Adaptive Information Technology for Service Lifecycle Management

Jerry Rolia, Guillaume Belrose, Klaus Brand, Nigel Edwards, Daniel Gmach, Sven Graupner, Johannes Kirschnick, Bryan Stephenson, Paul Vickers, Lawrence Wilcock
HP Laboratories
HPL-2008-80

Keyword(s):
Benchmarking, Business Applications, Performance Sizing, Optimization, Software as a Service.

Abstract:
The shift from enterprise computing to service-oriented and cloud computing is happening very rapidly. As the demand for higher value services increases there will be a greater need for service customization, automation for the provisioning and management of services, and the ability to offer services that satisfy non-functional requirements. This paper describes our research on a model-driven approach for packaging high value enterprise software for use as a service and on managing the service lifecycle of service instances in shared virtualized resource pools. Our approach tackles non-functional issues such as availability, security, scalability, flexibility, and performance. These are important for high value enterprise IT services. A study involving an SAP system demonstrates our progress. Our results show that a model-driven approach can be attractive to software providers that aim to support a large number of service instance.

External Posting Date: July 21, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: July 21, 2008 [Fulltext]

© Copyright 2008 Hewlett-Packard Development Company, L.P.

Adaptive Information Technology for Service Lifecycle Management

JERRY ROLIA, GUILLAUME BELROSE, KLAUS BRAND, NIGEL EDWARDS, DANIEL GMACH, SVEN GRAUPNER, JOHANNES KIRSCHNICK, BRYAN STEPHENSON, PAUL VICKERS, LAWRENCE WILCOCK

***Abstract*—The shift from enterprise computing to service-oriented and cloud computing is happening very rapidly. As the demand for higher value services increases there will be a greater need for service customization, automation for the provisioning and management of services, and the ability to offer services that satisfy non-functional requirements. This paper describes our research on a model-driven approach for packaging high value enterprise software for use as a service and on managing the service lifecycle of service instances in shared virtualized resource pools. Our approach tackles non-functional issues such as availability, security, scalability, flexibility, and performance. These are important for high value enterprise IT services. A study involving an SAP system demonstrates our progress. Our results show that a model-driven approach can be attractive to software providers that aim to support a large number of service instances**

***Index Terms*— Software as a Service, Enterprise Computing, Information Technology Management.**

I. Introduction

It is anticipated that by 2015 more than 75% of information technology (IT) infrastructure will be purchased as a service from external and internal hosting providers [1]. There are many reasons for this trend. Those businesses that need IT infrastructure can acquire it as a service quickly, with less deployment risk, lower capital expenditures, and a reduced need for certain IT skills. Service providers can offer services with greater process maturity and improved efficiency while amortizing infrastructure and labor costs across the many businesses that they serve.

There are many kinds of IT infrastructure offered as a service. Servers, storage, and networking can be offered by internal corporate IT providers or Internet service providers [2]. Email, word processing, and other simple business applications are now offered by many providers [3]. More complex business applications that implement business processes such as customer relationship management, order and invoice processing, and supply chain management are also offered as a service [4][5][6]. A service can be offered in several ways. It can be a portal that is accessed via Web browsers, a Web service endpoint, or a combination of the two.

Our focus is on high value services that implement business processes for small business through to enterprise class customers. These customers may have thousands or more employees and thousands or millions of users or Web enabled devices that interact with their service. Our project's design point is to enable the cost-effective hosting of one million high value service instances. This drives our goal to make service lifecycle management as customer driven and automated as possible.

There are several actors that participate in SaaS. *Infrastructure providers* provide the infrastructure, physical and virtual, for the operation of service instances. *Software providers* provide software that is packaged as a service. *Software vendors* create software. *Customers* contract with an infrastructure provider or software provider to consume a service. A service implements *business processes* for customers. A *Service instance* provides the service to a customer. A customer may have development, testing, and production instances of a service. The *users* of the service are employees, IT systems, Web enabled devices, or business partners of the customer. In some cases, the infrastructure provider, software provider, and software vendor are one entity.

This paper describes our research on a model-driven approach for packaging high value enterprise software as services and on managing the service lifecycle of service instances within shared virtualized resource pools. We have used SAP use cases [6] for formulating our ideas and as examples presented throughout this paper. SAP is a rich source of high value enterprise services and presents many challenges that must be addressed by software providers. Our results show that a model-driven approach can be attractive to software providers that aim to support a large number of service instances.

The remainder of the paper is organized as follows. Section II describes related work. Section III introduces our model- driven approach for service lifecycle management. Section IV explains research projects that contribute to the approach. It describes our use of models, the automation of service instance configuration and management, and support for non-functional requirements. Section V describes our current implementation with an SAP system as an example. Section VI offers summary and concluding remarks and an outline of our future work.

II. Related Work

Model-driven techniques have been considered by many researchers and exploited in real world environments [4][6]. In general, the techniques capture information in models that can be used to automatically generate code, configuration information, or changes to configuration information. The goal of model-driven approaches is to increase automation and reduce the human effort and costs needed to support IT systems. Systems can have many aspect-specific viewpoints, e.g., functionality, security, performance, conformance, each with a model. The concept of viewpoints was introduced in the ODP Reference Model for Distributed Computing [7]. Although we use different viewpoints, the concept is similar and the principle of separation of concerns is the same. Our goal is to develop and integrate models that capture multiple viewpoints to support lifecycle management for service instances.

There are several different paradigms for how service instances can be rendered into shared resource pools. These can be classified as multi-tenancy, isolated-tenancy, and hybrid-tenancy [8][9]. Multi-tenancy hosts many customers with one instance of a software service. Isolated-tenancy creates a separate service instance for each customer. A hybrid may share some portion of a service instance such as a database across many customers while maintaining isolated application servers. Multi-tenancy systems can reduce maintenance and management challenges for providers, but it can be more difficult to ensure customer specific service levels. Isolated-tenancy systems provide for greatest performance flexibility and greatest security, but present greater maintenance challenges. Hybrid-tenancy approaches have features of both approaches. We focus on the isolated and hybrid approaches for rendering service instances into shared virtualized resource pools. Guo et. al consider issues supporting the configuration and deployment of multi-tenancy service instances [10].

Rendering service instances into shared virtualized resource pools presents configuration, deployment and management challenges. Zhang et al. [11] present a policy driven approach for specifying configuration alternatives for services. Ramshaw et al. [12] explore the use of constraint satisfaction tools to specify alternative system configurations. Our approach is related to these but links configuration models with other models for deployment, run-time and change management. Industrial approaches offer visual design tools to software and infrastructure providers that help to manually specify software stack and infrastructure topology models for services and service instances [13][14]. With our goal of supporting one million high value service instances, we assume that customers specify their non-functional requirements but it is the responsibility of our framework to render to an appropriate software and infrastructure configuration for the service instance.

Cloud computing offers the integrated deployment of software, hosts, storage and networking to shared virtualized resource pools. It is being explored by several research teams [15] [18] [19]. Industrial approaches are available [2][13][16]. Our goal is to enable the deployment and run-time management of service instances to multiple cloud computing targets thereby providing more flexibility to customers. To facilitate our research we have developed our own shared virtualized resource pool based on Xen [20] and a corresponding Resource Pool Management Service.

Autonomic computing aims to reduce the human effort needed for the on-going management of service instances [21]. This includes daily, weekly, and monthly tasks such as ensuring database tables are properly sized, rolling over appropriate log files, and performing conformance tests. Our approach integrates such run-time management into the service lifecycle for service instances. This aspect of management is essential to achieve our target for hosting a very large number of service instances.

There has been recent work on automating the performance management of virtualized shared resource pools [22][23][24][25][26][27][28][25][29][30]. The techniques address issues such as capacity planning and providing performance, power, and economic based qualities of service. Our approach is to capture sufficient information in models to complement such methods and interact with them to support service lifecycle management. Our recent work on resource pool management is helping us to further understand issues that arise [27][29][30].

Finally, the model-based approach we present enables a high level of automation in service lifecycle management, but it is not an automation platform in itself. Rather, a service which is created and managed will typically leverage one or more automation platforms. We are using SmartFrog [17] to automate deployment of software and hardware in our prototype resource pool, but we could use other automation platforms instead or in addition.

III. LIFECYCLE MANAGEMENT FOR SERVICE INSTANCES

This section presents a service lifecycle management framework for service instances. Service lifecycle management governs the creation and management of service instances for customers. The framework exploits a model-driven approach with information about a service instance captured in a series of model states and separable supplemental models that describe software and infrastructure provider information. We refer to our model-driven approach as the Model Information Flow (MIF) [31] because of its focus on capturing and re-using model information to automate service lifecycle management.

The MIF links three viewpoints.

1. The configuration of service functionality offered by a software provider.
2. The configuration of software components that implement the service instance.
3. The configuration of infrastructure, virtual and physical, that hosts service instances.

