Pattern Based SOA Deployment

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. A key function of a Service Oriented Architecture is the separation between business logic and the platform of its implementation and deployment. Much of the focus in SOA research has been on service design, implementation, composition, and placement. In this paper we address the challenge of configuring the hosting infrastructure for SOA service deployment. The functional and non-functional requirements of services impose constraints on the configuration of their containers at different levels. Presently, such requirements are captured in informal documents, making service deployment a slow, expensive, and error-prone process. In this paper, we introduce a novel approach to formally capturing service deployment best-practices as model-based patterns. Deployment patterns capture the structure of a solution, without bindings to specific resource instances. They can be defined at different levels of abstraction supporting reuse, and role-based iterative refinement and composition. We show how we extended an existing model driven deployment platform to support pattern based deployment. We formally define pattern semantics, validation, and refinement. We also present an algorithm for automatically instantiating such patterns on multiple distributed service environments. Our approach has been verified in a large prototype that has been used to capture a variety of functional and non-functional deployment constraints, and demonstrate their end-to-end maintenance and realization.

1 Introduction

Much of the focus in SOA research has been on service design, implementation, composition, and placement [1]. In order to fully realize the promise of SOA, similar attention must also be paid to the deployment, configuration, and runtime management phases of the service life cycle. While SOA allows designers and programmers to access business logic independent of implementation platform, from the operator's view the situation is the extreme opposite. SOA services are typically implemented using standard distributed application platforms such as J2EE, CORBA, and .NET, and are hosted on large middleware stacks with complex configuration interdependencies. Deployment of SOA services, and the composite applications that implement them, often involves creation of operational resources such as databases, messaging queues, and topics. The runtime container of the service must then be configured to access these resources. Establishing access may require installation and configuration of client software,

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 1–12, 2007.

security credentials, as well as network-level configuration. Deployers must assure that all service resources have been correctly instantiated and configured, satisfying all functional and non-functional requirements. Cross cutting interdependencies and constraints make this a very challenging and error-prone task [2]. In addition to the communication and hosting configuration challenge, SOA poses additional challenges in transforming non-functional requirements and goals to deployment solutions including configuration for security, availability, and performance.

One approach to reducing the complexity of designing the deployment of services is to capture common deployment patterns. Such patterns describe proven solutions that exhibit certain non-functional properties. For example, experts in WebSphere Process Server (WPS) have identified 12 best practices patterns for WPS deployment [3]. Each pattern offers a different set of capabilities (high availability, scalability, security) and supports classes of applications with different characteristics. Today, these SOA deployment patterns are still captured informally in lengthy unstructured documents. Information about what combinations of products and versions are known to work must be looked up in manuals and documents libraries. Tradeoffs between cost, availability, security, scalability, and performance are investigated in an ad hoc fashion. There are no models, methodology and tools to define, reuse, assemble, customize, validate, and instantiate SOA deployment patterns.

In this paper we present a novel approach to capturing SOA deployment patterns through formal methods, models, and tools. We use and extend a model-driven deployment platform that we have previously presented in the context of middleware[4] and network[5] configuration design, validation and deployment. We present how we extend the platform with the ability to express model-based patterns representing abstract deployment topologies. These models are used by experts to capture the essential outline and requirements of a deployment solution without specific resource bindings. We formally define the semantics and validation of the realization of such abstract patterns. Using our deployment platform, we enable non-expert users to safely compose and iteratively refine such patterns to design a fully specified topology 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. We also show how automatic resource binding can be introduced to reduce the steps required to reach a valid and complete deployment topology. The desired state of the complete deployment topology can then be provisioned automatically by generating a one-time workflow as we presented in [6].

The paper is structured as follows. In Section 2, we describe our deploy platform resource model, architecture and concepts. In Section 3, we present our novel pattern modeling constructs, their semantics and validation. We also present an algorithm to automate the instantiation of patterns over an existing infrastructure. Section 4 covers related work. Finally, we conclude with a brief discussion of our prototype implementation and on-going work.

2 Deployment Platform

Our model-driven SOA deployment platform supports the construction of desired deployment state models that are complete, correct, and actionable. These models include detailed software stacks and configuration [4,5]. They are consumed by provisioning automation technologies [7] to drive automated provisioning [6]. The deployment platform is built on a core configuration meta-model, and exposes a number of services for model extension, validation, problem resolution, resource discovery, query, and provisioning.

