status just by *adding* units and links to the topology. A topology is *weakly valid* if its validation does not produce any *violated* statuses; *valid* if only *satisfied* statuses are produced; and *invalid* otherwise. Note that complete topologies (where all attributes are set, and all requirements are associated with links), can either be *valid* or *invalid*. Complete and valid topologies can be consumed by a provisioning technology for automated deployment. The topology in Figure 1 is valid and complete.

Virtual Units and Realization Links. To allow modeling at different levels of abstraction, we introduced the concept of a *Virtual Unit*. A virtual unit is one which does not directly represent an existing or an installable resource, but instead should be *realized* by another (concrete) unit. Typically, virtual units will be of an abstract type, will include attributes with unspecified values, and will be associated with constraints. A *Realization Link* connects a virtual unit with a concrete unit that forms its realization.

Topology Realization Semantics. It is convenient to separate the validation of the realization mapping from the core validation operating on topologies with only concrete units. The rules for locally validating realization of a virtual unit by a concrete unit are formally defined in two stages as follows. For any two requirements $r{=}(t_r, t_l)$ and $r'{=}(t'_r, t'_l)$, $match(r, r')$ iff $t_l = t'_l$ and $supertype(t_r, t'_r)$. We say that a concrete unit u_2 is a *locally valid realization* of a virtual unit u_1

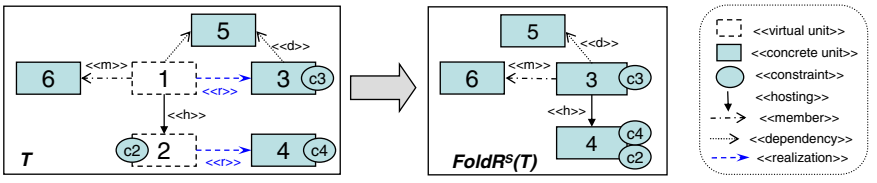

Fig. 2. A realization folding example

(denoted $validR(u_1, u_2)$) iff (1) $supertype(type(u_1), type(u_2))$, (2) for every attribute $a \in attributes(type(u_1))$, $isSet(u_1, a) \rightarrow value\ (u_1, a) = value(u_2, a)$, (3) for every constraint $c \in constraints(u_1), c(u_2) = satisfied$, (4) there exists a *unique* mapping of the set of requirements $r_1, r_2, \ldots r_n$ on unit u_1 to a set of *distinct* requirements $r'_{i_1}, r'_{i_2}, \ldots r'_{i_n}$ on unit u_2 (denoted map^R_{req}, w.r.t. unit realization mapping R), such that $match(r_k, r'_{i_k})$,[2] and (5) for every constraint $c \in constraints(r_k)$, $c(r'_{i_k}) \in \{satisfied, undefined\}$ (inclusion of the *undefined* validation status accounts for the fact that r'_{i_k}'s target may not be defined yet).

Given a topology with virtual units and realization links, it is not enough to locally check the validity of individual realization links. For example, consider a virtual unit u hosted on a non virtual unit v. A valid local realization of u can map it to a non virtual unit u' hosted on a non virtual unit v', where $v' \neq v$, thus violating the hosting relationship many-to-one multiplicity constraint. To fully validate realization mappings, we define the *strict folded topology* $FoldR^S(T)$ of a given topology T, where, intuitively, we collapse all realized virtual units, relationships and constraints (into the respective concrete units), and remove unrealized virtual units (see [1] for a formal definition, and Figure 2 for an example).

We say that a topology T forms a *weakly valid topology realization* iff (1) every virtual unit is realized by at most one unit, (2) each realization link in T is locally valid, and (3) $FoldR^S(T)$ is *weakly valid*. A topology T forms a *valid topology realization* iff $FoldR^S(T)$ in the definition above is *valid*. A topology realization is *complete* when all its virtual units are realized. Note that $FoldR^S(T)$, for a complete valid (or weakly valid) topology realization T, is a more succinct (normalized) representation of the deployment information available in T. In particular, new links, local, and structural constraints may be introduced on concrete and pairs of concrete units. Condition (3) prevents realizations that violate link multiplicity constraints, and checks validity of local and structural constrains defined on virtual units, against their corresponding realizing units. $FoldR^S(T)$ can substitute T in an iterative, pattern based, deployment planning process. We will use $FoldR^S(T)$ later, when defining the auto-realization algorithms. An example of topology realization is illustrated in Figure 2. Note that the folding introduces a new membership link between units 3 and 6. This

