Model-based IT Change Management for Large System Definitions with State-related Dependencies

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*—The rapid evolution and changes in the number and variety of information technology (IT) resource requires more efficient IT management schemes. Model-based management of IT resources is a promising approach that offers administrators an intuitive and declarative way to define a change requirement by using a "desired state" model, which alleviates the need to require deep knowledge about the operational logic of the resources. However, prevalent management tools mainly provide provisioning software applications from scratch only, and are not applicable to the change management of already deployed systems, particularly those that consist of hardware components. A change process in already deployed systems requires some components to step through a few temporary states before they reach their desired states, however, most tools seldom assume such temporary states. This paper addresses the above concerns and describes a model-based management scheme for changing IT systems including the hardware resources. Our approach automatically generates the required tasks to apply changes from a desired state model with state-related dependencies between the resources. The contributions of the paper include: (1) a simple base methodology of task planning for system change with our desired state model, (2) a component-based system model to define desired states in large-scale systems, and (3) enhancement of task planning scalability for a number of managed resources. Our approach provides an easier approach for administrators to define their system by offering a library of reusable, pre-defined models. Our approach is described in the context of a case study. We present specifications to define the system changes efficiently and validate the scalability of our approach by measuring the processing time of task planning.**

I. Introduction

Emerging information technologies (IT), such as Cloud Computing, BigData and Internet of Things, require a large number and variety of IT resources, which often change over time. Provisioning these resources in the face of frequent changes and upgrades call for an effective management scheme. There is a trend towards using model-based approaches to IT resource management, particularly for Cloud Computing infrastructures [1], [2], [3], [4]. By using such tools, an administrator of IT resources can define tasks to update the types and quantities of resources in the form of "desired state" model in contrast to a workflow.

The "desired state" model describes the states that the resources should be in as a result of performing the update actions. It offers an intuitive and declarative way to define change requests without requiring deep knowledge about the operational logic of internal resource models [5], [6]. It also offers higher reusability of the definitions and an efficient way to define change requests by combining pre-defined models that may be available as libraries. Expressing the change request as a desired state model instead of a workflow implies that the administrator does not care about the actual process to realize it thereby leaving it up to the tool to adopt a desired process. Therefore, the model is easier to reuse as a component rather than a hard-coded workflow because the tools can generate a workflow from the model components, which is appropriate for the particular combination of the model components.

Despite the prevalence of model-based approaches to address the challenges of change management for IT resources, existing tools (e.g., [1], [2]) incur several limitations. For example, the tools assume that no more than a single operation is needed to get a resource to its desired state [6]. Such an approach is applicable only when deploying applications for the first time when nothing is provisioned prior to that, and hence software resources can just be created or simply overwritten. However, this approach has significant limitations in the change management of an existing system. Updating a system from its current state to the desired state will need several steps to get each individual resource to its desired state even for a software system [7]. We believe this problem will be further amplified when we also consider hardware resources since handling hardware is not as simple as handling software.

To elaborate, consider the case of change management in software IT resources where a middleware package is to be installed and the package depends on a temporary file. In this case, the temporary file would be created first and the package would be installed next. Subsequently, if it is desired that the temporary file be cleaned up, then existing tools cannot express such a requirement in a convenient manner. In this case, the file should have "absent" as its desired state, and the installation of the middleware package should depend on a "present" state of the file. This example demonstrates the challenges in change management in the context of just software resources. Now consider an example that also includes hardware where the change management involves adding extra memory to a server. For the memory chip to be added to the server, first the top (i.e., casing) of the server should be opened, then the chip should be added to the server, and finally the top would be closed. In the change management of IT resources, such temporary and intermediate steps need to be generated correctly.

In our previous work [8], we proposed a model-based task generation scheme for hardware provisioning which can generate all the tasks in the right order to deploy a system. Our prior work demonstrated a model-driven approach that adopted the concept of a "desired state" and generative techniques to automate the process of task generation. However, our prior work did not address the issue of change management, which is a substantially harder problem to resolve. In this paper we leverage and extend our earlier work by presenting a model-based IT change management scheme that can generate tasks that can not only deploy a system from scratch but also make changes to an already deployed system.

Sebastian et al. [9] have shown an IT change planning scheme based on the desired state model for large number of managed resources. They generate change tasks from the model based on state transition system and state-related dependencies. Our approach adopts the same idea where we use a model to express resources as a kind of state transition system, and let the resources be related to each other by associating a dependency on particular state of the resources. The key contributions of our work that distinguishes it from [9] are:

- Our approach presents (1) a new and simpler base methodology of task planning with our desired state model and state-related dependencies, (2) component-based system model to define desired states of large scale systems, and (3) further improvements enhancing scalability of task planning for a number of managed resources.
- We provide an administrator an easy and efficient way to define large scale systems by reusing pre-defined components. The generated change tasks are in the form of partial order tasks.
- We present a simple task planning algorithm which is further enhanced for improved scalability.
- Through a case study of system change, we present the practical specifications to define system change requirements and the process to generate the tasks in detail. Additionally we show the effectiveness to define a large system by our component-model and feasibility of the task planning in terms of its calculation time.

Although we present examples of hardware resource models as a case study in the rest of this paper, our methodology is not limited to management of hardware resources alone.

The rest of this paper is organized as follows: We present the fundamental methodology, model specification for system definition and scalability improvement of our scheme in Section II; We evaluate our approach in the context of a case study and discuss additional topics and future work in Section III; Related research efforts are compared with our approach in Section IV; and finally we offer concluding remarks in Section V.