2.1 Core Configuration Meta-model

The core model captures common aspects of deployment configuration syntax, structure and semantics. Core types are extended to capture domain-specific information. Domain objects and links are instantiated in a *Topology* which is used to design the *desired* state of the infrastructure after deployment. The *Unit* core type represents a unit of deployment. A *Topology* contains *Unit* instances that may represent resources that are already installed, or ones that are to be installed. The install *state* of a unit is a tuple $(init, desired)$ representing the install state of the unit when it was provided to the topology, and its state after publishing. The values of *init* and *desired* can be one of $\{uninstalled, installed\}$. Installable *Unit*s, may be associated with one or more *Artifacts*. A *Unit* may also represent a configuration node, such as J2EE data source, in a hierarchical structure (current or desired). A *Unit* can contain any number of *Capability* instances. Subtypes of *Capability* group domain-specific configuration attributes by function. The relationships of a *Unit* with other *Units* are expressed through containment of *Requirement* objects. The core model defines three types of relationships: *hosting*, *dependency*, and *membership*. Each *Requirement* is associated with one of these types. Relationships are represented using a *Link* association class. All these types extend a common *DeployObject* super-type. All *DeployObject* can be associated with any number of *Constraint* instances. The semantics and validation of a *Constraint* are defined by the subtypes extending it. The context of *Constraint* evaluation is the object on which it is defined. In the case of a *Constraint* contained in a *Requirement*, the constraint context is the target of the requirement link. This allows users to define constraints that must be satisfied by the resource at the other end of a relationship. Figure 1 outlines a deployment *Topology* instance model example. The topology captures the deployment of a new J2EE Enterprise Application (EAR) on an existing IBM WebSphere Application Server (WAS), using a pre-configured J2EE data-source. We have similarly defined extension schemas and instance models for a variety of other product domains and vendors.

2.2 Deploy Platform Architecture

The overall architecture of our model-based SOA deployment platform is depicted in Figure 2. At its base lies the core configuration model, on top of which