[2] To simplify the presentation of some of the definitions and algorithms, we assume that the injective mapping map^R_{req} is unique. The definitions and algorithms for pattern realization can be easily generalized to deal with multiple such mappings.

does not violate condition 3 in the definition above since the multiplicity of membership links is many-to-many. Assuming that local constraints are all satisfied on the respective concrete units, the figure represents a valid and complete topology realization. Finally, we say that a complete topology realization T is *strict* if T further satisfies the following property: for every non-constraint link $l{=}(u, r, v, t) \in T$, where both units u and v are virtual, there exists a corresponding link $l'{=}(u', r', v', t') \in T$, such that $u' = R(u)$, $v' = R(v)$, $r' = map^{R}_{req}(r)$, and $t = t'$, where R is the realization mapping function. This property requires effectively that no non-constraint links are "inherited" by a pair of concrete units from the pair of virtual units that they realize, during the topology folding process. Topology realization in Figure 2 does not satisfy this property, because the hosting link between units 1 and 2 does not have corresponding link between units 3 and 4.

3 Automatic Pattern Realization

In this section, we formally introduce the pattern realization problem, and several variants of it. We discuss the motivation for the problem and its variants based on real life use cases.

Let P be a *pattern topology*, where all units are virtual, and let T be a *target topology*, where all units are concrete. Let R be a set of realization links between units in P and T. We denote by $P \cup T \cup R$ the topology formed by taking the union of P, T, and R (following the common definition of graph unions). The *Pattern Realization (PR) problem* is formally defined as follows.

The PR Problem. Given a pattern topology P and a target topology T, produce $PR = T \cup P \cup R$, where R is a set of realization links between units in P and T, and PR forms a *complete weakly valid topology realization* (as defined in Section 2). ■

Note that the definition above can be easily generalized, where P and/or T contain both virtual and concrete units, and realization links. The PR problem definition is useful to describe the process of incremental elaboration and refinement for pattern-based deployment, where a basic step maps an abstract (pattern) topology into a concrete (but maybe incomplete) topology, in which some of the units represent resources that are installed or "to be installed". Each such mapping step produces a normalized folded concrete topology that conforms to the input pattern, and can be used as input to the next elaboration and refinement step, eventually leading to a valid and complete deployment topology that satisfies multiple required patterns and that can serve to drive automated provisioning.

There are several variants of the problem that we find useful in real life scenarios. Specifically, consider a situation where the target topology T represents an existing computing infrastructure (where all units are *installed*). A very common case is where no changes are allowed to the infrastructure. In such a case, the automatic pattern realization process may be used for (1) resource selection, where a pattern topology is used to select an environment with certain

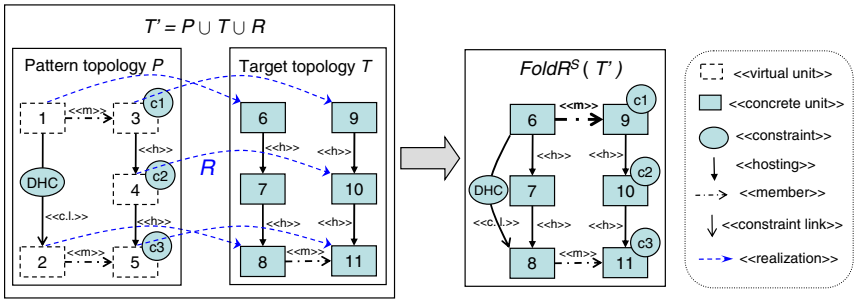

Fig. 3. Example of pattern realization

characteristics that can serve as, e.g., a hosting environment for downstream composite service deployment, or for (2) IT compliance verification, where a computing infrastructure is continuously validated against a set of organizational constraints, policies and best practices (represented as patterns). We term this variant of the problem *Strict Pattern Realization in Infrastructure Topology (PRIT)*. The *Strict PRIT Problem* is formally defined below.