II. Model-based Change Management

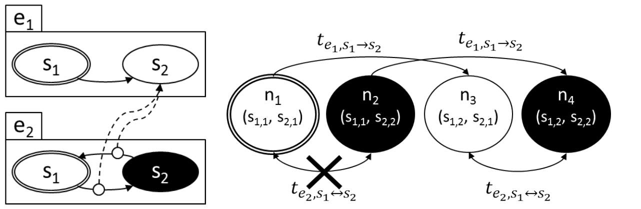
This section delves into the details of the model-based change management approach we have developed. First, we present the base methodology of task planning. Next, we present a component-based model to define large-scale systems by an administrator. Finally, we present some improvements to enhance the methodology of task planning.

A. Formal Description of State Model and Task Generation

The key functionality of our task generation scheme is obtained from a *State Model* and a *Task Generator*. The state model represents an internal expression of a system that is to be managed. The task generator generates tasks that are responsible for changing the states of all the resources into the desired states according to the state model. In the following we provide a formal definition of these concepts and illustrate them with examples. A formal model is important since it helps in analytical reasoning and verification for correctness. This capability is needed in our system since we allow an administrator to model their system including the change management through composition.

In our system model we define a resource as an entity that serves some purpose for the user, e.g., a server. Each resource is assumed to be made up of individual parts or units, which on their own are not useful to user, e.g., the top or casing of a server. However, to represent the entire state of a resource, it is important to understand the state of each of its units, e.g., the top of a server may be open or closed. Thus, the state model of a resource is a combination of the states of its elements. This aspect is captured in our formal model described below.

Let $E = \{e_1, ..., e_n\}$ be a set of *State Elements* of resources which compose the system to be managed. A state element represents a unit in a resource, and is assumed to hold a state. Formally, a state element $e \in E$ is a 5-tuple (id, S, cs, ds, T) where $id(e)$[1] is the unique id of e, $S(e) = \{s_1, ..., s_n\}$ the set of *States* which the state element can be in, $cs(e) \in S(e)$ the *Current State* of e, $ds(e) \in S(e)$ the *Desired State* of e, and $T(e) = \{t_1, ..., t_n\}$ the set of *Transitions* which shift the current state of the element from one state to another in $s \in S(e)$. A transition $t \in T(e)$ is a 3-tuple (s_{src}, s_{dest}, D) where $s_{src} \in S(e)$ is the source state, $s_{dest} \in S(e)$ is the destination state, and $D(t) = \{d_1, ..., d_n\}$ is a set of *Dependencies* needed to be satisfied for executing the transition t. A dependency $d \in D(t)$ is a tuple (t, s) where the dependency implies that a transition t depends on a state s. Note that a transition $t \in T(e)$ may have dependencies on the states of other state elements $\{s \mid s \notin S(e)\}$.


e1
s1
s2
e2
s1
s2
$t_{e_1,s_1\rightarrow s_2}$
$t_{e_1,s_1\rightarrow s_2}$
n1 (s1,1, s2,1)
n2 (s1,1, s2,2)
n3 (s1,2, s2,1)
n4 (s1,2, s2,2)
$t_{e_2,s_1\leftrightarrow s_2}$
$t_{e_2,s_1\leftrightarrow s_2}$

(a) State Model. (b) State Graph.

Fig. 1: State Model and State Graph.

An example of a state model is illustrated in Figure 1a where a state element is shown as rectangle, the state as an oval, the current state as a double-lined oval, the desired state as a filled oval, the transition as a solid-lined arrow, and the dependency as a dashed-lined arrow.

[1]We use parenthesis and the parameter e when describing the concept in the context of a specific element otherwise it is omitted.

The task generator is responsible for generating a shortest length array of transitions $R = (t_1, ..., t_n)$ from the state model which changes all state elements from their initial state to their desired states by satisfying all the dependencies. In the example of Figure 1, there are two state elements $\{e_1, e_2\}$ where each of them has two states $\{s_1, s_2\}$. Both current states, $cs(e_1)$ and $cs(e_2)$, are s_1 and the desired state $ds(e_2)$ is s_2, while $ds(e_1)$ is none. When the desired state is not defined, it means that any state is acceptable as its final state. Here, e_1 has a transition $t = (s_1, s_2, \emptyset)$, and e_2 has mutual transitions between s_1 and s_2 which have dependencies on $s_2 \in S(e_1)$. Let $t_{i,src \to dest}$ denotes a transition from a state s_{src} to s_{dest} in an element e_i. Obviously, the expected result is $R = (t_{1,1\to2}, t_{2,1\to2})$.

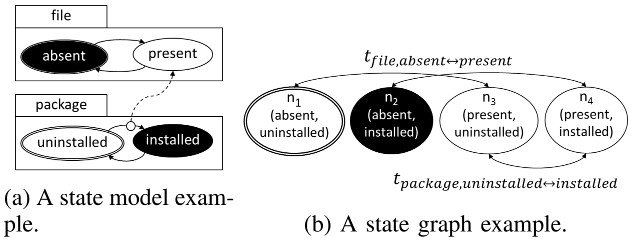
We present the basic idea behind our algorithm to solve this problem below. Our algorithm is categorized as a state-space search in the AI planning research area. Algorithms in this category are known not to be efficient for task planning in terms of time and memory space consumption [10] but are simple and easy to understand. We will present an additional scheme to overcome such drawbacks later in this paper.