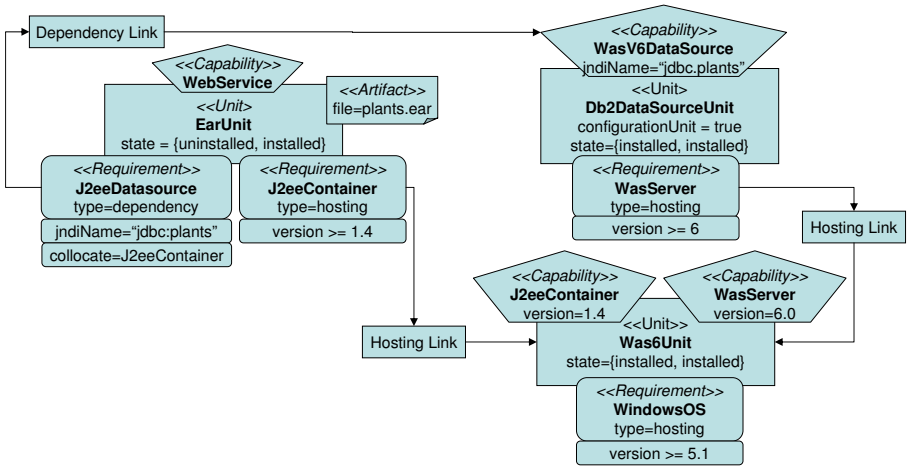

Fig. 1. Topology instance example modeling the deployment of a J2EE Application

a number of extensible services are supported. The platform defines the deployment service interfaces and provides the managers for registering extensions. The *Domain Service* is used to extend the core model types in domain-specific schemas. The *Validation Service* is used by domain authors to inject semantic validation logic. An example validation rule may express that the database name attribute of a J2EE datasource must match the actual name of the database on which it depends. The validation service invokes the validation rules when types from the domains with which they are associated are instantiated or changed. Validation rules generate semantically rich status errors markers. These error markers identify the areas in the model that violate a registered validation rule. The *Resolution Service* is used to declare logic for fixing the errors underlying the markers generated by validators. For example, in the earlier datasource database name validation example, a resolution may be declared to propagate the name of the database to all of the datasources that have a *dependency* relationship to it. For a given error status, the resolution service can be queried to provide the list of possible resolution actions. These resolutions can be invoked, either manually or programmatically, to modify the model. The core platform is packaged together with a core set of *Constraint*s, *Requirement*s, and an accompanying set of core validation and resolution rules. The *Provider Service* is used to discover and query configuration repositories (e.g. CMDBs), so that units that represent existing resources can be discovered and incorporated in deployment topologies. The *Publisher Service* is used to register provisioning agents whose role is to configure the infrastructure to match the desired state expressed in the topology. Finally, the platform supports a *Core Editor* which is a standard graphical interface for creating and editing topologies. The editor interfaces with the existing platform services for resource discovery, topology validation, error resolution, and publishing.

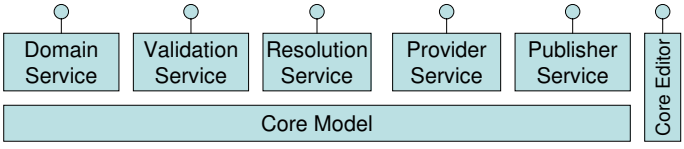

Fig. 2. Deployment Platform Extension Architecture

2.3 Valid Deployment Models

Users construct deployment models by adding or modifying units, and by executing resolution rules. The goal is to reach complete and valid deployment topologies that do not contain any error markers. A topology is validated against a set of core validation rules, a set of domain-specific type-level rules, and the constraints associated at the object instance level. The core validation rules check the cardinality of the topology links, as well as the type and configuration of their endpoints. Domain-specific validation rules can be expressed at the type level, to apply to all instances, or at the instance level as constraints. Formally, we say that a topology T is *valid w.r.t. a given set of validation rules* V iff (1) all core link validation rules are satisfied, (2) all type-level validation rules $v \in V$ evaluate to *true* on any object $u \in T$, and (3) all constraints on topology objects evaluate to *true* in the context of their evaluation. Recall that for a constraint defined in a capability the evaluation context is the capability's attribute set, while for a constraint defined in a requirement the context is the target of the relationship (and its contained capabilities). The logic for evaluating constraints is itself extensible.

3 Pattern Platform

A common requirement across different SOA deployments is the ability to describe a deployment topology at various levels of abstraction. At a base level of abstraction, the topology may represent a fully defined deployment structure that is only missing the relationships to the specific resources on to which it will be deployed. At higher levels of abstraction, the topology may partially specify the configuration of resources, focusing on key parameters and structures, while leaving others to be determined at deployment time. The deployment platform described in the previous section is well suited for modeling the *concrete* desired state of services, components, and their relationships that are directly mappable to native configuration models. In this section, we describe how we extend the deployment platform to support abstract models, termed *patterns*. As depicted in Figure 3, pattern models are defined by experts using a rich design tool and instantiated by deployers, potentially using a simple installation wizard for resource and parameter selection. First, we describe the modeling extensions, including structural constraints, virtual units, and realization links. Then we describe how we use views to execute the original set of validation rules

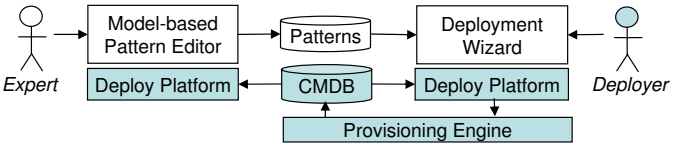

Fig. 3. Pattern use-case

on the extended class of models. Last we describe an approach for automatically realizing pattern topologies on multiple distributed environments.

3.1 Pattern Modeling Extensions

Structural Constraints. When defining patterns, it is often necessary to express structural constraints. For example, a fail over high-availability service pattern may include a structural constraint to anti-collocate the primary and standby services at the operating system level. To support structural constraints, we introduce a new *constraint* link type, and we extended it for two common types of structural constraints: collocation and deferred host.

