Model-based Automation for Hardware Provisioning in IT Infrastructure

Takayuki Kuroda*
NEC Corporation
1753 Shimonumabe Nakahara-ku, Kawasaki, Kanagawa, Japan
Email: t-kuroda@ax.jp.nec.com

Aniruddha Gokhale
ISIS, Vanderbilt University
Nashville, TN 37235, USA
Email: a.gokhale@vanderbilt.edu

***Abstract*—An IT infrastructure comprises both hardware and software resources. While much progress has been made on automating the provisioning of software resources, provisioning hardware resources continues to use labor-intensive manual processes. For example, hardware provisioning tasks including most of the peripheral tasks, such as designing the system architecture, planning the tasks to deploy on it, describing the operational procedures, and managing the progress of hardware deployment, remain a manual process despite being amenable to automation. To address these problems, this paper presents a model-based approach to automating hardware provisioning. First, we describe a specification to define a hardware system in the form of a desired state model. Second, we propose a scheme to generate operational procedures to deploy the system detailing how we handle dependencies and address issues of practical importance useful to improve worker productivity. Third, through a case study involving hardware provisioning for a private cloud platform, we show the benefits accrued through our scheme, which not only addresses the issues with peripheral administrative tasks but also the practical operations by supporting improved operational procedures. Our system is implemented in Java. We qualitatively evaluate our solution alluding to how reusability can be enhanced, and also elicit its current limitations providing insights into ongoing work to address these limitations.**

I. Introduction

Two recent innovations making significant impacts on the field of enterprise system provisioning are virtualization and model-based provisioning. Model-based provisioning as described in this paper implies a series of techniques to deploy system components effectively using a model-based approach. A number of products [1], [2], [3] and standardized specifications [4] supporting model-based provisioning are increasingly being used in the enterprise world.

At the same time, virtualization has shifted the context of provisioning computational resources from hardware to software by focusing on provisioning virtual machines instead of actual hardware. Consequently, the actions to be undertaken to deploy most of the enterprise applications can now be written as a computational workflow or a collection of scripts that can be executed without involving human labor. However, the cost now shifts to creating these workflows and scripts. It is here that model-based provisioning makes the most impact. It enhances the efficiency in preparing the provisioning workflow in terms of ease, quality and the speed at which the workflows can be prepared.

* Work done by the author as a visiting researcher at Vanderbilt University

The key contributions of model-based provisioning include (a) increasing the reusability of workflows due to the well-organized semantics to define components containing small pieces of workflow and their composites, and (b) supporting an intuitive approach to express a system that is to be deployed as a "desired state" model which does not require knowledge of the resource model to develop the operation logic in the form of scripts or workflows [5].

Virtualization and model-based provisioning are in some sense symbiotic thereby helping each other advance their individual state-of-the-art. For example, as model-based provisioning techniques mature, more varieties of system resources, which have hitherto not been virtualized, are being virtualized. Software-defined data center is an example of such a trend [6]. In turn, this will require improvements to model-based techniques.

However, there still exist many types of equipment used in the enterprise IT world that are not yet virtualized and hence continue to use labor-intensive processes. Note that virtualization does not preclude the need for physical machines and networks, such as the computational resources in a data center and in a private cloud, or personal IT environments in an office, for example. Moreover, many emerging services beyond traditional web services involve specialized equipment, such as client terminals, cameras, sensors, and alarms. A system which consists of such resources is still dealt in conventional ways. System vendors or in-house administrators who manage these IT resources continue to suffer from the tedious and error-prone work of provisioning these resources. In fact, their responsibilities include not only the practical operations to deploy the systems but also many peripheral works, such as designing the architecture, planning the tasks to deploy, describing operational procedural documents, assigning workers and managing the progress of practical operations. In particular, planning deployment tasks and describing operational procedures are time consuming activities for administrators. Moreover, the complex documents written by hand are hard to understand for workers and may be subject to multiple interpretations due to ambiguities in the natural language description.

Thus, there is a compelling need to adopt model-based automation for hardware provisioning. Our focus for this paper is to address this key requirement and provide an approach to automate most of the administrative tasks in hardware provisioning and support practical operations in preparing quality and advanced operational procedures. We present the concept of model-based hardware provisioning and propose a scheme to realize the concept and specification to define the models.