As a first step of our algorithm, the task generator creates a *State Graph* from the state model. Though the state model is composed of several state transition systems, the state graph is a single state transition system. A node in a state graph shows the state of all elements in the state model obtained from a parallel composition of the states. The state of the node has components of every state element, and every node has different state combination of the elements. A link between the nodes shows a workable transition in a state element.

Figure 1b illustrates an example of a state graph generated automatically by the task generator from the state model shown in Figure 1a. In this example, since there are two state elements $\{e_1, e_2\}$ and each of them has two states $\{s_1, s_2\}$, the state graph includes four nodes $\{n_1, n_2, n_3, n_4\}$. Each of these nodes has state combinations, such as $(s_{1,1}, s_{2,1})$, $(s_{1,1}, s_{2,2})$, $(s_{1,2}, s_{2,1})$ and $(s_{1,2}, s_{2,2})$ respectively, where $s_{i,j}$ denotes a state $s_j \in S(e_i)$. A transition $t_{e,i\to j}$ makes links between every pair of nodes when the source node has a state $s_{e,i}$ and the destination node has a state $s_{e,j}$, and the other states of these nodes are the same, except when their states do not satisfy the dependencies of the transition. For example, a transition $t_{1,1\to2}$ makes two links $n_1 \to n_3$ and $n_2 \to n_4$. One link always makes one state change. In the same manner, the mutual transitions $t_{2,1\to2}$ and $t_{2,2\to1}$ make links $n_3 \leftrightarrow n_4$, while links $n_1 \leftrightarrow n_2$ are not made because they do not satisfy the dependencies of the transitions on state $s_{1,2}$.

Once the state graph is made, the problem is now to find the shortest path from the initial node, which has the set of initial states, to the desired nodes. Note that there can be more than one desired nodes because a state element can have no desired state. In this example, the initial node is n_1 because it has $s_{1,1}$ and $s_{2,1}$, and the desired nodes are n_2 and n_4 because they have $s_{2,2}$ regardless of the state of e_1 component. One can find the shortest path using any shortest path algorithms, such as Dijkstra's [11]. In our example the result is $(n_1 \to n_3 \to n_4)$. Since these links are derived from $t_{1,1\to2}$ and $t_{2,1\to2}$, it means the same array of transitions with the expected result $R = (t_{1,1\to2}, t_{2,1\to2})$. Since the link $n_1 \to n_3$ is derived from the transition $t_{1,1\to2}$ and the link $n_3 \to n_4$ is derived from the transition $t_{2,1\to2}$, hence $n_1 \to n_3 \to n_4$ implies ($t_{1,1\to2}$ and $t_{2,1\to2}$) as expected.

An example state model and state graph in the case of a temporary file and a middleware package example we described in Section I is shown in Figures 2a and 2b, respectively. The file has "absent" as both its current and desired states. The package has "uninstalled" as its current state and "installed" as its desired state. Also, the transition $t_{package,uninstalled\to installed}$ depends on the state $s_{file,present}$. The shortest path in the generated state graph is $(n_1 \to n_3 \to n_4 \to n_2)$. The shortest path implies the following steps: creating the file, installing the package and deleting the file, in that order.


file
absent
present
package
uninstalled
installed
$t_{file,absent\leftrightarrow present}$
n_1 (absent, uninstalled)
n_2 (absent, installed)
n_3 (present, uninstalled)
n_4 (present, installed)
$t_{package,uninstalled\leftrightarrow installed}$

(a) A state model example.

(b) A state graph example.

Fig. 2: An example of a state model and a state graph.

When each state element corresponds to an actual resource and each transition has a concrete task description, the scheme described above can generate executable tasks. We present a way to define these information and practical examples below.

B. Component Model

In this section, we present our *Component Model* which will be used by an administrator to define their systems to be managed. Its specification embraces a state model to describe the behavior and an architecture to describe the structure that together enhance conceptualization, standardization and reusability of the model. The hierarchical structure of our component model is highly influenced by SCA (Service Component Architecture) [12], [13] but the internal specifications within the component have many differences. Moreover, our prior work [8] described the original version of the component model, however, for this work we have made some key extensions (e.g., wireports that can have states and transitions, and dependencies between parts or wireports) that were required to solve the challenges we aim to resolve in this research. For completeness sake, we present the original component model as well as the extensions we developed.

The component model is structured by *Type*, *Instance* and *Version*. The type includes abstract notions such as the class of the resource, interface to connect resources, and the patterns reflecting valid compositions of these resources. On the other hand, an instance illustrates an actual system or actual resource composing the actual system. An instance is defined by assigning an instance id, a version and current states on a type. A change request is input by an administrator in the form of instance definition of its latest version. Subsequently, a state model is generated from the difference between the existing and new versions of the same instance. The detailed specifications to define type, instance and difference are presented in the rest of this section.

1) Type: Our component model is composed of three first class types: *Primitive*, *Composite* and *Wire Interface*. Primitive is a fundamental unit which corresponds to an actual resource. Composite is a conceptual component which is used to define a pattern of *component* combinations. Component is an abstract notion which contains primitive and composite. Both primitive and composite can be assumed as component when they are used. Wire interface defines an interface to combine two components. A *Task* is a description of a practical action in a transition of a state. It is defined in the wire interface or primitive definition.

a) Primitive: defines a fundamental unit which corresponds to an actual resource. For example "server", "switch" and "rack" and others are shown in Figure 3. A Primitive can have two types of state elements: *Part* and *Wireport*. A Part defines a piece of a primitive which holds a state independently but not be combined with other components. A Wireport defines a port to connect the primitive with other component. There are two types of wireports: *Consume* and *Accept*. A wire[2] (i.e., a connection) is always defined between a consume and accept of different components. Every wireport has a wire interface so that only pairs of consume and accept which have the same wire interface can be connected with each other. While a part can have any number of states and transitions, a wireport always has the states of s_{con} and s_{sep}. A primitive can have any number of parts and wireports. Dependencies can be defined between any pair of parts and wireports in a primitive as long as they do not create cycles.