A *collocation* constraint restricts the valid hosting of two units. It is associated with a *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 hosted on the target. For example, a deployer may wish to constrain the deployment of a service on a particular system without having to model the entire software stack. A valid topology, realizing a pattern with a deferred host constraint, must include a direct or indirect hosting link path from the source to the identified target.

Virtual Units and Realization Links. Many patterns can be expressed in the form of a model with partially specified units of abstract or non-abstract types. Our approach to presenting such patterns is through the concept of a *virtual unit*. A virtual unit is one which does not directly represent an existing or installable service, but instead must be *realized* by another unit. The *Virtual* property of a unit is a Boolean attribute on the base *Unit* type. Typically, virtual units will include capabilities with unspecified values and associated constraints. Every unit type can be instantiated as a virtual unit. A new *realizedBy* relationship can be defined between any two objects, where the source is virtual. The semantics of the relationship is that the source acts as a constraint over its target. Often, a *realizedBy* link will be defined from a virtual unit to a concrete (non-virtual) unit, although it may also target a more specific virtual unit. For simplicity, in this paper we restrict ourselves to the case where virtual units are realized only by concrete units. In cases where Unit-level realization is ambiguous in terms of the mapping of contained objects such as capabilities and requirements, additional realization links may be required between these objects. In the rest of

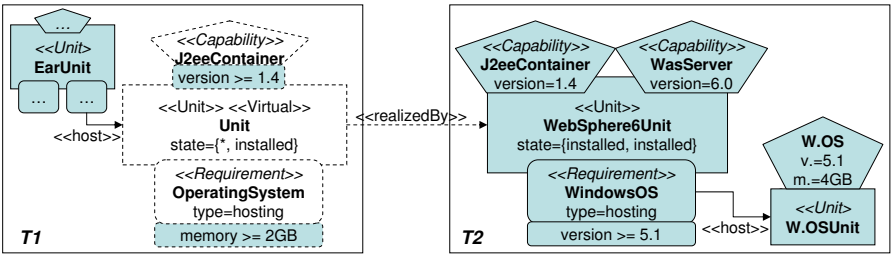

Fig. 4. Example of a virtual unit realized by a concrete unit

the paper, in order to simplify the formal definition, we assume all capabilities and requirements of realized units are also *explicitly* realized.

Figure 4 is a valid realization example which shows a virtual unit in a topology T_1 that is realized by a unit in a topology T_2 representing an installed WebSphere Application Server. Note that all constraints are satisfied by the realizing unit, and the type hierarchy is respected. The rules for locally validating a realization relationship between two units are formally defined in two stages as follows.

For any two model objects o_1, o_2, $matchR(o_1, o_2)$ iff (1) $supertype(type(o_1), type(o_2))$, (2) For every attribute $a \in attributes(type(o_1))$, $isSet(o_1, a) \rightarrow value(o_1, a) = value(o_2, a)$, and (3) For every constraint $c \in constraints(o_1), c(o_2)$.

For any two unit objects u_1, u_2, $validR(u_1, u_2)$ iff (1) $virtual(u_1)$, (2) $matchR(u_1, u_2)$, (3) For every capability $c_1 \in cap(u_1)$, there exists a unique capability $c_2 \in cap(u_2)$ s.t. $realizedBy(c_1, c_2) \wedge matchR(c_1, c_2)$, and (4) For every requirement $r_1 \in req(u_1)$, there exists a unique requirement $r_2 \in req(u_2)$ s.t. $realizedBy(r_1, r_2) \wedge matchR(r_1, (r_2))$.

3.2 Pattern Validation

By design, patterns are incomplete topologies. To meaningfully validate patterns, we have to distinguish between two sources of errors: model violations and model incompleteness. To formalize this concept we define three different validation states on attributes, relationships, or constraints, associated with virtual units in the model: $undefined$, $satisfied$, and $violated$. An element O is in an $undefined$ state in a model M if objects can be added to M, and undefined attributes set, such that O transitions to a $satisfied$ state.