The rest of this paper is organized as follows: In Section II we provide a motivating example to elicit the challenges in model-based provisioning; we present the concept of model-based hardware provisioning, architecture overview, model specification and implementation of a system to realize the concept in Section III; We evaluate our approach in the context of a case study in Section IV; We discuss additional topics and future work in this field in Section V; Related research is compared with our approach in Section VI; and finally we offer concluding remarks in Section VII.

II. Background and Challenges

This section provides some background on how hardware provisioning is done and describes the challenges in realizing a model-based hardware provisioning capability. To better elicit the challenges, we use a motivating example.

A. Terminology

As mentioned before, prior to the advent of virtualization technology, the provisioning of system components such as servers, and even middleware was done manually. There are two typical roles for this kind of work: an integrator and a worker. The integrator represents an administrative role whose responsibilities are recognizing the customer's requirements, designing the system architecture, planning the tasks to deploy, describing the operational procedural documents, assigning workers, ordering all the component parts, and managing the overall progress of the deployment. The worker role represents the human asset that actually performs the deployment tasks according to the procedural documents prepared by the integrator.

B. Motivating Example

Consider the need for provisioning hardware to create a private cloud platform. At a minimum, the provisioning consists of installing one or more server racks, multiple servers on that rack, network switches, and network cables. The servers and the switches are inserted into the rack and every server and switch are connected with cables in the final state that is desired. We assume that every component is shipped from a vendor as a product so that every item is individually packaged to begin with, which we call as the initial state. The workers are responsible for transforming the system from the initial state (*i.e.*, all individual components in their packages) to the final state (*i.e.*, a functional private cloud).

C. Challenges

There are multiple challenges in provisioning the hardware in our motivating example. For instance, the tasks should be performed in a proper order to complete them efficiently without failure. Proper order is important because otherwise some steps may have to be rolled back in order to address a step that was expected to be done before some others. In this example, this means unpacking and setting up each component individually in isolation before combining them. The cables should be connected after both servers and switches are inserted into the rack. Furthermore, there can be many conditional tasks even in such a simple example. The actual products used for each component and the number of servers should be selectable and configurable.

With this variability, it is possible that each product can have slightly different setup tasks. Beyond this, it is possible that for specific configurations, additional components may be needed. For example, in our motivating example, one server may need to have an extra network port to connect with an external network in order for it to work as a gateway. If a selected server product for the gateway does not have enough network ports, then an extra network interface card should be added before the server is inserted into the rack. Satisfying these interdependencies is critical for successful provisioning.

From the perspective of the operational procedure, an integrator must be able to unambiguously capture this information. However, documents expressed in natural language are often prone to multiple interpretations. Moreover, the workers have to read and understand an operational procedure.

The next issue is that of reuse. Note that the task to setup the same component will not vary significantly from configuration to configuration but reusing and automating this capability has been hard thus far since the actual products adopted and their configurations can have slight variations in each case. To improve reuse, manually produced operational procedures may be kept in separate individual documents with one document referring to another as part of the operational procedure or may include many conditional sections to cover slight variations.

Such document structures, however, reduce maintainability and readability of the documents. Consequently, it diminishes the productivity of workers who must interpret these documents to take the concrete action. Moreover, workers are required to have enough knowledge about the provisioning of each system domain making the individual an irreplaceable asset of the organization, which may become problematic if the individual leaves the organization.

The above-mentioned problems, such as complexity, non-structural characteristics and slight differences between cases, have also been discussed in the software provisioning domain, and have been resolved using model-based provisioning based on standardized and well-organized semantics. Given the success of applying model-based techniques for provisioning software assets, we surmise that model-based hardware provisioning may be equally effective. Investigating the validity of this hypothesis is thus the focus of this paper.

III. Model-based Hardware Provisioning

This section delves into the details of the model-based hardware provisioning approach we have developed.

A. Solution Overview

The concept of model-based hardware provisioning is to generate a set of tasks to deploy a hardware system from the model that describes the operational procedure. The model expresses the desired state model that is editable by integrators, and the generated tasks are in natural language so as to be readable by workers. We define well-organized semantics to describe a hardware system that allow composing components. Each component includes its internal structure comprising parts, their states and tasks to deploy the component in the system. The structure enables the component to be combined with other components, the states allow an integrator to define its desired state, and tasks generate the overall operational procedure in natural language according to the assembled system definition.