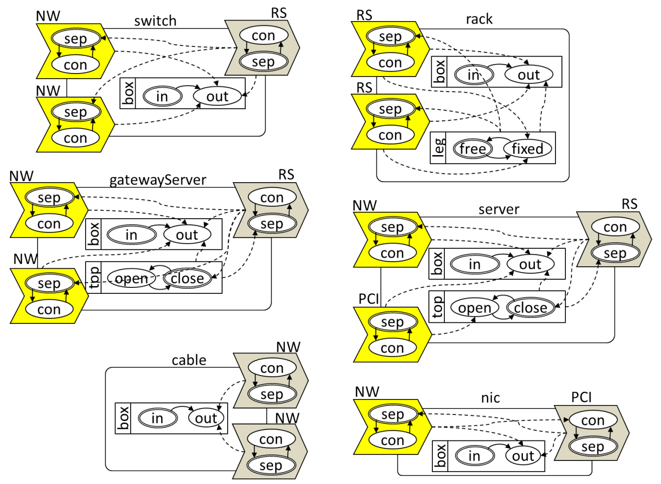


Fig. 3: An example of primitive models.

An example of primitive is shown in Figure 3, where the parts are shown as rectangles, and wireports are shown as the shape of chevron placed on left side and right side of *components*. Consumes are always placed on the right and accepts are on the left. The notations of state models are the same, but note that dependencies cannot be drawn from transitions but from state elements when all transitions in the element have dependencies on the same state.

[2]We use the term wire since it is used by the SCA specification.

The "server" in Figure 3 has two parts: "box" and "top," and three wireports: "NW", "PCI" and "RS". "NW" implies a connection of a network port, "PCI" means a connection of a PCIExpress port, and "RS" implies an insertion point for the server in the rack space. Everything depends on state $s_{box,out}$ because no operation can be performed without opening its package. "PCI" and "RS" depend on different states of "top". "top" is needed to be open when something is connecting with "PCI" or separating from it, but it is needed to be closed when the "server" is being inserted into "RS" or taken out of it. "top" and "RS" have mutual dependencies on their states. "server" has to be separated from "RS" when its top is being opened or closed. Dependencies can be defined between wireports. The dependency of "RS" onto $s_{NW.sep}$ means that the network port should be connected after the "server" is inserted into a rack space, and it should be separated before "server" is taken from the rack space.

b) Composite: defines a pattern of *component* combinations. A composite is composed of *Component Definitions*, *Wires* and *Promotes* in addition to wireports and dependencies. A component definition defines an inner component of the composite with a type and default current states of the component. A wire defines a connection of two components included in the same composite. The Promote assigns a wireport of a composite to a corresponding wireport of its inner component. Dependencies can be defined between any state elements in a composite as long as they do not create cyclic dependencies. Figure 4 depicts an example of a composite, where wires and promotes are shown as as a double line. "serverWithNIC" in Figure 4 shows a set of "server" and "nic" which should be interpreted as a server having two network ports, which may be the case because that server is acting as a gateway. A dependency from "RS" onto the second "NW" wireport is added so that the network port of "nic" will work properly with a rack space of "server".

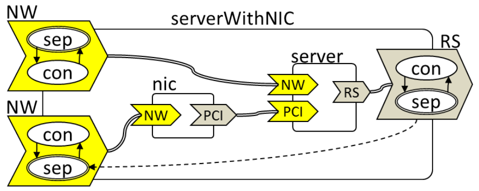


Fig. 4: An example of a composite model.

c) Wire Interface: defines an interface to combine two components. For example, a wire interface "PCIExpress" defines an interface to connect the "NIC" primitive with "Server" primitive. Actually, the connection is defined as a *Wire* which binds two *Wireports* of different components having the same wire interface. Namely, the definition of wire interface is about which pair of wireports can be connected by wires. The wire interface also defines a common class of state element for wires. It has two states s_{con} and s_{sep} which have mutual transitions. Here, s_{con} and s_{sep} means the states of "Connect" and "Separate", respectively.

d) Task: is an action to be performed when changing the current state of an element from one state to another. Every task is connected with one transition and the effect made by the task is limited to realize the change of state.