The MIF enables a change in one viewpoint to be linked to changes in other viewpoints. For example it links a change in selected service functionality or non-functional requirements to necessary changes in application configuration and infrastructure topology. Conversely, model information can also be used to determine the consequences of changes to infrastructure on service instance behaviour.

The principle model of our approach is the Service Lifecycle Model. The Service Lifecycle Model encapsulates service instance specific model information and evolves through the states shown in Figure 1. The lifecycle starts on the left hand side with a catalog of services that are

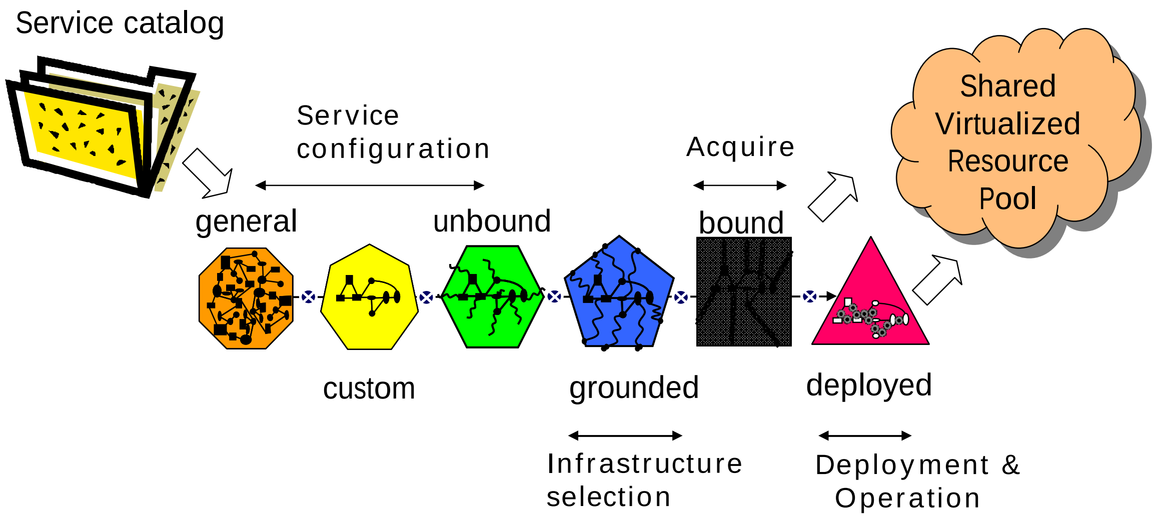


Figure 1: States of the Service Lifecycle Model

supported by our approach and ends on the right with a deployed and running system. Cycles can occur at every step, but are not shown in the figure for reasons of simplicity. The states govern service configuration, infrastructure selection, resource acquisition, deployment and run-time operation of the service instance. The following subsections describe the service catalog and the states of the Service Lifecycle Model.

A. *Service Catalogue*

A service catalogue identifies the services that can be provided using our approach. Given our context of supporting high value enterprise services for a software vendor such as SAP, each entry in the catalog describes a *service* that is a collection of related business processes. Examples of business processes include sales and delivery, and supply chain management. The description includes textual descriptions and visual notations such as BPMN [32] to illustrate the business processes. In addition, the catalogue entry specifies a tool-set that supports the creation and management of a corresponding service instance.

Once a service has been selected by the customer we use the entry in the catalogue to create a Service Lifecycle Model for the service instance. The Service Lifecycle Model can be in one of six states: general through deployed. The Service Lifecycle Model transitions between states as the tool-set operates on the service instance. The following subsections describe the model information that is captured in each state and give examples of tools that are used to support the transition between states.

B. General

This is the initial state of the Service Lifecycle Model. Once the Service Lifecycle Model data structure is prepared it is able to transition to the custom state.

C. Custom

The custom state augments the Service Lifecycle Model with functional and non-functional requirements. These requirements are collected by one or more tools in the tool-set.

A functionality configuration tool for the service lets a customer specify the subset of the service's business processes that are to be used. For example, sales and delivery may be needed but not supply chain management. Furthermore, each business process may have several business process variants, i.e., logic that handles different business circumstances. The desired set of business process variants for each chosen process must also be specified. For example, if the customer's business does not accept returned goods then a sales and delivery process variant that supports returned goods would be excluded from the service instance.

Configuration parameters are presented to the customer by the tools that reflect what can be instantiated later. Currently we offer a binary option for availability which controls whether or not a fail-over pair is created for appropriate hosts in a service instance. A fail-over pair consumes additional resources and may therefore affect cost. Similarly security is offered as a binary option in the current implementation. It controls the subnet architecture of infrastructure and whether or not firewalls are used. A scalability option determines whether a solution is deployed to a *centralized* solution with a single host or *decentralized* solution with multiple hosts.

The custom state also gathers customer performance requirements. These are specified in terms of throughput and response time goals for business process variants. The information is used by subsequent tools to support infrastructure design selection and performance sizing.

Once a customer's functional and non-functional requirements for the service are fully specified, the Service Lifecycle Model is able to transition to the unbound state.

D. Unbound

The unbound state augments the requirements for the system with information from the software vendor. Information from the software vendor includes a description of components needed to support the chosen business process variants. These may include application servers, search servers, and software code artifacts. Knowledge of which components are needed can affect the choice of infrastructure in the next state.

Software vendor information also identifies external software components that are not part of the service being deployed but that are used by the service instance. For example, an order and invoice processing business process variant may require external output management services for invoice printing and credit check services for checking financial details. A tool recognizes which external services are needed, prompts the customer to choose from a list of known service providers, and obtains any additional configuration information from the customer.

Once software vendor specific requirements are completed, the service instance has its requirements fully specified. The System Lifecycle Model is able to transition to the grounded state.

E. Grounded

The grounded state develops a complete design for the service instance. This includes the detailed infrastructure design, the mapping of software components to infrastructure components and references to configuration data required by the components. The current implementation uses three tools to refine information from the unbound state to create the design information for the grounded state.

The first tool is the Infrastructure Design Template Service. This tool uses configuration parameters and requirements information collected from the customer and software vendor in previous states to select an appropriate infrastructure design pattern from a collection of design alternatives for the service. The pattern addresses many aspects of the service instance including hardware and software deployment through to operations needed for run-time management. Once the alternative is selected, the Infrastructure Design Template Service initializes a System Template Model for the service instance and stores it in the Service Lifecycle Model. The template is made from a vocabulary of real-world concepts, such as computer system, subnet, and application server.

A System Template Model specifies ranges and default values for *performance parameters* such as the number of application servers, the amount of memory for each application server, and the number of worker processes in the application servers. Options selected by the customer such as high-availability and security are also reflected in the template, e.g., fail-over pairs and subnet architectures.

A second tool specifies the performance parameters described above. We have two implementations which perform this function, illustrating the flexibility of our approach for exploiting alternative tool-sets. The first implementation simply inspects the template for performance parameters and allows the customer to set them. The customer can set a parameter within the range specified, or a default can be selected. The second implementation is the Automated Performance Engineering (APE) Service. It exploits performance requirements and predictive performance models to automatically specify appropriate performance parameter values.

The third tool is the Template Instantiation Service. It takes as input the System Template Model and corresponding performance parameters. It outputs a System Model that becomes part of the Service Lifecycle Model. The System Model is a completed design for the service instance that is expected to satisfy non-functional requirements. Once the System Model is created, the Service Lifecycle Model is able to transition to the bound state.

F. Bound

The bound state refines the grounded state with the binding to resources, e.g., hosts, storage, and networking from a shared virtualized resource pool. A Resource Acquisition Service interacts with a Resource Pool Management Service from an infrastructure provider to acquire resource reservations according to the service instance's System Model.

In the bound state the service instance can have side-effects on other service instances. It may have locks on resources that prevent them from being used by others and it may compete for access to shared resources. Once all resources have been acquired, the Service Lifecycle Model is able to transition to the deployed state.

G. *Deployed*

The deployed state refines the bound state with information about the deployed and running components that comprise the service instance. This includes binding information to management and monitoring services in the running system. A Resource Deployment Service configures and starts the resources. A Software Deployment Service installs the software components specified in the System Model and starts the service instance so that it can be managed. The System Model includes sufficient information to ensure that components are deployed and started in the correct order. A Software Configuration Service loads service configuration data previously obtained from the customer, such as product entries to be added to a database. Finally, the service instance is made available to users.

Figure 2 illustrates a Service Lifecycle Model and its sub-models. It shows the transformation from an Infrastructure Design Template Model through to a System Model. Configuration parameter values, provided by the customer and software vendor via functional and non-functional requirements, guide how the System Template Model is rendered by the Infrastructure Design Template Service. APE can be used to decide optimal performance parameter values for the System Template Model. The Template Instantiation Service creates a System Model using the System Template Model and performance parameter values. The System Model is used to direct the subsequent acquisition of resources, deployment, and run-time operation for the service instance. Design pattern operations in the Infrastructure Design Template Model propagate through the System Template Model to the System Model.