The Strict PRIT Problem. Given a pattern topology P, and a target topology T, produce $PRIT^S = P \cup T \cup R$, where R is a set of realization links between units in P and T, and $PRIT^S$ forms a *strict valid and complete topology realization* (as defined in Section 2). ■

Note that there are two differences between the *PR* and the *Strict PRIT* problems. In *Strict PRIT* the goal is to produce a *valid* topology realization (not just weakly valid). Further, the "strictness" property mandates that effectively no new non-constraint links are added between concrete units in the folded topology $FoldR^S(PRIT^S)$. Moreover, the only objects "inherited" by the folded topology $FoldR^S(PRIT^S)$ from the pattern topology P are constraints and constraint links). Indeed, both these requirements stem from the fact that no changes are allowed in the infrastructure; new links such as a new membership link between two units, imply necessary infrastructure reconfiguration. For example, Figure 3 represents realization of pattern topology P (which contains among other constraints, a Deferred Hosting constraint) in target topology T. Topology $T' = P \cup T \cup R$ is a complete valid topology realization (and thus a valid output of the *PR* problem), but not a strict topology realization, because it introduces a new member relationship between units 6 and 9 (in the corresponding folded topology $FoldR^S(T')$). Thus it is not a valid output of the *Strict PRIT* problem.

Pattern realization with infrastructure reconfiguration. Consider a case where a deployment pattern does not have a Strict PRIT realization in the target infrastructure. However, if changes are allowed to the infrastructure, we may be able to *modify* it in a way that the pattern's realization in the infrastructure is possible. In such a case, the automatic pattern realization process may be used to

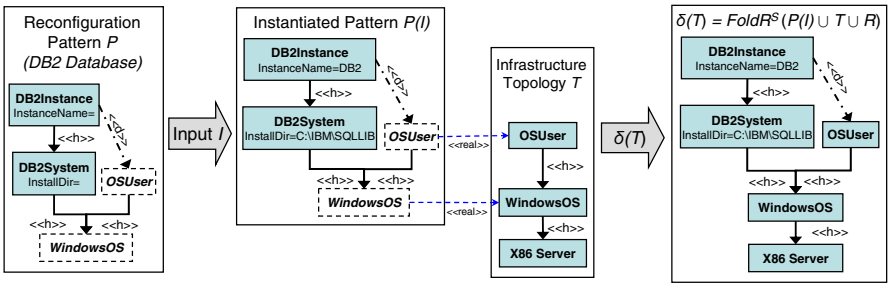

Fig. 4. Example of valid application of a reconfiguration action to an infrastructure topology

drive *reconfiguration* of the infrastructure toward a state where it conforms to the pattern structure and constraints. We term this variant of the problem *Relaxed Pattern Realization in Infrastructure Topology (Relaxed PRIT)*. Note that ability to reconfigure the infrastructure to enable pattern realization depends on the set of allowed *infrastructure reconfiguration actions*. We first present a formal model for reconfiguration actions.

Infrastructure reconfiguration actions may include adding new hardware resources (e.g., adding new servers from the pool of available machines), installing new products from the product catalog (e.g., installing DB2 Database), configuring new managed middleware resources (e.g., creating a new *DB2 JDBC Provider* on a WebSphere Application Server), or configuring a new relationship between existing resources (e.g., adding existing JBoss application server to existing cluster). Note, that we only consider actions that *add* new resources or resource relationships. Reconfiguration actions that *remove* resources (or relationships) are beyond the scope of this paper.

There are two challenges in modeling reconfiguration actions. First, presence of certain resource types in a certain configuration state may be a precondition for the execution of a reconfiguration action. Second, the result of these actions may imply addition of a connected set of units and multiple relationships between new units and existing ones. To model both action preconditions and its effects we introduce the notion of a *Parameterized Reconfiguration Pattern*.

Parameterized Reconfiguration Pattern is a topology (consisting of virtual and concrete units) where some of the attributes on concrete units may be designated as *installation parameters*. Concrete units correspond to the affect of the action; namely, resources that will be added (provisioned) as a result of the action execution. Virtual units correspond to the pre-conditions for executing the action; for example, resources on which the new resources will be installed or created. *Installation parameters* correspond to values that are received from the user at installation time (e.g., name to be used for a newly created Database).

An *Instantiated Reconfiguration Pattern* $P(I)$ is a pattern P and a set of input parameters I, where values from I are assigned to all *installation parameters* (note that default values may be used while still allowing downstream changes at the actual time of provisioning). A *bounded* re-configuration action w.r.t. a

topology T is a triple $\delta = (P, I, R)$, where P is the pattern topology associated with the action, I is the set of input installation parameters, and R is a realization function from P to T. A bounded action is *valid* iff R is a valid and complete realization. The *effect* of applying a valid bounded action $\delta = (P, I, R)$ in a target topology T is the folded topology $\delta(T) = FoldR^S(P(I) \cup T \cup R)$. Note that we do not require that the topology realization be strict; this allows us to add links between existing concrete units in the target topology that represent required resource re-configuration. Figure 4 shows an example of a valid application of a reconfiguration action, corresponding to the installation of a DB2 Database on a Windows operation system. In this rather simplified example of product installation, pattern topology P consists of two concrete units (*DB2System* and *DB2Instance*), which will be added to the topology. The set of input installation parameters I consists of DB2 *InstanceName* and *InstallDir*. The database is installed on a Windows operating system (represented as a virtual unit) and requires a 'db2admin' OS user, which is modeled as a virtual unit upon which the *DB2Instance* unit depends.