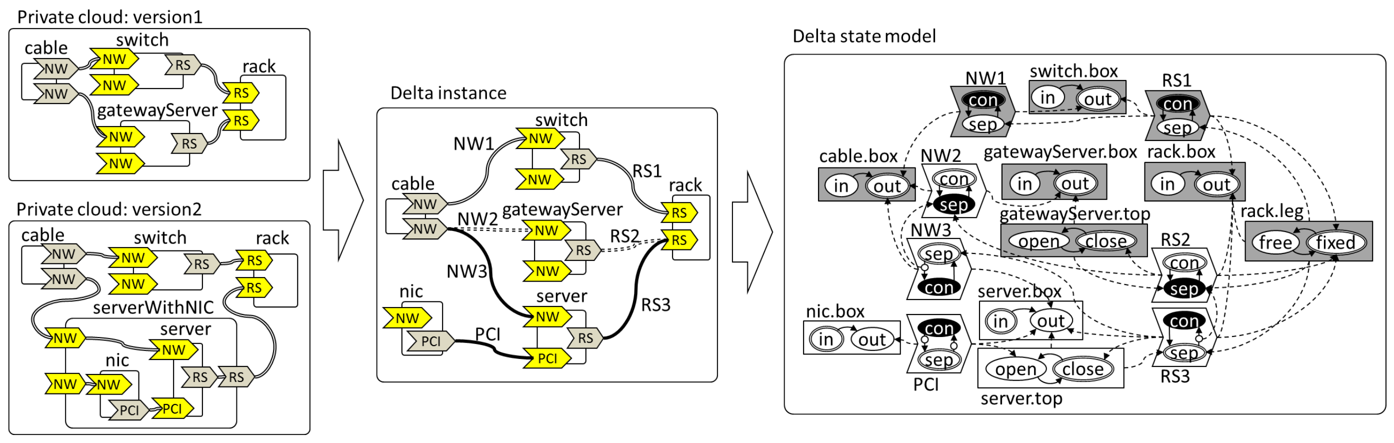


Fig. 5: An example of delta state model generation.

A task comprises templates of descriptions about what should be done. For example, a task template for "PCIExpress" is described in Lines 3-4 of Table I. Line 3 is an instruction to insert a resource such as "nic" into a PCI Express slot of another resource such as "server". Conversely, Line 4 is an instruction to eject it. Blanks in the templates will be filled with information of related instance.

TABLE I: Examples of task template definitions.

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

2) Instantiation of Component Model: An instance definition of a component model comprises instance id, version, type name and current states of all inner state elements. Given these definitions, our *Constructor* of component model generates an instance. All composites are extracted into primitives and wires. If there are wires or dependencies connected with a wireport of a composite, they are transfered into its inner component by tracing their promote definition. All extracted primitives and their state elements and wires should have their own instance id. The id can be generated according to the root instance id and its position in the instance definition. For example, if an instance is identified as "test" and its type is the "serverWithNIC" composite shown in Figure 4, then the "server" primitive in the composite can be identified as "test.server" and the "top" element as "test.server.top". The id is used to identify a specific primitive or element when comparing two different versions of a system instance definition. Thus, it should be defined explicitly if needed.

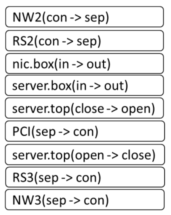
3) Generation of the State Model Difference Due to Change Management: A *Delta State Model* (see Figure 5 for an example) is generated taking the difference of two different versions of a system instance definition. Its current states show the current version of the system definition and desired states show the new version. Change tasks which transfer information from the current state to the new desired state can be generated by the difference (i.e., the delta) between the two state models using the *Task Generator* described in Section II-A. A delta state model is generated by integrating two versions of system instances into one delta instance and picking up state elements from the delta instance. In effect, first the delta instance is generated from the two versions of the system instance definitions and comprises only primitives and wires. Next, the state models are generated from these primitives and wires by importing the state elements in the primitives and wires. The details of the process are described below.

First, all primitives included in any of these versions are imported into the delta instance without duplication. Primitives defined in both versions are marked as "existing", their current states are set as the same as the states in the current version, and desired states are set as same with the states in the new version. Primitives defined only in the current version are marked as "old", their current states are set as same with the states in the current version, but desired states of all their elements are not set (it means "any"). Primitives defined only in the new version are marked as "new", their current states are set as same with the default states defined in their types (which show the initial states when the products correspond to the primitives are shipped), and desired states are set as same with the states in its new version. Note that a wireport can be associated with one wire in an instance definition, but it can have two wires in a delta instance when a wireport is connected with different wireport in the two versions.

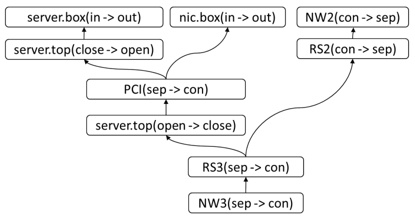
We now explain how the current and desired state of each wire is determined based on its marking. Wires connecting "existing" primitives should have state s_{con} for both of their current states and desired states. Wires connecting "existing"primitive and "new" primitive or connecting "new" primitives should have state s_{sep} for their current states and s_{con} for their desired states. Wires connecting "existing" primitive and "old" primitive should have state s_{con} for their current states and s_{sep} for their desired states. Wires connecting "old" primitives should have s_{con} for their current states, and their desired states should be not set. Additionally, every pair of wires which is attatched to the same wireport are added a dependency such that a transition $s_{sep} \rightarrow s_{con}$ of new wire depends on a state s_{sep} of an old wire, because these two wires can not be connected at the same time actually.

Secondly, all state elements in the delta instance are chosen to get a delta state model. In this step, a wire and two wireports connected by the wire are merged into a state element. Dependencies related with the wireports are transferred to the merged state element. When there are two wires connecting to a wireport, the dependencies related with the wireport are transferred to both of them.