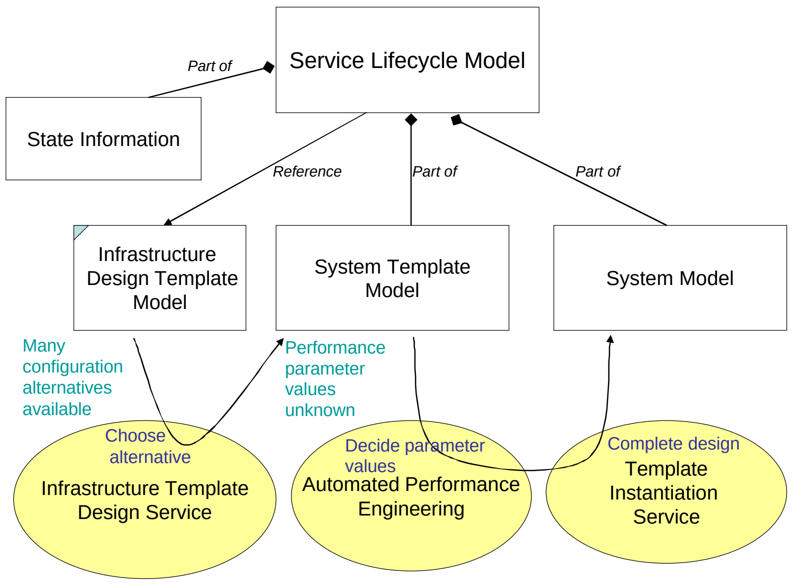

Service Lifecycle Model
Part of
State Information
Reference
Part of
Part of
Infrastructure Design Template Model
System Template Model
System Model
Many configuration alternatives available
Performance parameter values unknown
Choose alternative
Infrastructure Template Design Service
Decide parameter values
Automated Performance Engineering
Complete design
Template Instantiation Service

Figure 2: Service Lifecycle Models and Services

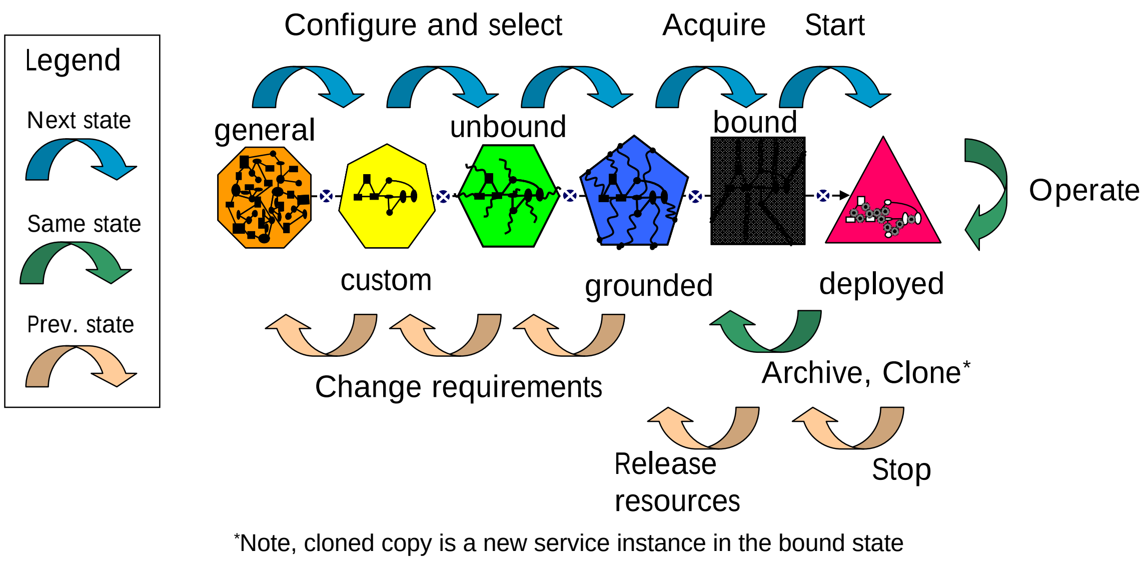


Figure 3: Lifecycle Management for a Service Instance

H. Supplemental Models and Transformations

We make extensive use of supplemental models to guide model transformations and transitions between Service Lifecycle Model states. Such models are specific to particular tools and approaches for addressing non-functional requirements and are not part of the Service Lifecycle Model. A Service Lifecycle Model only includes references to supplemental models. Section's IV.B, IV.E, and IV.F explain how supplemental models support an Infrastructure Design Template Service, a Security Service, and an Automated Performance Engineering Service, respectively.

Many of our tools require the use of heterogeneous models to implement model transformations. Section IV.A describes our work on a framework named ModelWeaver that supports the creation of such tools.

I. Service lifecycle management

Our approach supports lifecycle management for service instances. This includes service design, creation, run-time management, and change management. Figure 4 illustrates these four aspects of service lifecycle management and their relationship to Service Lifecycle Model states.

A Service Lifecycle Model can be in only one state at a time. Tools transition a Service Lifecycle Model from the general state through to the deployed state. Back-tracking is permitted so that it is possible to explore the impact of changes to service configuration and non-functional requirements on the resulting design for the service instance.

The System Model includes a description of the operations that can be performed on a service instance for run-time management. These correspond to operations on the service instance when its Service Lifecycle Model is in the bound or deployed state. Bound operations support the acquisition of resources, archiving a service instance for later use, and cloning of a service instance. Deployed operations support the configuration and operation of a service instance, including operations to vary the number of resources. A deployed service instance can be stopped and returned to the bound state. It may then be started again to resume in the deployed state. A service instance in the bound state may transition to the grounded state. If desired, the instance's computing and or storage resources can be returned to the resource pool.

Cloning is used to create multiple instances of a service for development, testing, or production service instances. It is an operation in the bound state that creates another service instance with a Service Lifecycle Model in the bound state. The clone can then be started and run in parallel with the original instance. The clone receives a full copy of a service instance's System Lifecycle Model up to information for the grounded state. Different resource instances are acquired to provide an isolated system in the bound state.

Model transformations and operations are all implemented using a common Change Request (CR) mechanism. The CR mechanism is described in Section IV.C.

IV. Research Themes

This section describes research themes that contribute to our model-driven method for service lifecycle management.

A. *ModelWeaver*

ModelWeaver is a research platform that enables the use of information from heterogeneous models to create tools that support our approach. There are many modeling frameworks and standardization efforts for models of IT systems [34][35][36][37][38][39][40]. Each is best suited to modeling a particular aspect of systems. Each modeling tool has input and output formats described by a modeling language. ModelWeaver lets us work with many kinds of models and tools.

We require each model to have meta-data that provides information about the location of inputs, outputs and the model format. The minimal set of meta-data captured by ModelWeaver is the location of a model, e.g., a URL. Additional meta-data for a model includes the identity of tools that can provide specific operations on the model, e.g., to invoke a model editor.

ModelWeaver employs a model transformation strategy that compilers use. Information is extracted and parsed from input models in their native input formats and brought into a normalized interim format upon which transformations are performed before target models are synthesized.

Figure 4 shows RDF tokenizers [41] being used to transform model information from models *A* and *B* into their normalized formats. ModelWeaver uses RDF triples as normalized interim format. The triples are stored in an RDF Model Store [42]. An RDF synthesizer then uses appropriate subsets of the normalized information from *A* and *B* to create an output model *C*. The synthesizer is analogous to a complier's code generator.

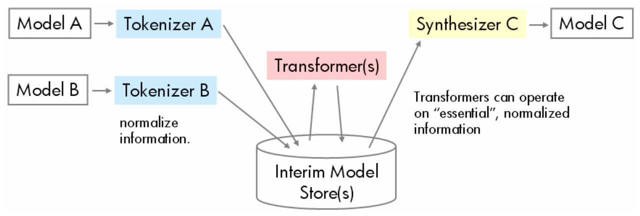


Figure 4: Weaving and Transforming Models

The use and reuse of normalized formats reduces effort needed to create new transformations. This principle avoids the quadratic growth of point transformations, requiring $O(n)$ normalizations instead of $O(n^2)$ pair-wise transformations for n formats. It also decouples transformation logic from input and output formats and minor changes to input and output formats. Other approaches [45] incur quadratic growth for pair-wise transformations.

An advantage of using RDF for the normalized format is that its offers powerful functions for implementing transformation logic. The functions include simple selection primitives, query languages such as SPARQL [43], and rule and inference engines such as the Jena Rules Engine. No other modeling framework provides such a rich tool set. RDF's human-readable text format N3 [44] also permits the definition, inspection and manipulation of models by humans. N3 is helpful for inspecting interim models when employing complex chains of tools to support a transformation. Other model transformation languages are also available [46].

Section I describes ModelWeaver's support for service customization and its orchestration of APE.