The input to the *Relaxed PRIT Problem* is a pattern P, a target topology T, and a set of allowable reconfiguration patterns $\Gamma = \{P_i | i = 1 \ldots M\}$. To find a realization of P, it may be necessary to first apply a *sequence* of reconfiguration actions. Let $\Delta = \{\delta_1, \delta_2 \ldots \delta_n\}$ be a sequence of bounded reconfiguration actions, such that $\delta_1 = (P_{i_1}, I_1, R_1)$ is a valid bounded reconfiguration action applied to T, and for $k = 2, \ldots n$, $\delta_k = (P_{i_k}, I_k, R_k)$ is a valid bounded reconfiguration action applied to topology $\delta_{k-1}(\ldots \delta_2(\delta_1(T)))$. We also define $\Delta(T) = \delta_n(\ldots \delta_2(\delta_1(T)))$.

The Relaxed PRIT Problem. Given a pattern topology P, a target topology T, and a set of allowable reconfiguration patterns Γ, produce a valid reconfiguration sequence Δ and a topology $R\text{-}PRIT^S = P \cup \Delta(T) \cup R$, such that R is a valid strict topology realization of P in $\Delta(T)$. ■

To conclude this section, we discuss the relationship between the three variants of the pattern realization problem defined in this section (*PR*, *Strict PRIT*, and, *Relaxed PRIT*). *Strict PRIT* is a more strict version of the *PR* problem, i.e., every $(input, solution)$ pair of the *Strict PRIT* problem is also an $(input, solution)$ pair of the *PR* problem. *Strict PRIT* is also a more strict version of the *Relaxed PRIT* problem. The *Relaxed PRIT* problem is parameterized by the set Γ of allowable reconfiguration actions. If $\Gamma = \emptyset$ then *Relaxed PRIT* becomes equivalent to the *Strict PRIT* problem. If the only allowed reconfiguration action in the *Relaxed PRIT* problem is link creation between existing units, then it is equivalent to a modification of the *PR* problem that requires valid (not only weakly valid) topology realization.

4 Algorithms for Automatic Pattern Realization

Our approach to the problem of automatic pattern realization is based on the observation that it is reducible (with some variations) to the subgraph isomorphism problem [9], where realization links represent the isomorphism mapping. We view deployment topologies as *labeled graphs* (augmented with local and

Table 1. Algorithm for the Strict PRIT problem

```
[01] FindStrictPRIT(Topology P, Topology T, Map R) {
[02]     if (P_R = units(P)) return R;      // all pattern units realized
[03]     select unit u ∈ units(P) − P_R;    // select unrealized pattern unit (heuristic)
[04]     for (unit v ∈ units(T)) {          // iterate over all target units
[05]         if (not(validR(u, v))) continue to next target unit; // locally invalid realization
[06]         for (non-constraint link l = (u, r, u', t) ∈ links(P))
[07]             if ((u' ∈ P_R) ∧ (l' = (v, map^R_req(r), R(u'), t) ∉ links(T)))
[08]                 continue to next target unit; // target unit not linked to previous choices
[09]         for (non-constraint link l = (u', r, u, t) ∈ links(P))
[10]             if ((u' ∈ P_R) ∧ (l' = (R(u'), map^R_req(r), v, t) ∉ links(T)))
[11]                 continue to next target unit; // target unit not linked to previous choices
[12]         let T' = (T − {v}) ∪ {FoldR^S(u → v)}
[13]                  ∪ {constraint links cl = (u', u) ∨ cl = (u, u') ∈ links(P) | u' ∈ P_R};
[14]         if (T' is a valid topology) { // no constraints violated
[15]             let R' = FindStrictPRIT(P, T', R ∪ (u → v));
[16]             if (R' ≠ ∅) return R'; // all remaining realizations found.
[17]         }
[18]     }
[19]     return ∅; // no realizations found (backtrack)
[20] }
```

structural constraints), and infrastructure reconfiguration actions as *graph edit operations* [10]. We propose to use modified *graph matching* algorithms for the Strict PRIT problem and *error-correcting graph matching* algorithms [10] for the Relaxed PRIT problem.