The deferred host structural constraint between a source A and a target B is $undefined$ as long as the hosting stack for A is incomplete and there is no hosting link path from A to B' where $type(B') = type(B) \wedge B \neq B'$. The collocation relationship, with target type t, between units A and B is $undefined$ as long as there are no hosting link paths from A to C and from B to C' where $type(C) = type(C') = t$. A topology T is *weakly valid* iff all constraints, requirements and links associated with virtual units are in either $satisfied$ or $undefined$ states. For example, consider a pattern containing a virtual unit u with an associated hosting requirement r. If r is not linked, then the model will still be weakly valid, however if its linked to two different units it will be invalid.

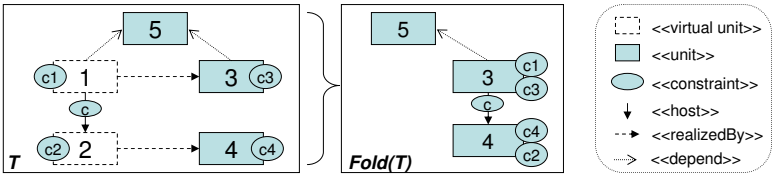

Fig. 5. Topology folding example

Topology Folding. Given a topology (or a set of topologies) with virtual units and realization links, it is not enough to locally check the validity of individual realization links using the rules defined in the previous section. For example, in Figure 4, the realization would be locally valid even if the WebSphere Application Server is hosted on an operating system with less than $2GB$ of memory. As another 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 $v' \neq v$. In this section we complete the semantic definition of patterns by defining the full set of validation and realization rules.

For this purpose, it is helpful to define the *folded topology* $foldR(T)$ of a given topology T, where, intuitively, we collapse all realized virtual units, relationships and constraints. An example of a folded topology is illustrated in Figure 5. The folded topology $foldR(T)$ satisfies the following rules: (1) For every $o \in T$, $o \in foldR(T)$ iff o is not the source of any $realizedBy$ relationship, (2) For every $o \in T$, $constraints(o)$ is the union of constraints defined on all $o' \in T$ such that $realizedBy(o', o)$, (3) For every $o_1, o_2 \in foldR(T) : \exists r$ of type t from o_1 to o_2 iff there exists a relationship $r' \in T$ s.t. $type(r') = t$, and for $o'_1 = source(r')$: $((o_1 = o'_1) \vee realizedBy(o'_1, o_1))$ (resp. $target(r')$ and o_2), and (4) For every $r \in foldR(T)$, $constraints(r)$ includes the union of constraints defined on the set of relationships $r' \in T$ as defined in item (3).

A *strict folded topology* $foldR^S(T)$ of a topology T, is $foldR(T)$ where all virtual units and their associated relationships are removed. Note that the class of strict folded topologies is identical to the class of concrete topologies defined in Section 2. Thus, for a given pattern T all of the core deploy platform validation rules can run on $foldR^S(T)$ without requiring any changes.

Topology Realization Semantics. Given our definition of a locally valid realization, and the folded view of a topology, we can now define the validity of a topology containing multiple realization links. Given a topology T, T forms a *valid topology realization* iff the following properties are satisfied: (1) Every virtual unit is realized by at most one unit, (2) Each realization link in T is locally valid, and (3) $FoldR(T)$ is weakly valid (defined earlier in this section). Note that Item (3) guarantees that links between virtual units "agree" with links between their realizing concrete units (if they don't we will get a link multiplicity constraint violation in the folded topology). A topology realization is *complete* when it is valid and all its virtual units are realized. Note that if T forms a complete topology realization then $foldR(T) = foldR^S(T)$. Now that

we extended the set of topologies defined in Section 2 to include patterns, the definition of a valid topology must be generalized, as follows. A topology T is $valid^*$ iff (1) T forms a valid and complete realization, and (2) $foldR(T)$ is valid (according to the definition in Section 2). Note that for the provisioning phase of a $valid^*$ topology T, only $foldR(T)$ is needed.

3.3 Automatic Pattern Realization

In the beginning of this section, we introduced the idea of automatically instantiating patterns in multiple environments. To do that, we have to have a way to automatically generate a $valid^*$ topology T', given an input pattern T_1 and a target topology T_2 representing the target environment. To simplify the discussion, lets assume that the inputs T_1 and T_2 are merged into one topology T. Now, T must be automatically modified by adding realization links between virtual units originating in T_1 and concrete units originating in T_2. When the modified topology T' forms a valid and complete realization, it may still be necessary to automatically execute some resolution rules to reach a $valid^*$ state. For example, values of attributes, originating in objects in T_2 may need to be propagated to units originating in T_1. An approach for automatic resolution execution was proposed in [4]. Hereafter, we limit the discussion to the automatic realization function.