B. Infrastructure Design Template Models and the Template Instantiation Service

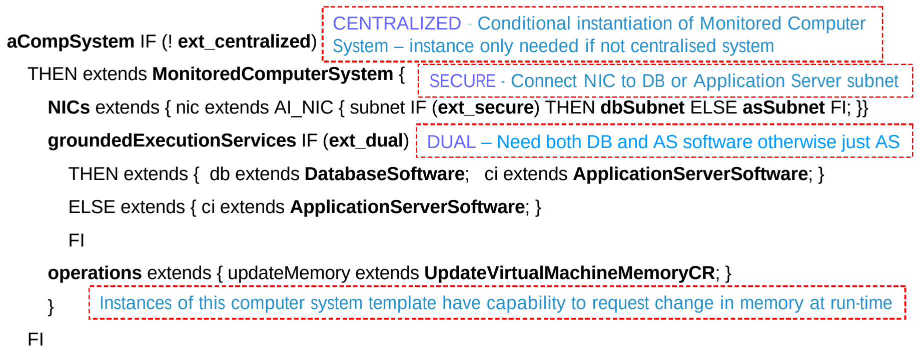
Designing and managing an IT system to support a service is a complex, error-prone activity that requires considerable human expertise, time, and expense. An important goal is to automate this process using best-in-class strategies distilled from human experts. An Infrastructure Design Template Model captures integrated best-practice design patterns for a service. It is prepared by humans and takes into account configuration options and non-functional requirements. Infrastructure Design Template Models are supplemental models.

An Infrastructure Design Template Model is made from a vocabulary of real-world concepts, such as computer system, subnet, or software service. It includes the following.

- The structure and configuration of the hardware infrastructure such as computer systems, disks, NICs, subnets, and firewalls.
- The characteristics of the required hardware are specified, such as the type, processing power and memory of a computer system, the bandwidth of a NIC, or the size or latency of a disk.
- The internal structure and configuration of the software services running on each computer system, in sufficient detail to automatically deploy, configure, and manage them; additionally, the deployment dependencies between the software services, such that they are installed, configured, started, taken on-line, taken off-line, stopped and removed in the correct order.
- The configuration of the monitoring and alarms for the hardware and software landscape.
- The set of operations, represented as Change Requests (CR), which can be applied to extend or modify the system.
- Configuration parameters and performance parameters.

An Infrastructure Design Template Model includes embedded logic that matches configuration pameters to a particular design. Configuration parameters give us the ability to encode related families of structural alternatives in a single Infrastructure Design Template Model thereby preventing an explosion in the number of instances of such models. Without this ability, a system characterized by just 7 Boolean choices would, in the worst case, require 2^7 (128) distinct Infrastructure Design Template Models that must be maintained separately. Infrastructure Design Template Models provide a powerful way to model topological alternatives – modules are only instantiated if required and relationships between modules are appropriately configured.

Infrastructure Design Template Models are expressed using the SmartFrog [17][18] textual notation as a data modeling language. The language provides typing, composition, inheritance, refinement, conditional instantiation, information hiding, and constraints, allowing us to create compact, modular, configurable descriptions of systems.



```
aCompSystem IF (! ext_centralized)
  THEN extends MonitoredComputerSystem {
    NICs extends { nic extends AI_NIC { subnet IF (ext_secure) THEN dbSubnet ELSE asSubnet FI; }}
    groundedExecutionServices IF (ext_dual)
      THEN extends {  db extends DatabaseSoftware;   ci extends ApplicationServerSoftware; }
      ELSE extends { ci extends ApplicationServerSoftware; }
      FI
    operations extends { updateMemory extends UpdateVirtualMachineMemoryCR; }
    }
  FI
```

CENTRALIZED - Conditional instantiation of Monitored Computer System – instance only needed if not centralised system
SECURE - Connect NIC to DB or Application Server subnet
DUAL – Need both DB and AS software otherwise just AS
Instances of this computer system template have capability to request change in memory at run-time

Figure 5: Fragment of Infrastructure Design Template Model Showing Reference to Template Parameters, Conditional Instantiation, and Operations

Figure 5 presents an Infrastructure Design Template Model fragment driven from three Boolean template parameters (*ext_centralized, ext_secure, and ext_dual*) that illustrates the conditional instantiation of a monitored computer system. The conditional instantiation of the computer system (*aCompSystem*) is controlled by the variable *ext_centralized*. Conditional reconfiguration of software running on it (*groundedExecutionServices*) is controlled by the variable (*ext_dual*), and the networking topology (*NICs*) is controlled by the variable *ext_secure*. Also note that the template fragment defines the set of allowed CRs as prototype *operations*. The allowed CRs may also depend on the configuration alternative.

We now consider the Infrastructure Design Template Service and the Template Instantiation Service. They support the creation of a System Template Model and System Model, respectively.

The Infrastructure Design Template Service loads the SmartFrog description of an Infrastructure Design Template Model. For each choice of configuration parameter values, the Infrastructure Design Template Service is able to render a corresponding System Template Model in the Eclipse Modeling Framework (EMF) modeling notation [40].

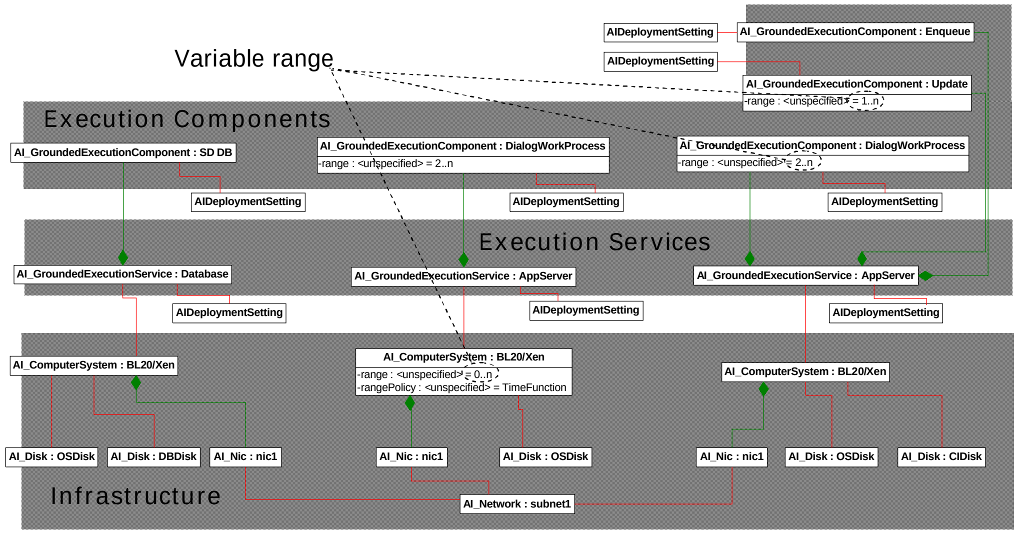

Variable range
Execution Components
Execution Services
Infrastructure
AIDeploymentSetting
AI_GroundedExecutionComponent : Enqueue
AI_GroundedExecutionComponent : Update
-range : <unspecified> = 1..n
AI_GroundedExecutionComponent : SD DB
AI_GroundedExecutionComponent : DialogWorkProcess
-range : <unspecified> = 2..n
AI_GroundedExecutionService : Database
AI_GroundedExecutionService : AppServer
AI_ComputerSystem : BL20/Xen
-range : <unspecified> = 0..n
-rangePolicy : <unspecified> = TimeFunction
AI_Disk : OSDisk
AI_Disk : DBDisk
AI_Nic : nic1
AI_Disk : CIDisk
AI_Network : subnet1

Figure 6: System Template Model for a Decentralized SAP System

Figure 6 shows a Unified Modelling Language (UML) diagram for a System Template Model for a decentralized SAP system. The template describes three types of computer system – a Database (left), a distinguished Application Server called the Central Instance (right), and additional Application Servers called Dialog Instances (middle) – and how they are connected on a subnet. For each type of computer system, the model specifies the type of software services running on it, referred to as Execution Services, the internal structure of that service such as the type of worker threads, referred to as Execution Components, and the deployment settings for the software that reference deployment instructions and parameters. The template describes the minimum, maximum and default values for modeled entities that can be replicated. The ranges for the performance parameters of these entities are encircled. Either a human or a service such as APE must decide specific values for performance parameters.

The Template Instantiation Service transforms a System Template Model with specific values for performance parameters into a System Model. The System Model has a separate object for each replicated instance of an entity whereas the System Template Model has only one instance with a range. This supports further management for each replicated instance.

C. Change Request Framework

This section describes our work on a Change Request (CR) framework that enables the planning, submission, and execution of CRs. CRs can cause updates to models and run-time and change management for service instances.

Change requests are declarative, they state what needs to be accomplished, but leave out the details of how the modifications should be carried out. CR state includes the following.

- A *requestID* that identifies the task to execute, e.g., create, clone, migrate, and stop.
- A *requestVersion* identifies the implementation version.
- The *context* describes the model entity against which the change request is submitted. The context can be the whole model, or particular entities within the model such as elements corresponding to software components or infrastructure nodes.