There are certain properties of automatic pattern realization that differentiate it from classic graph isomorphism. First, two virtual units can potentially be realized by (mapped to) a single concrete unit. Second, local and structural constraints defined in the pattern topology need to be satisfied in the target topology. In this section we present the algorithm for the *Strict PRIT* problem. Algorithms and heuristics for the *Relaxed PRIT* problem are deferred for future publications.

Algorithm for the Strict PRIT Problem. Our algorithm for *Strict PRIT* is based on the classic depth-first backtracking search subgraph isomorphism algorithm [10], modified to account for the properties of SOA pattern realization not exhibited in classic graph isomorphism. In Section 5, we analyze the complexity of the algorithm, and present its performance evaluation.

We use the following notations in describing our algorithm. R is the realization mapping of units in pattern P to units in target T, computed so far. P_R denotes the set of units in P that are already mapped by the realization. When a virtual unit u is realized by a concrete unit v, $FoldR^S(u \rightarrow v)$ denotes the unit v with constraints defined on the unit u folded onto it (this includes constraints defined in requirements contained by u folded on to the corresponding requirements in v). Table 1 presents the algorithm in pseudo code. Function *FindStrictPRIT* should be invoked with arguments {pattern topology P, target topology T, $\emptyset$}. At each step of the recursive iteration the following is true for the arguments of the recursive call: P is the pattern topology (unmodified), T is the folded topology $FoldR^S(P \cup T \cup R)$. The output of the function is a valid strict realization mapping of units in P to units in T, if one exists, or the empty set otherwise. Note that this algorithm finds the first such realization mapping, if it exists. In

our deployment modeling prototype, we have also implemented a variation of the algorithm that finds *all* such realization mappings.

The algorithm works by mapping (virtual) units in pattern topology P to (concrete) units in target topology T, one by one. At each iteration, the algorithm selects the next unmapped unit u in pattern P (line 3) and attempts to find a realization mapping for it against all the target units (unit v in topology T, line 4). For this, it checks that the realization is locally valid (line 5), and that for every non-constraint link in the pattern with u as source or target, and the other endpoint already realized, there exists a corresponding link in the target topology (lines 6–11). Note, that potentially several units in pattern topology P can be mapped to the same unit in target topology T. If such a unit v is found, the algorithm folds constraint links and constraints (those that now can be folded due to the newly computed unit realization $u \rightarrow v$) from pattern P onto target topology T (lines 12–13). The modified topology T' is then validated to check if the realization satisfies structural constraints (line 14). Incremental validation techniques can be applied to enforce only the constraints affected. If all constraints are *valid* or *undefined*, the unit pair $(u \rightarrow v)$ is added to the realization mapping R, and the algorithm recurses to map the next unit in the pattern topology (line 15). If no further unit mapping is possible, the algorithm backtracks (line 19).

The incremental folding of constraints by the above algorithm can interact negatively with constraints that reason about the presence of other constraints. The above algorithm assumes that such "meta" constraints will return *unknown* when the constraint they are reasoning about is missing and could be added later in the process of realization. The performance of the algorithm will also be affected by the computational complexity of topology validation in step 14. Incremental validation techniques can be used to statically analyze declarative constraints, and monitor the access patterns of opaque constraints to reduce the number of operations per topology validation.

5 Performance Evaluation

We have implemented the *FindStrictPRIT* algorithm (Table 1) as an integral part of our deployment modeling platform [1], which has been recently released as an integral part of the IBM Rational Software Architect (RSA) tool [2]. Our implementation includes two versions of the algorithm: the *FindFirst* version finds the first realization mapping of the pattern and stops, while the *FindAll* version finds *all possible* realization mappings. *FindStrictPRIT* is a search algorithm that has, in the worst case, exponential complexity. However, we have identified that its performance greatly depends on the *heuristics* used to navigate the problem search tree. To evaluate the algorithm's performance, we used different combinations of heuristics to execute searches for pattern realization for a fixed set of patterns in target topologies of varying sizes.

Pattern Topologies. We experimented with a variety of patterns including some that are small and abstract and some that are more complex and detailed.