Defining the system in the form of a desired state model is an important notion in model-based provisioning in order to generate proper workflows regardless of the states of the deployment configuration. For example, let us assume that a file named "sample.txt" should be placed in a path. When this requirement is expressed as a workflow, it might be "`cp template.txt sample.txt`" (where "cp" is the copy command). This workflow can fail when it is executed more than once and the file is placed in advance. Rather, every workflow or task should result in the same end outcome regardless of the state of the system. This characteristic is called as "input insensitive" [5] or "idempotent" [1]. By expressing the provisioning requirement as a desired state model, an integrator can delegate the responsibility of the provisioning process to be "input insensitive" to the models.

Our scheme to realize this concept consists of three main functions:

- **Modeling function:** is to create models of system components which will be used to define systems by an integrator. The modeler is a role to develop the model using this function. The models created by modelers are stored in a repository. An integrator refers to the repository in order to look up the necessary models.
- **Designing function:** enables an integrator to design a system to be deployed. The integrator can customize and assemble predefined models of system components to define the system.
- **Task generation function:** generates tasks to deploy a system defined by an integrator. Each generated task has a short sentence about what should be done. The task can have a detailed description as well. Both are written in a natural language to be easily readable by workers. The tasks are presented in the proper order as part of the entire generated operational procedure.

In the rest of this section, we describe the detailed specification of the model and how tasks are generated from a system definition. We also show our current implementation of these functions.

B. Component Model

The *Component* is the primary unit to compose a system. There are two types of *components: primitive* and *composite*. A *Primitive* component (or Primitive for short) is a fundamental modeling unit which shows an actual system component operated by workers. A *Composite* component (or Composite for short) is a model of a conceptual component which is used to define a pattern of *component* compositions. Figures 1 and 2 illustrate an example of models of *primitive* and *composite* components, respectively. The details are provided below.

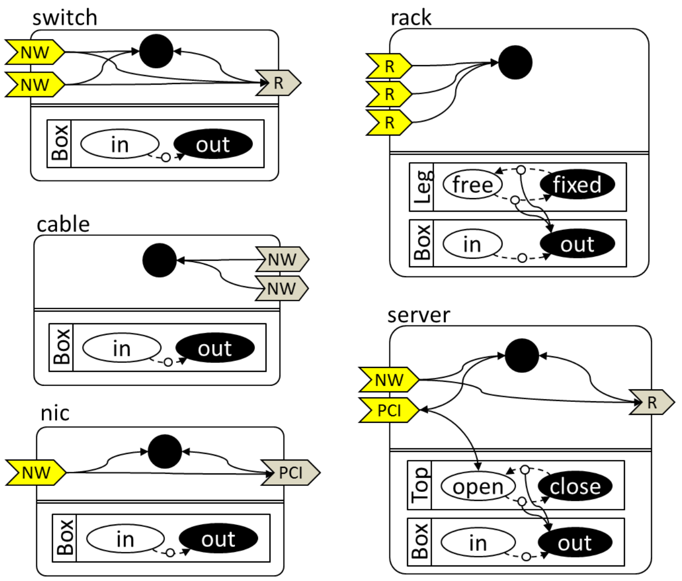

Fig. 1: An example of *primitive* models.

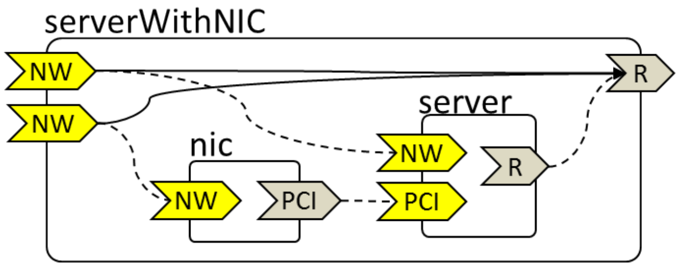

Fig. 2: An example of a *composite* model.

- ***Part* and *State*:** The *Primitive* in our modeling paradigm comprises *Parts* shown as rectangles. For example, the "rack" primitive will have parts, such as its "legs" and its packaging "box", as shown in Figure 1. The *State* shows the details of the internal conditions of the primitive in the context of that part. For instance, a switch primitive may be either "in" its packaging box or taken "out" of its package. In general, a primitive can have any number of states associated with its part. *PossibleStates* is a list of valid states that the part can be in. Each part will have an *initial state* and a *final state*, and is capable of keeping track of its *current state* during the system provisioning. When a primitive is installed, all of its parts should be in their final state. *StateShift* is another notion which shows a shift of a particular state from the set of possibleStates to another. In Figure 1, the lower half of each primitive shows its parts and their states. The name of a part is shown in the left box, and possibleStates are shown in the right box as ovals. A white (*i.e.*, unfilled) oval shows an initial state, a black (*i.e.*, filled) oval shows the final state, and a stateShift is shown as a dashed lined arrow between possibleStates.