Figure 5 illustrates an example of a delta instance and a delta state model generation. The two figures depict different versions: 1 and 2 of "private cloud" instances, which consists of the components shown in Figures 3 and 4, and are integrated into a delta instance. Version 1 shows the current existing system states stored in our change management function and version 2 shows the definition of the desired states input by an administrator. In version 1, "gatewayServer" is connected with "rack" but it is replaced with "serverWithNIC" composite in version 2. Thus, the "gatewayServer" is connected with old wires ("NW2" and "RS2"), and "server" and "nic" are connected with new wires ("NW3", "RS3" and "PCI") in the delta instance. In the delta state model, all state elements are picked up from primitives and wires. The current and desired states of the old wires are set as s_{con} and s_{sep}, and states of the new wires are set as s_{sep} and s_{con}, while both states of other wires are set as s_{con}. All dependencies related to wireports are transferred to its related wires. For example, the wire "PCI" has dependencies on $s_{nix.box.out}$, $s_{server.box.out}$ and $s_{server.top.open}$. They are transferred from wireports "PCI" of "nic" and "server". Additionally, each transition $t_{sep \rightarrow con}$ of new wires has a dependency on state s_{sep} of the corresponding old wire connecting to the same wireport. For example, $t_{NW3,sep \rightarrow con}$ depends on $s_{NW2,sep}$ and $t_{RS3,sep \rightarrow con}$ depends on $s_{RS2,sep}$. Figure 6a shows the generated task array from this delta state model.



(a) result in total order.



(b) result in partial order.

Fig. 6: Conversion from total order to partial order.

C. Improvement to the task generation scheme

The basic task generation scheme presented in Section II-A has two critical drawbacks for practical use. The first is that the form of generated tasks is provided in a total order as shown in Figure 6a. In practice, often this is not efficient because no task can be conducted concurrently despite no dependencies. Thus, we desire that the generated tasks maintain only a partial order as shown in Figure 6b. The second problem is that the computation of shortest paths becomes an expensive operation when the number of managed resources increases because in our current approach, the number of nodes of a state graph generated from a state model increases exponentially by the number of elements in the state model. The computation cost of Dijkstra's original shortest path algorithm is $O(n^2)$ where n is the number of nodes, which can become significant with large n. We present additional schemes to overcome these problem in the following sections.

1) Conversion of tasks to partial order: The conversion of tasks from total order to partial order is basically conducted by neglecting orders between state elements which have no dependency relationships on each other. This way the transitions included in such state elements can be conducted concurrently. Naturally, when other elements are related between these elements, the transitions should be ordered indirectly by the intermediate transitions. To do this, every task in the totally ordered array is picked from the top to the bottom, and each task is checked if its state element has a dependency on preceding tasks. If the element of the task depends on an element of a preceding task or vise versa, the task is ordered after the preceding task. In the example shown in Figure 6, tasks are picked from $t_{NW2,con \rightarrow sep}$ to $t_{NW3,sep \rightarrow con}$. When the second task $t_{RS2,con \rightarrow sep}$ is checked, state element "RS2" depends on state s_{sep} of "NW2" so that the task is ordered after the preceding task $t_{NW2,con \rightarrow sep}$ as shown in Figure 6b. The sixth task $t_{PCI,sep \rightarrow con}$ can be ordered after three preceding tasks: $t_{nic.box,in \rightarrow out}$, $t_{server.box,in \rightarrow out}$ and $t_{server.top,close \rightarrow open}$, but it is not needed to be ordered after $t_{server.box,in \rightarrow out}$ because $t_{server.top,close \rightarrow open}$ is already ordered after the task. The checking of preceding tasks are started from ones in the tail end, and if an order was assigned with a preceding task, checking of the remaining tasks which are already ordered before the preceding task are skipped.

2) Scalability enhancement for task generation: As described above, the number of nodes of a state graph increases exponentially with the number of elements of a state model. Therefore, reducing the number of state element before creating a state graph is the most effective improvement. Based on this simple inference, we propose the following two schemes: (1) invariant element elimination, and (2) state element separation and merging of each result.

a) Invariant element elimination: State elements which are not changed during a change session will not generate tasks and their dependencies will not influence the other state elements. In effect they are invariant for that change session and hence can be eliminated from a state model before creating a state graph. The state elements which are invariant can be found by the following two criteria: (1) its desired state is not defined or is the same as the current state, and (2) there is no other changing transition depending on its states other than the current state. To decide the transition in the second criterion is movable or not, the state element having the transition has to be decided if it can move or not. Therefore, the algorithm for this elimination scheme forms a depth-first search. If a cycle is detected in the searching stack, all state elements in the stack will be judged as movable. In the delta state model shown in Figure 5, the gray elements are invariant.

b) State element separation and merging: After the invariant elements are eliminated, the remaining elements are separated into groups. As described above, the task graph is created from each group and a shortest path is calculated with the graph. Therefore complicated parts in a state model, such as parts where state elements are connected to each other by dependencies, are supposed to be included in the same group. The problem is deciding what criteria to use for determining the group separation.

In order to reveal the criteria, we observe a state model shown in Figure 5. For example, "server.top" and "RS3" have mutual dependencies but "nic.box" is just connected with "PCI" by an one way dependency. In fact, the "nic.box" can be calculated separately from other elements because the transition of the box element is just triggered by the transition of "PCI" and the existence of the box element does not influence the movement of others. Therefore, after the transitions of the "PCI" are decided, a transition of the box element can be calculated by taking only a part of the transitions of "PCI" which depends on the box element into account. More formally, we take state elements which make cycles of dependencies into the same group. Additionally, we set another type of relationship between the groups such that when there is a dependency between state elements included in different groups, the group having the depending element should be calculated in advance.

Figure 7 illustrates the result of separating the state model shown in Figure 5. Elements are separated into seven groups: $(G1, G2, ..., G7)$. "server.top" and "RS3" are included in the same group because they comprise a cycle of dependencies. In our case study of Figure 5, the order of groups was decided as $(G1, G2, ..., G7)$. $G1$ came first because it was not dependent on elements in other groups. Naturally, the generated task from $G1$ was $R_{G1} = (t_{PCI,sep \to con})$.