- *parameters*: primitive types or reference to any model entities.
- *pre-conditions and post-conditions*: logical conditions that must be true prior/after the execution of a CR along with an implementation that evaluates the conditions.
- *subtasks*: contains optional refinements of the change request into finer grain steps which are also CRs. Steps can execute in sequence or in parallel as defined by an *ordering* field.
- *dependencies:* an optional set of references to external CRs that must complete before the change request can be processed.

The lifecycle of a CR is described as follows. A submission tool creates a CR and links it to the model entity it will operate on. First, a *static* validation takes place. Since the model entity contains only the set of CRs it allows, the validity of the request can be verified prior to submission. Assuming that the CR is valid, its current state is persisted in the model and passed to a CR Orchestrator that initiates processing.

The Orchestrator is a core backend service that coordinates tools and causes the execution of CRs. Tools register with the Orchestrator to specify the request and model entity types they can support. For example, a virtual machine management tool registers that it supports migrate CRs on model entities of type virtual machine. Given a request to execute, the Orchestrator looks at its request ID and the model entity against which the request is submitted and finds the appropriate service. Each tool has a unique identifier: a URL. Assuming a tool is found and once the matching is done, the Orchestrator persists the tool identifier in the CR in order to keep track of the implementer.

The Orchestrator invokes the tool and a second round of *dynamic* checking takes place where the tool itself evaluates the CR’s pre-conditions. For example, a request to increase the memory of a virtual machine will be rejected if the specified amount exceeds the free capacity of the physical host. Assuming the CR’s pre-conditions are all validated, the tool proceeds to execute its finer grain processing steps. Once the finer grain steps are completed the tool enters a finalization processing phase where post-conditions are evaluated and current state is persisted in the model. State information captures change history for a service instance and can be used to support charge back mechanisms.

Finer grain steps for a CR are represented as a directed graph of CRs where the children of a node are subtasks, i.e., refinements, of the root CR. The graph encodes how the subtasks are ordered, and their dependencies. Whether the requests are handled in sequence or in parallel is defined by an *ordering* attribute. As an example of how these are used, in the case of SAP, the installation of a database and an application server can take place in parallel. However, strict ordering must ensure that the database is started before the application server.

The execution of a CR by a tool takes place *asynchronously* with respect to the orchestration environment. Each tool is responsible for updating and persisting progress for the run-time state of the request in the model and, in the case of failure, for being able to roll-back its changes or initiate an interaction with a human operator.

The change request framework is compatible with fully automated and partially automated management. Even though we envision that most tasks will be dealt with in automated fashion, some tasks may require human intervention. Operation prototypes for CRs enable the dynamic creation of human readable forms for CRs that permit humans to complete CRs when necessary.

At present we assume that CR planning is prescriptive. CRs are hand crafted by humans as part of the development of an Infrastructure Design Template Model. In particular, to implement each

CR they specify the sequence of tools that will be run and the parameters that are passed to each tool. In the future, we will exploit information about pre and post-conditions to enable descriptive CR subtask planning. Technologies such as model-checking [47] may be used to reason about a CR and automatically develop a plan for a CR that exploits other CRs as subtasks to implement it.

Finally, approach shares many similarities with the CHAMPS system [55]. CHAMPS deals with the planning and execution of CRs submitted by human administrators. In the context of our work, CRs are most likely submitted by automated tools.

D. State Transitions for Service Instances

In our approach, state transitions for Service Lifecycle Models are model-driven and performed by CRs. A Service Lifecycle Model is able to transition from one state to the next when post conditions and pre conditions are satisfied, respectively. The identity of the CRs and the operational parameters used to carry out state transformations are themselves stored in the Service Lifecycle Model, and depend on the type of service and customer configuration choices.

The Model Lifecycle Service (MLS) performs the lifecycle management for a Service Lifecycle Model. The MLS relies on a Model State Transition (MST) data structure that is stored in the Service Lifecycle Model. It encodes the states of the model, the allowable transitions between states, and the sequence of CRs required for each transition together with the parameters to be applied to the CR.

The MLS implements CRs of the form *ChangeModelStateTo: desiredState*, to transition a Service Lifecycle Model to its desired state. When a CR is issued to the MLS, it causes the required intermediate state transitions and actions by transforming the information in the MST to a valid sequence of child CRs. The scheme is shown in Figure 7. In this example the MLS uses the MST to determine that to move the model from State 1 to State 3 requires the following sequence of CRs: A, B, C, D, and E.

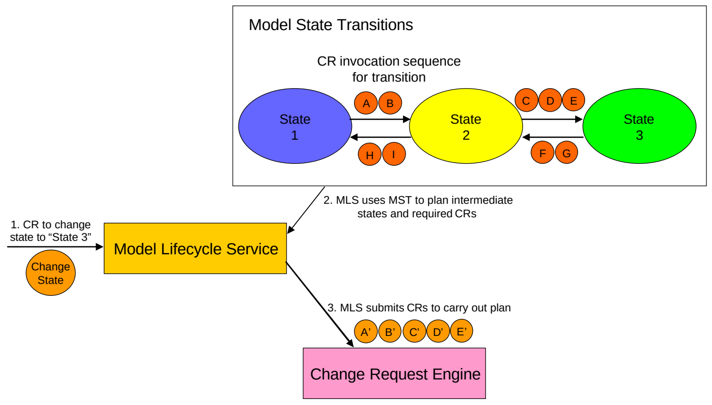

Model State Transitions
CR invocation sequence for transition
State 1
State 2
State 3
A
B
C
D
E
F
G
H
I
2. MLS uses MST to plan intermediate states and required CRs
1. CR to change state to "State 3"
Change State
Model Lifecycle Service
3. MLS submits CRs to carry out plan
A'
B'
C'
D'
E'
Change Request Engine

Figure 7: Use of MST by MLS to Determine CRs to Transition between Model States

The model-driven nature of the service lifecycle is very powerful. The sequence of allowed state changes, and the required CRs and their parameters to transition between states, can be modified at run-time by the tools invoked by CRs. Thus the behaviour of the system can be changed in response to information collected while progressing through the lifecycle. For example, if APE is required then a CR can be issued to update the MST to include CRs that cause the appropriate services to execute. In this way service lifecycle management is customized for the type of service and service configuration required by a customer.

E. Security Service

This section describes a Security Service that implements alternative security control policies for service instances. The approach integrates with the supplemental Infrastructure Design Template Models and affects the rendering of System Template Models and System Models. Customers choose from categorical security levels that are then realized by the Security Service for service instances.

The Security Service relies on the notion of compartments. Each *compartment* is used to encapsulate and protect system elements, e.g., hardware and/or software within an Infrastructure Design Template Model. Each compartment has formally modeled security controls which specify the particular protections provided by the compartment to implement security policies. The modeling approach is an extension of the observer/controller (OC) security pattern of Schöler and Müller-Schloer [48].

Our realization of the OC pattern models the entities and interactions that enforce a compartment's protection for its system elements. The models are referred to as OC models. *Observers* and *controllers* are trusted system elements referred to as *security control elements* (SCE) which provide one or more protection functions.

The OC models are designed by a security expert who determines the classes of events which must be observed and controlled to provide protection for a particular compartment and how they provide security for a service as a whole. As examples of security control policies, consider a simple example of a compartment that protects an application server with the following event classes for requests being observed and controlled.

- receiveApplicationRequest: models application traffic flowing into the compartment.
- receiveOperatorRequest: models operator traffic flowing from outside the compartment into the compartment, for example an OS login session used for system administration.
- sendMonitoringRequest: models monitoring traffic flowing from inside the compartment to a monitoring system outside the compartment, such as an SNMP trap.

The security expert may create models for several controllers to govern the receiveApplicationRequest event class. These control models are used to select and configure security mechanisms such that the controls will be enforced by the compartment's infrastructure at run-time. Though manually created, the set of control models are reused across many Service Lifecycle Model instances and many Infrastructure Design Template Models.