Table 2. Summary of Pattern Topologies

Pattern	Description (number of units in the pattern)
Pattern 1	Standalone WebSphere Application Server collocated with a DB2 Instance on an x86 server; expressed using a detailed hosting stack. (12)
Pattern 2	*Pattern 1* expressed using *Collocation* and *Deferred Hosting* constraints instead of a detailed stack. (8)
Pattern 3	WebSphere cluster containing two application server members. (3)
Pattern 4	*Pattern 3* with an additional *Anti-Collocation* constraint between the servers. (5)
Pattern 5	*Pattern 3* with additional relationships to WebSphere nodes, a nodegroup and a cell. (17)

For example, one pattern for a two-server cluster contains three units while a more detailed version contains 17 units. In addition, some patterns include structural constraints while others do not. A summary of patterns used in the experiments is given in Table 2.

Target Topologies. For our experiments, we generated models of target topologies of varying sizes, designed to approximate a typical data center in which there are a large number of servers, each hosting a particular software. Each target topology contains a variable number of servers hosting one of the following software stacks: standalone WebSphere 6.0, DB2 8.2 Server, standalone WebSphere 6.0 with DB2 8.2 Server, standalone WebSphere 6.0 with DB2 8.2 Client, Apache, and WebSphere 6.0 ND. The WebSphere 6.0 ND stack contains multiple application servers hosted on different nodes grouped in multiple clusters. Each generated topology contains a fixed proportion of each type of stack. This allows us to linearly scale the size of the target topology using a simple scaling factor. Generation of target topologies ensures at least one match of a pattern in each target topology, thus allowing us to separate the impact of failed realizations.

Search Heuristics. We applied heuristics to the following key decisions in the search: (1) selection of the *initial unit* in the pattern to realize and (2) selection of the *next unit* in the pattern to realize. We assume that given a unit from the pattern topology, the set of units in the target topology of the same type can be identified in constant time. We believe this to be a reasonable assumption for infrastructures whose resources are maintained in indexed databases. In addition to indexing by type, resources are also typically indexed by one or more key attributes. The heuristics we tested are defined in Table 3. Those appended with "-*I*" apply to the selection of the initial unit in pattern to realize, while those appended with "-*S*" apply to the selection of subsequent units to realize. Given a pattern unit and a set of equivalent candidate target units, we select the target unit in a random order.

Experiment Results. For each pattern we executed the *FindFirst* and *FindAll* versions of *FindStrictPRIT* against target topologies ranging in size from 124 units (11 servers) units to 2764 units (210 servers). In each case we measured the number of *unit-to-unit realization attempts* as a function of the size of the target topology. We report realization attempts as a platform independent measure of the algorithm's running time. Each test was repeated 100 times and the

Table 3. Summary of Heuristics

Heuristic	Definition
Fixed-I (-S)	The unit is selected in a fixed order from among all units (all units linked to the currently realized units). This heuristic allows us to minimize variance in tests of other heuristics.
Random-I (-S)	The unit is selected randomly from among all units (all units linked to the currently realized units).
StackBottom-I	The unit is selected randomly from among those at the base of hosting stacks. This heuristic tries to take advantage of the units related by hosting links.
TTypeFreq-I (-S)	The unit is selected randomly from among units (units linked to the currently realized units) with the type matching the least common type in the *target* topology. This heuristic tries to minimize the search space by starting with the least common target.
PTypeFreq-I (-S)	The unit is selected randomly from among units (units linked to the currently realized units) with the type matching the least common type in the *pattern* topology. This heuristic attempts to eliminate search combinatorics caused by common elements in the pattern by trying to select units with unique types.

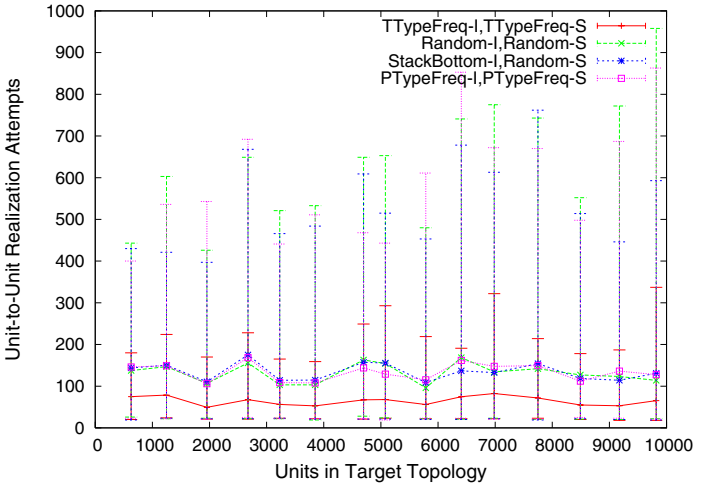

Fig. 5. Number of unit-to-unit realization attempts vs. target topology size for different heuristics, applied to *Pattern 1*

results averaged. Figure 5 shows a representative result for the *FindFirst* algorithm version, used with four representative heuristic combinations, executed on *Pattern 1*. In addition to showing averages, min and max values are shown for each tested combination of heuristics.