- ***Wiring***: *Wiring* shows a connection between two components and a combining task between them. Wiring has a *wiringInterface* which shows the type of the wiring. For example, when the server component is inserted into the rack component, this connection can be modeled as a wiring which has a "RackSpace" wiringInterface. Every component has any number of *wiringPorts* which are connector ports of wirings. A wiringPort has a wiringInterface as well. There are two types of wiringPorts: *consume* and *accept*. A wiring connects a consume port with an accept port of a different component only when they share the same wiringInterface. In Figures 1 and 2, the shapes of chevron placed on the left and right of components show wiringPorts. *Consumes* are always placed on the right and *accepts* are on the left. In these figures, "R", "NW" and "PCI" in wiringPorts mean "RackSpace", "NetworkPort" and "PCIExpress" wiringInterfaces, respectively. Note that wiring is defined only in composites. In Figure 2, a dashed line drawn from "nic" to "server" shows an example of wiring.
- ***Task***: A *Task* is the action to be performed when transitioning from one state to another, *i.e.*, during a stateShift. A task comprises templates of descriptions about what should be done by workers. Blanks in the templates will be filled with information of related primitives or wirings. Basically, tasks are defined in a stateShift or a wiring so that every execution of a task makes a change in the state of a part or the wiring. For example, when "Open the top of the server" task is done by a worker, the "top" part of the "server" primitive will be changed from "closed" to "open". Alternately, when the "Insert the server into space1 of the rack" task is performed, the "rackspace" wiring will be marked as connected. Table I shows examples of task template definitions. An element type such as "Rack.Leg (free to fix)" shows a stateShift from "free" to "fix" in "Leg" part of "rack" primitive in this case. Double braces in the task template example show blanks to fill actual information.

TABLE I: Examples of *task* template definitions.

Element Type	*Task* Template Example
RackSpace wiring	Insert {{consume.id}} into {{accept.id}}
PCIExpress wiring	Insert {{consume.id}} into {{accept.id}}
NetworkPort wiring	Connect {{consume.id}} with {{accept.id}}
Rack.Leg(free to fix)	Fix the leg of {{id}} on the floor
Rack.Leg(fix to free)	Release the leg of {{id}}
Rack.Box(in to out)	Take {{id}} from its package
Server.Top(close to open)	Open the top of {{id}}
Server.Top(open to close)	Close the top of {{id}}

- ***Dependency***: *Dependency* is a directional link which suggests a proper order of task executions. A dependency from an element "x" to another element "y" means that a task related to "y" has to be done before a task related to "x" is executed. In a primitive, dependencies between wiringPorts and primitive itself will be defined. Additionally, dependencies from wiringPorts to possibleStates and from stateShifts to possibleStates will also be defined. A dependency on wiringPort requires its combining tasks, and a dependency on primitive itself requires its installation tasks which include all stateShifts to satisfy their final states. When a wiringPort depends on a possibleState, the dependency requires a stateShift to the possibleState before the wiringPort is combined. When a stateShift depends on a possibleState, the dependency requires a stateShift to the possibleState before the depending stateShift is invoked. For example, the stateShifts "Server.top(open to closed)" depend on the "box" part being in the "out" state because the top of the server cannot open or close without unboxing the server from its package. A dependency from a "PCIExpress" wiringPort to "open" of the "top" part is another example. In Figures 1 and 2, dependencies are illustrated as solid lined arrows. A black circle shows the primitive itself. When an arrow is depicted from a wiringPort to the black circle, it means that the wiring connecting to the wiringPort should be connected after the primitive itself has been setup into its final state. Naturally, such wiringPorts are not allowed to have a dependency to other state of the parts. One can also define dependencies between wiringPorts in the composite; an example is illustrated in Figure 2, however, dependencies should be defined in a primitive as far as possible because they can be reused with the primitive.
- ***Property***: Each component has a *property* which is a set of key-value pairs. It is used to feed some configuration to a component. Property is omitted in the figures to keep them simple.