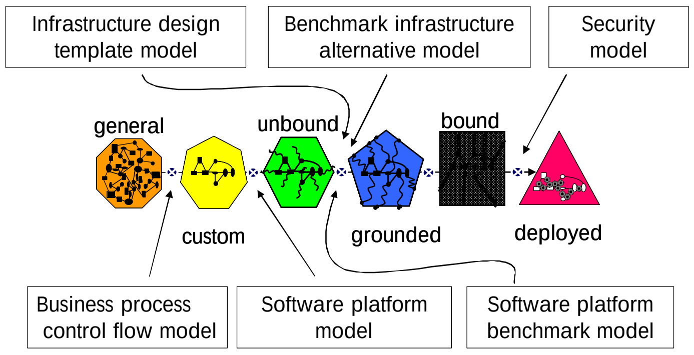

Infrastructure design template model
Benchmark infrastructure alternative model
Security model
general
unbound
bound
custom
grounded
deployed
Business process control flow model
Software platform model
Software platform benchmark model

Figure 8: Supplemental Models

Consider the following example that shows user authentication and network filtering controls for requests. At run-time, the following condition must evaluate to true in order to allow a receiveApplicationRequest and thereby permit a user access to the application server capabilities:

```
 (     ( isMemberOfSet(user, stronglyAuthenticatedUsersSet) )
    AND ( protocol == "TCP" )
    AND ( (destinationPort == "80") OR (destinationPort == "443") )
    AND ( isMemberOfSet(destinationIP, applicationServerIPAddressSet) ) )
```

All conditions needed to determine the correct action to take in response to an event class for a security configuration alternative are in an OC model for a compartment. The OC models are part of a supplementary Security Model that is referenced by the Service Lifecycle Model. This facilitates defining, locating, and programmatically evaluating and verifying the complete set of security controls for compartments and service instances. Figure 8 illustrates the supplemental Security Model.

The security expert needs to work with the designer of the Infrastructure Design Template Models to ensure any SCE and topology alternatives needed to implement security control policies are rendered in the System Template Model. The SCEs include devices such as firewalls. The management of network subnet topology is often used to decide whether certain traffic is protected by SCEs.

During the configuration activities that occur in the transformation from bound to deployed states, a Security Service tool is invoked to transform the complete set of modelled security controls from the OC compartment models in the Security Model into device-specific configurations for the service instance’s SCEs. The controls enforce the chosen security control policy. For example, if the System Model specified IPTables [49] as the controller mechanism to provide the needed network filtering function, and only two application servers at IP addresses 15.2.3.4 and 15.2.3.5 were to be used, and eth1 were the outward looking interface name, then the firewall configuration tool is run to generate an IPTables configuration file that includes four lines like the following to allow packets classified in the receiveApplicationRequest event class into the compartment:

```
iptables -A FIREWALL -i eth1 -d 15.2.3.4 -p tcp --dport 80 -j ACCEPT
iptables -A FIREWALL -i eth1 -d 15.2.3.5 -p tcp --dport 80 -j ACCEPT
iptables -A FIREWALL -i eth1 -d 15.2.3.4 -p tcp --dport 443 -j ACCEPT
iptables -A FIREWALL -i eth1 -d 15.2.3.5 -p tcp --dport 443 -j ACCEPT
```

The above configuration commands enforce the following sub-conditions at runtime:

```
( protocol == "TCP" )
AND ( (destinationPort == "80") OR (destinationPort == "443") )
AND ( isMemberOfSet(destinationIP, applicationServerIPAddressSet) )
```

With the above approach, IPTables has been configured such that if the condition evaluates to true then packets will be permitted to reach the application server. In addition, the application server must also be configured to implement the following control condition to ensure that the user specified in the request is strongly authenticated:

(isMemberOfSet(user, stronglyAuthenticatedUsersSet)).

The ability to couple controls across multiple SCE types, in this case firewalls and application servers, illustrates the advantage of the compartment concept. It also illustrates the effectiveness of the Infrastructure Design Template Models at supporting matched changes to multiple entities within a design to support non-functional requirements.

F. Automated Performance Engineering Service

The Automated Performance Engineering (APE) Service [50] has two purposes. First, it decides values for performance parameters that are needed to transform a System Template Model to a System Model. Second, it creates a performance validation test that can be executed against a deployed service instance to verify that it supports the customer's expected workloads.

APE requires the following supplemental models: Business Process Control Flow Model; Software Platform Model; Software Platform Benchmark Model; and, the Benchmark Infrastructure Alternative Model. These, in addition to the Infrastructure Design Template Model, are used to support APE and are illustrated in Figure 8.

The Business Process Control Flow Model, Software Platform Model, and Software Platform Benchmark Model are provided by a software vendor. The Benchmark Infrastructure Alternative Model is a repository of performance information.

The *Business Process Control Flow Model* describes the expected execution paths of customers as visits through business process steps. The *Software Platform Model* describes visits by these steps to the vendor's business objects. Business objects are software components such as Customer, Order, and Supplier objects that have meaning within the vendor's offering. The *Software Platform Benchmark Model* includes many software vendor benchmarks for a service. Each benchmark exercises a small number of objects in a manner typical for the platform. The benchmarks provide coverage over the software vendor's business objects.

The *Benchmark Infrastructure Alternative Model* acts as a repository of reusable performance information for APE. Benchmarks from a software platform benchmark model are run against each System Template Model and stored. The results of each run include measured resource demands.

APE creates customized performance and benchmark models that match the expected customer usage mix for business objects. The customer business object mix is determined by the throughput requirements for each business process variant as specified in the custom state and visit information from the Business Process Control Flow Model and Software Platform Model. Different business objects can cause significantly different resource usage and as a result such information must be taken into account in predictive models and benchmarks. The mix determines the average resource demand incurred by a service instance when supporting user requests.

APE uses a workload matching technique [51] that reuses a subset of results from the Benchmark Infrastructure Alternative Model to create performance models [52][53] that decide optimal performance parameter values for a System Template Model and to create the performance validation test. Figure 9 illustrates the APE process.

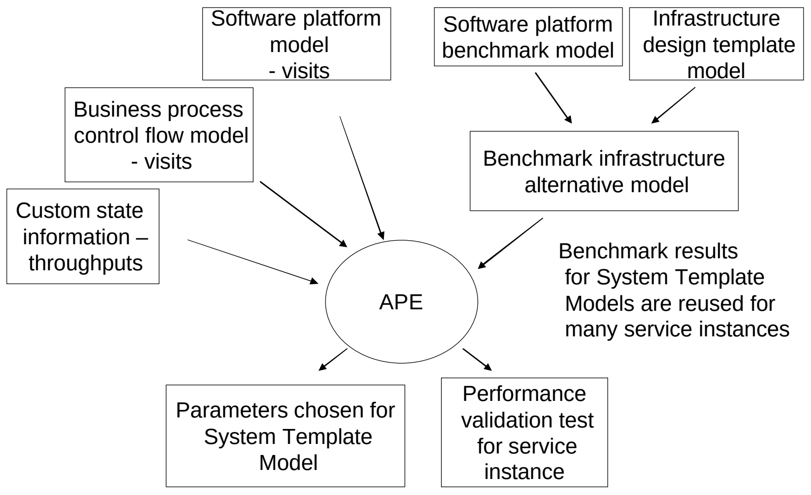

Software platform model - visits
Software platform benchmark model
Infrastructure design template model
Business process control flow model - visits
Benchmark infrastructure alternative model
Custom state information – throughputs
APE
Benchmark results for System Template Models are reused for many service instances
Parameters chosen for System Template Model
Performance validation test for service instance

Figure 9: Automated Performance Engineering Process

The Infrastructure Design Template Service can use APE repeatedly to choose the most appropriate design alternative for a service instance taking into account non-functional issues such as performance and cost. The validation test is then used to verify that a deployed service instance is able to support the customer's expected workloads while satisfying response time goals.

G. *Service instance management*

Service provisioning is a first step in the lifecycle of a service. High value enterprise services often have periodic maintenance tasks that must be performed to ensure the correct operation of a service instance [33] [54]. For example, an application with a database may have a database table that can become full and cause functional errors for the customer. If the table becomes full then either data needs to be purged from the table or the size of the tables must increase. Each Infrastructure Design Template Model specifies operations that are executed periodically to automate such *autonomic* tasks. These operations are implemented using the CR mechanism.

H. *Resource pool management*

Infrastructure and software providers amortize management, infrastructure, facility, and power costs across the customers they serve. An efficient and flexible use of the available computing resources and power is crucial to control costs.

To support our research, we developed a generic and expandable Resource Pool Management Service that integrates autonomous controllers to manage available resources. The Resource Pool Management Service supports control loops that govern resource access for customer service instances through their lifecycle. It monitors metrics for the service and infrastructure and enables controllers to automatically adjust service and infrastructure configurations when necessary. Metrics include security observations, service availability observations, and capacity and performance measurements. Controllers implement policies that are guided by the metrics. They can interact with service instances by causing operations made available by the System Models.

We are also evaluating the effects of short term versus the long term optimization of resource allocations in the resource pool and whether we can gain by combining these approaches. For this, we are exploring the integrated use of three different controllers that address capacity planning for the resource pool, virtual machine migration and the dynamic allocation of physical host resources. Simulation results have show that the combination of controllers achieves better results when managing 138 enterprise services than the controllers separately [29][30].

V. CURRENT IMPLEMENTATION

This section describes our current implementation of the service lifecycle management framework for service instances. Our approach has been to develop the framework iteratively, validating our use of multi-viewpoint models and support for design subject to non-functional requirements. The example describes the configuration and design of an SAP system.

Service creation begins with browsing the catalogue of supported services. Figure 10 shows ModelWeaver being used to browse a catalogue. ModelWeaver can be used to start an appropriate service visualization tool when an entry is selected.