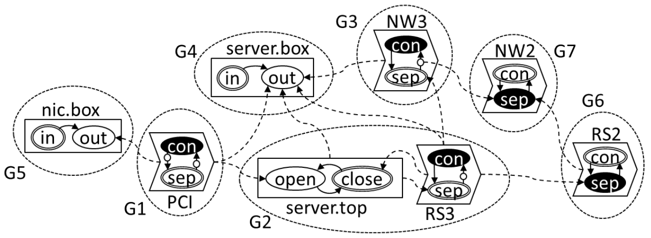


Fig. 7: An example of state model separation.

When generating the tasks of $G2$, the transition $t_{PCI,sep \to con}$ had to be taken into consideration because it depended on a state in $G2$ and would trigger some transitions. To do this, a pseudo element which included only such transitions was created and added to the single state graph upon which the shortest path computation is performed. In this case, an element $e_{pseud} = (pseudo, \{s_1, s_2\}, s_1, s_2, \{t_{pseudo}\})$ such that $t_{pseudo} = (s_1, s_2, s_{server.top,open})$ was created and calculated with elements in $G2$. The generated tasks of $G2$ included the pseudo transitions, and they are simply merged with the original after they are converted into a partially ordered result. When more than one transitions depend on the elements in the group, more pseudo transitions and states would be created. The entire result had a few redundant orders between tasks but no practical difference with the result without separation.

III. Evaluation and Discussion

In the earlier sections we have discussed a case study of a "private cloud" system definition and its result of task generation. By using the models in this definition, we used four different patterns of change requests:

1) from scratch to version 1,
2) from scratch to version 2,
3) from version 1 to version 2, and
4) from version 2 to version 1.

In this section, we confirm the final result of task generation for the third case which we have already shown in Figure 5, and which is chosen because it demonstrates a change. Then we evaluate the effectiveness of system definition using our component models. Finally, we evaluate the performance of our improved task generation algorithm.

A. *Case Study: Change Management from Version 1 to 2*

Table II shows the final view of the generated tasks to change the "private cloud" system from version 1 to version 2 as shown in Figures 5 and 6. Each definition of the transitions is related to a task definition and template description presented in Table I. Blanks in the templates are filled with the values of model instances and rendered. This result is assumed to be presented to human workers who actually deploy the hardware system. The workers can simply carry out tasks starting from the top to bottom in order. When the managed resource is software, instead of a human worker, it will be a kind of software agent and the template can render executable scripts or a workflow. We omit showing the tasks of these other cases but we have validated that these task are also generated properly.

TABLE II: The final view of generated tasks.

#	Task	Depends on
1	Take "server" from its package.	
2	Open the top of "server".	1
3	Take "nic" from its package.	
4	Insert "nic" into "server".	2,3
5	Close the top of "server".	4
6	Separate "cable" from "gatewayServer"	
7	Take "gatewayServer" from "rack"	6
8	Insert "server" into "rack"	7
9	Connect "cable" with "server"	8

B. *Qualitative Evaluation of the Effectiveness of System Definition*

We make a qualitative evaluation of the effectiveness of our modeling approach. Given an existing definition of "private cloud" version 1, for change management what the administrator needs to do to define version 2 is simply picking up the name of "serverWithNIC" from a model repository and replacing "gatawayServer" with it. Naturally, both these resulting versions of "private cloud" definitions can themselves be added to the repository, and reused and combined with other definitions. A remarkable advantage of this model is that the administrator does not need to define dependencies on adjacent components connected with wires because the dependencies are included in the components and transferred when they are instantiated. In the example of Figure 5, "nic" and "server" are simply connected with a wire, but the necessary dependencies are transferred from the primitives. It means that the dependency definitions are also reused along with primitives. Since our component model is based on the SCA specification, we will be able to adopt the features of SCA such as autowires, customization with properties and others, but this discussion is beyond the scope of this paper.

C. Evaluating the Performance of Task Generation Algorithm

We also evaluated the scalability improvement of our task generation algorithm that accounts for the the invariant element elimination scheme and state element separation scheme. Table III describes numbers of all elements, moving elements and groups in addition to their calculation times in milliseconds, of each change request. The calculation times are average of ten times trials. The total element number is larger when the change request is from an existing version, but moving elements are less than those when changing from scratch because these two versions share most of their components.

TABLE III: Element or group numbers and calculation times.

Request	All Elements (ms)		Movings (ms)		Groups, Max (ms)		
$\emptyset \rightarrow ver.1$	12	(266.6)	12	(267.8)	11,	3	(10.4)
$\emptyset \rightarrow ver.2$	10	(82.6)	9	(41.6)	11,	2	(15.8)
$ver.1 \rightarrow ver.2$	16	(5633.6)	7	(20.8)	6,	3	(10.5)
$ver.2 \rightarrow ver.1$	16	(5703.8)	8	(29.8)	7,	3	(4.0)

This result shows that calculation times increase rapidly with the number of elements as expected. In the case of $ver.1 \rightarrow ver.2$, the time is 5,633.6 ms when the number of elements is 16 but the time is reduced to 20.8 ms when the number of elements was reduced to 7 due to the invariant element elimination scheme. These results highlight the significant savings and the effectiveness of the invariant element elimination scheme. The separation scheme also works effectively. In all the cases, elements are separated into many small groups. Most of the groups have one or two elements and maximally have three elements.