Composite components define a pattern of component compositions which are connected with wirings. A composite will be used as a component again to define a system in the same manner as primitives. All composites are extracted into primitives when the tasks are generated from the system definition. To do this, the wiringPorts on a composite are associated with its inner components. We call this definition of association as *promote*, which is similar to how the SCA standard [7] uses it. In Figure 2, "RackSpace" and "NetworkPort" are promoted from "serverWithNIC" to "server" and another "NetworkPort" of "serverWithNIC" is promoted to "nic", for example.

C. Task Generation Algorithm

The task generation algorithm walks through the component model and produces the operational procedure comprising the tasks. This process eliminates the need for manual creation of the operational procedures. The process is as follows: First, all composites in a system definition are extracted into primitives and wirings connecting the primitives. The wirings connected with wiringPorts of a composite will be promoted to a corresponding wiringPort of an inner component according to the promote definition. Values in a property of a composite will be propagated if they are referred from an inner component. The dependencies defined in a composite are also delegated into corresponding inner elements.

The extracted primitives and wirings are sorted by their dependencies. The sorting is done using *topological sort* [8], which is applied to vertices in a directed graph. We need such a capability because in our case we have tasks that must be captured in the sorted order that preserve dependencies. The elements which have no dependency to others come first. Then all dependencies to the selected elements will be addressed and these dependencies are removed, and finally all those elements whose dependencies were resolved come next. Later, we repeat this operation until all elements are selected. Note that cycles in the graph will cause the sorting to fail. The selected element, namely primitive or wiring, generates tasks to be done at the turn. Which element should come first when more than one element can be selected is another problem. There should be any strategy to select proper element in terms of efficiency in practical operation. We discuss this problem in Section V.

Tasks are generated from the elements in the sorted order. A primitive generates tasks to shift all of its states to the final state. If necessary stateShift depends on other states, tasks to satisfy the depended states are also extracted as prior tasks. For example, if the "top" state of "server" primitive is required to be "open" when it is "closed" and "box" state is "in", then stateShift "Server.box(in to out)" and "Server.top(closed to open)" are extracted in this order, and tasks "Take server from its package" and "open the top of server" are generated at the end. Wiring generates tasks to connect primitives at both end of the wiring. If the wiringPorts which the wiring is connecting depend on other elements, the tasks to satisfy the dependency are extracted as prior tasks. For example, when a "PCIExpress" wiring is connected to a wiringPort of "server" and the wiringPort depends on the "open" state of "top" part, then a task to satisfy the state is extracted before the task to combine the wiring.

D. Implementation

We implemented the designing and task generation functions (See Section III-A for their definitions) in Java to demonstrate our scheme. The designing function includes Java classes to express the elements in our model. Figure 3 illustrates a simplified pseudo class diagram of our implementation. Each of the component: *primitive* or *composite*, the *part*, the *state*, *property* and *task* has Java classes of their type and instance. The primitive has PrimitiveType class and PrimitiveInstance class, for example. The type and instance classes of primitive and composite extend ComponentType and ComponentInstance classes, respectively. The ComponentType object expresses a model specification, such as "server" or "serverWithNIC". Each instance object has an id, a type object and customized data as an instance of the type object. CompositeType object can include definitions of inner components as ComponentType objects with data of their default instances. A list of *possibleState* is defined on a StateType object. StateInstance object has a StateType object and keeps its *current state*. *Task* templates are defined on TaskType objects. TaskInstance renders them with filling data of its related instance objects.

Each model definition is defined in Java as a ComponentType object in the current implementation. The task generation function generates ComponentInstance objects from ComponentType objects, extracts them into PrimitiveInsance objects, and renders all tasks by calculating the shifting of the states.

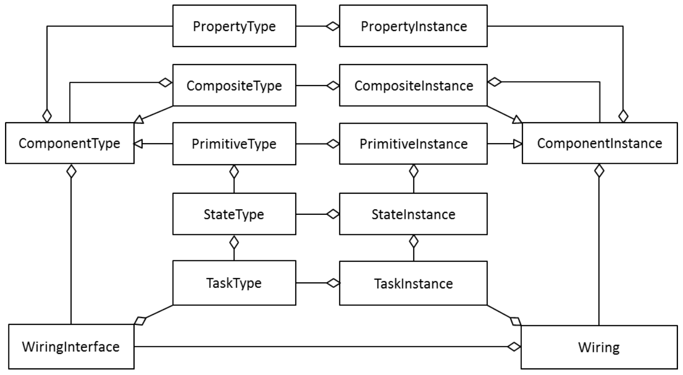

Fig. 3: A pseudo class diagram of our implementation.