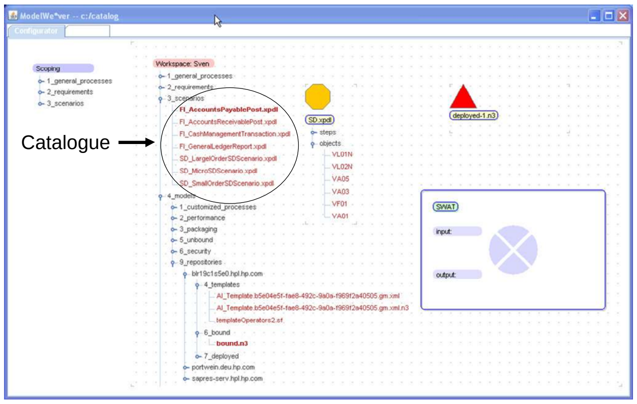
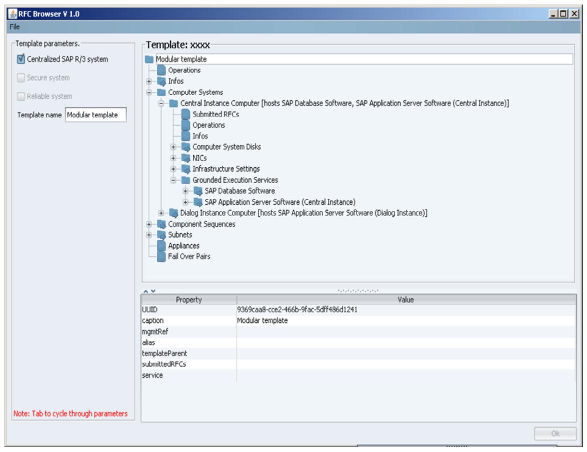


Figure 10: Browsing the Service Catalog with ModelWeaver

Once a business process is chosen from the catalogue, a Service Lifecycle Model is created for the service instance. This Service Lifecycle Model is in the general state as shown in Figure 1.

Next, a tool is presented to the customer to customize the service instance. A simple check box user interface is used to gather customer non-functional requirements regarding availability, security, and scalability. Performance requirements are captured by associating each selected business process variant with a throughput requirement, e.g., number of completions per hour and a response time goal for interactive response times. Once all the initial configuration requirements are captured from the customer, the Service Lifecycle Model transitions to the custom state.

The Service Lifecycle Model is then populated with software vendor information about requirements for software components such as application servers. These components may restrict potential designs. Once complete, the Service Lifecycle Model transitions to the unbound state.



Centralized SAP R/3 system [✓]
Secure system
Reliable system
Template name: Modular template

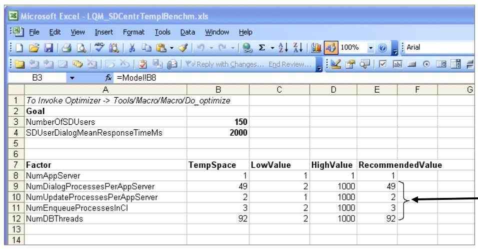
Template: xxxx

- Modular template
  - Operations
  - Infos
  - Computer Systems
    - Central Instance Computer [hosts SAP Database Software, SAP Application Server Software (Central Instance)]
      - Submitted RFCs
      - Operations
      - Infos
      - Computer System Disks
      - NICs
      - Infrastructure Settings
      - Grounded Execution Services
        - SAP Database Software
        - SAP Application Server Software (Central Instance)
    - Dialog Instance Computer [hosts SAP Application Server Software (Dialog Instance)]
  - Component Sequences
  - Subnets
  - Appliances
  - Fail Over Pairs

Property	Value
UUID	9369caa8-cce2-466b-9fac-5dff486d1241
caption	Modular template
mgmtRef	
alias	
templateParent	
submittedRFCs	
service	

Note: Tab to cycle through parameters

Figure 11: Browsing an Infrastructure Design Template



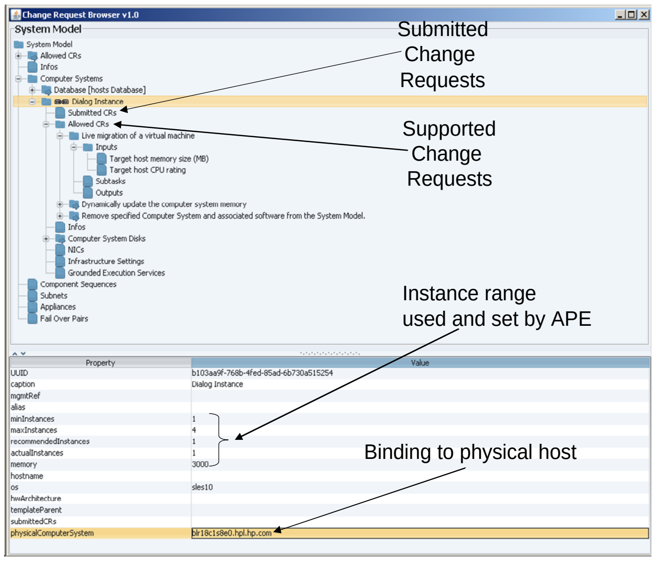
B3 =Model!B8

	A	B	C	D	E	F	G
1	*To Invoke Optimizer -> Tools/Macro/Macro/Do_optimize*						
2	**Goal**						
3	NumberOfSDUsers	**150**					
4	SDUserDialogMeanResponseTimeMs	**2000**					
5							
6							
7	**Factor**	**TempSpace**	**LowValue**	**HighValue**	**RecommendedValue**		
8	NumAppServer	1	1	1	1		
9	NumDialogProcessesPerAppServer	49	2	1000	49		
10	NumUpdateProcessesPerAppServer	2	1	1000	2		
11	NumEnqueueProcessesInCI	3	2	1000	3		
12	NumDBThreads	92	2	1000	92		
13							
14							

Figure 12: APE Performance Parameter Choices

Figure 11 shows an Infrastructure Design Template Model being browsed using the Infrastructure Design Template Service. The Infrastructure Design Template Service uses this model to render a System Template Model. The upper left hand region of the window shows the parameters that have been selected for this template, while the view on the right is the rendered System Template Model.

APE is used to compute values for the performance parameters. Inputs for APE are obtained from the custom state of the Service Lifecycle Model and supplementary models. Figure 12 shows a spreadsheet produced by APE in which the performance parameters are all computed. The model transformations for APE and the transport of computed parameter values into the Service Lifecycle Model are implemented using ModelWeaver. The performance parameters are then used by the Template Instantiation Service to transform the System Template Model into the System Model. Afterwards the Service Lifecycle Model transitions to the grounded state. After acquiring resources from the Resource Pool Management Service, the Service Lifecycle Model transitions to the bound state.

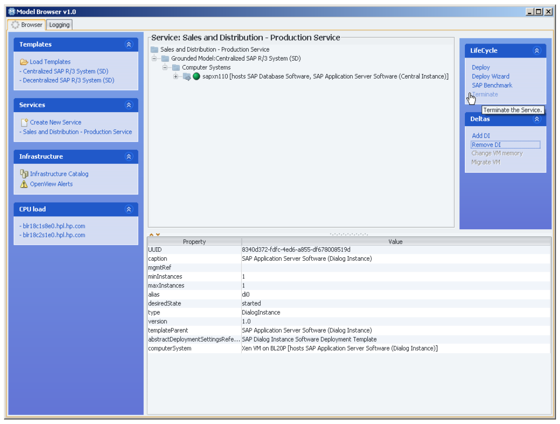


System Model

- System Model
  - Allowed CRs
  - Infos
  - Computer Systems
    - Database [hosts Database]
    - Dialog Instance
      - Submitted CRs
      - Allowed CRs
        - Live migration of a virtual machine
          - Inputs
            - Target host memory size (MB)
            - Target host CPU rating
          - Subtasks
          - Outputs
        - Dynamically update the computer system memory
        - Remove specified Computer System and associated software from the System Model.
      - Infos
      - Computer System Disks
      - NICs
      - Infrastructure Settings
      - Grounded Execution Services
  - Component Sequences
  - Subnets
  - Appliances
  - Fail Over Pairs

Property	Value
UUID	b103aa9f-768b-4fed-85ad-6b730a515254
caption	Dialog Instance
mgmtRef	
alias	
minInstances	1
maxInstances	4
recommendedInstances	1
actualInstances	1
memory	3000
hostname	
os	sles10
hwArchitecture	
templateParent	
submittedCRs	
physicalComputerSystem	blr18c1s8e0.hpl.hp.com

Figure 13: Bound State



Service: Sales and Distribution - Production Service

- Sales and Distribution - Production Service
  - Grounded Model:Centralized SAP R/3 System (SD)
    - Computer Systems
      - sapxn110 [hosts SAP Database Software, SAP Application Server Software (Central Instance)]

Property	Value
UUID	8340d372-fdfc-4ed6-a855-df678008519d
caption	SAP Application Server Software (Dialog Instance)
mgmtRef	
minInstances	1
maxInstances	1
alias	di0
desiredState	started
type	DialogInstance
version	1.0
templateParent	SAP Application Server Software (Dialog Instance)
abstractDeploymentSettingsRefe…	SAP Dialog Instance Software Deployment Template
computerSystem	Xen VM on BL20P [hosts SAP Application Server Software (Dialog Instance)]

Figure 14: Deployed State

Figure 13 uses a model browser to illustrate information in the bound state. It shows that instance range values have been set and that three CRs can be applied to the dialog instance computer system. These are labeled as Request for Change (RFC) and are for live migration, changing memory size, and removing unwanted application servers. The figure also shows the binding to a host acquired from the Resource Pool Management Service.