Following is the formal definition of the automatic realization problem. Given a source topology T_1 and a target topology T_2, let R be a set of realization links from T_1 to T_2, and let $T' = T_1 \cup T_2 \cup R$ be the merged topology. The tuple (T_1, T_2, R) is a *maximum valid realization* iff (1) T' forms a valid realization, and (2) $|R|$ is maximum. The goal of the automatic realization problem is to find a maximum realization for given source and target topologies.

Note that a maximum valid realization may be incomplete: unrealized virtual units may exist. An incomplete topology may still be automatically completed to a $valid^*$ topology in some cases. We defer the discussion of automatic completion to future publications.

Our approach to address the automatic realization problem is based on the observation that the problem is reducible (with some variations) to the error correcting subgraph isomorphism problem [8], where realization links play the role of the isomorphism mapping. Given T_1, T_2, R, and T', as defined above, let r be the mapping function. Then we define the following changes to the original definition of the error correcting subgraph isomorphism problem. For every two units $a \in T_1$ and $b \in T_2$, $r(a) = b$ is permissible only if the following conditions are satisfied. (1) $validR(a, b)$, (2) every constraint $c \in constraints(a)$ is not in a *violated* state in $foldR(T')$, and (3) for every link $l \in T_1$ with $source(l) = a$, the corresponding link l' in $foldR(T')$ is not in a *violated* state.

Consider the example in Figure 6, where colored nodes represent concrete units. The mapping r in all of the of the topologies in the figure is a valid error correcting subgraph isomorphism mapping. However, only topologies (a) and (c) show valid mapping according to our modified definition of the problem, and according to the definition of a valid realization in the previous section. Topologies (b) and (d) violate Item (3) in the the definition above.

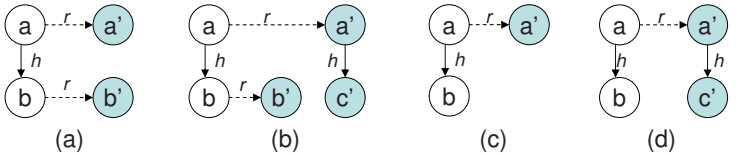

Fig. 6. Valid and invalid mappings according to our modified definition of error correcting subgraph isomorphism

Clearly, existing algorithms for error correcting subgraph isomorphism can be modified to handle the modified version defined above. The formal definition of such an algorithm is beyond the scope of the paper. We implemented and studied the performance of a simpler variant of the problem, where all virtual units must be matched. Preliminary performance results are on the practical side, although maximum subgraph isomorphism is NP-Complete. We speculate that this is because our graphs are heavily labeled, and sparse.

4 Related Work

Service deployment often refers to service selection [9] and service composition to satisfy functional and QoS requirements, for example, [10,11]. In contrast to this, our work focuses on the deployment, configuration and management of complex services including their supporting middleware. Work that addresses this depth of deployment and configuration often assumes a simplified model such as common middleware already deployed [10] or knowledge of the specific target environment so that provisioning steps are known in advance [12]. In [13] models are used to realize a conceptual service interface with one or more interfaces of its concrete implementation. The focus is on interface realization, not middleware configuration and deployment. Models are also used to capture non-functional aspects in [14] at the service design level. Such constraints can be used as input to our deployment refinement process.

The use of object-relationship models for the design and configuration of systems[15] and networks[16] has been widely adopted in industry[17,18] and their use is being standardized for service deployment models as SML[18]. [19] used a spreadsheet-style system to propagate configuration attributes over an object-relationship structure. Design tools for application deployment[5,20] have adopted Model Driven Architecture (MDA)[21] approaches.

Patterns have been used for the deployment of network services[22]. In this case, a pattern represented a detailed description of the conditions needed for the deployment of a service. While our patterns can be used as a key to find necessary conditions, they can also be used to create the necessary conditions. Patterns were used as a mechanism for service deployment in [12]. In this latter case, patterns are pre-defined and associated with concrete provisioning steps or workflows. Pattern selection is identified by mapping from a service level agreement. In our work, we divorce the pattern from the provisioning actions.