IV. Case Study

In this section we show a case study of generating operational procedures utilizing our scheme based on the example of the private cloud platform. We show two cases. The model for Case1 is illustrated in Figure 4.

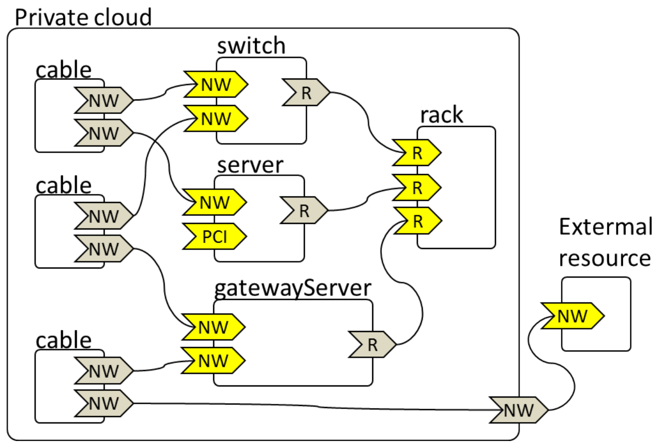

Fig. 4: A model of a private cloud platform in case1.

The "PrivateCloud" composite in Case1 consists of seven primitives: "rack", "switch", "server", "gatewayServer" and three "cables". The definitions of these primitives, other than "gatewayServer", are illustrated in Figure 1 and Table I. The "GatewayServer" is a primitive which has the same definition as "server" primitive but has the second "NetworkPort" accept instead of "PCIExpress" accept. The "switch", "server" and "gatewayServer" are connected with "rack" through "RackSpace" wiring. The "server" and "gatewayServer" are connected to the "switch" by "cable" primitives through "NetworkPort" wiring. The second "NetworkPort" of "gatewayServer" is connected with a "cable" whose another end is promoted to a "NetworkPort" consume of "PrivateCloud". It will be connected to a "NetworkPort" of an "external resource" in the customer's site.

The model for Case2 is almost the same as that of Case1 except it contains "serverWithNIC" composite, shown in Figure 2, instead of "gatewayServer". We assume Case2 supports a requirement of the customer to adopt a particular kind of server for the gateway as well.

Tables II and III show the result for Case1 and Case2, respectively. As seen, every task is generated properly from the templates with filling id of its related component instances. Their order is workable because it satisfies all the dependencies defined in the containing components. In Case2, "nic" is inserted into "server2" before the "server" is inserted into "rack". The task to open the top of "server2" (#12 in Table III) is derived from the dependency which is defined from "PCIExpress" consume to "open" of "top" state in the "server" primitive. Moreover, the task to close the top (#14 in Table III) is derived from the final state definition of the "server" primitive.

TABLE II: The Result of Case1.

#	Task
1	Take "rack" from its package
2	Fix the leg of "rack" on the floor
3	Take "switch" from its package
4	Take "server1" from its package
5	Take "server2" from its package
6	Take "cable1" from its package
7	Take "cable2" from its package
8	Take "cable3" from its package
9	Insert "switch" into "rack"
10	Insert "server1" into "rack"
11	Insert "server2" into "rack"
12	Connect "cable1" with "switch"
13	Connect "cable1" with "server1"
14	Connect "cable2" with "server2"
15	Connect "cable2" with "switch"
16	Connect "cable3" with "resource"
17	Connect "cable3" with "server2"

TABLE III: The Result of Case2.

#	Task
1-5	(Same with 1-5 of case1)
6	**Take "nic" from its package**
7-11	(Same with 6-10 of case1)
12	**Open the top of "server2"**
13	**Insert "nic" into "server2"**
14	**Close the top of "server2"**
15-20	(Same with 11-16 of case1)
21	**Connect "cable3" with "nic"**

Once the model for Case1 is ready, the only effort the integrator has to put in to change the operational procedures from Case1 to Case2 is shifting a type of a component from "gatewayServer" to "serverWithNIC". Everything else is reusable. Obviously, it is easier to plan the modification of these tasks and describe the operational procedures (in natural language) again by hand. The generated tasks including all instructions are defined in its components. Every task is generated only when it is needed in the particular circumstance of the states; no conditional section is included. The workers are not required to refer to the other documents or evaluate the situation.