Finally, Figure 14 shows the service instance with its Service Lifecycle Model in the deployed state. The Infrastructure box on the left margin shows integration with management services. CR operations are illustrated in boxes in the right margin.

Our current implementation transitions a Service Lifecycle Model from the general state through to the deployed state. It assumes customers are only aware of their non-functional requirements and automatically chooses an infrastructure design based on these requirements. The design is then transitioned into an on-line system for load testing or use by users.

VI. Summary and Conclusions

We have described a model-driven approach for packaging high value enterprise software for use as a service, for managing the service lifecycle of service instances, and for interacting with shared virtualized resource pools. The framework targets the hosting of very large numbers of service instances that may operate in resource pools supported by the cloud computing paradigm. It supports the customization of service instances by customers that do not have infrastructure design skills. Finally, it addresses non-functional issues such as availability, security, and performance that are important for high value customizable service instances.

We have demonstrated the feasibility of gathering information needed for the models we have employed. The prototype tools we have developed have shown that a model-driven approach can be helpful for packaging software as a service and for automating important aspects of service provisioning and management. The approach we present is very flexible. The configuration of a service instance determines the tools used to support its service lifecycle management. Supplemental models capture service specific information. As a result, we believe the approach can be applied to many different kinds of services. Model information is re-used and shared by a variety of tools that support lifecycle management. Tools are used in combination to create powerful model transformations and state transitions. These are advantages of a model-driven approach for software providers that aim to support a large number of service instances.

Our current implementation of the framework has focused on support for SAP applications. We choose to work with SAP applications because of their high value, and because of the complex challenges they present.

Our future work includes further development of multi-viewpoint models and templates, run-time management and change request planning that better exploits the formal notions of pre and post conditions, and further work on performance prediction, autonomic management, and resource pool management. We will apply the framework to other kinds of applications to validate its effectiveness as a packaging technology for software as a service.

Acknowledgements

We gratefully acknowledge many interesting discussions with our colleagues at SAP Research, including Burkhard Neidecker-Lutz, Wolfgang Theilmann, and Phillip Robinson.

References

[1] Gartner Research, June 2007, ID Number: G00148987.

[2] Amazon Simple Storage Service, S3, www.amazon.com.

[3] Google Apps, http://www.google.com/a/enterprise/.

[4] Salesforce.com, www.salesforce.com.

[5] NetSuite, www.netsuite.com.

[6] SAP Business by Design, http://www.sap.com/solutions/sme/businessbydesign/index.epx

[7] RM-ODP, http://www.rm-odp.net/

[8] G. Carraro and F. Chong, “Software as a Service (SaaS): An Enterprise Perspective”, Microsoft Corporation, http://msdn2.microsoft.com/ , October 2006.

[9] F. Chong, G. Carraro and R. Wolter, “Multi-Tenant Data Architecture”, Microsoft Corporation, http://msdn2.microsoft.com/, 2006.

[10] C. Guo, W. Sun, Y. Huang, Z. Wang, and B. Gao, “A framework for native multi-tenancy application development and management,” *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, pp. 551-558, July 2007.

[11] K. Zhang, X. Zhang, W. Sun, L. Wei, H. Liang, Y. Huang, L. Zeng, and X. Liu “A Policy-driven approach for Software-as-Services Customization,” *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, pp.123 – 130, July 2007.

[12] Ramshaw, L. Sahai, A. Saxe, J. Singhal, S., Cauldron: a policy-based design tool, Appears in the proceedings of the Seventh IEEE International Workshop on Policy, June 2006.

[13] 3-Tera, http://www.3tera.com/

[14] CohesiveFT, http://www.cohesiveft.com/

[15] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, F. Gittler, SoftUDC: A Software-Based Data Center for Utility Computing., Computer, November 2004 (Vol. 37, No. 11), pp. 38-46.

[16] http://www.vmware.com/

[17] SmartFrog - Smart Framework for Object Groups, http://www.smartfrog.org/

[18] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and automatic ignition of distributed applications, HP 10th OpenView University Association Workshop, June 2003.

[19] http://www.ibm.com/ibm/ideasfromibm/us/google/index.shtml

[20] www.xensource.com

[21] http://autonomiccomputing.org/

[22] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak, Statistical Service Assurances for Applications in Utility Grid Environments. Performance Evaluation Journal, vol. 58, 2004.

[23] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak. A CapacityManagement Service for Resource Pools. Proc. of the 5th Intl. Workshop on Software and Performance (WOSP), Spain, 2005.

[24] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full -System Power Analysis and Modeling for Server Environments. Workshop on Modeling, Benchmarking, and Simulation (MoBS), June 2006.

[25] S. Seltzsam, D.Gmach, S. Krompass, and A. Kemper. AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications. Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE), Industrial Track, Atlanta, GA, 2006.

[26] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. To appear in Proc. of ASPLOS 2008.

[27] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper: Workload Analysis and Demand Prediction of Enterprise Data Center Applications. Proc. of the 2007 IEEE Intl. Symposium on Workload Characterization (IISWC), Boston, MA, USA, September, 2007.

[28] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Blackbox and gray-box strategies for virtual machine migration. In Proc. of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI), April, 2007.

[29] D. Gmach J. Rolia, L. Cherkasova, G. Belrose T. Turicchi, and A. Kemper, An Integrated Approach to Resource Pool Management: Policies, Efficiency and Quality Metrics, To appear in the proceedings of DSN 2008.

[30] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, 1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center. To appear in the proceedings of ICAC 2008.

[31] G. Belrose, K. Brand, N. Edwards, S. Graupner, J. Rolia, and L. Wilcock, "Business-driven IT for SAP – The model information flow," *Second IEEE/IFIP International Workshop on Business-driven IT Management (BDIM 2007)* in conjunction with *IM 2007*, Munich, Germany, pp. 45-54, May 21, 2007.

[32] Business Process Modeling Notation, http://www.bpmn.org/

[33] S. Hagemann, L. Wil. SAP R/3 System Administration. SAP Press. 2003.

[34] Object Management Group, www.omg.org

[35] Web Service Modeling Ontology, http://www.wsmo.org/

[36] OASIS, http://www.oasis-open.org/home/index.php

[37] World Wide Web Consortium, http://www.w3.org/

[38] System Modeling Language, http://www.omgsysml.org/

[39] Common Information Model, http://www.dmtf.org/standards/cim/

[40] Eclipse Modelling Framework, http://www.eclipse.org/modeling/emf/

[41] Resource Description Framework, http://www.w3.org/RDF.

[42] Jena – A Semantic Web Framework for Java, http://jena.sourceforge.net.

[43] SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query.

[44] Getting into RDF & Semantic Web using N3, http://www.w3.org/2000/10/swap/Primer.

[45] ATLAS Transformation Language, http://www.eclipse.org/m2m/atl.

[46] Model Transformation Languages, http://en.wikipedia.org/wiki/Model_Transformation_Language.

[47] Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In Proceedings of ECP-99.

[48] Thorsten Schöler and Christian Müller-Schloer, "An Observer/Controller Architecture for Adaptive Reconfigurable Stacks", ARCS 2005, pages 139-153.

[49] IPTables, http://www.netfilter.org/

[50] J. Rolia, D. Krishnamurthy, M. Xu, and S. Graupner, APE: An Automated Performance Engineering Process for Software as a Service Environments, submitted to the IEEE Transactions on Software Engineering, QEST Special Issue.

[51] D. Krishnamurthy, J. Rolia, and S. Majumdar, "A synthetic workload generation technique for stress testing session-based systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 868-882, November 2006.J.A.

[52] Rolia and K.C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689-700, August 1995.

[53] U. Herzog and J. Rolia, "Performance validation tools for software/hardware systems," *Performance Evaluation*, vol. 45, no. 2-3, pp. 125-146, 2001.

[54] T.Schneider, SAP Performance Optimization Guide, Third Edition, SAP Press.

[55] A. Keller, J. Hellerstein, J.L. Wolf, K. Wu, V. Krishnan, The CHAMPS System: Change Management with Planning and Scheduling, Proceedings of the *IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, IEEE Press, April 2004