The shortest path calculation times also reduced drastically especially when the total number of moving elements is large. In the case of $\emptyset \rightarrow ver.1$, computation time reduced from 267.8 ms to 10.4 ms due to the separation approach. We surmise that the calculation time increases as the number of moving elements in almost a linear fashion as long as the number of elements included in a group is limited.

In fact, the groups which are not ordered by dependencies(e.g., $G2$ and $G5$ in the Figure 7) can be calculated concurrently. Thus, we will be able to improve the time to plan tasks more by proper parallelization of the merging process. Additionally, we can estimate the time approximately by numbers of groups and elements. If the estimated time is unfeasibly large, we will be able to suggest the administrator to adjust the change request or separate it into several steps by showing the problematic parts. These two topics will be part of our future work.

One problem of the current separation scheme is that it can fail task generation even if it would succeed when not using separation. Naturally, more than one shortest paths can be found in a group. If successor groups have more than one dependency relationships with a precedent group, the selection of the shortest path can affect contents of pseudo elements in the successor groups, and can cause an unexpected failure. Thus, when several shortest paths are found, all of them should be tried. It can make the number of branches in the successor processes, however, these branches will be calculated concurrently. So, if we can use concurrent processing environments, such as Hadoop [14], we can still expect the entire processes will be end in a linear time scale. Giving a proof of this hypothesis is also one of our future works.

IV. Related Research

We now compare our work with related efforts. Approaches to IT change management can roughly be divided into two categories based on the form of user input: (1) action-based: where the user input means some kind of actions to change a target system, and (2) state-based: where the user input indicates the desired states that the target system is expected to be in.

In action-based approaches (e.g., [15], [16], [17]), the CHAMPS System [15] represents a seminal work for automation and optimization in change management. It accepts requested operations from a user and determines workable tasks with analyzing dependencies between influenced artifacts. Refinement approaches [18], [16] allow users to input abstract high-level requirements and refine it into low-level executable operations. While they provide a powerful decision support functionality, state based approaches (e.g., [5], [6], [7], [9]) offer users an intuitive way to express change request using the desired states of the individual managed object.

Some efforts [5], [6], [7] on this topic proposed a scheme to generate executable procedures from declarative state models. They maintain a repository of best practice actions primarily defined by IT practitioners apart from resource models. On the other hand, prevalent tools in practical use (e.g., [1], [2]) adopt a model such that the resource and relevant operations are tightly coupled. We believe this is a promising approach for better reusability and validity of the models. When the best practice actions decoupled with resources become large, its maintenance issue will cause another problem. Therefore, to promote reuse of large sets of best practice models, we did not adopt refinement of operations as in [18], [16] but adopt SCA-like component model which can abstract a structural aspect of a system even though refinement is a complementary approach to define desired state of a large system definition in resource-across manner.

Sebastian et al. [9] have conducted research similar to ours, and as explained earlier, our work is influenced by their work. They proposed a planning algorithm with object models based on state transition systems. This work focuses on efficient planning algorithms for a large set of objects in data centers. The algorithms of ours are totally different but they also focus on the dependencies between objects to calculate tasks correctly and efficiently. Due to a lack of access to the details of this related work, we can only surmise that the two approaches may derive from the same principle. Understanding the similarities and performance comparison will also be our future work. We believe that the advantages of our algorithm are simplicity of the base methodology with state model and state graph, and the point that the calculation time of task planning can be roughly estimated by the number and size of the element groups. Moreover, providing an effective component model based on standards to define large system to administrators is an added plus of our approach.

V. Conclusions

In this paper an approach to model-based change management of hardware IT infrastructures was presented. To realize the concept, a fundamental methodology for task generation based on a state model and a state graph, a specification of component models and improved task generation scheme for a large-scale definitions are presented. Through a simple case study of change management of a component in a private cloud platform, its performance and efficiency to generate tasks 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 thorough practical experiments.

Acknowledgment

The first author conducted the research while on a sabbatical leave at Vanderbilt University. Both authors thank NEC for the support.

References

[1] "Chef," 2014, http://www.getchef.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] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A. V. Konstantinou, "Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools," in *Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 404–423.

[6] 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.

[7] 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.

[8] T. Kuroda and A. Gokhale, "Model-based automation for hardware provisioning in it infrastructure," in *To Appear in the Proceedings of Systems Conference (SysCon), 2014 IEEE International*, March 2014.

[9] S. Hagen and A. Kemper, "Model-based planning for state-related changes to infrastructure and software as a service instances in large data centers," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, July 2010, pp. 11–18.

[10] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artif. Intell.*, vol. 90, no. 1-2, pp. 281–300, Feb. 1997.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs." *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

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

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

[14] "hadoop," 2012, http://hadoop.apache.org/.

[15] A. Keller, J. Hellerstein, J. Wolf, K.-L. Wu, and V. Krishnan, "The champs system: change management with planning and scheduling," in *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, vol. 1, April 2004, pp. 395–408 Vol.1.

[16] D. Trastour, R. Fink, and F. Liu, "Changerefinery: Assisted refinement of high-level it change requests," in *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on*, July 2009, pp. 68–75.

[17] S. Hagen and A. Kemper, "A performance and usability comparison of automated planners for it change planning," in *Network and Service Management (CNSM), 2011 7th International Conference on*, Oct 2011, pp. 1–9.

[18] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "Shop2: An htn planning system," *J. Artif. Int. Res.*, vol. 20, no. 1, pp. 379–404, Dec. 2003.