V. Discussion

This section discusses current limitations in our work and planned efforts to overcome these limitations.

As mentioned in Section III-C, we need to consider the task order in addition to dependencies to generate an efficient task order in a practical operation. Otherwise, tasks in two somewhat unrelated task groups, for example "setting up server with nic" and "setting up rack", can come alternately. The following criteria to select the next target will help to solve this problem: 1) element which is depended from more number of other elements, 2) element which is followed by longer steps of dependencies and 3) element which is closer to previous element selected in terms of relation with wiring connections. We need experimental validation to obtain actual solutions in terms of time taken for these tasks.

Currently we do not have a "progress tracking" capability. We will be able to provide a simple progress management function by letting an integrator and workers share synchronized data of the task list, and letting workers mark tasks as completed through a client terminal every time a task has been completed. By adding information on the estimated time spent on each task, both an integrator and workers will be able to know the total estimated time to complete all tasks.

We present some additional ideas to improve the views of our functions. For integrators, presenting a quotation of the entire cost and estimated time to deploy the system will be of significant importance to design a model that will generate efficient workflows. From the system definition, all products and tasks can be extracted. For example, we assume that for a primitive, when the product is shipped from a vendor, that primitive can have information about the product including its price as well as the time to perform the task. Thus, the total price and total time to deploy it can be extracted from the system definition. For workers, a serving task tree and Gantt chart will be of help to grasp entire tasks and get a better understanding on the order of task executions. They will be constructed from definitions of a hierarchy of components built by composites and dependencies.

In Section IV, we presented case studies that all tasks are generated assuming everything would be conducted as expected. In practical cases, however, states of a system being deployed in the task generation function can be different from its actual states due to various factors such as incorrect modeling or failure of an operation. When any situational difference were to occur, the worker should be able to feedback the actual states and the tasks should be recalculated. To do this, the entire graph, composed of primitive and wiring as nodes, and dependency as edge, should be kept with the status of each element until all tasks are completed. Moreover, a client interface for a worker to feedback actual states and modified task generation algorithm to recalculate tasks based on the modified states will be needed.

We considered only the simplest cases about assignment of workers in this paper, which assumed all the tasks can be conducted by single and that too any worker. However, in many cases, more than one worker will have to cooperate to do the tasks, and a task may need more than one workers or have certain conditions to hold such as the worker knowing the technique, their qualifications or permission on its workers' assignment. We need to modify the task generation algorithm by taking these possibilities and constraints into consideration in order to utilize workers' productivity maximally.

Showing a picture in task description for workers would be good practice, however, preparing the picture for tasks in wirings may be difficult because more than two components should be shown in the picture and the size or shape of the image of each component is unpredictable. We will need to prepare another specification to standardize such images.

In this paper, we defined a primitive as a notion corresponding to a product and its initial states are the same as when it is shipped from a vendor. Here, there can be a request to extract a primitive into finer granularity in order to change its inner parts. For example, changing a "CPU" of a "server" may be required in addition to adding equipment like "nic". This requirement includes two separate problems. The first problem is that a product should be modeled as a composite whose inner primitives are setup and combined preliminarily. To do this, a wiring should be able to have "combined" state as not only its final state but also as its initial state. The second problem is that it needs to extend our specification in order for models to express their transformation from existing components to others. The task generation function also needs to be improved to calculate tasks to take old components away and combine new products. Such a function will be needed to manage a life cycle of a system. Prior work about this topic exists [9] in software provisioning domain.

We also need to improve our component model in terms of convenience to define a system model. The multiplicity support of component and wiringPort, *e.g.* "RackSpace" wiringPort of "rack" in this paper will be needed to efficiently define a large scale system. The inheritance and interface support for a component will also be helpful for abstraction of a component and improving reusability of the composite. We will also need more number of and also more sophisticated models to ensure that our modeling capability is sufficient to capture a range of provisioning requirements.

VI. Related Research

Our component model specification is influenced by existing specifications. We borrowed many notions such as *component*, *composite*, *property*, *wiring*, *wiringInterface* and *promotion* from SCA (Service Component Architecture) [7], [10]. Our two types of *wiringPorts*, *accept* and *consume*, are also inspired by the notions of service and reference in SCA. However, they have distinct semantics; for example the latter implies a kind of dependency from a reference to service while in our case it means simply a physical form of connectors. In our model, something can be needed to be wired with an *accept* of a *primitive* in order for the *primitive* to work correctly, while service in SCA never requires such a connection.