Figure 5 reflects what we observed in most tests: that for our combinations of patterns and target topologies, *FindFirst* completes in approximately constant time independent of target topology size. We believe that this is a consequence of the target topology indexing which, given a pattern unit type, efficiently identifies a set of potential matches in the target topology. In such a set, many of the returned candidate units are in complete pattern realizations. The first complete realization can therefore be constructed in a constant number of steps. We also observed that heuristic *TTypeFreq-I* was most effective at reducing the number of unit-to-unit realizations and the variance. For those patterns where

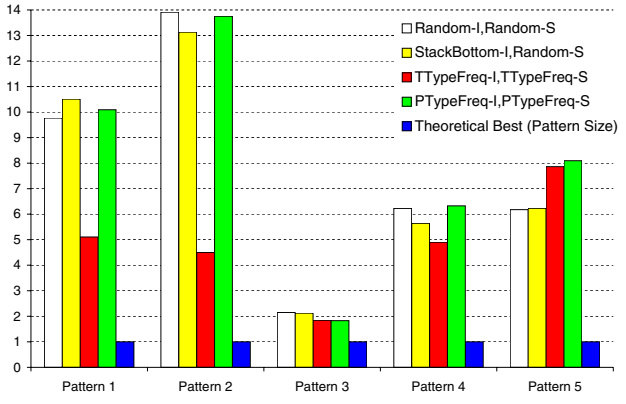

Fig. 6. Number of unit-to-unit realization attempts of different heuristics, relative to the *Theoretical Best* one, applied to different patterns

the heuristic did not provide much advantage, the frequency of the pattern unit types in the target topology was approximately the same for all units, minimizing the advantage of the heuristic.

To measure the quality of *TTypeFreq-I* as a heuristic, we compared average number of unit-to-unit realizations for each heuristic (averaged over all target topologies of different sizes) to the theoretical minimum number of unit-to-unit realizations (which is equal to the number of units in the pattern, given an oracle that always makes the right choice). Figure 6 shows the relative number of unit-to-unit realizations: a measure of 1 is the theoretical minimum number of realizations. The figure shows that of the heuristics, *TTypeFreq-I* is usually better than the other heuristics.

Due to a lack of space we do not present our results for the *FindAll* version of the algorithm. They showed that the number of unit-to-unit realizations was directly proportional to the expected number of complete pattern realizations for each of the patterns in the target topologies. Further, the *TTypeFreq-I* heuristic was, again, most effective.

6 Related Work

The idea of using *patterns (templates)* to capture important properties of a reusable service solution and to drive its deployment and configuration has been recently explored in the SOA research literature [5,6,1]. In [5], patterns describing conditions needed for the deployment of a service, were used for network level service deployment in the domain of active networks. In [6], templates were used to capture parameterized provisioning workflows, where pattern selection is identified by mapping from a Service Level Agreement (SLA). In our work [1], a pattern captures the structure and constraints of a composite solution, without bindings to specific resources, and without specifying provisioning actions.

Instead, the pattern can be used in the *pattern realization* process, which drives resource selection and necessary reconfiguration of the target infrastructure, creating a detailed infrastructure reconfiguration plan, if necessary. Such plan can then be consumed by other tools such as [13,12] for provisioning.

We use (modified) graph matching algorithms [9] for pattern realization. Graph matching has been the focus of intensive research for several decades [14,15,16,17,18,19,20,10]. One of its drawbacks is computational complexity. It is known that both subgraph isomorphism and error-correcting subgraph isomorphism problems are NP-complete [9]. The most common approach for graph matching is to directly construct graph isomorphism in a procedural manner, using *depth-first backtracking search* [14]. Several variations of this algorithm have been proposed. Some employ additional checks such as forward checking in Ullman's algorithm [15] or lookahead procedures for backtracking [18]. Others employ different heuristics for navigating the search tree to improve algorithm performance in the specific area of its application [21,20]. Our approach belongs to the latest category.

Although general graph matching algorithms are exponential, polynomial algorithms have been proposed by imposing certain restrictions on the graphs. For example, graphs with bounded *valence* can be matched in polynomial time [19]. This algorithm, however, is not applicable to the pattern realization problem, because general deployment topologies have unbounded valence (e.g., a server cluster may contain arbitrary number of servers). Moreover, this algorithm has poor performance in practice due to a large constant overhead. Unlike *precise* algorithms for graph matching that are guaranteed to find a match if one exists, *approximate* algorithms do not always find the solution but require only polynomial time [22]. Applying approximate algorithms to pattern realization may be an area of future research.