Instead, the pattern can be used to drive resource selection and to complete configuration planning, creating a detailed configuration plan. Such a plan can then be consumed by other tools such as [23,6] for provisioning.

5 Future Work

The model extensions for patterns that we have presented in this paper have been implemented in our model-driven deployment platform prototype. We are currently using this prototype to capture deployment patterns for complex domains such as WPS [3], as well as complex high-availability patterns for databases, messaging, and application servers. We have also implemented same basic structural constraints, such as collocation and deferred hosting, as well as more complex ones, such as communication. A rich visual interface supports simple model-based pattern creation and refinement. An initial implementation of the automatic realization algorithm allows users to automatically realize complex patterns over existing infrastructure resources. We plan on extending our implementation to also support installation of resources that may be missing. In future research we plan on investigating automated pattern composition, reverse pattern discovery, and pattern maintenance.

Acknowledgements

The authors would like to thank Daniel Berg, Andrew Trossman, Michael Elder, Edward Snible, and John Pershing for helping to shape our vision, contributing ideas, and assisting in implementation.

References

1. Curbera, F., Ferguson, D., Nally, M., Stockton, M.L.: Towards a programming model for service oriented computing. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 33–47. Springer, Heidelberg (2005)
2. Brown, A.B., Keller, A., Hellerstein, J.: A model of configuration complexity and its applications to a change management system. In: Integrated Management (2005)
3. Redlin, C., Carlson-Neumann, K.: Websphere process server and websphere enterprise service bus deployment patterns. Technical report, IBM (2006)
4. Eilam, T., Kalantar, M., Konstantinou, A., Pacifici, G.: Reducing the complexity of application deployment in large data centers. In: Integrated Management (2005)
5. Eilam, T., Kalantar, M., Konstantinou, A., Pacifici, G., Pershing, J., Agrawal, A.: Managing the configuration complexity of distributed applications in internet data centers. IEEE Communication Magazine 44(3), 166–177 (2006)
6. 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)
7. IBM: Tivoli Provisioning Manager (TPM) (2006)

8. 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)
9. Su, X., Rao, J.: A survey of automated web service composition methods. In: SWSWPC (2004)
10. Kichkaylo, T., Karamcheti, V.: Optimal resource-aware deployment planning for component-based distributed applications. In: HPDC, Washington, DC, USA, pp. 150–159. IEEE Computer Society Press, Los Alamitos (2004)
11. Canfora, G., Penta, M.D., Esposito, R., Perfetto, F., Villani, M.L.: Service composition (re)binding driven by application-specific QoS. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 141–152. Springer, Heidelberg (2006)
12. 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)
13. Emig, C., Krutz, K., Link, S., Momm, C., Abeck, S.: Model-driven development of SOA services. Technical report, Forschungsbericht (2007)
14. Wada, H., Suzuki, J., Oba, K.: Modeling non-functional aspects in service oriented architecture. In: IEEE Int. Conf. on Service Computing, IEEE Computer Society Press, Los Alamitos (2006)
15. Sloman, M.: Management for open distributed processing. DCS 1(9), 25–39 (1990)
16. Sengupta, S., Dupuy, A., Schwartz, J., Yemini, Y.: An Object-Oriented Model for Network Management. In: OO Databases with Applic. to CASE, Networks and VLSI CAD. Series in Data and Knowledge base systems, Prentice-Hall, Englewood Cliffs (1991)
17. DMTF: Common Information Model (CIM). Technical report, DMTF (2006)
18. W3C: Service Modeling Language, version 1.0. Technical report (2007)
19. Yemini, Y., Konstantinou, A., Florissi, D.: NESTOR: An architecture for self-management and organization. In: JSAC, vol. 18(5) (2000)
20. Microsoft: DSI: Applications of model-based management (Technical report)
21. Soley, R.: Model driven architecture. Technical report, OMG (2000)
22. Bossardt, M., Mühlemann, A., Zürcher, R., Plattner, B.: Pattern based service deployment for active networks. In: ANTA (2003)
23. 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)