Our work is also greatly influenced by many existing products [2], [1], [11], [3], specifications [4], [12] and research efforts [5], [9] related to model-based software provisioning. The distinction between these works and ours is that these works are primarily in the design of *task* definition. They write a logic to generate small workflow to deploy each component as program code. The logic need not be understandable to the human workers. In contrast, human workers are involved in our case. The workers need to get the overall view of the systems and the *tasks* to perform. Therefore, we defined *tasks* on *stateShift* and *wiringInterface*. In our scheme, the situational judgment to pick up necessary *tasks* is done by the task generation function because it involves tedious work. But we believe the *parts*, their *states* and judgment logic should not be hidden from workers but rather opened up for better understanding of the *tasks*. So, we explicitly define the *states* of *primitives* and simply standardized the logic to pick up *tasks* with *stateShift* concept. To explain the reason why the *tasks* on *wiringInterface* are needed to combine two *components*, we explain why it is not needed in software provisioning. We think so because file system takes the same kind of role and the way to combine things in deployment phase is already standardized. For example when a "WAR file" is deployed to an "application server", we just put it on the file system. In most cases, the only thing needed to know is a path to save this file to.

Another field which is related to this research is Human-Task, which appears in some specifications such as BPMN [13], BPEL4People [14] and WS-HumanTask [15], and many research works [16], [17]. The point of difference of our research from these works is what the task is on. The task which is generated from a workflow engine in their work requires replying something to the workflow engine, such as clicking an approval button or inputing personal information. They never affect physical objects of the world. In our work, the task affects something out of the task generation function.

VII. Conclusions

In this paper an approach to model-based hardware provisioning was presented. To realize the concept, a specification of models and task generation function are shown. Through a simple case study based on provisioning of a private cloud platform, its feasibility and capability to generate a quality operational procedure with an intuitive and simple modeling work are demonstrated. We discuss additional topics to enhance the utility of this scheme including our planned future work to address some of the limitations, such as the need for advanced functionalities and the need to validate our scheme with more case studies and through practical experiments.

References

[1] "Chef," 2013, http://www.opscode.com/chef/.

[2] "Puppet," 2013, http://puppetlabs.com/.

[3] I. Redbooks, *Virtualization With IBM Workload Deployer: Designing and Deploying Virtual Systems*. Vervante, 2011.

[4] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca," *Internet Computing, IEEE*, vol. 16, no. 3, pp. 80–85, 2012.

[5] T. Eilam, M. Elder, A. Konstantinou, and E. Snible, "Pattern-based composite application deployment," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, 2011, pp. 217–224.

[6] E. W. D. Rozier, P. Zhou, and D. Divine, "Building intelligence for software defined data centers: modeling usage patterns," in *Proceedings of the 6th International Systems and Storage Conference*, ser. SYSTOR '13. New York, NY, USA: ACM, 2013, pp. 20:1–20:10. [Online]. Available: http://doi.acm.org/10.1145/2485732.2485752

[7] "Service component architecture (sca), sca assembly model v1.00 specifications," 2007.

[8] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369025

[9] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Pattern-based Runtime Management of Composite Cloud Applications,” in *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*. SciTePress Digital Library, May 2013, Conference Paper.

[10] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*, 1st ed. Addison-Wesley Professional, 2009.

[11] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, “The smartfrog configuration management framework,” *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 16–25, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1496909.1496915

[12] P. Derek and S. Thomas, “Topology and orchestration specification for cloud applications version 1.0,” 2013.

[13] “Business process model and notation (bpmn) version 2.0,” 2011.

[14] C. Luc, K. Dieter, M. Vinkesh, M. Ralf, R. Ravi, R. Michael, and T. Ivana, “Ws-bpel extension for people (bpel4people) specification version 1.1,” 2010.

[15] ——, “Web services - human task (ws-humantask) version 1.1,” 2010.

[16] A. Sasa, M. Juric, and M. Krisper, “Service-oriented framework for human task support and automation,” *Industrial Informatics, IEEE Transactions on*, vol. 4, no. 4, pp. 292–302, 2008.

[17] S. Link, P. Hoyer, T. Schuster, and S. Abeck, “Model-driven development of human tasks for workflows,” in *Software Engineering Advances, 2008. ICSEA ’08. The Third International Conference on*, 2008, pp. 329–335.