7 Conclusions and Future Work

In previous work [1], we have shown how complex SOA deployment patterns can be effectively expressed in a formal object-relationship based modeling language. We further showed how such pattern models can be efficiently validated through a folding transformation into the target objects by which they are realized. In this paper we have shown that in practice, this realization mapping can also be efficiently computed. We defined three variations of the automatic realization problem, and detailed the algorithm and performance of the Strict PRIT problem. We then presented experimental results of its behavior, identifying the *TTypeFreq-I* heuristic as the most effective. The algorithm and heuristic have been incorporated into our model-driven deployment platform, which has been released as a part of the IBM Rational Software Architect (RSA) tool [2]. In addition to resource selection, automatic pattern realization is being used to assist in operation modeling [23]. We are in the process of extending our implementation to support *Relaxed PRIT*. In future SOA deployment patterns research we plan on investigating interactive pattern realization, reverse pattern discovery, pattern composition, and pattern maintenance.

Acknowledgements

The authors would like to thank Daniel Berg, Harm Sluiman, Andrew Trossman, Michael Elder, John Pershing, and Edward Snible, for providing valuable feedback, contributing ideas, and helping to shape our vision.

References

1. Arnold, W., Eilam, T., Kalantar, M., Konstantinou, A., Totok, A.: Pattern based SOA deployment. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 1–12. Springer, Heidelberg (2007)
2. IBM: Rational Software Architect for WebSphere Software (RSA) V7.5 (September 2008)
3. Mehra, P.: Global deployment of data centers. IEEE Internet Computing 6(5) (September 2002)
4. Brown, A.B., Keller, A., Hellerstein, J.: A model of configuration complexity and its applications to a change management system. In: Integrated Management (2005)
5. Bossardt, M., Mühlemann, A., Zürcher, R., Plattner, B.: Pattern based service deployment for active networks. In: ANTA (2003)
6. Ludwig, H., Gimpel, H., Dan, A., Kearney, B.: Template based automated service provisioning supporting the agreement driven service life-cycle. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 283–295. Springer, Heidelberg (2005)
7. Redlin, C., Carlson-Neumann, K.: WebSphere Process Server and WebSphere Enterprise Service Bus deployment patterns. Technical report, IBM (November 2006)
8. IBM: WebSphere Process Server (WPS) V6.1 (2007)
9. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman and Company, New York (1979)
10. Messmer, B.T.: Efficient Graph Matching Algorithms. PhD thesis, University of Bern, Switzerland (November 1995)
11. IBM: Tivoli Provisioning Manager, TPM (2006)
12. 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: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 404–423. Springer, Heidelberg (2006)
13. Keller, A., Hellerstein, J., Wolf, J., Wu, K.L., Krishnan, V.: The CHAMPS system: change management with planning and scheduling. In: NOMS. IEEE Press, Los Alamitos (2004)
14. Corneil, D., Gotlieb, C.: An efficient algorithm for graph isomorphism. Journal of the ACM 17, 51–64 (1970)
15. Ullman, J.: An algorithm for subgraph isomorphism. Journal of the ACM 23(1), 31–42 (1976)
16. Gati, G.: Further annotated bibliography on the isomorphism disease. Journal of Graph Theory, 96–109 (1979)
17. Kitchen, L., Rosenfeld, A.: Discrete relaxation for matching relational structures. IEEE Transactions on Systems, Man, and Cybernetics 9(12), 869–874 (1979)
18. Haralick, R., Elliot, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence 14, 263–313 (1980)

19. Hoffman, C.: Group-Theoretic Algorithms and Graph Isomorphism. Springer, Heidelberg (1982)
20. Kim, W., Kak, A.: 3-D object recognition using bipartite matching embedded in discrete relaxation. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 13, 224–251 (1991)
21. Tsai, W., Fu, K.: Error-correcting isomorphisms of attributed relational graphs for pattern recognition. IEEE Trans. on Sys., Man, and Cybernetics 9, 757–768 (1979)
22. De Jong, K., Spears, W.: Using genetic algorithms to solve NP-Complete problems. In: Schaffer, J.D. (ed.) Genetic Algorithms, pp. 124–132. Morgan Kaufmann, San Francisco (1989)
23. 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 Systems Journal 45(3) (2006)